ST.MARYS'S
WOMENS ENGINEERING COLLEGE
BUDAMPADU

DEPARTMENT OF CSE – Data Science

II B.Tech I Semister

(R20)

Name of the Subject:

FUNDAMENTALS OFDATA SCIENCE

Prepared By

BADE ANKAMMARAO M.Tech

UNIT - I: INTRODUCTION

Data science: definition, Datafication, Exploratory Data Analysis, The Data science process, A data scientist role in this process.

NumPy Basics: The NumPy ndarray: A Multidimensional Array Object, Creating ndarrays ,Data Types for ndarrays, Operations between Arrays and Scalars, Basic Indexing and Slicing, Boolean Indexing, Fancy Indexing, Data Processing Using Arrays, Expressing Conditional Logic as Array Operations, Methods for Boolean Arrays , Sorting , Unique.

What is Data Science?

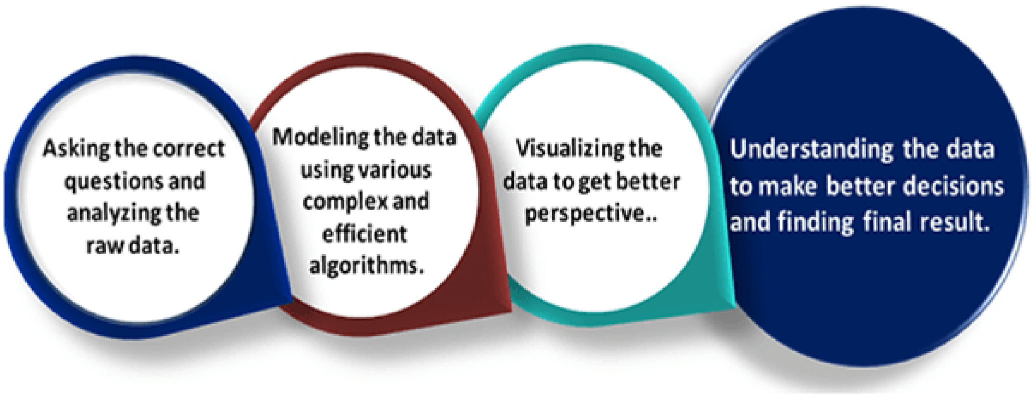
Data science is a deep study of the massive amount of data, which involves extracting meaningful insights from **raw**, **structured, and unstructured data** that is processed using the scientific method, different technologies, and algorithms.

It is a multidisciplinary field that uses tools and techniques to manipulate the data so that you can find something new and meaningful.

Data science uses the most powerful hardware, programming systems, and most efficient algorithms to solve the data related problems. It is the future of artificial intelligence.

In short, we can say that data science is all about:

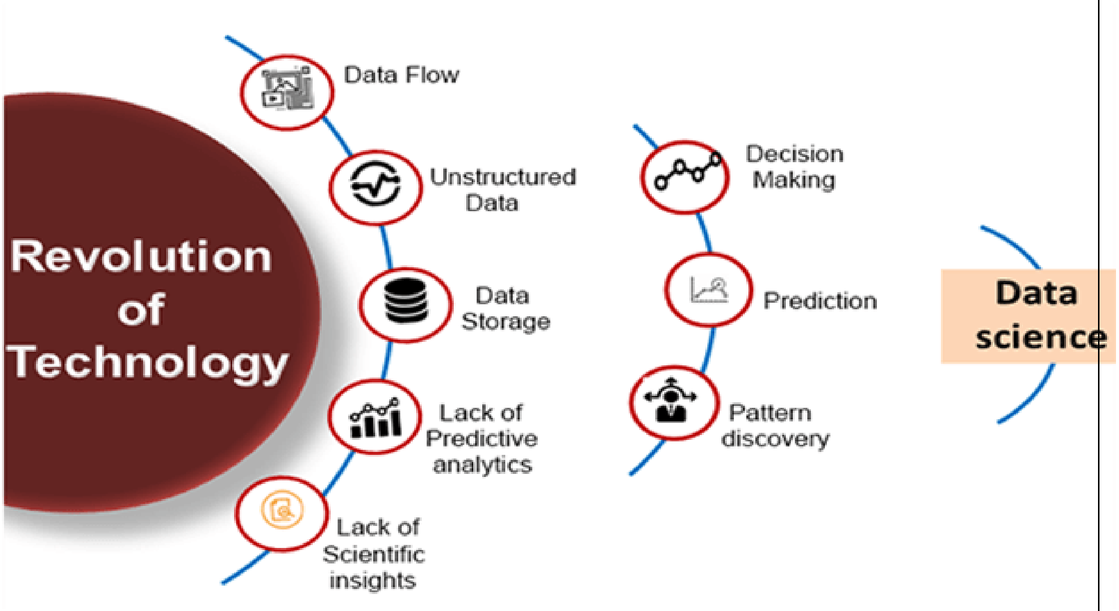
- Asking the correct questions and analyzing the raw data.
- Modeling the data using various complex and efficient algorithms.
- Visualizing the data to get a better perspective.
- Understanding the data to make better decisions and finding the final result.


Asking the correct questions and analyzing the raw data.
Modeling the data using various complex and efficient algorithms.
Visualizing the data to get better perspective..
Understanding the data to make better decisions and finding final result.

Example:

Let suppose we want to travel from station A to station B by car. Now, we need to take some decisions such as which route will be the best route to reach faster at the location, in which route there will be no traffic jam, and which will be cost-effective. All these decision factors will act as input data, and we will get an appropriate answer from these decisions, so this analysis of data is called the data analysis, which is a part of data science.

Need for Data Science:



Some years ago, data was less and mostly available in a structured form, which could be easily stored in excel sheets, and processed using BI tools.

But in today's world, data is becoming so vast, i.e., approximately **2.5 quintals bytes** of data is generating on every day, which led to data explosion. It is estimated as per researches, that by 2020, 1.7 MB of data will be created at every single second, by a single person on earth. Every Company requires data to work, grow, and improve their businesses.

Now, handling of such huge amount of data is a challenging task for every organization. So to handle, process, and analysis of this, we required some complex, powerful, and efficient algorithms and technology, and that technology came into existence as data Science. Following are some main reasons for using data science technology:

- With the help of data science technology, we can convert the massive amount of raw and unstructured data into meaningful insights.
- Data science technology is opting by various companies, whether it is a big brand or a startup. Google, Amazon, Netflix, etc, which handle the huge amount of data, are using data science algorithms for better customer experience.
- Data science is working for automating transportation such as creating a self-driving car, which is the future of transportation.
- Data science can help in different predictions such as various survey, elections, flight ticket confirmation, etc.

Data science Jobs:

As per various surveys, data scientist job is becoming the most demanding Job of the 21st century due to increasing demands for data science. Some people also called it "the **hottest job title of the 21st century**". Data scientists are the experts who can use various statistical tools and machine learning algorithms to understand and analyze the data.

The average salary range for data scientist will be approximately **$95,000 to $ 165,000 per annum**, and as per different researches, about **11.5 millions** of job will be created by the year **2026**.

Types of Data Science Job /Data Scientist Roles

If you learn data science, then you get the opportunity to find the various exciting job roles in this domain. The main job roles are given below:

1. Data Scientist
2. Data Analyst
3. Machine learning expert
4. Data engineer
5. Data Architect
6. Data Administrator
7. Business Analyst
8. Business Intelligence Manager

Below is the explanation of some critical job titles of data science.

1. Data Analyst:

Data analyst is an individual, who performs mining of huge amount of data, models the data, looks for patterns, relationship, trends, and so on. At the end of the day, he comes up with visualization and reporting for analyzing the data for decision making and problem-solving process.

Skill required: For becoming a data analyst, you must get a good background in **mathematics, business intelligence, data mining**, and basic knowledge of **statistics**. You should also be familiar with some computer languages and tools such as **MATLAB, Python, SQL, Hive, Pig, Excel, SAS, R, JS, Spark**, etc.

2. Machine Learning Expert:

The machine learning expert is the one who works with various machine learning algorithms used in data science such as **regression, clustering, classification, decision tree, random forest**, etc.

Skill Required: Computer programming languages such as Python, C++, R, Java, and Hadoop. You should also have an understanding of various algorithms, problem-solving analytical skill, probability, and statistics.

3. Data Engineer:

A data engineer works with massive amount of data and responsible for building and maintaining the data architecture of a data science project. Data engineer also works for the creation of data set processes used in modeling, mining, acquisition, and verification.

Skill required: Data engineer must have depth knowledge of **SQL, MongoDB, Cassandra, HBase, Apache Spark, Hive, MapReduce**, with language knowledge of **Python, C/C++, Java, Perl**, etc.

4. Data Scientist:

A data scientist is a professional who works with an enormous amount of data to come up with compelling business insights through the deployment of various tools, techniques, methodologies, algorithms, etc.

Skill required: To become a data scientist, one should have technical language skills such as **R, SAS, SQL, Python, Hive, Pig, Apache spark, MATLAB**. Data scientists must have an understanding of Statistics, Mathematics, visualization, and communication skills.

Prerequisite for Data Science

Non-Technical Prerequisite:

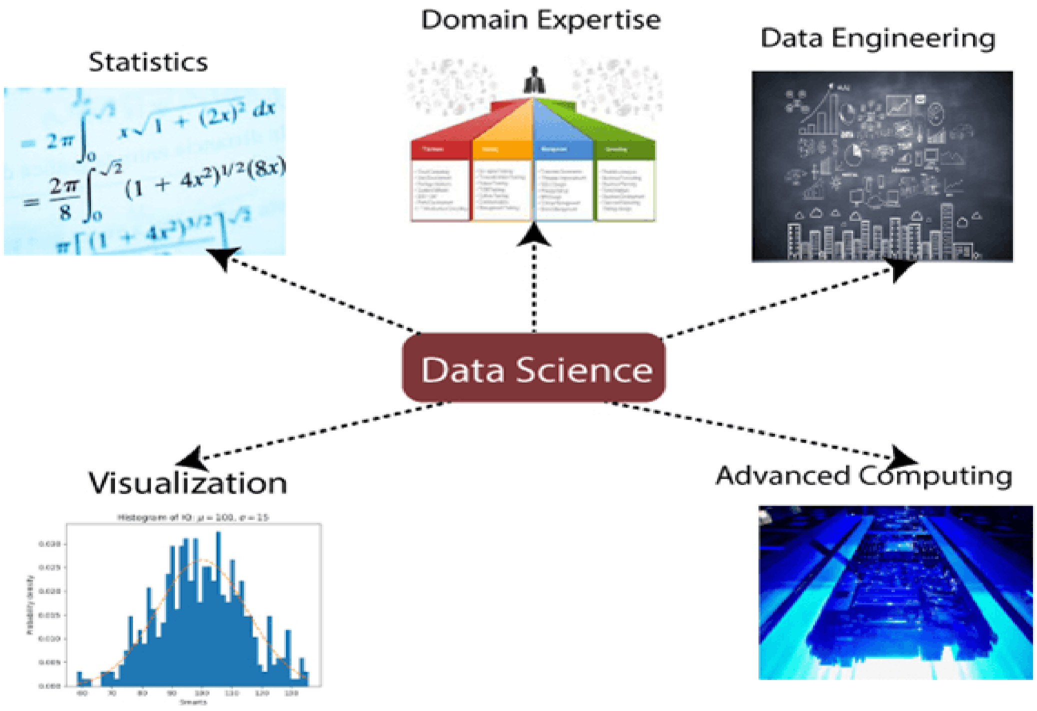
- **Curiosity:** To learn data science, one must have curiosities. When you have curiosity and ask various questions, then you can understand the business problem easily.
- **Critical Thinking:** It is also required for a data scientist so that you can find multiple new ways to solve the problem with efficiency.

- **Communication skills:** Communication skills are most important for a data scientist because after solving a business problem, you need to communicate it with the team.

Technical Prerequisite:

- **Machine learning:** To understand data science, one needs to understand the concept of machine learning. Data science uses machine learning algorithms to solve various problems.
- **Mathematical modeling:** Mathematical modeling is required to make fast mathematical calculations and predictions from the available data.
- **Statistics:** Basic understanding of statistics is required, such as mean, median, or standard deviation. It is needed to extract knowledge and obtain better results from the data.
- **Computer programming:** For data science, knowledge of at least one programming language is required. R, Python, Spark are some required computer programming languages for data science.
- **Databases:** The depth understanding of Databases such as SQL, is essential for data science to get the data and to work with data.

Data Science Components:



The main components of Data Science are given below:

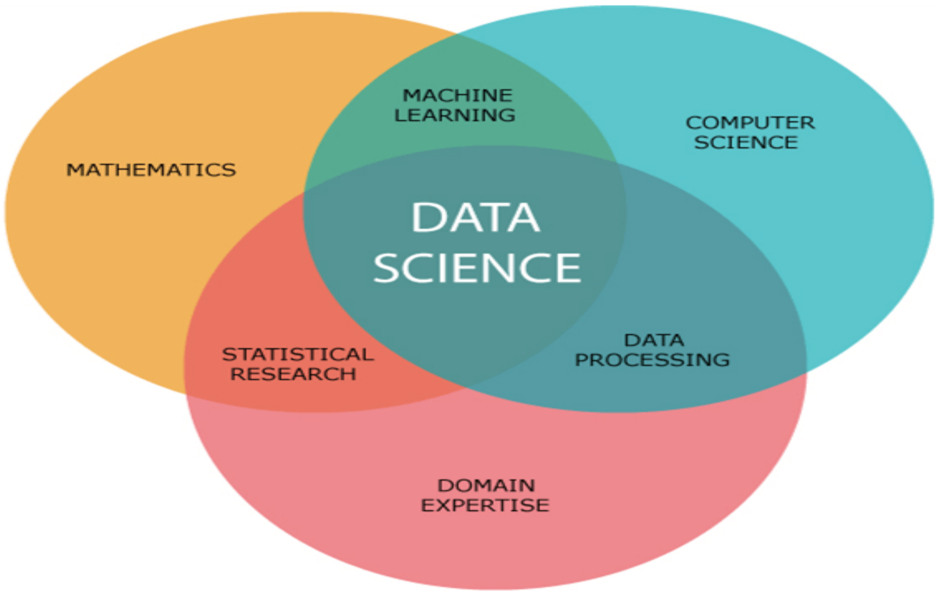
1. Statistics: Statistics is one of the most important components of data science. Statistics is a way to collect and analyze the numerical data in a large amount and finding meaningful insights from it.

2. Domain Expertise: In data science, domain expertise binds data science together. Domain expertise means specialized knowledge or skills of a particular area. In data science, there are various areas for which we need domain experts.

3. Data engineering: Data engineering is a part of data science, which involves acquiring, storing, retrieving, and transforming the data. Data engineering also includes metadata (data about data) to the data.

4. Visualization: Data visualization is meant by representing data in a visual context so that people can easily understand the significance of data. Data visualization makes it easy to access the huge amount of data in visuals.

5. Advanced computing: Heavy lifting of data science is advanced computing. Advanced computing involves designing, writing, debugging, and maintaining the source code of computer programs.



6. Mathematics: Mathematics is the critical part of data science. Mathematics involves the study of quantity, structure, space, and changes. For a data scientist, knowledge of good mathematics is essential.

7. Machine learning: Machine learning is backbone of data science. Machine learning is all about to provide training to a machine so that it can act as a human brain. In data science, we use various machine learning algorithms to solve the problems.

Tools for Data Science

Following are some tools required for data science:

- **Data Analysis tools:** R, Python, Statistics, SAS, Jupyter, R Studio, MATLAB, Excel, RapidMiner.
- **Data Warehousing:** ETL, SQL, Hadoop, Informatica/Talend, AWS Redshift

- **Data Visualization tools:** R, Jupyter, Tableau, Cognos.
- **Machine learning tools:** Spark, Mahout, Azure ML studio.

Machine learning in Data Science

To become a data scientist, one should also be aware of machine learning and its algorithms, as in data science, there are various machine learning algorithms which are broadly being used. Following are the name of some machine learning algorithms used in data science:

- Regression
- Decision tree
- Clustering
- Principal component analysis
- Support vector machines
- Naive Bayes
- Artificial neural network
- Apriori

Applications of Data Science:

- **Image recognition and speech recognition:**
  Data science is currently using for Image and speech recognition. When you upload an image on Facebook and start getting the suggestion to tag to your friends. This automatic tagging suggestion uses image recognition algorithm, which is part of data science.
  When you say something using, "Ok Google, Siri, Cortana", etc., and these devices respond as per voice control, so this is possible with speech recognition algorithm.
- **Gaming world:**
  In the gaming world, the use of Machine learning algorithms is increasing

day by day. EA Sports, Sony, Nintendo, are widely using data science for enhancing user experience.

- **Internet search:**
  When we want to search for something on the internet, then we use different types of search engines such as Google, Yahoo, Bing, Ask, etc. All these search engines use the data science technology to make the search experience better, and you can get a search result with a fraction of seconds.
- **Transport:**
  Transport industries also using data science technology to create self-driving cars. With self-driving cars, it will be easy to reduce the number of road accidents.
- **Healthcare:**
  In the healthcare sector, data science is providing lots of benefits. Data science is being used for tumor detection, drug discovery, medical image analysis, virtual medical bots, etc.
- **Recommendation systems:**
  Most of the companies, such as Amazon, Netflix, Google Play, etc., are using data science technology for making a better user experience with personalized recommendations. Such as, when you search for something on Amazon, and you started getting suggestions for similar products, so this is because of data science technology.
- **Risk detection:**
  Finance industries always had an issue of fraud and risk of losses, but with the help of data science, this can be rescued.
  Most of the finance companies are looking for the data scientist to avoid risk and any type of losses with an increase in customer satisfaction.

Datafication:

In the May/June 2013 issue of *Foreign Affairs*, Kenneth Neil Cukier and Viktor Mayer-Schoenberger wrote an article called "The Rise of Big Data". In it they discuss the concept of datafication, and their example is how we quantify friendships with "likes": it's the way everything we do, online or otherwise, ends up recorded for later examination in someone's data storage units. Or may be multiple storage units, and maybe also for sale.

They define datafication as a process of "taking all aspects of life and turning them into data." As examples, they mention that "Google's augmented-reality glasses datafy the gaze. Twitter datafies stray thoughts. LinkedIn datafies professional networks."

Datafication is an interesting concept and led us to consider its importance with respect to people's intentions about sharing their own data. We are being datafied, or rather our actions are, and when we "like" someone or something online, we are intending to be datafied, or at least we should expect to be. But when we merely browse the Web, we are unintentionally, or at least passively, being datafied through cookies that we might or might not be aware of. And when we walk around in a store, or even on the street, we are being datafied in a completely unintentional way, via sensors, cameras, or Google glasses.

This spectrum of intentionality ranges from us gleefully taking part in a social media experiment we are proud of, to all-out surveillance and stalking. But it's all datafication. Our intentions may run the gamut, but the results don't.

They follow up their definition in the article with a line that speaks volumes about their perspective:

Once we datafy things, we can transform their purpose and turn the information into new forms of value.

Here's an important question that we will come back to throughout the book: who is "we" in that case? What kinds of *value* do they refer to? Mostly, given their examples, the "we" is the modelers and entrepreneurs making money from getting people to buy stuff, and the "value" translates into something like increased efficiency through automation.

What is exploratory data analysis?

Exploratory Data Analysis (EDA) is widely used by Data Scientists while analyzing and investigating Data sets, summarizing the main characteristics of data to the visualizing method. It helps the Data Scientist to discover Data Patterns, Spot anomalies, hypothesis testing, and or assumption.

So in a simple way, it can be defined as a method that helps the Data Scientist determine the best ways to manipulate the given data source to get the answer that is needed as a goal.

How important Exploratory Data Analysis is Data Science

The primary purpose of EDA is to help deep look at the data set before making any assumptions, identifying obvious errors, gain a better understanding of the patterns within the dataset, figure out outliers and/or anomalous events, and last but not least, to find out the exciting relationships among the variables.

Exploratory Data Analysis is extremely important to Data Analysis in the Data Science arena. First, EDA is used to ensure the results the Data scientists are producing are valid and applicable to any desired goals. Second, EDA helps the stakeholders to ensure that they are always asking the right questions. It also helps answer the questions about standard deviations, categorical variables, and confidence intervals. Finally, once EDA is complete and insights are drawn, its features can then be used for more sophisticated data analysis or modeling, including machine learning.

Exploratory data analysis Types

Well there are primarily four types of EDA:

- **Univariate non-graphical:**

Univariate Non Graphical is the most simplest form of data analysis. here it consists of just one variable. Being a single variable, it doesn’t deal with causes or relationships. Instead, the primary purpose of the univariate thematic analysis is to describe the data and find patterns within it.

- **Univariate graphical**

Non-graphical methods cannot provide a complete picture of the data. Graphical methods are therefore required here. The Common types of univariate graphics are:

1. **Stem-and-leaf plots:** These shows all data values and the shape of the distribution.
2. **Histograms a bar plot:** in which each bar represents the frequency (count) or proportion (count/total count) of cases for a range of values.
3. **Box plots:** graphically depict the five-number summary of minimum, first quartile, median, third quartile, and maximum.

- **Multivariate non-graphical**

Multivariate data arises from more than one variable. Generally, Multivariate non-graphical EDA techniques show the relationship between two or more data variables through cross-tabulation or statistics.

- **Multivariate graphical**

Multivariate data uses graphics while displaying relationships between two or more Dataset. The Most used graphic is a grouped bar plot or bar chart with each group representing one level of one of the variables and each bar within a group representing the levels of the other variable.

The Other common types of multivariate graphics include:

Scatter plot: Is used to plot data points on a horizontal and a vertical axis to show how much one variable is affected by another.

- **Multivariate chart:** Is a graphical representation of the relationships between factors and a response.
- **Run chart:** Is a line graph of data plotted over time.
- **Bubble chart:** Is a data visualization that displays multiple circles (bubbles) in a two-dimensional plot.
- **Heat map:** Is a graphical representation of data where values are depicted by color.

Exploratory data analysis Tools

There are many tools available for exploratory data analysis. Some of the most popular ones are R, Python, and SAS. However, each has its strengths and weaknesses, so choosing the right tool for the job is essential.

R is an excellent tool for visualizing data. It has a wide variety of plots and charts that can be used to explore data. It also has a lot of statistical functions that can be used to perform more advanced analyses.

Python is another great tool for EDA. It has many of the same features as R, but it’s also more user-friendly. As a result, Python is an excellent choice for beginners who want to get started with data analysis.

SAS is a powerful statistical software package that can be used for EDA. SAS is more expensive than R and Python, but it’s worth the investment if you need to perform more complex calculations.

QuestionPro and exploratory data analysis

You can always have your data from a different data source, and QuestionPro can definitely help you gather the survey data from multiple channels. But what happens when you want to go beyond the data that’s already been collected? That’s where exploratory data analysis comes in.

QuestionPro’s built-in analysis tools make it easy to get started with EDA. You can quickly see summary statistics for your data, create interactive visualizations, and more. And because QuestionPro integrates with R, you can use all the powerful statistical tools R offers.

So if you’re ready to take your data analysis to the next level, QuestionPro is one of the perfect tools.

What Is the Data Science Process?

The data science process is a systematic approach to solving a data problem. It provides a structured framework for articulating your problem as a question, deciding how to solve it, and then presenting the solution to stakeholders.

NumPy Basics: Arrays and Vectorized Computation:

NumPy, short for Numerical Python, is the fundamental package required for high performance scientific computing and data analysis. It is the foundation on which nearly all of the higher-level tools in this book are built. Here are some of the things it provides:

- ndarray, a fast and space-efficient multidimensional array providing vectorized arithmetic operations and sophisticated *broadcasting* capabilities
- Standard mathematical functions for fast operations on entire arrays of data without having to write loops
- Tools for reading / writing array data to disk and working with memory-mapped files
- Linear algebra, random number generation, and Fourier transform capabilities
- Tools for integrating code written in C, C++, and Fortran

The last bullet point is also one of the most important ones from an ecosystem point of view. Because NumPy provides an easy-to-use C API, it is very easy to pass data to external libraries written in a low-level language and also for external libraries to return data to Python as NumPy arrays. This feature has made Python a language of choice for wrapping legacy C/C++/Fortran codebases and giving them a dynamic and easy-to-use interface.

While NumPy by itself does not provide very much high-level data analytical functionality, having an understanding of NumPy arrays and array-oriented computing will help you use tools like pandas much more effectively.

For most data analysis applications, the main areas of functionality I'll focus on are:

- Fast vectorized array operations for data munging and cleaning, subsetting and filtering, transformation, and any other kinds of computations
- Common array algorithms like sorting, unique, and set operations
- Efficient descriptive statistics and aggregating/summarizing data
- Data alignment and relational data manipulations for merging and joining together heterogeneous data sets
- Expressing conditional logic as array expressions instead of loops with if-elif-elsebranches
- Group-wise data manipulations (aggregation, transformation, function application).

While NumPy provides the computational foundation for these operations, you will likely want to use pandas as your basis for most kinds of data analysis (especially for structured or tabular data) as it provides a rich, high-level interface making most common data tasks very concise and simple. pandas also provides some more domain-specific functionality like time series manipulation, which is not present in NumPy.

The NumPy ndarray: A Multidimensional Array Object

One of the key features of NumPy is its N-dimensional array object, or ndarray, whichis a fast, flexible container for large data sets in Python. Arrays enable you to perform mathematical operations on whole blocks of data using similar syntax to the equivalent operations between scalar elements:

```
In [8]: data
Out[8]:
array([[ 0.9526, -0.246 , -0.8856],[ 0.5639,  0.2379,  0.9104]])
```

```
In [9]: data * 10
Out[9]:
array([[ 9.5256, -2.4601, -8.8565],
       [ 5.6385,  2.3794,  9.104 ]])
```

```
In [10]: data + data
Out[10]:
array([[ 1.9051, -0.492 , -1.7713],
       [ 1.1277,  0.4759,  1.8208]])
```

An ndarray is a generic multidimensional container for homogeneous data; that is, all of the elements must be the same type. Every array has a shape, a tuple indicating the size of each dimension, and a dtype, an object describing the *data type* of the array:

```
In [11]: data.shape
Out[11]: (2, 3)
In [12]: data.dtype
Out[12]:
dtype('float64')
```

Creating ndarrays

The easiest way to create an array is to use the array function. This accepts any sequence-like object (including other arrays) and produces a new NumPy array containing the passed data. For example, a list is a good candidate for conversion:

```
In [13]: data1 = [6, 7.5, 8, 0, 1]

In [14]: arr1 = np.array(data1)In [15]:

arr1
Out[15]: array([ 6. ,  7.5,  8. ,  0. ,  1. ])
```

Nested sequences, like a list of equal-length lists, will be converted into a multidimensional array:

```
In [16]: data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]

In [17]: arr2 = np.array(data2)In [18]:

arr2
Out[18]:
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

```
In [19]: arr2.ndim
Out[19]: 2

In [20]: arr2.shape
Out[20]: (2, 4)
```

Unless explicitly specified (more on this later), np.array tries to infer a good data type for the array that it creates. The data type is stored in a special dtype object; for example, in the above two examples we have:

```
In [21]: arr1.dtype Out[21]:
dtype('float64')

In [22]: arr2.dtype Out[22]:
dtype('int64')
```

In addition to np.array, there are a number of other functions for creating new arrays. As examples, zeros and ones create arrays of 0's or 1's, respectively, with a given length or shape. empty creates an array without initializing its values to any particular value. To create a higher dimensional array with these methods, pass a tuple for the shape:

```
In [23]: np.zeros(10)
Out[23]: array([ 0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.])

In [24]: np.zeros((3, 6))
Out[24]:
            array([[ 0.,   0.,   0.,   0.,   0.,   0.],
                   [ 0.,   0.,   0.,   0.,   0.,   0.],
                   [ 0.,   0.,   0.,   0.,   0.,   0.]])

In    [25]:    np.empty((2,    3,    2))
Out[25]:
array([[[     4.94065646e-324,           4.94065646e-324],    [
        3.87491056e-297,             2.46845796e-130],    [
        4.94065646e-324,     4.94065646e-324]],

       [[   1.90723115e+083,         5.73293533e-053],[ -
        2.33568637e+124,   -6.70608105e-012],
        [   4.42786966e+160,         1.27100354e+025]]])
```

It's not safe to assume that np.empty will return an array of all zeros. In many cases, as previously shown, it will return uninitialized garbage values.

arange is an array-valued version of the built-in Python range function:

```
In [26]: np.arange(15)
Out[26]: array([ 0,   1,   2,   3,   4,   5,   6,   7,   8,   9, 10, 11, 12, 13, 14])
```

See Table 4-1 for a short list of standard array creation functions. Since NumPy is focused on numerical computing, the data type, if not specified, will in many cases be float64(floating point).

Table 4-1. Array creation functions

Function	Description
array	Convert input data (list, tuple, array, or other sequence type) to an ndarray either byinferring a dtype or explicitly specifying a dtype. Copies the input data

	by default.
asarray	Convert input to ndarray, but do not copy if the input is already an ndarray
arange	Like the built-in range but returns an ndarray instead of a list.
ones, ones_like	Produce an array of all 1's with the given shape and dtype. ones_like takes another array and produces a ones array of the same shape and dtype.
zeros, zeros_like	Like ones and ones_like but producing arrays of 0's instead

Function	Description
empty, empty_like like ones and zeros	Create new arrays by allocating new memory, but do not populate with any values
eye, identity	Create a square N x N identity matrix (1's on the diagonal and 0's elsewhere)

Data Types for ndarrays

The *data type* or dtypeis a special object containing the information the ndarray needs to interpret a chunk of memory as a particular type of data:

```
In [27]: arr1 = np.array([1, 2, 3], dtype=np.float64)

In [28]: arr2 = np.array([1, 2, 3], dtype=np.int32)

In [29]: arr1.dtype                    In [30]: arr2.dtype Out[29]:
dtype('float64')                       Out[30]: dtype('int32')
```

Dtypes are part of what make NumPy so powerful and flexible. In most cases they map directly onto an underlying machine representation, which makes it easy to read and write binary streams of data to disk and also to connect to code written in a low-level language like C or Fortran. The numerical dtypes are named the same way: a type name, like float or int, followed by a number indicating the number of bits per element. A standard double-precision floating point value (what's used under the hood in Python's float object) takes up 8 bytes or 64 bits. Thus, this type is known in NumPy as float64. See Table 4-2 for a full listing of NumPy's supported data types.

Table 4-2. NumPy data types

Type	Type Code	Description
int8, uint8	i1, u1	Signed and unsigned 8-bit (1 byte) integer types
int16, uint16	i2, u2	Signed and unsigned 16-bit integer types
int32, uint32	i4, u4	Signed and unsigned 32-bit integer types
int64, uint64	i8, u8	Signed and unsigned 32-bit integer types
float16	f2	Half-precision floating point
float32	f4 or f	Standard single-precision floating point. Compatible with C float
float64, float128	f8 or d	Standard double-precision floating point. Compatible with C double and Python floatobject

float128	f16 or g	Extended-precision floating point
complex64,	c8,	Complex numbers represented by two 32,64,or 128 floats,respectively
complex128,	c16	
complex256	c32	

Type	Type Code	Description
bool	?	Boolean type storing True and False values
object	O	Python object type
string_	S	Fixed-length string type (1 byte per character). For example, to createa string dtype with length 10, use 'S10'.
unicode_ specific). Same	U	Fixed-length unicode type (number of bytes platform specification semantics as string_ (e.g. 'U10').

You can explicitly convert or *cast* an array from one dtype to another using ndarray's astypemethod:

```
In [31]: arr = np.array([1, 2, 3, 4, 5])

In [32]: arr.dtype Out[32]:
dtype('int64')

In [33]: float_arr = arr.astype(np.float64)In [34]:

float_arr.dtype
Out[34]: dtype('float64')
```

In this example, integers were cast to floating point. If I cast some floating point numbers to be of integer dtype, the decimal part will be truncated:

```
In [35]: arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])

In [36]: arr
Out[36]: array([   3.7,   -1.2,   -2.6,             0.5,   12.9,   10.1])

In [37]: arr.astype(np.int32)
Out[37]: array([ 3, -1, -2,   0, 12, 10], dtype=int32)
```

Should you have an array of strings representing numbers, you can use astypeto convert them to numeric form:

```
In [38]: numeric_strings = np.array(['1.25', '-9.6', '42'], dtype=np.string_)

In [39]: numeric_strings.astype(float)
Out[39]: array([   1.25,   -9.6 ,   42.   ])
```

If casting were to fail for some reason (like a string that cannot be converted to float64), a TypeError will be raised. See that I was a bit lazy and wrote float instead of np.float64; NumPy is smart enough to alias the Python types to the equivalent dtypes.

You can also use another array's dtype attribute:

```
In [40]: int_array = np.arange(10)
In [41]: calibers = np.array([.22, .270, .357, .380, .44, .50], dtype=np.float64)

In [42]: int_array.astype(calibers.dtype)
Out[42]: array([ 0.,   1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.])
```

There are shorthand type code strings you can also use to refer to a dtype:

```
In [43]: empty_uint32 = np.empty(8, dtype='u4')

In [44]: empty_uint32
Out[44]:
array([          0,          0, 65904672,          0, 64856792,          0,
        39438163,          0], dtype=uint32)
```

Calling astype *always* creates a new array (a copy of the data), even if the new dtype is the same as the old dtype.

It's worth keeping in mind that floating point numbers, such as thosein float64 and float32 arrays, are only capable of approximating frac- tional quantities. In complex computations, you may accrue some *floating point error*, making comparisons only valid up to a certain num-ber of decimal places.

Operations between Arrays and Scalars

Arrays are important because they enable you to express batch operations on data without writing any for loops. This is usually called *vectorization*. Any arithmetic operations between equal-size arrays applies the operation elementwise:

```
In [45]: arr = np.array([[1., 2., 3.], [4., 5., 6.]])

In [46]: arr
Out[46]:
        array([[ 1.,  2.,  3.],
               [ 4.,  5.,  6.]])

