Python Data Analytics

DATA ANALYSIS AND SCIENCE USING
PANDAS, MATPLOTLIB AND THE PYTHON
PROGRAMMING LANGUAGE

Fabio Nelli

Apress®

Python Data Analytics

Data Analysis and Science Using
Pandas, matplotlib, and the
Python Programming Language

Fabio Nelli

Apress®

Python Data Analytics

Contents at a Glance

Contents

About the Author

Fabio Nelli is an IT Scientific Application Specialist at IRBM Science Park, a private research center in Pomezia, Roma (Italy). He has been a computer consultant for many years at IBM, EDS, Merck Sharp, and Dohme, along with several banks and insurance companies.

He has an Organic Chemistry degree and many years of experience in Information Technologies and Automation Systems applied to Life Sciences (Tech Specialist at Beckman Coulter Italy and Spain).

He is currently developing Java applications that interface Oracle databases with scientific instrumentations, generating data and Web server applications and providing analysis of the results to researchers in real time.

Moreover, he is the coordinator of the Meccanismo Complesso community (www.meccanismocomplesso.org).

About the Technical Reviewer

Shubham Singh Tomar is a Data Engineer at `Predikt.co`. He lives and works in Bangalore, India. On weekends he volunteers to work on Data Science projects for NGOs and social organizations with the Bangalore chapter of . He writes about Python, Data Science and Machine Learning on his blog: `shubhamtomar.me`.

Acknowledgments

I'd like to thank my friends, particularly Alberto, Daniele, Roberto, and Alex for putting up with me and providing much-needed moral support through a year of difficulties and frustration.

Deepest thanks to my mother.

■ ■ ■

An Introduction to Data Analysis

With this chapter, you will begin to take the first steps in the world of data analysis, seeing in detail all the concepts and processes that make up this discipline. The concepts discussed in this chapter will be helpful background for the following chapters, where these concepts and procedures will be applied in the form of Python code, through the use of several libraries that will be discussed in just as many chapters.

Data Analysis

In a world increasingly centralized around information technology, huge amounts of data are produced and stored each day. Often these data come from automatic detection systems, sensors, and scientific instrumentation, or you produce them daily and unconsciously every time you make a withdrawal from the bank or make a purchase, when you record on various blogs, or even when you post on social networks.

But what are the data? The data actually are not information, at least in terms of their form. In the formless stream of bytes, at first glance it is difficult to understand their essence if not strictly the number, word, or time that they report. Information is actually the result of processing, which taking into account a certain set of data, extracts some conclusions that can be used in various ways. This process of extracting information from the raw data is precisely **data analysis**.

The purpose of data analysis is precisely to extract information that is not easily deducible but that, when understood, leads to the possibility of carrying out studies on the mechanisms of the systems that have produced them, thus allowing the possibility of making forecasts of possible responses of these systems and their evolution in time.

Starting from a simple methodical approach on data protection, data analysis has become a real discipline leading to the development of real methodologies generating **models**. The model is in fact the translation into a mathematical form of a system placed under study. Once there is a mathematical or logical form able to describe system responses under different levels of precision, you can then make predictions about its development or response to certain inputs. Thus the aim of data analysis is not the model, but the goodness of its **predictive power**.

The predictive power of a model depends not only on the quality of the modeling techniques but also on the ability to choose a good dataset upon which to build the entire data analysis. So the **search for data**, their **extraction**, and their subsequent **preparation**, while representing preliminary activities of an analysis, also belong to the data analysis itself, because of their importance in the success of the results.

So far we have spoken of data, their handling, and their processing through calculation procedures. In parallel to all stages of processing of the data analysis, various methods of **data visualization** have been developed. In fact, to understand the data, both individually and in terms of the role they play in the entire data set, there is no better system than to develop the techniques of graphic representation capable of transforming information, sometimes implicitly hidden, in figures, which help you more easily understand their meaning. Over the years lots of display modes have been developed for different modes of data display: the **charts**.

At the end of the data analysis, you will have a model and a set of graphical displays and then you will be able to predict the responses of the system under study; after that, you will move to the test phase. The model will be tested using another set of data for which we know the system response. These data are, however, not used for the definition of the predictive model. Depending on the ability of the model to replicate real observed responses, you will have an error calculation and a knowledge of the validity of the model and its operating limits.

These results can be compared with any other models to understand if the newly created one is more efficient than the existing ones. Once you have assessed that, you can move to the last phase of data analysis—the **deployment**. This consists of the implementation of the results produced by the data analysis, namely, the implementation of the decisions to be taken based on the predictions generated by the model and the risks that such a decision will also be predicted.

Data analysis is a discipline that is well suited to many professional activities. So, knowledge of what it is and how it can be put into practice will be relevant for consolidating the decisions to be made. It will allow us to test hypotheses, and to understand more deeply the systems analyzed.

Knowledge Domains of the Data Analyst

Data analysis is basically a discipline suitable to the study of problems that may occur in several fields of applications. Moreover, in processes of data analysis you have many tools and methodologies that require good knowledge of computing and mathematical and statistical concepts.

So a good data analyst must be able to move and act in many different disciplinary areas. Many of these disciplines are the basis of the methods of data analysis, and proficiency in them is almost necessary. Knowledge of other disciplines is necessary depending on the area of application and study of the particular data analysis project you are about to undertake, and, more generally, sufficient experience in these areas can just help you better understand the issues and the type of data needed to start with the analysis.

Often, regarding major problems of data analysis, it is necessary to have an interdisciplinary team of experts made up of members who are all able to contribute in the best possible way in their respective fields of competence. Regarding smaller problems, a good analyst must be able to recognize problems that arise during data analysis, inquire to find out which disciplines and skills are necessary to solve the problem, study these disciplines, and maybe even ask the most knowledgeable people in the sector. In short, the analyst must be able to know how to search not only for data, but also for information on how to treat them.

Computer Science

Knowledge of Computer Science is a basic requirement for any data analyst. In fact, only one who has good knowledge of and experience in Computer Science is able to efficiently manage the necessary tools for data analysis. In fact, all the steps concerning the data analysis involve the use of computer technology as calculation software (such as IDL, Matlab, etc.) and programming languages (such as C ++, Java, Python).

The large amount of data available today thanks to information technology requires specific skills in order to be managed as efficiently as possible. Indeed, data research and extraction require knowledge of the various formats. The data are structured and stored in files or database tables with particular formats. XML, JSON, or simply XLS or CSV files are now the common formats for storing and collecting data, and many applications also allow their reading and managing data stored on them. For the extraction of data contained in a database, things are not so immediate, but you need to know SQL query language or use software specially developed for the extraction of data from a given database.

Moreover, for some specific types of data research, the data are not available in a pre-treated and explicit format, but are present in text files (documents, log files) or in web pages, shown as charts, measures, number of visitors, or HTML tables that require specific technical expertise for the parsing and the eventual extraction of these data (**Web Scraping**).

So, knowledge of information technology is necessary to know how to use the various tools made available by contemporary computer science, such as applications and programming languages. These tools, in turn, are needed to perform the data analysis and data visualization.

The purpose of this book is precisely to provide all the necessary knowledge, as far as possible, regarding the development of methodologies for data analysis using Python as a programming language and specialized libraries that provide a decisive contribution to the performance of all the steps constituting the data analysis, from data research to data mining, up to getting to the publication of the results of the predictive model.

Mathematics and Statistics

As you will see throughout the book, data analysis requires a lot of math, which can be quite complex, during the treatment and processing of data. So competence in all of this is necessary, at least to understand what you are doing. Some familiarity with the main statistical concepts is also necessary because all the methods that are applied in the analysis and interpretation of data are based on these concepts. Just as you can say that computer science gives you the tools for data analysis, so you can say that the statistics provides the concepts that form the basis of the data analysis.

Many are the tools and methods that this discipline provides to the analyst, and a good knowledge of how to best use them requires years of experience. Among the most commonly used statistical techniques in data analysis are

- Bayesian methods

- regression

- clustering

Having to deal with these cases, you'll discover how the mathematics and statistics are closely related to each other, but thanks to special Python libraries covered in this book, you will have the ability to manage and handle them.

Machine Learning and Artificial Intelligence

One of the most advanced tools that falls in the data analysis is Machine Learning. In fact, despite data visualization and techniques such as clustering and regression, which should greatly help us to find information about our data set, during this phase of research, you may often prefer to use special procedures which are highly specialized in searching patterns within the data set.

Machine Learning is a discipline that makes use of a whole series of procedures and algorithms which analyze the data in order to recognize patterns, clusters, or trends and then extract useful information for data analysis in a totally automated way.

This discipline is increasingly becoming a fundamental tool of data analysis, and thus knowledge of it, at least in general, is of fundamental importance for the data analyst.

Professional Fields of Application

Another very important point is also the domain of competence from where the data come (biology, physics, finance, materials testing, statistics on population, etc.). In fact, although the analyst has had specialized preparation in the field of statistics, he must also be able to delve into the field of application and/or document the source of the data, with the aim of perceiving and better understanding the mechanisms that generated data. In fact, the data are not simple strings or numbers, but they are the expression, or rather the

measure, of any parameter observed. Thus, better understanding of the field of application where the data come from can improve their interpretation. Often, however, this is too costly for a data analyst, even one with the best intentions, and so it is good practice to find consultants or key figures to whom you can pose the right questions.

Understanding the Nature of the Data

The object of study of the data analysis is basically the data. The data then will be the key players in all processes of the data analysis. They constitute the raw material to be processed, and thanks to their processing and analysis it is possible to extract a variety of information in order to increase the level of knowledge of the system under study, that is, one from which the data came from.

When the Data Become Information

Data are the events recorded in the world. Anything that can be measured or even categorized can be converted into data. Once collected, these data can be studied and analyzed both to understand the nature of the events and very often also to make predictions or at least to make informed decisions.

When the Information Becomes Knowledge

You can speak of knowledge when the information is converted into a set of rules that help you to better understand certain mechanisms and so consequently, to make predictions on the evolution of some events.

Types of Data

The data can be divided into two distinct categories:

- categorical
  - nominal
  - ordinal
- numerical
  - discrete
  - continuous

Categorical data are values or observations that can be divided into groups or categories. There are two types of categorical values: **nominal** and **ordinal**. A nominal variable has no intrinsic order that is identified in its category. An ordinal variable instead has a predetermined order.

Numerical data are values or observations that come from measurements. There are two types of different numerical values: **discrete** and **continuous** numbers. Discrete values are values that can be counted and that are distinct and separated from each other. Continuous values, on the other hand, are values produced by measurements or observations that assume any value within a defined range.

The Data Analysis Process

Data analysis can be described as a process consisting of several steps in which the raw data are transformed and processed in order to produce data visualizations and can make predictions thanks to a mathematical model based on the collected data. Then, data analysis is nothing more than a sequence of steps, each of which plays a key role in the subsequent ones. So, data analysis is almost schematized as a process chain consisting of the following sequence of stages:

- Problem definition

- Data extraction

- Data cleaning

- Data transformation

- Data exploration

- Predictive modeling

- Model validation/test

- Visualization and interpretation of results

- Deployment of the solution

Figure 1-1 is a schematic representation of all the processes involved in the data analysis.
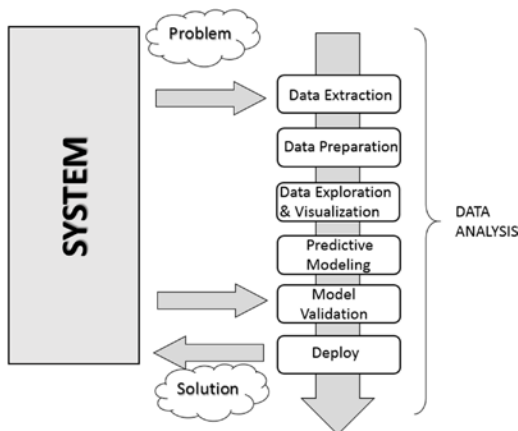
Figure 1-1. *The data analysis process*

Problem Definition

The process of data analysis actually begins long before the collection of raw data. In fact, a data analysis always starts with a problem to be solved, which needs to be defined.

The problem is defined only after you have well-focused the system you want to study: this may be a mechanism, an application, or a process in general. Generally this study can be in order to better understand its operation, but in particular the study will be designed to understand the principles of its behavior in order to be able to make predictions, or to make choices (defined as an informed choice).

The definition step and the corresponding documentation (*deliverables*) of the scientific problem or business are both very important in order to focus the entire analysis strictly on getting results. In fact, a comprehensive or exhaustive study of the system is sometimes complex and you do not always have enough information to start with. So the definition of the problem and especially its planning can determine uniquely the guidelines to follow for the whole project.

Once the problem has been defined and documented, you can move to the **project planning** of a data analysis. Planning is needed to understand which professionals and resources are necessary to meet the requirements to carry out the project as efficiently as possible. So you're going to consider the issues in the area involving the resolution of the problem. You will look for specialists in various areas of interest and finally install the software needed to perform the data analysis.

Thus, during the planning phase, the choice of an effective team takes place. Generally, these teams should be cross-disciplinary in order to solve the problem by looking at the data from different perspectives. So, the choice of a good team is certainly one of the key factors leading to success in data analysis.

Data Extraction

Once the problem has been defined, the first step is to obtain the data in order to perform the analysis. The data must be chosen with the basic purpose of building the predictive model, and so their selection is crucial for the success of the analysis as well. The sample data collected must reflect as much as possible the real world, that is, how the system responds to stimuli from the real world. In fact, even using huge data sets of raw data, often, if they are not collected competently, these may portray false or unbalanced situations compared to the actual ones.

Thus, a poor choice of data, or even performing analysis on a data set which is not perfectly representative of the system, will lead to models that will move away from the system under study.

The search and retrieval of data often require a form of intuition that goes beyond the mere technical research and data extraction. It also requires a careful understanding of the nature of the data and their form, which only good experience and knowledge in the problem's application field can give.

Regardless of the quality and quantity of data needed, another issue is the search and the correct choice of **data sources**.

If the studio environment is a laboratory (technical or scientific), and the data generated are experimental, then in this case the data source is easily identifiable. In this case, the problems will be only concerning the experimental setup.

But it is not possible for data analysis to reproduce systems in which data are gathered in a strictly experimental way in every field of application. Many fields of application require searching for data from the surrounding world, often relying on experimental data external, or even more often collecting them through interviews or surveys. So in these cases, the search for a good data source that is able to provide all the information you need for data analysis can be quite challenging. Often it is necessary to retrieve data from multiple data sources to supplement any shortcomings, to identify any discrepancies, and to make our data set as general as possible.

When you want to get the data, a good place to start is just the Web. But most of the data on the Web can be difficult to capture; in fact, not all data are available in a file or database, but can be more or less implicitly content that is inside HTML pages in many different formats. To this end, a methodology called **Web Scraping**, which allows the collection of data through the recognition of specific occurrence of HTML tags within the web pages, has been developed. There are software specifically designed for this purpose, and once an occurrence is found, they extract the desired data. Once the search is complete, you will get a list of data ready to be subjected to the data analysis.

Data Preparation

Among all the steps involved in data analysis, data preparation, though seemingly less problematic, is in fact one that requires more resources and more time to be completed. The collected data are often collected from different data sources, each of which will have the data in it with a different representation and format. So, all of these data will have to be prepared for the process of data analysis.

The preparation of the data is concerned with obtaining, cleaning, normalizing, and transforming data into an optimized data set, that is, in a prepared format, normally tabular, suitable for the methods of analysis that have been scheduled during the design phase.

Many are the problems that must be avoided, such as invalid, ambiguous, or missing values, replicated fields, or out-of-range data.

Data Exploration/Visualization

Exploring the data is essentially the search for data in a graphical or statistical presentation in order to find patterns, connections, and relationships in the data. Data visualization is the best tool to highlight possible patterns.

In recent years, data visualization has been developed to such an extent that it has become a real discipline in itself. In fact, numerous technologies are utilized exclusively for the display of data, and equally many are the types of display applied to extract the best possible information from a data set.

Data exploration consists of a preliminary examination of the data, which is important for understanding the type of information that has been collected and what they mean. In combination with the information acquired during the definition problem, this categorization will determine which method of data analysis will be most suitable for arriving at a model definition.

Generally, this phase, in addition to a detailed study of charts through the visualization data, may consist of one or more of the following activities:

- Summarizing data

- Grouping data

- Exploration of the relationship between the various attributes

- Identification of patterns and trends

- Construction of regression models

- Construction of classification models

Generally, the data analysis requires processes of summarization of statements regarding the data to be studied. The **summarization** is a process by which data are reduced to interpretation without sacrificing important information.

Clustering is a method of data analysis that is used to find groups united by common attributes (**grouping**).

Another important step of the analysis focuses on the **identification** of relationships, trends, and anomalies in the data. In order to find out this kind of information, one often has to resort to the tools as well as performing another round of data analysis, this time on the data visualization itself.

Other methods of data mining, such as decision trees and association rules, automatically extract important facts or rules from data. These approaches can be used in parallel with the data visualization to find information about the relationships between the data.

Predictive Modeling

Predictive modeling is a process used in data analysis to create or choose a suitable statistical model to predict the probability of a result.

After exploring data you have all the information needed to develop the mathematical model that encodes the relationship between the data. These models are useful for understanding the system under study, and in a specific way they are used for two main purposes. The first is to make predictions about the data values produced by the system; in this case, you will be dealing with **regression models**. The second is to classify new data products, and in this case, you will be using **classification models** or **clustering models**. In fact, it is possible to divide the models according to the type of result that they produce:

- **Classification models**: If the result obtained by the model type is categorical.

- **Regression models**: If the result obtained by the model type is numeric.

- **Clustering models**: If the result obtained by the model type is descriptive.

Simple methods to generate these models include techniques such as linear regression, logistic regression, classification and regression trees, and k-nearest neighbors. But the methods of analysis are numerous, and each has specific characteristics that make it excellent for some types of data and analysis. Each of these methods will produce a specific model, and then their choice is relevant for the nature of the product model.

Some of these models will provide values corresponding to the real system, and also according to their structure they will explain some characteristics of the system under study in a simple and clear way. Other models will continue to give good predictions, but their structure will be no more than a "black box" with limited ability to explain some characteristics of the system.

Model Validation

Validation of the model, that is, the test phase, is an important phase that allows you to validate the model built on the basis of starting data. That is important because it allows you to assess the validity of the data produced by the model by comparing them directly with the actual system. But this time, you are coming out from the set of starting data on which the entire analysis has been established.

Generally, you will refer to the data as the **training set**, when you are using them for building the model, and as the **validation set**, when you are using them for validating the model.

Thus, by comparing the data produced by the model with those produced by the system you will be able to evaluate the error, and using different test datasets, you can estimate the limits of validity of the generated model. In fact the correctly predicted values could be valid only within a certain range, or have different levels of matching depending on the range of values taken into account.

This process allows you not only to numerically evaluate the effectiveness of the model but also to compare it with any other existing models. There are several techniques in this regard; the most famous is the **cross-validation**. This technique is based on the division of the training set into different parts. Each of these parts, in turn, will be used as the validation set and any other as the training set. In this iterative manner, you will have an increasingly perfected model.

Deployment

This is the final step of the analysis process, which aims to present the results, that is, the conclusions of the analysis. In the deployment process, in the business environment, the analysis is translated into a benefit for the client who has commissioned it. In technical or scientific environments, it is translated into design solutions or scientific publications. That is, the deployment basically consists of putting into practice the results obtained from the data analysis.

There are several ways to deploy the results of a data analysis or data mining. Normally, a data analyst's deployment consists in writing a report for management or for the customer who requested the analysis. This document will conceptually describe the results obtained from the analysis of data. The report should be directed to the managers, who are then able to make decisions. Then, they will really put into practice the conclusions of the analysis.

In the documentation supplied by the analyst, each of these four topics will generally be discussed in detail:

- Analysis results

- Decision deployment

- Risk analysis

- Measuring the business impact

When the results of the project include the generation of predictive models, these models can be deployed as a stand-alone application or can be integrated within other software.

Quantitative and Qualitative Data Analysis

Data analysis is therefore a process completely focused on data, and, depending on the nature of the data, it is possible to make some distinctions.

When the analyzed data have a strictly numerical or categorical structure, then you are talking about **quantitative analysis**, but when you are dealing with values that are expressed through descriptions in natural language, then you are talking about **qualitative analysis**.

Precisely because of the different nature of the data processed by the two types of analyses, you can observe some differences between them.

Quantitative analysis has to do with data that have a logical order within them, or that can be categorized in some way. This leads to the formation of structures within the data. The order, categorization, and structures in turn provide more information and allow further processing of the data in a more strictly mathematical way. This leads to the generation of models can provide **quantitative predictions**, thus allowing the data analyst to draw **more objective conclusions**.

Qualitative analysis instead has to do with data that generally do not have a structure, at least not one that is evident, and their nature is neither numeric nor categorical. For example, data for the qualitative study could include written textual, visual, or audio data. This type of analysis must therefore be based on methodologies, often *ad hoc,* to extract information that will generally lead to models capable of providing **qualitative predictions**, with the result that the conclusions to which the data analyst can arrive may also include **subjective interpretations**. On the other hand, qualitative analysis can explore more complex systems and draw conclusions which are not possible with a strictly mathematical approach. Often this type of analysis involves the study of systems such as social phenomena or complex structures which are not easily measurable.

Figure 1-2 shows the differences between the two types of analysis:
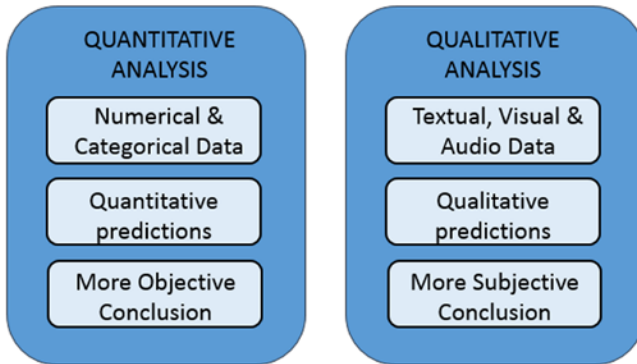
Figure 1-2. *Quantitative and qualitative analyses*

Open Data

In support of the growing demand for data, a huge number of data sources are now available in Internet. These data sources provide information freely to anyone in need, and they are called **Open Data**.

Here is a list of some Open Data available online. You can find a more complete list and details of the Open Data available online in Appendix B.

- DataHub (http://datahub.io/dataset)

- World Health Organization (http://www.who.int/research/en/)

- Data.gov (http://data.gov)

- European Union Open Data Portal (http://open-data.europa.eu/en/data/)

- Amazon Web Service public datasets (http://aws.amazon.com/datasets)

- Facebook Graph (http://developers.facebook.com/docs/graph-api)

- Healthdata.gov (http://www.healthdata.gov)

- Google Trends (http://www.google.com/trends/explore)

- Google Finance (https://www.google.com/finance)

- Google Books Ngrams (http://storage.googleapis.com/books/ngrams/books/datasetsv2.html)

- Machine Learning Repository (http://archive.ics.uci.edu/ml/)

In this regard, to give an idea of open data sources available online, you can look at the **LOD cloud diagram** (http://lod-cloud.net), which displays all the connections of the data link between several open data sources currently available in the network (see Figure 1-3).
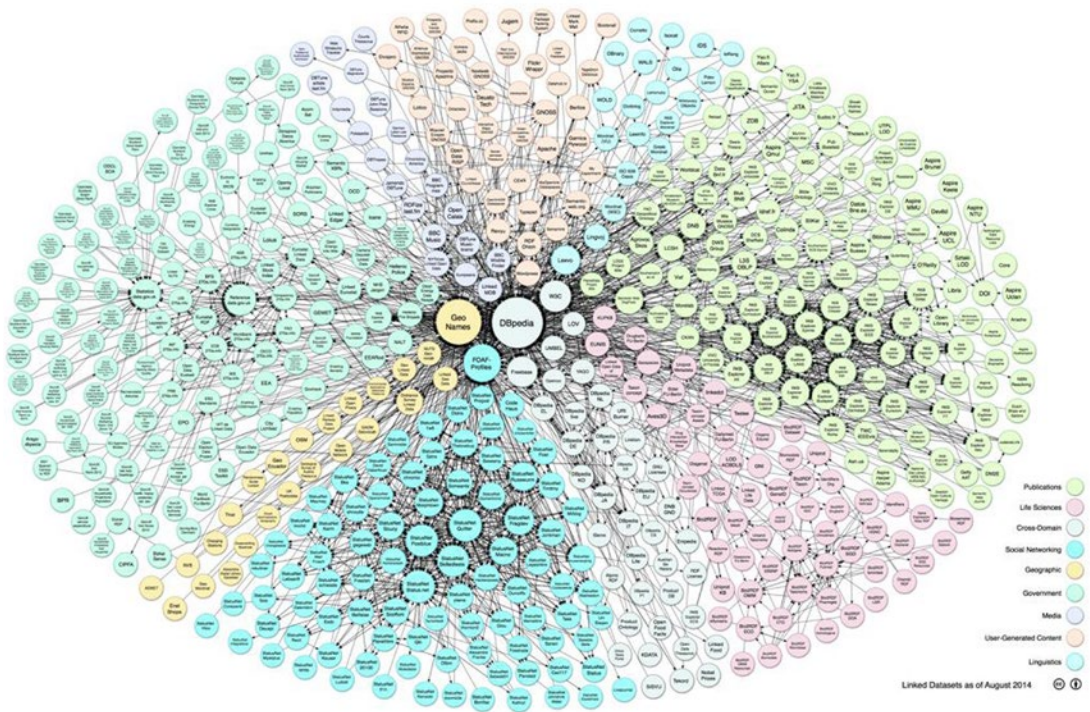
Figure 1-3. *Linking Open Data cloud diagram 2014, by Max Schmachtenberg, Christian Bizer, Anja Jentzsch, and Richard Cyganiak.* `http://lod-cloud.net/` *[CC-BY-SA license]*

Python and Data Analysis

The main argument of this book is to develop all the concepts of data analysis by treating them in terms of **Python**. Python is a programming language widely used in scientific circles because of its large number of libraries providing a complete set of tools for analysis and data manipulation.

Compared to other programming languages generally used for data analysis, such as **R** and **Matlab**, Python not only provides a platform for the processing of data but also has some features that make it unique compared to other languages and specialized applications. The development of an ever-increasing number of support libraries, the implementation of algorithms of more innovative methodologies, and the ability to interface with other programming languages (C and Fortran) make Python unique among its kind.

Furthermore, Python is not only specialized for the data analysis, but also has many other applications, such as generic programming, scripting, interfacing to databases, and more recently web development as well, thanks to web frameworks like Django. So it is possible to develop data analysis projects that are totally compatible with the Web Server with the possibility to integrate it on the Web.

So, for those who want to perform data analysis, Python, with all its packages, can be considered the best choice for the foreseeable future.

Conclusions

In this chapter, you saw what analysis of data is and, more specifically, the various processes that comprise it. Also, you have begun to see the role played by data in building a prediction model and how their careful selection is at the basis of a careful and accurate data analysis.

In the next chapter, take you will take this vision of Python and the tools it provides to perform data analysis.

CHAPTER 2

■ ■ ■

Introduction to the Python's World

The Python language, and the world around it, is made by interpreters, tools, editors, libraries, notebooks, etc. This Python's world has expanded greatly in recent years, enriching and taking forms that developers who approach for the first time can sometimes find to be complex and somewhat misleading. Thus if you are approaching the programming in Python for the first time, you might feel lost among so much choice, especially on where to start.

This chapter will give you an overview to the entire Python's world. First you will have a description of the Python language and its characteristics that made it unique. You'll see where to start, what an interpreter is, and how to begin to write the first lines of code in Python. Then you are presented with some new more advanced forms of interactive writing with respect to the shells such as IPython and IPython Notebook.

Python—The Programming Language

Python is a programming language created by Guido Von Rossum in 1991 starting with the previous language called ABC. This language can be characterized by a series of adjectives:

- interpreted

- portable

- object-oriented

- interactive

- interfaced

- open-source

- easy to understand and use

Python is a programming language **interpreted**, that is pseudo-compiled. Once you have written the code of a program, this in order to be run needs an **interpreter**. The interpreter is a program that is installed on each machine that has the task of interpreting the source code and run it. Therefore unlike language such as C, C ++, and Java, there is no compile time.

Python is a highly **portable** programming language. The decision to use an interpreter as an interface for reading and running the code has a key advantage: portability. In fact, you can install on any existing platform (Linux, Windows, Mac) an interpreter specifically adapted to it while the Python code to be interpreted will remain unchanged. Python also, for this aspect, was chosen as the programming language for many small-form devices, such as the Raspberry Pi and other microcontrollers.

Python is an **object-oriented** programming language. In fact, it allows you to specify classes of objects and implement their inheritance. But unlike C ++ and Java there are no constructors or destructors. Python also allows you to implement specific constructs in your code for manage exceptions. However, the structure of language is so flexible that allows to program with alternative approaches with respect to the object-oriented one, for example the functional or vectorial.

Python is an interactive programming language. Thanks to the fact that Python uses an interpreter to be executed, this language can take on very different aspects depending on the context in which it is used. In fact, you can write a code made up of a lot of lines, similar to what we would do in languages like C ++ or Java, and then launch the program, or you can enter a command line at once and execute it, immediately getting the results of the command, and depending on them you can decide what will be the next line of command to be run. This highly interactive mode to execute code makes Python a computing environment perfectly similar to Matlab. This is a feature of Python that brought the success of this programming language in the scientific community.

Python is a programming language that can be **interfaced**. In fact, this programming language has among its features the characteristic to be interfaced with code written in other programming languages such as C / C ++ and Fortran. Even this was a winning choice. In fact, thanks to this aspect Python could compensate for what is perhaps the only weak point, the speed of execution. The nature of Python, as a highly dynamic programming language, can lead sometimes to execution of programs up to 100 times slower than the corresponding static programs compiled with other languages. Thus the solution to this kind of performance problems is to interface Python to compiled code of other languages by using it as if it were its own.

Python is an **open-source** programming language. CPython, which is the reference implementation of the Python language is completely free and open-source. Additionally every module or library in the network is open-source and their code is available online. Every month, an extensive developer community brings some improvements to make this language and all its libraries even richer and more efficient. CPython is managed by the nonprofit **Python Software Foundation**, which was created in 2001 and has set itself the task of promoting, protecting, and advancing the Python programming language.

Finally, Python is a **simple** language to use and learn. This aspect is perhaps the most important of all because it is the most direct aspect which a developer, even a novice, is facing. The high intuitiveness and ease of reading of the Python code often leads to a "sympathy" for this programming language, and consequently it is the choice of most newcomers in programming. However, its simplicity does not mean narrowness, since Python is a language that is spreading in every field of computing. Furthermore, Python is doing all of this so simply, in comparison to existing programming languages such as C ++, Java, and Fortran, which by their nature are very complex.

Python—The Interpreter

As described in the previous sections, each time you run the *python* command the Python interpreter starts, characterized by a >>> prompt.

The Python interpreter is simply a program that reads and interprets the commands passed to the prompt. You have seen that the interpreter can accept either a single command at a time or entire files of Python code. However the approach by which it performs this is always the same.

Each time you press the Enter key, the interpreter begins to scan the code written (either a row or a full file of code) token by token (**tokenization**). These tokens are fragments of text which the interpreter will arrange in a tree structure. The tree obtained is the logical structure of the program which is then converted to **bytecode** (.pyc or .pyo). The process chain ends with the bytecode which will be executed by a **Python virtual machine (PVM)**. See Figure 2-1.
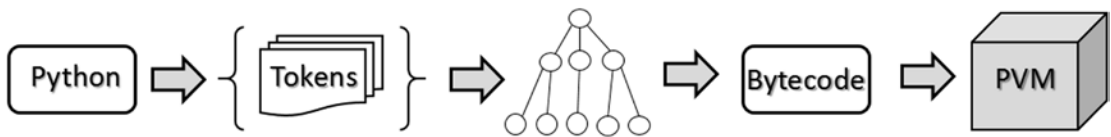
Figure 2-1. *The steps performed by the Python interpreter*

You can find a very good documentation on this topic at the link https://www.ics.uci.edu/~pattis/ICS-31/lectures/tokens.pdf.

The standard interpreter of Python is reported as Cython, since it was totally written in C. There are other areas that have been developed using other programming languages such as **Jython**, developed in Java; **IronPython,** developed in C # (and then only for Windows); and **PyPy,** developed entirely in Python.

Cython

The project Cython is based on creating a compiler that translates Python code into C code equivalent. This code is then executed within a Cython environment at runtime. This type of compilation system has made possible the introduction of C semantics within the Python code to make it even more efficient. This system has led to the merging of two worlds of programming language with the birth of **Cython** that can be considered a new programming language. You can find a lot of documentation about it online; I advise you to visit this link (http://docs.cython.org).

Jython

In parallel to Cython, there is a version totally built and compiled in Java, named **Jython**. It was created by Jim Hugunin in 1997 (http://www.jython.org). Jython is a version of implementation of the Python programming language in Java; it is further characterized by using Java classes instead of Python modules to implement extensions and packages of Python.

PyPy

The PyPy interpreter is a JIT (just-in-time) compiler, which converts the Python code directly in machine code at runtime. This choice was made to speed up the execution of Python. However, this choice has led to the use of a small subset of Python commands, defined as **RPython**. For more information on this please consult the official website: http://pypy.org.

Python 2 and Python 3

The Python community is still in transition from interpreters of the Series 2 to Series 3. In fact, currently you will find two releases of Python that are used in parallel (version 2.7 and version 3.4). This kind of ambiguity can create much confusion, especially in terms of choosing which version to use and the differences between these two versions. One question that you surely must be asking is why version 2.x is still being released if it is distributed around a much more enhanced version such as 3.x.

When Guido Van Rossum (the creator of the Python language) decided to bring significant changes to the Python language, he soon found that these changes would make the new Python incompatible with a lot of existing code. Thus he decided to start with a new version of Python called Python 3.0. To overcome the problem of compatibility and create huge amounts of code unusable spread to the network, it was decided to maintain a compatible version, 2.7 to be precise.

15

Python 3.0 made its first appearance in 2008, while version 2.7 was released in 2010 with a promise that it would not be followed by big releases, and at the moment the current version is 3.4 (2014).

In the book we will refer to the Python 2.x version; however, with some few exceptions, there should be no problems with the Python 3.x version.

Installing Python

In order to develop programs in Python you have to install it on your operating system. Differently from Windows, Linux distributions and Mac OS X should already have within them a preinstalled version of Python. If not, or if you would like to replace it with another version, you can easily install it. The installation of Python differs from the operating system; however, it is a rather simple operation.

On Debian-Ubuntu Linux systems

```
apt-get install python
```

On Red Hat, Fedora Linux systems working with rpm packages

```
yum install python
```

If your operating system is Windows or Mac OS X you can go on the official Python site (http://www.python.org) and download the version you prefer. The packages in this case are installed automatically.

However, today there are distributions that provide along with the Python interpreter a number of tools that make the management and installation of Python, all libraries, and associated applications easier. I strongly recommend you choose one of the distributions available online.

Python Distributions

Due to the success of the Python programming language, over the years the tools in the package, which have been developed to meet the most various functionalities, have become such a large number so as to make it virtually impossible to manage all of them manually.

In this regard, many Python distributions that allow efficient management of hundreds of Python packages are now available. In fact, instead of individually downloading the interpreter, which has within it only the standard libraries, and then needing to install later individually all the additional libraries, it is much easier to install a Python distribution.

The heart of these distributions are the **package managers**, which are nothing more than applications which automatically manage, install, upgrade, configure, and remove Python packages that are part of the distribution.

Their functionality is very useful, since the user simply makes a request on a particular package (which could be an installation for example), and the package manager, usually via the Internet, performs the operation by analyzing the necessary version, alongside all dependencies with any other packages, and downloading them if not present.

Anaconda

Anaconda is a free distribution of Python packages distributed by Continuum Analytics (https://store.continuum.io/cshop/anaconda/). This distribution supports Linux, Windows, and Mac OSX operating systems. Anaconda, in addition to providing the latest packages released in the Python world,

comes bundled with most of the tools you need to set up a development environment for Python programming language.

Indeed, when you install the Anaconda distribution on your system, you have the opportunity to use many tools and applications described in this chapter, without worrying about having to install and manage each of them separately. The basic distribution includes Spyder as IDE, IPython QtConsole, and Notebook.

The management of the entire Anaconda distribution is performed by an application called **conda**. This is the package manager and the environment manager of the Anaconda distribution that handles all of the packages and their versions.

```
conda install <package name>
```

One of the most interesting aspects of this distribution is the ability to manage multiple development environments, each with its own version of Python. Indeed, when you install Anaconda, the Python version 2.7 is installed by default. All installed packages then will refer to that version. This is not a problem, because Anaconda offers the possibility to work simultaneously and independently with other Python versions by creating a new environment. You can create, for instance, an environment based on Python 3.4.

```
conda create -n py34 python=3.4 anaconda
```

This will generate a new Anaconda environment with all the packages related to the Python 3.4 version. This installation will not affect in any way the environment built with Python 2.7. Once installed, you can activate the new environment entering the following command.

```
source activate py34
```

on Windows instead:

```
activate py34
C:\Users\Fabio>activate py34
Activating environment "py34"...
[py34] C:\Users\Fabio>
```

You can create as many versions of Python as you want; you need only to change the parameter passed with the *python* option in the command **conda create**. When you want to return to work with the original Python version you have to use the following command:

```
source deactivate
```

on Windows

```
[py34] C:\Users\Fabio>deactivate
Deactivating environment "py34"
C:\Users\Fabio>
```

Enthought Canopy

There is another distribution very similar to Anaconda and it is the Canopy distribution provided by Enthought, a company founded in 2001 and very famous especially for the SciPy project (https://www.enthought.com/products/canopy/). This distribution supports Linux, Windows, and Mac OX systems and it consists of a large amount of packages, tools, and applications managed by a package manager. The package manager of Canopy, as opposed to conda, is totally graphic.

Unfortunately, only the basic version of this distribution, defined **Canopy Express**, is free; in addition to the package normally distributed, it also includes IPython and an IDE of Canopy that has a special feature that is not present in other IDEs. It has embedded the IPython in order to use this environment as a window for testing and debugging code.

Python(x,y)

Python(x,y) is a free distribution that only works on Windows and is downloadable from http://code.google.com/p/pythonxy/. This distribution has Spyder as IDE.

Using Python

Python is a language rich but simple at the same time, very flexible; it allows expansion of your development activities in many areas of work (data analysis, scientifics, graphic interfaces, etc.). Precisely for this reason, the possibility of using Python can take very many different contexts, often according to the taste and ability of the developer. This section presents the various approaches to using Python in the course of the book. According to the various topics discussed in different chapters, these different approaches will be used specifically as they will be more suited to the task in charge.

Python Shell

The easiest way to approach the Python world is to open a session on the Python shell, a terminal running command lines. In fact, you can enter a command line at a time and test its operation immediately. This mode makes clear the nature of the interpreter that underlies the operation of Python. In fact the interpreter is able to read a command at a time, keeping the status of the variables specified in the previous lines, a behavior similar to that of Matlab and other calculation software.

This approach is very suitable for those approaches for the first time with the Python language. You have the ability to test command to command every time without having to write, edit, and run an entire program, sometimes composed of many lines of code.

This mode is also indicated to do to test and debug Python code one line at a time, or simply to make calculations. To start a session on the terminal, simply write in the command line

```
>>> python
Python 2.7.8 (default, Jul  2 2014, 15:12:11) [MSC v.1500 64 bit (AMD64)]
on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Now the Python shell is active and the interpreter is ready to receive commands in Python. Start by entering the simplest of commands but a classic for getting started with programming.

```
>>> print "Hello World!"
Hello World!
```

Run an Entire Program Code

The most familiar way for each programmer is to write an entire program code and then run it from the terminal. First write a program using a simple text editor; you can use as example the code shown in Listing 2-1 and save it as *MyFirstProgram.py*.

Listing 2-1. MyFirstProgram.py

```
myname = raw_input("What is your name? ")
print "Hi " + myname + ", I'm glad to say: Hello world!"
```

Now you've written your first program in Python, and you can run it directly from the command line by calling the python command and then the name of the file containing the program code.

```
python myFirstProgram.py
What is your name? Fabio Nelli
Hi Fabio Nelli, I'm glad to say: Hello world!
```

Implement the Code Using an IDE

A more comprehensive approach than the previous ones is the use of an IDE (or better, an **Integrated Development Environment**). These editors are real complex software that provide a work environment on which to develop your Python code. They are rich in tools that make life easier for developers, especially when debugging. In the following sections you will see in detail what IDEs are currently available.

Interact with Python

The last approach, and in my opinion, perhaps the most innovative, is the interactive one. In fact, in addition to the three previous approaches, which are those that for better or worse are used by all developers of other programming languages, this approach provides the opportunity to interact directly with the Python code.

In this regard, the Python's world has been greatly enriched with the introduction of **IPython**. IPython is a very powerful tool, designed specifically to meet the needs of interaction between the Python interpreter and the developer, which under this approach takes the role of analyst, engineer, or researcher. In a later section IPython and its features will be explained in more detail.

Writing Python Code

In the previous section you saw how to write a simple program in which the string "Hello World" was printed. Now in this section you will get a brief overview of the basics of Python language just to get familiar with the most important basic aspects.

This section is not intended to teach you to program in Python, or to illustrate syntax rules of the programming language, but just to give you a quick overview of some basic principles of Python necessary to continue with the topics covered in this book.

If you already know the Python language you can safely skip this introductory section. Instead if you are not familiar with the programming and you find it difficult to understand the topics, I highly recommend you to see the online documentation, tutorials, and courses of various kinds.

Make Calculations

You have already seen that the **print()** function is useful for printing almost anything. Python, in addition to being a printing tool, is also a great calculator. Start a session on the Python shell and begin to perform these mathematical operations:

```
>>> 1 + 2
3
>>> (1.045 * 3)/4
0.78375
>>> 4 ** 2
16
>>> ((4 + 5j) * (2 + 3j))
(-7+22j)
>>> 4 < (2*3)
True
```

Python is able to calculate many types of data including complex numbers and conditions with Boolean values. As you can see from the above calculations, the Python interpreter returns directly the result of the calculations without the need to use the **print()** function. The same thing applies to values contained within variables. It's enough to call the variable to see its contents.

```
>>> a = 12 * 3.4
>>> a
40.8
```

Import New Libraries and Functions

You saw that Python is characterized by the ability to extend its functionality by importing numerous packages and modules available. To import a module in its entirety you have to use the **import** command.

```
>>> import math
```

In this way all the functions contained within the *math* package are available in your Python session so you can call them directly. Thus you have extended the standard set of functions available when you start a Python session. These functions are called with the following expression.

```
library_name.function_name()
```

For example, you are now able to calculate the sine of the value contained within the variable *a*.

```
>>> math.sin(a)
```

As you can see the function is called along with the name of the library. Sometimes you might find the following expression for declaring an import.

```
>>> from math import *
```

Even if this works properly, it is to be avoided for a good practice. In fact writing an import in this way involves the importation of all functions without necessarily defining the library to which they belong.

```
>>> sin(a)
0.040693257349864856
```

This form of import can actually lead to very large errors, especially if the imported libraries are beginning to be numerous. In fact, it is not unlikely that different libraries have functions with the same name, and importing all of these would result in an override of all functions with the same name previously imported. Therefore the behavior of the program could generate numerous errors or worse, abnormal behavior.

Actually, this way to import is generally used for only a limited number of functions, that is, functions that are strictly necessary for the functioning of the program, thus avoiding the importation of an entire library when it is completely unnecessary.

```
>>> from math import sin
```

Data Structure

You saw in the previous examples how to use simple variables containing a single value. Actually Python provides a number of extremely useful data structures. These data structures are able to contain several data simultaneously, and sometimes even of different types. The various data structures provided are defined differently depending on how their data are structured internally.

- list
- set
- strings
- tuples
- dictionary
- deque
- heap

This is only a small part of all the data structures that can be made with Python. Among all these data structures, the most commonly used are **dictionaries** and **lists**.

The type **dictionary**, defined also as **dicts**, is a data structure in which each particular value is associated with a particular label called **key**. The data collected in a dictionary have no internal order but only definitions of key/value pairs.

```
>>> dict = {'name':'William', 'age':25, 'city':'London'}
```

If you want to access a specific value within the dictionary you have to indicate the name of the associated key.

```
>>> dict["name"]
'William'
```

If you want to iterate the pairs of values in a dictionary you have to use the **for-in** construct. This is possible through the use of the **items()** function.

```
>>> for key, value in dict.items():
...     print(key,value)
...
name William
city London
age 25
```

The type **list** is a data structure that contains a number of objects in a precise order to form a sequence to which elements can be added and removed. Each item is marked with a number corresponding to the order of the sequence, called **index**.

```
>>> list = [1,2,3,4]
>>> list
[1, 2, 3, 4]
```

If you want to access the individual elements it is sufficient to specify the index in square brackets (the first item in the list has 0 as index), while if you take out a portion of the list (or a sequence), it is sufficient to specify the range with the indices i and j corresponding to the extremes of the portion.

```
>>> list[2]
3
>>> list[1:3]
[2, 3]
```

Instead if you are using negative indices, this means you are considering the last item in the list and gradually moving to the first.

```
>>> list[-1]
4
```

In order to do a scan of the elements of a list you can use the **for-in** construct.

```
>>> items = [1,2,3,4,5]
>>> for item in items:
...         item + 1
...
2
3
4
5
6
```

Functional Programming (Only for Python 3.4)

The **for-in** loop shown in the previous example is very similar to those found in other programming languages. But actually, if you want to be a "Python" developer you have to avoid using explicit loops. Python offers other alternative approaches, specifying these programming techniques such as **functional programming** (expression-oriented programming).

The tools that Python provides to develop functional programming comprise a series of functions:

- map(function, list)
- filter(function, list)
- reduce(function, list)
- lambda
- list comprehension

The *for* loop that you have just seen has a specific purpose, which is to apply an operation on each item and then somehow gather the result. This can be done by the **map()** function.

```
>>> items = [1,2,3,4,5]
>>> def inc(x): return x+1
...
>>> list(map(inc,items))
[2, 3, 4, 5, 6]
```

In the previous example, first you have defined the function that performs the operation on every single element, and then you have passed it as the first argument to the *map()*. Python allows you to define the function directly within the first argument using **lambda** as a function. This greatly reduces the code, and compacts the previous construct, in a single line of code.

```
>>> list(map((lambda x: x+1),items))
[2, 3, 4, 5, 6]
```

Two other functions working in a similar way are **filter()** and **reduce()**. The **filter()** function extracts the elements of the list for which the function returns True. The **reduce()** function instead considers all the elements of the list to produce a single result. To use **reduce()**, you must import the module **functools**.

```
>>> list(filter((lambda x: x < 4), items))
[1, 2, 3]
>>> from functools import reduce
>>> reduce((lambda x,y: x/y), items)
0.008333333333333333
```

Both of these functions implement other types of using the *for* loop. They are going to replace these cycles and their functionality, which can be alternatively expressed with simple functions calling. That is what constitutes the **functional programming**.

The final concept of functional programming is the **list comprehension**. This concept is used to build lists in a very natural and simple way, referring to them in a manner similar to how the mathematicians describe data sets. The values of the sequence are defined through a particular function or operation.

```
>>> S = [x**2 for x in range(5)]
>>> S
[0, 1, 4, 9, 16]
```

Indentation

A peculiarity for those coming from other programming languages is the role that **indentation** plays. Whereas you used to manage the indentation for purely aesthetic reasons, making the code somewhat more readable, in Python it assumes an integral role in the implementation of the code, dividing it into logical blocks. In fact, while in Java, C, and C ++ each command line of code is separated from the next by a ';', in Python you should not specify any symbol that separates them, included the braces to indicate a logical block. These roles in Python are handled through indentation; that is, depending on the starting point of the line of code, the interpreter considers that line whether it belongs to a logical block or not.

```
>>> a = 4
>>> if a > 3:
...    if a < 5:
...        print("I'm four")
... else:
...    print("I'm a little number")
...
I'm four

>>> if a > 3:
...    if a < 5:
...        print("I'm four")
...    else:
...        print("I'm a big number")
...
I'm four
```

In this example you can see that depending on how the else command is indented, the conditions assume two different meanings (specified by me in the strings themselves).

IPython

IPython is a further development of Python that includes a number of tools: **IPython shell**, a powerful interactive shell resulting in a greatly enhanced Python terminal; a **QtConsole,** which is a hybrid between a shell and a GUI, allowing in this way to display graphics inside the console instead of in separate windows; and finally the **IPython Notebook,** which is a web interface that allows you to mix text, executable code, graphics, and formulas in a single representation.

IPython Shell

This shell apparently resembles a Python session run from a command line, but actually, it provides many other features that make this shell much more powerful and versatile than the classic one. To launch this shell just type *ipython* in the command line.

```
> ipython
Python 2.7.8 (default, Jul  2 2014, 15:12:11) [M
Type "copyright", "credits", or "license" for more information.
```

```
IPython 2.4.1 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]:
```

As you can see, a particular prompt appears with the value *In [1]*. This means that it is the first line of input. Indeed, IPython offers a system of numbered prompts (indexed) with input and output caching.

```
In [1]: print "Hello World!"
Hello World!

In [2]: 3/2
Out[2]: 1

In [3]: 5.0/2
Out[3]: 2.5

In [4]:
```

The same thing applies to values in output that are indicated with the value *Out[1]*, *Out [2]*, and so on. IPython saves all inputs that you enter storing them as variables. In fact, all the inputs entered were included as fields within a list called **In**.

```
In [4]: In
Out[4]: ['', u'print "Hello World!"', u'3/2', u'5.0/2', u'_i2', u'In']
```

The indices of the list elements are precisely the values that appear in each prompt. Thus, to access a single line of input you can simply specify precisely that value.

```
In [5]: In[3]
Out[5]: u'5.0/2'
```

Even for output you can apply the same.

```
{2: 1,
 3: 2.5,
 4: ['',
  u'print "Hello World!"',
  u'3/2',
  u'5.0/2',
  u'_i2',
  u'In',
  u'In[3]',
  u'Out'],
 5: u'5.0/2'}
```

IPython Qt-Console

In order to launch this application from the command line you must enter the following command:

```
ipython qtconsole
```

The application consists of a GUI in which you have all the functionality present in the IPython shell. See Figure 2-2.
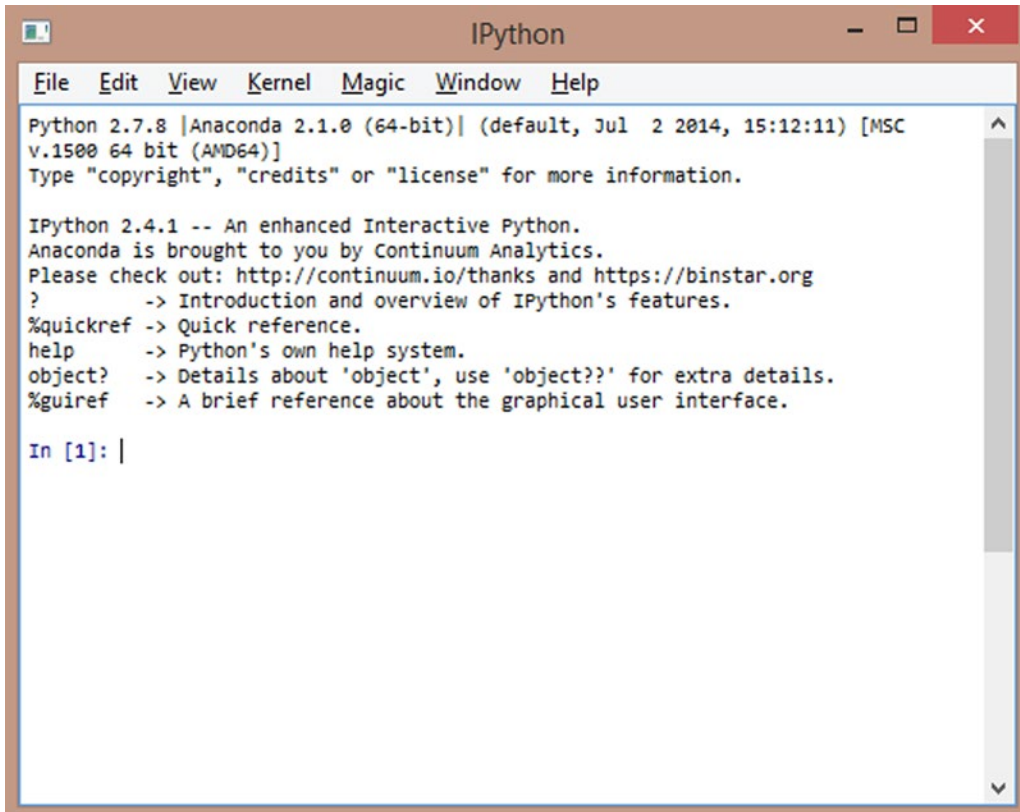
Figure 2-2. *The IPython QtConsole*

IPython Notebook

IPython Notebook is the latest evolution of this interactive environment (see Figure 2-3). In fact, with IPython Notebook, you can merge executable code, text, formulas, images, and animations into a single Web document, useful for many purposes such as presentations, tutorials, debug, and so forth.
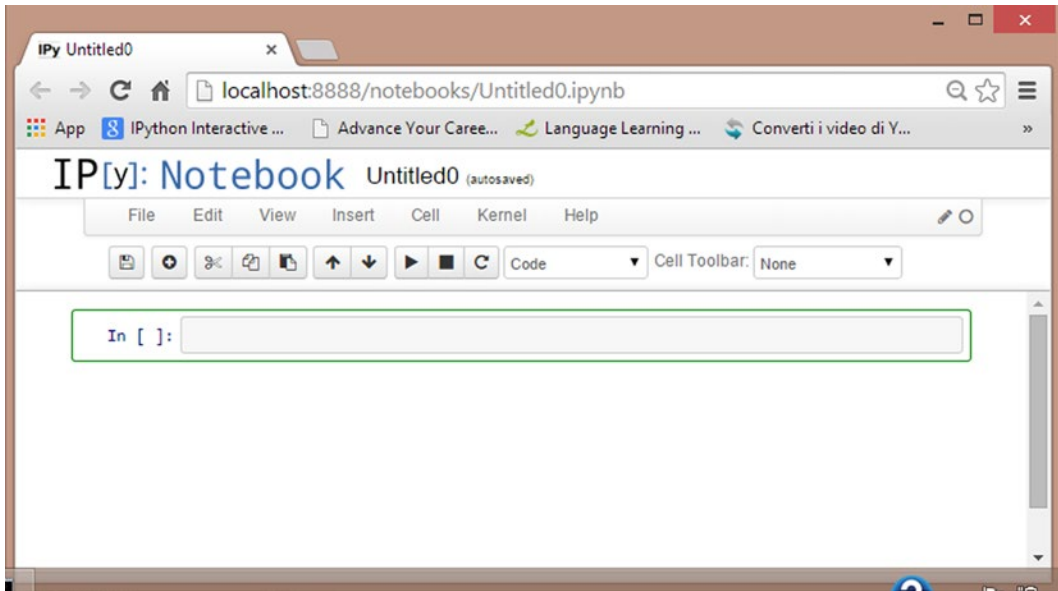
Figure 2-3. *The web page showing the IPython Notebook*

The Jupyter Project

IPython is a project that has grown enormously in recent times, and with the release of IPython 3.0, everything is moving toward a new project called **Jupyter** (https://jupyter.org).

IPython will continue to exist as a Python shell, and as a kernel of Jupyter, but the Notebook and the other language-agnostic components belonging to the IPython project will all move to form the new Jupyter project.

Figure 2-4. *The Jupyter project's logo*

PyPI—The Python Package Index

The **Python Package Index (PyPI)** is a software repository that contains all the software needed for programming in Python, for example, all Python packages belonging to other Python libraries. The content repository is managed directly by the developers of individual packages that deal with updating the repository with the latest versions of their released libraries. For a list of the packages contained within the repository you should go to see the official page of PyPI with this link: https://pypi.python.org/pypi.

As far as the administration of these packages, you can use the **pip** application which is the package manager of PyPI.

Launching it from the command line, you can manage all the packages individually deciding if the package is to be installed, upgraded, or removed. Pip will check if the package is already installed, of if it needs to be updated, to control dependencies, that is, to assess whether other packages are necessary. Furthermore, it manages their downloading and installation.