In [47]: arr * arr                  In [48]: arr - arr
Out[47]:                            Out[48]:
array([[  1.,    4.,    9.],        array([[ 0.,  0.,  0.],
       [ 16.,   25.,   36.]])              [ 0.,  0.,  0.]])
```

Arithmetic operations with scalars are as you would expect, propagating the value to each element:

```
In [49]: 1 / arr                           In [50]: arr ** 0.5
Out[49]:                                   Out[50]:
array([[ 1.    ,  0.5   ,  0.3333],        array([[ 1.    ,  1.4142,  1.7321],
       [ 0.25  ,  0.2   ,  0.1667]])              [ 2.    ,  2.2361,  2.4495]])
```

Operations between differently sized arrays is called *broadcasting* and will be discussed in more detail in Chapter 12. Having a deep understanding of broadcasting is not necessary for most of this book.

Basic Indexing and Slicing

NumPy array indexing is a rich topic, as there are many ways you may want to select a subset of your data or individual elements. One-dimensional arrays are simple; on the surface they act similarly to Python lists:

```
In [51]: arr = np.arange(10)

In [52]: arr
Out[52]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [53]: arr[5]
Out[53]: 5

In [54]: arr[5:8]
Out[54]: array([5, 6, 7])

In [55]: arr[5:8] = 12

In [56]: arr
Out[56]: array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
```

As you can see, if you assign a scalar value to a slice, as in arr[5:8] = 12, the value is propagated (or *broadcasted* henceforth) to the entire selection. An important first distinction from lists is that array slices are *views* on the original array. This means that the data is not copied, and any modifications to the view will be reflected in the source array:

```
In [57]: arr_slice = arr[5:8]In [58]:

arr_slice[1] = 12345

In [59]: arr
Out[59]: array([        0,        1,        2,        3,        4,      12, 12345,        12,        8,        9])

In [60]: arr_slice[:] = 64

In [61]: arr
Out[61]: array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])
```

If you are new to NumPy, you might be surprised by this, especially if they have used other array programming languages which copy data more zealously. As NumPy has been designed with large data use cases in mind, you could imagine performance and memory problems if NumPy insisted on copying data left and right.

If you want a copy of a slice of an ndarray instead of a view, you will need to explicitly copy the array; for example arr[5:8].copy().

With higher dimensional arrays, you have many more options. In a two-dimensional array, the elements at each index are no longer scalars but rather one-dimensional arrays:

```
In [62]: arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

In [63]: arr2d[2]
Out[63]: array([7, 8, 9])
```

Thus, individual elements can be accessed recursively. But that is a bit too much work, so you can pass a comma-separated list of indices to select individual elements. So these are equivalent:

```
In [64]: arr2d[0][2]
Out[64]: 3

In [65]: arr2d[0, 2]
Out[65]: 3
```

See Figure 4-1 for an illustration of indexing on a 2D array.

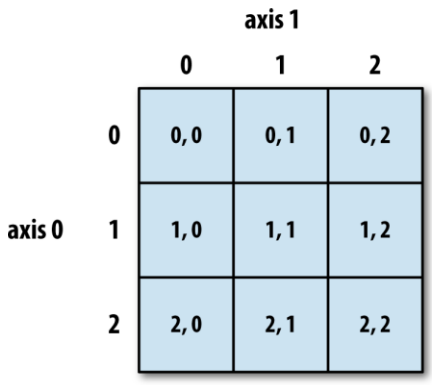


Figure 4-1. Indexing elements in a NumPy array

In multidimensional arrays, if you omit later indices, the returned object will be a lower-dimensional ndarray consisting of all the data along the higher dimensions. So in the 2 × 2 × 3 array arr3d

```
In [66]: arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

In [67]: arr3d
Out[67]:
array([[[ 1,   2,   3],

             [ 4,   5,   6]],
            [[ 7,   8,   9],
             [10, 11, 12]]])
```

arr3d[0]is a 2 × 3 array:

In [68]: arr3d[0]bool	?	Boolean type storing True and False values
object	O	Python object type
string_	S	Fixed-length string type (1 byte per character). For example, to createa string dtype with length 10, use 'S10'.
unicode_	U	Fixed-length unicode type (number of bytes platform specific). Same specification semantics as string_(e.g. 'U10').

```
Out[68]:
        array([[1, 2, 3],
                   [4, 5, 6]])
```

Both scalar values and arrays can be assigned to arr3d[0]:

```
In [69]: old_values = arr3d[0].copy()In [70]:

arr3d[0] = 42

In [71]: arr3d
```

```
Out[71]:
        array([[[42, 42, 42],
                  [42, 42, 42]],
                 [[ 7,   8,   9],
                  [10, 11, 12]]])

In [72]: arr3d[0] = old_valuesIn [73]:

arr3d
Out[73]:
array([[[ 1,   2,   3],
             [ 4,   5,   6]],
           [[ 7,   8,   9],
             [10, 11, 12]]])
```

Similarly, arr3d[1, 0] gives you all of the values whose indices start with (1, 0), form- ing a 1-dimensional array:

```
In [74]: arr3d[1, 0]
Out[74]: array([7, 8, 9])
```

Note that in all of these cases where subsections of the array have been selected, the returned arrays are views.

Indexing with slices

Like one-dimensional objects such as Python lists, ndarrays can be sliced using the familiar syntax:

```
In [75]: arr[1:6]
Out[75]: array([ 1,   2,   3,   4, 64])
```

Higher dimensional objects give you more options as you can slice one or more axes and also mix integers. Consider the 2D array above, arr2d. Slicing this array is a bit different:

```
In [76]: arr2d                          In [77]: arr2d[:2]
Out[76]:                                Out[77]:

       array([[1, 2, 3],                       array([[1, 2, 3],
                 [4, 5, 6],                             [4, 5, 6]])
           [7, 8, 9]])
```

As you can see, it has sliced along axis 0, the first axis. A slice, therefore, selects a range of elements along an axis. You can pass multiple slices just like you can pass multiple indexes:

```
In [78]: arr2d[:2, 1:]
Out[78]:
     array([[2, 3],
               [5, 6]])
```

When slicing like this, you always obtain array views of the same number of dimensions. By mixing integer indexes and slices, you get lower dimensional slices:

```
In [79]: arr2d[1, :2]                    In [80]: arr2d[2, :1]
Out[79]: array([4, 5])                   Out[80]: array([7])
```

See Figure 4-2 for an illustration. Note that a colon by itself means to take the entire axis, so you can slice only higher dimensional axes by doing:

```
In [81]: arr2d[:, :1]
Out[81]:
    array([[1],
            [4],
            [7]])
```

Of course, assigning to a slice expression assigns to the whole selection:

```
In [82]: arr2d[:2, 1:] = 0
```

Boolean Indexing

Let’s consider an example where we have some data in an array and an array of names with duplicates. I’m going to use here the randn function in numpy.random to generate some random normally distributed data:

```
In [83]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])In [84]:

data = randn(7, 4)

In [85]:
names
Out[85]:
array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'],
        dtype='|S4')

In [86]: data
Out[86]:
 array([[-0.048 ,  0.5433, -0.2349,  1.2792],
        [-0.268 ,  0.5465,  0.0939, -2.0445],
        [-0.047 , -2.026 ,  0.7719,  0.3103],
        [ 2.1452,  0.8799, -0.0523,  0.0672],
        [-1.0023, -0.1698,  1.1503,  1.7289],
        [ 0.1913,  0.4544,  0.4519,  0.5535],
        [ 0.5994,  0.8174, -0.9297, -1.2564]])
```

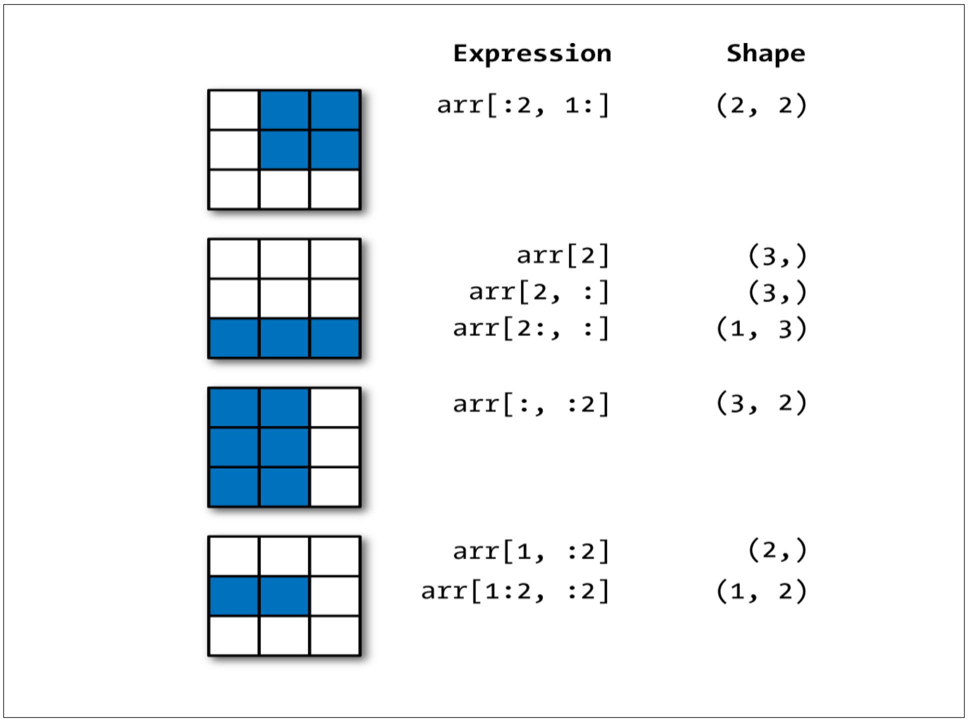


Figure 4-2. Two-dimensional array slicing

Suppose each name corresponds to a row in the data array. If we wanted to select all the rows with corresponding name 'Bob'. Like arithmetic operations, comparisons (such as ==) with arrays are also vectorized. Thus, comparing names with the string 'Bob'yields a boolean array:

```
In [87]: names == 'Bob'
Out[87]: array([ True, False, False, True, False, False, False], dtype=bool)
```

This boolean array can be passed when indexing the array:

```
In [88]: data[names == 'Bob']
Out[88]:
            array([[-0.048 ,   0.5433, -0.2349,   1.2792],
                   [ 2.1452,   0.8799, -0.0523,   0.0672]])
```

The boolean array must be of the same length as the axis it's indexing. You can even mix and match boolean arrays with slices or integers (or sequences of integers, more on this later):

```
In [89]: data[names == 'Bob', 2:]Out[89]:
array([[-0.2349,   1.2792],
            [-0.0523,   0.0672]])

In [90]: data[names == 'Bob', 3]
Out[90]: array([ 1.2792,   0.0672])
```

To select everything but 'Bob', you can either use !=or negate the condition using -:

```
In [91]: names != 'Bob'
Out[91]: array([False, True, True, False, True, True, True], dtype=bool)

In [92]: data[-(names == 'Bob')]
Out[92]:
        array([[-0.268 ,   0.5465,   0.0939, -2.0445],
               [-0.047 , -2.026 ,   0.7719,   0.3103],
               [-1.0023, -0.1698,   1.1503,   1.7289],
               [ 0.1913,   0.4544,   0.4519,   0.5535],
               [ 0.5994,   0.8174, -0.9297, -1.2564]])
```

Selecting two of the three names to combine multiple boolean conditions, use boolean arithmetic operators like &(and) and |(or):

```
In [93]: mask = (names == 'Bob') | (names == 'Will')

In [94]: mask
Out[94]: array([True, False, True, True, True, False, False], dtype=bool)

In [95]: data[mask]
Out[95]:
        array([[-0.048 ,   0.5433, -0.2349,   1.2792],
               [-0.047 , -2.026 ,   0.7719,   0.3103],
               [ 2.1452,   0.8799, -0.0523,   0.0672],
               [-1.0023, -0.1698,   1.1503,   1.7289]])
```

Selecting data from an array by boolean indexing *always* creates a copy of the data, even if the returned array is unchanged.

The Python keywords and and or do not work with boolean arrays.

Setting values with boolean arrays works in a common-sense way. To set all of the negative values in data to 0 we need only do:

```
In [96]: data[data < 0] = 0

In [97]: data
Out[97]:
    array([[ 0.     ,   0.5433,   0.     ,   1.2792],
           [ 0.     ,   0.5465,   0.0939,   0.     ],
           [ 0.     ,   0.     ,   0.7719,   0.3103],
           [ 2.1452,   0.8799,   0.     ,   0.0672],
           [ 0.     ,   0.     ,   1.1503,   1.7289],
           [ 0.1913,   0.4544,   0.4519,   0.5535],
           [ 0.5994,   0.8174,   0.     ,   0.     ]])
```

Setting whole rows or columns using a 1D boolean array is also easy:

```
In [98]: data[names != 'Joe'] = 7

In [99]: data
Out[99]:
 array([[ 7.      ,  7.      ,  7.      ,  7.      ],
        [ 0.      ,  0.5465,  0.0939,  0.        ],
        [ 7.      ,  7.      ,  7.      ,  7.      ],
        [ 7.      ,  7.      ,  7.      ,  7.      ],
        [ 7.      ,  7.      ,  7.      ,  7.      ],
        [ 0.1913,  0.4544,  0.4519,  0.5535],
        [ 0.5994,  0.8174,  0.       ,  0.      ]])
```

Fancy Indexing

Fancy indexing is a term adopted by NumPy to describe indexing using integer arrays. Suppose we had a 8 × 4 array:

```
In [100]: arr = np.empty((8, 4))

    In [101]: for i in range(8):
     .....:         arr[i] = i

In [102]: arr
Out[102]:
        array([[ 0.,  0.,  0.,  0.],
               [ 1.,  1.,  1.,  1.],
               [ 2.,  2.,  2.,  2.],
               [ 3.,  3.,  3.,  3.],
               [ 4.,  4.,  4.,  4.],
               [ 5.,  5.,  5.,  5.],
               [ 6.,  6.,  6.,  6.],
               [ 7.,  7.,  7.,  7.]])
```

To select out a subset of the rows in a particular order, you can simply pass a list or ndarray of integers specifying the desired order:

```
In [103]: arr[[4, 3, 0, 6]]
Out[103]:
        array([[ 4.,  4.,  4.,  4.],
               [ 3.,  3.,  3.,  3.],
               [ 0.,  0.,  0.,  0.],
               [ 6.,  6.,  6.,  6.]])
```

Hopefully this code did what you expected! Using negative indices select rows from the end:

```
In [104]: arr[[-3, -5, -7]]
Out[104]:
        array([[ 5.,  5.,  5.,  5.],
               [ 3.,  3.,  3.,  3.],
               [ 1.,  1.,  1.,  1.]])
```

Passing multiple index arrays does something slightly different; it selects a 1D array of elements corresponding to each tuple of indices:

```
# more on reshape in Chapter 12
In [105]: arr = np.arange(32).reshape((8, 4))

In [106]: arr
Out[106]:
        array([[ 0,  1,  2,  3],
               [ 4,  5,  6,  7],
               [ 8,  9, 10, 11],
               [12, 13, 14, 15],
               [16, 17, 18, 19],
               [20, 21, 22, 23],
               [24, 25, 26, 27],
               [28, 29, 30, 31]])

In [107]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]
Out[107]: array([ 4, 23, 29, 10])
```

Take a moment to understand what just happened: the elements (1, 0), (5, 3), (7,1), and (2, 2) were selected. The behavior of fancy indexing in this case is a bit different from what some users might have expected (myself included), which is the rectangular region formed by selecting a subset of the matrix's rows and columns. Here is one way to get that:

```
In [108]: arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]Out[108]:
        array([[ 4,  7,  5,  6],
               [20, 23, 21, 22],
               [28, 31, 29, 30],
               [ 8, 11,  9, 10]])
```

Another way is to use the np.ix_function, which converts two 1D integer arrays to an indexer that selects the square region:

```
In [109]: arr[np.ix_([1, 5, 7, 2], [0, 3, 1, 2])]Out[109]:
        array([[ 4,  7,  5,  6],
               [20, 23, 21, 22],
               [28, 31, 29, 30],
               [ 8, 11,  9, 10]])
```

Keep in mind that fancy indexing, unlike slicing, always copies the data into a new array.

Transposing Arrays and Swapping Axes

Transposing is a special form of reshaping which similarly returns a view on the underlying data without copying anything. Arrays have the transpose method and also the special Tattribute:

```
In [110]: arr = np.arange(15).reshape((3, 5))

In [111]: arr                                    In [112]: arr.T
```

```
Out[111]:
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

```
Out[112]:
array([[ 0,  5, 10],
       [ 1,  6, 11],
       [ 2,  7, 12],
       [ 3,  8, 13],
       [ 4,  9, 14]])
```

When doing matrix computations, you will do this very often, like for example computing the inner matrix product X^TX using np.dot:

```
In [113]: arr = np.random.randn(6, 3)

In [114]: np.dot(arr.T, arr)
Out[114]:
     array([[ 2.584 ,  1.8753,  0.8888],
            [ 1.8753,  6.6636,  0.3884],
            [ 0.8888,  0.3884,  3.9781]])
```

For higher dimensional arrays, transposewill accept a tuple of axis numbers to permute the axes (for extra mind bending):

```
In [115]: arr = np.arange(16).reshape((2, 2, 4))

In [116]: arr
Out[116]:
     array([[[ 0,  1,  2,  3],
             [ 4,  5,  6,  7]],
            [[ 8,  9, 10, 11],
             [12, 13, 14, 15]]])

In [117]: arr.transpose((1, 0, 2))Out[117]:
     array([[[ 0,  1,  2,  3],
             [ 8,  9, 10, 11]],
            [[ 4,  5,  6,  7],
             [12, 13, 14, 15]]])
```

Simple transposing with .T is just a special case of swapping axes. ndarray has the method swapaxeswhich takes a pair of axis numbers:

```
In [118]: arr
Out[118]:
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],

       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])
```

```
In [119]: arr.swapaxes(1, 2)
Out[119]:
array([[[ 0,  4],
        [ 1,  5],
        [ 2,  6],
        [ 3,  7]],

       [[ 8, 12],
        [ 9, 13],
        [10, 14],
        [11, 15]]])
```

swapaxessimilarly returns a view on the data without making a copy.

Universal Functions: Fast Element-wise Array Functions

A universal function, or *ufunc*, is a function that performs elementwise operations on data in ndarrays. You can think of them as fast vectorized wrappers for simple functions that take one or more scalar values and produce one or more scalar results.

Many ufuncs are simple elementwise transformations, like sqrtor exp:

```
In [120]: arr = np.arange(10)

In [121]: np.sqrt(arr)
Out[121]:
array([ 0.        ,  1.        ,  1.4142,  1.7321,  2.        ,  2.2361,  2.4495,
        2.6458,  2.8284,  3.        ])

In [122]: np.exp(arr)
Out[122]:
 array([      1.      ,      2.7183,      7.3891,     20.0855,     54.5982,
            148.4132,    403.4288,  1096.6332,  2980.958 ,  8103.0839])
```

These are referred to as *unary* ufuncs. Others, such as add or maximum, take 2 arrays (thus, *binary* ufuncs) and return a single array as the result:

```
In [123]: x = randn(8)In

[124]: y = randn(8)

In [125]: x
Out[125]:
array([ 0.0749,  0.0974,  0.2002, -0.2551,  0.4655,  0.9222,  0.446 ,
       -0.9337])

In [126]: y
Out[126]:
array([ 0.267 , -1.1131, -0.3361,  0.6117, -1.2323,  0.4788,  0.4315,
       -0.7147])

In [127]: np.maximum(x, y) # element-wise maximum
Out[127]:
array([ 0.267 ,  0.0974,  0.2002,  0.6117,  0.4655,  0.9222,  0.446 ,
       -0.7147])
```

While not common, a ufunc can return multiple arrays. modfis one example, a vectorized version of the built-in Python divmod: it returns the fractional and integral parts of a floating point array:

```
In [128]: arr = randn(7) * 5

In [129]: np.modf(arr)
Out[129]:
(array([-0.6808,  0.0636, -0.386 ,  0.1393, -0.8806,  0.9363, -0.883 ]),
 array([-2.,  4., -3.,  5., -3.,  3., -6.]))
```

See Table 4-3 and Table 4-4 for a listing of available ufuncs.

Table 4-3. Unary ufuncs

Function	Description
abs, fabs	Compute the absolute value element-wise for integer, floating point, or complex values. Use fabsas a faster alternative for non-complex-valued data
sqrt	Compute the square root of each element. Equivalent to arr ** 0.5
square	Compute the square of each element. Equivalent to arr ** 2
exp	Compute the exponent e^x of each element
log, log10, log2, log1p	Natural logarithm (base *e*), log base 10, log base 2, and log(1 + x), respectively
sign	Compute the sign of each element: 1 (positive), 0 (zero), or -1 (negative)
ceil	Compute the ceiling of each element, i.e. the smallest integer greater than or equal toeach element
floor	Compute the floor of each element, i.e. the largest integer less than or equal to eachelement
rint	Round elements to the nearest integer, preserving the dtype
modf	Return fractional and integral parts of array as separate array
isnan	Return boolean array indicating whether each value is NaN(Not a Number)
isfinite, isinf	Return boolean array indicating whether each element is finite (non-inf, non-NaN) or infinite, respectively
cos, cosh, sin, sinh,tan, tanh	Regular and hyperbolic trigonometric functions Inverse
arccos, arccosh, arcsin, arcsinh, arctan, arctanh	trigonometric functions
logical_not	Compute truth value of not xelement-wise. Equivalent to -arr.

Table 4-4. Binary universal functions

Function	Description
add	Add corresponding elements in arrays
subtract	Subtract elements in second array from first array
multiply	Multiply array elements
divide, floor_divide	Divide or floor divide (truncating the remainder)
power	Raise elements in first array to powers indicated in second array
maximum, fmax	Element-wise maximum. fmaxignores NaN
minimum, fmin	Element-wise minimum. fminignores NaN
mod	Element-wise modulus (remainder of division)
copysign	Copy sign of values in second argument to values in first argument

Function	Description
greater, greater_equal,less, less_equal, equal,not_equal	Perform element-wise comparison, yielding boolean array. Equivalent to infix operators >, >=, <, <=, ==, !=
logical_and, logical_or, logical_xor	Compute element-wise truth value of logical operation. Equivalent to infix operators & |, ^

Data Processing Using Arrays

Using NumPy arrays enables you to express many kinds of data processing tasks as concise array expressions that might otherwise require writing loops. This practice of replacing explicit loops with array expressions is commonly referred to as *vectorization*. In general, vectorized array operations will often be one or two (or more) orders of magnitude faster than their pure Python equivalents, with the biggest impact in any kind of numerical computations.

As a simple example, suppose we wished to evaluate the function sqrt(x^2 + y^2) across a regular grid of values. The np.meshgrid function takes two 1D arrays and produces two 2D matrices corresponding to all pairs of (x, y) in the two arrays:

```
In [130]: points = np.arange(-5, 5, 0.01) # 1000 equally spaced pointsIn [131]: xs, ys =

np.meshgrid(points, points)

In [132]: ys
Out[132]:
                    array([[-5.   , -5.   , -5.   , ..., -5.   , -5.   , -5.   ],
                      [-4.99, -4.99, -4.99, ..., -4.99, -4.99, -4.99],
                      [-4.98, -4.98, -4.98, ..., -4.98, -4.98, -4.98],
       ...,
       [ 4.97,   4.97,   4.97, ...,   4.97,   4.97,   4.97],
       [ 4.98,   4.98,   4.98, ...,   4.98,   4.98,   4.98],
       [ 4.99,   4.99,   4.99, ...,   4.99,   4.99,   4.99]])
```

Now, evaluating the function is a simple matter of writing the same expression you would write with two points:

```
In [134]: import matplotlib.pyplot as pltIn [135]: z

= np.sqrt(xs ** 2 + ys ** 2)

In [136]: z
Out[136]:
array([[ 7.0711,   7.064 ,   7.0569, ...,   7.0499,   7.0569,   7.064 ],
       [ 7.064 ,   7.0569,   7.0499, ...,   7.0428,   7.0499,   7.0569],
       [ 7.0569,   7.0499,   7.0428, ...,   7.0357,   7.0428,   7.0499],
       ...,
       [ 7.0499,   7.0428,   7.0357, ...,   7.0286,   7.0357,   7.0428],
       [ 7.0569,   7.0499,   7.0428, ...,   7.0357,   7.0428,   7.0499],
       [ 7.064 ,   7.0569,   7.0499, ...,   7.0428,   7.0499,   7.0569]])
```

```
In [137]: plt.imshow(z, cmap=plt.cm.gray); plt.colorbar() Out[137]:
<matplotlib.colorbar.Colorbar instance at 0x4e46d40>

In [138]: plt.title("Image plot of $\sqrt{x^2 + y^2}$ for a grid of values")Out[138]:
<matplotlib.text.Text at 0x4565790>
```

See Figure 4-3. Here I used the matplotlib function imshowto create an image plot from a 2D array of function values.

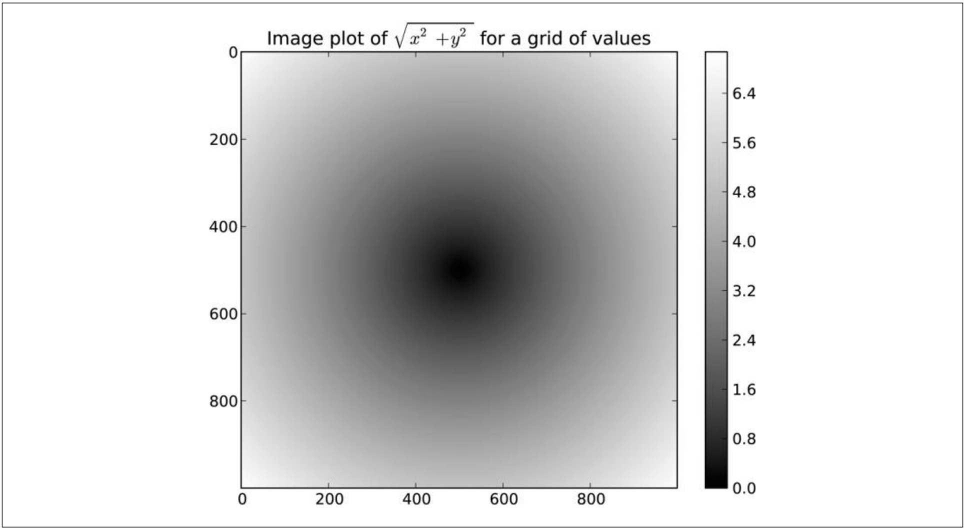


Figure 4-3. Plot of function evaluated on grid

Expressing Conditional Logic as Array Operations

The numpy.where function is a vectorized version of the ternary expression x if condi tion else y. Suppose we had a boolean array and two arrays of values:

```
In [140]: xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])

In [141]: yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])

In [142]: cond = np.array([True, False, True, True, False])
```

Suppose we wanted to take a value from xarr whenever the corresponding value in condis Trueotherwise take the value from yarr. A list comprehension doing this might look like:

```
In [143]: result = [(x if c else y)
   .....:                   for x, y, c in zip(xarr, yarr, cond)]

In [144]: result
Out[144]: [1.1000000000000001, 2.2000000000000002, 1.3, 1.3999999999999999, 2.5]
```

This has multiple problems. First, it will not be very fast for large arrays (because all the work is being done in pure Python). Secondly, it will not work with multidimensional arrays. With np.whereyou can write this very concisely:

```
In [145]: result = np.where(cond, xarr, yarr)

In [146]: result
Out[146]: array([ 1.1,   2.2,   1.3,   1.4,   2.5])
```

The second and third arguments to np.where don't need to be arrays; one or both of them can be scalars. A typical use of wherein data analysis is to produce a new array of values based on another array. Suppose you had a matrix of randomly generated data and you wanted to replace all positive values with 2 and all negative values with -2. This is very easy to do with np.where:

```
In [147]: arr = randn(4, 4)

In [148]: arr
Out[148]:
          array([[ 0.6372,   2.2043,   1.7904,   0.0752],
                 [-1.5926, -1.1536,   0.4413,   0.3483],
                 [-0.1798,   0.3299,   0.7827, -0.7585],
                 [ 0.5857,   0.1619,   1.3583, -1.3865]])

In [149]: np.where(arr > 0, 2, -2)Out[149]:
       array([[ 2,   2,   2,   2],
              [-2, -2,   2,   2],
              [-2,   2,   2, -2],
              [ 2,   2,   2, -2]])

In [150]: np.where(arr > 0, 2, arr) # set only positive values to 2Out[150]:
 array([[ 2.        ,   2.       ,   2.       ,   2.       ],
        [-1.5926, -1.1536,   2.            ,   2.       ],
          [-0.1798,   2.        ,   2.       , -0.7585],
          [ 2.       ,   2.       ,   2.       , -1.3865]])
```

The arrays passed to wherecan be more than just equal sizes array or scalers.

With some cleverness you can use whereto express more complicated logic; consider this example where I have two boolean arrays, cond1 and cond2, and wish to assign a different value for each of the 4 possible pairs of boolean values:

```
result = []
for i in range(n):
    if cond1[i] and cond2[i]:
        result.append(0)
    elif cond1[i]:
        result.append(1)
    elif cond2[i]:
        result.append(2)
    else:
        result.append(3)
```

While perhaps not immediately obvious, this for loop can be converted into a nested where expression:

```
np.where(cond1 & cond2, 0,
         np.where(cond1, 1,
                  np.where(cond2, 2, 3)))
```

In this particular example, we can also take advantage of the fact that boolean values are treated as 0 or 1 in calculations, so this could alternatively be expressed (though a bit more cryptically) as an arithmetic operation:

```
result = 1 * cond1 + 2 * cond2 + 3 * -(cond1 | cond2)
```

Mathematical and Statistical Methods

A set of mathematical functions which compute statistics about an entire array or about the data along an axis are accessible as array methods. Aggregations (often called *reductions*) like sum, mean, and standard deviation std can either be used by calling the array instance method or using the top level NumPy function:

```
In [151]: arr = np.random.randn(5, 4) # normally-distributed data

In [152]: arr.mean()
Out[152]: 0.062814911084854597

In [153]: np.mean(arr)
Out[153]: 0.062814911084854597

In [154]: arr.sum()
Out[154]: 1.2562982216970919
```

Functions like mean and sum take an optional axis argument which computes the statistic over the given axis, resulting in an array with one fewer dimension:

```
In [155]: arr.mean(axis=1)
Out[155]: array([-1.2833,  0.2844,  0.6574,  0.6743, -0.0187])

In [156]: arr.sum(0)
Out[156]: array([-3.1003, -1.6189,  1.4044,  4.5712])
```

Other methods like cumsum and cumprod do not aggregate, instead producing an array of the intermediate results:

```
In [157]: arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])

In [158]: arr.cumsum(0)                In [159]: arr.cumprod(1)
Out[158]:                              Out[159]:
array([[ 0,  1,  2],                   array([[  0,   0,   0],
       [ 3,  5,  7],                          [  3,  12,  60],
       [ 9, 12, 15]])                         [  6,  42, 336]])
```

See Table 4-5 for a full listing. We'll see many examples of these methods in action in later chapters.

Table 4-5. Basic array statistical methods

Method	Description
sum	Sum of all the elements in the array or along an axis. Zero-length arrays have sum 0.
mean	Arithmetic mean. Zero-length arrays have NaNmean.
std, var	Standard deviation and variance, respectively, with optional degrees of freedom adjust- ment (default denominator n).
min, max	Minimum and maximum.
argmin, argmax	Indices of minimum and maximum elements, respectively.
cumsum	Cumulative sum of elements starting from 0
cumprod	Cumulative product of elements starting from 1

Methods for Boolean Arrays

Boolean values are coerced to 1 (True) and 0 (False) in the above methods. Thus, sum is often used as a means of counting Truevalues in a boolean array:

```
In [160]: arr = randn(100)

In [161]: (arr > 0).sum() # Number of positive valuesOut[161]: 44
```

There are two additional methods, any and all, useful especially for boolean arrays. any tests whether one or more values in an array is True, while all checks if every value is True:

```
In [162]: bools = np.array([False, False, True, False])

In [163]: bools.any()
Out[163]: True

In [164]: bools.all()
Out[164]: False
```

These methods also work with non-boolean arrays, where non-zero elements evaluate to True.

Sorting

Like Python’s built-in list type, NumPy arrays can be sorted in-place using the sort method:

```
In [165]: arr = randn(8)

In [166]: arr
Out[166]:
array([ 0.6903,   0.4678,   0.0968, -0.1349,   0.9879,   0.0185, -1.3147,
           -0.5425])

In [167]: arr.sort()
In [168]: arr
Out[168]:
array([-1.3147, -0.5425, -0.1349,   0.0185,   0.0968,   0.4678,   0.6903,
              0.9879])
```

Multidimensional arrays can have each 1D section of values sorted in-place along an axis by passing the axis number to sort:

```
In [169]: arr = randn(5, 3)

In [170]: arr
Out[170]:
            array([[-0.7139, -1.6331, -0.4959],
                   [ 0.8236, -1.3132, -0.1935],
                   [-1.6748,   3.0336, -0.863 ],
                   [-0.3161,   0.5362, -2.468 ],
                   [ 0.9058,   1.1184, -1.0516]])

In [171]: arr.sort(1)

In [172]: arr
Out[172]:
            array([[-1.6331, -0.7139, -0.4959],
                   [-1.3132, -0.1935,   0.8236],
                   [-1.6748, -0.863 ,   3.0336],
                   [-2.468 , -0.3161,   0.5362],
                   [-1.0516,   0.9058,   1.1184]])
```

The top level method np.sort returns a sorted copy of an array instead of modifying the array in place. A quick-and-dirty way to compute the quantiles of an array is to sort it and select the value at a particular rank:

```
In [173]: large_arr = randn(1000)In [174]:

large_arr.sort()

In [175]: large_arr[int(0.05 * len(large_arr))] # 5% quantileOut[175]: -
1.5791023260896004
```

For more details on using NumPy's sorting methods, and more advanced techniques like indirect sorts, see Chapter 12. Several other kinds of data manipulations related to sorting (for example, sorting a table of data by one or more columns) are also to be found in pandas.

Unique and Other Set Logic

NumPy has some basic set operations for one-dimensional ndarrays. Probably the most commonly used one is np.unique, which returns the sorted unique values in an array:

```
In [176]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])

In [177]: np.unique(names)
Out[177]:
```

```
array(['Bob', 'Joe', 'Will'],
      dtype='|S4')

In [178]: ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])

In [179]: np.unique(ints)
Out[179]: array([1, 2, 3, 4])
```

Contrast np.uniquewith the pure Python alternative:

```
In [180]: sorted(set(names)) Out[180]:
['Bob', 'Joe', 'Will']
```

Another function, np.in1d, tests membership of the values in one array in another, returning a boolean array:

```
In [181]: values = np.array([6, 0, 0, 3, 2, 5, 6])

In [182]: np.in1d(values, [2, 3, 6])
Out[182]: array([ True, False, False,   True,   True, False,   True], dtype=bool)
```

See Table 4-6 for a listing of set functions in NumPy.