```
$ pip install <<package_name>>
$ pip search <<package_name>>
$ pip show <<package_name>>
$ pip unistall <<package_name>>
```

Regarding the installation, if you have Python 3.4+ (released March 2014) and Python 2.7.9+ (released December 2014) already installed on your system, the pip software is already included in these releases of Python. However, if you are still using an older version of Python you need to install pip on your system. The installation of pip on your system depends on the operating system on which you are working.

On Linux Debian-Ubuntu:

```
$ sudo apt-get install python-pip
```

On Linux Fedora

```
$ sudo yum install python-pip
```

On Windows:

Visit the site *www.pip-installer.org/en/latest/installing.html* and download **get-pip.py** on your PC. Once the file is downloaded, run the command

```
python get-pip.py
```

In this way, you will install the package manager. Remember to add *C:\Python2.X\Scripts* in the PATH environment variable.

The IDEs for Python

Although most of the Python developers are used to implement their code directly from the shell (Python or IPython), some **IDEs (Interactive Development Environments)** are also available. In fact, in addition to a text editor, these graphics editors also provide a series of tools very useful during the drafting of the code. For example, the auto-completion of code, viewing the documentation associated with the commands, debugging, and breakpoints are only some of the tools that this kind of application can provide.

IDLE (Integrated DeveLopment Environment)

IDLE is a very simple IDE created specifically for development in Python. It is the official IDE included in the standard Python release, so it is embedded within the standard distribution of Python (see Figure 2-5). IDLE is a piece of software that is fully implemented in Python.
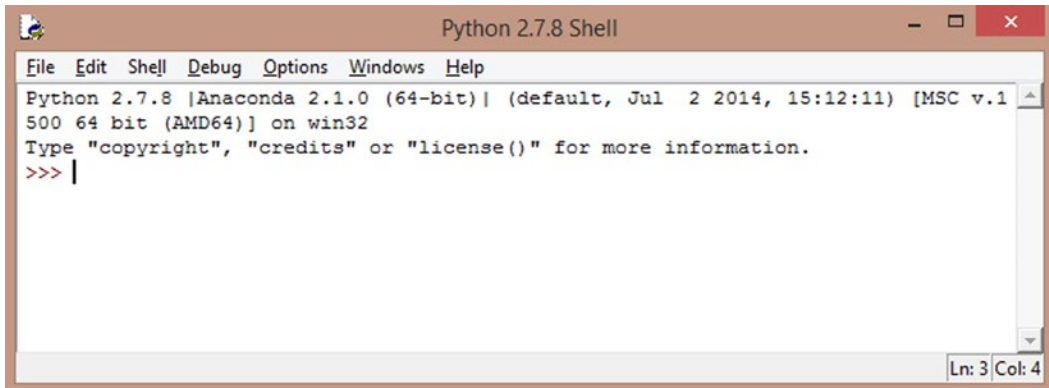
Figure 2-5. *The IDLE Python shell*

Spyder

Spyder (Scientific Python Development Environment) is an IDE that has similar features to the IDE of Matlab (see Figure 2-6). The text editor is enriched with syntax highlighting and code analysis tools. Also, using this IDE you have the option to integrate ready-to-use widgets in your graphic applications.
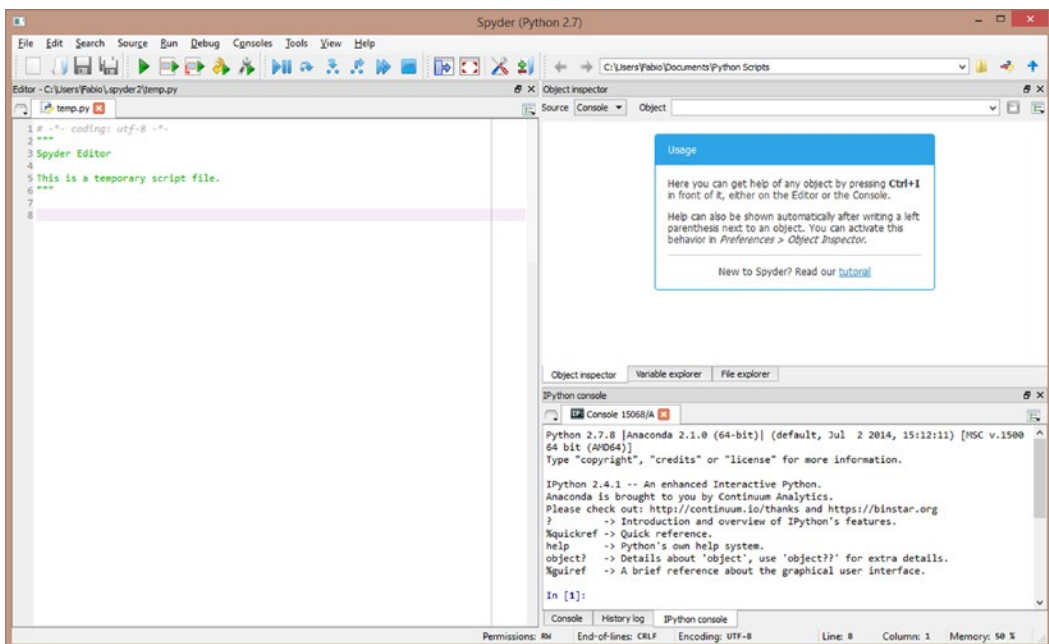
Figure 2-6. *The Spyder IDE*

Eclipse (pyDev)

Those who developed in other programming languages certainly know Eclipse, a universal IDE developed entirely in Java (therefore requiring Java installation on your PC) that provides a development environment for many programming languages (see Figure 2-7). So there is also an Eclipse version for developing in Python thanks to the installation of an additional plug-in called **pyDev**.
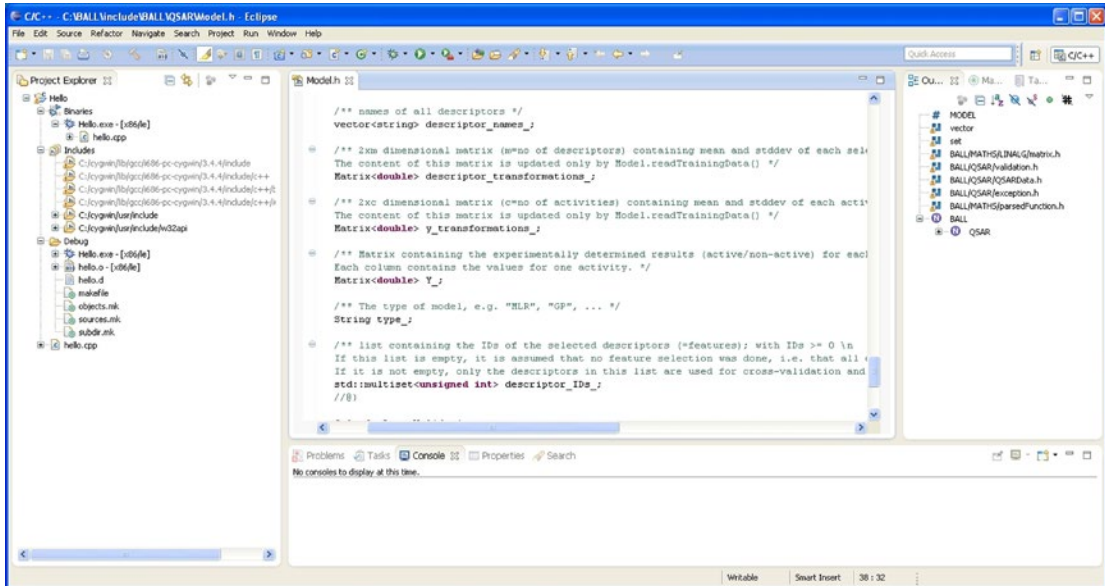
Figure 2-7. *The Eclipse IDE*

Sublime

This text editor is one of the preferred environment for Python programmers (see Figure 2-8). In fact, there are several plug-ins available for this application that make Python implementation easy and enjoyable.
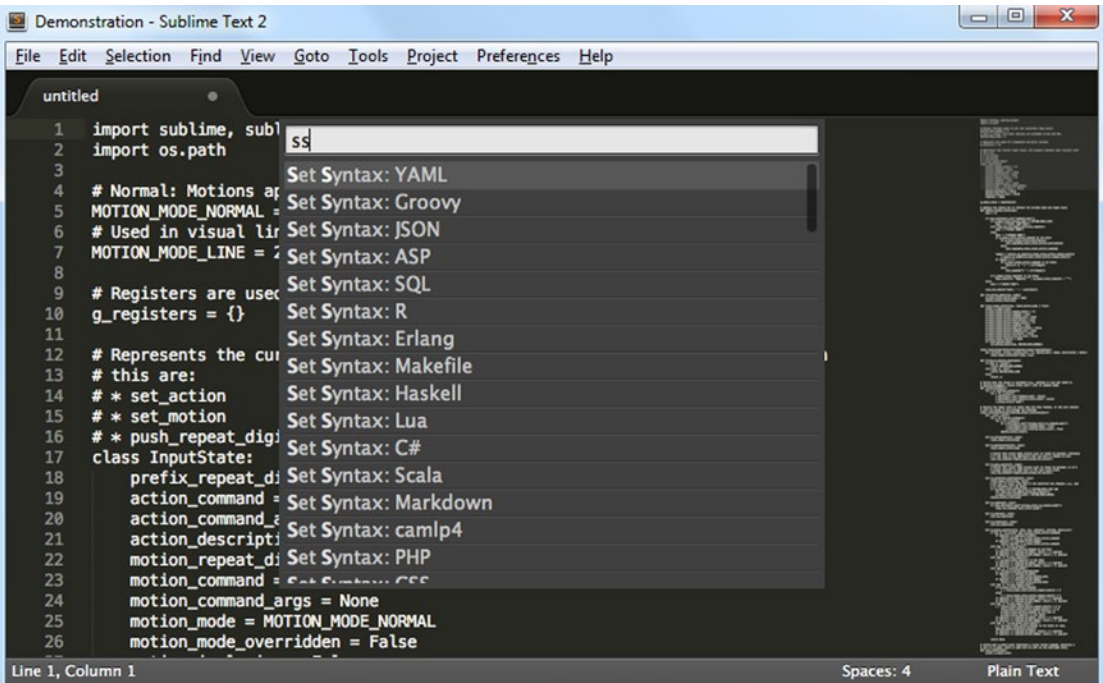
Figure 2-8. *The Sublime IDE*

Liclipse

Liclipse, similarly to Spyder, is a development environment specifically designed for the Python language (see Figure 2-9). Basically it is totally similar to the Eclipse IDE but it is fully adapted for a specific use of Python, without installing plug-ins like PyDev. So its installation and its setting are much simpler than Eclipse.
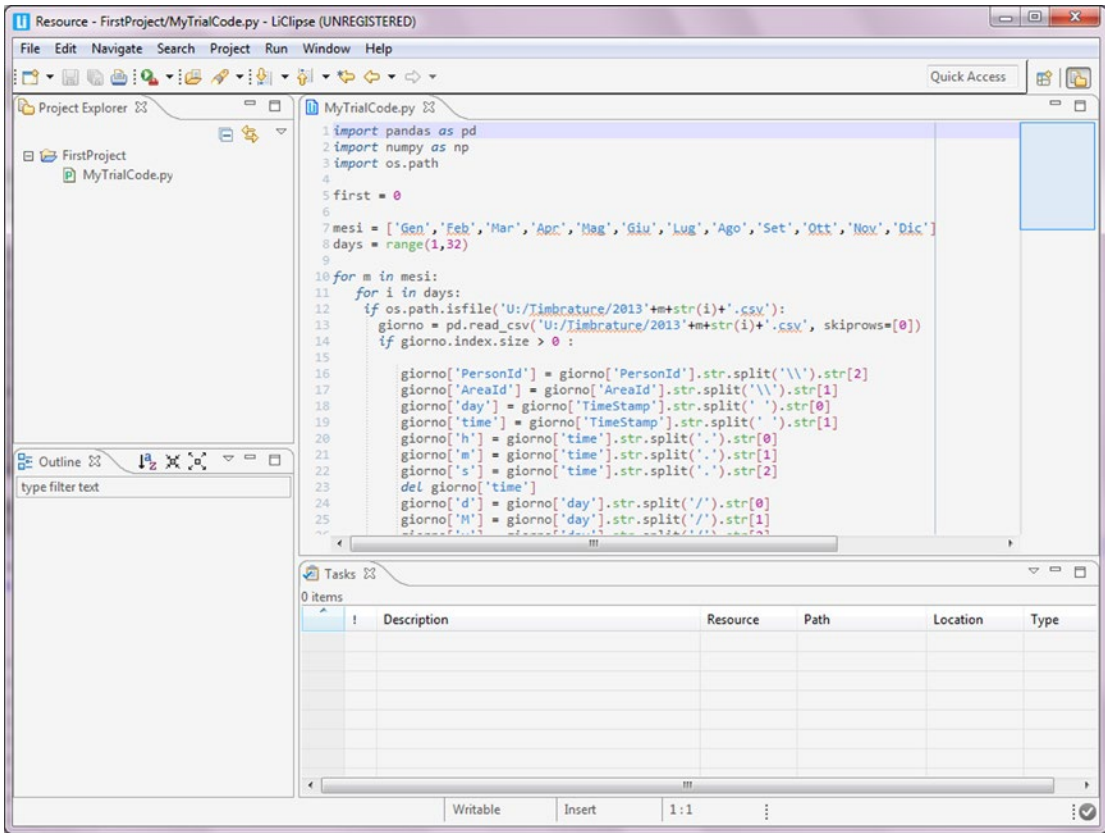
Figure 2-9. The Liclipse IDE

NinjaIDE

NinjaIDE (NinjaIDE is "Not Just Another **IDE"**) characterized by a name that is a recursive acronym, is a specialized IDE for the Python language. It's a very recent application on which the efforts of many developers are focused. Being already very promising, it is likely that in the coming years, this IDE will be a source of many surprises.

Komodo IDE

Komodo is a very powerful IDE full of tools that make it a complete and professional development environment. Paid software, written in C ++, Komodo is an IDE that provides a development environment adaptable to many programming languages, including Python.

SciPy

SciPy (pronounced "Sigh Pie") is a set of open-source Python libraries specialized for scientific computing. Many of these libraries will be the protagonists of many chapters of the book, given that their knowledge is critical to the data analysis. Together they constitute a set of tools for calculating and displaying data that has

little to envy from other specialized environments for calculation and data analysis (such as R or Matlab). Among the libraries that are part of the SciPy group, there are some in particular that will be discussed in the following chapters:

- NumPy

- Matplotlib

- Pandas

NumPy

This library, whose name means Numerical Python, actually constitutes the core of many other Python libraries that have originated from it. Indeed NumPy is the foundation library for scientific computing in Python since it provides data structures and high-performing functions that the basic package of the Python cannot provide. In fact, as you will see later in the book, NumPy defines a specific data structure that is an *N*-dimensional array defined as **ndarray**.

The knowledge of this library is revealed in fact essential in terms of numerical calculations since its correct use can greatly influence the performance of a computation. Throughout the book, this library will be almost omnipresent because of its unique characteristics, so its discussion in a chapter devoted to it (Chapter 3) proves to be necessary.

This package provides some features that will be added to the standard Python:

- ndarray: a multidimensional array much faster and more efficient than those provided by the basic package of Python.

- element-wise computation: a set of functions for performing this type of calculation with arrays and mathematical operations between arrays.

- reading-writing data sets: a set of tools for reading and writing data stored in the hard disk.

- Integration with other languages such as C, C ++, and FORTRAN: a set of tools to integrate code developed with these programming languages

Pandas

This package provides complex data structures and functions specifically designed to make the work on them easy, fast, and effective. This package is the core for the data analysis with Python. Therefore, the study and application of this package will be the main argument on which you will work throughout the book (especially Chapters 4, 5, and 6). So its knowledge in every detail, especially when it is applied to the data analysis, is a fundamental objective of this book.

The fundamental concept of this package is the **DataFrame**, a two-dimensional tabular data structure with row and column labels.

Pandas combines the high performance properties of the NumPy library to apply them to the manipulation of data in spreadsheets or in relational databases (SQL database). In fact, using sophisticated indexing it will be easy to carry out many operations on this kind of data structures, such as reshaping, slicing, aggregations, and the selection of subsets.

matplotlib

This package is the Python library that is currently most popular for producing plots and other data visualizations in 2D. Since the data analysis requires visualization tools, this is the library that best suits the purpose. In Chapter 7 you will see in detail this rich library so you will know how to represent the results of your analysis in the best way.

Conclusions

In the course of this chapter all the fundamental aspects characterizing the Python's world have been illustrated. The Python programming language is introduced in its basic concepts with brief examples, explaining the innovative aspects that it introduces and especially how it stands out compared to other programming languages. In addition, different ways of using Python at various levels have been presented. First you have seen how to use a simple command-line interpreter, then a set of simple graphical user interfaces are shown until you get to such complex development environments, known as IDE, such as Spyder and NinjaIDE.

Even the highly innovative project IPython was presented, showing the possibility to develop the Python code interactively, in particular with the IPython Notebook.

Moreover, the modular nature of Python has been highlighted with the ability to expand the basic set of standard functions provided by Python with external libraries. In this regard, the PyPI online repository has been shown along with other Python distributions such as Anaconda and Enthought Canopy.

In the next chapter you will deal with the first library that is the basis of numerical calculation in Python: **NumPy**. You will learn about the **ndarray**, a data structure which will be the basis of all the more complex data structures used in the data analysis and shown in the following chapters.

CHAPTER 3

■ ■ ■

The NumPy Library

NumPy is a basic package for scientific computing with Python and especially for data analysis. In fact, this library is the basis of a large amount of mathematical and scientific Python packages, and among them, as you will see later in the book, the pandas library. This library, totally specialized for data analysis, is fully developed using the concepts introduced by NumPy. In fact, the built-in tools provided by the standard Python library could be too simple or inadequate for most of the calculations in the data analysis.

So the knowledge of the NumPy library is a prerequisite in order to face, in the best way, all scientific Python packages, and particularly, to use and understand more about the pandas library and how to get the most out of it. The pandas library will be the main subject in the following chapters.

If you are already familiar with this library, you can proceed directly to the next chapter; otherwise you may see this chapter as a way to revise the basic concepts or to regain familiarity with it by running the examples described in this chapter.

NumPy: A Little History

At the dawn of the Python language, the developers began to need to perform numerical calculations, especially when this language began to be considered by the scientific community.

The first attempt was Numeric, developed by Jim Hugunin in 1995, which was successively followed by an alternative package called Numarray. Both packages were specialized for the calculation of arrays, and each of them had strengths depending on which case they were used. Thus, they were used differently depending on where they showed to be more efficient. This ambiguity led then to the idea of unifying the two packages and therefore Travis Oliphant started to develop the NumPy library. Its first release (v 1.0) occurred in 2006.

From that moment on, NumPy has proved to be the extension library of Python for scientific computing, and it is currently the most widely used package for the calculation of multidimensional arrays and large arrays. In addition, the package also comes with a range of functions that allow you to perform operations on arrays in a highly efficient way and to perform high-level mathematical calculations.

Currently, NumPy is open source and licensed under BSD. There are many contributors that with their support have expanded the potential of this library.

The NumPy Installation

Generally, this module is present as a basic package in most Python distributions; however, if not, you can install it later.

On Linux (Ubuntu and Debian)

```
sudo apt-get install python-numpy
```

On Linux (Fedora)

```
sudo yum install numpy scipy
```

On Windows with Anaconda

```
conda install numpy
```

Once NumPy is installed in your distribution, to import the NumPy module within your Python session, write:

```
>>> import numpy as np
```

Ndarray: The Heart of the Library

The whole NumPy library is based on one main object: **ndarray** (which stands for *N*-dimensional array). This object is a multidimensional homogeneous array with a predetermined number of items: homogeneous because virtually all the items within it are of the same type and the same size. In fact, the data type is specified by another NumPy object called **dtype** (data-type); each ndarray is associated with only one type of dtype.

The number of the dimensions and items in an array is defined by its **shape**, a tuple of *N*-positive integers that specifies the size for each dimension. The dimensions are defined as **axes** and the number of axes as **rank**.

Moreover, another peculiarity of NumPy arrays is that their size is fixed, that is, once you defined their size at the time of creation, it remains unchanged. This behavior is different from Python lists, which can grow or shrink in size.

To define a new ndarray, the easiest way is to use the **array()** function, passing a Python list containing the elements to be included in it as an argument.

```
>>> a = np.array([1, 2, 3])
>>> a
array([1, 2, 3])
```

You can easily check that a newly created object is an ndarray, passing the new variable to the **type()** function.

```
>>> type(a)
<type 'numpy.ndarray'>
```

In order to know the associated dtype to the just created ndarray, you have to use the **dtype** attribute.

```
>>> a.dtype
dtype('int32')
```

The just-created array has one axis, and then its rank is 1, while its shape should be (3,1). To obtain these values from the corresponding array it is sufficient to use the **ndim** attribute for getting the axes, the **size** attribute to know the array length, and the **shape** attribute to get its shape.

```
>>> a.ndim
1
>>> a.size
3
>>> a.shape
(3L,)
```

What you have just seen is the simplest case that is a one-dimensional array. But the use of arrays can be easily extended to the case with several dimensions. For example, if you define a two-dimensional array 2x2:

```
>>> b = np.array([[1.3, 2.4],[0.3, 4.1]])
>>> b.dtype
dtype('float64')
>>> b.ndim
2
>>> b.size
4
>>> b.shape
(2L, 2L)
```

This array has rank 2, since it has two axis, each of length 2.

Another important attribute is **itemsize**, which can be used with ndarray objects. It defines the size in bytes of each item in the array, and **data** is the buffer containing the actual elements of the array. This second attribute is still not generally used, since to access the data within the array you will use the indexing mechanism that you will see in the next sections.

```
>>> b.itemsize
8
>>> b.data
<read-write buffer for 0x0000000002D34DF0, size 32, offset 0 at 0x0000000002D5FEA0>
```

Create an Array

To create a new array you can follow different paths. The most common is the one you saw in the previous section through a list or sequence of lists as arguments to the **array()** function.

```
>>> c = np.array([[1, 2, 3],[4, 5, 6]])
>>> c
array([[1, 2, 3],
       [4, 5, 6]])
```

The array() function in addition to the lists can accept even tuples and sequences of tuples.

```
>>> d = np.array(((1, 2, 3),(4, 5, 6)))
>>> d
array([[1, 2, 3],
       [4, 5, 6]])
```

and also, even sequences of tuples and lists interconnected make no difference.

```
>>> e = np.array([(1, 2, 3), [4, 5, 6], (7, 8, 9)])
>>> e
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

Types of Data

So far you have used only numeric values as simple integer and float, but NumPy arrays are designed to contain a wide variety of data types (see Table 3-1). For example, you can use the data type string:

```
>>> g = np.array([['a', 'b'],['c', 'd']])
>>> g
array([['a', 'b'],
          ['c', 'd']],
      dtype='|S1')
>>> g.dtype
dtype('S1')
>>> g.dtype.name
'string8'
```

Table 3-1. *Data Types Supported by NumPy*

Data type	Description
bool_	Boolean (True or False) stored as a byte
int_	Default integer type (same as C long; normally either int64 or int32)
intc	Identical to C int (normally int32 or int64)
intp	Integer used for indexing (same as C size_t; normally either int32 or int64)
int8	Byte (–128 to 127)
int16	Integer (–32768 to 32767)
int32	Integer (–2147483648 to 2147483647)
int64	Integer (–9223372036854775808 to 9223372036854775807)
uint8	Unsigned integer (0 to 255)
uint16	Unsigned integer (0 to 65535)
uint32	Unsigned integer (0 to 4294967295)
uint64	Unsigned integer (0 to 18446744073709551615)
float_	Shorthand for float64
float16	Half precision float: sign bit, 5-bit exponent, 10-bit mantissa
float32	Single precision float: sign bit, 8-bit exponent, 23-bit mantissa
float64	Double precision float: sign bit, 11-bit exponent, 52-bit mantissa
complex_	Shorthand for complex128
complex64	Complex number, represented by two 32-bit floats (real and imaginary components)
complex128	Complex number, represented by two 64-bit floats (real and imaginary components)

The dtype Option

The array() function does not accept a single argument. You have seen that each ndarray object is associated with a dtype object that uniquely defines the type of data that will occupy each item in the array. By default, the array() function is able to associate the most suitable type according to the values contained in the sequence of lists or tuples used. Actually, you can explicitly define the dtype using the **dtype** option as argument of the function.

For example if you want to define an array with complex values you can use the dtype option as follows:

```
>>> f = np.array([[1, 2, 3],[4, 5, 6]], dtype=complex)
>>> f
array([[ 1.+0.j,  2.+0.j,  3.+0.j],
       [ 4.+0.j,  5.+0.j,  6.+0.j]])
```

Intrinsic Creation of an Array

The NumPy library provides a set of functions that generate the ndarrays with an initial content, created with some different values depending on the function. Throughout the chapter, but also throughout the book, you'll discover that these features will be very useful. In fact, they allow a single line of code to generate large amounts of data.

The **zeros()** function, for example, creates a full array of zeros with dimensions defined by the shape argument. For example, to create a two-dimensional array 3x3:

```
>>> np.zeros((3, 3))
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

while the **ones()** function creates an array full of ones in a very similar way.

```
>>> np.ones((3, 3))
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

By default, the two functions have created arrays with float64 data type. A feature that will be particularly useful is **arange()**. This function generates NumPy arrays with numerical sequences that respond to particular rules depending on the passed arguments. For example, if you want to generate a sequence of values between 0 and 10, you will be passed only one argument to the function, that is the value with which you want to end the sequence.

```
>>> np.arange(0, 10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

If instead of starting from zero you want to start from another value, simply specify two arguments: the first is the starting value and the second is the final value.

```
>>> np.arange(4, 10)
array([4, 5, 6, 7, 8, 9])
```

It is also possible to generate a sequence of values with precise intervals between them. If the third argument of the **arange()** function is specified, this will represent the gap between a value and the next one in the sequence of values.

```
>>> np.arange(0, 12, 3)
array([0, 3, 6, 9])
```

In addition, this third argument can also be a float.

```
>>> np.arange(0, 6, 0.6)
array([ 0. ,  0.6,  1.2,  1.8,  2.4,  3. ,  3.6,  4.2,  4.8,  5.4])
```

But so far you have only created one-dimensional arrays. To generate two-dimensional arrays you can still continue to use the **arange()** function but combined with the **reshape()** function. This function divides a linear array in different parts in the manner specified by the shape argument.

```
>>> np.arange(0, 12).reshape(3, 4)
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

Another function very similar to arange() is **linspace()**. This function still takes as its first two arguments the initial and end values of the sequence, but the third argument, instead of specifying the distance between one element and the next, defines the number of elements into which we want the interval to be split.

```
>>> np.linspace(0,10,5)
array([ 0. ,  2.5,  5. ,  7.5, 10. ])
```

Finally, another method to obtain arrays already containing values is to fill them with random values. This is possible using the **random()** function of the **numpy.random** module. This function will generate an array with many elements as specified in the argument.

```
>>> np.random.random(3)
array([ 0.78610272, 0.90630642, 0.80007102])
```

The numbers obtained will vary with every run. To create a multidimensional array simply pass the size of the array as an argument.

```
>>> np.random.random((3,3))
array([[ 0.07878569, 0.7176506 , 0.05662501],
       [ 0.82919021, 0.80349121, 0.30254079],
       [ 0.93347404, 0.65868278, 0.37379618]])
```

Basic Operations

So far you have seen how to create a new NumPy array and how the items are defined within it. Now it is the time to see how to apply various operations on them.

Arithmetic Operators

The first operations that you will perform on arrays are the application of arithmetic operators. The most obvious are the sum or the multiplication of an array with a scalar.

```
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])

>>> a+4
array([4, 5, 6, 7])
>>> a*2
array([0, 2, 4, 6])
```

These operators can also be used between two arrays. In NumPy, these operations are **element-wise**, that is, the operators are applied only between corresponding elements, namely, that occupy the same position, so that at the end as a result there will be a new array containing the results in the same location of the operands (see Figure 3-1).
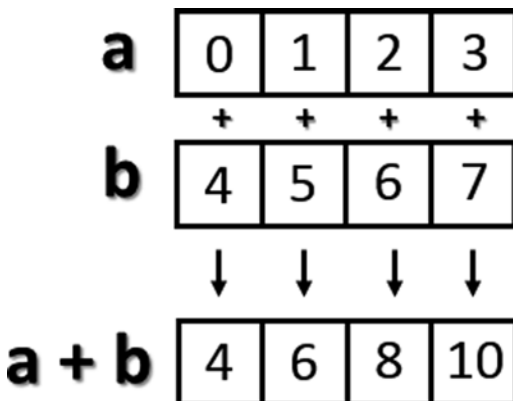
Figure 3-1. *Element-wise addition*

```
>>> b = np.arange(4,8)
>>> b
array([4, 5, 6, 7])
>>> a + b
array([ 4,  6,  8, 10])
>>> a - b
array([-4, -4, -4, -4])
>>> a * b
array([ 0,  5, 12, 21])
```

Moreover, these operators are also available for functions, provided that the value returned is a NumPy array. For example, you can multiply the array with the sine or the square root of the elements of the array b.

```
>>> a * np.sin(b)
array([-0.        , -0.95892427, -0.558831  ,  1.9709598 ])
>>> a * np.sqrt(b)
array([ 0.        ,  2.23606798,  4.89897949,  7.93725393])
```

Moving on to the multidimensional case, even here the arithmetic operators continue to operate element-wise.

```
>>> A = np.arange(0, 9).reshape(3, 3)
>>> A
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> B = np.ones((3, 3))
>>> B
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>> A * B
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.],
       [ 6.,  7.,  8.]])
```

The Matrix Product

The choice of operating element-wise is a peculiar aspect of the NumPy library. In fact in many other tools for data analysis, the * operator is understood as **matrix product** when it is applied to two matrices. Using NumPy, this kind of product is instead indicated by the **dot()** function. This operation is not element-wise.

```
>>> np.dot(A,B)
array([[  3.,   3.,   3.],
       [ 12.,  12.,  12.],
       [ 21.,  21.,  21.]])
```

The result at each position is the sum of the products between each element of the corresponding row of the first matrix with the corresponding element of the corresponding column of the second matrix. However, Figure 3-2 illustrates the process carried out during the matrix product (only run for two elements).

$$\sum_i a_i * b_i =$$
$$0 * 1 + 1 * 1 + 2 * 1 = 3$$

$$\sum_i a_i * b_i =$$
$$3 * 1 + 4 * 1 + 5 * 1 = 12$$
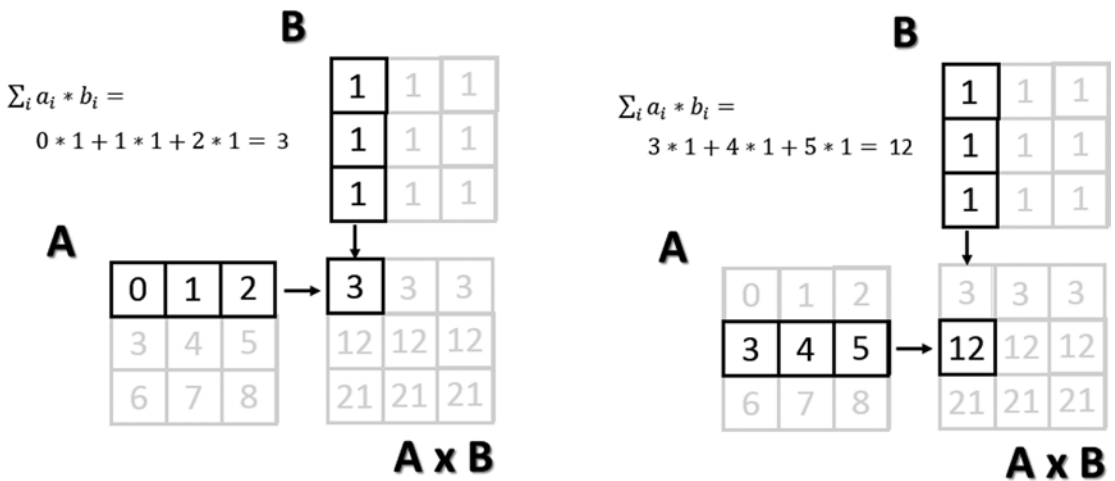
Figure 3-2. *The calculation of matrix elements as result of a matrix product*

An alternative way to write the matrix product is to see the dot() function as an object's function of one of the two matrices.

```
>>> A.dot(B)
array([[  3.,   3.,   3.],
       [ 12.,  12.,  12.],
       [ 21.,  21.,  21.]])
```

I want to add that since the matrix product is not a commutative operation, then the order of the operands is important. Indeed A * B is not equal to B * A.

```
>>> np.dot(B,A)
array([[  9.,  12.,  15.],
       [  9.,  12.,  15.],
       [  9.,  12.,  15.]])
```

Increment and Decrement Operators

Actually, there is no such operators in Python, since there are no operators ++ or --. To increase or decrease the values you have to use operators such as += or -=. These operators are not different from those that we saw earlier, except that instead of creating a new array with the results, they will reassign the results to the same array.

```
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> a += 1
>>> a
array([1, 2, 3, 4])
>>> a -= 1
>>> a
array([0, 1, 2, 3])
```

Therefore, the use of these operators is much more extensive than the simple incremental operators that increase the values by one unit, and then they can be applied in many cases. For instance, you need them every time you want to change the values in an array without generating a new one.

```
array([0, 1, 2, 3])
>>> a += 4
>>> a
array([4, 5, 6, 7])
>>> a *= 2
>>> a
array([ 8, 10, 12, 14])
```

Universal Functions (ufunc)

A universal function, generally called **ufunc**, is a function operating of an array in an element-by-element fashion. This means that it is a function that acts individually on each single element of the input array to generate a corresponding result in a new output array. At the end, you will obtain an array of the same size of the input.