Table 4-6. Array set operations

Method	Description
unique(x)	Compute the sorted, unique elements in x
intersect1d(x, y)	Compute the sorted, common elements in xand y
union1d(x, y)	Compute the sorted union of elements
in1d(x, y)	Compute a boolean array indicating whether each element of xis contained in y
setdiff1d(x, y)	Set difference, elements in x that are not in y
setxor1d(x, y)	Set symmetric differences; elements that are in either of the arrays, but not both

Unit-II:

Getting Started with pandas: Introduction to pandas, Library Architecture, Features, Applications, Data Structures, Series, DataFrame, Index Objects, Essential Functionality Reindexing, Dropping entries from an axis, Indexing, selection, and filtering),Sorting and ranking, Summarizing and Computing Descriptive Statistics, Unique Values, Value Counts, Handling Missing Data, filtering out missing data.

Getting Started with pandas

pandas will be the primary library of interest throughout much of the rest of the book. It contains high-level data structures and manipulation tools designed to make data analysis fast and easy in Python. pandas is built on top of NumPy and makes it easy to use in NumPy-centric applications.

As a bit of background, I started building pandas in early 2008 during my tenure at AQR, a quantitative investment management firm. At the time, I had a distinct set of requirements that were not well-addressed by any single tool at my disposal:

- Data structures with labeled axes supporting automatic or explicit data alignment. This prevents common errors resulting from misaligned data and working with differently-indexed data coming from different sources.
- Integrated time series functionality.
- The same data structures handle both time series data and non-time series data.
- Arithmetic operations and reductions (like summing across an axis) would pass on the metadata (axis labels).
- Flexible handling of missing data.
- Merge and other relational operations found in popular database databases (SQL-based, for example).

I wanted to be able to do all of these things in one place, preferably in a language well-suited to general purpose software development. Python was a good candidate language for this, but at that time there was not an integrated set of data structures and tools providing this functionality.

Over the last four years, pandas has matured into a quite large library capable of solving a much broader set of data handling problems than I ever anticipated, but it has expanded in its scope without compromising the simplicity and ease-of-use that I desired from the very beginning. I hope that after reading this book, you will find it to be just as much of an indispensable tool as I do.

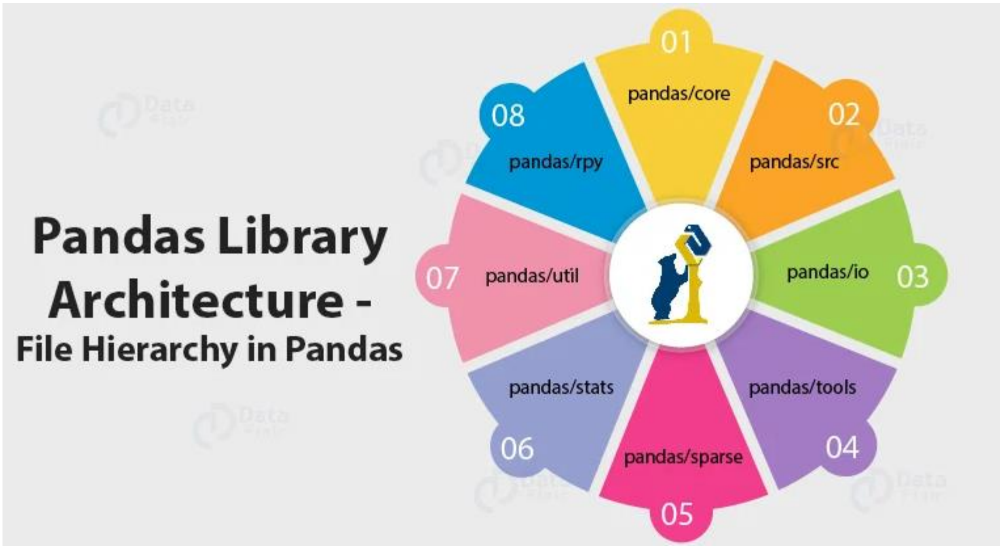
Throughout the rest of the book, I use the following import conventions for pandas:

```
In [1]: from pandas import Series, DataFrameIn [2]:

import pandas as pd
```

Thus, whenever you see pd.in code, it's referring to pandas. Series and DataFrame are used so much that I find it easier to import them into the local namespace.

Library Architecture



Pandas Library Architecture

The following list gives us an idea about the hierarchy of the files within Pandas Library Architecture:

1. pandas/core

In Pandas library architecture, this part consists of basic files about the data structures present within the library. For examples, data structures – Series and DataFrames. There are various Python files within the core. The most important of them being:

- **api.py:** Important key modules which will be used later are imported using these files.
- **base.py:** This will provides the base for all the other classes present, like PandasObject and StringMIxin.
- **common.py:** It controls the common utility methods which help in handling various data structures.

- **config.py:** This helps to handle configurable objects found throughout the package.

These are the **essential python classes** which handle most of the working in the core of Pandas.

2. pandas/src

This contains algorithms which provide basic functionality to the library. The code here is usually written in C or Cython.

3. pandas/io

pandas/io, an essential part of the Pandas library architecture. This contains input and output tools which help Pandas handle files of various file formats. Essential modules found here are:

- **api.py:** This module handles various imports needed for input and output functions.
- **auth.py:** This module handles authentications and the methods dealing with it.
- **common.py:** Common functionality of input and output functions are taken care of by this module.
- **data.py:** This module helps to handle data with is input or output.

4. pandas/tools

The algorithms of pandas/tools are for auxiliary data. These help various functions like pivot, merge, join, concatenation, and other such functions for manipulating the data sets.

5. pandas/sparse

This part consists of sparse versions of various data structures like DataFrames and Series. A sparse version means that the data is mostly missing or unavailable.

6. pandas/stats

This part of the Pandas library architecture consists of a panel and linear regression and also contains moving window regression. Various statistics-related functions can be found in this portion.

7. pandas/util

Various utilities, testing tools, development can be found here. In pandas/util, classes are used to make testing and debugging any part of the library.

8. pandas/rpy

It consists of an interface to connect to R programming, called RPy2. Using Pandas with both R and Python can help you to have a much better grasp over data analysis.

Key Features of Pandas

- Fast and efficient DataFrame object with default and customized indexing.
- Tools for loading data into in-memory data objects from different file formats.
- Data alignment and integrated handling of missing data.
- Reshaping and pivoting of date sets.
- Label-based slicing, indexing and subsetting of large data sets.
- Columns from a data structure can be deleted or inserted.
- Group by data for aggregation and transformations.
- High performance merging and joining of data.
- Time Series functionality.

Major Applications of Pandas

In this list we will cover the most fundamental applications of Pandas:

1. Economics

Economics is in constant demand for data analysis. Analyzing data to form patterns and understanding trends about how the economy in various sectors is growing, is something very essential for economists. Therefore, a lot of economists have started using Python and Pandas to analyze huge datasets. Pandas provide a comprehensive set of tools, like dataframes and file-handling. These tools help immensely in accessing and manipulating data to get the desired results. Through these applications of Pandas, economists all around the world have been able to make breakthroughs like never before.

2. Recommendation Systems

We all have used Spotify or Netflix and been appalled at the brilliant recommendations provided by these sites. These systems are a **miracle of Deep Learning**. Such models for providing recommendations is one of the most important applications of Pandas. Mostly, these models are made in python and Pandas being the main libraries of python, used when handling data in such models. We know that Pandas are best for managing huge amounts of data. And the recommendation system is possible only by learning and handling huge masses of data. Functions like groupBy and mapping help tremendously in making these systems possible.

3. Stock Prediction

The stock market is extremely volatile. However, that doesn't mean that it cannot be predicted. With the help of Pandas and a few other libraries like NumPy and matplotlib, we can easily make models which can predict how the stock markets turn out. This is possible because there is a lot of previous data of stocks which tells us about how they behave. And by learning these data of stocks, a model can easily predict the next move to be taken with some accuracy. Not only this, but people can also automate buying and selling of stocks with the help of such prediction models.

4. Neuroscience

Understanding the nervous system has always been in the minds of humankind because there are a lot of potential mysteries about our bodies which we haven't solved as of yet. **Machine learning** has helped this field immensely with the help of the various applications of Pandas. Again, the data manipulation capabilities of Pandas have played a major role in compiling a huge amount of data which has helped neuroscientists in understanding trends that are followed inside our bodies and the effect of various things on our entire nervous system.

5. Statistics

Pure maths itself has made much progress with the various applications of Pandas. Since Statistic deals with a lot of data, a library like Pandas which deals with data handling has helped in a lot of different ways. The functions of mean, median and mode are just very basic ones which help in performing statistical calculations. There are a lot of other complex functions associated with statistics and pandas plays a huge role in these so as to bring perfect results.

6. Advertising

Advertising has taken a huge leap in the 21st Century. Nowadays advertising has become very personalized which helps companies to get more and more customers. This again has been possible only because of the likes of Machine Learning and Deep Learning. Models going through customer data learn to understand what exactly the customer wants, providing companies with great advertisement ideas. There are many applications of Pandas in this. The customer data often rendered with the help of this library, and a lot of functions present in Pandas also help.

7. Analytics

Analytics has become easier than ever with the use of Pandas. Whether it is website analytics or analytics of some other platform, Pandas do it all, with its amazing data manipulation and handling capabilities. The visualization capabilities of pandas play a big role too in this field. It not only takes in data and displays it but also helps in applying a lot of functions over the data.

8. Natural Language Processing

NLP or **Natural Language processing** has taken the world by a storm and it is creating a lot of buzzes. The main concept is to decipher human language and several nuances related to it. This is very difficult, but with the help of the various applications of Pandas and Scikit-learn, it is easier to create an NLP model which we can be improved continuously with the help of various other libraries and their functions.

9. Big Data

One of the applications of Pandas is that it can work with Big data too. Python has a good connection with Hadoop and Spark, allowing Pandas to have access to Big Data. One can easily write to Spark or **Hadoop** also with the help of Pandas.

10. Data Science

Pandas and Data science are almost synonymous. Most of the examples are a product of Data Science itself. It is a very broad umbrella which encompasses anything that deals with analyzing data, and thus almost all applications of Pandas fall under the **scope of Data science**. Pandas mainly used for processing the data. Therefore Data Science on Python without Pandas is very difficult.

Introduction to pandas Data Structures

To get started with pandas, you will need to get comfortable with its two workhorse data structures: Series and DataFrame. While they are not a universal solution for every problem, they provide a solid, easy-to-use basis for most applications.

Series

A Series is a one-dimensional array-like object containing an array of data (of any NumPy data type) and an associated array of data labels, called its *index*. The simplest Series is formed from only an array of data:

```
In [4]: obj = Series([4, 7, -5, 3])

In [5]: obj
Out[5]:
0      4
1      7
2     -5
3      3
```

The string representation of a Series displayed interactively shows the index on the left and the values on the right. Since we did not specify an index for the data, a default one consisting of the integers 0 through N - 1 (where N is the length of the data) is created. You can get the array representation and index object of the Series via its values and index attributes, respectively:

```
In [6]: obj.values
Out[6]: array([ 4,   7, -5,   3])

In [7]: obj.index
Out[7]: Int64Index([0, 1, 2, 3])
```

Often it will be desirable to create a Series with an index identifying each data point:

```
In [8]: obj2 = Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])

In [9]: obj2
Out[9]:
d      4
b      7
a     -5
c      3
```

```
In [10]: obj2.index
Out[10]: Index([d, b, a, c], dtype=object)
```

Compared with a regular NumPy array, you can use values in the index when selecting single values or a set of values:

```
In [11]: obj2['a']
Out[11]: -5

In [12]: obj2['d'] = 6

In [13]: obj2[['c', 'a', 'd']]Out[13]:
c    3
a   -5
d    6
```

NumPy array operations, such as filtering with a boolean array, scalar multiplication, or applying math functions, will preserve the index-value link:

```
In [14]: obj2
Out[14]:
d    6
b    7
a   -5
c    3
```

```
In [15]: obj2[obj2 > 0]      In [16]: obj2 * 2      In [17]: np.exp(obj2)
Out[15]:                     Out[16]:              Out[17]:
d    6                       d    12               d       403.428793
b    7                       b    14               b      1096.633158
c    3                       a   -10               a         0.006738
                             c     6               c        20.085537
```

Another way to think about a Series is as a fixed-length, ordered dict, as it is a mapping of index values to data values. It can be substituted into many functions that expect a dict:

```
In [18]: 'b' in obj2
Out[18]: True

In [19]: 'e' in obj2
Out[19]: False
```

Should you have data contained in a Python dict, you can create a Series from it by passing the dict:

```
In [20]: sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}In [21]: obj3 =

Series(sdata)

In [22]: obj3
Out[22]:
Ohio      35000
Oregon    16000
```

```
Texas      71000
Utah        5000
```

When only passing a dict, the index in the resulting Series will have the dict’s keys in sorted order.

```
In [23]: states = ['California', 'Ohio', 'Oregon', 'Texas']In [24]: obj4 =

Series(sdata, index=states)

In [25]: obj4
Out[25]:
California         NaN
Ohio             35000
Oregon           16000
Texas            71000
```

In this case, 3 values found in sdata were placed in the appropriate locations, but since no value for 'California' was found, it appears as NaN (not a number) which is considered in pandas to mark missing or *NA* values. I will use the terms “missing” or “NA” to refer to missing data. The isnull and notnull functions in pandas should be used to detect missing data:

```
In [26]: pd.isnull(obj4)            In [27]: pd.notnull(obj4)
Out[26]:                            Out[27]:
California       True               California      False
Ohio            False               Ohio             True
Oregon          False               Oregon           True
Texas           False               Texas            True
```

Series also has these as instance methods:

```
In [28]: obj4.isnull()
Out[28]:
California       True
Ohio            False
Oregon          False
Texas           False
```

I discuss working with missing data in more detail later in this chapter.

A critical Series feature for many applications is that it automatically aligns differently-indexed data in arithmetic operations:

```
In [29]: obj3               In [30]: obj4
Out[29]:                    Out[30]:
Ohio       35000            California        NaN
Oregon     16000            Ohio            35000
Texas      71000            Oregon          16000
Utah        5000            Texas           71000

In [31]: obj3 + obj4
Out[31]:
California          NaN
Ohio              70000
Oregon            32000
```

```
Texas          142000
Utah              NaN
```

Data alignment features are addressed as a separate topic.

Both the Series object itself and its index have a name attribute, which integrates with other key areas of pandas functionality:

```
In [32]: obj4.name = 'population' In [33]:

obj4.index.name = 'state'

In [34]: obj4
Out[34]:
state
California      NaN
Ohio          35000
Oregon        16000
Texas         71000
Name: population
```

A Series's index can be altered in place by assignment:

```
In [35]: obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']

In [36]: obj
Out[36]:
Bob      4
Steve    7
Jeff    -5
Ryan     3
```

DataFrame

A DataFrame represents a tabular, spreadsheet-like data structure containing an ordered collection of columns, each of which can be a different value type (numeric, string, boolean, etc.). The DataFrame has both a row and column index; it can be thought of as a dict of Series (one for all sharing the same index). Compared with other such DataFrame-like structures you may have used before (like R's data.frame), row-oriented and column-oriented operations in DataFrame are treated roughly symmetrically. Under the hood, the data is stored as one or more two-dimensional blocks rather than a list, dict, or some other collection of one-dimensional arrays. The exact details of DataFrame's internals are far outside the scope of this book.

While DataFrame stores the data internally in a two-dimensional format, you can easily represent much higher-dimensional data in a tabular format using hierarchical indexing, a subject of a later section and a key ingredient in many of the more advanced data-handling features in pandas.

There are numerous ways to construct a DataFrame, though one of the most common is from a dict of equal-length lists or NumPy arrays

```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],'year': [2000, 2001,
             2002, 2001, 2002],
             'pop': [1.5, 1.7, 3.6, 2.4, 2.9]}
frame = DataFrame(data)
```

The resulting DataFrame will have its index assigned automatically as with Series, and the columns are placed in sorted order:

```
In [38]: frame
Out[38]:
     pop     state   year0
1.5            Ohio   2000
1   1.7        Ohio   2001
2   3.6        Ohio   2002
3   2.4   Nevada   2001
4   2.9   Nevada   2002
```

If you specify a sequence of columns, the DataFrame’s columns will be exactly what you pass:

```
In [39]: DataFrame(data, columns=['year', 'state', 'pop'])Out[39]:
     year      state   pop0
2000            Ohio   1.5
1   2001        Ohio   1.7
2   2002        Ohio   3.6
3   2001   Nevada   2.4
4   2002   Nevada   2.9
```

As with Series, if you pass a column that isn’t contained in data, it will appear with NA values in the result:

```
In [40]: frame2 = DataFrame(data, columns=['year', 'state', 'pop', 'debt'],
....:                                   index=['one', 'two', 'three', 'four', 'five'])

In [41]: frame2
Out[41]:
           year      state pop debtone
           2000       Ohio   1.5
           NaN
two        2001       Ohio   1.7       NaN
three   2002          Ohio   3.6       NaN
four       2001   Nevada   2.4       NaN
five       2002   Nevada   2.9       NaN

In [42]: frame2.columns
Out[42]: Index([year, state, pop, debt], dtype=object)
```

A column in a DataFrame can be retrieved as a Series either by dict-like notation or by attribute:

```
In [43]: frame2['state']                In [44]: frame2.year
Out[43]:                                Out[44]:
one              Ohio                   one          2000
```

```
two        Ohio                 two      2001
three      Ohio                 three    2002
four     Nevada                 four     2001
five     Nevada                 five     2002
Name: state                     Name: year
```

Note that the returned Series have the same index as the DataFrame, and their name attribute has been appropriately set.

Rows can also be retrieved by position or name by a couple of methods, such as the ixindexing field (much more on this later):

```
In [45]: frame2.ix['three']Out[45]:
year      2002
state     Ohio
pop        3.6
debt
           Na
NName: three
```

Columns can be modified by assignment. For example, the empty 'debt'column couldbe assigned a scalar value or an array of values:

```
In [46]: frame2['debt'] = 16.5

In [47]: frame2
Out[47]:
       year    state pop debtone
       2000     Ohio  1.5  16.5
two    2001     Ohio  1.7  16.5
three  2002     Ohio  3.6  16.5
four   2001   Nevada  2.4  16.5
five   2002   Nevada  2.9  16.5

In [48]: frame2['debt'] = np.arange(5.)In [49]:

frame2
Out[49]:
       year    state pop debtone
       2000     Ohio  1.5      0
two    2001     Ohio  1.7      1
three  2002     Ohio  3.6      2
four   2001   Nevada  2.4      3
five   2002   Nevada  2.9      4
```

When assigning lists or arrays to a column, the value’s length must match the length of the DataFrame. If you assign a Series, it will be instead conformed exactly to the DataFrame’s index, inserting missing values in any holes:

```
In [50]: val = Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])In [51]: frame2['debt'] =

val

In [52]: frame2
Out[52]:
       year    state  pop  debt
```

```
one      2000     Ohio  1.5     NaN
two      2001     Ohio  1.7  -1.2
three  2002       Ohio  3.6     NaN
four     2001  Nevada  2.4  -1.5
five     2002  Nevada  2.9  -1.7
```

Assigning a column that doesn’t exist will create a new column. The delkeyword will delete columns as with a dict:

```
In [53]: frame2['eastern'] = frame2.state == 'Ohio'

In [54]: frame2
Out[54]:
         year     state  pop  debt easternone
         2000     Ohio  1.5     NaN     True
two      2001     Ohio  1.7  -1.2       True
three  2002       Ohio  3.6     NaN     True
four     2001  Nevada  2.4  -1.5       False
five     2002  Nevada  2.9  -1.7       False

In [55]: del frame2['eastern']

In [56]: frame2.columns
Out[56]: Index([year, state, pop, debt], dtype=object)
```

The column returned when indexing a DataFrame is a *view* on the underlying data, not a copy. Thus, any in-place modifications to the Series will be reflected in the DataFrame. The column can be explicitly copied using the Series’s copymethod.

Another common form of data is a nested dict of dicts format:

```
In [57]: pop = {'Nevada': {2001: 2.4, 2002: 2.9},
   ....:                 'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}
```

If passed to DataFrame, it will interpret the outer dict keys as the columns and the inner keys as the row indices:

```
In [58]: frame3 = DataFrame(pop)

In [59]: frame3
Out[59]:
        Nevada  Ohio
 2000       NaN   1.5
 2001       2.4   1.7
 2002       2.9   3.6
```

Of course you can always transpose the result:

```
In [60]: frame3.T
Out[60]:
            2000  2001  2002
 Nevada     NaN   2.4    2.9
 Ohio       1.5   1.7    3.6
```

The keys in the inner dicts are unioned and sorted to form the index in the result. This isn't true if an explicit index is specified:

```
In [61]: DataFrame(pop, index=[2001, 2002, 2003])Out[61]:
          Nevada   Ohio
 2001        2.4      1.7
 2002        2.9      3.6
2003        NaN    NaN
```

Dicts of Series are treated much in the same way:

```
In [62]: pdata = {'Ohio': frame3['Ohio'][:-1],
     ....:                       'Nevada': frame3['Nevada'][:2]}

In [63]: DataFrame(pdata)
Out[63]:
          Nevada   Ohio
 2000        NaN    1.5
 2001        2.4      1.7
```

For a complete list of things you can pass the DataFrame constructor, see Table 5-1.

If a DataFrame's index and columns have their name attributes set, these will also be displayed:

```
In [64]: frame3.index.name = 'year'; frame3.columns.name = 'state'

In [65]: frame3
Out[65]:
state Nevada Ohioyear
2000          NaN    1.5
2001          2.4      1.7
2002          2.9      3.6
```

Like Series, the values attribute returns the data contained in the DataFrame as a 2Dndarray:

```
In [66]: frame3.values
Out[66]:
        array([[ nan,   1.5],
                  [ 2.4,   1.7],
                  [ 2.9,   3.6]])
```

If the DataFrame's columns are different dtypes, the dtype of the values array will be chosen to accomodate all of the columns:

```
In [67]: frame2.values
Out[67]:
          array([[2000, Ohio, 1.5, nan],
                    [2001, Ohio, 1.7, -1.2],
                   [2002, Ohio, 3.6, nan],
           [2001, Nevada, 2.4, -1.5],
           [2002, Nevada, 2.9, -1.7]], dtype=object)
```

Table 5-1. Possible data inputs to DataFrame constructor

Type	Notes
2D ndarray	A matrix of data, passing optional row and column labels
dict of arrays, lists, or tuples	Each sequence becomes a column in the DataFrame. All sequences must be the same length.
NumPy structured/record array	Treated as the "dict of arrays" case
dict of Series	Each value becomes a column. Indexes from each Series are unioned together to form theresult's row index if no explicit index is passed.
dict of dicts	Each inner dict becomes a column. Keys are unioned to form the row index as in the "dict ofSeries" case.
list of dicts or Series	Each item becomes a row in the DataFrame. Union of dict keys or Series indexes become theDataFrame's column labels
List of lists or tuples	Treated as the "2D ndarray" case
Another DataFrame	The DataFrame's indexes are used unless different ones are passed
NumPy MaskedArray	Like the "2D ndarray" case except masked values become NA/missing in the DataFrame result

Index Objects

pandas's Index objects are responsible for holding the axis labels and other metadata (like the axis name or names). Any array or other sequence of labels used when constructing a Series or DataFrame is internally converted to an Index:

```
In [68]: obj = Series(range(3), index=['a', 'b', 'c'])In [69]: index =

obj.index

In [70]: index
Out[70]: Index([a, b, c], dtype=object)

In [71]: index[1:]
Out[71]: Index([b, c], dtype=object)
```

Index objects are immutable and thus can't be modified by the user:

```
In [72]: index[1] = 'd'
---------------------------------------------------------------------
Exception                                         Traceback (most recent call last)
<ipython-input-72-676fdeb26a68> in <module>()
----> 1 index[1] = 'd'
/Users/wesm/code/pandas/pandas/core/index.pyc in       __setitem  __(self, key, value)
      302        def  __setitem  __(self, key, value):
      303              """Disable the setting of values."""
--> 304              raise Exception(str(self.       __class  __) + ' object is immutable')
      305
      306        def  __getitem  __(self, key):
Exception: <class 'pandas.core.index.Index'> object is immutable
```

Immutability is important so that Index objects can be safely shared among data structures:

```
In [73]: index = pd.Index(np.arange(3))

In [74]: obj2 = Series([1.5, -2.5, 0], index=index)In [75]:

obj2.index is index
Out[75]: True
```

Table 5-2 has a list of built-in Index classes in the library. With some development effort, Index can even be subclassed to implement specialized axis indexing functionality.

Many users will not need to know much about Index objects, but they're nonetheless an important part of pandas's data model.

Table 5-2. Main Index objects in pandas

Class	Description
Index	The most general Index object, representing axis labels in a NumPy array of Python objects.
Int64Index	Specialized Index for integer values.
MultiIndex	"Hierarchical" index object representing multiple levels of indexing on a single axis. Can be thought ofas similar to an array of tuples.
DatetimeIndex	Stores nanosecond timestamps (represented using NumPy's datetime64 dtype).
PeriodIndex	Specialized Index for Period data (timespans).

In addition to being array-like, an Index also functions as a fixed-size set:

```
In [76]: frame3
Out[76]:
state Nevada Ohioyear
2000         NaN    1.5
2001         2.4    1.7
2002         2.9    3.6

In [77]: 'Ohio' in frame3.columnsOut[77]: True

In [78]: 2003 in frame3.index
Out[78]: False
```

Each Index has a number of methods and properties for set logic and answering other common questions about the data it contains. These are summarized in Table 5-3.

Table 5-3. Index methods and properties

Method	Description
append	Concatenate with additional Index objects, producing a new Index
diff	Compute set difference as an Index
intersection	Compute set intersection
union	Compute set union
isin	Compute boolean array indicating whether each value is contained in the passed collection
delete	Compute new Index with element at index ideleted
drop	Compute new index by deleting passed values
insert	Compute new Index by inserting element at index i
is_monotonic	Returns Trueif each element is greater than or equal to the previous element
is_unique	Returns Trueif the Index has no duplicate values
unique	Compute the array of unique values in the Index

Essential Functionality

In this section, I’ll walk you through the fundamental mechanics of interacting with the data contained in a Series or DataFrame. Upcoming chapters will delve more deeply into data analysis and manipulation topics using pandas. This book is not intended to serve as exhaustive documentation for the pandas library; I instead focus on the most important features, leaving the less common (that is, more esoteric) things for you to explore on your own.

Reindexing

A critical method on pandas objects is reindex, which means to create a new object with the data *conformed* to a new index. Consider a simple example from above:

```
In [79]: obj = Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])

In [80]: obj
Out[80]:
d    4.5
b    7.2
a   -5.3
c    3.6
```

Calling reindex on this Series rearranges the data according to the new index, introducing missing values if any index values were not already present:

```
In [81]: obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])

In [82]: obj2
Out[82]:
a   -5.3
```

```
b       7.2
c       3.6
d       4.5
e      NaN

In [83]: obj.reindex(['a', 'b', 'c', 'd', 'e'], fill_value=0)Out[83]:
a     -5.3
b       7.2
c       3.6
d       4.5
e      0.0
```

For ordered data like time series, it may be desirable to do some interpolation or filling of values when reindexing. The methodoption allows us to do this, using a method such as ffillwhich forward fills the values:

```
In [84]: obj3 = Series(['blue', 'purple', 'yellow'], index=[0, 2, 4])

In [85]: obj3.reindex(range(6), method='ffill')Out[85]:
0         blue
1         blue
2      purple
3      purple
4      yellow
5      yellow
```

Table 5-4 lists available methodoptions. At this time, interpolation more sophisticated than forward- and backfilling would need to be applied after the fact.

Table 5-4. reindex method (interpolation) options

Argument	Description
ffillor pad	Fill (or carry) values forward
bfillor backfill	Fill (or carry) values backward

With DataFrame, reindex can alter either the (row) index, columns, or both. When passed just a sequence, the rows are reindexed in the result:

```
In [86]: frame = DataFrame(np.arange(9).reshape((3, 3)), index=['a', 'c', 'd'],
     ....:                                   columns=['Ohio', 'Texas', 'California'])

In [87]: frame
Out[87]:
     Ohio   Texas   Californiaa   0
          1         2
c        3         4                   5
d        6         7                   8

In [88]: frame2 = frame.reindex(['a', 'b', 'c', 'd'])In [89]: frame2
Out[89]:
    Ohio   Texas   California
a      0       1            2
b    NaN     NaN          NaN
c      3       4            5
d      6       7            8
```

The columns can be reindexed using the columnskeyword:

```
In [90]: states = ['Texas', 'Utah', 'California']

In [91]: frame.reindex(columns=states)
Out[91]:
    Texas  Utah  Californiaa  1
           NaN 2
c       4    NaN           5
d       7    NaN           8
```

Both can be reindexed in one shot, though interpolation will only apply row-wise (axis 0):

```
In [92]: frame.reindex(index=['a', 'b', 'c', 'd'], method='ffill',
    ....:                        columns=states)
Out[92]:
    Texas  Utah  Californiaa  1
           NaN 2
b       1    NaN           2
c       4    NaN           5
d       7    NaN           8
```

As you'll see soon, reindexing can be done more succinctly by label-indexing with ix:

```
In [93]: frame.ix[['a', 'b', 'c', 'd'], states]Out[93]:
    Texas  Utah  California
a      1    NaN           2
b    NaN    NaN         NaN
c      4    NaN           5
d      7    NaN           8
```

Table 5-5. reindex function arguments

Argument	Description
index	New sequence to use as index. Can be Indexinstance or any other sequence-like Python data structure. An Indexwill be used exactly as is without any copying
method	Interpolation (fill) method, see Table 5-4 for options.
fill_value	Substitute value to use when introducing missing data by reindexing
limit	When forward- or backfilling, maximum size gap to fill
level	Match simple Index on level of MultiIndex, otherwise select subset of
copy	Do not copy underlying data if new index is equivalent to old index. Trueby default (i.e. always copy data).

Dropping entries from an axis

Dropping one or more entries from an axis is easy if you have an index array or list without those entries. As that can require a bit of munging and set logic, the drop method will return a new object with the indicated value or values deleted from an axis:

```
In [94]: obj = Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])In [95]: new_obj =

obj.drop('c')

In [96]: new_obj
Out[96]:
a       0
b       1
d       3
e       4

In [97]: obj.drop(['d', 'c'])Out[97]:
a       0
b       1
e       4
```

With DataFrame, index values can be deleted from either axis:

```
In [98]: data = DataFrame(np.arange(16).reshape((4, 4)),
     ....:                          index=['Ohio', 'Colorado', 'Utah', 'New York'],
     ....:                          columns=['one', 'two', 'three', 'four'])

In [99]: data.drop(['Colorado', 'Ohio'])Out[99]:
            one   two   three   four
Utah          8     9      10     11
New York     12    13      14     15
```

```
In [100]: data.drop('two', axis=1)                In [101]: data.drop(['two', 'four'], axis=1)Out[100]:
                                                  Out[101]:
            one   three   four                              one   three
Ohio          0       2      3                    Ohio          0       2
Colorado      4       6      7                    Colorado      4       6
Utah          8      10     11                    Utah          8      10
New York     12      14     15                    New York     12      14
```

Indexing, selection, and filtering

Series indexing (obj[.   ]) works analogously to NumPy array indexing, except you can use the Series’s index values instead of only integers. Here are some examples this:

```
In [102]: obj = Series(np.arange(4.), index=['a', 'b', 'c', 'd'])

In [103]: obj['b']              In [104]: obj[1]
Out[103]: 1.0                   Out[104]: 1.0

In [105]: obj[2:4]              In [106]: obj[['b', 'a', 'd']]Out[105]:
                                Out[106]:
```

```
c       2
d       3
```

```
b       1
a       0
d       3
```

```
In [107]: obj[[1, 3]]
Out[107]:
b       1
d       3
```

```
In [108]: obj[obj < 2]
Out[108]:
a       0
b       1
```

Slicing with labels behaves differently than normal Python slicing in that the endpoint is inclusive:

```
In [109]: obj['b':'c']
Out[109]:
b       1
c       2
```

Setting using these methods works just as you would expect:

```
In [110]: obj['b':'c'] = 5

In [111]: obj
Out[111]:
a       0
b       5
c       5
d       3
```

As you've seen above, indexing into a DataFrame is for retrieving one or more columns either with a single value or sequence:

```
In [112]: data = DataFrame(np.arange(16).reshape((4, 4)),
   .....:                 index=['Ohio', 'Colorado', 'Utah', 'New York'],
   .....:                 columns=['one', 'two', 'three', 'four'])

In [113]: data
Out[113]:
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

```
In [114]: data['two']

Ohio        1
Colorado    5
Utah        9
New York   13
Name: two
```

```
In [115]: data[['three', 'one']]Out[114]:
Out[115]:
          three  one
Ohio          2    0
Colorado      6    4
Utah         10    8
New York     14   12
```

Indexing like this has a few special cases. First selecting rows by slicing or a boolean array:

```
In [116]: data[:2]
Out[116]:
          one  two  three  four
```

```
In [117]: data[data['three'] > 5]
Out[117]:
          one  two  three  four
```

```
Ohio          0    1      2     3          Colorado    4    5      6      7
Colorado      4    5      6     7          Utah        8    9     10     11
                                           New York   12   13     14     15
```

This might seem inconsistent to some readers, but this syntax arose out of practicality and nothing more. Another use case is in indexing with a boolean DataFrame, such as one produced by a scalar comparison:

```
In [118]: data < 5
Out[118]:
               one   two   three   four  Ohio
True    True    True    True  Colorado  True
False  False    False  Utah      False    False
False   False New York   False   False   False
False

In [119]: data[data < 5] = 0

In [120]: data
Out[120]:
             one   two   three   four
Ohio           0     0       0      0
Colorado       0     5       6      7
Utah           8     9      10     11
New York      12    13      14     15
```

This is intended to make DataFrame syntactically more like an ndarray in this case.

For DataFrame label-indexing on the rows, I introduce the special indexing field ix. It enables you to select a subset of the rows and columns from a DataFrame with NumPy-like notation plus axis labels. As I mentioned earlier, this is also a less verbose way to do reindexing:

```
In [121]: data.ix['Colorado', ['two', 'three']]Out[121]:
two         5
three       6
Name: Colorado

In [122]: data.ix[['Colorado', 'Utah'], [3, 0, 1]]Out[122]:
            four   one   two
Colorado       7     0     5
Utah          11     8     9

In [123]: data.ix[2]               In [124]: data.ix[:'Utah', 'two']Out[123]:
                                   Out[124]:
one           8                    Ohio         0
two           9                    Colorado     5
three        10                    Utah         9
four         11                    Name: two
Name: Utah

In [125]: data.ix[data.three > 5, :3]Out[125]:
```

```
            one two three
Colorado      0    5     6
Utah          8    9    10
New York     12   13    14
```

So there are many ways to select and rearrange the data contained in a pandas object. For DataFrame, there is a short summary of many of them in Table 5-6. You have a number of additional options when working with hierarchical indexes as you'll later see.

When designing pandas, I felt that having to type frame[:, col] to select a column was too verbose (and error-prone), since column selection is one of the most common operations. Thus I made the design trade-off to push all of the rich label-indexing into ix.

Table 5-6. Indexing options with DataFrame

Type	Notes
obj[val]	Select single column or sequence of columns from the DataFrame. Special case con-veniences: boolean array (filter rows), slice (slice rows), or boolean DataFrame (set values based on some criterion).
obj.ix[val]	Selects single row of subset of rows from the DataFrame.
obj.ix[:, val]	Selects single column of subset of columns.
obj.ix[val1, val2]	Select both rows and columns.
reindexmethod	Conform one or more axes to new indexes.
xsmethod	Select single row or column as a Series by label.
icol, irowmethods	Select single column or row, respectively, as a Series by integer location.
get_value, set_valuemethods	Select single value by row and column label.

Arithmetic and data alignment

One of the most important pandas features is the behavior of arithmetic between objects with different indexes. When adding together objects, if any index pairs are not the same, the respective index in the result will be the union of the index pairs. Let's look at a simple example:

```
In [126]: s1 = Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e'])

In [127]: s2 = Series([-2.1, 3.6, -1.5, 4, 3.1], index=['a', 'c', 'e', 'f', 'g'])

In [128]: s1             In [129]: s2
Out[128]:                Out[129]:
a    7.3                 a   -2.1
c   -2.5                 c    3.6
d    3.4                 e   -1.5
```

```
e       1.5                 f       4.0
                            g       3.1
```

Adding these together yields:

```
In [130]: s1 + s2
Out[130]:
a       5.2
c       1.1
d       NaN
e       0.0
f       NaN
g       NaN
```

The internal data alignment introduces NA values in the indices that don’t overlap. Missing values propagate in arithmetic computations.

In the case of DataFrame, alignment is performed on both the rows and the columns:

```
In [131]: df1 = DataFrame(np.arange(9.).reshape((3, 3)), columns=list('bcd'),
   .....:                            index=['Ohio', 'Texas', 'Colorado'])

In [132]: df2 = DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'),
   .....:                            index=['Utah', 'Ohio', 'Texas', 'Oregon'])

In [133]: df1                     In [134]: df2
Out[133]:                         Out[134]:
           b  c  d                          b    d    e
Ohio       0  1  2                Utah      0    1    2
Texas      3  4  5                Ohio      3    4    5
Colorado   6  7  8                Texas     6    7    8
                                  Oregon    9   10   11
```

Adding these together returns a DataFrame whose index and columns are the unions of the ones in each DataFrame:

```
In [135]: df1 + df2
Out[135]:
             b c d eColorado
NaN NaN NaN NaNOhio   3
NaN    6  NaN  Oregon  NaN
NaN NaN NaN Texas       9
NaN   12 NaN
Utah     NaN NaN NaN NaN
```

Arithmetic methods with fill values

In arithmetic operations between differently-indexed objects, you might want to fill with a special value, like 0, when an axis label is found in one object but not the other:

```
In [136]: df1 = DataFrame(np.arange(12.).reshape((3, 4)), columns=list('abcd'))In [137]: df2 =

DataFrame(np.arange(20.).reshape((4, 5)), columns=list('abcde'))

In [138]: df1                  In [139]: df2
Out[138]:                      Out[139]:
   a  b    c    d                 a    b    c    d    e
```

```
0  0  1      2    3            0     0     1     2     3     4
1  4  5      6    7            1     5     6     7     8     9
2  8  9  10   11                  2   10   11   12   13   14
                                  3   15   16   17   18   19
```

Adding these together results in NA values in the locations that don’t overlap:

```
In [140]: df1 + df2
Out[140]:
      a     b     c     d     e
0     0     2     4     6
NaN
1     9   11   13   15 NaN
2   18   20   22   24 NaN
3 NaN NaN NaN NaN NaN
```

Using the addmethod on df1, I pass df2and an argument to fill_value:

```
In [141]: df1.add(df2, fill_value=0)
Out[141]:
      a     b     c     d     e
0     0     2     4     6     4
1     9   11   13   15         9
2   18   20   22   24   14
3   15   16   17   18   19
```

Relatedly, when reindexing a Series or DataFrame, you can also specify a different fill value:

```
In [142]: df1.reindex(columns=df2.columns, fill_value=0)Out[142]:
    a  b      c     d  e
0  0  1      2     3  0
1  4  5      6     7  0
2  8  9   10   11   0
```

Table 5-7. Flexible arithmetic methods

Method	Description
add	Method for addition (+)
sub	Method for subtraction (-)
div	Method for division (/)
mul	Method for multiplication (*)

Operations between DataFrame and Series

As with NumPy arrays, arithmetic between DataFrame and Series is well-defined. First, as a motivating example, consider the difference between a 2D array and one of its rows:

```
In [143]: arr = np.arange(12.).reshape((3, 4))

In [144]: arr
Out[144]:
  array([[   0.,          1.,        2.,        3.],
             [   4.,         5.,        6.,        7.],
        [  8.,      9.,  10.,  11.]])

In [145]: arr[0]
Out[145]: array([ 0.,  1.,  2.,  3.])

In [146]: arr - arr[0]
Out[146]:
        array([[ 0.,  0.,  0.,  0.],
               [ 4.,  4.,  4.,  4.],
               [ 8.,  8.,  8.,  8.]])
```

This is referred to as *broadcasting* and is explained in more detail in Chapter 12. Operations between a DataFrame and a Series are similar:

```
In [147]: frame = DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'),
   .....:                          index=['Utah', 'Ohio', 'Texas', 'Oregon'])In [148]:

series = frame.ix[0]

In [149]: frame                 In [150]: series
Out[149]:                       Out[150]:
          b   d   e             b    0
Utah      0   1   2             d    1
Ohio      3   4   5             e    2
Texas     6   7   8             Name: Utah
Oregon  9  10  11
```

By default, arithmetic between DataFrame and Series matches the index of the Series on the DataFrame's columns, broadcasting down the rows:

```
In [151]: frame - series
Out[151]:
          b  d  e
 Utah     0  0  0
 Ohio     3  3  3
 Texas    6  6  6
  Oregon  9  9  9
```

If an index value is not found in either the DataFrame’s columns or the Series’s index, the objects will be reindexed to form the union:

```
In [152]: series2 = Series(range(3), index=['b', 'e', 'f'])

In [153]: frame + series2
Out[153]:
          b   d    e   f
Utah      0 NaN    3
NaN
Ohio      3 NaN    6 NaN
Texas     6 NaN    9 NaN
Oregon  9 NaN  12 NaN
```

If you want to instead broadcast over the columns, matching on the rows, you have to use one of the arithmetic methods. For example:

```
In [154]: series3 = frame['d']

In [155]: frame           In [156]: series3
```

```
  Out[155]:                Out[156]:
            b   d   e      Utah        1
Utah        0   1   2      Ohio        4
Ohio        3   4   5      Texas       7
Texas       6   7   8      Oregon     10
 Oregon     9  10  11      Name: d

In [157]: frame.sub(series3, axis=0)
Out[157]:
          b d e
Utah     -1  0  1
Ohio     -1  0  1
Texas   -1  0  1
Oregon -1  0  1
```

The axis number that you pass is the *axis to match on*. In this case we mean to match on the DataFrame’s row index and broadcast across.

Function application and mapping

NumPy ufuncs (element-wise array methods) work fine with pandas objects:

```
In [158]: frame = DataFrame(np.random.randn(4, 3), columns=list('bde'),
 .....:                            index=['Utah', 'Ohio', 'Texas', 'Oregon'])

In [159]: frame                                    In [160]: np.abs(frame)
Out[159]:                                          Out[160]:
                  b          d          e                      b          d          e
Utah     -0.204708   0.478943 -0.519439   Utah     0.204708   0.478943   0.519439
Ohio     -0.555730   1.965781   1.393406           Ohio     0.555730   1.965781   1.393406
Texas     0.092908   0.281746   0.769023           Texas    0.092908   0.281746   0.769023
Oregon   1.246435   1.007189 -1.296221             Oregon   1.246435   1.007189   1.296221
```

Another frequent operation is applying a function on 1D arrays to each column or row. DataFrame’s applymethod does exactly this:

```
In [161]: f = lambda x: x.max() - x.min()

In [162]: frame.apply(f)           In [163]: frame.apply(f, axis=1)
Out[162]:                          Out[163]:
b      1.802165                    Utah       0.998382
d      1.684034                    Ohio       2.521511
e      2.689627                    Texas      0.676115
                                   Oregon     2.542656
```

Many of the most common array statistics (like sumand mean) are DataFrame methods, so using applyis not necessary.

The function passed to applyneed not return a scalar value, it can also return a Series with multiple values:

```
In [164]: def f(x):
   .....:          return Series([x.min(), x.max()], index=['min', 'max'])In [165]:

frame.apply(f)
```

```
Out[165]:
            b          d          e
min -0.555730   0.281746 -1.296221
max  1.246435   1.965781  1.393406
```

Element-wise Python functions can be used, too. Suppose you wanted to compute a formatted string from each floating point value in frame. You can do this with applymap:

```
In [166]: format = lambda x: '%.2f' % x

In [167]: frame.applymap(format)
Out[167]:
            b     d      e
Utah    -0.20  0.48  -0.52
Ohio    -0.56  1.97   1.39
Texas    0.09  0.28   0.77
Oregon   1.25  1.01  -1.30
```

The reason for the name applymap is that Series has a map method for applying an element-wise function:

```
In [168]: frame['e'].map(format)
Out[168]:
Utah      -0.52
Ohio       1.39
Texas      0.77
Oregon    -1.30
Name: e
```

Sorting and ranking

Sorting a data set by some criterion is another important built-in operation. To sort lexicographically by row or column index, use the sort_index method, which returns a new, sorted object:

```
In [169]: obj = Series(range(4), index=['d', 'a', 'b', 'c'])

In [170]: obj.sort_index()
Out[170]:
a    1
b    2
c    3
d    0
```

With a DataFrame, you can sort by index on either axis:

```
In [171]: frame = DataFrame(np.arange(8).reshape((2, 4)), index=['three', 'one'],
   .....:                   columns=['d', 'a', 'b', 'c'])

In [172]: frame.sort_index()          In [173]: frame.sort_index(axis=1)
Out[172]:                             Out[173]:
       d  a  b  c                            a  b  c  d
one    4  5  6  7                     three  1  2  3  0
three  0  1  2  3                     one    5  6  7  4
```

The data is sorted in ascending order by default, but can be sorted in descending order, too:

```
In [174]: frame.sort_index(axis=1, ascending=False)Out[174]:
             d c b athree
0   3   2   1
one         4   7   6   5
```

To sort a Series by its values, use its order method:

```
In [175]: obj = Series([4, 7, -3, 2])

In [176]: obj.order()
Out[176]:
2     -3
3      2
0      4
1      7
```

Any missing values are sorted to the end of the Series by default:

```
In [177]: obj = Series([4, np.nan, 7, np.nan, -3, 2])

In [178]: obj.order()
Out[178]:
4       -3
5        2
0        4
2        7
1      NaN
3      NaN
```

On DataFrame, you may want to sort by the values in one or more columns. To do so, pass one or more column names to the by option:

```
In [179]: frame = DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 1]})

In [180]: frame                In [181]: frame.sort_index(by='b')
Out[180]:                      Out[181]:
      a   b                          a   b
  0   0   4                      2   0 -3
  1   1   7                      3   1   2
  2   0 -3                       0   0   4
  3   1   2                      1   1   7
```

To sort by multiple columns, pass a list of names:

```
In [182]: frame.sort_index(by=['a', 'b'])Out[182]:
      a   b
  2   0 -3
  0   0   4
  3   1   2
  1   1   7
```

Ranking is closely related to sorting, assigning ranks from one through the number of valid data points in an array. It is similar to the indirect sort indices produced by numpy.argsort, except that ties are broken according to a rule. The rank methods for Series and DataFrame are the place to look; by default rank breaks ties by assigning each group the mean rank:

```
In [183]: obj = Series([7, -5, 7, 4, 2, 0, 4])

In [184]: obj.rank()
Out[184]:
0      6.5
1      1.0
2      6.5
3      4.5
4      3.0
5      2.0
6      4.5
```

Ranks can also be assigned according to the order they're observed in the data:

```
In [185]: obj.rank(method='first')
Out[185]:
0      6
1      1
2      7
3      4
4      3
5      2
6      5
```

Naturally, you can rank in descending order, too:

```
In [186]: obj.rank(ascending=False, method='max')Out[186]:
0      2
1      7
2      2
3      4
4      5
5      6
6      4
```

See Table 5-8 for a list of tie-breaking methods available. DataFrame can compute ranks over the rows or the columns:

```
In [187]: frame = DataFrame({'b': [4.3, 7, -3, 2], 'a': [0, 1, 0, 1],
   .....:                              'c': [-2, 5, 8, -2.5]})

In [188]: frame                  In [189]: frame.rank(axis=1)
Out[188]:                        Out[189]:
     a    b     c                     a  b  c
  0  0   4.3 -2.0                  0  2  3  1
  1  1   7.0  5.0                  1  1  3  2
  2  0  -3.0  8.0                  2  2  1  3
  3  1   2.0 -2.5                  3  2  3  1
```

Table 5-8. Tie-breaking methods with rank

Method	Description
'average'	Default: assign the average rank to each entry in the equal group.
'min'	Use the minimum rank for the whole group.
'max'	Use the maximum rank for the whole group.
'first'	Assign ranks in the order the values appear in the data.

Axis indexes with duplicate values

Up until now all of the examples I've showed you have had unique axis labels (index values). While many pandas functions (like reindex) require that the labels be unique, it's not mandatory. Let's consider a small Series with duplicate indices:

```
In [190]: obj = Series(range(5), index=['a', 'a', 'b', 'b', 'c'])

In [191]: obj
Out[191]:
a    0
a    1
b    2
b    3
c    4
```

The index's is_uniqueproperty can tell you whether its values are unique or not:

```
In [192]: obj.index.is_unique
Out[192]: False
```

Data selection is one of the main things that behaves differently with duplicates. Indexing a value with multiple entries returns a Series while single entries return a scalar value:

```
In [193]: obj['a']          In [194]: obj['c']
Out[193]:                   Out[194]: 4
a    0
a    1
```

The same logic extends to indexing rows in a DataFrame:

```
In [195]: df = DataFrame(np.random.randn(4, 3), index=['a', 'a', 'b', 'b'])

In [196]: df
Out[196]:
          0         1         2
a  0.274992  0.228913  1.352917
a  0.886429 -2.001637 -0.371843
b  1.669025 -0.438570 -0.539741
b  0.476985  3.248944 -1.021228

In [197]: df.ix['b']
Out[197]:
          0         1         2
```

```
b   1.669025 -0.438570 -0.539741
b   0.476985   3.248944 -1.021228
```

Summarizing and Computing Descriptive Statistics

pandas objects are equipped with a set of common mathematical and statistical methods. Most of these fall into the category of *reductions* or *summary statistics*, methods that extract a single value (like the sum or mean) from a Series or a Series of values from the rows or columns of a DataFrame. Compared with the equivalent methods of vanilla NumPy arrays, they are all built from the ground up to exclude missing data. Consider a small DataFrame:

```
In [198]: df = DataFrame([[1.4, np.nan], [7.1, -4.5],
     .....:                                        [np.nan, np.nan], [0.75, -1.3]],
     .....:                                      index=['a', 'b', 'c', 'd'],
     .....:                                      columns=['one', 'two'])

In [199]: df
Out[199]:
       one   two
a   1.40   NaN
b   7.10 -4.5
c      NaN
NaNd   0.75 -
1.3
```

Calling DataFrame’s summethod returns a Series containing column sums:

```
In [200]: df.sum()
Out[200]:
one         9.25
two      -5.80
```

Passing axis=1 sums over the rows instead:

```
In [201]: df.sum(axis=1)
Out[201]:
a         1.40
b         2.60
c          NaN
d      -0.55
```

NA values are excluded unless the entire slice (row or column in this case) is NA. This can be disabled using the skipnaoption:

```
In [202]: df.mean(axis=1, skipna=False)
Out[202]:
a            NaN
b        1.300
c            NaN
d      -0.275
```

See Table 5-9 for a list of common options for each reduction method options.

Table 5-9. Options for reduction methods

Method	Description
axis	Axis to reduce over. 0 for DataFrame's rows and 1 for columns.
skipna	Exclude missing values, True by default.
level	Reduce grouped by level if the axis is hierarchically-indexed (MultiIndex).

Some methods, like idxmin and idxmax, return indirect statistics like the index value where the minimum or maximum values are attained:

```
In [203]: df.idxmax()
Out[203]:
one       b
two       d
```

Other methods are *accumulations*:

```
In [204]: df.cumsum()
Out[204]:
       one   two
a   1.40   NaN
b   8.50 -4.5
c      NaN
NaNd   9.25 -
5.8
```

Another type of method is neither a reduction nor an accumulation. describe is one such example, producing multiple summary statistics in one shot:

```
In [205]: df.describe()
Out[205]:
                  one           two
    count   3.000000   2.000000
   mean     3.083333 -2.900000
  std        3.493685   2.262742
min          0.750000 -4.500000
25%      1.075000 -3.700000
50%      1.400000 -2.900000
75%      4.250000 -2.100000
max      7.100000 -1.300000
```

On non-numeric data, describe produces alternate summary statistics:

```
In [206]: obj = Series(['a', 'a', 'b', 'c'] * 4)

In [207]: obj.describe()
Out[207]:
count          16
unique          3
top             a
freq            8
```

See Table 5-10 for a full list of summary statistics and related methods.

Table 5-10. Descriptive and summary statistics

Method	Description
count	Number of non-NA values
describe	Compute set of summary statistics for Series or each DataFrame column
min, max	Compute minimum and maximum values
argmin, argmax	Compute index locations (integers) at which minimum or maximum value obtained, respectively
idxmin, idxmax	Compute index values at which minimum or maximum value obtained, respectively
quantile	Compute sample quantile ranging from 0 to 1
sum	Sum of values
mean	Mean of values
median	Arithmetic median (50% quantile) of values
mad	Mean absolute deviation from mean value
var	Sample variance of values
std	Sample standard deviation of values
skew	Sample skewness (3rd moment) of values
kurt	Sample kurtosis (4th moment) of values
cumsum	Cumulative sum of values
cummin, cummax	Cumulative minimum or maximum of values, respectively
cumprod	Cumulative product of values
diff	Compute 1st arithmetic difference (useful for time series)
pct_change	Compute percent changes

Correlation and Covariance

Some summary statistics, like correlation and covariance, are computed from pairs of arguments. Let’s consider some DataFrames of stock prices and volumes obtained from Yahoo! Finance:

```
import pandas.io.data as web

all_data = {}
for ticker in ['AAPL', 'IBM', 'MSFT', 'GOOG']:
    all_data[ticker] = web.get_data_yahoo(ticker, '1/1/2000', '1/1/2010')

price = DataFrame({tic: data['Adj Close']
                    for tic, data in all_data.iteritems()})volume =
DataFrame({tic: data['Volume']
                    for tic, data in all_data.iteritems()})
```

I now compute percent changes of the prices:

```
In [209]: returns = price.pct_change()In [210]:

returns.tail()
```

```
Out[210]:
                  AAPL      GOOG       IBM      MSFT
Date
2009-12-24  0.034339  0.011117  0.004420  0.002747
2009-12-28  0.012294  0.007098  0.013282  0.005479
2009-12-29 -0.011861 -0.005571 -0.003474  0.006812
2009-12-30  0.012147  0.005376  0.005468 -0.013532
2009-12-31 -0.004300 -0.004416 -0.012609 -0.015432
```

The corr method of Series computes the correlation of the overlapping, non-NA, aligned-by-index values in two Series. Relatedly, covcomputes the covariance:

```
In [211]: returns.MSFT.corr(returns.IBM)
Out[211]: 0.49609291822168838

In [212]: returns.MSFT.cov(returns.IBM)
Out[212]: 0.00021600332437329015
```

DataFrame’s corr and cov methods, on the other hand, return a full correlation or covariance matrix as a DataFrame, respectively:

```
In [213]: returns.corr()
Out[213]:
          AAPL      GOOG       IBM      MSFT
AAPL  1.000000  0.470660  0.410648  0.424550
GOOG  0.470660  1.000000  0.390692  0.443334
 IBM  0.410648  0.390692  1.000000  0.496093
MSFT  0.424550  0.443334  0.496093  1.000000

In [214]: returns.cov()
Out[214]:
          AAPL                 IBM      MSFT
                     GOO
          G
AAPL  0.001028  0.000303  0.000252  0.000309
GOOG  0.000303  0.000580  0.000142  0.000205
 IBM  0.000252  0.000142  0.000367  0.000216
MSFT  0.000309  0.000205  0.000216  0.000516
```

Using DataFrame’s corrwithmethod, you can compute pairwise correlations betweena DataFrame’s columns or rows with another Series or DataFrame. Passing a Series returns a Series with the correlation value computed for each column:

```
In [215]: returns.corrwith(returns.IBM)
Out[215]:
AAPL    0.410648
GOOG    0.390692
IBM     1.000000
MSFT    0.496093
```

Passing a DataFrame computes the correlations of matching column names. Here I compute correlations of percent changes with volume:

```
In [216]: returns.corrwith(volume)
Out[216]:
AAPL   -0.057461
GOOG    0.062644
```

```
IBM     -0.007900
MSFT    -0.014175
```

Passing axis=1does things row-wise instead. In all cases, the data points are aligned by label before computing the correlation.

Unique Values, Value Counts, and Membership

Another class of related methods extracts information about the values contained in a one-dimensional Series. To illustrate these, consider this example:

```
In [217]: obj = Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c', 'c'])
```

The first function is unique, which gives you an array of the unique values in a Series:

```
In [218]: uniques = obj.unique()

In [219]: uniques
Out[219]: array([c, a, d, b], dtype=object)
```

The unique values are not necessarily returned in sorted order, but could be sorted after the fact if needed (uniques.sort()). Relatedly, value_counts computes a Series con- taining value frequencies:

```
In [220]: obj.value_counts()
Out[220]:
c      3
a      3
b      2
d      1
```

The Series is sorted by value in descending order as a convenience. value_counts is also available as a top-level pandas method that can be used with any array or sequence:

```
In [221]: pd.value_counts(obj.values, sort=False)Out[221]:
a      3
b      2
c      3
d      1
```

Lastly, isin is responsible for vectorized set membership and can be very useful in filtering a data set down to a subset of values in a Series or column in a DataFrame:

```
In [222]: mask = obj.isin(['b', 'c'])

In [223]: mask             In [224]: obj[mask]
Out[223]:                  Out[224]:
0      True                0     c
1      False               5     b
2      False               6     b
3      False               7     c
4      False               8     c
5      True
6      True
```

```
7       True
8       True
```

See Table 5-11 for a reference on these methods.

Table 5-11. Unique, value counts, and binning methods

Method	Description
isin	Compute boolean array indicating whether each Series value is contained in the passed sequence of values.
unique	Compute array of unique values in a Series, returned in the order observed.
value_counts	Return a Series containing unique values as its index and frequencies as its values, ordered count in descending order.

In some cases, you may want to compute a histogram on multiple related columns in a DataFrame. Here's an example:

```
In [225]: data = DataFrame({'Qu1': [1, 3, 4, 3, 4],
.....:                    'Qu2': [2, 3, 1, 2, 3],
.....:                    'Qu3': [1, 5, 2, 4, 4]})

In [226]: data
Out[226]:
     Qu1  Qu2
Qu30 1    2    1
1    3    3    5
2    4    1    2
3    3    2    4
4    4    3    4
```

Passing pandas.value_counts to this DataFrame's apply function gives:

```
In [227]: result = data.apply(pd.value_counts).fillna(0)

In [228]: result
Out[228]:
     Qu1  Qu2
Qu31 1    1    1
2    0    2    1
3    2    2    0
4    2    0    2
5    0    0    1
```

Handling Missing Data

Missing data is common in most data analysis applications. One of the goals in designing pandas was to make working with missing data as painless as possible. For example, all of the descriptive statistics on pandas objects exclude missing data as you've seen earlier in the chapter.

pandas uses the floating point value NaN (Not a Number) to represent missing data in both floating as well as in non-floating point arrays. It is just used as a *sentinel* that can be easily detected:

```
In [229]: string_data = Series(['aardvark', 'artichoke', np.nan, 'avocado'])

In [230]: string_data              In [231]: string_data.isnull()
Out[230]:                          Out[231]:
0      aardvark                    0     False
1     artichoke                    1     False
2           NaN                    2      True
3       avocado                    3     False
```

The built-in Python Nonevalue is also treated as NA in object arrays:

```
In [232]: string_data[0] = None

In [233]: string_data.isnull()
Out[233]:
0      True
1     False
2      True
3     False
```

I do not claim that pandas's NA representation is optimal, but it is simple and reasonably consistent. It's the best solution, with good all-around performance characteristics and a simple API, that I could concoct in the absence of a true NA data type or bit pattern in NumPy's data types. Ongoing development work in NumPy may change this in the future.

Table 5-12. NA handling methods

Argument	Description
dropna	Filter axis labels based on whether values for each label have missing data, with varying thresholds for how muchmissing data to tolerate.
fillna	Fill in missing data with some value or using an interpolation method such as 'ffill'or 'bfill'.isnull
	Return like-type object containing boolean values indicating which values are missing / NA.
notnull	Negation of isnull.

Filtering Out Missing Data

You have a number of options for filtering out missing data. While doing it by hand is always an option, dropnacan be very helpful. On a Series, it returns the Series with only the non-null data and index values:

```
In [234]: from numpy import nan as NA

In [235]: data = Series([1, NA, 3.5, NA, 7])

In [236]: data.dropna()
Out[236]:
```

```
0       1.0
2       3.5
4       7.0
```

Naturally, you could have computed this yourself by boolean indexing:

```
In [237]: data[data.notnull()]
Out[237]:
0       1.0
2       3.5
4       7.0
```

With DataFrame objects, these are a bit more complex. You may want to drop rows or columns which are all NA or just those containing any NAs. dropnaby default drops any row containing a missing value:

```
                    In [238]: data = DataFrame([[1., 6.5, 3.], [1., NA, NA],
               .....:                                   [NA, NA, NA], [NA, 6.5, 3.]])

In [239]: cleaned = data.dropna()

In [240]: data                  In [241]: cleaned
Out[240]:                       Out[241]:
      0     1    2                   0     1   2
  0    1  6.5     3               0  1  6.5  3
1     1  NaN NaN
2  NaN   NaN
NaN3 NaN   6.5
  3
```

Passing how='all' will only drop rows that are all NA:

```
In [242]: data.dropna(how='all')
Out[242]:
      0     1    2
0     1  6.5     3
1     1  NaN
NaN3 NaN   6.5
      3
```

Dropping columns in the same way is only a matter of passing axis=1:

```
In [243]: data[4] = NA

In [244]: data                      In [245]: data.dropna(axis=1, how='all')Out[244]:
                                    Out[245]:
      0     1    2    4                   0     1    2
0     1  6.5     3 NaN              0     1  6.5     3
1     1  NaN NaN NaN                1     1  NaN NaN
2  NaN   NaN NaN NaN                2 NaN   NaN
NaN3 NaN   6.5                      3 NaN            3
NaN  6.5         3
```

A related way to filter out DataFrame rows tends to concern time series data. Suppose you want to keep only rows containing a certain number of observations. You can indicate this with the threshargument:

```
In [246]: df = DataFrame(np.random.randn(7, 3))In [247]:

df.ix[:4, 1] = NA; df.ix[:2, 2] = NA
```

```
In [248]: df                          In [249]: df.dropna(thresh=3)
Out[248]:                             Out[249]:
          0          1          2               0          1          2
0 -0.577087        NaN        NaN     5  0.332883 -2.359419 -0.199543
1  0.523772        NaN        NaN     6 -1.541996 -0.970736 -1.307030
2 -0.713544        NaN        NaN
3 -1.860761        NaN   0.560145
4 -1.265934        NaN  -1.063512
5  0.332883 -2.359419 -0.199543
6 -1.541996 -0.970736 -1.307030
```

Filling in Missing Data

Rather than filtering out missing data (and potentially discarding other data along with it), you may want to fill in the "holes" in any number of ways. For most purposes, the fillna method is the workhorse function to use. Calling fillna with a constant replaces missing values with that value:

```
In [250]: df.fillna(0)
Out[250]:
          0          1          2
0 -0.577087   0.000000   0.000000
1  0.523772   0.000000   0.000000
2 -0.713544   0.000000   0.000000
3 -1.860761   0.000000   0.560145
4 -1.265934   0.000000  -1.063512
5  0.332883  -2.359419  -0.199543
6 -1.541996  -0.970736  -1.307030
```

Calling fillna with a dict you can use a different fill value for each column:

```
In [251]: df.fillna({1: 0.5, 3: -1})Out[251]:
          0          1          2
0 -0.577087   0.500000        NaN
1  0.523772   0.500000        NaN
2 -0.713544   0.500000        NaN
3 -1.860761   0.500000   0.560145
4 -1.265934   0.500000  -1.063512
5  0.332883  -2.359419  -0.199543
6 -1.541996  -0.970736  -1.307030
```

fillnareturns a new object, but you can modify the existing object in place:

```
# always returns a reference to the filled objectIn [252]: _ =
df.fillna(0, inplace=True)

In [253]: df
Out[253]:
          0          1          2
0 -0.577087   0.000000   0.000000
1  0.523772   0.000000   0.000000
2 -0.713544   0.000000   0.000000
3 -1.860761   0.000000   0.560145
```

```
4 -1.265934   0.000000 -1.063512
5   0.332883 -2.359419 -0.199543
6 -1.541996 -0.970736 -1.307030
```

The same interpolation methods available for reindexing can be used with fillna:

```
In [254]: df = DataFrame(np.random.randn(6, 3))In [255]:

df.ix[2:, 1] = NA; df.ix[4:, 2] = NA

In [256]: df
Out[256]:
           0          1          2
0   0.286350   0.377984 -0.753887
1   0.331286   1.349742   0.069877
2   0.246674        NaN   1.004812
3   1.327195        NaN  -1.549106
4   0.022185        NaN        NaN
5   0.862580        NaN        NaN
```

```
In [257]: df.fillna(method='ffill')

           0          1          2
0   0.286350   0.377984 -0.753887
1   0.331286   1.349742  0.069877
2   0.246674   1.349742  1.004812
3   1.327195   1.349742 -1.549106
4   0.022185   1.349742 -1.549106
5   0.862580   1.349742 -1.549106
```

```
In [258]: df.fillna(method='ffill', limit=2)Out[257]:
Out[258]:
           0          1          2
0   0.286350   0.377984 -0.753887
1   0.331286   1.349742  0.069877
2   0.246674   1.349742  1.004812
3   1.327195   1.349742 -1.549106
4   0.022185        NaN -1.549106
5   0.862580        NaN -1.549106
```

With fillnayou can do lots of other things with a little creativity. For example, you might pass the mean or median value of a Series:

```
In [259]: data = Series([1., NA, 3.5, NA, 7])

In [260]: data.fillna(data.mean())
Out[260]:
0       1.000000
1       3.833333
2       3.500000
3       3.833333
4       7.000000
```

See Table 5-13 for a reference on fillna.

Table 5-13. fillna function arguments

Argument	Description
value	Scalar value or dict-like object to use to fill missing values
method	Interpolation, by default 'ffill' if function called with no other arguments
axis	Axis to fill on, default axis=0
inplace	Modify the calling object without producing a copy
limit	For forward and backward filling, maximum number of consecutive periods to fill

Unit-III:

Data Loading, Storage, and File Formats : Reading and Writing Data in Text Format, Reading Text Files in Pieces, Writing Data Out to Text Format, Manually Working with Delimited Formats, JSON Data, XML and HTML: Web Scraping, Binary Data Formats,Using HDF5 Format, Reading Microsoft Excel Files, Interacting with Databases, Storing and Loading Data in MongoDB

Data Loading, Storage, and File Formats

The tools in this book are of little use if you can't easily import and export data in Python. I'm going to be focused on input and output with pandas objects, though there are of course numerous tools in other libraries to aid in this process. NumPy, for example, features low-level but extremely fast binary data loading and storage, including support for memory-mapped array. See Chapter 12 for more on those.

Input and output typically falls into a few main categories: reading text files and other more efficient on-disk formats, loading data from databases, and interacting with network sources like web APIs.

Reading and Writing Data in Text Format

Python has become a beloved language for text and file munging due to its simple syntax for interacting with files, intuitive data structures, and convenient features like tuple packing and unpacking.

pandas features a number of functions for reading tabular data as a DataFrame object. Table 6-1 has a summary of all of them, though read_csv and read_table are likely the ones you'll use the most.

Table 6-1. Parsing functions in pandas

Function	Description
read_csv	Load delimited data from a file, URL, or file-like object. Use comma as default delimiter read_table
	Load delimited data from a file, URL, or file-like object. Use tab ('\t') as default delimiterread_fwf
	Read data in fixed-width column format (that is, no delimiters)
read_clipboard	Version ofread_tablethatreadsdatafromtheclipboard. Useful for converting tables fromwebpages

I’ll give an overview of the mechanics of these functions, which are meant to convert text data into a DataFrame. The options for these functions fall into a few categories:

- Indexing: can treat one or more columns as the returned DataFrame, and whether to get column names from the file, the user, or not at all.
- Type inference and data conversion: this includes the user-defined value conversions and custom list of missing value markers.
- Datetime parsing: includes combining capability, including combining date and time information spread over multiple columns into a single column in the result.
- Iterating: support for iterating over chunks of very large files.
- Unclean data issues: skipping rows or a footer, comments, or other minor things like numeric data with thousands separated by commas.

Type inference is one of the more important features of these functions; that means you don’t have to specify which columns are numeric, integer, boolean, or string. Handling dates and other custom types requires a bit more effort, though. Let’s start with a small comma-separated (CSV) text file:

```
In [846]: !cat ch06/ex1.csv
a,b,c,d,message 1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

Since this is comma-delimited, we can use read_csvto read it into a DataFrame:

```
In [847]: df = pd.read_csv('ch06/ex1.csv')

In [848]: df
Out[848]:
    a    b    c     d message
      0   1     2  3  4     hello
    1          5     6     7     8
           world2   9   10   11
              12                foo
```

We could also have used read_table and specifying the delimiter:

```
In [849]: pd.read_table('ch06/ex1.csv', sep=',')Out[849]:
    a    b    c     d message
      0   1     2  3  4     hello
    1          5     6     7     8
           world2   9   10   11
              12                foo
```

Here I used the Unix cat shell command to print the raw contents of the file to the screen. If you’re on Windows, you can use type instead of catto achieve the same effect.