There are many mathematical and trigonometric operations that meet this definition, for example, the calculation of the square root with **sqrt()**, the logarithm with **log()**, or the sin with **sin()**.

```
>>> a = np.arange(1, 5)
>>> a
array([1, 2, 3, 4])
>>> np.sqrt(a)
array([ 1.        , 1.41421356, 1.73205081, 2.        ])
>>> np.log(a)
array([ 0.        , 0.69314718, 1.09861229, 1.38629436])
>>> np.sin(a)
array([ 0.84147098, 0.90929743, 0.14112001, -0.7568025 ])
```

Many functions are already implemented within the library NumPy.

Aggregate Functions

Aggregate functions are those functions that perform an operation on a set of values, an array for example, and produce a single result. Therefore, the sum of all the elements in an array is an aggregate function. Many functions of this kind are implemented within the class ndarray.

```
>>> a = np.array([3.3, 4.5, 1.2, 5.7, 0.3])
>>> a.sum()
15.0
>>> a.min()
0.29999999999999999
>>> a.max()
5.7000000000000002
>>> a.mean()
3.0
>>> a.std()
2.0079840636817816
```

Indexing, Slicing, and Iterating

In the previous sections you have seen how to create an array and how to perform operations on it. In this section you will see how to manipulate these objects, how to select some elements through indexes and slices, in order to obtain the views of the values contained within them or to make assignments in order to change the value in them. Finally, you will also see how you can make the iterations within them.

Indexing

Array indexing always refers to the use of square brackets ('[ ]') to index the elements of the array so that it can then be referred individually for various uses such as extracting a value, selecting items, or even assigning a new value.

When you create a new array, an appropriate scale index is also automatically created (see Figure 3-3).
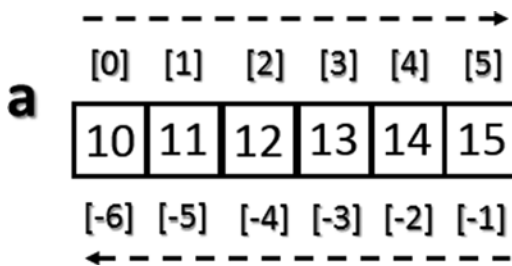
Figure 3-3. *The indexing of an ndarray monodimensional*

In order to access a single element of an array you can refer to its index.

```
>>> a = np.arange(10, 16)
>>> a
array([10, 11, 12, 13, 14, 15])
>>> a[4]
14
```

The NumPy arrays also accept negative indexes. These indexes have the same incremental sequence from 0 to –1, –2, and so on, but in practice they start to refer the final element to move gradually towards the initial element, which will be the one with the more negative index value.

```
>>> a[-1]
15
>>> a[-6]
10
```

To select multiple items at once, you can pass array of indexes within the square brackets.

```
>>> a[[1, 3, 4]]
array([11, 13, 14])
```

Moving on to the two-dimensional case, namely, the matrices, they are represented as rectangular arrays consisting of rows and columns, defined by two axes, where axis 0 is represented by the rows and axis 1 is represented by the columns. Thus, indexing in this case is represented by a pair of values: the first value is the index of the row and the second is the index of the column. Therefore, if you want to access the values or to select elements within the matrix you will still use square brackets, but this time the values are two [row index, column index] (Figure 3-4).

Figure 3-4. *The indexing of a bidimensional array*

```
>>> A = np.arange(10, 19).reshape((3, 3))
>>> A
array([[10, 11, 12],
       [13, 14, 15],
       [16, 17, 18]])
```

So if you want to remove the element of the third column in the second row, you have to insert the pair [1, 2].

```
>>> A[1, 2]
15
```

Slicing

Slicing is the operation which allows you to extract portions of an array to generate new ones. Whereas using the Python lists the arrays obtained by slicing are copies, in NumPy, arrays are views onto the same underlying buffer.

Depending on the portion of the array that you want to extract (or view) you must make use of the slice syntax; that is, you will use a sequence of numbers separated by colons (':') within the square brackets.

If you want to extract a portion of the array, for example one that goes from the second to the sixth element, then you have to insert the index of the starting element, that is 1, and the index of the final element, that is 5, separated by ':'.

```
>>> a = np.arange(10, 16)
>>> a
array([10, 11, 12, 13, 14, 15])
>>> a[1:5]
array([11, 12, 13, 14])
```

Now if you want to extract from the previous portion an item, skip a specific number of following items, then extract the next, and skip again ..., you can use a third number that defines the gap in the sequence of the elements between one element and the next one to take. For example, with a value of 2, the array will take the elements in an alternating fashion.

```
>>> a[1:5:2]
array([11, 13])
```

To better understand the slice syntax, you also should look at cases where you do not use explicit numerical values. If you omit the first number, then implicitly NumPy interprets this number as 0 (i.e., the initial element of the array); if you omit the second number, this will be interpreted as the maximum index of the array; and if you omit the last number this will be interpreted as 1, and then all elements will be considered without intervals.

```
>>> a[::2]
array([10, 12, 14])
>>> a[:5:2]
array([10, 12, 14])
>>> a[:5:]
array([10, 11, 12, 13, 14])
```

As regards the case of two-dimensional array, the slicing syntax still applies, but it is separately defined both for the rows and for the columns. For example if you want to extract only the first row:

```
>>> A = np.arange(10, 19).reshape((3, 3))
>>> A
array([[10, 11, 12],
       [13, 14, 15],
       [16, 17, 18]])
>>> A[0,:]
array([10, 11, 12])
```

As you can see in the second index, if you leave only the colon without defining any number, then you will select all the columns. Instead, if you want to extract all the values of the first column, you have to write the inverse.

```
>>> A[:,0]
array([10, 13, 16])
```

Instead, if you want to extract a smaller matrix then you need to explicitly define all intervals with indexes that define them.

```
>>> A[0:2, 0:2]
array([[10, 11],
       [13, 14]])
```

If the indexes of the rows or columns to be extracted are not contiguous, you can specify an array of indexes.

```
>>> A[[0,2], 0:2]
array([[10, 11],
       [16, 17]])
```

Iterating an Array

In Python, the iteration of the items in an array is really very simple; you just need to use the for construct.

```
>>> for i in a:
...     print i
...
10
11
12
13
14
15
```

Of course, even here, moving to the two-dimensional case, you could think of applying the solution of two nested loops with the **for** construct. The first loop will scan the rows of the array, and the second loop will scan the columns. Actually, if you apply the for loop to a matrix, you will find out that it will always perform a scan according to the first axis.

```
>>> for row in A:
...     print row
...
[10 11 12]
[13 14 15]
[16 17 18]
```

If you want to make an iteration element by element you may use the following construct, using the for loop on **A.flat**.

```
>>> for item in A.flat:
...        print item
...
10
11
12
13
14
15
16
17
18
```

However, despite all this, NumPy offers us an alternative and more elegant solution than the for loop. Generally, you need to apply an iteration to apply a function on the rows or on the columns or on an individual item. If you want to launch an aggregate function that returns a value calculated for every single column or on every single row, there is an optimal way to makes that it will be entirely NumPy to manage the iteration: the **apply_along_axis()** function.

This function takes three arguments: the aggregate function, the axis on which to apply the iteration, and finally the array. If the option **axis** equals 0, then the iteration evaluates the elements column by column, whereas if the **axis** equals 1 then the iteration evaluates the elements row by row. For example, you can calculate the average of the values for the first by column and then by row.

```
>>> np.apply_along_axis(np.mean, axis=0, arr=A)
array([ 13.,  14.,  15.])
>>> np.apply_along_axis(np.mean, axis=1, arr=A)
array([ 11.,  14.,  17.])
```

In the previous case, you used a function already defined within the NumPy library, but nothing prevents you from defining functions on their own. You also used an aggregate function. However, nothing forbids us from using an ufunc. In this case, using the iteration both by column and by row, the end result is the same. In fact, using a ufunc is how to perform one iteration element-by-element.

```
>>> def foo(x):
...      return x/2
...
>>> np.apply_along_axis(foo, axis=1, arr=A)
array([[5, 5, 6],
       [6, 7, 7],
       [8, 8, 9]])
>>> np.apply_along_axis(foo, axis=0, arr=A)
array([[5, 5, 6],
       [6, 7, 7],
       [8, 8, 9]])
```

As you can see, the ufunc function halves the value of each element of the input array regardless of whether the iteration is performed by row or by column.

Conditions and Boolean Arrays

So far you have used the indexing and the slicing to select or extract a subset of the array. These methods make use of the indexes in a numerical form. An alternative way to perform the selective extraction of the elements in an array is to use the conditions and Boolean operators.

See this alternative method in detail. For example, suppose you want to select all the values less than 0.5 in a 4x4 matrix containing random numbers between 0 and 1.

```
>>> A = np.random.random((4, 4))
>>> A
array([[ 0.03536295,  0.0035115 ,  0.54742404,  0.68960999],
       [ 0.21264709,  0.17121982,  0.81090212,  0.43408927],
       [ 0.77116263,  0.04523647,  0.84632378,  0.54450749],
       [ 0.86964585,  0.6470581 ,  0.42582897,  0.22286282]])
```

Once a matrix of random numbers is defined, if you apply an operator condition, that is, as we said the operator greater, you will receive as a return value Boolean array containing True values in the positions in which the condition is satisfied, that is, all the positions in which the values are less than 0.5.

```
>>> A < 0.5
array([[ True,  True, False, False],
       [ True,  True, False,  True],
       [False,  True, False, False],
       [False, False,  True,  True]], dtype=bool)
```

Actually, the Boolean arrays are used implicitly for making selections of parts of arrays. In fact, by inserting the previous condition directly inside the square brackets, you will extract all elements smaller than 0.5, so as to obtain a new array.

```
>>> A[A < 0.5]
array([ 0.03536295,  0.0035115 ,  0.21264709,  0.17121982,  0.43408927,
        0.04523647,  0.42582897,  0.22286282])
```

Shape Manipulation

You have already seen during the creation of a two-dimensional array how it is possible to convert a one-dimensional array into a matrix, thanks to the **reshape()** function.

```
>>> a = np.random.random(12)
>>> a
array([ 0.77841574,  0.39654203,  0.38188665,  0.26704305,  0.27519705,
        0.78115866,  0.96019214,  0.59328414,  0.52008642,  0.10862692,
        0.41894881,  0.73581471])
>>> A = a.reshape(3, 4)
>>> A
array([[ 0.77841574,  0.39654203,  0.38188665,  0.26704305],
       [ 0.27519705,  0.78115866,  0.96019214,  0.59328414],
       [ 0.52008642,  0.10862692,  0.41894881,  0.73581471]])
```

The **reshape()** function returns a new array and therefore it is useful to create new objects. However if you want to modify the object by modifying the shape, you have to assign a tuple containing the new dimensions directly to its **shape** attribute.

```
>>> a.shape = (3, 4)
>>> a
array([[ 0.77841574,  0.39654203,  0.38188665,  0.26704305],
       [ 0.27519705,  0.78115866,  0.96019214,  0.59328414],
       [ 0.52008642,  0.10862692,  0.41894881,  0.73581471]])
```

As you can see, this time it is the starting array to change shape and there is no object returned. The inverse operation is possible, that is, to convert a two-dimensional array into a one-dimensional array, through the **ravel()** function.

```
>>> a = a.ravel()
array([ 0.77841574,  0.39654203,  0.38188665,  0.26704305,  0.27519705,
        0.78115866,  0.96019214,  0.59328414,  0.52008642,  0.10862692,
        0.41894881,  0.73581471])
```

or even here acting directly on the **shape** attribute of the array itself.

```
>>> a.shape = (12)
>>> a
array([ 0.77841574,  0.39654203,  0.38188665,  0.26704305,  0.27519705,
        0.78115866,  0.96019214,  0.59328414,  0.52008642,  0.10862692,
        0.41894881,  0.73581471])
```

Another important operation is the transposition of a matrix that is an inversion of the columns with rows. NumPy provides this feature with the **transpose()** function.

```
>>> A.transpose()
array([[ 0.77841574,  0.27519705,  0.52008642],
       [ 0.39654203,  0.78115866,  0.10862692],
       [ 0.38188665,  0.96019214,  0.41894881],
       [ 0.26704305,  0.59328414,  0.73581471]])
```

Array Manipulation

Often you need to create an array using already created arrays. In this section, you will see how to create new arrays by joining or splitting arrays that are already defined.

Joining Arrays

You can merge multiple arrays together to form a new one that contains all of them. NumPy uses the concept of stacking, providing a number of functions in this regard. For example, you can run the vertical stacking with the **vstack()** function, which combines the second array as new rows of the first array. In this case you have a growth of the array in the vertical direction. By contrast, the **hstack()** function performs horizontal stacking; that is, the second array is added to the columns of the first array.

```
>>> A = np.ones((3, 3))
>>> B = np.zeros((3, 3))
>>> np.vstack((A, B))
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>> np.hstack((A,B))
array([[ 1.,  1.,  1.,  0.,  0.,  0.],
       [ 1.,  1.,  1.,  0.,  0.,  0.],
       [ 1.,  1.,  1.,  0.,  0.,  0.]])
```

Two other functions performing stacking between multiple arrays are **column_stack()** and **row_stack()**. These functions operate differently than the two previous functions. Generally these functions are used with one-dimensional arrays that are stacked as columns or rows in order to form a new two-dimensional array.

```
>>> a = np.array([0, 1, 2])
>>> b = np.array([3, 4, 5])
>>> c = np.array([6, 7, 8])
>>> np.column_stack((a, b, c))
array([[0, 3, 6],
       [1, 4, 7],
       [2, 5, 8]])
>>> np.row_stack((a, b, c))
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

Splitting Arrays

First you saw how to assemble multiple arrays to each other through the operation of stacking. Now you will see the opposite operation, that is, to divide an array into several parts. In NumPy, to do this you will use splitting. Here too, you have a set of functions that work both horizontally with the **hsplit()** function and vertically with the **vsplit()** function.

```
>>> A = np.arange(16).reshape((4, 4))
>>> A
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

Thus, if you want to split the array horizontally, meaning the width of the array divided into two parts, the 4x4 matrix A will be split into two 2x4 matrices.

```
>>> [B,C] = np.hsplit(A, 2)
>>> B
array([[ 0,  1],
       [ 4,  5],
       [ 8,  9],
       [12, 13]])
>>> C
array([[ 2,  3],
       [ 6,  7],
       [10, 11],
       [14, 15]])
```

Instead, if you want to split the array vertically, meaning the height of the array divided into two parts, the 4x4 matrix A will be split into two 4x2 matrices.

```
>>> [B,C] = np.vsplit(A, 2)
>>> B
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
>>> C
array([[ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

A more complex command is the **split()** function, which allows you to split the array into nonsymmetrical parts. In addition, passing the array as an argument, you have also to specify the indexes of the parts to be divided. If you use the option **axis** = 1, then the indexes will be those the columns; if instead the option is axis = 0, then they will be the row indexes.

For example, if you want to divide the matrix into three parts, the first of which will include the first column, the second will include the second and the third column, and the third will include the last column, then you must specify three indexes in the following way.

```
>>> [A1,A2,A3] = np.split(A,[1,3],axis=1)
>>> A1
array([[ 0],
       [ 4],
       [ 8],
       [12]])
>>> A2
array([[ 1,  2],
       [ 5,  6],
       [ 9, 10],
       [13, 14]])
>>> A3
array([[ 3],
       [ 7],
       [11],
       [15]])
```

You can do the same thing by row.

```
>>> [A1,A2,A3] = np.split(A,[1,3],axis=0)
>>> A1
array([[0, 1, 2, 3]])
>>> A2
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> A3
array([[12, 13, 14, 15]])
```

This feature also includes the functionalities of the **vsplit()** and **hsplit()** functions.

General Concepts

This section describes the general concepts underlying NumPy library. The difference between copies and views will be illustrated especially when they return values. Also the mechanism of broadcasting that occurs implicitly in many transactions with the functions of NumPy will be covered in this section.

Copies or Views of Objects

As you may have noticed with NumPy, especially when you are performing operations or manipulations on the array, you can have as a return value either a copy or a view of the array. In NumPy, all assignments do not produce copies of arrays, nor any element contained within them.

```
>>> a = np.array([1, 2, 3, 4])
>>> b = a
>>> b
array([1, 2, 3, 4])
>>> a[2] = 0
>>> b
array([1, 2, 0, 4])
```

If you assign one array a to another array b, actually you are not doing a copy but b is just another way to call array a. In fact, by changing the value of the third element you change the third value of b too. When you perform the slicing of an array, actually the object returned is only a view of the original array.

```
>>> c = a[0:2]
>>> c
array([1, 2])
>>> a[0] = 0
>>> c
array([0, 2])
```

As you can see, even with the slicing, you are actually pointing to the same object. If you want to generate a complete copy and distinct array you can use the **copy()** function.

```
>>> a = np.array([1, 2, 3, 4])
>>> c = a.copy()
>>> c
array([1, 2, 3, 4])
>>> a[0] = 0
>>> c
array([1, 2, 3, 4])
```

In this case, even changing the items in array a, array c remains unchanged.

Vectorization

Vectorization is a concept that, along with the broadcasting, is the basis of the internal implementation of NumPy. The vectorization is the absence of explicit loop during the developing of the code. These loops actually cannot be omitted, but are implemented internally and then they are replaced by other constructs in the code. The application of vectorization leads to a more concise and readable code, and you can say that it will appear more "Pythonic" in its appearance. In fact, thanks to the vectorization, many operations take on a more mathematical expression, for example NumPy allows you to express the multiplication of two arrays as shown:

```
a * b
```

or even two matrices:

```
A * B
```

In other languages, such operations would be expressed with many nested loops with the for construct. For example, the first operation would be expressed in the following way:

```
for (i = 0; i < rows; i++){
  c[i] = a[i]*b[i];
}
```

While the product of matrices would be expressed as follows:

```
for( i=0; i < rows; i++){
   for(j=0; j < columns; j++){
      c[i][j] = a[i][j]*b[i][j];
   }
}
```

From all this, it is clear that using NumPy the code is more readable and especially expressed in a more mathematical way.

Broadcasting

Broadcasting is the operation that allows an operator or a function to act on two or more arrays to operate even if these arrays do not have exactly the same shape. Actually, not all the dimensions are compatible with each other in order to be subjected to broadcasting but they must meet certain rules.

You saw that using NumPy, you can classify multidimensional arrays through the shape that is a tuple representing the length of the elements for each dimension.

Thus, two arrays may be subjected to broadcasting when all their dimensions are compatible, i.e., the length of each dimension must be equal between the two arrays or one of them must be equal to 1. If these two conditions are not met, you get an exception given that the two arrays are not compatible.

```
>>> A = np.arange(16).reshape(4, 4)
>>> b = np.arange(4)
>>> A
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
>>> b
array([0, 1, 2, 3])
```

In this case, you obtain two arrays:

```
4 x 4
4
```

The rules of broadcasting are two. The first rule is to add a 1 to each missing dimension. If the compatibility rules are now satisfied you can apply the broadcasting and move to the second rule.

```
4 x 4
4 x 1
```

The rule of compatibility is met. Then you can move on to the second rule of broadcasting. This rule explains how to extend the size of the smallest array so that it takes on the same size of the biggest, so that then the element-wise function or operator is applicable.

The second rule assumes that the missing elements (size, length 1) are filled with replicas of the values contained in extended sizes (see Figure 3-5).
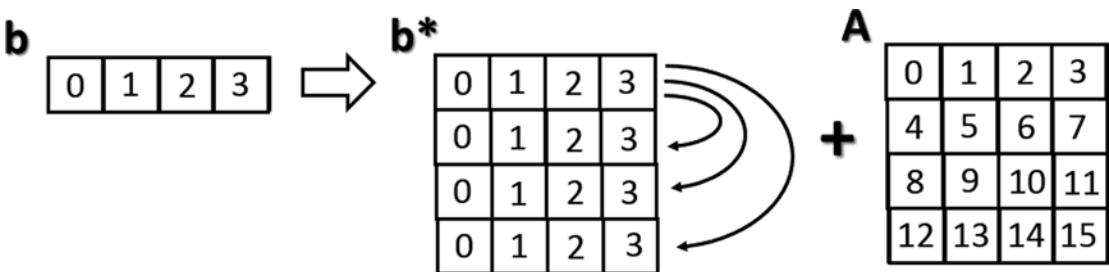
Figure 3-5. *An application of the second broadcasting rule*

Now that the two arrays have the same dimensions, the values inside may be added together.

```
>>> A + b
array([[ 0,  2,  4,  6],
       [ 4,  6,  8, 10],
       [ 8, 10, 12, 14],
       [12, 14, 16, 18]])
```

In this case you are in a simple case in which one of the two arrays is smaller than the other. There may be more complex cases in which the two arrays have different shapes but each of them is smaller than the other only for some dimensions.

```
>>> m = np.arange(6).reshape(3, 1, 2)
>>> n = np.arange(6).reshape(3, 2, 1)
>>> m
array([[[0, 1]],

       [[2, 3]],

       [[4, 5]]])
>>> n
array([[[0],
        [1]],

       [[2],
        [3]],

       [[4],
        [5]]])
```

Even in this case, by analyzing the shapes of the two arrays, you can see that they are compatible and therefore the rules of broadcasting can be applied.

```
3 x 1 x 2
3 x 2 x 1
```

In this case both undergo the extension of dimensions (broadcasting).

```
m* = [[[0,1],              n* = [[[0,0],
       [0,1]],                    [1,1]],
      [[2,3],                    [[2,2],
       [2,3]],                    [3,3]],
      [[4,5],                    [[4,4],
       [4,5]]]                    [5,5]]]
```

And then you can apply, for example, the addition operator between the two arrays, operating element-wise.

```
>>> m + n
array([[[ 0,  1],
        [ 1,  2]],

       [[ 4,  5],
        [ 5,  6]],

       [[ 8,  9],
        [ 9, 10]]])
```

Structured Arrays

So far in the various examples in previous sections, you saw monodimensional and two-dimensional arrays. Actually, NumPy provides the possibility of creating arrays that are much more complex not only in size, but in the structure itself, called precisely **structured arrays**. This type of array contains structs or records instead of the individual items.

For example, you can create a simple array of structs as items. Thanks to the **dtype** option, you can specify a list of comma-separated specifiers to indicate the elements that will constitute the struct, along with its data type and order.

```
bytes                 b1
int                   i1, i2, i4, i8
unsigned ints         u1, u2, u4, u8
floats                f2, f4, f8
complex               c8, c16
fixed length strings  a<n>
```

For example, if you want to specify a struct consisting of an integer, a character string of length 6 and a Boolean value, you will specify the three types of data in the dtype option with the right order using the corresponding specifiers.

```
>>> structured = np.array([(1, 'First', 0.5, 1+2j),(2, 'Second', 1.3, 2-2j),
(3, 'Third', 0.8, 1+3j)],dtype=('i2, a6, f4, c8'))
>>> structured
array([(1, 'First', 0.5, (1+2j)),
       (2, 'Second', 1.2999999523162842, (2-2j)),
       (3, 'Third', 0.800000011920929, (1+3j))],
      dtype=[('f0', '<i2'), ('f1', 'S6'), ('f2', '<f4'), ('f3', '<c8')])
```

You may also use the data type explicitly specifying int8, uint8, float16, complex64, and so forth.

```
>>> structured = np.array([(1, 'First', 0.5, 1+2j),(2, 'Second', 1.3,2-2j),
(3, 'Third', 0.8, 1+3j)],dtype=('
int16, a6, float32, complex64'))
>>> structured
array([(1, 'First', 0.5, (1+2j)),
       (2, 'Second', 1.2999999523162842, (2-2j)),
       (3, 'Third', 0.800000011920929, (1+3j))],
      dtype=[('f0', '<i2'), ('f1', 'S6'), ('f2', '<f4'), ('f3', '<c8')])
```

However, both cases have the same result. Inside the array you see a dtype sequence containing the name of each item of the struct with the corresponding type of data.

Writing the appropriate reference index, you obtain the corresponding row which contains the struct.

```
>>> structured[1]
(2, 'Second', 1.2999999523162842, (2-2j))
```

The names that are assigned automatically to each item of struct can be considered as the names of the columns of the array, and then using them as a structured index, you can refer to all the elements of the same type, or of the same 'column'.

```
>>> structured['f1']
array(['First', 'Second', 'Third'],
      dtype='|S6')
```

As you have just seen, the names are assigned automatically with an **f** (which stands for field) and a progressive integer that indicates the position in the sequence. In fact, it would be more useful to specify the names with something more meaningful. This is possible and you can do it at the time of the declaration of the array:

```
>>> structured = np.array([(1,'First',0.5,1+2j),(2,'Second',1.3,2-2j),(3,'Third',0.8,1+3j)],
dtype=[('id','i2'),('position','a6'),('value','f4'),('complex','c8')])
>>> structured
array([(1, 'First', 0.5, (1+2j)),
       (2, 'Second', 1.2999999523162842, (2-2j)),
       (3, 'Third', 0.800000011920929, (1+3j))],
      dtype=[('id', '<i2'), ('position', 'S6'), ('value', '<f4'), ('complex', '<c8')])
```

or at a later time, redefining the tuples of names assigned to the dtype attribute of the structured array.

```
>>> structured.dtype.names = ('id','order','value','complex')
```

Now you can use meaningful names for the various types of fields:

```
>>> structured['order']
array(['First', 'Second', 'Third'],
      dtype='|S6')
```

Reading and Writing Array Data on Files

A very important aspect of NumPy that has not been taken into account yet is the reading of the data contained within a file. This procedure is very useful, especially when you have to deal with large amounts of data collected within arrays. This is a very common operation in data analysis, since the size of the dataset to be analyzed is almost always huge, and therefore it is not advisable or even possible to manage the transcription and subsequent reading of data by a computer to another manually, or from one session of the calculation to another.

Indeed NumPy provides in this regard a set of functions that allow the data analyst to save the results of his calculations in a text or binary file. Similarly, NumPy allows reading and conversion of written data within a file into an array.

Loading and Saving Data in Binary Files

Regarding for saving and then later retrieving data stored in binary format, NumPy provides a pair of functions called **save()** and **load()**.

Once you have an array to save, for example, containing the results of your processing during data analysis, you simply call the save() function, specifying as arguments the name of the file, to which **.npy** extension will be automatically added, and then the array itself.

```
>>> data
array([[ 0.86466285,  0.76943895,  0.22678279],
       [ 0.12452825,  0.54751384,  0.06499123],
       [ 0.06216566,  0.85045125,  0.92093862],
       [ 0.58401239,  0.93455057,  0.28972379]])
>>> np.save('saved_data',data)
```

But when you need to recover the data stored within a .npy file, you can use the load() function by specifying the file name as argument, this time adding the extension .npy.

```
>>> loaded_data = np.load('saved_data.npy')
>>> loaded_data
array([[ 0.86466285,  0.76943895,  0.22678279],
       [ 0.12452825,  0.54751384,  0.06499123],
       [ 0.06216566,  0.85045125,  0.92093862],
       [ 0.58401239,  0.93455057,  0.28972379]])
```

Reading File with Tabular Data

Many times, the data that you want to read or save are in textural format (TXT or CSV, for example). Generally, you decide to save the data in this format, instead of binary, because the files can be accessed outside independently if you are working with NumPy or with any other application. Take for example the case of a set of data in CSV (Comma-Separated Values) format, in which data are collected in a tabular form and where all values are separated from each other by commas (see Listing 3-1).

Listing 3-1. data.csv

```
id,value1,value2,value3
1,123,1.4,23
2,110,0.5,18
3,164,2.1,19
```

To be able to read your data in a text file and insert them into an array, NumPy provides a function called **genfromtxt()**. Normally, this function takes three arguments, including the name of the file containing the data, the character that separates a value from another which in our case is a paragraph, and whether it contains the column headers.

```
>>> data = np.genfromtxt('data.csv', delimiter=',', names=True)
>>> data
array([(1.0, 123.0, 1.4, 23.0), (2.0, 110.0, 0.5, 18.0),
       (3.0, 164.0, 2.1, 19.0)],
      dtype=[('id', '<f8'), ('value1', '<f8'), ('value2', '<f8'), ('value3', '<f8')])
```

As we can see from the result, you get a structured array in which the column headings have become the names of the field.

This function performs implicitly two loops: the first reads a line at a time, and the second separates and converts the values contained in it, inserting the consecutive elements created specifically. One positive aspect of this feature is that if within the file there are some missing data, the function is able to handle them.

Take for example the previous file (see Listing 3-2) and remove some items. Save as **data2.csv**.

Listing 3-2. data2.csv

```
id,value1,value2,value3
1,123,1.4,23
2,110,,18
3,,2.1,19
```

Launching these commands, you can see how the genfromtxt() function replaces the blanks in the file with **nan** values.

```
>>> data2 = np.genfromtxt('data2.csv', delimiter=',', names=True)
>>> data2
array([(1.0, 123.0, 1.4, 23.0), (2.0, 110.0, nan, 18.0),
       (3.0, nan, 2.1, 19.0)],
      dtype=[('id', '<f8'), ('value1', '<f8'), ('value2', '<f8'), ('value3', '<f8')])
```

At the bottom of the array, you can find the column headings contained in the file. These headers can be considered as labels which act as indexes to extract the values by column:

```
>>> data2['id']
array([ 1.,  2.,  3.])
```

Instead, using the numerical indexes in the classic way you will extract data corresponding to the rows.