A file will not always have a header row. Consider this file:

```
In [850]: !cat ch06/ex2.csv
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

To read this in, you have a couple of options. You can allow pandas to assign default column names, or you can specify names yourself:

```
In [851]: pd.read_csv('ch06/ex2.csv', header=None)Out[851]:
    X.1   X.2   X.3   X.4     X.5
0     1     2     3     4   hello
1     5     6     7     8   world
2     9    10    11    12     foo

In [852]: pd.read_csv('ch06/ex2.csv', names=['a', 'b', 'c', 'd', 'message'])Out[852]:
     a     b     c     d message
      0   1     2  3  4     hello
     1          5     6     7     8
            world2   9   10   11
              12               foo
```

Suppose you wanted the message column to be the index of the returned DataFrame. You can either indicate you want the column at index 4 or named 'message' using the index_colargument:

```
In [853]: names = ['a', 'b', 'c', 'd', 'message']

In [854]: pd.read_csv('ch06/ex2.csv', names=names, index_col='message')Out[854]:
           a    b    c    d
message
hello      1    2    3    4
world      5    6    7    8
foo        9   10   11   12
```

In the event that you want to form a hierarchical index from multiple columns, just pass a list of column numbers or names:

```
In [855]: !cat ch06/csv_mindex.csv
key1,key2,value1,value2
one,a,1,2
one,b,3,4
one,c,5,6
one,d,7,8
two,a,9,10
two,b,11,12
two,c,13,14
two,d,15,16

In [856]: parsed = pd.read_csv('ch06/csv_mindex.csv', index_col=['key1', 'key2'])In [857]: parsed
Out[857]:
```

```
                    value1    value2
key1 key2
one   a                1         2
      b                3         4
      c                5         6
      d                7         8
two   a                9        10
      b               11        12
      c               13        14
      d               15        16
```

In some cases, a table might not have a fixed delimiter, using whitespace or some other pattern to separate fields. In these cases, you can pass a regular expression as a delimiter for read_table. Consider a text file that looks like this:

```
In [858]: list(open('ch06/ex3.txt'))Out[858]:
['                  A              B              C\n',
  'aaa -0.264438 -1.026059 -0.619500\n',
  'bbb   0.927272   0.302904 -0.032399\n',
  'ccc -0.264273 -0.386314 -0.217601\n',
  'ddd -0.871858 -0.348382   1.100491\n']
```

While you could do some munging by hand, in this case fields are separated by a variable amount of whitespace. This can be expressed by the regular expression \s+, so we have then:

```
In [859]: result = pd.read_table('ch06/ex3.txt', sep='\s+')

In [860]: result
Out[860]:
                  A              B
                                 Caaa -
0.264438 -1.026059 -0.619500
bbb   0.927272   0.302904 -0.032399
ccc -0.264273 -0.386314 -0.217601
ddd -0.871858 -0.348382   1.100491
```

Because there was one fewer column name than the number of data rows, read_table infers that the first column should be the DataFrame's index in this special case.

The parser functions have many additional arguments to help you handle the wide variety of exception file formats that occur (see Table 6-2). For example, you can skip the first, third, and fourth rows of a file with skiprows:

```
In [861]: !cat ch06/ex4.csv#
hey!
a,b,c,d,message
# just wanted to make things more difficult for you#
who reads CSV files with computers, anyway?
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
In [862]: pd.read_csv('ch06/ex4.csv', skiprows=[0, 2, 3])
Out[862]:
    a    b    c    d message
```

```
0   1     2     3     4      hello
1   5     6     7     8      world
2   9   10   11   12         foo
```

Handling missing values is an important and frequently nuanced part of the file parsing process. Missing data is usually either not present (empty string) or marked by some *sentinel* value. By default, pandas uses a set of commonly occurring sentinels, such as NA, -1.#IND, and NULL:

```
In [863]: !cat ch06/ex5.csv
something,a,b,c,d,message
one,1,2,3,4,NA
two,5,6,,8,world
three,9,10,11,12,foo
In [864]: result = pd.read_csv('ch06/ex5.csv')

In [865]: result
Out[865]:
   something  a   b   c   d message
0        one  1   2   3   4     NaN
1        two  5   6 NaN   8   world
2      three  9  10  11  12     foo

In [866]: pd.isnull(result)
Out[866]:
   something      a      b      c      d message
0      False  False  False  False  False    True
1      False  False  False   True  False   False
2      False  False  False  False  False   False
```

The na_valuesoption can take either a list or set of strings to consider missing values:

```
In [867]: result = pd.read_csv('ch06/ex5.csv', na_values=['NULL'])

In [868]: result
Out[868]:
   something  a   b   c   d message
0        one  1   2   3   4     NaN
1        two  5   6 NaN   8   world
2      three  9  10  11  12     foo
```

Different NA sentinels can be specified for each column in a dict:

```
In [869]: sentinels = {'message': ['foo', 'NA'], 'something': ['two']}

In [870]: pd.read_csv('ch06/ex5.csv', na_values=sentinels)
Out[870]:
   something  a   b   c   d message
0        one  1   2   3   4     NaN
1        NaN  5   6 NaN   8   world
2      three  9  10  11  12     NaN
```

Table 6-2. read_csv /read_table function arguments

Argument	Description
path	String indicating filesystem location, URL, or file-like object
sepor delimiter	Character sequence or regular expression to use to split fields in each row
header	Row number to use as column names. Defaults to 0 (first row), but should be Noneif there is no headerrow
index_col	Column numbers or names to use as the row index in the result. Can be a single name/number or a listof them for a hierarchical index
names	List of column names for result, combine with header=None
skiprows	Number of rows at beginning of file to ignore or list of row numbers (starting from 0) to skip
na_values	Sequence of values to replace with NA
comment	Character or characters to split comments off the end of lines
parse_dates	Attempt to parse data to datetime; False by default. If True, will attempt to parse all columns. Otherwisecan specify a list of column numbers or name to parse. If element of list is tuple or list, will combine multiple columns together and parse to date (for example if date/time split across two columns)
keep_date_col	If joining columns to parse date, drop the joined columns. Default True
converters	Dict containing column number of name mapping to functions. For example {'foo': f}would applythe function fto all values in the 'foo'column
dayfirst	When parsing potentially ambiguous dates, treat as international format (e.g. 7/6/2012 -> June 7, 2012). Default False
date_parser	Function to use to parse dates
nrows	Number of rows to read from beginning of file iterator
	Return a TextParser object for reading file piecemeal
chunksize	For iteration, size of file chunks
skip_footer	Number of lines to ignore at end of file
verbose	Print various parser output information, like the number of missing values placed in non-numeric columns
encoding	Text encoding for unicode. For example 'utf-8'for UTF-8 encoded text
squeeze	If the parsed data only contains one column return a Series
thousands	Separator for thousands, e.g. ','or '.'

Reading Text Files in Pieces

When processing very large files or figuring out the right set of arguments to correctly process a large file, you may only want to read in a small piece of a file or iterate through smaller chunks of the file.

```
In [871]: result = pd.read_csv('ch06/ex6.csv')

In [872]: result
Out[872]:
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 10000 entries, 0 to 9999 Data
columns:
one      10000   non-null  values  two
10000   non-null  values  three  10000
non-null  values  four   10000   non-
null values key 10000 non-null values
dtypes: float64(4), object(1)
```

If you want to only read out a small number of rows (avoiding reading the entire file), specify that with nrows:

```
In [873]: pd.read_csv('ch06/ex6.csv', nrows=5)Out[873]:
            one           two         three          four key0
0.467976 -0.038649 -0.295344 -1.824726       L
1 -0.358893   1.404453   0.704965 -0.200638            B
2 -0.501840   0.659254 -0.421691 -0.057688            G
3   0.204886   1.074134   1.388361 -0.982404            R
4   0.354628 -0.133116   0.283763 -0.837063            Q
```

To read out a file in pieces, specify a chunksize as a number of rows:

```
In [874]: chunker = pd.read_csv('ch06/ex6.csv', chunksize=1000)

In [875]: chunker
Out[875]: <pandas.io.parsers.TextParser at 0x8398150>
```

The TextParser object returned by read_csv allows you to iterate over the parts of the file according to the chunksize. For example, we can iterate over ex6.csv, aggregating the value counts in the 'key' column like so:

```
chunker = pd.read_csv('ch06/ex6.csv', chunksize=1000)

tot = Series([])
for piece in chunker:
    tot = tot.add(piece['key'].value_counts(), fill_value=0)tot =

tot.order(ascending=False)
```

We have then:

```
In [877]: tot[:10]
Out[877]:
E     368
X     364
L     346
O     343
Q     340
M     338
J     337
F     335
K     334
H     330
```

TextParseris also equipped with a get_chunkmethod which enables you to read piecesof an arbitrary size.

Writing Data Out to Text Format

Data can also be exported to delimited format. Let's consider one of the CSV files read above:

```
In [878]: data = pd.read_csv('ch06/ex5.csv')

In [879]: data
Out[879]:
   something  a   b    c   d message
0        one  1   2    3   4     NaN
1        two  5   6  NaN   8   world
2      three  9  10   11  12     foo
```

Using DataFrame's to_csvmethod, we can write the data out to a comma-separated file:

```
In [880]: data.to_csv('ch06/out.csv')

In [881]: !cat ch06/out.csv
,something,a,b,c,d,message
0,one,1,2,3.0,4,
1,two,5,6,,8,world
2,three,9,10,11.0,12,foo
```

Other delimiters can be used, of course (writing to sys.stdoutso it just prints the text result):

```
In [882]: data.to_csv(sys.stdout, sep='|')
|something|a|b|c|d|message
0|one|1|2|3.0|4|
1|two|5|6||8|world
2|three|9|10|11.0|12|foo
```

Missing values appear as empty strings in the output. You might want to denote them by some other sentinel value:

```
In [883]: data.to_csv(sys.stdout, na_rep='NULL')
,something,a,b,c,d,message
0,one,1,2,3.0,4,NULL
1,two,5,6,NULL,8,world
2,three,9,10,11.0,12,foo
```

With no other options specified, both the row and column labels are written. Both of these can be disabled:

```
In [884]: data.to_csv(sys.stdout, index=False, header=False)one,1,2,3.0,4,
two,5,6,,8,world
three,9,10,11.0,12,foo
```

You can also write only a subset of the columns, and in an order of your choosing:

```
In [885]: data.to_csv(sys.stdout, index=False, cols=['a', 'b', 'c'])a,b,c
1,2,3.0
5,6,
9,10,11.0
```

Series also has a to_csv method:

```
In [886]: dates = pd.date_range('1/1/2000', periods=7)In [887]: ts =

Series(np.arange(7), index=dates)

In [888]: ts.to_csv('ch06/tseries.csv')

In [889]: !cat ch06/tseries.csv2000-01-
01 00:00:00,0
2000-01-02 00:00:00,1
2000-01-03 00:00:00,2
2000-01-04 00:00:00,3
2000-01-05 00:00:00,4
2000-01-06 00:00:00,5
2000-01-07 00:00:00,6
```

With a bit of wrangling (no header, first column as index), you can read a CSV version of a Series with read_csv, but there is also a from_csvconvenience method that makesit a bit simpler:

```
In [890]: Series.from_csv('ch06/tseries.csv', parse_dates=True)Out[890]:
2000-01-01     0
2000-01-02     1
2000-01-03     2
2000-01-04     3
2000-01-05     4
2000-01-06     5
2000-01-07     6
```

See the docstrings for to_csvand from_csvin IPython for more information.

Manually Working with Delimited Formats

Most forms of tabular data can be loaded from disk using functions like pan das.read_table. In some cases, however, some manual processing may be necessary. It's not uncommon to receive a file with one or more malformed lines that trip up read_table. To illustrate the basic tools, consider a small CSV file:

```
In [891]: !cat ch06/ex7.csv
"a","b","c"
"1","2","3"
"1","2","3","4"
```

For any file with a single-character delimiter, you can use Python's built-in csvmodule. To use it, pass any open file or file-like object to csv.reader:

```
import csv
f = open('ch06/ex7.csv')reader

= csv.reader(f)
```

Iterating through the reader like a file yields tuples of values in each like with any quote characters removed:

```
In [893]: for line in reader:
   .....:          print line
['a', 'b', 'c']
['1', '2', '3']
['1', '2', '3', '4']
```

From there, it's up to you to do the wrangling necessary to put the data in the form that you need it. For example:

```
In [894]: lines = list(csv.reader(open('ch06/ex7.csv')))In [895]:

header, values = lines[0], lines[1:]

In [896]: data_dict = {h: v for h, v in zip(header, zip(*values))}

In [897]: data_dict
Out[897]: {'a': ('1', '1'), 'b': ('2', '2'), 'c': ('3', '3')}
```

CSV files come in many different flavors. Defining a new format with a different delimiter, string quoting convention, or line terminator is done by defining a simple subclass of csv.Dialect:

```
class my_dialect(csv.Dialect):
    lineterminator = '\n' delimiter =
    ';'
    quotechar = '"'

reader = csv.reader(f, dialect=my_dialect)
```

Individual CSV dialect parameters can also be given as keywords to csv.reader without having to define a subclass:

```
reader = csv.reader(f, delimiter='|')
```

The possible options (attributes of csv.Dialect) and what they do can be found in Table 6-3.

Table 6-3. CSV dialect options

Argument	Description
delimiter	One-character string to separate fields. Defaults to ','.
lineterminator	Line terminator for writing, defaults to '\r\n'. Reader ignores this and recognizes cross-platform line terminators.
quotechar	Quote character for fields with special characters (like a delimiter). Default is '"'.
quoting	Quoting convention. Options include csv.QUOTE_ALL(quote all fields), csv.QUOTE_MINIMAL(only fields with special characters like the delimiter),

Argument	Description
	csv.QUOTE_NONNUMERIC, and csv.QUOTE_NON(no quoting). See Python's documentation for full details. Defaults to QUOTE_MINIMAL.
skipinitialspace	Ignore whitespace after each delimiter. Default False.
doublequote	How to handle quoting character inside a field. If True, it is doubled. See online documentation for full detail and behavior.
escapechar	String to escape the delimiter if quotingis set to csv.QUOTE_NONE. Disabled by default

For files with more complicated or fixed multicharacter delimiters, you will not be able to use the csv module. In those cases, you'll have to do the line splitting and other cleanup using string's split method or the regular expression method re.split.

To *write* delimited files manually, you can use csv.writer. It accepts an open, writable file object and the same dialect and format options as csv.reader:

```
with open('mydata.csv', 'w') as f:
    writer = csv.writer(f, dialect=my_dialect)
    writer.writerow(('one', 'two', 'three'))
    writer.writerow(('1', '2', '3'))
    writer.writerow(('4', '5', '6'))
    writer.writerow(('7', '8', '9'))
```

JSON Data

JSON (short for JavaScript Object Notation) has become one of the standard formats for sending data by HTTP request between web browsers and other applications. It is a much more flexible data format than a tabular text form like CSV. Here is an example:

```
obj = """
{"name": "Wes",
  "places_lived": ["United States", "Spain", "Germany"],"pet": null,
               "siblings": [{"name": "Scott", "age": 25, "pet": "Zuko"},
                            {"name": "Katie", "age": 33, "pet": "Cisco"}]
}
"""
```

JSON is very nearly valid Python code with the exception of its null value null and some other nuances (such as disallowing trailing commas at the end of lists). The basic types are objects (dicts), arrays (lists), strings, numbers, booleans, and nulls. All of the keys in an object must be strings. There are several Python libraries for reading and writing JSON data. I'll use json here as it is built into the Python standard library. To convert a JSON string to Python form, use json.loads:

```
In [899]: import json
In [900]: result = json.loads(obj)

In [901]: result
Out[901]:
{u'name': u'Wes',
 u'pet': None,
 u'places_lived': [u'United States', u'Spain', u'Germany'], u'siblings': [{u'age': 25,
 u'name': u'Scott', u'pet': u'Zuko'},
   {u'age': 33, u'name': u'Katie', u'pet': u'Cisco'}]}
```

json.dumps on the other hand converts a Python object back to JSON:

```
In [902]: asjson = json.dumps(result)
```

How you convert a JSON object or list of objects to a DataFrame or some other data structure for analysis will be up to you. Conveniently, you can pass a list of JSON objects to the DataFrame constructor and select a subset of the data fields:

```
In [903]: siblings = DataFrame(result['siblings'], columns=['name', 'age'])

In [904]: siblings
Out[904]:
      name   age
  0   Scott   25
  1   Katie   33
```

For an extended example of reading and manipulating JSON data (including nested records), see the USDA Food Database example in the next chapter.

An effort is underway to add fast native JSON export (to_json) and decoding (from_json) to pandas. This was not ready at the time of writing.

XML and HTML: Web Scraping

Python has many libraries for reading and writing data in the ubiquitous HTML and XML formats. lxml (*http://lxml.de*) is one that has consistently strong performance in parsing very large files. lxml has multiple programmer interfaces; first I'll show using lxml.html for HTML, then parse some XML using lxml.objectify.

Many websites make data available in HTML tables for viewing in a browser, but not downloadable as an easily machine-readable format like JSON, HTML, or XML. I noticed that this was the case with Yahoo! Finance's stock options data. If you aren't familiar with this data; options are derivative contracts giving you the right to buy (*call* option) or sell (*put* option) a company's stock at some particular price (the *strike*) between now and some fixed point in the future (the *expiry*). People trade both call and put options across many strikes and expiries; this data can all be found together in tables on Yahoo! Finance.

To get started, find the URL you want to extract data from, open it with urllib2 and parse the stream with lxml like so:

```
from lxml.html import parsefrom
urllib2 import urlopen

parsed = parse(urlopen('http://finance.yahoo.com/q/op?s=AAPL+Options'))doc =

parsed.getroot()
```

Using this object, you can extract all HTML tags of a particular type, such as tabletags containing the data of interest. As a simple motivating example, suppose you wanted to get a list of every URL linked to in the document; links are atags in HTML. Using the document root's findall method along with an XPath (a means of expressing "queries" on the document):

```
In [906]: links = doc.findall('.//a')

In [907]: links[15:20]
Out[907]:
[<Element a at 0x6c488f0>,
  <Element a at 0x6c48950>,
  <Element a at 0x6c489b0>,
  <Element a at 0x6c48a10>,
  <Element a at 0x6c48a70>]
```

But these are objects representing HTML elements; to get the URL and link text you have to use each element's get method (for the URL) and text_content method (forthe display text):

```
In [908]: lnk = links[28]

In [909]: lnk
Out[909]: <Element a at 0x6c48dd0>

In [910]: lnk.get('href')
Out[910]: 'http://biz.yahoo.com/special.html'

In [911]: lnk.text_content()
Out[911]: 'Special Editions'
```

Thus, getting a list of all URLs in the document is a matter of writing this list comprehension:

```
In [912]: urls = [lnk.get('href') for lnk in doc.findall('.//a')]

In [913]: urls[-10:]
Out[913]:
['http://info.yahoo.com/privacy/us/yahoo/finance/details.html',
  'http://info.yahoo.com/relevantads/', 'http://docs.yahoo.com/info/terms/',
  'http://docs.yahoo.com/info/copyright/copyright.html',
  'http://help.yahoo.com/l/us/yahoo/finance/forms_index.html',
  'http://help.yahoo.com/l/us/yahoo/finance/quotes/fitadelay.html',
  'http://help.yahoo.com/l/us/yahoo/finance/quotes/fitadelay.html',
  'http://www.capitaliq.com',
  'http://www.csidata.com',
  'http://www.morningstar.com/']
```

Now, finding the right tables in the document can be a matter of trial and error; some websites make it easier by giving a table of interest an id attribute. I determined that these were the two tables containing the call data and put data, respectively:

```
tables = doc.findall('.//table')calls =
tables[9]
puts = tables[13]
```

Each table has a header row followed by each of the data rows:

```
In [915]: rows = calls.findall('.//tr')
```

For the header as well as the data rows, we want to extract the text from each cell; in the case of the header these are thcells and tdcells for the data:

```
def _unpack(row, kind='td'):
    elts = row.findall('.//%s' % kind)
    return [val.text_content() for val in elts]
```

Thus, we obtain:

```
In [917]: _unpack(rows[0], kind='th')
Out[917]: ['Strike', 'Symbol', 'Last', 'Chg', 'Bid', 'Ask', 'Vol', 'Open Int']

In [918]: _unpack(rows[1], kind='td')
Out[918]:
['295.00',
  'AAPL120818C00295000',
  '310.40',
  ' 0.00',
  '289.80',
  '290.80',
  '1',
  '169']
```

Now, it's a matter of combining all of these steps together to convert this data into a DataFrame. Since the numerical data is still in string format, we want to convert some, but perhaps not all of the columns to floating point format. You could do this by hand, but, luckily, pandas has a class TextParser that is used internally in the read_csv and other parsing functions to do the appropriate automatic type conversion:

```
from pandas.io.parsers import TextParser

def parse_options_data(table): rows =
    table.findall('.//tr')
    header = _unpack(rows[0], kind='th') data =
    [_unpack(r) for r in rows[1:]]
    return TextParser(data, names=header).get_chunk()
```

Finally, we invoke this parsing function on the lxml table objects and get DataFrame results:

```
In [920]: call_data = parse_options_data(calls)In [921]:

put_data = parse_options_data(puts)

In [922]: call_data[:10]
Out[922]:
   Strike                Symbol    Last  Chg     Bid     Ask  Vol Open Int
0     295  AAPL120818C002950  310.40  0.0  289.80  290.80    1      169
                         00
1     300  AAPL120818C003000  277.10  1.7  284.80  285.60    2      478
                         00
2     305  AAPL120818C003050  300.97  0.0  279.80  280.80   10      316
                         00
3     310  AAPL120818C003100  267.05  0.0  274.80  275.65    6      239
                         00
4     315  AAPL120818C003150  296.54  0.0  269.80  270.80   22       88
                         00
5     320  AAPL120818C003200  291.63  0.0  264.80  265.80   96      173
                         00
6     325  AAPL120818C003250  261.34  0.0  259.80  260.80  N/A      108
                         00
7     330  AAPL120818C003300  230.25  0.0  254.80  255.80  N/A       21
                         00
8     335  AAPL120818C003350  266.03  0.0  249.80  250.65    4       46
                         00
9     340  AAPL120818C003400  272.58  0.0  244.80  245.80    4       30
                         00
```

Parsing XML with lxml.objectify

XML (extensible markup language) is another common structured data format supporting hierarchical, nested data with metadata. The files that generate the book you are reading actually form a series of large XML documents.

Above, I showed the lxml library and its lxml.html interface. Here I show an alternate interface that's convenient for XML data, lxml.objectify.

The New York Metropolitan Transportation Authority (MTA) publishes a number of data series about its bus and train services (*http://www.mta.info/developers/download.html*). Here we'll look at the performance data which is contained in a set of XML files. Each train or bus service has a different file (like Performance_MNR.xml for the Metro-North Railroad) containing monthly data as a series of XML records that look like this:

```
<INDICATOR>
  <INDICATOR_SEQ>373889</INDICATOR_SEQ>
  <PARENT_SEQ></PARENT_SEQ>
  <AGENCY_NAME>Metro-North Railroad</AGENCY_NAME>
  <INDICATOR_NAME>Escalator Availability</INDICATOR_NAME>
  <DESCRIPTION>Percent of the time that escalators are operational
  systemwide. The availability rate is based on physical observations performed the morning of
  regular business days only. This is a new indicator the agency began reporting in
  2009.</DESCRIPTION>
  <PERIOD_YEAR>2011</PERIOD_YEAR>
  <PERIOD_MONTH>12</PERIOD_MONTH>
  <CATEGORY>Service Indicators</CATEGORY>
  <FREQUENCY>M</FREQUENCY>
  <DESIRED_CHANGE>U</DESIRED_CHANGE>
  <INDICATOR_UNIT>%</INDICATOR_UNIT>
```

```
    <DECIMAL_PLACES>1</DECIMAL_PLACES>
    <YTD_TARGET>97.00</YTD_TARGET>
    <YTD_ACTUAL></YTD_ACTUAL>
    <MONTHLY_TARGET>97.00</MONTHLY_TARGET>
    <MONTHLY_ACTUAL></MONTHLY_ACTUAL>
  </INDICATOR>
```

Using lxml.objectify, we parse the file and get a reference to the root node of the XML file with getroot:

```
from lxml import objectify

path = 'Performance_MNR.xml'
parsed = objectify.parse(open(path))root =
parsed.getroot()
```

root.INDICATOR return a generator yielding each <INDICATOR> XML element. For each record, we can populate a dict of tag names (like YTD_ACTUAL) to data values (excludinga few tags):

```
data = []

skip_fields = ['PARENT_SEQ', 'INDICATOR_SEQ',
               'DESIRED_CHANGE', 'DECIMAL_PLACES']

for elt in root.INDICATOR:
    el_data = {}
    for child in elt.getchildren(): if child.tag
        in skip_fields:
            continue
        el_data[child.tag] = child.pyval
    data.append(el_data)
```

Lastly, convert this list of dicts into a DataFrame:

```
In [927]: perf = DataFrame(data)

In [928]: perf
Out[928]:
Empty DataFrame
Columns: array([], dtype=int64)Index:
array([], dtype=int64)
```

XML data can get much more complicated than this example. Each tag can have metadata, too. Consider an HTML link tag which is also valid XML:

```
from StringIO import StringIO
tag = '<a href="http://www.google.com">Google</a>'root =

objectify.parse(StringIO(tag)).getroot()
```

You can now access any of the fields (like href) in the tag or the link text:

```
In [930]: root
Out[930]: <Element a at 0x88bd4b0>

In [931]: root.get('href') Out[931]:
'http://www.google.com'

In [932]: root.text
Out[932]: 'Google'
```

Binary Data Formats

One of the easiest ways to store data efficiently in binary format is using Python's built-in pickle serialization. Conveniently, pandas objects all have a save method which writes the data to disk as a pickle:

```
In [933]: frame = pd.read_csv('ch06/ex1.csv')

In [934]: frame
Out[934]:
     a    b    c    d message
       0   1     2  3  4     hello
     1          5     6     7     8
            world2   9   10   11
              12                 foo

In [935]: frame.save('ch06/frame_pickle')
```

You read the data back into Python with pandas.load, another pickle convenience function:

```
In [936]: pd.load('ch06/frame_pickle')
Out[936]:
     a    b    c    d message
       0   1     2  3  4     hello
     1          5     6     7     8
            world2   9   10   11
              12                 foo
```

> pickle is only recommended as a short-term storage format. The problem is that it is hard to guarantee that the format will be stable over time; an object pickled today may not unpickle with a later version of a library. I have made every effort to ensure that this does not occur with pandas, but at some point in the future it may be necessary to "break" the pickle format.

Using HDF5 Format

There are a number of tools that facilitate efficiently reading and writing large amounts of scientific data in binary format on disk. A popular industry-grade library for this is HDF5, which is a C library with interfaces in many other languages like Java, Python, and MATLAB. The "HDF" in HDF5 stands for *hierarchical data format*. Each HDF5 file contains an internal file system-like node structure enabling you to store multiple datasets and supporting metadata. Compared with simpler formats, HDF5 supports on-the-fly compression with a variety of compressors, enabling data with repeated patterns to be stored more efficiently. For very large datasets that don't fit into memory, HDF5 is a good choice as you can efficiently read and write small sections of much larger arrays.

There are not one but two interfaces to the HDF5 library in Python, PyTables and h5py, each of which takes a different approach to the problem. h5py provides a direct, but high-level interface to the HDF5 API, while PyTables abstracts many of the details of

HDF5 to provide multiple flexible data containers, table indexing, querying capability, and some support for out-of-core computations.

pandas has a minimal dict-like HDFStore class, which uses PyTables to store pandas objects:

```
In [937]: store = pd.HDFStore('mydata.h5')In [938]:

store['obj1'] = frame

In [939]: store['obj1_col'] = frame['a']

In [940]: store
Out[940]:
<class 'pandas.io.pytables.HDFStore'>
File path: mydata.h5 obj1
                  DataFrame
obj1_col          Series
```

Objects contained in the HDF5 file can be retrieved in a dict-like fashion:

```
In [941]: store['obj1']
Out[941]:
     a     b     c     d message
       0   1     2  3  4     hello
     1         5     6     7     8
           world2   9   10   11
              12               foo
```

If you work with huge quantities of data, I would encourage you to explore PyTables and h5py to see how they can suit your needs. Since many data analysis problems are IO-bound (rather than CPU-bound), using a tool like HDF5 can massively accelerate your applications.

HDF5 is *not* a database. It is best suited for write-once, read-many datasets. While data can be added to a file at any time, if multiple writers do so simultaneously, the file can become corrupted.

Reading Microsoft Excel Files

pandas also supports reading tabular data stored in Excel 2003 (and higher) files using the ExcelFile class. Interally ExcelFile uses the xlrd and openpyxl packages, so you may have to install them first. To use ExcelFile, create an instance by passing a path to an xlsor xlsxfile:

```
xls_file = pd.ExcelFile('data.xls')
```

Data stored in a sheet can then be read into DataFrame using parse:

```
table = xls_file.parse('Sheet1')
```

Interacting with HTML and Web APIs

Many websites have public APIs providing data feeds via JSON or some other format. There are a number of ways to access these APIs from Python; one easy-to-use method that I recommend is the requests package (*http://docs.python-requests.org*). To search for the words “python pandas” on Twitter, we can make an HTTP GET request like so:

```
In [944]: import requests

In [945]: url = 'http://search.twitter.com/search.json?q=python%20pandas'In [946]: resp =

requests.get(url)

In [947]: resp
Out[947]: <Response [200]>
```

The Response object’s text attribute contains the content of the GET query. Many web APIs will return a JSON string that must be loaded into a Python object:

```
In [948]: import json

In [949]: data = json.loads(resp.text)In [950]:

data.keys()
Out[950]:
[u'next_page',
 u'completed_in',
 u'max_id_str',
 u'since_id_str',
 u'refresh_url', u'results',
 u'since_id',
 u'results_per_page',
 u'query',
 u'max_id',
 u'page']
```

The results field in the response contains a list of tweets, each of which is representedas a Python dict that looks like:

```
{u'created_at': u'Mon, 25 Jun 2012 17:50:33 +0000',
 u'from_user': u'wesmckinn',
 u'from_user_id': 115494880,
 u'from_user_id_str': u'115494880',
 u'from_user_name': u'Wes McKinney',
 u'geo': None,
 u'id': 217313849177686018, u'id_str':
 u'217313849177686018',
 u'iso_language_code': u'pt',
 u'metadata': {u'result_type': u'recent'},
 u'source': u'<a href="http://twitter.com/">web</a>',
 u'text': u'Lunchtime pandas-fu http://t.co/SI70xZZQ #pydata',u'to_user':
 None,
 u'to_user_id': 0,
 u'to_user_id_str': u'0',
 u'to_user_name': None}
```

We can then make a list of the tweet fields of interest then pass the results list to DataFrame:

```
In [951]: tweet_fields = ['created_at', 'from_user', 'id', 'text']In [952]: tweets =

DataFrame(data['results'], columns=tweet_fields)

In [953]: tweets
Out[953]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 15 entries, 0 to 14
Data columns:
created_at 15 non-null values from_user
15   non-null values id      15      non-null
values
text        15     non-null   values   dtypes:
int64(1), object(3)
```

Each row in the DataFrame now has the extracted data from each tweet:

```
In [121]: tweets.ix[7]
Out[121]:
created_at                              Thu, 23 Jul 2012 09:54:00 +0000
from_user                                                       deblike
id                                                   227419585803059201
text                 pandas: powerful Python data analysis toolkitName: 7
```

With a bit of elbow grease, you can create some higher-level interfaces to common web APIs that return DataFrame objects for easy analysis.

Interacting with Databases

In many applications data rarely comes from text files, that being a fairly inefficient way to store large amounts of data. SQL-based relational databases (such as SQL Server, PostgreSQL, and MySQL) are in wide use, and many alternative non-SQL (so-called *NoSQL*) databases have become quite popular. The choice of database is usually dependent on the performance, data integrity, and scalability needs of an application.

Loading data from SQL into a DataFrame is fairly straightforward, and pandas has some functions to simplify the process. As an example, I'll use an in-memory SQLite database using Python's built-in sqlite3driver:

```
import sqlite3

query = """
CREATE TABLE test
(a VARCHAR(20), b
  VARCHAR(20),c REAL,
                        d INTEGER
);"""
```

```
con = sqlite3.connect(':memory:')
con.execute(query)
con.commit()
```

Then, insert a few rows of data:

```
data = [('Atlanta', 'Georgia', 1.25, 6),
        ('Tallahassee', 'Florida', 2.6, 3),
        ('Sacramento', 'California', 1.7, 5)]
stmt = "INSERT INTO test VALUES(?, ?, ?, ?)"

con.executemany(stmt, data)
con.commit()
```

Most Python SQL drivers (PyODBC, psycopg2, MySQLdb, pymssql, etc.) return a list of tuples when selecting data from a table:

```
In [956]: cursor = con.execute('select * from test')In [957]: rows =

cursor.fetchall()

In [958]: rows
Out[958]:
[(u'Atlanta', u'Georgia', 1.25, 6),
 (u'Tallahassee', u'Florida', 2.6, 3),
 (u'Sacramento', u'California', 1.7, 5)]
```

You can pass the list of tuples to the DataFrame constructor, but you also need the column names, contained in the cursor's descriptionattribute:

```
In [959]: cursor.description
Out[959]:
(('a', None, None, None, None, None, None),
 ('b', None, None, None, None, None, None),
 ('c', None, None, None, None, None, None),
 ('d', None, None, None, None, None, None))

In [960]: DataFrame(rows, columns=zip(*cursor.description)[0])Out[960]:
             a           b     c  d
0      Atlanta     Georgia  1.25  6
1  Tallahassee     Florida  2.60  3
2   Sacramento  California  1.70  5
```

This is quite a bit of munging that you'd rather not repeat each time you query the database. pandas has a read_frame function in its pandas.io.sql module that simplifies the process. Just pass the select statement and the connection object:

```
In [961]: import pandas.io.sql as sql

In [962]: sql.read_frame('select * from test', con)Out[962]:
             a           b     c  d
0      Atlanta     Georgia  1.25  6
1  Tallahassee     Florida  2.60  3
2   Sacramento  California  1.70  5
```

Storing and Loading Data in MongoDB

NoSQL databases take many different forms. Some are simple dict-like key-value stores like BerkeleyDB or Tokyo Cabinet, while others are document-based, with a dict-like object being the basic unit of storage. I've chosen MongoDB (*http://mongodb.org*) for my example. I started a MongoDB instance locally on my machine, and connect to it on the default port using pymongo, the official driver for MongoDB:

```
import pymongo
con = pymongo.Connection('localhost', port=27017)
```

Documents stored in MongoDB are found in collections inside databases. Each running instance of the MongoDB server can have multiple databases, and each database can have multiple collections. Suppose I wanted to store the Twitter API data from earlier in the chapter. First, I can access the (currently empty) tweets collection:

```
tweets = con.db.tweets
```

Then, I load the list of tweets and write each of them to the collection using tweets.save (which writes the Python dict to MongoDB):

```
import requests, json
url =
'http://search.twitter.com/search.json?q=python%20pandas'
data = json.loads(requests.get(url).text)

for tweet in data['results']:
    tweets.save(tweet)
```

Now, if I wanted to get all of my tweets (if any) from the collection, I can query the collection with the following syntax:

```
cursor = tweets.find({'from_user': 'wesmckinn'})
```

The cursor returned is an iterator that yields each document as a dict. As above I can convert this into a DataFrame, optionally extracting a subset of the data fields in each tweet:

```
tweet_fields = ['created_at', 'from_user', 'id', 'text']result =
DataFrame(list(cursor), columns=tweet_fields)
```

Unit-IV :

Data Wrangling: Combining and Merging Data Sets, Database style DataFrame Merges, Merging on Index, Concatenating Along an Axis, Combining Data with Overlap , Reshaping and Pivoting, Reshaping with Hierarchical Indexing, Data Transformation, Removing Duplicates, Replacing Values.

Data Wrangling: Clean, Transform,Merge, Reshape

Much of the programming work in data analysis and modeling is spent on data preparation: loading, cleaning, transforming, and rearranging. Sometimes the way that data is stored in files or databases is not the way you need it for a data processing application. Many people choose to do ad hoc processing of data from one form to another using a general purpose programming, like Python, Perl, R, or Java, or UNIX text processing tools like sed or awk. Fortunately, pandas along with the Python standard library provide you with a high-level, flexible, and high-performance set of core manipulations and algorithms to enable you to wrangle data into the right form without much trouble.

If you identify a type of data manipulation that isn't anywhere in this book or elsewhere in the pandas library, feel free to suggest it on the mailing list or GitHub site. Indeed, much of the design and implementation of pandas has been driven by the needs of real world applications.

Combining and Merging Data Sets

Data contained in pandas objects can be combined together in a number of built-in ways:

- pandas.merge connects rows in DataFrames based on one or more keys. This will be familiar to users of SQL or other relational databases, as it implements database *join* operations.
- pandas.concat glues or stacks together objects along an axis.
- combine_first instance method enables splicing together overlapping data to fill in missing values in one object with values from another.

I will address each of these and give a number of examples. They'll be utilized in examples throughout the rest of the book.

Database-style DataFrame Merges

Merge or *join* operations combine data sets by linking rows using one or more *keys*. These operations are central to relational databases. The merge function in pandas is the main entry point for using these algorithms on your data.

Let's start with a simple example:

```
In [15]: df1 = DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
   ....:                          'data1': range(7)})

     In [16]: df2 = DataFrame({'key': ['a', 'b', 'd'],
   ....:                          'data2': range(3)})

In [17]: df1              In [18]: df2
Out[17]:                  Out[18]:
   data1 key                 data2 key
0      0   b              0      0   a
1      1   b              1      1   b
2      2   a              2      2   d
3      3   c
4      4   a
5      5   a
6      6   b
```

This is an example of a *many-to-one* merge situation; the data in df1has multiple rows labeled aand b, whereas df2has only one row for each value in the keycolumn. Calling mergewith these objects we obtain:

```
In [19]: pd.merge(df1, df2)
Out[19]:
   data1 key  data20 2
       a   0
1      4   a      0
2      5   a      0
3      0   b      1
4      1   b      1
5      6   b      1
```

Note that I didn't specify which column to join on. If not specified, merge uses the overlapping column names as the keys. It's a good practice to specify explicitly, though:

```
In [20]: pd.merge(df1, df2, on='key')Out[20]:
   data1 key  data20 2
       a   0
1      4   a      0
2      5   a      0
3      0   b      1
4      1   b      1
5      6   b      1
```

If the column names are different in each object, you can specify them separately:

```
In [21]: df3 = DataFrame({'lkey': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
   ....:                          'data1': range(7)})
```

```
In [22]: df4 = DataFrame({'rkey': ['a', 'b', 'd'],
   ....:                          'data2': range(3)})

In [23]: pd.merge(df3, df4, left_on='lkey', right_on='rkey')Out[23]:
   data1 lkey   data2 rkey
0          2       a          0       a
1          4       a          0       a
2          5       a          0       a
3          0       b          1       b
4          1       b          1       b
5          6       b          1       b
```

You probably noticed that the 'c'and 'd'values and associated data are missing fromthe result. By default mergedoes an 'inner'join; the keys in the result are the intersec-tion. Other possible options are 'left', 'right', and 'outer'. The outer join takes theunion of the keys, combining the effect of applying both left and right joins:

```
In [24]: pd.merge(df1, df2, how='outer')Out[24]:
   data1 key   data20 2
          a     0
1          4     a          0
2          5     a          0
3          0     b          1
4          1     b          1
5          6     b          1
6          3     c       NaN
7       NaN    d          2
```

Many-to-many merges have well-defined though not necessarily intuitive behavior. Here's an example:

```
In [25]: df1 = DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],
   ....:                          'data1': range(6)})

In [26]: df2 = DataFrame({'key': ['a', 'b', 'a', 'b', 'd'],
   ....:                          'data2': range(5)})

In [27]: df1                In [28]: df2
Out[27]:                    Out[28]:
   data1 key                   data2 key
0          0     b             0          0     a
1          1     b             1          1     b
2          2     a             2          2     a
3          3     c             3          3     b
4          4     a             4          4     d
5          5     b

In [29]: pd.merge(df1, df2, on='key', how='left')Out[29]:
    data1 key   data20 2
           a     0
1           2     a          2
```

```
2          4    a        0
3          4    a        2
4          0    b        1
5          0    b        3
6          1    b        1
7          1    b        3
8          5    b        1
9          5    b        3
10         3    c      NaN
```

Many-to-many joins form the Cartesian product of the rows. Since there were 3 'b' rows in the left DataFrame and 2 in the right one, there are 6 'b' rows in the result. The join method only affects the distinct key values appearing in the result:

```
In [30]: pd.merge(df1, df2, how='inner')Out[30]:
     data1 key   data20 2
           a     0
1          2     a        2
2          4     a        0
3          4     a        2
4          0     b        1
5          0     b        3
6          1     b        1
7          1     b        3
8          5     b        1
9          5     b        3
```

To merge with multiple keys, pass a list of column names:

```
In [31]: left = DataFrame({'key1': ['foo', 'foo', 'bar'],
    ....:                    'key2': ['one', 'two', 'one'],
    ....:                    'lval': [1, 2, 3]})

In  [32]: right = DataFrame({'key1': ['foo', 'foo', 'bar', 'bar'],
    ....:                     'key2': ['one', 'one', 'one', 'two'],
    ....:                     'rval': [4, 5, 6, 7]})
In [33]: pd.merge(left, right, on=['key1', 'key2'], how='outer')Out[33]:
   key1 key2   lval   rval
0   bar   one        3       6
1   bar   two      NaN       7
2   foo   one        1       4
3   foo   one        1       5
4   foo   two        2     NaN
```

To determine which key combinations will appear in the result depending on the choice of merge method, think of the multiple keys as forming an array of tuples to be used as a single join key (even though it's not actually implemented that way).

When joining columns-on-columns, the indexes on the passed Data-Frame objects are discarded.

A last issue to consider in merge operations is the treatment of overlapping column names. While you can address the overlap manually (see the later section on renaming axis labels), merge has a suffixes option for specifying strings to append to overlapping names in the left and right DataFrame objects:

```
In [34]: pd.merge(left, right, on='key1')Out[34]:
   key1 key2_x  lval key2_y  rval
0  bar     one     3    one     6
1  bar     one     3    two     7
2  foo     one     1    one     4
3  foo     one     1    one     5
4  foo     two     2    one     4
5  foo     two     2    one     5

In [35]: pd.merge(left, right, on='key1', suffixes=('_left', '_right'))Out[35]:
   key1 key2_left  lval key2_right  rval
0  bar        one     3        one     6
1  bar        one     3        two     7
2  foo        one     1        one     4
3  foo        one     1        one     5
4  foo        two     2        one     4
5  foo        two     2        one     5
```

See Table 7-1 for an argument reference on merge. Joining on index is the subject of the next section.

Table 7-1. merge function arguments

Argument	Description
left	DataFrame to be merged on the left side
right	DataFrame to be merged on the right side
how	One of 'inner', 'outer', 'left' or 'right'. 'inner' by default
on	Column names to join on. Must be found in both DataFrame objects. If not specified and no other join keysgiven, will use the intersection of the column names in left and right as the join keys
left_on	Columns in left DataFrame to use as join keys
right_on	Analogous to left_on for left DataFrame
left_index	Use row index in left as its join key (or keys, if a MultiIndex)
right_index	Analogous to left_index
sort	Sort merged data lexicographically by join keys; True by default. Disable to get better performance in somecases on large datasets
suffixes	Tuple of string values to append to column names in case of overlap; defaults to ('_x', '_y'). For example, if 'data' in both DataFrame objects, would appear as 'data_x' and 'data_y' in result
copy	If False, avoid copying data into resulting data structure in some exceptional cases. By default always copies

Merging on Index

In some cases, the merge key or keys in a DataFrame will be found in its index. In this case, you can pass left_index=Trueor right_index=True(or both) to indicate that theindex should be used as the merge key:

```
In [36]: left1 = DataFrame({'key': ['a', 'b', 'a', 'a', 'b', 'c'],
   ....:                              'value': range(6)})

In [37]: right1 = DataFrame({'group_val': [3.5, 7]}, index=['a', 'b'])

In [38]: left1              In [39]: right1
Out[38]:                    Out[39]:
  key  value                   group_val
0   a      0                a        3.5
1   b      1                b        7.0
2   a      2
3   a      3
4   b      4
5   c      5

In [40]: pd.merge(left1, right1, left_on='key', right_index=True)Out[40]:
  key  value  group_val
0   a      0        3.5
2   a      2        3.5
3   a      3        3.5
1   b      1        7.0
4   b      4        7.0
```

Since the default merge method is to intersect the join keys, you can instead form the union of them with an outer join:

```
In [41]: pd.merge(left1, right1, left_on='key', right_index=True, how='outer')Out[41]:
  key  value  group_val
0   a      0        3.5
2   a      2        3.5
3   a      3        3.5
1   b      1        7.0
4   b      4        7.0
5   c      5        NaN
```

With hierarchically-indexed data, things are a bit more complicated:

```
In [42]: lefth = DataFrame({'key1': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],
   ....:                    'key2': [2000, 2001, 2002, 2001, 2002],
   ....:                    'data': np.arange(5.)})

In [43]: righth = DataFrame(np.arange(12).reshape((6, 2)),
   ....:                    index=[['Nevada', 'Nevada', 'Ohio', 'Ohio', 'Ohio', 'Ohio'],
   ....:                           [2001, 2000, 2000, 2000, 2001, 2002]],
   ....:                    columns=['event1', 'event2'])

In [44]: lefth                 In [45]: righth
Out[44]:                       Out[45]:
     data    key1  key2                     event1   event2
0      0    Ohio  2000        Nevada 2001       0        1
1      1    Ohio  2001               2000       2        3
2      2    Ohio  2002        Ohio   2000       4        5
3      3  Nevada  2001               2000       6        7
4      4  Nevada  2002               2001       8        9
                                     2002      10       11
```

In this case, you have to indicate multiple columns to merge on as a list (pay attention to the handling of duplicate index values):

```
In [46]: pd.merge(lefth, righth, left_on=['key1', 'key2'], right_index=True)Out[46]:
   data    key1  key2  event1  event2
3     3  Nevada  2001       0       1
0     0    Ohio  2000       4       5
0     0    Ohio  2000       6       7
1     1    Ohio  2001       8       9
2     2    Ohio  2002      10      11

In [47]: pd.merge(lefth, righth, left_on=['key1', 'key2'],
   ....:                  right_index=True, how='outer')
Out[47]:
   data    key1  key2  event1  event2
4   NaN  Nevada  2000       2       3
3     3  Nevada  2001       0       1
4     4  Nevada  2002     NaN     NaN
0     0    Ohio  2000       4       5
0     0    Ohio  2000       6       7
1     1    Ohio  2001       8       9
2     2    Ohio  2002      10      11
```

Using the indexes of both sides of the merge is also not an issue:

```
In [48]: left2 = DataFrame([[1., 2.], [3., 4.], [5., 6.]], index=['a', 'c', 'e'],
   ....:                   columns=['Ohio', 'Nevada'])

In [49]: right2 = DataFrame([[7., 8.], [9., 10.], [11., 12.], [13, 14]],
   ....:                    index=['b', 'c', 'd', 'e'], columns=['Missouri', 'Alabama'])

 In [50]: left2               In [51]: right2
 Out[50]:                     Out[51]:
   Ohio  Nevada                  Missouri  Alabama
a     1       2               b         7        8
c     3       4               c         9       10
e     5       6               d        11       12
                              e        13       14

In [52]: pd.merge(left2, right2, how='outer', left_index=True, right_index=True)Out[52]:
   Ohio  Nevada  Missouri  Alabama
a     1       2        Na      NaN
                       N
b   NaN     NaN         7        8
c     3       4         9       10
d   NaN     NaN        11       12
e     5       6        13       14
```

DataFrame has a more convenient join instance for merging by index. It can also be used to combine together many DataFrame objects having the same or similar indexes but non-overlapping columns. In the prior example, we could have written:

```
In [53]: left2.join(right2, how='outer')Out[53]:
    Ohio  Nevada  Missouri  Alabama
a      1       2        Na      NaN
                        N
b    NaN     NaN         7        8
c      3       4         9       10
d    NaN     NaN        11       12
e      5       6        13       14
```

In part for legacy reasons (much earlier versions of pandas), DataFrame's joinmethod performs a left join on the join keys. It also supports joining the index of the passed DataFrame on one of the columns of the calling DataFrame:

```
In [54]: left1.join(right1, on='key')Out[54]:
  key  value  group_val
0   a      0        3.5
1   b      1        7.0
2   a      2        3.5
3   a      3        3.5
4   b      4        7.0
5   c      5        NaN
```

Lastly, for simple index-on-index merges, you can pass a list of DataFrames to join as an alternative to using the more general concatfunction described below:

```
In [55]: another = DataFrame([[7., 8.], [9., 10.], [11., 12.], [16., 17.]],
    ....:                   index=['a', 'c', 'e', 'f'], columns=['New York', 'Oregon'])

In [56]: left2.join([right2, another])Out[56]:
    Ohio  Nevada  Missouri  Alabama  New York  Oregon
a      1       2        Na      NaN         7       8
                        N
c      3       4         9       10         9      10
e      5       6        13       14        11      12

In [57]: left2.join([right2, another], how='outer')Out[57]:
    Ohio  Nevada  Missouri  Alabama  New York  Oregona 1
      2       NaN       NaN      7          8
b    NaN      NaN         7        8       NaN
     NaNc       3                  4       9  10      9
      10
d    NaN      NaN        11       12       NaN
     NaNe       5                  6       13        14
      11       12
f    NaN      NaN       NaN      NaN        16       17
```

Concatenating Along an Axis

Another kind of data combination operation is alternatively referred to as concatenation, binding, or stacking. NumPy has a concatenate function for doing this with raw NumPy arrays:

```
In [58]: arr = np.arange(12).reshape((3, 4))

In [59]: arr
Out[59]:
        array([[ 0,   1,   2,   3],
               [ 4,   5,   6,   7],
               [ 8,   9, 10, 11]])

In [60]: np.concatenate([arr, arr], axis=1)Out[60]:
             array([[ 0,   1,   2,   3,   0,   1,   2,   3],
                    [ 4,   5,   6,   7,   4,   5,   6,   7],
                    [ 8,   9, 10, 11,   8,   9, 10, 11]])
```

In the context of pandas objects such as Series and DataFrame, having labeled axes enable you to further generalize array concatenation. In particular, you have a number of additional things to think about:

- If the objects are indexed differently on the other axes, should the collection of axes be unioned or intersected?
- Do the groups need to be identifiable in the resulting object?
- Does the concatenation axis matter at all?

The concatfunction in pandas provides a consistent way to address each of these concerns. I'll give a number of examples to illustrate how it works. Suppose we have three Series with no index overlap:

```
In [61]: s1 = Series([0, 1], index=['a', 'b'])

In [62]: s2 = Series([2, 3, 4], index=['c', 'd', 'e'])

In [63]: s3 = Series([5, 6], index=['f', 'g'])
```

Calling concatwith these object in a list glues together the values and indexes:

```
In [64]: pd.concat([s1, s2, s3])Out[64]:
a      0
b      1
c      2
d      3
e      4
f      5
g      6
```

By default concat works along axis=0, producing another Series. If you pass axis=1, the result will instead be a DataFrame (axis=1is the columns):

```
In [65]: pd.concat([s1, s2, s3], axis=1)Out[65]:
      0    1    2
a     0 NaN NaN
b     1 NaN
NaNc NaN     2
NaN
d  NaN    3 NaN
e  NaN    4
NaNf NaN NaN
   5
g NaN NaN    6
```

In this case there is no overlap on the other axis, which as you can see is the sorted union (the 'outer' join) of the indexes. You can instead intersect them by passing join='inner':

```
In [66]: s4 = pd.concat([s1 * 5, s3])

In [67]: pd.concat([s1, s4], axis=1)

       0   1
  a     0   0
  b     1   5
f  NaN   5
g  NaN   6
```

```
In [68]: pd.concat([s1, s4], axis=1, join='inner')Out[67]:
Out[68]:
     0   1
  a  0   0
  b  1   5
```

You can even specify the axes to be used on the other axes with join_axes:

```
In [69]: pd.concat([s1, s4], axis=1, join_axes=[['a', 'c', 'b', 'e']])Out[69]:
      0    1
a     0    0c
NaN   NaN
b    1    5e
NaN NaN
```

One issue is that the concatenated pieces are not identifiable in the result. Suppose instead you wanted to create a hierarchical index on the concatenation axis. To do this, use the keysargument:

```
In [70]: result = pd.concat([s1, s1, s3], keys=['one', 'two', 'three'])

In [71]: result
Out[71]:
 one       a       0
           b       1
 two       a       0
           b       1
   three   f       5
           g       6

# Much more on the unstack function laterIn [72]:
result.unstack()
Out[72]:
```

```
              a    b    f    g
one          0     1 NaN NaN
two          0     1 NaN NaN
three NaN NaN      5     6
```

In the case of combining Series along axis=1, the keysbecome the DataFrame column headers:

```
In [73]: pd.concat([s1, s2, s3], axis=1, keys=['one', 'two', 'three'])Out[73]:
     one   two   threea
         0   NaN
         NaN
b        1   NaN       NaN
c    NaN       2       NaN
d    NaN       3       NaN
e    NaN       4       NaN
f    NaN   NaN         5
g    NaN   NaN         6
```

The same logic extends to DataFrame objects:

```
In [74]: df1 = DataFrame(np.arange(6).reshape(3, 2), index=['a', 'b', 'c'],
     ....:                                   columns=['one', 'two'])

In [75]: df2 = DataFrame(5 + np.arange(4).reshape(2, 2), index=['a', 'c'],
     ....:                                   columns=['three', 'four'])

In [76]: pd.concat([df1, df2], axis=1, keys=['level1', 'level2'])Out[76]:
     level1              level2
          one   two       three   four
a           0      1           5        6
b           2      3       NaN     NaN
c           4      5           7        8
```

If you pass a dict of objects instead of a list, the dict’s keys will be used for the keys option:

```
In [77]: pd.concat({'level1': df1, 'level2': df2}, axis=1)Out[77]:
     level1              level2
          one   two       three   four
a           0      1           5        6
b           2      3       NaN     NaN
c           4      5           7        8
```

There are a couple of additional arguments governing how the hierarchical index is created (see Table 7-2):

```
In [78]: pd.concat([df1, df2], axis=1, keys=['level1', 'level2'],
     ....:                         names=['upper', 'lower'])
Out[78]:
upper   level1                level2
lower           one   two       three   foura
                0  1            5     6
b                  2       3        NaN     NaN
c                  4       5           7        8
```

A last consideration concerns DataFrames in which the row index is not meaningful in the context of the analysis:

```
In [79]: df1 = DataFrame(np.random.randn(3, 4), columns=['a', 'b', 'c', 'd'])

In [80]: df2 = DataFrame(np.random.randn(2, 3), columns=['b', 'd', 'a'])In [81]: df1 In

[82]: df2
Out[81]:                                        Out[82]:
          a         b         c         d                 b         d         a
0 -0.204708  0.478943 -0.519439 -0.555730   0  0.274992  0.228913  1.352917
1  1.965781  1.393406  0.092908  0.281746   1  0.886429 -2.001637 -0.371843
2  0.769023  1.246435  1.007189 -1.296221
```

In this case, you can pass ignore_index=True:

```
In [83]: pd.concat([df1, df2], ignore_index=True)Out[83]:
          a         b         c         d
0 -0.204708  0.478943 -0.519439 -0.555730
1  1.965781  1.393406  0.092908  0.281746
2  0.769023  1.246435  1.007189 -1.296221
3  1.352917  0.274992       NaN  0.228913
4 -0.371843  0.886429       NaN -2.001637
```

Table 7-2. concat function arguments

Argument	Description
objs	List or dict of pandas objects to be concatenated. The only required argument
axis	Axis to concatenate along; defaults to 0
join	One of 'inner', 'outer', defaulting to'outer'; whether to intersection (inner) or union(outer) together indexes along the other axes
join_axes	Specific indexes to use for the other n-1 axes instead of performing union/intersection logic
keys	Values to associate with objects being concatenated, forming a hierarchical index along the concatenation axis. Can either be a list or array of arbitrary values, an array of tuples, or a list ofarrays (if multiple level arrays passed in levels)
levels	Specific indexes to use as hierarchical index level or levels if keys passed
names	Names for created hierarchical levels if keys and / or levels passed
verify_integrity	Check new axis in concatenated object for duplicates and raise exception if so. By default(False) allows duplicates
ignore_index	Do not preserve indexes along concatenation axis, instead producing a new range(total_length) index

Combining Data with Overlap

Another data combination situation can't be expressed as either a merge or concatenation operation. You may have two datasets whose indexes overlap in full or part. As a motivating example, consider NumPy's where function, which expressed a vectorized if-else:

```
In [84]: a = Series([np.nan, 2.5, np.nan, 3.5, 4.5, np.nan],
   ....:            index=['f', 'e', 'd', 'c', 'b', 'a'])

In [85]: b = Series(np.arange(len(a), dtype=np.float64),
   ....:            index=['f', 'e', 'd', 'c', 'b', 'a'])

In [86]: b[-1] = np.nan

In [87]: a          In [88]: b          In [89]: np.where(pd.isnull(a), b, a)
Out[87]:            Out[88]:            Out[89]:
f    NaN            f    0              f    0.0
e    2.5            e    1              e    2.5
d    NaN            d    2              d    2.0
c    3.5            c    3              c    3.5
b    4.5            b    4              b    4.5
a    NaN            a  NaN              a    NaN
```

Series has a combine_first method, which performs the equivalent of this operation plus data alignment:

```
In [90]: b[:-2].combine_first(a[2:])Out[90]:
a    NaN
b    4.5
c    3.0
d    2.0
e    1.0
f    0.0
```

With DataFrames, combine_first naturally does the same thing column by column, so you can think of it as "patching" missing data in the calling object with data from the object you pass:

```
In [91]: df1 = DataFrame({'a': [1., np.nan, 5., np.nan],
   ....:                  'b': [np.nan, 2., np.nan, 6.],
   ....:                  'c': range(2, 18, 4)})

In [92]: df2 = DataFrame({'a': [5., 4., np.nan, 3., 7.],
   ....:                  'b': [np.nan, 3., 4., 6., 8.]})

In [93]: df1.combine_first(df2)
Out[93]:
   a    b    c
0  1  NaN    2
1  4    2    6
2  5    4   10
3  3    6   14
4  7    8  NaN
```

Reshaping and Pivoting

There are a number of fundamental operations for rearranging tabular data. These are alternatingly referred to as *reshape* or *pivot* operations.

Reshaping with Hierarchical Indexing

Hierarchical indexing provides a consistent way to rearrange data in a DataFrame. There are two primary actions:

- stack: this “rotates” or pivots from the columns in the data to the rows
- unstack: this pivots from the rows into the columns

I’ll illustrate these operations through a series of examples. Consider a small DataFrame with string arrays as row and column indexes:

```
In [94]: data = DataFrame(np.arange(6).reshape((2, 3)),
   ....:                              index=pd.Index(['Ohio', 'Colorado'], name='state'),
   ....:                              columns=pd.Index(['one', 'two', 'three'], name='number'))

In [95]: data
Out[95]:
number       one two three
state
Ohio           0     1        2
Colorado       3     4        5
```

Using the stack method on this data pivots the columns into the rows, producing a Series:

```
In [96]: result = data.stack()

In [97]: result
Out[97]:
state         number
Ohio          one          0
              two          1
              three        2
Colorado    one            3
              two          4
              three        5
```

From a hierarchically-indexed Series, you can rearrange the data back into a DataFrame with unstack:

```
In [98]: result.unstack()
Out[98]:
number       one two three
state
Ohio           0     1        2
Colorado       3     4        5
```

By default the innermost level is unstacked (same with stack). You can unstack a different level by passing a level number or name:

```
In [99]: result.unstack(0)                    In [100]: result.unstack('state')Out[99]:
                                              Out[100]:
state       Ohio   Colorado                   state       Ohio   Colorado
number                                        number
one            0            3                 one            0            3
```

```
two          1          4               two          1          4
three        2          5               three        2          5
```

Unstacking might introduce missing data if all of the values in the level aren't found in each of the subgroups:

```
In [101]: s1 = Series([0, 1, 2, 3], index=['a', 'b', 'c', 'd'])

In [102]: s2 = Series([4, 5, 6], index=['c', 'd', 'e'])

In [103]: data2 = pd.concat([s1, s2], keys=['one', 'two'])In [104]:

data2.unstack()
Out[104]:
       a    b  c  d     e
one    0    1  2  3 NaN
two NaN NaN   4  5     6
```

Stacking filters out missing data by default, so the operation is easily invertible:

```
In [105]: data2.unstack().stack()                In [106]: data2.unstack().stack(dropna=False)Out[105]:
                                                 Out[106]:
one   a       0                                  one   a       0
      b       1                                        b       1
      c       2                                        c       2
      d       3                                        d       3
two   c       4                                        e     NaN
              d       5            two   a                   NaN
      e       6                                        b     NaN
                                                       c       4
                                                       d       5
                                                       e       6
```

When unstacking in a DataFrame, the level unstacked becomes the lowest level in the result:

```
In [107]: df = DataFrame({'left': result, 'right': result + 5},
     .....:                          columns=pd.Index(['left', 'right'], name='side'))

In [108]: df
Out[108]:
side                  left    right
state       number
Ohio        one          0        5
            two          1        6
              three      2        7
Colorado one             3        8
            two          4        9
              three      5       10
```

```
In [109]: df.unstack('state')                         In [110]: df.unstack('state').stack('side')Out[109]:
                                                      Out[110]:
side       left                  right                state              Ohio Colorado
state      Ohio   Colorado       Ohio   Colorado      number side
number                                                 one      left         0          3
one            0           3         5          8               right        5          8
two            1           4         6          9     two      left         1          4
```

```
three        2          5       7          10                     right      6         9
                                                            three   left        2         5
                                                                           right      7        10
```

Pivoting “long” to “wide” Format

A common way to store multiple time series in databases and CSV is in so-called *long* or *stacked* format:

```
In [116]: ldata[:10]
Out[116]:
                  date     item     value
   0 1959-03-31 00:00:00  realgdp  2710.349
   1 1959-03-31 00:00:00     infl     0.000
   2 1959-03-31 00:00:00    unemp     5.800
   3 1959-06-30 00:00:00  realgdp  2778.801
   4 1959-06-30 00:00:00     infl     2.340
   5 1959-06-30 00:00:00    unemp     5.100
   6 1959-09-30 00:00:00  realgdp  2775.488
   7 1959-09-30 00:00:00     infl     2.740
   8 1959-09-30 00:00:00    unemp     5.300
   9 1959-12-31 00:00:00  realgdp  2785.204
```

Data is frequently stored this way in relational databases like MySQL as a fixed schema (column names and data types) allows the number of distinct values in the itemcolumn to increase or decrease as data is added or deleted in the table. In the above example date and item would usually be the primary keys (in relational database parlance), offering both relational integrity and easier joins and programmatic queries in many cases. The downside, of course, is that the data may not be easy to work with in long format; you might prefer to have a DataFrame containing one column per distinct item value indexed by timestamps in the date column. DataFrame’s pivot method performs exactly this transformation:

```
In [117]: pivoted = ldata.pivot('date', 'item', 'value')

In [118]: pivoted.head()
Out[118]:
item          infl     realgdp  unemp
date
1959-03-31  0.00  2710.349     5.8
1959-06-30  2.34  2778.801     5.1
1959-09-30  2.74  2775.488     5.3
1959-12-31  0.27  2785.204     5.6
1960-03-31  2.31  2847.699     5.2
```

The first two values passed are the columns to be used as the row and column index, and finally an optional value column to fill the DataFrame. Suppose you had two value columns that you wanted to reshape simultaneously:

```
In [119]: ldata['value2'] = np.random.randn(len(ldata))

In [120]: ldata[:10]
Out[120]:
```

```
                 date      item      value     value2
0 1959-03-31 00:00:00   realgdp   2710.349   1.669025
1 1959-03-31 00:00:00      infl      0.000  -0.438570
2 1959-03-31 00:00:00     unemp      5.800  -0.539741
3 1959-06-30 00:00:00   realgdp   2778.801   0.476985
4 1959-06-30 00:00:00      infl      2.340   3.248944
5 1959-06-30 00:00:00     unemp      5.100  -1.021228
6 1959-09-30 00:00:00   realgdp   2775.488  -0.577087
7 1959-09-30 00:00:00      infl      2.740   0.124121
8 1959-09-30 00:00:00     unemp      5.300   0.302614
9 1959-12-31 00:00:00   realgdp   2785.204   0.523772
```

By omitting the last argument, you obtain a DataFrame with hierarchical columns:

```
In [121]: pivoted = ldata.pivot('date', 'item')

In [122]: pivoted[:5]
Out[122]:
            value                    value2
item         infl   realgdp unemp      infl   realgdp     unemp
date
1959-03-31   0.00  2710.349   5.8 -0.438570  1.669025 -0.539741
1959-06-30   2.34  2778.801   5.1  3.248944  0.476985 -1.021228
1959-09-30   2.74  2775.488   5.3  0.124121 -0.577087  0.302614
1959-12-31   0.27  2785.204   5.6  0.000940  0.523772  1.343810
1960-03-31   2.31  2847.699   5.2 -0.831154 -0.713544 -2.370232

In [123]: pivoted['value'][:5]
Out[123]:
item        infl   realgdp  unemp
date
1959-03-31  0.00  2710.349    5.8
1959-06-30  2.34  2778.801    5.1
1959-09-30  2.74  2775.488    5.3
1959-12-31  0.27  2785.204    5.6
1960-03-31  2.31  2847.699    5.2
```

Note that pivotis just a shortcut for creating a hierarchical index using set_indexandreshaping with unstack:

```
In [124]: unstacked = ldata.set_index(['date', 'item']).unstack('item')

In [125]: unstacked[:7]
Out[125]:
            value                    value2
item         infl   realgdp unemp      infl   realgdp     unemp
date
1959-03-31   0.00  2710.349   5.8 -0.438570  1.669025 -0.539741
1959-06-30   2.34  2778.801   5.1  3.248944  0.476985 -1.021228
1959-09-30   2.74  2775.488   5.3  0.124121 -0.577087  0.302614
1959-12-31   0.27  2785.204   5.6  0.000940  0.523772  1.343810
1960-03-31   2.31  2847.699   5.2 -0.831154 -0.713544 -2.370232
1960-06-30   0.14  2834.390   5.2 -0.860757 -1.860761  0.560145
1960-09-30   2.70  2839.022   5.6  0.119827 -1.265934 -1.063512
```

Data Transformation

So far in this chapter we've been concerned with rearranging data. Filtering, cleaning, and other tranformations are another class of important operations.

Removing Duplicates

Duplicate rows may be found in a DataFrame for any number of reasons. Here is an example:

```
In [126]: data = DataFrame({'k1': ['one'] * 3 + ['two'] * 4,
 .....:                    'k2': [1, 1, 2, 3, 3, 4, 4]})

In [127]: data
Out[127]:
    k1  k2
0  one   1
1  one   1
2  one   2
3  two   3
4  two   3
5  two   4
6  two   4
```

The DataFrame method duplicated returns a boolean Series indicating whether each row is a duplicate or not:

```
In [128]: data.duplicated()
Out[128]:
0    False
1     True
2    False
3    False
4     True
5    False
6     True
```

Relatedly, drop_duplicates returns a DataFrame where the duplicated array is True:

```
In [129]: data.drop_duplicates()
Out[129]:
    k1  k2
0  one   1
2  one   2
3  two   3
5  two   4
```

Both of these methods by default consider all of the columns; alternatively you can specify any subset of them to detect duplicates. Suppose we had an additional column of values and wanted to filter duplicates only based on the 'k1'column:

```
In [130]: data['v1'] = range(7)

In [131]: data.drop_duplicates(['k1'])
```

```
Out[131]:
      k1  k2  v1
0  one     1    0
3  two     3    3
```

duplicated and drop_duplicates by default keep the first observed value combination. Passing take_last=True will return the last one:

```
In [132]: data.drop_duplicates(['k1', 'k2'], take_last=True)Out[132]:
     k1  k2  v1
1   one    1    1
2   one    2    2
4  two     3    4
6  two     4    6
```

Transforming Data Using a Function or Mapping

For many data sets, you may wish to perform some transformation based on the values in an array, Series, or column in a DataFrame. Consider the following hypothetical data collected about some kinds of meat:

```
In [133]: data = DataFrame({'food': ['bacon', 'pulled pork', 'bacon', 'Pastrami',
   .....:                            'corned beef', 'Bacon', 'pastrami', 'honey ham',
   .....:                            'nova lox'],
   .....:                   'ounces': [4, 3, 12, 6, 7.5, 8, 3, 5, 6]})

In [134]: data
Out[134]:
            food  ounces
0          bacon     4.0
1  pulled pork       3.0
2          bacon    12.0
3       Pastrami     6.0
4  corned beef       7.5
5          Bacon     8.0
6       pastrami     3.0
7      honey ham     5.0
8       nova lox     6.0
```

Suppose you wanted to add a column indicating the type of animal that each food came from. Let’s write down a mapping of each distinct meat type to the kind of animal:

```
meat_to_animal = {
  'bacon': 'pig',
  'pulled pork': 'pig',
  'pastrami': 'cow',
  'corned beef': 'cow',
  'honey ham': 'pig', 'nova
  lox': 'salmon'
}
```

The mapmethod on a Series accepts a function or dict-like object containing a mapping, but here we have a small problem in that some of the meats above are capitalized and others are not. Thus, we also need to convert each value to lower case:

```
In [136]: data['animal'] = data['food'].map(str.lower).map(meat_to_animal)

In [137]: data
Out[137]:
             food  ounces  animal
0           bacon     4.0     pig
1     pulled pork     3.0     pig
2           bacon    12.0     pig
3        Pastrami     6.0     cow
4     corned beef     7.5     cow
5           Bacon     8.0     pig
6        pastrami     3.0     cow
7       honey ham     5.0     pig
8        nova lox     6.0  salmon
```

We could also have passed a function that does all the work:

```
In [138]: data['food'].map(lambda x: meat_to_animal[x.lower()])Out[138]:
0        pig
1        pig
2        pig
3        cow
4        cow
5        pig
6        cow
7        pig
8     salmon
Name: food
```

Using mapis a convenient way to perform element-wise transformations and other data cleaning-related operations.

Replacing Values

Filling in missing data with the fillna method can be thought of as a special case of more general value replacement. While map, as you've seen above, can be used to modify a subset of values in an object, replaceprovides a simpler and more flexible way to do so. Let's consider this Series:

```
In [139]: data = Series([1., -999., 2., -999., -1000., 3.])