```
>>> data2[0]
(1.0, 123.0, 1.4, 23.0)
```

Conclusions

In this chapter, you saw all the main aspects of the NumPy library and through a series of examples you got familiar with a range of features that form the basis of many other aspects you'll face in the course of the book. In fact, many of these concepts will be taken from other scientific and computing libraries that are more specialized, but that have been structured and developed on the basis of this library.

You saw how thanks to the ndarray you can extend the functionalities of Python, making it a suitable language for scientific computing and in a particular way for data analysis.

Knowledge of NumPy proves therefore to be crucial for anyone who wants to take on the world of the data analysis.

In the next chapter, we will begin to introduce a new library, pandas, that being structured on NumPy will encompass all the basic concepts illustrated in this chapter, but extending them to make them more suitable for data analysis.

■ ■ ■

The pandas Library—An Introduction

With this chapter, you can finally get into the heart of this book: the pandas library. This fantastic Python library is a perfect tool for anyone who wants to practice data analysis using Python as a programming language.

First you will find out the fundamental aspects of this library and how to install it on your system, and finally you will become familiar with the two data structures called Series and DataFrame. In the course of the chapter you will work with a basic set of functions, provided by the pandas library, to perform the most common tasks in the data processing. Getting familiar with these operations will be a key issue for the rest of the book. This is the reason that it is very important for you to repeat this chapter until you will have familiarity with all its content.

Furthermore, with a series of examples you will learn some particularly new concepts introduced by the pandas library: the indexing of its data structures. How to get the most of this feature for data manipulation will be shown both in this chapter and in the next chapters.

Finally, you will see how it is possible to extend the concept of indexing to multiple levels at the same time: the hierarchical indexing.

pandas: The Python Data Analysis Library

Pandas is an open source Python library for highly specialized data analysis. Currently it is the reference point that all professionals using the Python language need to study and analyze data sets for statistical purposes of analysis and decision making.

This library has been designed and developed primarily by Wes McKinney starting in 2008; later, in 2012, Sien Chang, one of his colleagues, was added to the development. Together they set up one of the most used libraries in the Python community.

Pandas arises from the need to have a specific library for analysis of the data which provides, in the simplest possible way, all the instruments for the processing of data, data extraction, and data manipulation.

This Python package is designed on the basis of the NumPy library. This choice, we can say, was critical to the success and the rapid spread of pandas. In fact, this choice not only makes this library compatible with most of the other modules, but also takes advantage of the high quality of performance in calculating of the NumPy module.

Another fundamental choice has been to design ad hoc data structures for the data analysis. In fact, instead of using existing data structures built into Python or provided by other libraries, two new data structures have been developed.

These data structures are designed to work with relational data or labeled, thus allowing you to manage data with features similar to those designed for SQL relational databases and Excel spreadsheets.

Throughout the book, in fact, we will see a series of basic operations for data analysis, normally used on the database tables or spreadsheets. Pandas in fact provides an extended set of functions and methods that allow you to perform, and in many cases even in the best way, these operations.

So pandas has as its main purpose to provide all the building blocks for anyone approaching the world of data analysis.

Installation

The easiest and most general way to install the pandas library is to use a prepackaged solution, i.e., installing it through the distribution Anaconda or Enthought.

Installation from Anaconda

For those who have chosen to use the distribution Anaconda, the management of the installation is very simple. First we have to see if the pandas module is installed and which version. To do this, type the following command from terminal:

```
command.conda list pandas
```

Since I already have the module installed on my PC (Windows), I get the following result:

```
# packages in environment at C:\Users\Fabio\Anaconda:
#
pandas                    0.14.1              np19py27_0
```

If in your case it should not be so, you will need to install the module pandas. Enter the following command:

```
conda install pandas
```

Anaconda will immediately check all dependencies, managing the installation of other modules, without you having to worry too much.

If you want to upgrade your package to a newer version, the command is very simple and intuitive:

```
conda update pandas
```

The system will check the version of pandas and the version of all the modules on which it depends and suggest any updates, and then ask if you want to proceed or not to the update.

```
Fetching package metadata: ....
Solving package specifications: .
Package plan for installation in environment C:\Users\Fabio\Anaconda:
```

The following packages will be downloaded:

```
package                   |                    build
--------------------------|-------------------------------
pytz-2014.9               |               py27_0      169 KB
requests-2.5.3            |               py27_0      588 KB
six-1.9.0                 |               py27_0       16 KB
conda-3.9.1               |               py27_0      206 KB
pip-6.0.8                 |               py27_0      1.6 MB
scipy-0.15.1              |           np19py27_0     71.3 MB
pandas-0.15.2             |           np19py27_0      4.3 MB
------------------------------------------------------------
                                        Total:       78.1 MB
```

The following packages will be UPDATED:

```
conda:    3.9.0-py27_0       --> 3.9.1-py27_0
pandas:   0.14.1-np19py27_0 --> 0.15.2-np19py27_0
pip:      1.5.6-py27_0       --> 6.0.8-py27_0
pytz:     2014.7-py27_0      --> 2014.9-py27_0
requests: 2.5.1-py27_0       --> 2.5.3-py27_0
scipy:    0.14.0-np19py27_0 --> 0.15.1-np19py27_0
six:      1.8.0-py27_0       --> 1.9.0-py27_0

Proceed ([y]/n)?
```

After pressing 'y', Anaconda will begin to do the download of all modules updated from the network. So when you perform this step it will be necessary that the PC is connected to the network.

```
Fetching packages ...
scipy-0.15.1-n 100% |#############################| Time: 0:01:11   1.05 MB/s
Extracting packages ...
[      COMPLETE      ] |#############################################| 100%
Unlinking packages ...
[      COMPLETE      ] |#############################################| 100%
Linking packages ...
[      COMPLETE      ] |#############################################| 100%
```

Installation from PyPI

pandas can also be installed by PyPI:

```
pip install pandas
```

Installation on Linux

If you're working on a Linux distribution, and you choose not to use any of these prepackaged distributions, you can install the module pandas like any other package.

```
On Debian and Ubuntu distributions:

sudo apt-get install python-pandas
```

While on OpenSuse and Fedora enter the following command:

```
zypper in python-pandas
```

Installation from Source

If you want to compile your module pandas starting from the source code, you can find what you need on Github to the link http://github.com/pydata/pandas.

```
git clone git://github.com/pydata/pandas.git
cd pandas
python setup.py install
```

Make sure you have installed Cython at compile time. For more information please read the documentation available on the Web, including the official page (http://pandas.pydata.org/pandas-docs/stable/install.html).

A Module Repository for Windows

For those who are working on Windows and prefer to manage their packages themselves in order to always have the most current modules, there is also a resource on the Internet where you can download many third-party modules: **Christoph Gohlke's Python Extension Packages for Windows** repository (www.lfd.uci.edu/~gohlke/pythonlibs/). Each module is supplied with the format archival WHL (wheel) in both 32-bit and 64-bit. To install each module you have to use the application pip (see PyPI in Chapter 2).

```
pip install SomePackege-1.0.whl
```

In the choice of the module, be careful to choose the correct version for your version of Python and the architecture on which you're working. Furthermore while NumPy does not require the installation of other packages, on the contrary, pandas has many dependencies. So make sure you get them all. The installation order is not important.

The disadvantage of this approach is that you need to install the packages individually without any package manager that will help you in some way to manage the versioning and interdependencies between the various packages. The advantage is greater mastery of the modules and their versions, so you have the most current modules possible without depending on the choices of distributions as in Anaconda.

Test Your pandas Installation

The pandas library also provides the ability to run after its installation a test to check on the executability of its internal controls (the official documentation states that the test provides a 97% coverage of all the code inside).

First make sure you have installed the module nose in your Python distribution (see the "Nose Module" sidebar). If you do, then you can start the test by entering the following command:

```
nosetests pandas
```

The test will take several minutes to perform its task, and in the end it will show a list of the problems encountered.

NOSE MODULE

This module is designed for testing the Python code during the development phases of a project or a Python module in particular. This module extends the capabilities of the unittest module: the Python module involved in testing the code, however, making its coding much simpler and easier.

I suggest you read this article (http://pythontesting.net/framework/nose/nose-introduction/)

Getting Started with pandas

Given the nature of the topic covered in this chapter, centered on the explanation of the data structures and functions/methods applied to it, the writing of large listings or scripts is not required.

Thus the approach I felt better for this chapter is opening a Python shell and typing commands one by one. In this way, the player has the opportunity to become familiar with the individual functions and data structures, gradually explained in this chapter.

Furthermore, the data and functions defined in the various examples remain valid in the following ones, thus avoiding the reader needing to define each time around. The reader is invited, at the end of each example, to repeat the various commands, modifying them if appropriate, and to control how the values within the data structures vary during operation. This approach is great for getting familiar with the different topics covered in this chapter, leaving the reader the opportunity to interact freely with what you are reading and not to end up in easy automation as write and execute.

■ **Note** This chapter assumes that you have some familiarity with Python and NumPy in general. If you have any difficulty, you should read Chapters 2 and 3 of this book.

First, open a session on the Python shell and then import the pandas library. The general practice for importing the module pandas is as follows:

```
>>> import pandas as pd
>>> import numpy as np
```

Thus, in this chapter and throughout the book, every time you see pd and np, you'll make reference to an object or method referring to these two libraries, even though you will often be tempted to import the pandas module in this way:

```
>>> from pandas import *
```

Thus, you no longer have to reference function, object, or method with pd; this approach is not considered a good practice by the Python community in general.

Introduction to pandas Data Structures

The heart of pandas is just the two primary data structures on which all transactions, which are generally made during the analysis of data, are centralized:

- Series
- DataFrame

The Series, as you will see, constitutes the data structure designed to accommodate a sequence of one-dimensional data, while the DataFrame, a more complex data structure, is designed to contain cases with several dimensions.

Although these data structures are not the universal solution to all the problems, they do provide a valid and robust tool for most applications. In fact, in their simplicity, they remain very simple to understand and use. In addition, many cases of more complex data structures can still be traced to these simple two cases.

However, their peculiarities are based on a particular feature, integration in their structure of Index objects and Labels. You will see that this factor will lead to a high manipulability of these data structures.

The Series

The **Series** is the object of the pandas library designed to represent one-dimensional data structures, similarly to an array but with some additional features. Its internal structure is simple (see Figure 4-1) and is composed of two arrays associated with each other. The main array has the purpose to hold the data (data of any NumPy type) to which each element is associated with a label, contained within the other array, called the **Index**.

Figure 4-1. *The structure of the Series object*

Declaring a Series

To create the Series as specified in Figure 4-1, simply call the Series() constructor passing as an argument an array containing the values to be included in it.

```
>>> s = pd.Series([12,-4,7,9])
>>> s
0    12
1    -4
2     7
3     9
dtype: int64
```

As you can see from the output of the Series, on the left there are the values in the Index, which is a series of labels, and on the right the corresponding values.

If you do not specify any index during the definition of the Series, by default, pandas will assign numerical values increasing from 0 as labels. In this case the labels correspond to the indexes (position in the array) of the elements within the Series object.

Often, however, it is preferable to create a Series using meaningful labels in order to distinguish and identify each item regardless of the order in which they were inserted into the Series.

So in this case it will be necessary, during the constructor call, to include the **index** option assigning an array of strings containing the labels.

```
>>> s = pd.Series([12,-4,7,9], index=['a','b','c','d'])
>>> s
a    12
b    -4
c     7
d     9
dtype: int64
```

If you want to individually see the two arrays that make up this data structure you can call the two attributes of the Series as follows: **index** and **values**.

```
>>> s.values
array([12, -4,  7,  9], dtype=int64)
>>> s.index
Index([u'a', u'b', u'c', u'd'], dtype='object')
```

Selecting the Internal Elements

For that concerning individual elements, you can select them as an ordinary numpy array, specifying the key.

```
>>> s[2]
7
```

Or you can specify the label corresponding to the position of the index.

```
>>> s['b']
-4
```

In the same way you select multiple items in a numpy array, you can specify the following:

```
>>> s[0:2]
a    12
b    -4
dtype: int64
```

or even in this case, use the corresponding labels, but specifying the list of labels within an array.

```
>>> s[['b','c']]
b    -4
c     7
dtype: int64
```

Assigning Values to the Elements

Now that you understand how to select individual elements, you also know how to assign new values to them. In fact, you can select the value by index or label.

```
>>> s[1] = 0
>>> s
a    12
b     0
c     7
d     9
dtype: int64
>>> s['b'] = 1
>>> s
a    12
b     1
c     7
d     9
dtype: int64
```

Defining Series from NumPy Arrays and Other Series

You can define new Series starting with NumPy arrays or existing Series.

```
>>> arr = np.array([1,2,3,4])
>>> s3 = pd.Series(arr)
>>> s3
0    1
1    2
2    3
3    4
dtype: int32

>>> s4 = pd.Series(s)
>>> s4
a    12
b     4
c     7
d     9
dtype: int64
```

When doing this, however, you should always keep in mind that the values contained within the NumPy array or the original Series are not copied, but are passed by reference. That is, the object is inserted dynamically within the new Series object. If it changes, for example its internal element varies in value, then those changes will also be present in the new Series object.

```
>>> s3
0    1
1    2
2    3
3    4
dtype: int32
```

```
>>> arr[2] = -2
>>> s3
0    1
1    2
2   -2
3    4
dtype: int32
```

As we can see in this example, by changing the third element of the **arr** array we also modified the corresponding element in the **s3** Series.

Filtering Values

Thanks to the choice of NumPy library as the base for the development of the pandas library and as a result, for its data structures, many operations applicable to NumPy arrays are extended to the Series. One of these is the filtering of the values contained within the data structure through conditions.

For example, if you need to know which elements within the series have value greater than 8, you will write the following:

```
>>> s[s > 8]
a    12
d     9
dtype: int64
```

Operations and Mathematical Functions

Other operations such as operators (+, -, *, /) or mathematical functions that are applicable to NumPy array can be extended to objects Series.

Regarding the operators you can simply write the arithmetic expression.

```
>>> s / 2
a    6.0
b   -2.0
c    3.5
d    4.5
dtype: float64
```

However, regarding the NumPy mathematical functions, you must specify the function referenced with np and the instance of the Series passed as argument.

```
>>> np.log(s)
a    2.484907
b         NaN
c    1.945910
d    2.197225
dtype: float64
```

Evaluating Values

Often within a Series there may be duplicate values and then you may need to have information on what are the samples contained, counting duplicates and whether a value is present or not in the Series.

In this regard, declare a series in which there are many duplicate values.

```
>>> serd = pd.Series([1,0,2,1,2,3], index=['white','white','blue','green','green','yellow'])
>>> serd
white    1
white    0
blue     2
green    1
green    2
yellow   3
dtype: int64
```

To know all the values contained within the Series excluding duplicates, you can use the **unique()** function. The return value is an array containing the unique values in the Series, though not necessarily in order.

```
>>> serd.unique()
array([1, 0, 2, 3], dtype=int64)
```

A function similar to unique() is the **value_counts()** function, which not only returns the unique values but calculates occurrences within a Series.

```
>>> serd.value_counts()
2    2
1    2
3    1
0    1
dtype: int64
```

Finally, **isin()** is a function that evaluates the membership, that is, given a list of values, this function lets you know if these values are contained within the data structure. Boolean values that are returned can be very useful during the filtering of data within a series or in a column of a DataFrame.

```
>>> serd.isin([0,3])
white    False
white     True
blue     False
green    False
green    False
yellow    True
dtype: bool
>>> serd[serd.isin([0,3])]
white    0
yellow   3
dtype: int64
```

NaN Values

As you can see in the previous case we tried to run the logarithm of a negative number and received NaN as a result. This specific value **NaN (Not a Number)** is used within pandas data structures to indicate the presence of an empty field or not definable numerically.

Generally, these NaN values are a problem and must be managed in some way, especially during data analysis. These data are generated especially when extracting data from some source gave some trouble, or even when the source is a missing data. Furthermore, as you have just seen, the NaN values can also be generated in special cases, such as calculations of logarithms of negative values, or exceptions during execution of some calculation or function. In later chapters we will see how to apply different strategies to address the problem of NaN values.

Despite their problematic nature, however, pandas allows to explicitly define and add this value in a data structure, such as Series. Within the array containing the values you enter **np.NaN** wherever we want to define a missing value.

```
>>> s2 = pd.Series([5,-3,np.NaN,14])
>>> s2
0     5
1    -3
2   NaN
3    14
```

The **isnull()** and **notnull()** functions are very useful to identify the indexes without a value.

```
>>> s2.isnull()
0    False
1    False
2     True
3    False
dtype: bool
>>> s2.notnull()
0     True
1     True
2    False
3     True
dtype: bool
```

In fact, these functions return two Series with Boolean values that contains the 'True' and 'False' values depending on whether the item is a NaN value or less. The isnull() function returns 'True' at NaN values in the Series; inversely, the notnull() function returns 'True' if they are not NaN. These functions are useful to be placed inside a filtering to make a condition.

```
>>> s2[s2.notnull()]
0     5
1    -3
3    14
dtype: float64
>>> s2[s2.isnull()]
2   NaN
dtype: float64
```

Series as Dictionaries

An alternative way to see a Series is to think of them as an object dict (dictionary). This similarity is also exploited during the definition of an object Series. In fact, you can create a series from a dict previously defined.

```
>>> mydict = {'red': 2000, 'blue': 1000, 'yellow': 500, 'orange': 1000}
>>> myseries = pd.Series(mydict)
blue      1000
orange    1000
red       2000
yellow     500
dtype: int64
```

As you can see from this example, the array of index is filled with the values of the keys while the data with the corresponding values. You can also define the array indexes separately. In this case, control of correspondence between the keys of the dict and labels array of indexes will run. In case of mismatch, pandas will add value NaN.

```
>>> colors = ['red','yellow','orange','blue','green']
>>> myseries = pd.Series(mydict, index=colors)
red       2000
yellow     500
orange    1000
blue      1000
green      NaN
dtype: float64
```

Operations between Series

We have seen how to perform arithmetic operations between Series and scalar values. The same thing is possible by performing operations between two Series, but in this case even the labels come into play.

In fact one of the great potentials of this type of data structures is the ability of Series to align data addressed differently between them by identifying their corresponding label.

In the following example you sum two Series having only some elements in common with label.

```
>>> mydict2 = {'red':400,'yellow':1000,'black':700}
>>> myseries2 = pd.Series(mydict2)
>>> myseries + myseries2
black      NaN
blue       NaN
orange     NaN
green      NaN
red       2400
yellow    1500
dtype: float64
```

You get a new object Series in which only the items with the same label are added. All other label present in one of the two series are still added to the result but have a NaN value.

The DataFrame

The **DataFrame** is a tabular data structure very similar to the Spreadsheet (the most familiar are Excel spreadsheets). This data structure is designed to extend the case of the Series to multiple dimensions. In fact, the DataFrame consists of an ordered collection of columns (see Figure 4-2), each of which can contain a value of different type (numeric, string, Boolean, etc.).

DataFrame			
	columns		
index	**color**	**object**	**price**
0	blue	ball	1.2
1	green	pen	1.0
2	yellow	pencil	0.6
3	red	paper	0.9
4	white	mug	1.7

Figure 4-2. *The DataFrame structure*

Unlike Series, which had an **Index** array containing labels associated with each element, in the case of the data frame, there are two index arrays. The first, associated with the lines, has very similar functions to the index array in Series. In fact, each label is associated with all the values in the row. The second array instead contains a series of labels, each associated with a particular column.

A DataFrame may also be understood as a dict of Series, where the keys are the column names and values are the Series that will form the columns of the data frame. Furthermore, all elements of each Series are mapped according to an array of labels, called Index.

Defining a DataFrame

The most common way to create a new DataFrame is precisely to pass a dict object to the DataFrame() constructor. This dict object contains a key for each column that we want to define, with an array of values for each of them.

```
>>> data = {'color' : ['blue','green','yellow','red','white'],
                   'object' : ['ball','pen','pencil','paper','mug'],
                   'price' : [1.2,1.0,0.6,0.9,1.7]}
```

```
frame = pd.DataFrame(data)
>>> frame
    color  object price
0    blue    ball   1.2
1   green     pen   1.0
2  yellow  pencil   0.6
3     red   paper   0.9
4   white     mug   1.7
```

If the object dict from which we want to create a DataFrame contains more data than we are interested, you can make a selection. In the constructor of the data frame, you can specify a sequence of columns, using the **columns** option. The columns will be created in the order of the sequence regardless of how they are contained within the object dict.

```
>>> frame2 = pd.DataFrame(data, columns=['object','price'])
>>> frame2
   object price
0    ball   1.2
1     pen   1.0
2  pencil   0.6
3   paper   0.9
4     mug   1.7
```

Even for DataFrame objects, if the labels are not explicitly specified within the Index array, pandas automatically assigns a numeric sequence starting from 0. Instead, if you want to assign labels to the indexes of a DataFrame, you have to use the **index** option assigning it an array containing the labels.

```
>>> frame2 = pd.DataFrame(data, index=['one','two','three','four','five'])
>>> frame2
        color  object  price
one      blue    ball    1.2
two     green     pen    1.0
three  yellow  pencil    0.6
four      red   paper    0.9
five    white     mug    1.7
```

Now that we have introduced the two new options **index** and **columns**, it is easy to imagine an alternative way to define a DataFrame. Instead of using a dict object, you can define within the constructor three arguments, in the following order: a data matrix, then an array containing the labels assigned to the **index** option, and finally an array containing the names of the columns assigned to the **columns** option.

In many examples, as you will see from now on in this book, to create quickly and easily a matrix of values you can use **np.arange(16).reshape((4,4))** that generates a 4x4 matrix of increasing numbers from 0 to 15.

```
>>> frame3 = pd.DataFrame(np.arange(16).reshape((4,4)),
...                 index=['red','blue','yellow','white'],
...                 columns=['ball','pen','pencil','paper'])
>>> frame3
        ball  pen  pencil  paper
red        0    1       2      3
blue       4    5       6      7
yellow     8    9      10     11
white     12   13      14     15
```

Selecting Elements

First, if we want to know the name of all the columns of a DataFrame is sufficient to specify the **columns** attribute on the instance of the DataFrame object.

```
>>> frame.columns
Index([u'colors', u'object', u'price'], dtype='object')
```

Similarly, to get the list of indexes, you should specify the **index** attribute.

```
>>> frame.index
Int64Index([0, 1, 2, 3, 4], dtype='int64')
```

As regards the values contained within the data structure, you can get the entire set of data using the **values** attribute.

```
>>> frame.values
array([['blue', 'ball', 1.2],
       ['green', 'pen', 1.0],
       ['yellow', 'pencil', 3.3],
       ['red', 'paper', 0.9],
       ['white', 'mug', 1.7]], dtype=object)
```

Or, if you are interested to select only the contents of a column, you can write the name of the column.

```
>>> frame['price']
0    1.2
1    1.0
2    0.6
3    0.9
4    1.7
Name: price, dtype: float64
```

As you can see, the return value is a Series object. Another way is to use the column name as an attribute of the instance of the DataFrame.

```
>>> frame.price
0    1.2
1    1.0
2    0.6
3    0.9
4    1.7
Name: price, dtype: float64
```

Regarding the rows within a data frame, it is possible to use the **ix** attribute with the index value of the row that you want to extract.

```
>>> frame.ix[2]
color     yellow
object    pencil
price        0.6
Name: 2, dtype: object
```

The object returned is again a Series, in which the names of the columns have become the label of the array index, whereas the values have become the data of Series.

To select multiple rows you specify an array with the sequence of rows to insert:

```
>>> frame.ix[[2,4]]
    color  object  price
2  yellow  pencil    0.6
4   white     mug    1.7
```

If you need to extract a portion of a DataFrame, selecting the lines that you want to extract, you can use the reference numbers of the indexes. In fact you can consider a row as a portion of a data frame that has the index of the row as the source (in the next 0) value and the line above the one we want as a second value (in the next one).

```
>>> frame[0:1]
  color object  price
0  blue   ball    1.2
```

As you can see, the return value is an object data frame containing a single row. If you want more than one line, you must extend the selection range.

```
>>> frame[1:3]
    color  object  price
1   green     pen    1.0
2  yellow  pencil    0.6
```

Finally, if what you want to achieve is a single value within a DataFrame, first you have use the name of the column and then the index or the label of the row.

```
>>> frame['object'][3]
'paper'
```

Assigning Values

Once you understand how to access the various elements that make up a DataFrame, just follow the same logic to add or change the values in it.

For example, you have already seen that within the DataFrame structure an array of indexes is specified by the **index** attribute, and the row containing the name of the columns is specified with the **columns** attribute. Well, you can also assign a label, using the **name** attribute, to these two substructures for identifying them.

```
>>> frame.index.name = 'id'; frame.columns.name = 'item'
>>> frame
item    color  object  price
id
0        blue    ball    1.2
1       green     pen    1.0
2      yellow  pencil    3.3
3         red   paper    0.9
4       white     mug    1.7
```

One of the best features of the data structures of pandas is their high flexibility. In fact you can always intervene at any level to change the internal data structure. For example, a very common operation is to add a new column.

You can do this by simply assigning a value to the instance of the DataFrame specifying a new column name.

```
>>> frame['new'] = 12
>>> frame
   colors  object price  new
0    blue    ball   1.2   12
1   green     pen   1.0   12
2  yellow  pencil   0.6   12
3     red   paper   0.9   12
4   white     mug   1.7   12
```

As you can see from the result, there is a new column called 'new' with the value within 12 replicated for each of its elements.

If, however, you want to do an update of the contents of a column, you have to use an array.

```
>>> frame['new'] = [3.0,1.3,2.2,0.8,1.1]
>>> frame
    color  object price  new
0    blue    ball   1.2  3.0
1   green     pen   1.0  1.3
2  yellow  pencil   0.6  2.2
3     red   paper   0.9  0.8
4   white     mug   1.7  1.1
```

You can follow a similar approach if you want to update an entire column, for example, by using the function **np.arange()** to update the values of a column with a predetermined sequence.

The columns of a data frame can also be created by assigning a Series to one of them, for example by specifying a series containing an increasing series of values through the use of **np.arange()**.

```
>>> ser = pd.Series(np.arange(5))
>>> ser
0    0
1    1
2    2
3    3
4    4
dtype: int32
>>> frame['new'] = ser
>>> frame
    color  object price  new
0    blue    ball   1.2    0
1   green     pen   1.0    1
2  yellow  pencil   0.6    2
3     red   paper   0.9    3
4   white     mug   1.7    4
```

Finally, to change a single value, simply select the item and give it the new value.

```
>>> frame['price'][2] = 3.3
```

Membership of a Value

You have already seen the function **isin()** applied to the Series to decide the membership of a set of values. Well, this feature is also applicable on DataFrame objects.

```
>>> frame.isin([1.0,'pen'])
   color object  price
0  False  False  False
1  False   True   True
2  False  False  False
3  False  False  False
4  False  False  False
```

You get a DataFrame containing only Boolean values, where True has only the values that meet the membership. If you pass the value returned as a condition then you'll get a new DataFrame containing only the values that satisfy the condition.

```
>>> frame[frame.isin([1.0,'pen'])]
  color object  price
0   NaN   NaN    NaN
1   NaN   pen      1
2   NaN   NaN    NaN
3   NaN   NaN    NaN
4   NaN   NaN    NaN
```

Deleting a Column

If you want to delete an entire column with all its contents, then use the **del** command.

```
>>> del frame['new']
>>> frame
   colors  object price
0    blue    ball   1.2
1   green     pen   1.0
2  yellow  pencil   0.6
3     red   paper   0.9
4   white     mug   1.7
```

Filtering

Even for a DataFrame you can apply the filtering through the application of certain conditions, for example if you want to get all values smaller than a certain number, for example 12.

```
>>> frame[frame < 12]
        ball  pen  pencil  paper
red        0    1       2      3
blue       4    5       6      7
yellow     8    9      10     11
white    NaN  NaN     NaN    NaN
```

You will get as returned object a DataFrame containing values less than 12, keeping their original position. All others will be replaced with NaN.

DataFrame from Nested dict

A very common data structure used in Python is a nested dict, as the one represented as follows:

```
nestdict = { 'red': { 2012: 22, 2013: 33 },
                  'white': { 2011: 13, 2012: 22; 2013: 16},
                  'blue': {2011: 17, 2012: 27; 2013: 18}}
```

This data structure, when it is passed directly as an argument to the DataFrame() constructor, will be interpreted by pandas so as to consider external keys as column names and internal keys as labels for the indexes.

During the interpretation of the nested structure, it is possible that not all fields find a successful match. pandas will compensate for this inconsistency by adding the value NaN values missing.

```
>>> nestdict = {'red':{2012: 22, 2013: 33},
...             'white':{2011: 13, 2012: 22, 2013: 16},
...             'blue': {2011: 17, 2012: 27, 2013: 18}}
>>> frame2 = pd.DataFrame(nestdict)
>>> frame2
      blue   red   white
2011    17   NaN      13
2012    27    22      22
2013    18    33      16
```

Transposition of a DataFrame

An operation that might be needed when dealing with tabular data structures is the transposition (that is, the columns become rows and rows columns). pandas allows you to do this in a very simple way. You can get the transpose of the data frame by adding the **T** attribute to its application.

```
>>> frame2.T
       2011  2012  2013
blue     17    27    18
red     NaN    22    33
white    13    22    16
```

The Index Objects

Now that you know what the Series and the data frame are and how they are structured, you can certainly perceive the peculiarities of these data structures. Indeed, the majority of their excellent characteristics in the data analysis are due to the presence of an Index object totally integrated within these data structures.

The Index objects are responsible for the labels on the axes and other metadata as the name of the axes. You have already seen as an array containing labels is converted into an Index object: you need to specify the **index** option within the constructor.

```
>>> ser = pd.Series([5,0,3,8,4], index=['red','blue','yellow','white','green'])
>>> ser.index
Index([u'red', u'blue', u'yellow', u'white', u'green'], dtype='object')
```

Unlike all other elements within pandas data structures (Series and data frame), the Index objects are immutable objects. Once declared, these cannot be changed. This ensures their secure sharing between the various data structures.

Each Index object has a number of methods and properties especially useful when you need to know the values they contain.

Methods on Index

There are some specific methods for indexes available to get some information about index from a data structure. For example, **idmin()** and **idmax()** are two functions that return, respectively, the index with the lowest value and more.

```
>>> ser.idxmin()
'red'
>>> ser.idxmax()
'green'
```

Index with Duplicate Labels

So far, you have met all cases in which the indexes within a single data structure had the unique label. Although many functions require this condition to run, for the data structures of pandas this condition is not mandatory.

Define by way of example, a Series with some duplicate labels.

```
>>> serd = pd.Series(range(6), index=['white','white','blue','green','green','yellow'])
>>> serd
white     0
white     1
blue      2
green     3
green     4
yellow    5
dtype: int64
```

Regarding the selection of elements within a data structure, if in correspondence of the same label there are more values, you will get a Series in place of a single element.

```
>>> serd['white']
white     0
white     1
dtype: int64
```

The same logic applies to the data frame with duplicate indexes that will return the data frame.

In the case of data structures with small size, it is easy to identify any duplicate indexes, but if the structure becomes gradually larger this starts to become difficult. Just in this respect, pandas provides you with the **is_unique** attribute belonging to the Index objects. This attribute will tell you if there are indexes with duplicate labels inside the structure data (both Series and DataFrame).