In [140]: data
Out[140]:
0        1
1     -999
2        2
3     -999
4    -1000
5        3
```

The -999 values might be sentinel values for missing data. To replace these with NA values that pandas understands, we can use replace, producing a new Series:

```
In [141]: data.replace(-999, np.nan)
Out[141]:
0        1
1      NaN
2        2
3      NaN
4    -1000
5        3
```

If you want to replace multiple values at once, you instead pass a list then the substitute value:

```
In [142]: data.replace([-999, -1000], np.nan)Out[142]:
0      1
1    NaN
2      2
3    NaN
4    NaN
5      3
```

To use a different replacement for each value, pass a list of substitutes:

```
In [143]: data.replace([-999, -1000], [np.nan, 0])Out[143]:
0      1
1    NaN
2      2
3    NaN
4      0
5      3
```

The argument passed can also be a dict:

```
In [144]: data.replace({-999: np.nan, -1000: 0})Out[144]:
0      1
1    NaN
2      2
3    NaN
4      0
5      3
```

Unit-V:

Plotting and Visualization: A Brief matplotlib API Primer, Figures and Subplots, Colors, Markers, and Line Styles, Ticks, Labels, and Legends, Annotations and Drawing on a Subplot, Saving Plots to File, Plotting Functions in pandas, Line Plots, Bar Plots, Histograms and Density Plots, Scatter Plots.

Plotting and Visualization

Making plots and static or interactive visualizations is one of the most important tasks in data analysis. It may be a part of the exploratory process; for example, helping identify outliers, needed data transformations, or coming up with ideas for models. For others, building an interactive visualization for the web using a toolkit like d3.js (*http://d3js.org/*) may be the end goal. Python has many visualization tools (see the end of this chapter), but I'll be mainly focused on matplotlib (*http://matplotlib.sourceforge.net*).

matplotlib is a (primarily 2D) desktop plotting package designed for creating publication-quality plots. The project was started by John Hunter in 2002 to enable a MATLAB-like plotting interface in Python. He, Fernando Pérez (of IPython), and others have collaborated for many years since then to make IPython combined with matplotlib a very functional and productive environment for scientific computing. When used in tandem with a GUI toolkit (for example, within IPython), matplotlib has interactive features like zooming and panning. It supports many different GUI backends on all operating systems and additionally can export graphics to all of the common vector and raster graphics formats: PDF, SVG, JPG, PNG, BMP, GIF, etc. I have used it to produce almost all of the graphics outside of diagrams in this book.

matplotlib has a number of add-on toolkits, such as mplot3d for 3D plots and basemap for mapping and projections. I will give an example using basemapto plot data on a map and to read *shapefiles* at the end of the chapter.

To follow along with the code examples in the chapter, make sure you have started IPython in Pylab mode (ipython --pylab) or enabled GUI event loop integration withthe %guimagic.

A Brief matplotlib API Primer

There are several ways to interact with matplotlib. The most common is through *pylab mode* in IPython by running ipython --pylab. This launches IPython configured to be able to support the matplotlib GUI backend of your choice (Tk, wxPython, PyQt, Mac

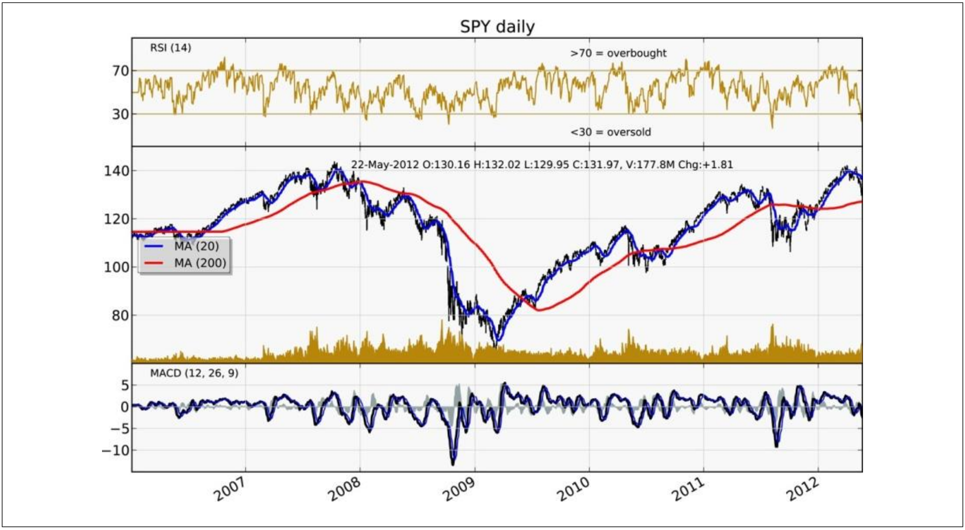


Figure 8-1. A more complex matplotlib financial plot

OS X native, GTK). For most users, the default backend will be sufficient. Pylab mode also imports a large set of modules and functions into IPython to provide a more MATLAB-like interface. You can test that everything is working by making a simple plot:

```
plot(np.arange(10))
```

If everything is set up right, a new window should pop up with a line plot. You can close it by using the mouse or entering close(). Matplotlib API functions like plot and close are all in the matplotlib.pyplot module, which is typically imported by conven- tion as:

```
import matplotlib.pyplot as plt
```

While the pandas plotting functions described later deal with many of the mundane details of making plots, should you wish to customize them beyond the function options provided you will need to learn a bit about the matplotlib API.

There is not enough room in the book to give a comprehensive treatment to the breadth and depth of functionality in matplotlib. It should be enough to teach you the ropes to get up and running. The matplotlib gallery and documentation are the best resource for becoming a plotting guru and using advanced features.

Figures and Subplots

Plots in matplotlib reside within a Figure object. You can create a new figure with plt.figure:

```
In [13]: fig = plt.figure()
```

If you are in pylab mode in IPython, a new empty window should pop up. plt.fig ure has a number of options, notably figsize will guarantee the figure has a certain size and aspect ratio if saved to disk. Figures in matplotlib also support a numbering scheme(for example, plt.figure(2)) that mimics MATLAB. You can get a reference to the active figure using plt.gcf().

You can't make a plot with a blank figure. You have to create one or more subplots using add_subplot:

```
In [14]: ax1 = fig.add_subplot(2, 2, 1)
```

This means that the figure should be 2 × 2, and we're selecting the first of 4 subplots (numbered from 1). If you create the next two subplots, you'll end up with a figure that looks like Figure 8-2.

```
In [15]: ax2 = fig.add_subplot(2, 2, 2)

In [16]: ax3 = fig.add_subplot(2, 2, 3)
```

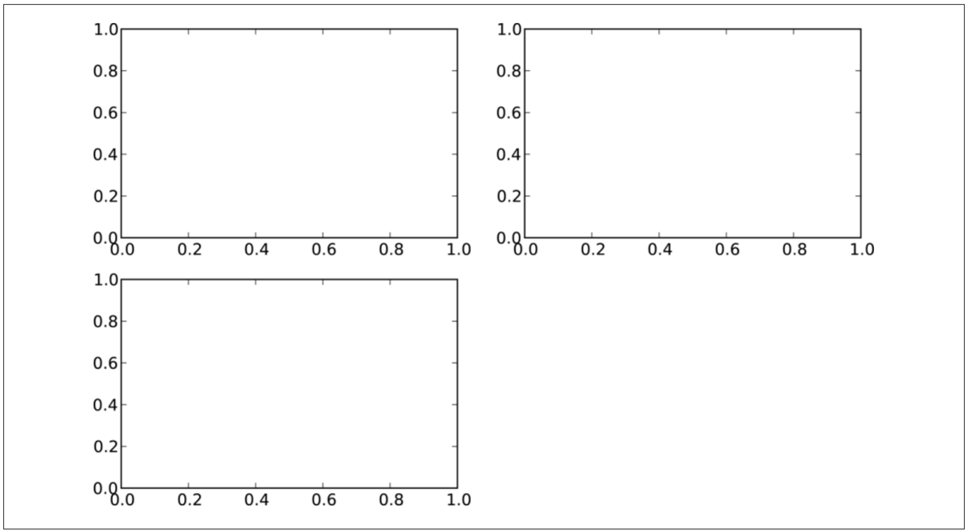


Figure 8-2. An empty matplotlib Figure with 3 subplots

When you issue a plotting command like plt.plot([1.5, 3.5, -2, 1.6]), matplotlibdraws on the last figure and subplot used (creating one if necessary), thus hiding the figure and subplot creation. Thus, if we run the following command, you'll get some-thing like Figure 8-3:

```
In [17]: from numpy.random import randn

In [18]: plt.plot(randn(50).cumsum(), 'k--')
```

The 'k--'is a *style* option instructing matplotlib to plot a black dashed line. The objects returned by fig.add_subplot above are AxesSubplot objects, on which you can directly plot on the other empty subplots by calling each one's instance methods, see Figure 8-4:

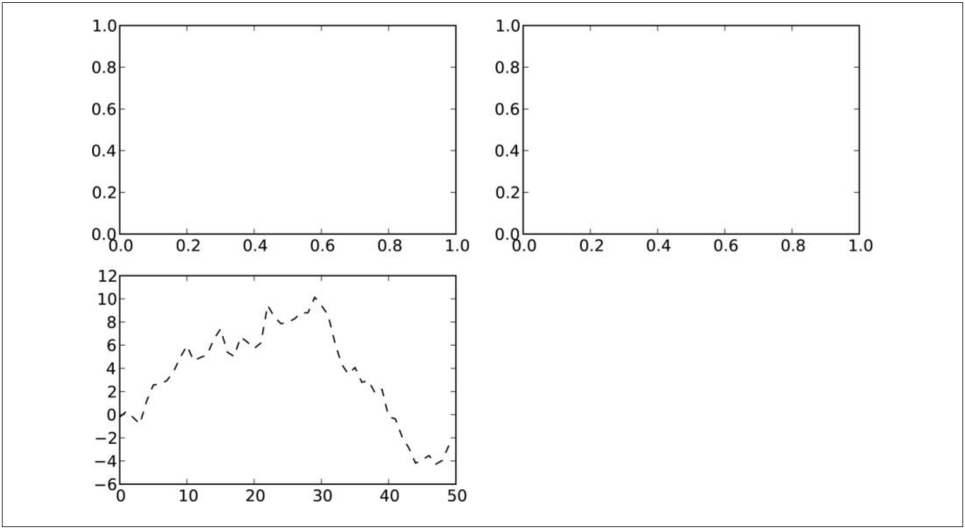


Figure 8-3. Figure after single plot

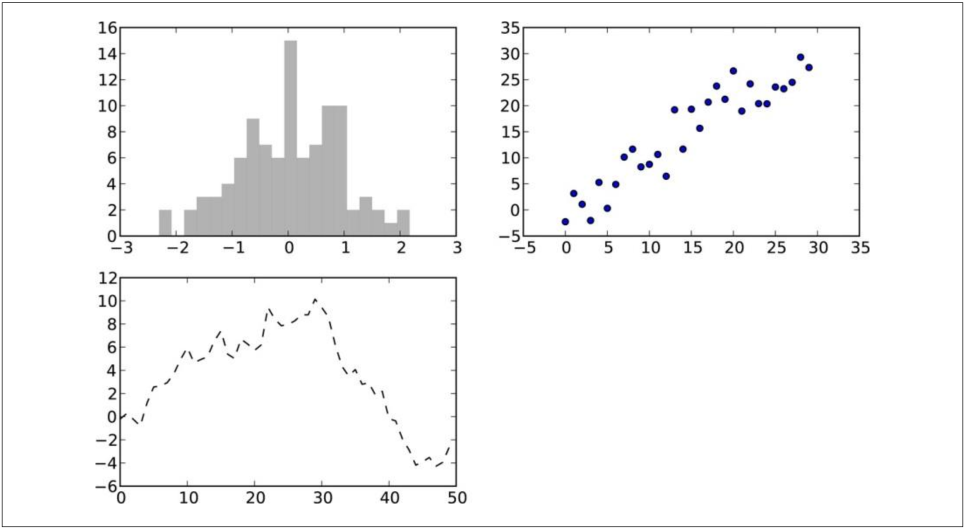


Figure 8-4. Figure after additional plots

```
In [19]: _ = ax1.hist(randn(100), bins=20, color='k', alpha=0.3) In [20]:

ax2.scatter(np.arange(30), np.arange(30) + 3 * randn(30))
```

You can find a comprehensive catalogue of plot types in the matplotlib documentation.

Since creating a figure with multiple subplots according to a particular layout is such a common task, there is a convenience method, plt.subplots, that creates a new figure and returns a NumPy array containing the created subplot objects:

```
In [22]: fig, axes = plt.subplots(2, 3)

In [23]: axes
Out[23]:
array([[Axes(0.125,0.536364;0.227941x0.363636),
        Axes(0.398529,0.536364;0.227941x0.363636),
        Axes(0.672059,0.536364;0.227941x0.363636)],
       [Axes(0.125,0.1;0.227941x0.363636), Axes(0.398529,0.1;0.227941x0.363636),
        Axes(0.672059,0.1;0.227941x0.363636)]], dtype=object)
```

This is very useful as the axes array can be easily indexed like a two-dimensional array; for example, axes[0, 1]. You can also indicate that subplots should have the same X or Y axis using sharex and sharey, respectively. This is especially useful when comparing data on the same scale; otherwise, matplotlib auto-scales plot limits independently. See Table 8-1 for more on this method.

Table 8-1. pyplot.subplots options

Argument	Description
nrows	Number of rows of subplots
ncols	Number of columns of subplots
sharex	All subplots should use the same X-axis ticks (adjusting the xlim will affect all subplots)
sharey	All subplots should use the same Y-axis ticks (adjusting the ylim will affect all subplots)
subplot_kw	Dict of keywords for creating the
**fig_kw	Additional keywords to subplots are used when creating the figure, such as plt.subplots(2, 2, figsize=(8, 6))

Adjusting the spacing around subplots

By default matplotlib leaves a certain amount of padding around the outside of the subplots and spacing between subplots. This spacing is all specified relative to the height and width of the plot, so that if you resize the plot either programmatically or manually using the GUI window, the plot will dynamically adjust itself. The spacing can be most easily changed using the subplots_adjust Figure method, also available as a top-level function:

```
subplots_adjust(left=None, bottom=None, right=None, top=None,
                wspace=None, hspace=None)
```

wspace and hspace controls the percent of the figure width and figure height, respectively, to use as spacing between subplots. Here is a small example where I shrink the spacing all the way to zero (see Figure 8-5):

```
fig, axes = plt.subplots(2, 2, sharex=True, sharey=True)
for i in range(2):
    for j in range(2):
        axes[i, j].hist(randn(500), bins=50, color='k', alpha=0.5)
plt.subplots_adjust(wspace=0, hspace=0)
```

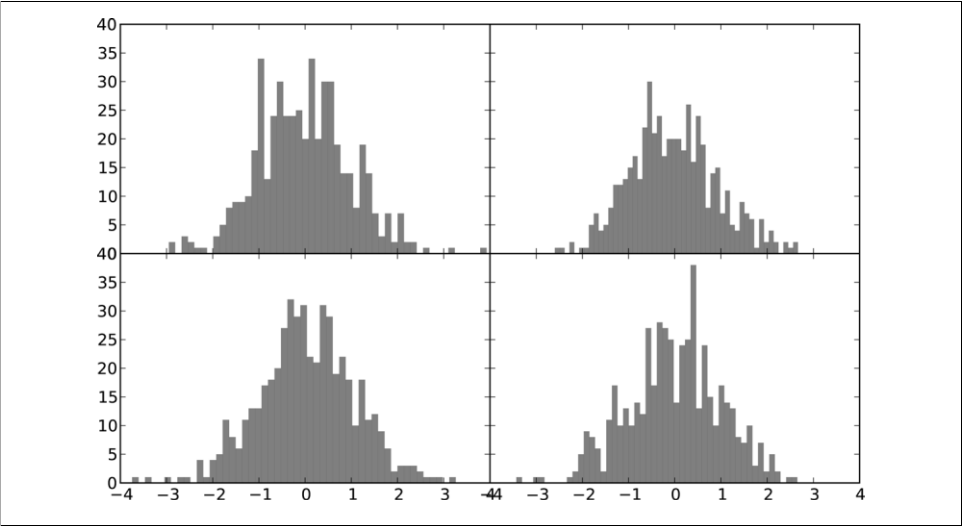


Figure 8-5. Figure with no inter-subplot spacing

You may notice that the axis labels overlap. matplotlib doesn't check whether the labels overlap, so in a case like this you would need to fix the labels yourself by specifying explicit tick locations and tick labels. More on this in the coming sections.

Colors, Markers, and Line Styles

Matplotlib's main plotfunction accepts arrays of X and Y coordinates and optionallya string abbreviation indicating color and line style. For example, to plot xversus ywith green dashes, you would execute:

```
ax.plot(x, y, 'g--')
```

This way of specifying both color and linestyle in a string is provided as a convenience; in practice if you were creating plots programmatically you might prefer not to have to munge strings together to create plots with the desired style. The same plot could also have been expressed more explicitly as:

```
ax.plot(x, y, linestyle='--', color='g')
```

There are a number of color abbreviations provided for commonly-used colors, but any color on the spectrum can be used by specifying its RGB value (for example, '#CECE CE'). You can see the full set of linestyles by looking at the docstring for plot.

Line plots can additionally have *markers* to highlight the actual data points. Since matplotlib creates a continuous line plot, interpolating between points, it can occasionally be unclear where the points lie. The marker can be part of the style string, which must have color followed by marker type and line style (see Figure 8-6):

```
In [28]: plt.plot(randn(30).cumsum(), 'ko--')
```

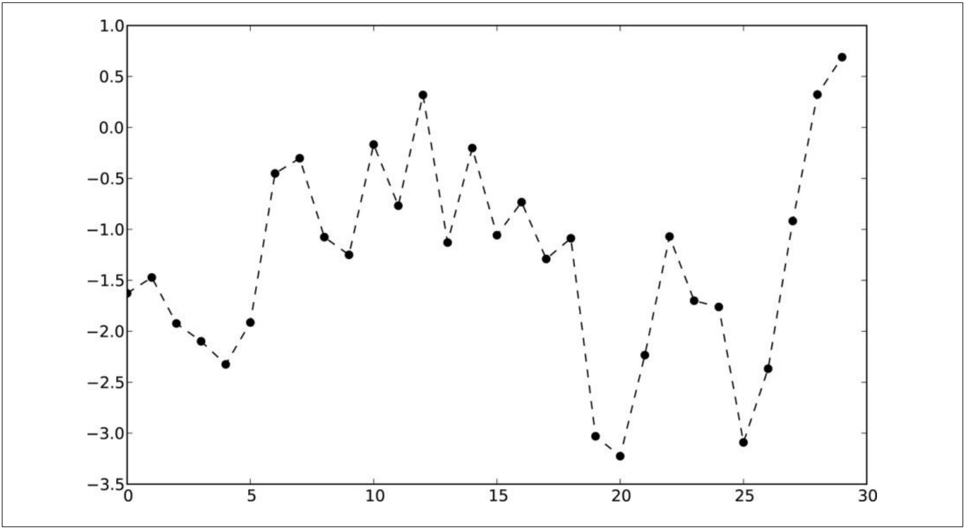


Figure 8-6. Line plot with markers example

This could also have been written more explicitly as:

```
plot(randn(30).cumsum(), color='k', linestyle='dashed', marker='o')
```

For line plots, you will notice that subsequent points are linearly interpolated by default. This can be altered with the drawstyleoption:

```
In [30]: data = randn(30).cumsum()

In [31]: plt.plot(data, 'k--', label='Default') Out[31]:
[<matplotlib.lines.Line2D at 0x461cdd0>]

In [32]: plt.plot(data, 'k-', drawstyle='steps-post', label='steps-post')Out[32]:
[<matplotlib.lines.Line2D at 0x461f350>]

In [33]: plt.legend(loc='best')
```

Ticks, Labels, and Legends

For most kinds of plot decorations, there are two main ways to do things: using the procedural pyplot interface (which will be very familiar to MATLAB users) and the more object-oriented native matplotlib API.

The pyplot interface, designed for interactive use, consists of methods like xlim, xticks, and xticklabels. These control the plot range, tick locations, and tick labels, respectively. They can be used in two ways:

- Called with no arguments returns the current parameter value. For example plt.xlim()returns the current X axis plotting range

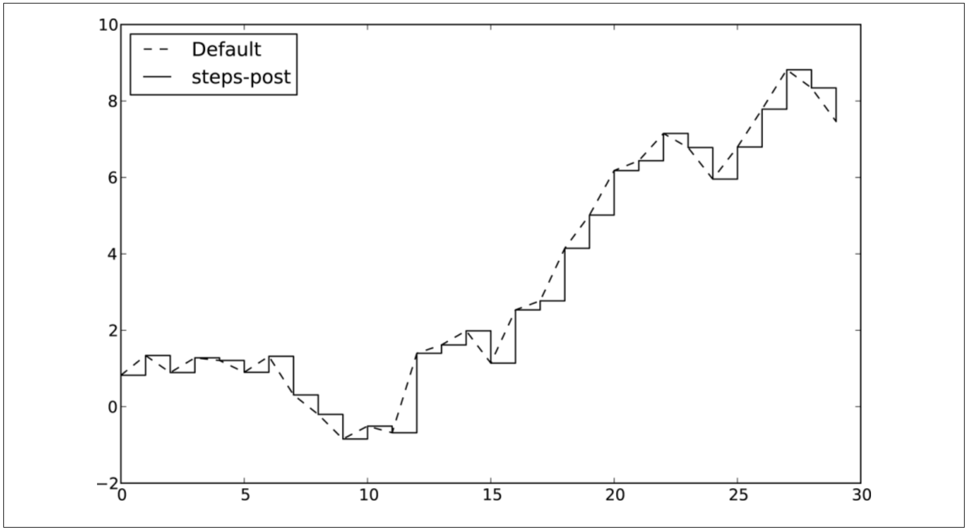


Figure 8-7. Line plot with different drawstyle options

- Called with parameters sets the parameter value. So plt.xlim([0, 10]), sets the Xaxis range to 0 to 10

All such methods act on the active or most recently-created AxesSubplot. Each of them corresponds to two methods on the subplot object itself; in the case of xlim these are ax.get_xlim and ax.set_xlim. I prefer to use the subplot instance methods myself in the interest of being explicit (and especially when working with multiple subplots), butyou can certainly use whichever you find more convenient.

Setting the title, axis labels, ticks, and ticklabels

To illustrate customizing the axes, I'll create a simple figure and plot of a random walk (see Figure 8-8):

```
In [34]: fig = plt.figure(); ax = fig.add_subplot(1, 1, 1)In [35]:

ax.plot(randn(1000).cumsum())
```

To change the X axis ticks, it's easiest to use set_xticks and set_xticklabels. The former instructs matplotlib where to place the ticks along the data range; by default these locations will also be the labels. But we can set any other values as the labels using set_xticklabels:

```
In [36]: ticks = ax.set_xticks([0, 250, 500, 750, 1000])

In [37]: labels = ax.set_xticklabels(['one', 'two', 'three', 'four', 'five'],
   ....:                                   rotation=30, fontsize='small')
```

Lastly, set_xlabel gives a name to the X axis and set_title the subplot title:

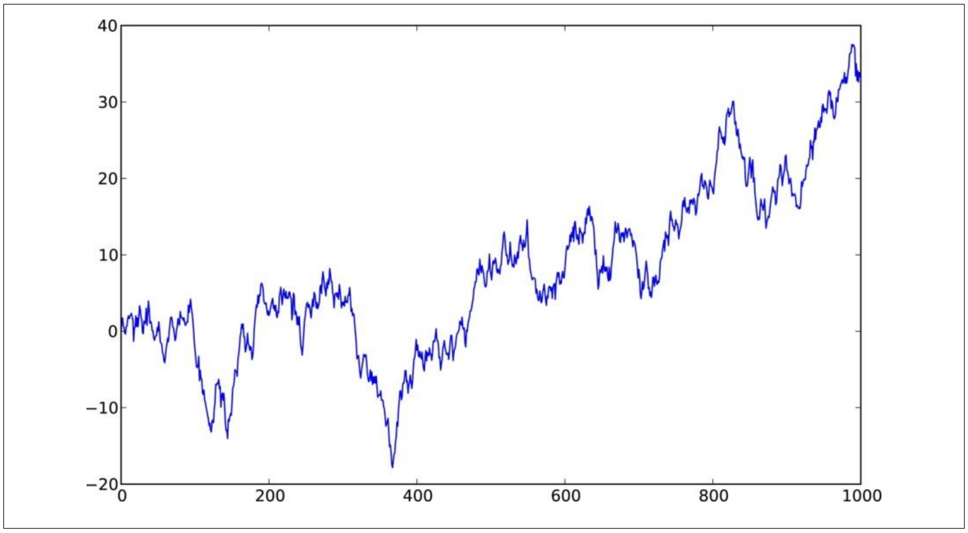


Figure 8-8. Simple plot for illustrating xticks

```
In [38]: ax.set_title('My first matplotlib plot')Out[38]:
<matplotlib.text.Text at 0x7f9190912850>

In [39]: ax.set_xlabel('Stages')
```

See Figure 8-9 for the resulting figure. Modifying the Y axis consists of the same process, substituting yfor xin the above.

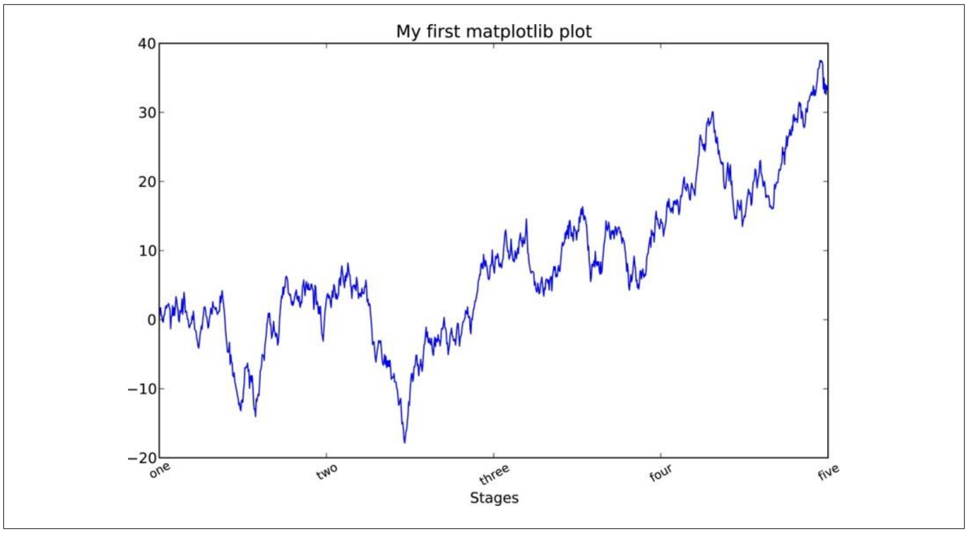


Figure 8-9. Simple plot for illustrating xticks

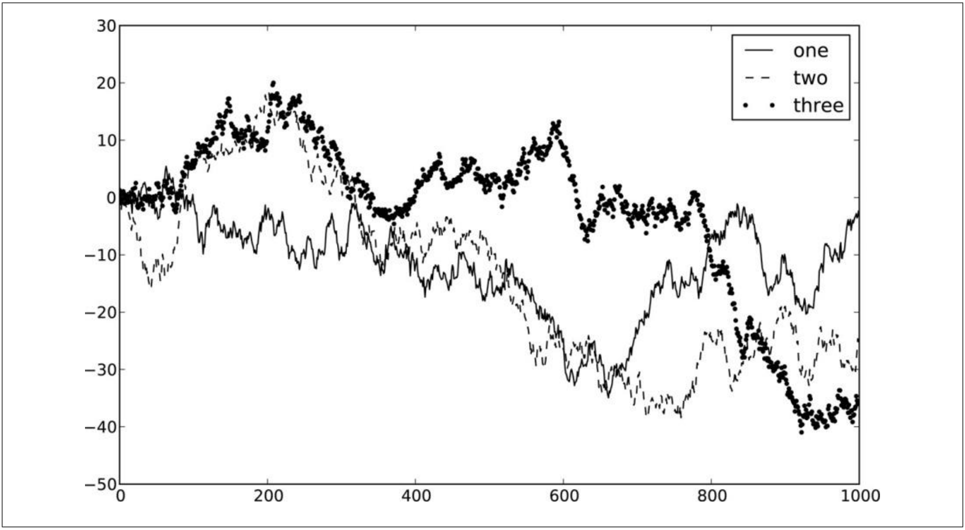


Figure 8-10. Simple plot with 3 lines and legend

Adding legends

Legends are another critical element for identifying plot elements. There are a couple of ways to add one. The easiest is to pass the label argument when adding each piece of the plot:

```
In [40]: fig = plt.figure(); ax = fig.add_subplot(1, 1, 1)

In [41]: ax.plot(randn(1000).cumsum(), 'k', label='one')Out[41]:
[<matplotlib.lines.Line2D at 0x4720a90>]

In [42]: ax.plot(randn(1000).cumsum(), 'k--', label='two')Out[42]:
[<matplotlib.lines.Line2D at 0x4720f90>]

In [43]: ax.plot(randn(1000).cumsum(), 'k.', label='three')Out[43]:
[<matplotlib.lines.Line2D at 0x4723550>]
```

Once you've done this, you can either call ax.legend() or plt.legend() to automaticallycreate a legend:

```
In [44]: ax.legend(loc='best')
```

See Figure 8-10. The loc tells matplotlib where to place the plot. If you aren't picky 'best' is a good option, as it will choose a location that is most out of the way. To exclude one or more elements from the legend, pass no label or label='_nolegend_'.

Annotations and Drawing on a Subplot

In addition to the standard plot types, you may wish to draw your own plot annotations, which could consist of text, arrows, or other shapes.

Annotations and text can be added using the text, arrow, and annotate functions. textdraws text at given coordinates (x, y)on the plot with optional custom styling:

```
ax.text(x, y, 'Hello world!', family='monospace',
        fontsize=10)
```

Annotations can draw both text and arrows arranged appropriately. As an example, let's plot the closing S&P 500 index price since 2007 (obtained from Yahoo! Finance) and annotate it with some of the important dates from the 2008-2009 financial crisis. See Figure 8-11 for the result:

```
from datetime import datetime

fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)

data = pd.read_csv('ch08/spx.csv', index_col=0, parse_dates=True)spx =
data['SPX']

spx.plot(ax=ax, style='k-')

crisis_data = [
    (datetime(2007, 10, 11), 'Peak of bull market'),
    (datetime(2008, 3, 12), 'Bear Stearns Fails'),
    (datetime(2008, 9, 15), 'Lehman Bankruptcy')
]

for date, label in crisis_data:
    ax.annotate(label, xy=(date, spx.asof(date) + 50),xytext=(date,
                spx.asof(date) + 200),
                arrowprops=dict(facecolor='black'),
                horizontalalignment='left', verticalalignment='top')

# Zoom in on 2007-2010
ax.set_xlim(['1/1/2007', '1/1/2011'])
ax.set_ylim([600, 1800])

ax.set_title('Important dates in 2008-2009 financial crisis')
```

See the online matplotlib gallery for many more annotation examples to learn from.

Drawing shapes requires some more care. matplotlib has objects that represent many common shapes, referred to as *patches*. Some of these, like Rectangle and Circle are found in matplotlib.pyplot, but the full set is located in matplotlib.patches.

To add a shape to a plot, you create the patch object shp and add it to a subplot by calling ax.add_patch(shp)(see Figure 8-12):

```
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)

rect = plt.Rectangle((0.2, 0.75), 0.4, 0.15, color='k', alpha=0.3)circ = plt.Circle((0.7,
0.2), 0.15, color='b', alpha=0.3)
      pgon = plt.Polygon([[0.15, 0.15], [0.35, 0.4], [0.2, 0.6]],
                         color='g', alpha=0.5)
```

```
ax.add_patch(rect)
ax.add_patch(circ)
ax.add_patch(pgon)
```

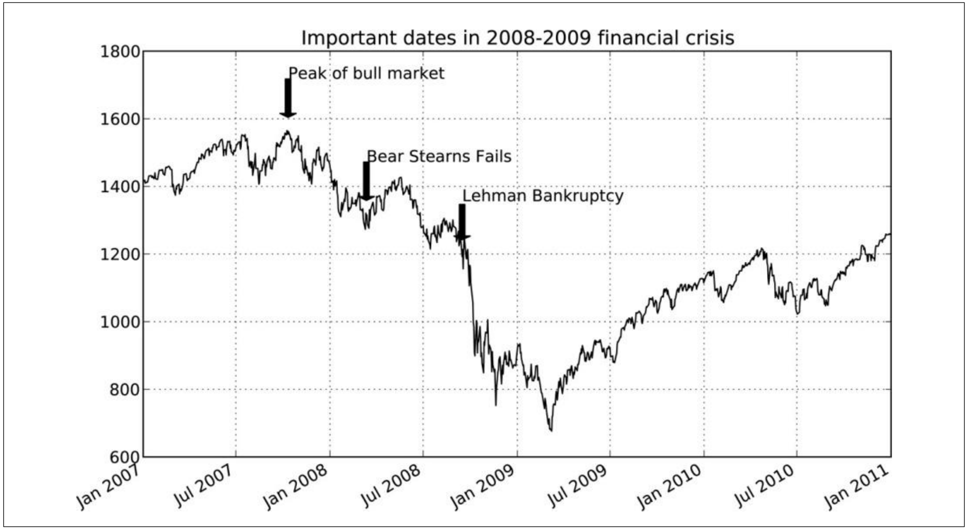


Figure 8-11. Important dates in 2008-2009 financial crisis

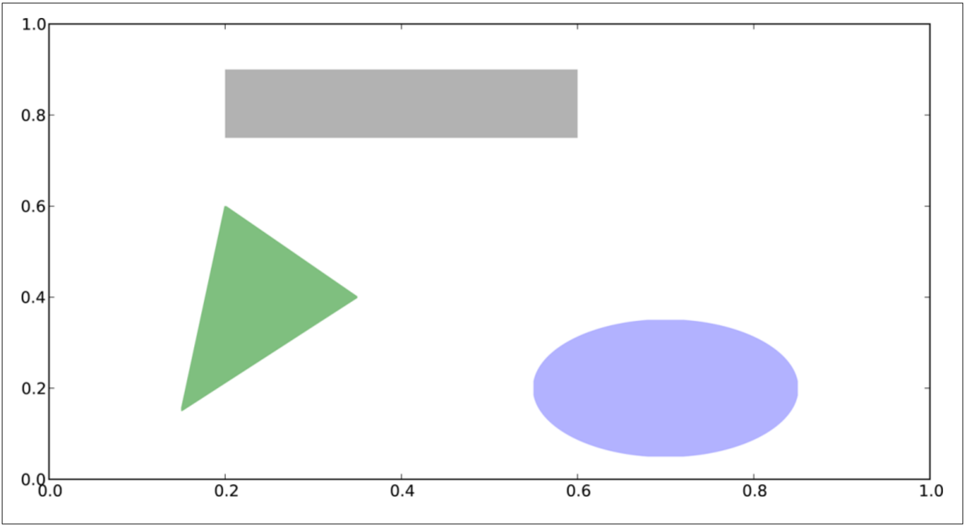


Figure 8-12. Figure composed from 3 different patches

If you look at the implementation of many familiar plot types, you will see that they are assembled from patches.

Saving Plots to File

The active figure can be saved to file using plt.savefig. This method is equivalent tothe figure object's savefig instance method. For example, to save an SVG version of a figure, you need only type:

```
plt.savefig('figpath.svg')
```

The file type is inferred from the file extension. So if you used .pdf instead you would get a PDF. There are a couple of important options that I use frequently for publishing graphics: dpi, which controls the dots-per-inch resolution, and bbox_inches, which can trim the whitespace around the actual figure. To get the same plot as a PNG above with minimal whitespace around the plot and at 400 DPI, you would do:

```
plt.savefig('figpath.png', dpi=400, bbox_inches='tight')
```

savefigdoesn't have to write to disk; it can also write to any file-like object, such as a StringIO:

```
from io import StringIO
buffer = StringIO()
plt.savefig(buffer)
plot_data = buffer.getvalue()
```

For example, this is useful for serving dynamically-generated images over the web.

Table 8-2. Figure.savefig options

Argument	Description
fname	String containing a filepath or a Python file-like object. The figure format is inferred from the fileextension, e.g. .pdffor PDF or .pngfor PNG.
dpi	The figure resolution in dots per inch; defaults to 100 out of the box but can be configured
facecolor, edge color	The color of the figure background outside of the subplots. 'w'(white), by default
format	The explicit file format to use ('png', 'pdf', 'svg', 'ps', 'eps', ...)
bbox_inches	The portion of the figure to save. If 'tight'is passed, will attempt to trim the empty space around the figure

matplotlib Configuration

matplotlib comes configured with color schemes and defaults that are geared primarily toward preparing figures for publication. Fortunately, nearly all of the default behavior can be customized via an extensive set of global parameters governing figure size, sub-plot spacing, colors, font sizes, grid styles, and so on. There are two main ways to interact with the matplotlib configuration system. The first is programmatically from Python using the rcmethod. For example, to set the global default figure size to be 10x 10, you could enter:

```
plt.rc('figure', figsize=(10, 10))
```

The first argument to rc is the component you wish to customize, such as 'figure', 'axes', 'xtick', 'ytick', 'grid', 'legend' or many others. After that can follow a sequence of keyword arguments indicating the new parameters. An easy way to write down the options in your program is as a dict:

```
font_options = {'family' : 'monospace',
                'weight' : 'bold',
                'size'   : 'small'}
plt.rc('font', **font_options)
```

For more extensive customization and to see a list of all the options, matplotlib comes with a configuration file matplotlibrc in the matplotlib/mpl-data directory. If you cus- tomize this file and place it in your home directory titled .matplotlibrc, it will be loadedeach time you use matplotlib.

Plotting Functions in pandas

As you've seen, matplotlib is actually a fairly low-level tool. You assemble a plot from its base components: the data display (the type of plot: line, bar, box, scatter, contour, etc.), legend, title, tick labels, and other annotations. Part of the reason for this is that in many cases the data needed to make a complete plot is spread across many objects. In pandas we have row labels, column labels, and possibly grouping information. This means that many kinds of fully-formed plots that would ordinarily require a lot of matplotlib code can be expressed in one or two concise statements. Therefore, pandas has an increasing number of high-level plotting methods for creating standard visual- izations that take advantage of how data is organized in DataFrame objects.

As of this writing, the plotting functionality in pandas is undergoing quite a bit of work. As part of the 2012 Google Summer of Code pro- gram, a student is working full time to add features and to make the interface more consistent and usable. Thus, it's possible that this code may fall out-of-date faster than the other things in this book. The online pandas documentation will be the best resource in that event.

Line Plots

Series and DataFrame each have a plot method for making many different plot types. By default, they make line plots (see Figure 8-13):

```
In [55]: s = Series(np.random.randn(10).cumsum(), index=np.arange(0, 100, 10))

In [56]: s.plot()
```

The Series object's index is passed to matplotlib for plotting on the X axis, though this can be disabled by passing use_index=False. The X axis ticks and limits can be adjusted using the xticks and xlim options, and Y axis respectively using yticks and ylim. See

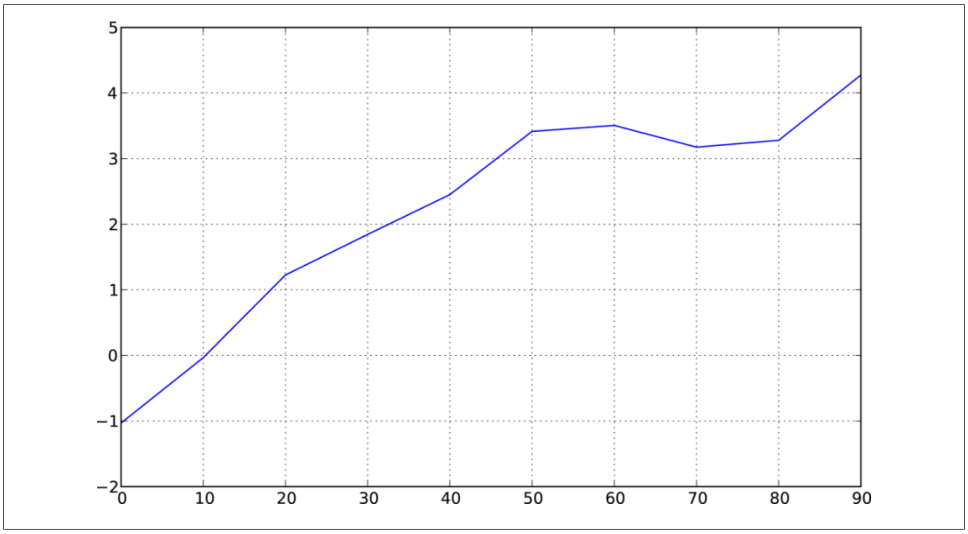


Figure 8-13. Simple Series plot example

Table 8-3 for a full listing of plot options. I'll comment on a few more of them throughout this section and leave the rest to you to explore.

Most of pandas's plotting methods accept an optional ax parameter, which can be a matplotlib subplot object. This gives you more flexible placement of subplots in a grid layout. There will be more on this in the later section on the matplotlib API.

DataFrame's plot method plots each of its columns as a different line on the same subplot, creating a legend automatically (see Figure 8-14):

```
In [57]: df = DataFrame(np.random.randn(10, 4).cumsum(0),
   ....:                columns=['A', 'B', 'C', 'D'],
   ....:                index=np.arange(0, 100, 10))

In [58]: df.plot()
```

Additional keyword arguments to plot are passed through to the respective matplotlib plotting function, so you can further customize these plots by learning more about the matplotlib API.

Table 8-3. Series.plot method arguments

Argument	Description
label	Label for plot legend
ax	matplotlib subplot object to plot on. If nothing passed, uses active matplotlib subplot
style	Style string, like 'ko--', to be passed to matplotlib.
alpha	The plot fill opacity (from 0 to 1)

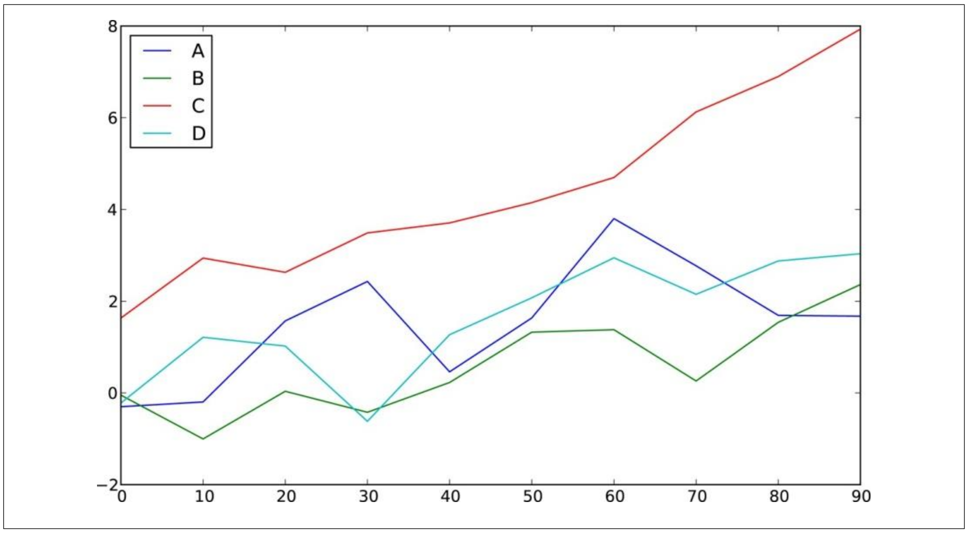


Figure 8-14. Simple DataFrame plot example

Argument	Description
kind	Can be 'line', 'bar', 'barh', 'kde'logy Use
logarithmic scaling on the Y axis use_index	Use the object index
for tick labels	
rot	Rotation of tick labels (0 through 360)
xticks	Values to use for X axis ticks yticks
Values to use for Y axis ticks xlim	X axis limits
(e.g. [0, 10])	
ylim	Y axis limits
grid	Display axis grid (on by default)

DataFrame has a number of options allowing some flexibility with how the columns are handled; for example, whether to plot them all on the same subplot or to create separate subplots. See Table 8-4 for more on these.

Table 8-4. DataFrame-specific plot arguments

Argument	Description
subplots	Plot each DataFrame column in a separate subplot
sharex	If subplots=True, share the same X axis, linking ticks and limits
sharey	If subplots=True, share the same Y axis
figsize	Size of figure to create as tuple

Argument	Description
title	Plot title as string
legend	Add a subplot legend (True by default)
sort_columns	Plot columns in alphabetical order; by default uses existing column order

Bar Plots

Making bar plots instead of line plots is a simple as passing kind='bar' (for vertical bars) or kind='barh' (for horizontal bars). In this case, the Series or DataFrame index will be used as the X (bar) or Y (barh) ticks (see Figure 8-15):

```
In [59]: fig, axes = plt.subplots(2, 1)

In [60]: data = Series(np.random.rand(16), index=list('abcdefghijklmnop'))In [61]:

data.plot(kind='bar', ax=axes[0], color='k', alpha=0.7)
Out[61]: <matplotlib.axes.AxesSubplot at 0x4ee7750>

In [62]: data.plot(kind='barh', ax=axes[1], color='k', alpha=0.7)
```

For more on the plt.subplots function and matplotlib axes and figures,see the later section in this chapter.

With a DataFrame, bar plots group the values in each row together in a group in bars, side by side, for each value. See Figure 8-16:

```
In [63]: df = DataFrame(np.random.rand(6, 4),
    ....:                    index=['one', 'two', 'three', 'four', 'five', 'six'],
    ....:                    columns=pd.Index(['A', 'B', 'C', 'D'], name='Genus'))

In [64]: df
Out[64]:
Genus          A          B          C
        Done   0.301686   0.156333   0.371943
0.270731
two     0.750589   0.525587   0.689429   0.358974
three   0.381504   0.667707   0.473772   0.632528
four    0.942408   0.180186   0.708284   0.641783
five    0.840278   0.909589   0.010041   0.653207
six     0.062854   0.589813   0.811318   0.060217

In [65]: df.plot(kind='bar')
```

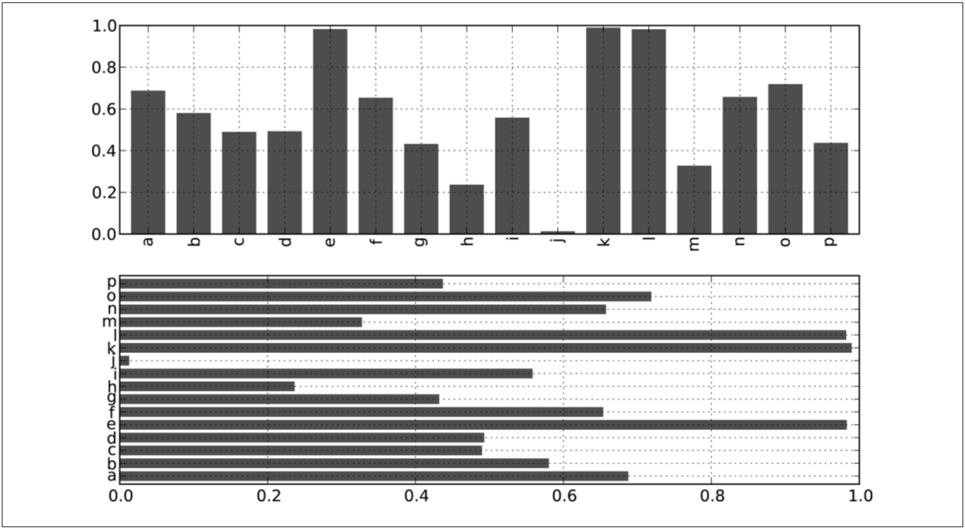


Figure 8-15. Horizonal and vertical bar plot example

Note that the name “Genus” on the DataFrame’s columns is used to title the legend.

Stacked bar plots are created from a DataFrame by passing stacked=True, resulting in the value in each row being stacked together (see Figure 8-17):

```
In [67]: df.plot(kind='barh', stacked=True, alpha=0.5)
```

A useful recipe for bar plots (as seen in an earlier chapter) is to visualize a Series’s value frequency using value_counts: s.value_counts ().plot(kind='bar')

Returning to the tipping data set used earlier in the book, suppose we wanted to make a stacked bar plot showing the percentage of data points for each party size on each day. I load the data using read_csvand make a cross-tabulation by day and party size:

```
In [68]: tips = pd.read_csv('ch08/tips.csv')

In [69]: party_counts = pd.crosstab(tips.day, tips.size)In [70]:

party_counts
Out[70]:
size  1     2    3     4  5  6
day
 Fri    1  16    1     1  0  0
 Sat    2  53   18    13  1  0
 Sun    0  39   15    18  3  1
 Thur  1  48     4     5  1  3
```

```
# Not many 1- and 6-person parties
In [71]: party_counts = party_counts.ix[:, 2:5]
```

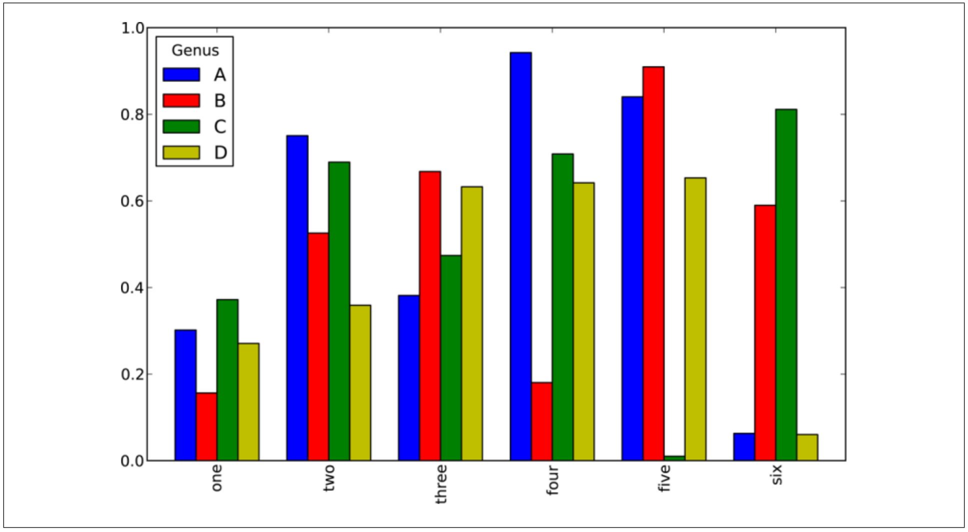


Figure 8-16. DataFrame bar plot example

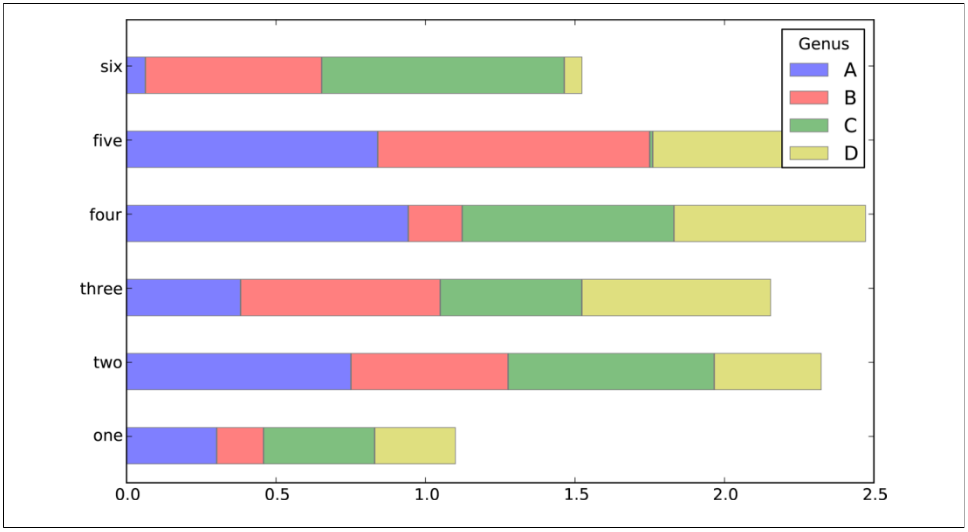


Figure 8-17. DataFrame stacked bar plot example

Then, normalize so that each row sums to 1 (I have to cast to float to avoid integer division issues on Python 2.7) and make the plot (see Figure 8-18):

```
# Normalize to sum to 1
In [72]: party_pcts = party_counts.div(party_counts.sum(1).astype(float), axis=0)
In [73]: party_pcts
Out[73]:
size          2         3         4         5
day
Fri     0.888889  0.055556  0.055556  0.000000
Sat     0.623529  0.211765  0.152941  0.011765
Sun     0.520000  0.200000  0.240000  0.040000
Thur   0.827586  0.068966  0.086207  0.017241

In [74]: party_pcts.plot(kind='bar', stacked=True)
```

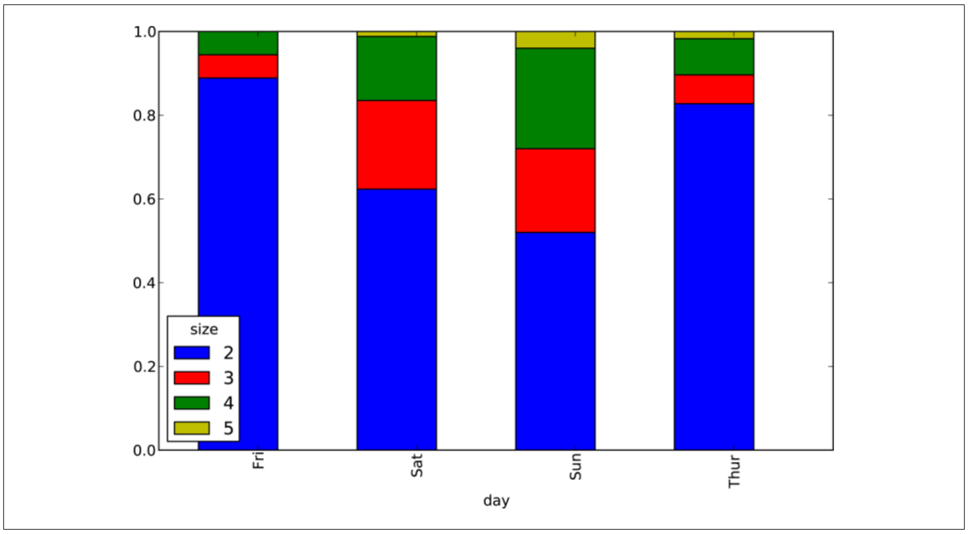


Figure 8-18. Fraction of parties by size on each day

So you can see that party sizes appear to increase on the weekend in this data set.

Histograms and Density Plots

A histogram, with which you may be well-acquainted, is a kind of bar plot that gives a discretized display of value frequency. The data points are split into discrete, evenly spaced bins, and the number of data points in each bin is plotted. Using the tipping data from before, we can make a histogram of tip percentages of the total bill using the histmethod on the Series (see Figure 8-19):

```
In [76]: tips['tip_pct'] = tips['tip'] / tips['total_bill']In [77]:

tips['tip_pct'].hist(bins=50)
```

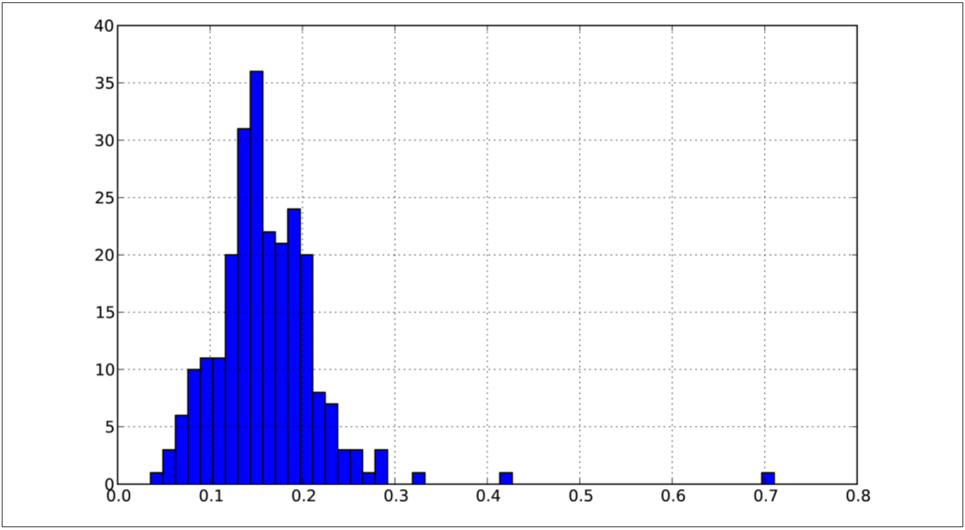


Figure 8-19. Histogram of tip percentages

A related plot type is a *density plot*, which is formed by computing an estimate of a continuous probability distribution that might have generated the observed data. A usual procedure is to approximate this distribution as a mixture of kernels, that is, simpler distributions like the normal (Gaussian) distribution. Thus, density plots are also known as KDE (kernel density estimate) plots. Using plot with kind='kde' makes a density plot using the standard mixture-of-normals KDE (see Figure 8-20):

```
In [79]: tips['tip_pct'].plot(kind='kde')
```

These two plot types are often plotted together; the histogram in normalized form (to give a binned density) with a kernel density estimate plotted on top. As an example, consider a bimodal distribution consisting of draws from two different standard normal distributions (see Figure 8-21):

```
In [81]: comp1 = np.random.normal(0, 1, size=200)   # N(0, 1) In [82]:

comp2 = np.random.normal(10, 2, size=200)   # N(10, 4)In [83]: values =

Series(np.concatenate([comp1, comp2]))

In [84]: values.hist(bins=100, alpha=0.3, color='k', normed=True)Out[84]:
<matplotlib.axes.AxesSubplot at 0x5cd2350>

In [85]: values.plot(kind='kde', style='k--')
```

Scatter Plots

Scatter plots are a useful way of examining the relationship between two one-dimensional data series. matplotlib has a scatter plotting method that is the workhorse of

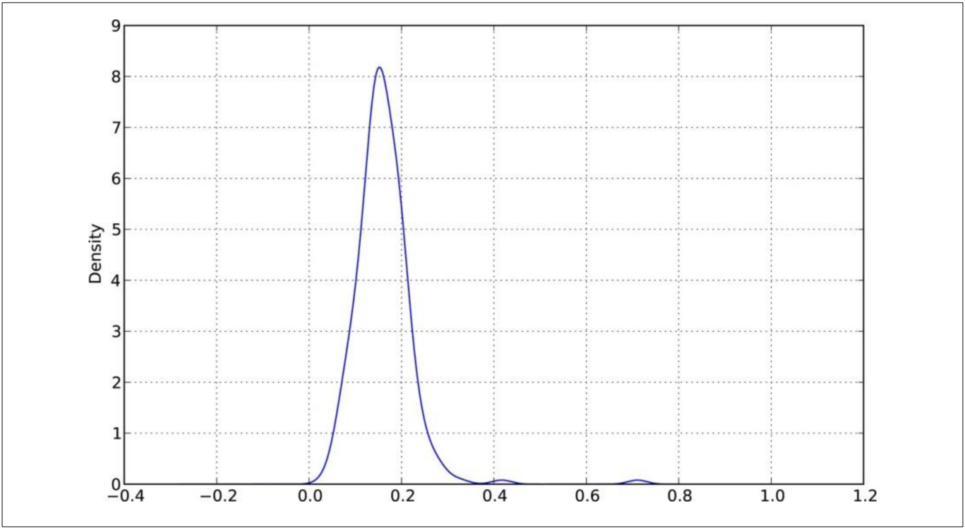


Figure 8-20. Density plot of tip percentages

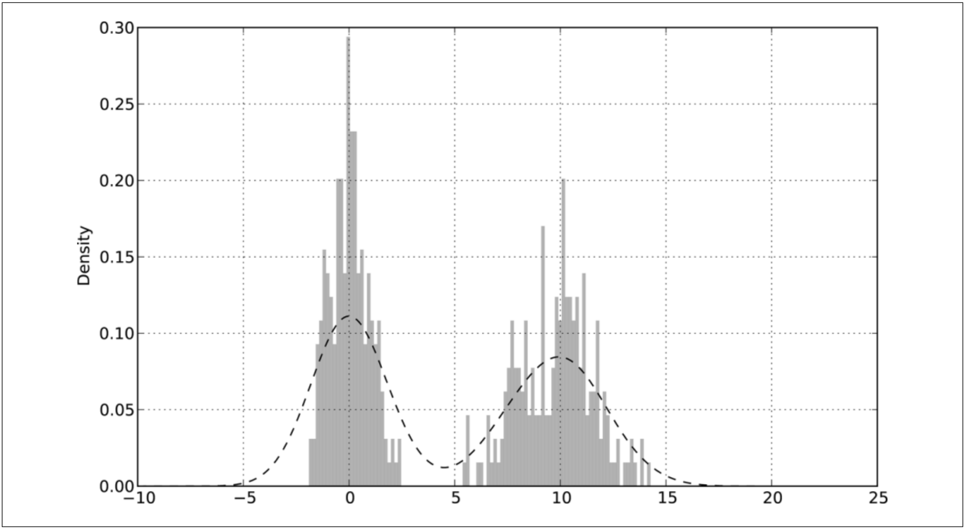


Figure 8-21. Normalized histogram of normal mixture with density estimate

making these kinds of plots. To give an example, I load the macrodatadataset from the statsmodels project, select a few variables, then compute log differences:

```
In [86]: macro = pd.read_csv('ch08/macrodata.csv')

In [87]: data = macro[['cpi', 'm1', 'tbilrate', 'unemp']]In [88]: trans_data

= np.log(data).diff().dropna()
```

```
In [89]: trans_data[-5:]
Out[89]:
              cpi             m1   tbilrate
                              unemp198 -0.007904
0.045361 -0.396881   0.105361
199 -0.021979   0.066753 -2.277267   0.139762
200   0.002340   0.010286   0.606136   0.160343
201   0.008419   0.037461 -0.200671   0.127339
202   0.008894   0.012202 -0.405465   0.042560
```

It's easy to plot a simple scatter plot using plt.scatter (see Figure 8-22):

```
In [91]: plt.scatter(trans_data['m1'], trans_data['unemp']) Out[91]:
<matplotlib.collections.PathCollection at 0x43c31d0>

In [92]: plt.title('Changes in log %s vs. log %s' % ('m1', 'unemp'))
```

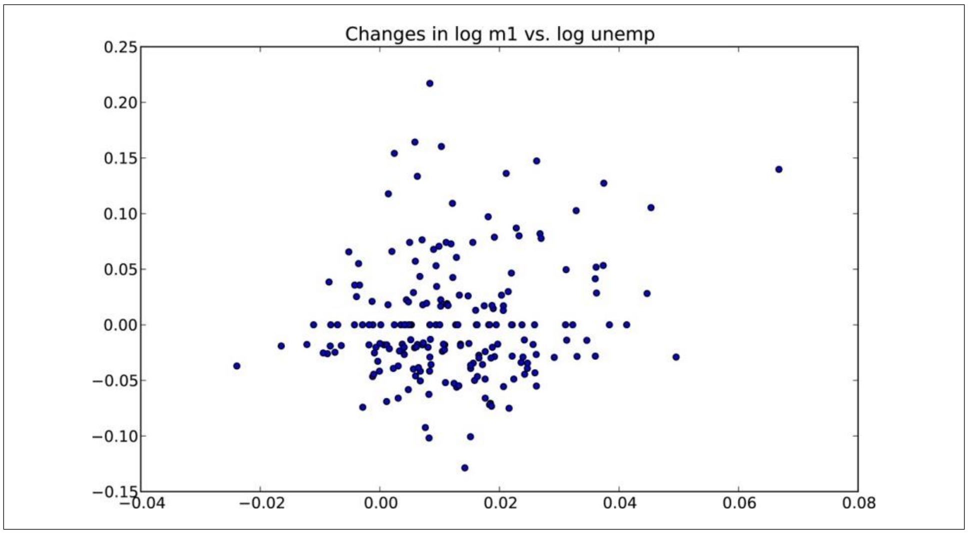


Figure 8-22. A simple scatter plot

In exploratory data analysis it's helpful to be able to look at all the scatter plots among a group of variables; this is known as a *pairs* plot or *scatter plot matrix*. Making such a plot from scratch is a bit of work, so pandas has a scatter_matrix function for creating one from a DataFrame. It also supports placing histograms or density plots of each variable along the diagonal. See Figure 8-23 for the resulting plot:

```
In [93]: scatter_matrix(trans_data, diagonal='kde', color='k', alpha=0.3)
```