```
>>> serd.index.is_unique
False
>>> frame.index.is_unique
True
```

Other Functionalities on Indexes

Compared to data structures commonly used with Python, you saw that pandas, as well as taking advantage of the high-performance quality offered by NumPy arrays, has chosen to integrate indexes within them.

This choice has proven somewhat successful. In fact, despite the enormous flexibility given by the dynamic structures that already exist, the capability to use the internal reference to the structure, such as that offered by the labels, allows those who must perform operations to carry out in a much more simple and direct way a series of operations that you will see in this and the next chapter.

In this section you will analyze in detail a number of basic features that take advantage of this mechanism of the indexes.

- Reindexing

- Dropping

- Alignment

Reindexing

It was previously stated that once declared within a data structure, the Index object cannot be changed. This is true, but by executing a reindexing you can also overcome this problem.

In fact it is possible to obtain a new data structure from an existing one where indexing rules can be defined again.

```
>>> ser = pd.Series([2,5,7,4], index=['one','two','three','four'])
>>> ser
one      2
two      5
three    7
four     4
dtype: int64
```

In order to make the reindexing of this series, pandas provides you with the **reindex()** function. This function creates a new Series object with the values of the previous Series rearranged according to the new sequence of labels.

During this operation of reindexing, it is therefore possible to change the order of the sequence of indexes, delete some of them, or add new ones. In the case of a new label, pandas add NaN as corresponding value.

```
>>> ser.reindex(['three','four','five','one'])
three      7
four       4
five     NaN
one        2
dtype: float64
```

As you can see from the value returned, the order of the labels has been completely rearranged. The value corresponding to the label 'two' has been dropped and a new label 'five' is present in the Series.

However, to measure the reindexing, the definition of the list of all the labels can be awkward, especially for a large data frame. So you could use some method that allows you to fill or interpolate values automatically.

To better understand the functioning of this mode of automatic reindexing, define the following Series.

```
>>> ser3 = pd.Series([1,5,6,3],index=[0,3,5,6])
>>> ser3
0    1
3    5
5    6
6    3
dtype: int64
```

As you can see in this example, the index column is not a perfect sequence of numbers; in fact there are some missing values (1, 2, and 4). A common need would be to perform an interpolation in order to obtain the complete sequence of numbers. To achieve this you will use the reindexing with the **method** option set to **ffill**. Moreover, you need to set a range of values for indexes. In this case, for specifying a set of values between 0 and 5, you can use **range(6)** as argument.

```
>>> ser3.reindex(range(6),method='ffill')
0    1
1    1
2    1
3    5
4    5
5    6
dtype: int64
```

As you can see from the result, the indexes that were not present in the original Series were added. By interpolation, those with the lowest index in the original Series, have been assigned as values. In fact the indexes 1 and 2 have the value 1 which belongs to index 0.

If you want this index value to be assigned during the interpolation, you have to use the **bfill** method.

```
>>> ser3.reindex(range(6),method='bfill')
0    1
1    5
2    5
3    5
4    6
5    6
dtype: int64
```

In this case the value assigned to the indexes 1 and 2 is the value 5, which belongs to index 3.

Extending the concepts of reindexing with Series to the DataFrame, you can have a rearrangement not only for indexes (rows), but also with regard to the columns, or even both. As previously mentioned, the addition of a new column or index is possible, but being missing values in the original data structure, pandas add NaN values to them.

```
>>> frame.reindex(range(5), method='ffill',columns=['colors','price','new','object'])
   colors price  new  object
0    blue   1.2  NaN  ballpand
1   green   1.0  NaN       pen
2  yellow   0.6  NaN    pencil
3     red   0.9  NaN     paper
4   white   1.7  NaN       mug
```

Dropping

Another operation that is connected to Index objects is dropping. Deleting a row or a column becomes simple, precisely due to the labels used to indicate the indexes and column names.

Also in this case, pandas provides a specific function for this operation: **drop()**. This method will return a new object without the items that you want to delete.

For example, take the case where we want to remove a single item from a Series. To do this, define generic Series 4 elements with four distinct labels.

```
>>> ser = Series(np.arange(4.), index=['red','blue','yellow','white'])
>>> ser
red       0
blue      1
yellow    2
white     3
dtype: float64
```

Now, for example, you want to delete the item corresponding to the label 'yellow'. Simply specify the label as an argument of the function **drop()** to delete it.

```
>>> ser.drop('yellow')
red      0
blue     1
white    3
dtype: float64
```

To remove more items, just pass an array with the corresponding labels.

```
>>> ser.drop(['blue','white'])
red       0
yellow    2
dtype: float64
```

Regarding the DataFrame, instead, the values can be deleted by referring to the labels of both axes. Declare the following frame by way of example.

```
>>> frame = pd.DataFrame(np.arange(16).reshape((4,4)),
...                      index=['red','blue','yellow','white'],
...                      columns=['ball','pen','pencil','paper'])
>>> frame
        ball  pen  pencil  paper
red        0    1       2      3
blue       4    5       6      7
yellow     8    9      10     11
white     12   13      14     15
```

To delete rows, just pass the indexes of the rows.

```
>>> frame.drop(['blue','yellow'])
        ball  pen  pencil  paper
red        0    1       2      3
white     12   13      14     15
```

To delete columns, you always need to specify the indexes of the columns, but you must specify the axis from which to delete the elements, and this can be done using the **axis** option. So to refer to the column names you should specify axis = 1.

```
>>> frame.drop(['pen','pencil'],axis=1)
        ball  paper
red        0      3
blue       4      7
yellow     8     11
white     12     15
```

Arithmetic and Data Alignment

Perhaps the most powerful feature involving the indexes in a data structure, is that pandas is able to perform the alignment of the indexes coming from two different data structures. This is especially true when you are performing an arithmetic operation between them. In fact, during these operations, not only may the indexes between the two structures be in a different order, but they also may be present in only one of the two structures.

As you can see from the examples that follow, pandas proves to be very powerful in the alignment of the indexes during these operations. For example, you can start considering two Series in which they are defined, respectively, two arrays of labels, not perfectly matching each other.

```
>>> s1 = pd.Series([3,2,5,1],['white','yellow','green','blue'])
>>> s2 = pd.Series([1,4,7,2,1],['white','yellow','black','blue','brown'])
```

Now among the various arithmetic operations, consider the simple sum. As you can see from the two Series just declared, some labels are present in both, while other labels are present only in one of the two. Well, when the labels are present in both operators, their values will be added, while in the opposite case, they will also be shown in the result (new series), but with the value NaN.

```
>>> s1 + s2
black    NaN
blue       3
brown    NaN
green    NaN
white      4
yellow     6
dtype: float64
```

In the case of the data frame, although it may appear more complex, the alignment follows the same principle, but is carried out both for the rows and for the columns.

```
>>> frame1 = pd.DataFrame(np.arange(16).reshape((4,4)),
...                     index=['red','blue','yellow','white'],
...                     columns=['ball','pen','pencil','paper'])
>>> frame2 = pd.DataFrame(np.arange(12).reshape((4,3)),
...                     index=['blue','green','white','yellow'],
...                     columns=['mug','pen','ball'])
>>> frame1
        ball  pen  pencil  paper
red        0    1       2      3
blue       4    5       6      7
yellow     8    9      10     11
white     12   13      14     15
>>> frame2
        mug  pen  ball
blue      0    1     2
green     3    4     5
white     6    7     8
yellow    9   10    11
>>> frame1 + frame2
        ball  mug  paper  pen  pencil
blue       6  NaN    NaN    6     NaN
green    NaN  NaN    NaN  NaN     NaN
red      NaN  NaN    NaN  NaN     NaN
white     20  NaN    NaN   20     NaN
yellow    19  NaN    NaN   19     NaN
```

Operations between Data Structures

Now that you have become familiar with the data structures such as Series and DataFrame and you have seen how various elementary operations can be performed on them, it's time to go to operations involving two or more of these structures.

For example, in the previous section we saw how the arithmetic operators apply between two of these objects. Now in this section you will deepen more the topic of operations that can be performed between two data structures.

CHAPTER 4 ■ THE PANDAS LIBRARY—AN INTRODUCTION

Flexible Arithmetic Methods

You've just seen how to use mathematical operators directly on the pandas data structures. The same operations can also be performed using appropriate methods, called **Flexible arithmetic methods**.

- add()

- sub()

- div()

- mul()

In order to call these functions, you'll need to use a specification different than what you're used to dealing with mathematical operators. For example, instead of writing a sum between two DataFrame 'frame1 + frame2', you'll have to use the following format:

```
>>> frame1.add(frame2)
        ball   mug  paper  pen  pencil
blue       6   NaN    NaN    6     NaN
green    NaN   NaN    NaN  NaN     NaN
red      NaN   NaN    NaN  NaN     NaN
white     20   NaN    NaN   20     NaN
yellow    19   NaN    NaN   19     NaN
```

As you can see the results are the same as what you'd get using the addition operator '+'. You can also note that if the indexes and column names differ greatly from one series to another, you'll find yourself with a new data frame full of NaN values. You'll see later in this chapter how to handle this kind of data.

Operations between DataFrame and Series

Coming back to the arithmetic operators, pandas allows you to make transactions even between different structures as for example, a DataFrame and a Series. For example, we define these two structures in the following way.

```
>>> frame = pd.DataFrame(np.arange(16).reshape((4,4)),
...                    index=['red','blue','yellow','white'],
...                    columns=['ball','pen','pencil','paper'])
>>> frame
        ball  pen  pencil  paper
red        0    1       2      3
blue       4    5       6      7
yellow     8    9      10     11
white     12   13      14     15
>>> ser = pd.Series(np.arange(4), index=['ball','pen','pencil','paper'])
>>> ser
ball      0
pen       1
pencil    2
paper     3
dtype: int32
```

The two newly defined data structures have been created specifically so that the indexes of Series match with the names of the columns of the DataFrame. This way, you can apply a direct operation.

```
>>> frame - ser
        ball  pen  pencil  paper
red        0    0       0      0
blue       4    4       4      4
yellow     8    8       8      8
white     12   12      12     12
```

As you can see, the elements of the series are subtracted from the values of the data frame corresponding to the same index on the column. The value is subtracted for all values of the column, regardless of their index.

If an index is not present in one of the two data structures, the result will be a new column with that index only that all its elements will be NaN.

```
>>> ser['mug'] = 9
>>> ser
ball      0
pen       1
pencil    2
paper     3
mug       9
dtype: int64
>>> frame - ser
        ball  mug  paper  pen  pencil
red        0  NaN      0    0       0
blue       4  NaN      4    4       4
yellow     8  NaN      8    8       8
white     12  NaN     12   12      12
```

Function Application and Mapping

This section covers pandas library functions

Functions by Element

The pandas library is built on the foundations of NumPy, and then extends many of its features adapting them to new data structures as Series and DataFrame. Among these are the **universal functions**, called **ufunc**. This class of functions is particular because it operates by element in the data structure.

```
>>> frame = pd.DataFrame(np.arange(16).reshape((4,4)),
...                 index=['red','blue','yellow','white'],
...                 columns=['ball','pen','pencil','paper'])
>>> frame
        ball  pen  pencil  paper
red        0    1       2      3
blue       4    5       6      7
yellow     8    9      10     11
white     12   13      14     15
```

For example you could calculate the square root of each value within the data frame, using the NumPy **np.sqrt()**.

```
>>> np.sqrt(frame)
            ball       pen    pencil     paper
red     0.000000  1.000000  1.414214  1.732051
blue    2.000000  2.236068  2.449490  2.645751
yellow  2.828427  3.000000  3.162278  3.316625
white   3.464102  3.605551  3.741657  3.872983
```

Functions by Row or Column

The application of the functions is not limited to the ufunc functions, but also includes those defined by the user. The important thing is that they operate on a one-dimensional array, giving a single number for result. For example, we can define a lambda function that calculates the range covered by the elements in an array.

```
>>> f = lambda x: x.max() - x.min()
```

It is possible to define the function also in this way:

```
>>> def f(x):
...     return x.max() - x.min()
...
```

Using the **apply()** function you can apply the function just defined on the DataFrame.

```
>>> frame.apply(f)
ball      12
pen       12
pencil    12
paper     12
dtype: int64
```

The result, however, this time it is only one value for the column, but if you prefer to apply the function by row instead of by column, you have to specify the **axis** option set to 1.

```
>>> frame.apply(f, axis=1)
red       3
blue      3
yellow    3
white     3
dtype: int64
```

It is not mandatory that the method **apply()** returns a scalar value. It can also return a Series. A useful case would be to extend the application to many functions simultaneously. In this case we will have two or more values for each feature applied. This can be done by defining a function in the following manner:

```
>>> def f(x):
...     return pd.Series([x.min(), x.max()], index=['min','max'])
...
```

Then, apply the function as before. But in this case as an object returned you get a DataFrame and no longer a Series, in which there will be as many rows as the values returned by the function.

```
>>> frame.apply(f)
     ball  pen  pencil  paper
min     0    1       2      3
max    12   13      14     15
```

Statistics Functions

However, the majority of the statistical functions for arrays are still valid for DataFrame, so the use of the **apply()** function is no longer necessary. For example, functions such as **sum()** and **mean()** can calculate the sum and the average, respectively, of the elements contained within a DataFrame.

```
>>> frame.sum()
ball      24
pen       28
pencil    32
paper     36
dtype: int64
>>> frame.mean()
ball      6
pen       7
pencil    8
paper     9
dtype: float64
```

There is also a function called **describe()** that allows to obtain a summary statistics at once.

```
>>> frame.describe()
            ball        pen     pencil      paper
count   4.000000   4.000000   4.000000   4.000000
mean    6.000000   7.000000   8.000000   9.000000
std     5.163978   5.163978   5.163978   5.163978
min     0.000000   1.000000   2.000000   3.000000
25%     3.000000   4.000000   5.000000   6.000000
50%     6.000000   7.000000   8.000000   9.000000
75%     9.000000  10.000000  11.000000  12.000000
max    12.000000  13.000000  14.000000  15.000000
```

Sorting and Ranking

Another fundamental operation that makes use of the indexing is sorting. Sorting the data is often a necessity and it is very important to be able to do easily. Pandas provides the **sort_index()** function that returns a new object which is identical to the start, but in which the elements are ordered.

You start by seeing how you can sort items in a Series. The operation is quite trivial since the list of indexes to be ordered is only one.

```
>>> ser = pd.Series([5,0,3,8,4], index=['red','blue','yellow','white','green'])
>>> ser
red       5
blue      0
yellow    3
white     8
green     4
dtype: int64fra
>>> ser.sort_index()
blue      0
green     4
red       5
white     8
yellow    3
dtype: int64
```

As you can see the items were sorted in the alphabetical order of the labels in ascending order (from A to Z). This is the default behavior, but you can set the opposite order, using the **ascending** option set to False.

```
>>> ser.sort_index(ascending=False)
yellow    3
white     8
red       5
green     4
blue      0
dtype: int64
```

As regards the DataFrame, the sorting can be performed independently on each of its two axes. So if you want to order by row following the indexes, just continue to use the function **sort_index()** without arguments as you've seen before, or if you prefer to order by columns, you will need to use the **axis** options set to 1.

```
>>> frame = pd.DataFrame(np.arange(16).reshape((4,4)),
...                      index=['red','blue','yellow','white'],
...                      columns=['ball','pen','pencil','paper'])
>>> frame
        ball  pen  pencil  paper
red        0    1       2      3
blue       4    5       6      7
yellow     8    9      10     11
white     12   13      14     15
>>> frame.sort_index()
        ball  pen  pencil  paper
blue       4    5       6      7
red        0    1       2      3
white     12   13      14     15
yellow     8    9      10     11
```

```
>>> frame.sort_index(axis=1)
        ball  paper  pen  pencil
red        0      3    1       2
blue       4      7    5       6
yellow     8     11    9      10
white     12     15   13      14
```

So far you have learned how to sort the values according to the indexes. But very often you may need to sort the values contained within the data structure. In this case you have to differentiate depending on whether you have to sort the values of a Series or a DataFrame.

If you want to order the series, you will use the **order()** function.

```
>>> ser.order()
blue      0
yellow    3
green     4
red       5
white     8
dtype: int64
```

If you need to order the values in a DataFrame, you will use the **sort_index()** function seen previously but with the **by** option. Then you have to specify the name of the column on which to sort.

```
>>> frame.sort_index(by='pen')
        ball  pen  pencil  paper
red        0    1       2      3
blue       4    5       6      7
yellow     8    9      10     11
white     12   13      14     15
```

If the criteria of sorting will be based on two or more columns, you can assign an array containing the names of the columns to the **by** option.

```
>>> frame.sort_index(by=['pen','pencil'])
        ball  pen  pencil  paper
red        0    1       2      3
blue       4    5       6      7
yellow     8    9      10     11
white     12   13      14     15
```

The **ranking** is an operation closely related to sorting. It mainly consists of assigning a rank (that is, a value that starts at 0 and then increase gradually) to each element of the series. The rank will be assigned starting from the lowest value to the highest value.

```
>>> ser.rank()
red       4
blue      1
yellow    2
white     5
green     3
dtype: float64
```

The rank can also be assigned in the order in which the data are already in the data structure (without a sorting operation). In this case, just add the **method** option with the 'first' value assigned.

```
>>> ser.rank(method='first')
red       4
blue      1
yellow    2
white     5
green     3
dtype: float64
```

By default, even the ranking follows an ascending sort. To reverse this criterion, set the **ascending** option to False.

```
>>> ser.rank(ascending=False)
red       2
blue      5
yellow    4
white     1
green     3
dtype: float64
```

Correlation and Covariance

Two important statistical calculations are correlation and covariance, expressed in pandas by the **corr()** and **cov()** functions. These kind of calculations normally involve two Series.

```
>>> seq2 = pd.Seri
es([3,4,3,4,5,4,3,2],['2006','2007','2008','2009','2010','2011','2012','2013'])
>>> seq = pd.Seri
es([1,2,3,4,4,3,2,1],['2006','2007','2008','2009','2010','2011','2012','2013'])
>>> seq.corr(seq2)
0.77459666924148329
>>> seq.cov(seq2)
0.8571428571428571
```

Another case could be that covariance and correlation are applied to a single DataFrame. In this case, they return their corresponding matrices in form of two new DataFrame objects.

```
>>> frame2 = DataFrame([[1,4,3,6],[4,5,6,1],[3,3,1,5],[4,1,6,4]],
...                    index=['red','blue','yellow','white'],
...                    columns=['ball','pen','pencil','paper'])
>>> frame2
        ball  pen  pencil  paper
red        1    4       3      6
blue       4    5       6      1
yellow     3    3       1      5
white      4    1       6      4
```

```
>>> frame2.corr()
           ball       pen    pencil     paper
ball   1.000000 -0.276026  0.577350 -0.763763
pen   -0.276026  1.000000 -0.079682 -0.361403
pencil 0.577350 -0.079682  1.000000 -0.692935
paper -0.763763 -0.361403 -0.692935  1.000000
>>> frame2.cov()
           ball       pen    pencil     paper
ball   2.000000 -0.666667  2.000000 -2.333333
pen   -0.666667  2.916667 -0.333333 -1.333333
pencil 2.000000 -0.333333  6.000000 -3.666667
paper -2.333333 -1.333333 -3.666667  4.666667
```

Using the method **corrwith()**, you can calculate the pairwise correlations between the columns or rows of a data frame with a Series or another DataFrame().

```
>>> serred      0
blue       1
yellow     2
white      3
green      9
dtype: float64
>>> frame2.corrwith(ser)
ball     0.730297
pen     -0.831522
pencil   0.210819
paper   -0.119523
dtype: float64
>>> frame2.corrwith(frame)
ball     0.730297
pen     -0.831522
pencil   0.210819
paper   -0.119523
dtype: float64
```

"Not a Number" Data

During the previous sections we have seen how easily the missing data can be formed. They are recognizable within the data structures with the value NaN (Not a Number). So, having values that are not defined in a data structure is a condition quite common for those who carry out data analysis.

However, pandas is designed to better manage this eventuality. In fact, in this section you will learn how to treat these values so that many issues can be obviated. In fact, for example, within the pandas library, the calculation of descriptive statistics excludes NaN values implicitly.

Assigning a NaN Value

Just in case you would like to specifically assign a NaN value to an element in a data structure, you can use the value **np.NaN**(or np.nan) of the NumPy library.

```
>>> ser = pd.Series([0,1,2,np.NaN,9], index=['red','blue','yellow','white','green'])
>>> ser
red        0
blue       1
yellow     2
white    NaN
green      9
dtype: float64
>>> ser['white'] = None
>>> ser
red        0
blue       1
yellow     2
white    NaN
green      9
dtype: float64
```

Filtering Out NaN Values

There are various options to eliminate the NaN values during the data analysis. However, the elimination by hand, element by element, can be very tedious and risky, because you never get the certainty of having eliminated all the NaN values. The **dropna()** function comes to your aid.

```
>>> ser.dropna()
red        0
blue       1
yellow     2
green      9
dtype: float64
```

Another possibility is to directly perform the filtering function by placing the **notnull()** in the selection condition.

```
>>> ser[ser.notnull()]
red        0
blue       1
yellow     2
green      9
dtype: float64
```

If you're dealing with the DataFrame it gets a little more complex. If you use the **dropna()** function on this type of object, it is sufficient that there is only one value NaN on a column or a row to eliminate it completely.

```
>>> frame3 = pd.DataFrame([[6,np.nan,6],[np.nan,np.nan,np.nan],[2,np.nan,5]],
...                       index = ['blue','green','red'],
...                       columns = ['ball','mug','pen'])
```

```
>>> frame3
       ball  mug  pen
blue      6  NaN    6
green   NaN  NaN  NaN
red       2  NaN    5
>>> frame3.dropna()
Empty DataFrame
Columns: [ball, mug, pen]
Index: []
```

Therefore to avoid having entire rows and columns disappear completely, you should specify the **how** option, assigning a value of 'all' to it, in order to inform the **dropna()** function to delete only the rows or columns in which all elements are NaN.

```
>>> frame3.dropna(how='all')
      ball  mug  pen
blue     6  NaN    6
red      2  NaN    5
```

Filling in NaN Occurrences

Rather than filter NaN values within data structures, with the risk of discarding them along with values that could be relevant in the context of data analysis, you could replace them with other numbers. For most purposes, the fillna() function could be a great choice. This method takes one argument, the value with which to replace any NaN. It can be the same for all, as in the following case.

```
>>> frame3.fillna(0)
      ball  mug  pen
blue     6    0    6
green    0    0    0
red      2    0    5
```

Or you can replace NaN with different values depending on the column, specifying one by one the indexes and the associated value.

```
>>> frame3.fillna({'ball':1,'mug':0,'pen':99})
      ball  mug  pen
blue     6    0    6
green    1    0   99
red      2    0    5
```

Hierarchical Indexing and Leveling

The **hierarchical indexing** is a very important feature of pandas, as it allows you to have multiple levels of indexes on a single axis. Somehow it gives you a way to work with data in multiple dimensions continuing to work in a two-dimensional structure.

You can start with a simple example, creating a series containing two arrays of indexes, that is, creating a structure with two levels.

```
>>> mser = pd.Series(np.random.rand(8),
...         index=[['white','white','white','blue','blue','red','red','red'],
...         ['up','down','right','up','down','up','down','left']])
>>> mser
white   up        0.461689
        down      0.643121
        right     0.956163
blue    up        0.728021
        down      0.813079
red     up        0.536433
        down      0.606161
        left      0.996686
dtype: float64

>>> mser.index
MultiIndex(levels=[[u'blue', u'red', u'white'], [u'down', u'left', u'right', u'up']],
           labels=[[2, 2, 2, 0, 0, 1, 1, 1], [3, 0, 2, 3, 0, 3, 0, 1]])ù
```

Through the specification of a hierarchical indexing, the selection of subsets of values is in a certain way simplified.

In fact, you can select the values for a given value of the first index, and you do it in the classic way:

```
>>> mser['white']
up      0.461689
down    0.643121
right   0.956163
dtype: float64
```

or you can select values for a given value of the second index, in the following manner:

```
>>> mser[:,'up']
white   0.461689
blue    0.728021
red     0.536433
dtype: float64
```

Intuitively, if we want to select a specific value, you will specify both indexes.

```
>>> mser['white','up']
0.46168915430531676
```

The hierarchical indexing plays a critical role in reshaping the data and group-based operations such as creating a pivot-table. For example, the data could be used just rearranged in a data frame using a special function called **unstack()**. This function converts the Series with hierarchical index in a simple DataFrame, where the second set of indexes is converted into a new set of columns.

```
>>> mser.unstack()
          down      left     right        up
blue   0.813079       NaN       NaN  0.728021
red    0.606161  0.996686       NaN  0.536433
white  0.643121       NaN  0.956163  0.461689
```

If what we want is to perform the reverse operation, which is to convert a DataFrame in a Series, you will use the **stack()** function.

```
>>> frame
        ball  pen  pencil  paper
red        0    1       2      3
blue       4    5       6      7
yellow     8    9      10     11
white     12   13      14     15
>>> frame.stack()
red     ball       0
        pen        1
        pencil     2
        paper      3
blue    ball       4
        pen        5
        pencil     6
        paper      7
yellow  ball       8
        pen        9
        pencil    10
        paper     11
white   ball      12
        pen       13
        pencil    14
        paper     15
dtype: int32
```

As regards the DataFrame, it is possible to define a hierarchical index both for the rows and for the columns. At the time of the declaration of the DataFrame, you have to define an array of arrays for both the **index** option and the **columns** option.

```
>>> mframe = pd.DataFrame(np.random.randn(16).reshape(4,4),
...      index=[['white','white','red','red'], ['up','down','up','down']],
...      columns=[['pen','pen','paper','paper'],[1,2,1,2]])
>>> mframe
                  pen                 paper
                    1         2         1         2
white up    -1.964055  1.312100 -0.914750 -0.941930
      down  -1.886825  1.700858 -1.060846 -0.197669
red   up    -1.561761  1.225509 -0.244772  0.345843
      down   2.668155  0.528971 -1.633708  0.921735
```

Reordering and Sorting Levels

Occasionally, you could need to rearrange the order of the levels on an axis or do a sorting for values at a specific level.

The **swaplevel()** function accepts as argument the names assigned to the two levels that you want to interchange, and returns a new object with the two levels interchanged between them, while leaving the data unmodified.

```
>>> mframe.columns.names = ['objects','id']
>>> mframe.index.names = ['colors','status']
>>> mframe
objects          pen              paper
id                1       2        1        2
colors status
white  up     -1.964055  1.312100 -0.914750 -0.941930
       down   -1.886825  1.700858 -1.060846 -0.197669
red    up     -1.561761  1.225509 -0.244772  0.345843
       down    2.668155  0.528971 -1.633708  0.921735

>>> mframe.swaplevel('colors','status')
objects          pen              paper
id                1       2        1        2
status colors
up     white  -1.964055  1.312100 -0.914750 -0.941930
down   white  -1.886825  1.700858 -1.060846 -0.197669
up     red    -1.561761  1.225509 -0.244772  0.345843
down   red     2.668155  0.528971 -1.633708  0.921735
```

Instead, the **sortlevel()** function orders the data considering only those of a certain level.

```
>>> mframe.sortlevel('colors')
objects          pen              paper
id                1       2        1        2
colors status
red    down    2.668155  0.528971 -1.633708  0.921735
       up     -1.561761  1.225509 -0.244772  0.345843
white  down   -1.886825  1.700858 -1.060846 -0.197669
       up     -1.964055  1.312100 -0.914750 -0.941930
```

Summary Statistic by Level

Many descriptive statistics and summary statistics performed on a DataFrame or on a Series have a **level** option, with which you can determine at what level the descriptive and summary statistics should be determined.

For example if you make a statistic at row level, you have to simply specify the **level** option with the level name.

```
>>> mframe.sum(level='colors')
objects         pen                  paper
id                1        2         1         2
colors
red       1.106394  1.754480 -1.878480  1.267578
white    -3.850881  3.012959 -1.975596 -1.139599
```

If you want to make a statistic for a given level of the column, for example, the **id**, you must specify the second axis as argument through the **axis** option set to 1.

```
>>> mframe.sum(level='id', axis=1)
id                      1         2
colors status
white  up       -2.878806  0.370170
       down     -2.947672  1.503189
red    up       -1.806532  1.571352
       down      1.034447  1.450706
```

Conclusions

In this chapter, the library pandas has been introduced. You have learned how to install it and then you have seen a general overview based on its characteristics.

In more detail, you saw the two basic structures data, called Series and DataFrame, along with their operation and their main characteristics. Especially, you discovered the importance of indexing within these structures and how best to perform some operations on them. Finally you looked at the possibility of extending the complexity of these structures creating hierarchies of indexes, thus distributing the data contained in them in different sub-levels.

In the next chapter, you will see how to capture data from external sources such as files, and inversely, how to write the results of our analysis on them.

■ ■ ■

pandas: Reading and Writing Data

In the previous chapter, you got familiar with the pandas library and with all the basic functionalities that it provides for the data analysis. You have seen that DataFrame and Series are the heart of this library. These are the material on which to perform all manipulations of data, calculations, and analysis.

In this chapter you will see all of the tools provided by pandas for reading data stored in many types of media (such as files and databases). In parallel, you will also see how to write data structures directly on these formats, without worrying too much about the technologies used.

This chapter is focused on a series of I/O API functions that pandas provides to facilitate as much as possible the reading and writing data process directly as DataFrame objects on all of the most commonly used formats. You start to see the text files, then move gradually to more complex binary formats.

At the end of the chapter, you'll also learn how to interface with all common databases, both SQL and NoSQL, with examples showing how to store the data in a DataFrame directly in them. At the same time, you will see how to read the data contained in a database and retrieve them already as a DataFrame.

I/O API Tools

pandas is a library specialized for data analysis, so you expect that it is mainly focused on calculation and data processing. Moreover, even the process of writing and reading data from/to external files can be considered a part of the data processing. In fact, you will see how, even at this stage, you can perform some operations in order to prepare the incoming data to further manipulations.

Thus, this part is very important for data analysis and therefore a specific tool for this purpose must be present in the library pandas: a set of functions called I/O API. These functions are divided into two main categories, completely symmetrical to each other: **readers** and **writers**.

Readers	Writers
read_csv	to_csv
read_excel	to_excel
read_hdf	to_hdf
read_sql	to_sql
read_json	to_json
read_html	to_html
read_stata	to_stata
read_clipboard	to_clipboard

(*continued*)

Readers	Writers
read_pickle	to_pickle
read_msgpack	to_msgpack (experimental)
read_gbq	to_gbq (experimental)

CSV and Textual Files

Everyone has become accustomed over the years to write and read files in text form. In particular, data are generally reported in tabular form. If the values in a row are separated by a comma, you have the CSV (comma-separated values) format, which is perhaps the best-known and most popular format.

Other forms with tabular data separated by spaces or tabs are typically contained in text files of various types (generally with the extension .txt).

So this type of file is the most common source of data and actually even easier to transcribe and interpret. In this regard pandas provides a set of functions specific for this type of file.

- read_csv

- read_table

- to_csv

Reading Data in CSV or Text Files

From common experience, the most common operation for a person approaching data analysis is to read the data contained in a CSV file, or at least in a text file.

In order to see how pandas handle this kind of data, start by creating a small CSV file in the working directory as shown in Listing 5-1 and save it as **myCSV_01.csv**.

Listing 5-1. myCSV_01.csv

```
white,red,blue,green,animal
1,5,2,3,cat
2,7,8,5,dog
3,3,6,7,horse
2,2,8,3,duck
4,4,2,1,mouse
```

Since this file is comma-delimited, you can use the **read_csv()** function to read its content and convert it at the same time in a DataFrame object.

```
>>> csvframe = read_csv('myCSV_01.csv')
>>> csvframe
   white  red  blue  green animal
0      1    5     2      3    cat
1      2    7     8      5    dog
2      3    3     6      7  horse
3      2    2     8      3   duck
4      4    4     2      1  mouse
```

As you can see the reading of the data in a CSV file is a rather trivial. CSV files are tabulated data in which the values on the same column are separated by commas. But since CSV files are considered text files, you can also use the **read_table()** function, but specifying the delimiter.

```
>>> read_table('ch05_01.csv',sep=',')
   white   red   blue   green animal
0      1     5      2       3    cat
1      2     7      8       5    dog
2      3     3      6       7  horse
3      2     2      8       3   duck
4      4     4      2       1  mouse
```

In the example you just saw, you can notice that in the CSV file, headers to identify all the columns are in the first row. But this is not a general case, it often happens that the tabulated data begin directly from the first line (see Listing 5-2).

Listing 5-2. myCSV_02.csv

```
1,5,2,3,cat
2,7,8,5,dog
3,3,6,7,horse
2,2,8,3,duck
4,4,2,1,mouse

>>> read_csv('ch05_02.csv')
   1  5  2  3    cat
0  2  7  8  5    dog
1  3  3  6  7  horse
2  2  2  8  3   duck
3  4  4  2  1  mouse
```

In this case, then you could make sure that it is precisely pandas to assign default names to the columns by using the **header** option set to None.

```
>>> read_csv('ch05_02.csv', header=None)
   0  1  2  3      4
0  1  5  2  3    cat
1  2  7  8  5    dog
2  3  3  6  7  horse
3  2  2  8  3   duck
4  4  4  2  1  mouse
```

In addition, there is also the possibility to specify the names directly assigning a list of labels to the **names** option.

```
>>> read_csv('ch05_02.csv', names=['white','red','blue','green','animal'])
   white   red   blue   green animal
0      1     5      2       3    cat
1      2     7      8       5    dog
2      3     3      6       7  horse
3      2     2      8       3   duck
4      4     4      2       1  mouse
```

In more complex cases, in which you want to create a DataFrame with a hierarchical structure by reading a CSV file, you can extend the functionality of the **read_csv()** function by adding the **index_col** option, assigning all the columns to be converted into indexes to it.

To better understand this possibility, create a new CSV file with two columns to be used as indexes of the hierarchy. Then, save it in the working directory as **myCSV_03.csv** (Listing 5-3).

Listing 5-3. myCSV_03.csv

```
color,status,item1,item2,item3
black,up,3,4,6
black,down,2,6,7
white,up,5,5,5
white,down,3,3,2
white,left,1,2,1
red,up,2,2,2
red,down,1,1,4

>>> read_csv('ch05_03.csv', index_col=['color','status'])
             item1   item2   item3
color status
black up         3       4       6
      down       2       6       7
white up         5       5       5
      down       3       3       2
      left       1       2       1
red   up         2       2       2
      down       1       1       4
```

Using RegExp for Parsing TXT Files

In other cases, it is possible that the files on which to parse the data do not show separators well defined as a comma or a semicolon. In these cases, the regular expressions come to our aid. In fact, you can specify a regexp within the **read_table()** function using the **sep** option.

To better understand the use of a regexp and how you can apply it as a criterion for separation of values, you can start from a simple case. For example, suppose that your file, such as a TXT file, has values separated by spaces or tabs in an unpredictable order. In this case, you have to use the regexp because only with it you will take into account as a separator both cases. You can do that using the wildcard /s*. /s stands for space or tab character (if you wanted to indicate only the tab, you would have used /t), while the pound indicates that these characters may be multiple (see Table 5-1 for other wildcards most commonly used). That is, the values may be separated by more spaces or more tabs.

Table 5-1. *Metacharacters*

.	single character, except newline
\d	digit
\D	non-digit character
\s	whitespace character
\S	non-whitespace character
\n	new line character
\t	tab character
\uxxxx	unicode character specified by the hexadecimal number xxxx

Take for example a case a little extreme, in which we have the values separated from each other by tab or space in a totally random order (Listing 5-4).

Listing 5-4. ch05_04.txt

```
white   red   blue   green
    1     5     2       3
    2     7     8       5
    3     3     6       7

>>> read_table('ch05_04.txt',sep='\s*')
   white   red   blue   green
0      1     5     2       3
1      2     7     8       5
2      3     3     6       7
```

As we can see the result we got is a perfect data frame in which the values are perfectly ordered.

Now you will see an example that may seem strange, or unusual, but actually it is not so rare as it may seem. This example can be very helpful to understand the high potential of a regexp. In fact, usually you think of the separators as special characters like commas, spaces, tabs, etc. but in reality you could consider separator characters as alphanumeric characters, or for example, as integers such as 0.

In this example, you need to extract the numeric part from a TXT file, in which there is a sequence of characters with numerical values and literal characters are completely fused.

Remember to use the **header** option set to **None** whenever the column headings are not present in the TXT file (Listing 5-5).

Listing 5-5. ch05_05.txt

```
000END123AAA122
001END124BBB321
002END125CCC333

>>> read_table('ch05_05.txt',sep='\D*',header=None)
   0    1    2
0  0  123  122
1  1  124  321
2  2  125  333
```

Another fairly common event is to exclude lines from parsing. In fact you do not always want to include headers or unnecessary comments contained within a file (see Listing 5-6). With the **skiprows** option you can exclude all the lines you want, just assigning an array containing the line numbers to not consider in parsing.

Pay attention when you are using this option. If you want to exclude the first five lines, then you have to write skiprows = 5, but if we want to rule out the fifth line you have to write skiprows = [5].

Listing 5-6. ch05_06.txt

```
########### LOG FILE ############
This file has been generated by automatic system
white,red,blue,green,animal
12-Feb-2015: Counting of animals inside the house
1,5,2,3,cat
2,7,8,5,dog
13-Feb-2015: Counting of animals outside the house
3,3,6,7,horse
2,2,8,3,duck
4,4,2,1,mouse

>>> read_table('ch05_06.txt',sep=',',skiprows=[0,1,3,6])
   white   red   blue   green animal
0      1     5      2       3    cat
1      2     7      8       5    dog
2      3     3      6       7  horse
3      2     2      8       3   duck
4      4     4      2       1  mouse
```

Reading TXT Files into Parts or Partially

When large files are processed, or when you're only interested in portions of these files, you often need to read the file into portions (chunks). This is both to apply any iterations and because we are not interested in doing the parsing of the entire file.

So if for example you want to read only a portion of the file, you can explicitly specify the number of lines on which to parse. Thanks to the **nrows** and **skiprows** options, you can select the starting line n (n = SkipRows) and the lines to be read after it (nrows = i).

```
>>> read_csv('ch05_02.csv',skiprows=[2],nrows=3,header=None)
   0  1  2  3     4
0  1  5  2  3   cat
1  2  7  8  5   dog
2  2  2  8  3  duck
```

Another interesting and fairly common operation is to split into portions that part of the text on which we want to parse. Then for each portion a specific operation may be carried out, in order to obtain an iteration, portion by portion.

For example, you want to add the values of a column every three rows of the file and then insert these sums within a series. This example is trivial and impractical but is very simple to understand, so that once you have learned the underlying mechanism you will be able to apply it in much more complex cases.

```
>>> out = Series()
>>> i = 0
>>> pieces = read_csv('ch05_01.csv',chunksize=3)
>>> for piece in pieces:
...     out.set_value(i,piece['white'].sum())
...     i = i + 1
...
0    6
dtype: int64
0    6
1    6
dtype: int64
>>> out
0    6
1    6
dtype: int64
```

Writing Data in CSV

In addition to reading the data contained within a file, the writing of a data file produced by a calculation, or in general the data contained in a data structure, is a common and necessary operation.

For example, you might want to write to a CSV file the data contained in a DataFrame. To do this writing process you will use the **to_csv()** function that accepts as an argument the name of the file you generate (Listing 5-7).

```
>>> frame2
ball  pen  pencil  paper
0      1      2      3
4      5      6      7
8      9     10     11
12    13     14     15
>>> frame2.to_csv('ch05_07.csv')
```

Listing 5-7. ch05_07.csv

```
ball,pen,pencil,paper
0,1,2,3
4,5,6,7
8,9,10,11
12,13,14,15
```

As you can see from the previous example, when you make the writing of a data frame to a file, by default both indexes and columns are marked on the file. This default behavior can be changed by placing the two options **index** and **header** set to False (Listing 5-8).

```
>>> frame2.to_csv('ch05_07b.csv', index=False, header=False)
```

Listing 5-8. ch05_08.csv

```
1,2,3
5,6,7
9,10,11
13,14,15
```

One thing to take into account when making the writing of files is that NaN values present in a data structure are shown as empty fields in the file (Listing 5-9).

```
>>> frame3
        ball  mug  paper  pen  pencil
blue       6  NaN    NaN    6     NaN
green    NaN  NaN    NaN  NaN     NaN
red      NaN  NaN    NaN  NaN     NaN
white     20  NaN    NaN   20     NaN
yellow    19  NaN    NaN   19     NaN
>>> frame3.to_csv('ch05_08.csv')
```

Listing 5-9. ch05_09.csv

```
,ball,mug,paper,pen,pencil
blue,6.0,,,6.0,
green,,,,,
red,,,,,
white,20.0,,,20.0,
yellow,19.0,,,19.0,
```

But you can replace this empty field with a value to your liking using the **na_rep** option in the **to_csv()** function. Common values may be NULL, 0, or the same NaN (Listing 5-10).

```
>>> frame3.to_csv('ch05_09.csv', na_rep ='NaN')
```

Listing 5-10. ch05_10.csv

```
,ball,mug,paper,pen,pencil
blue,6.0,NaN,NaN,6.0,NaN
green,NaN,NaN,NaN,NaN,NaN
red,NaN,NaN,NaN,NaN,NaN
white,20.0,NaN,NaN,20.0,NaN
yellow,19.0,NaN,NaN,19.0,NaN
```

■ **Note**   In the cases specified, data frame has always been the subject of discussion since usually these are the data structures that are written to the file. But all these functions and options are also valid with regard to the series.

Reading and Writing HTML Files

Also with regard to the HTML format pandas provides the corresponding pair of I/O API functions.

- read_html()

- to_html()

To have these two functions can be very useful. You will appreciate the ability to convert complex data structures such as DataFrame directly in HTML tables without having to hack a long listing in HTML, especially if you're dealing with the world web.

The inverse operation can be very useful, because now the major source of data is just the Web world. In fact a lot of data on the Internet does not always have the form "ready to use," that is packaged in some TXT or CSV file. Very often, however, the data are reported as part of the text of web pages. So also having available a function for reading could prove to be really useful.

This activity is so widespread that it is currently identified as Web Scraping. This process is becoming a fundamental part of the set of processes that will be integrated in the first part of the data analysis: data mining and data preparation.

■ **Note**   Many websites have now adopted the HTML5 format, to avoid any issues of missing modules and error messages. I recommend strongly to install the module html5lib. Anaconda specified:

```
conda install html5lib
```

Writing Data in HTML

Now you see how to convert a DataFrame into an HTML table. The internal structure of the data frame is automatically converted into nested tags <TH>, <TR>, <TD> retaining any internal hierarchies. Actually you do not need to know HTML to use this kind of function.

Because sometimes the data structures as the DataFrame can be quite complex and large, to have a function like this is a great resource for anyone who needs to develop web pages.

To better understand this potential, here's an example. You can start by defining a simple DataFrame.

Thanks to the **to_html()** function you have the ability to directly convert the DataFrame in an HTML table.

```
>>> frame = pd.DataFrame(np.arange(4).reshape(2,2))
```

Since the I/O API functions are defined within the pandas data structures, you can call the **to_html()** function directly on the instance of the DataFrame.

```
>>> print(frame.to_html())
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>0</th>
      <th>1</th>
    </tr>
  </thead>
```

```
  <tbody>
    <tr>
      <th>0</th>
      <td> 0</td>
      <td> 1</td>
    </tr>
    <tr>
      <th>1</th>
      <td> 2</td>
      <td> 3</td>
    </tr>
  </tbody>
</table>
```

As you can see, the whole structure formed by the HTML tags needed to create an HTML table was generated correctly in order to respect the internal structure of the data frame.

In the next example you'll see how the table appears automatically generated within an HTML file. In this regard, we create a data frame a bit more complex than the previous one, where there are the labels of the indexes and column names.

```
>>> frame = pd.DataFrame( np.random.random((4,4)),
...                   index = ['white','black','red','blue'],
...                   columns = ['up','down','right','left'])
>>> frame
              up        down      right      left
white   0.292434  0.457176  0.905139  0.737622
black   0.794233  0.949371  0.540191  0.367835
red     0.204529  0.981573  0.118329  0.761552
blue    0.628790  0.585922  0.039153  0.461598
```

Now you focus on writing an HTML page through the generation of a string. This is a simple and trivial example, but it is very useful to understand and to test the functionality of pandas directly on the web browser.

First of all we create a string that contains the code of the HTML page.

```
>>> s = ['<HTML>']
>>> s.append('<HEAD><TITLE>My DataFrame</TITLE></HEAD>')
>>> s.append('<BODY>')
>>> s.append(frame.to_html())
>>> s.append('</BODY></HTML>')
>>> html = ''.join(s)
```

Now that all the listing of the HTML page is contained within the variable html, you can write directly on the file that will be called **myFrame.html:**

```
>>> html_file = open('myFrame.html','w')
>>> html_file.write(html)
>>> html_file.close()
```

Now in your working directory will be a new HTML file, **myFrame.html**. Double-click it to open it directly from the browser. An HTML table will appear in the upper left as shown in Figure 5-1.

	up	down	right	left
white	0.292434	0.457176	0.905139	0.737622
black	0.794233	0.949371	0.540191	0.367835
red	0.204529	0.981573	0.118329	0.761552
blue	0.628790	0.585922	0.039153	0.461598

Figure 5-1. *The DataFrame is shown as an HTML table in the web page*

Reading Data from an HTML File

As you just saw, pandas can easily generate HTML tables starting from data frame. The opposite process is also possible; the function read_html () will perform a parsing an HTML page looking for an HTML table. If found, it will convert that table into an object DataFrame ready to be used in our data analysis.

More precisely, the **read_html()** function returns a list of DataFrame even if there is only one table. As regards the source to be subjected to parsing, this can be of different types. For example, you may have to read an HTML file in any directory. For example you can parse the HTML file you created in the previous example.

```
>>> web_frames = pd.read_html('myFrame.html')
>>> web_frames[0]
  Unnamed: 0        up      down     right      left
0      white  0.292434  0.457176  0.905139  0.737622
1      black  0.794233  0.949371  0.540191  0.367835
2        red  0.204529  0.981573  0.118329  0.761552
3       blue  0.628790  0.585922  0.039153  0.461598
```

As you can see, all of the tags that have nothing to do with HTML table are not considered absolutely. Furthermore **web_frames** is a list of DataFrames, although in your case, the DataFrame that you are extracting is only one. However, you can select the item in the list that we want to use, calling it in the classic way. In this case the item is unique and therefore the index will be 0.

However, the mode most commonly used regarding the **read_html()** function is that of a direct parsing of an URL on the Web. In this way the web pages in the network are directly parsed with the extraction of the tables within them.

For example, now you will call a web page where there is an HTML table that shows a ranking list with some names and scores.

```
>>> ranking = pd.read_html('http://www.meccanismocomplesso.org/en/
meccanismo-complesso-sito-2/classifica-punteggio/')
>>> ranking[0]
   Member        points  levels  Unnamed: 3
0       1    BrunoOrsini    1075         NaN
1       2      Berserker     700         NaN
2       3   albertosallu     275         NaN
3       4           Mr.Y     180         NaN
4       5            Jon     170         NaN
```

113

```
5        6  michele sisi     120         NaN
6        7  STEFANO GUST      120         NaN
7        8  Davide Alois      105         NaN
8        9  Cecilia Lala      105         NaN
...
```

The same operation can be run on any web page that has one or more tables.

Reading Data from XML

In the list of I/O API functions, there is no specific tool regarding the XML (Extensible Markup Language) format. In fact, although it is not listed, this format is very important, because many structured data are available in XML format. This presents no problem, since Python has many other libraries (besides pandas) that manage the reading and writing of data in XML format.

One of these libraries is the **lxml** library, which stands out for its excellent performance during the parsing of very large files. In this section you will be shown how to use this module for parsing XML files and how to integrate it with pandas to finally get the DataFrame containing the requested data. For more information about this library, I highly recommend visiting the official website of lxml: http://lxml.de/index.html.

Take for example an XML file as shown in Listing 5-11. Write down and save it with the name **books.xml** directly in your working directory.

Listing 5-11. books.xml

```
<?xml version="1.0"?>
<Catalog>
    <Book id="ISBN9872122367564">
       272103_1_EnRoss, Mark</Author>
       <Title>XML Cookbook</Title>
       <Genre>Computer</Genre>
       <Price>23.56</Price>
       <PublishDate>2014-22-01</PublishDate>
    </Book>
    <Book id="ISBN9872122367564">
       272103_1_EnBracket, Barbara</Author>
       <Title>XML for Dummies</Title>
       <Genre>Computer</Genre>
       <Price>35.95</Price>
       <PublishDate>2014-12-16</PublishDate>
    </Book>
</Catalog>
```

In this example you will take the data structure described in the XML file to convert it directly into a DataFrame. To do so the first thing to do is use the sub-module **objectify** of the lxml library, importing it in the following way.

```
>>> from lxml import objectify
```

Now you can do the parser of the XML file with just the **parse()** function.

```
>>> xml = objectify.parse('books.xml')
>>> xml
<lxml.etree._ElementTree object at 0x0000000009734E08>
You got an object tree, which is an internal data structure of the module lxml.
Look in more detail at this type of object. To navigate in this tree structure, so as to select
element by element, you must first define the root. You can do this with the getroot() function.
>>> root = xml.getroot()
```

Now that the root of the structure has been defined, you can access the various nodes of the tree, each corresponding to the tag contained within the original XML file. The items will have the same name as the corresponding tags. So to select them simply write the various separate tags with points, reflecting in a certain way the hierarchy of nodes in the tree.

```
>>> root.Book.Author
'Ross, Mark'
>>> root.Book.PublishDate
'2014-22-01'
```

In this way you access nodes individually, but you can access various elements at the same time using **getchildren()**. With this function, you'll get all the child nodes of the reference element.

```
>>> root.getchildren()
[<Element Book at 0x9c66688>, <Element Book at 0x9c66e08>]
```

With the **tag** attribute you get the name of the tag corresponding to the child node.

```
>>> [child.tag for child in root.Book.getchildren()]
['Author', 'Title', 'Genre', 'Price', 'PublishDate']
```

while with the **text** attribute you get the value contained between the corresponding tags.

```
>>> [child.text for child in root.Book.getchildren()]
['Ross, Mark', 'XML Cookbook', 'Computer', '23.56', '2014-22-01']
```

However, regardless of the ability to move through the **lxml.etree** tree structure, what you need is to convert it into a data frame. Define the following function, which has the task of analyzing the entire contents of a eTree to fill a DataFrame line by line.

```
>>> def etree2df(root):
...     column_names = []
...     for i in range(0,len(root.getchildren()[0].getchildren())):
...         column_names.append(root.getchildren()[0].getchildren()[i].tag)
...     xml:frame = pd.DataFrame(columns=column_names)
...     for j in range(0, len(root.getchildren())):
...         obj = root.getchildren()[j].getchildren()
...         texts = []
...         for k in range(0, len(column_names)):
...             texts.append(obj[k].text)
...         row = dict(zip(column_names, texts))
```

```
...         row_s = pd.Series(row)
...         row_s.name = j
...         xml:frame = xml:frame.append(row_s)
...     return xml:frame
...
>>> etree2df(root)
            Author            Title     Genre  Price PublishDate
0        Ross, Mark     XML Cookbook  Computer  23.56  2014-22-01
1  Bracket, Barbara  XML for Dummies  Computer  35.95  2014-12-16
```

Reading and Writing Data on Microsoft Excel Files

In the previous section, you saw how the data can be easily read from CSV files. It is not uncommon, however, that there are data collected in tabular form in the Excel spreadsheet.

pandas provides specific functions also for this type of format. You have seen that the I/O API provides two functions to this purpose:

- to_excel()
- read_excel()

As regards reading Excel files, the **read_excel()** function is able to read both Excel 2003 (.xls) files and Excel 2007 (.xlsx) files. This is possible thanks to the integration of the internal module **xlrd**.

First, open an Excel file and enter the data as shown in Figure 5-2. Copy data in sheet1 and sheet2. Then save it as **data.xls**.

	A	B	C	D	E
1		white	red	green	black
2	a	12	23	17	18
3	b	22	16	19	18
4	c	14	23	22	21
5					
6					

	A	B	C	D	E
		yellow	purple	blue	orange
	A	11	16	44	22
	B	20	22	23	44
	C	30	31	37	32

Figure 5-2. *The two data sets in sheet1 and sheet2 of an Excel file*

To read the data contained within the XLS file and obtain the conversion into a data frame, you only have to use the **read_excel()** function.

```
>>> pd.read_excel('data.xls')
   white   red   green   black
a     12    23      17      18
b     22    16      19      18
c     14    23      22      21
```

As you can see, by default, the returned DataFrame is composed of the data tabulated in the first spreadsheets. If, however, you'd need to load the data in the second spreadsheet, and then specify the name of the sheet or the number of the sheet (index) just as the second argument.

```
>>> pd.read_excel('data.xls','Sheet2')
   yellow   purple   blue   orange
A      11       16     44       22
B      20       22     23       44
C      30       31     37       32
>>> pd.read_excel('data.xls',1)
   yellow   purple   blue   orange
A      11       16     44       22
B      20       22     23       44
C      30       31     37       32
```

The same applies for writing. So to convert a data frame in a spreadsheet on Excel you have to write as follows.

```
>>> frame = pd.DataFrame(np.random.random((4,4)),
...                      index = ['exp1','exp2','exp3','exp4'],
...                      columns = ['Jan2015','Fab2015','Mar2015','Apr2005'])
>>> frame
        Jan2015    Fab2015    Mar2015    Apr2005
exp1   0.030083   0.065339   0.960494   0.510847
exp2   0.531885   0.706945   0.964943   0.085642
exp3   0.981325   0.868894   0.947871   0.387600
exp4   0.832527   0.357885   0.538138   0.357990
>>> frame.to_excel('data2.xlsx')
```

In the working directory you will find a new Excel file containing the data as shown in Figure 5-3.

	A	B	C	D	E
1		Jan2015	Fab2015	Mar2015	Apr2005
2	**exp1**	0,030083	0,065339	0,960494	0,510847
3	**exp2**	0,531885	0,706945	0,964943	0,085642
4	**exp3**	0,981325	0,868894	0,947871	0,3876
5	**exp4**	0,832527	0,357885	0,538138	0,35799
6					

Figure 5-3. *The DataFrame in the Excel file*

JSON Data

JSON (JavaScript Object Notation) has become one of the most common standard formats, especially for the transmission of data through the Web. So it is normal to have to do with this data format if you want to use the available data on the Web.

The special feature of this format is its great flexibility, though its structure is far from being the one to which you are well accustomed, i.e., tabular.

In this section you will see how to use the **read_json()** and **to_json()** functions to stay within the I/O API functions discussed in this chapter. But in the second part you will see another example in which you will have to deal with structured data in JSON format much more related to real cases.

In my opinion, a useful online application for checking the JSON format is JSONViewer, available at http://jsonviewer.stack.hu/. This web application, once you entered or copied data in JSON format, allows you to see if the format you entered is invalid. Moreover it displays the tree structure so that you can better understand its structure (as shown in Figure 5-4).
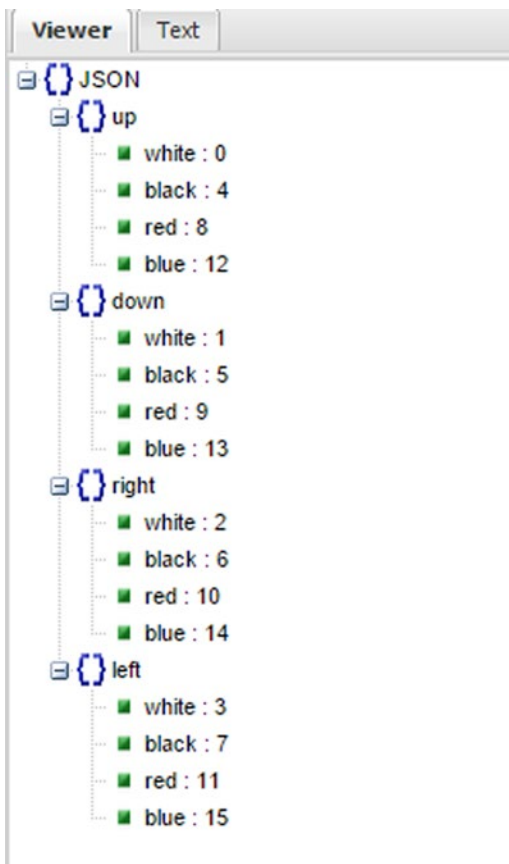
Figure 5-4. *JSONViewer*

Let's begin with the more useful case, that is, when you have a DataFrame and you need to convert it into a JSON file. So, define a DataFrame and then call the **to_json()** function on it, passing as argument the name of the file that you want to create.

```
>>> frame = pd.DataFrame(np.arange(16).reshape(4,4),
...                      index=['white','black','red','blue'],
...                      columns=['up','down','right','left'])
>>> frame.to_json('frame.json')
```

In the working directory you will find a new JSON file (see Listing 5-12) containing the DataFrame data translated into JSON format.

Listing 5-12. frame.json

```
{"up":{"white":0,"black":4,"red":8,"blue":12},"down":{"white":1,"black":5,"red":9,"blue":13},
"right":{"white":2,"black":6,"red":10,"blue":14},"left":{"white":3,"black":7,"red":11,"blue":15}}
```

The converse is possible, using the **read_json()** with the name of the file passed as an argument.

```
>>> pd.read_json('frame.json')
       down  left  right  up
black     5     7      6   4
blue     13    15     14  12
red       9    11     10   8
white     1     3      2   0
```

The example you have seen is a fairly simple case in which the JSON data were in tabular form (since the file **frame.json** comes from a DataFrame). Generally, however, the JSON files do not have a tabular structure. Thus, you will need to somehow convert the structure dict file in tabular form. You can refer this process as **normalization**.

The library pandas provides a function, called **json_normalize()**, that is able to convert a dict or a list in a table. First you have to import the function

```
>>> from pandas.io.json import json_normalize
```

Then write a JSON file as described in Listing 5-13 with any text editor. Save it in the working directory as **books.json**.

Listing 5-13. books.json

```
[{"writer": "Mark Ross",
 "nationality": "USA",
 "books": [
        {"title": "XML Cookbook", "price": 23.56},
        {"title": "Python Fundamentals", "price": 50.70},
        {"title": "The NumPy library", "price": 12.30}
            ]
},
```

```
{"writer": "Barbara Bracket",
 "nationality": "UK",
 "books": [
        {"title": "Java Enterprise", "price": 28.60},
        {"title": "HTML5", "price": 31.35},
        {"title": "Python for Dummies", "price": 28.00}
            ]
}]
```

As you can see, the file structure is no longer tabular, but more complex. Then the approach with the **read_json()** function is no longer valid. As you learn from this example, you can still get the data in tabular form from this structure. First you have to load the contents of the JSON file and convert it into a string.

```
>>> file = open('books.json','r')
>>> text = file.read()
>>> text = json.loads(text)
```

Now you are ready to apply the **json_normalize()** function. From a quick look at the contents of the data within the JSON file, for example, you might want to extract a table that contains all the books. Then write the **books** key as second argument.

```
>>> json_normalize(text,'books')
   price                 title
0  23.56          XML Cookbook
1  50.70   Python Fundamentals
2  12.30     The NumPy library
3  28.60       Java Enterprise
4  31.35                 HTML5
5  28.00    Python for Dummies
```

The function will read the contents of all the elements that have **books** as key. All properties will be converted into nested column names while the corresponding values will fill the DataFrame. As regards the indexes, the function assigns a sequence of increasing numbers.

However, you get a DataFrame containing only some internal information. It would be useful to add the values of other keys on the same level. In this case you can add other columns by inserting a key list as the third argument of the function.

```
>>> json_normalize(text2,'books',['writer','nationality'])
   price                 title nationality           writer
0  23.56          XML Cookbook         USA        Mark Ross
1  50.70   Python Fundamentals         USA        Mark Ross
2  12.30     The NumPy library         USA        Mark Ross
3  28.60       Java Enterprise          UK  Barbara Bracket
4  31.35                 HTML5          UK  Barbara Bracket
5  28.00    Python for Dummies          UK  Barbara Bracket
```

Now as a result you got a DataFrame from a starting tree structure.

The Format HDF5

So far you have seen how to write and read data in text format. When the data analysis involves large amounts of data it is preferable to use them in binary format. There are several tools in Python to handle binary data. A library that is having some success in this area is the **HDF5** library.

The HDF term stands for **Hierarchical Data Format**, and in fact this library is concerned with the reading and writing of HDF5 files containing a structure with nodes and the possibility to store multiple datasets.

This library, fully developed in C, however, has also interfaces with other types of languages like Python, Matlab, and Java. Thus, its extensive use is one of the reasons for its rapid spread. But not only for this, in fact; another reason is its efficiency, especially when using this format to save huge amounts of data. Compared to other formats that work more simply in binary, HDF5 supports compression in real time, thereby taking advantage of repetitive patterns within the data structure to compress the file size.

At present, the possible choices in Python are two: **PyTables** and **h5py**. These two forms differ in several aspects and therefore their choice depends very much on the needs of those who use it.

h5py provides a direct interface with the high-level APIs HDF5, while PyTables makes abstract many of the details of HDF5 to provide more flexible data containers, indexed tables, querying capabilities, and other media on the calculations.

pandas has a class like dict called **HDFStore**, using PyTables to store pandas objects. So before working with the format HDF5, you must import the class HDFStore.

```
>>> from pandas.io.pytables import HDFStore
```

Now you're ready to store the data of a data frame within a file .h5. First, create a DataFrame.

```
>>> frame = pd.DataFrame(np.arange(16).reshape(4,4),
...                      index=['white','black','red','blue'],
...                      columns=['up','down','right','left'])
```

Now create a file HDF5 calling it **mydata.h5**, then enter inside the data of the DataFrame.

```
>>> store = HDFStore('mydata.h5')
>>> store['obj1'] = frame
```
From here, you can guess how you can store multiple data structures within the same HDF5 file, specifying for each of them a label.
```
>>> frame2
       up  down  right  left
white   0   0.5      1   1.5
black   2   2.5      3   3.5
red     4   4.5      5   5.5
blue    6   6.5      7   7.5
>>> store['obj2'] = frame2
```

So with this type of format, you can store multiple data structures within a single file, represented by the **store** variable.

```
>>> store
<class 'pandas.io.pytables.HDFStore'>
File path: mydata.h5
/obj1              frame          (shape->[4,4])
```

Even the reverse process is very simple. Taking account of having an HDF5 file containing various data structures, objects inside can be called in the following way:

```
>>> store['obj2']
      up  down  right  left
white  0   0.5      1   1.5
black  2   2.5      3   3.5
red    4   4.5      5   5.5
blue   6   6.5      7   7.5
```

Pickle—Python Object Serialization

The **pickle** module implements a powerful algorithm for serialization and de-serialization of a data structure implemented in Python. Pickling is the process in which the hierarchy of an object is converted into a stream of bytes.

This allows an object to be transmitted and stored, and then to be rebuilt by the receiver itself retaining all the original features.

In Python, the picking operation is carried out by the **pickle** module, but currently there is a module called **cPickle** which is the result of an enormous amount of work optimizing the pickle module (written in C). This module can be in fact in many cases even 1,000 times faster than the pickle module. However, regardless of which module you do use, the interfaces of the two modules are almost the same.

Before moving to explicitly mention the I/O functions of pandas that operate on this format, let's look in more detail at cPickle module and how to use it.

Serialize a Python Object with cPickle

The data format used by the pickle module (or cPickle) is specific to Python. By default, an ASCII representation is used to represent it, in order to be readable from the human point of view. Then opening a file with a text editor you may be able to understand its contents. To use this module you must first import it

```
>>> import cPickle as pickle
```

Then create an object sufficiently complex to have an internal data structure, for example a dict object.

```
>>> data = { 'color': ['white','red'], 'value': [5, 7]}
```

Now you will perform a serialization of the **data** object through the **dumps()** function of the **cPickle** module.

```
>>> pickled_data = pickle.dumps(data)
```

Now, to see how it was serialized the dict object, you need to look at the content of the **pickled_data** variable.

```
>>> print pickled_data
(dp1
S'color'
p2
(lp3
S'white'
```

```
p4
aS'red'
p5
asS'value'
p6
(lp7
I5
aI7
as.
```

Once you have serialized data, they can easily be written on a file, or sent over a socket, pipe, etc.

After being transmitted, it is possible to reconstruct the serialized object (deserialization) with the **loads()** function of the **cPickle** module.

```
>>> nframe = pickle.loads(pickled_data)
>>> nframe
{'color': ['white', 'red'], 'value': [5, 7]}
```

Pickling with pandas

As regards the operation of pickling (and unpickling) with the pandas library, everything remains much facilitated. No need to import the cPickle module in the Python session and also the whole operation is performed implicitly.

Also, the serialization format used by pandas is not completely in ASCII.

```
>>> frame = pd.DataFrame(np.arange(16).reshape(4,4), index = ['up','down','left','right'])
>>> frame.to_pickle('frame.pkl')
```

Now in your working directory there is a new file called **frame.pkl** containing all the information about the **frame** DataFrame.

To open a PKL file and read the contents, simply use the command

```
>>> pd.read_pickle('frame.pkl')
        0   1   2   3
up      0   1   2   3
down    4   5   6   7
left    8   9  10  11
right  12  13  14  15
```

As you can see, all the implications on the operation of pickling and unpickling are completely hidden from the pandas user, allowing it to make the job as easy and understandable as possible, for those who must deal specifically with the data analysis.

■ **Note**   When you make use of this format make sure that the file you open is safe. Indeed, the pickle format was not designed to be protected against erroneous and maliciously constructed data.

Interacting with Databases

In many applications, the data rarely come from text files, given that this is certainly not the most efficient way to store data.

The data are often stored in an SQL-based relational database, and also in many alternative NoSQL databases that have become very popular in recent times.

Loading data from SQL in a DataFrame is sufficiently simple and pandas has some functions to simplify the process.

The **pandas.io.sql** module provides a unified interface independent of the DB, called **sqlalchemy**. This interface simplifies the connection mode, since regardless of the DB, the commands will always be the same. For making a connection you use the **create_engine()** function. With this feature you can configure all the properties necessary to use the driver, as a user, password, port, and database instance.

Here is a list of examples for the various types of databases:

```
>>> from sqlalchemy import create_engine
```

For PostgreSQL:

```
>>> engine = create_engine('postgresql://scott:tiger@localhost:5432/mydatabase')
For MySQL
>>> engine = create_engine('mysql+mysqldb://scott:tiger@localhost/foo')
For Oracle
>>> engine = create_engine('oracle://scott:tiger@127.0.0.1:1521/sidname')
For MSSQL
>>> engine = create_engine('mssql+pyodbc://mydsn')
For SQLite
>>> engine = create_engine('sqlite:///foo.db')
```

Loading and Writing Data with SQLite3

As a first example, you will use a SQLite database using the driver's built-in Python **sqlite3**. SQLite3 is a tool that implements a DBMS SQL in a very simple and lightweight way, so it can be incorporated within any application implemented with Python language. In fact, this practical software allows you to create an embedded database in a single file.

This makes it the perfect tool for anyone who wants to have the functions of a database without having to install a real database. SQLite3 could be the right choice for anyone who wants to practice before going on to a real database, or for anyone who needs to use the functions of a database for the collection of data, but remaining within a single program, without worry to interface with a database.

Create a data frame that you will use to create a new table on the SQLite3 database.

```
>>> frame = pd.DataFrame( np.arange(20).reshape(4,5),
...                       columns=['white','red','blue','black','green'])
>>> frame
   white  red  blue  black  green
0      0    1     2      3      4
1      5    6     7      8      9
2     10   11    12     13     14
3     15   16    17     18     19
```

Now it's time to implement the connection to the SQLite3 database.

```
>>> engine = create_engine('sqlite:///foo.db')
```

Convert the DataFrame in a table within the database.

```
>>> frame.to_sql('colors',engine)
```

Instead, to make a reading of the database, you have to use the **read_sql()** function with the name of the table and the **engine**.

```
>>> pd.read_sql('colors',engine)
   index  white  red  blue  black  green
0      0      0    1     2      3      4
1      1      5    6     7      8      9
2      2     10   11    12     13     14
3      3     15   16    17     18     19
```

As you can see, even in this case, the writing operation on the database has become very simple thanks to the I/O APIs available in the pandas library.

Now you'll see instead the same operations, but not using the I/O API. This can be useful to get an idea of how pandas proves to be an effective tool even as regards the reading and writing data to a database.

First, you must establish a connection to the DB and create a table by defining data types corrected, so as to accommodate the data to be loaded.

```
>>> import sqlite3
>>> query = """
... CREATE TABLE test
... (a VARCHAR(20), b VARCHAR(20),
...  c REAL,        d INTEGER
... );"""
>>> con = sqlite3.connect(':memory:')
>>> con.execute(query)
<sqlite3.Cursor object at 0x0000000009E7D730>
>>> con.commit()
```

Now you can enter data through the SQL INSERT statement.

```
>>> data = [('white','up',1,3),
...         ('black','down',2,8),
...         ('green','up',4,4),
...         ('red','down',5,5)]
>>> stmt = "INSERT INTO test VALUES(?,?,?,?)"
>>> con.executemany(stmt, data)
<sqlite3.Cursor object at 0x0000000009E7D8F0>
>>> con.commit()
```

Now that you've seen how to load the data on a table, it is time to see how to query the database to get the data you just recorded. This is possible through an SQL SELECT statement.

```
>>> cursor = con.execute('select * from test')
>>> cursor
<sqlite3.Cursor object at 0x0000000009E7D730>
>>> rows = cursor.fetchall()
>>> rows
[(u'white', u'up', 1.0, 3), (u'black', u'down', 2.0, 8), (u'green', u'up', 4.0, 4),
(u'red', 5.0, 5)]
```

You can pass the list of tuples to the constructor of the DataFrame, and if you need the name of the columns, you can find them within the **description** attribute of the cursor.

```
>>> cursor.description
(('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))
>>> pd.DataFrame(rows, columns=zip(*cursor.description)[0])
       a     b  c  d
0  white    up  1  3
1  black  down  2  8
2  green    up  4  4
3    red  down  5  5
```

As you may well see this approach is quite laborious.

Loading and Writing Data with PostgreSQL

From pandas 0.14 postgresql database is also supported. So double-check if the version on your PC corresponds to this version or greater.

```
>>> pd.__version__
>>> '0.15.2'
```

To make this example you must have installed on your system a PostgreSQL database. In my case I created a database called **postgres**, with 'postgres' as user and 'password' as password. Replace these values with the values corresponding to your system.

Now establish a connection with the database:

```
>>> engine = create_engine('postgresql://postgres:password@localhost:5432/postgres')
```

■ **Note**   In this example, depending on how you installed the package on Windows, often you get the following error message:

```
from psycopg2._psycopg import BINARY, NUMBER, STRING, DATETIME, ROWID
ImportError: DLL load failed: The specified module could not be found.
```

This probably means you don't have the PostgreSQL DLLs (libpq.dll in particular) in your PATH. Add one of the postgres\x.x\bin directories to your PATH and you should be able to connect from Python to your PostgreSQL installations.

Create a DataFrame object:

```
>>> frame = pd.DataFrame(np.random.random((4,4)),
            index=['exp1','exp2','exp3','exp4'],
            columns=['feb','mar','apr','may']);
```

Now we see how easily you can transfer this data to a table. With the **to_sql()** you will record the data in a table called **dataframe**.

```
>>> frame.to_sql('dataframe',engine)
```

pgAdmin III is a graphical application for managing PostgreSQL databases. It's a very useful tool and is present on both Linux and Windows. With this application is easy to see the table dataframe just created (see Figure 5-5).
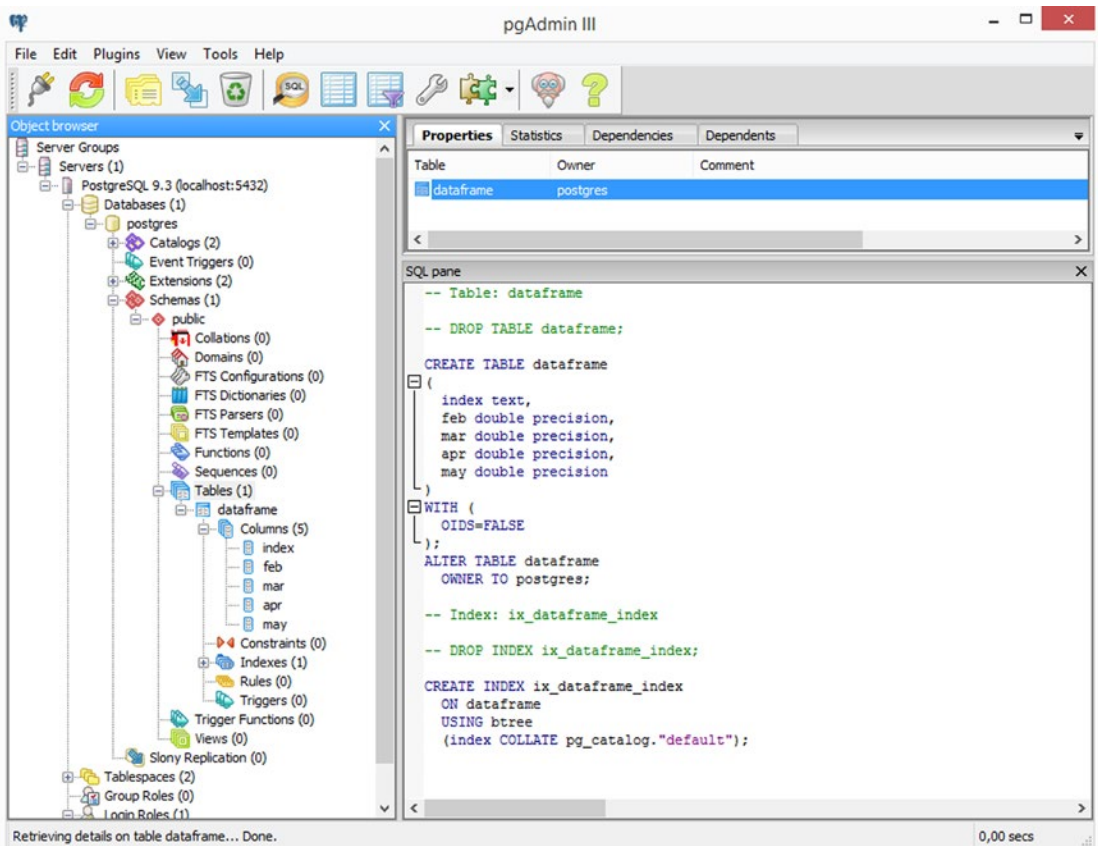
Figure 5-5. *The pdAdminIII application is a perfect graphical DB manager for PostgreSQL*

If you know the SQL language well, a more classic way to see the new created table and its contents is through a **psql** session.

```
>>> psql -U postgres
```

In my case I am connected with the postgres user; it may be different in your case. Once connected to the database, perform an SQL query on the newly created table.

```
postgres=# SELECT * FROM DATAFRAME;
 index |        feb         |        mar         |        apr         |         may
-------+--------------------+--------------------+--------------------+--------------------
 exp1  | 0.757871296789076  | 0.422582915331819  | 0.979085739226726  |  0.332288515791064
 exp2  | 0.124353978978927  | 0.273461421503087  | 0.049433776453223  | 0.0271413946693556
 exp3  | 0.538089036334938  | 0.097041417119426  | 0.905979807772598  |  0.123448718583967
 exp4  | 0.736585422687497  | 0.982331931474687  | 0.958014824504186  |  0.448063967996436
(4 righe)
```

Even the conversion of a table in a DataFrame is a trivial operation. Even here there is a **read_sql_table()** function that reads directly on the database and returns a DataFrame.

```
>>> pd.read_sql_table('dataframe',engine)
   index       feb       mar       apr       may
0   exp1  0.757871  0.422583  0.979086  0.332289
1   exp2  0.124354  0.273461  0.049434  0.027141
2   exp3  0.538089  0.097041  0.905980  0.123449
3   exp4  0.736585  0.982332  0.958015  0.448064
```

But when you want to make a reading of data in a database, the conversion of a whole and single table into a DataFrame is not the most useful operation. In fact, those who work with relational databases prefer to use the SQL language to choose what data and in what form to export by inserting an SQL query.

The text of an SQL query can be integrated in the **read_sql_query()** function.

```
>>> pd.read_sql_query('SELECT index,apr,may FROM DATAFRAME WHERE apr > 0.5',engine)
   index       apr       may
0   exp1  0.979086  0.332289
1   exp3  0.905980  0.123449
2   exp4  0.958015  0.448064
```

Reading and Writing Data with a NoSQL Database: MongoDB

Among all the NoSQL databases (BerkeleyDB, Tokyo Cabinet, MongoDB) MongoDB is becoming the most widespread. Given its diffusion in many systems, it seems appropriate to consider the possibility of reading and writing data produced with the pandas library during a data analysis.

First, if you have MongoDB installed on your PC, you can start the service to point to a given directory.

```
mongod --dbpath C:\MongoDB_data
```

Now that the service is listening on port 27017 you can connect to this database using the official driver for MongoDB: **pymongo**.

```
>>> import pymongo
>>> client = MongoClient('localhost',27017)
```

A single instance of MongoDB is able to support multiple databases at the same time. So now you need to point to a specific database.

```
>>> db = client.mydatabase
>>> db
Database(MongoClient('localhost', 27017), u'mycollection')
In order to refer to this object, you can also use
>>> client['mydatabase']
Database(MongoClient('localhost', 27017), u'mydatabase')
```

Now that you have defined the database, you have to define the collection. The collection is a group of documents stored in MongoDB and can be considered the equivalent of the tables in an SQL database.

```
>>> collection = db.mycollection
>>> db['mycollection']
Collection(Database(MongoClient('localhost', 27017), u'mydatabase'), u'mycollection')
>>> collection
Collection(Database(MongoClient('localhost', 27017), u'mydatabase'), u'mycollection')
Now it is the time to load the data in the collection. Create a DataFrame.
>>> frame = pd.DataFrame( np.arange(20).reshape(4,5),
...                       columns=['white','red','blue','black','green'])
>>> frame
   white  red  blue  black  green
0      0    1     2      3      4
1      5    6     7      8      9
2     10   11    12     13     14
3     15   16    17     18     19
```

Before being added to a collection, it must be converted into a JSON format. The conversion process is not as direct as you might imagine; this is because you need to set the data to be recorded on DB and at the same time in order to be re-extract as DataFrame as fairly and as simply as possible.

```
>>> import json
>>> record = json.loads(frame.T.to_json()).values()
>>> record
[{u'blue': 7, u'green': 9, u'white': 5, u'black': 8, u'red': 6}, {u'blue': 2, u'green': 4,
u'white': 0, u'black': 3, u'red': 1}, {u'blue': 17, u'green': 19, u'white': 15,
u'black': 18, u'red': 16}, {u'blue': 12, u'green': 14, u'white': 10, u'black': 13,
u'red': 11}]
Now you are finally ready to insert a document in the collection, and you can do this with
the insert() function.
>>> collection.mydocument.insert(record)
[ObjectId('54fc3afb9bfbee47f4260357'), ObjectId('54fc3afb9bfbee47f4260358'),
ObjectId('54fc3afb9bfbee47f4260359'), ObjectId('54fc3afb9bfbee47f426035a')]
```

As you can see, you have an object for each line recorded. Now that the data has been loaded into the document within the MongoDB database, you can execute the reverse process, i.e., reading data within a document and then converting them to a DataFrame.

```
>>> cursor = collection['mydocument'].find()
>>> dataframe = (list(cursor))
>>> del dataframe['_id']
>>> dataframe
   black  blue  green  red  white
0      8     7      9    6      5
1      3     2      4    1      0
2     18    17     19   16     15
3     13    12     14   11     10
```

You have removed the column containing the ID numbers for the internal reference of MongoDB.

Conclusions

In this chapter, you saw how to use the features of the I/O API of the pandas library in order to read and write data to files and databases while preserving the structure of the DataFrames. In particular several modes of writing and reading according to the type of format are illustrated.

In the last part of the chapter you have seen how to interface to the most popular models of Database to record and/or read the data into it directly as DataFrame ready to be processed with the pandas tools.

In the next chapter, you'll see the most advanced features of the library pandas. Complex instruments like the GroupBy and other forms of data processing are discussed in detail.

■ ■ ■

pandas in Depth:
Data Manipulation

In the previous chapter you have seen how to acquire data from data sources such as databases or files. Once you have the data in DataFrame format, they are ready to be manipulated. The manipulation of the data has the purpose of preparing the data so that they can be more easily subjected to analysis. In fact, their manipulation will depend a lot on purposes of those who must carry out the analysis, and it will be performed for making more explicit the information you are looking for. Especially in preparation for the next phase, the data must be ready to the data visualization that will follow in the next chapter.

In this chapter you will go in depth with the functionality that the pandas library offers for this stage of the data analysis. The three phases of data manipulation will be treated individually, illustrating the various operations with a series of examples and on how best to use the functions of this library for carrying out such operations. The three phases of data manipulation are

- Data preparation
- Data transformation
- Data aggregation

Data Preparation

Before you start manipulating data itself, it is necessary to prepare the data and assemble them in the form of data structures such that they can be manipulated later with the tools made available by the pandas library. The different procedures for data preparation are listed below.

- loading
- assembling
  - merging
  - concatenating
  - combining
- reshaping (pivoting)
- removing

As regards loading, all of the previous chapter is centered on this topic. In the loading phase, there is also that part of the preparation which concerns the conversion from many different formats into a data structure such as DataFrame. But even after you have gotten the data, probably from different sources and formats, and unified it into a DataFrame, you will need to perform further operations of preparation. In this chapter, and in particular in this section, you'll see how to perform all the operations necessary for the incorporation of data into a unified data structure.

The data contained in the pandas objects can be assembled together in different ways:

- Merging—the **pandas.merge( )** function connects the rows in a DataFrame based on one or more keys. This mode is very familiar to those who are confident with the SQL language, since it also implements join operations.

- Concatenating—the **pandas.concat( )** function concatenates the objects along an axis.

- Combining—the **pandas.DataFrame.combine_first( )** function is a method that allows you to connect overlapped data in order to fill in missing values in a data structure by taking data from another structure.

Furthermore, part of the preparation process is also pivoting, which consists of the exchange between rows and columns.

Merging

The merging operation, which corresponds to the JOIN operation for those who are familiar with SQL, consists of a combination of data through the connection of rows using one or more keys.

In fact, anyone working with relational databases usually makes use of the JOIN query with SQL to get data from different tables using some reference values (keys) shared between them. On the basis of these keys it is possible to obtain new data in a tabular form as the result of the combination of other tables. This operation with the library pandas is called **merging**, and **merge()** is the function to perform this kind of operation.

First, you have to import the pandas library and define two DataFrame that will serve you as examples for this section.

```
>>> import numpy as np
>>> import pandas as pd
>>> frame1 = pd.DataFrame( {'id':['ball','pencil','pen','mug','ashtray'],
...                         'price': [12.33,11.44,33.21,13.23,33.62]})
>>> frame1
        id  price
0      ball  12.33
1    pencil  11.44
2       pen  33.21
3       mug  13.23
4   ashtray  33.62
>>> frame2 = pd.DataFrame( {'id':['pencil','pencil','ball','pen'],
...                         'color': ['white','red','red','black']})
>>> frame2
   color      id
0  white  pencil
1    red  pencil
2    red    ball
3  black     pen
```

Carry out the merging applying the **merge( )** function to the two DataFrame objects.

```
>>> pd.merge(frame1,frame2)
       id  price  color
0    ball  12.33    red
1  pencil  11.44  white
2  pencil  11.44    red
3     pen  33.21  black
```

As you can see from the result, the returned DataFrame consists of all rows that have an **ID** in common between the two DataFeame. In addition to the common column, the columns from both the first and the second DataFrame are added.

In this case you used the **merge( )** function without specifying any column explicitly. In fact, in most cases you need to decide which is the column on which to base the merging.

To do this, add the **on** option with the column name as the key for the merging.

```
>>> frame1 = pd.DataFrame( {'id':['ball','pencil','pen','mug','ashtray'],
...                         'color': ['white','red','red','black','green'],
...                         'brand': ['OMG','ABC','ABC','POD','POD']})
>>> frame1
  brand  color       id
0   OMG  white     ball
1   ABC    red   pencil
2   ABC    red      pen
3   POD  black      mug
4   POD  green  ashtray
>>> frame2 = pd.DataFrame( {'id':['pencil','pencil','ball','pen'],
...                         'brand': ['OMG','POD','ABC','POD']})
>>> frame2
  brand      id
0   OMG  pencil
1   POD  pencil
2   ABC    ball
3   POD     pen
```

Now in this case you have two DataFrame having columns with the same name. So if you launch a merging you do not get any results.

```
>>> pd.merge(frame1,frame2)
Empty DataFrame
Columns: [brand, color, id]
Index: []
```

So it is necessary to explicitly define the criterion of merging that pandas must follow, specifying the name of the key column in the **on** option.

```
>>> pd.merge(frame1,frame2,on='id')
  brand_x  color      id brand_y
0     OMG  white    ball     ABC
1     ABC    red  pencil     OMG
2     ABC    red  pencil     POD
3     ABC    red     pen     POD
```

```
>>> pd.merge(frame1,frame2,on='brand')
  brand  color      id_x     id_y
0   OMG  white      ball    pencil
1   ABC    red    pencil     ball
2   ABC    red       pen     ball
3   POD  black       mug    pencil
4   POD  black       mug      pen
5   POD  green   ashtray    pencil
6   POD  green   ashtray      pen
```

As expected, the results vary considerably depending on the criteria of merging.

Often, however, the opposite problem arises, that is, to have two DataFrames in which the key columns do not have the same name. To remedy this situation, you have to use the **left_on** and **right_on** options that specify the key column for the first and for the second DataFrame. Now you can see an example.

```
>>> frame2.columns = ['brand','sid']
>>> frame2
  brand     sid
0   OMG  pencil
1   POD  pencil
2   ABC    ball
3   POD     pen
>>> pd.merge(frame1, frame2, left_on='id', right_on='sid')
  brand_x  color      id brand_y     sid
0     OMG  white    ball     ABC    ball
1     ABC    red  pencil     OMG  pencil
2     ABC    red  pencil     POD  pencil
3     ABC    red     pen     POD     pen
```

By default, the **merge( )** function performs an **inner join;** the keys in the result are the result of an intersection.

Other possible options are the **left join**, the **right join,** and the **outer join**. The outer join produces the union of all keys, combining the effect of a left join with a right join. To select the type of join you have to use the **how** option.

```
>>> frame2.columns['brand','id']
>>> pd.merge(frame1,frame2,on='id')
  brand_x  color      id brand_y
0     OMG  white    ball     ABC
1     ABC    red  pencil     OMG
2     ABC    red  pencil     POD
3     ABC    red     pen     POD
>>> pd.merge(frame1,frame2,on='id',how='outer')
  brand_x  color      id brand_y
0     OMG  white    ball     ABC
1     ABC    red  pencil     OMG
2     ABC    red  pencil     POD
3     ABC    red     pen     POD
4     POD  black     mug     NaN
5     POD  green ashtray     NaN
```

```
>>> pd.merge(frame1,frame2,on='id',how='left')
  brand_x  color      id brand_y
0    OMG  white    ball     ABC
1    ABC    red  pencil     OMG
2    ABC    red  pencil     POD
3    ABC    red     pen     POD
4    POD  black     mug     NaN
5    POD  green ashtray     NaN
>>> pd.merge(frame1,frame2,on='id',how='right')
  brand_x  color      id brand_y
0    OMG  white    ball     ABC
1    ABC    red  pencil     OMG
2    ABC    red  pencil     POD
3    ABC    red     pen     POD
```

To make the merge of multiple keys, you simply just add a list to the **on** option.

```
>>> pd.merge(frame1,frame2,on=['id','brand'],how='outer')
  brand  color      id
0   OMG  white    ball
1   ABC    red  pencil
2   ABC    red     pen
3   POD  black     mug
4   POD  green ashtray
5   OMG    NaN  pencil
6   POD    NaN  pencil
7   ABC    NaN    ball
8   POD    NaN     pen
```

Merging on Index

In some cases, instead of considering the columns of a DataFrame as keys, the indexes could be used as keys on which to make the criteria for merging. Then in order to decide which indexes to consider, set the **left_index** or **right_index** options to True to activate them, with the ability to activate them both.

```
>>> pd.merge(frame1,frame2,right_index=True, left_index=True)
  brand_x  color  id_x brand_y   id_y
0    OMG  white  ball     OMG pencil
1    ABC    red pencil     POD pencil
2    ABC    red   pen     ABC   ball
3    POD  black   mug     POD    pen
```

But the DataFrame objects have a **join()** function which is much more convenient when you want to do the merging by indexes. It can also be used to combine many DataFrame objects having the same or the same indexes but with columns not overlapping.

In fact, if you launch

```
>>> frame1.join(frame2)
```

CHAPTER 6 ■ PANDAS IN DEPTH: DATA MANIPULATION

You will get an error code because some columns of the frame1 have the same name of frame2. Then rename the columns of frame2 before launching the **join( )** function.

```
>>> frame2.columns = ['brand2','id2']
>>> frame1.join(frame2)
  brand  color       id brand2     id2
0   OMG  white     ball    OMG  pencil
1   ABC    red   pencil    POD  pencil
2   ABC    red      pen    ABC    ball
3   POD  black      mug    POD     pen
4   POD  green  ashtray    NaN     NaN
```

Here you've performed a merging but based on the values of the indexes instead of the columns. This time there is also the index 4 that was present only in frame1, but the values corresponding to the columns of frame2 report NaN as value.

Concatenating

Another type of data combination is referred to as **concatenation**. NumPy provides a **concatenate()** function to do this kind of operation with arrays.

```
>>> array1
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> array2 = np.arange(9).reshape((3,3))+6
>>> array2
array([[ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])
>>> np.concatenate([array1,array2],axis=1)
array([[ 0,  1,  2,  6,  7,  8],
       [ 3,  4,  5,  9, 10, 11],
       [ 6,  7,  8, 12, 13, 14]])
>>> np.concatenate([array1,array2],axis=0)
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])
```

As regards the pandas library and its data structures like Series and DataFrame, the fact of having labeled axes allows you to further generalize the concatenation of arrays. The **concat()** function is provided by pandas for this kind of operation.

```
>>> ser1 = pd.Series(np.random.rand(4), index=[1,2,3,4])
>>> ser1
1    0.636584
2    0.345030
3    0.157537
4    0.070351
dtype: float64
>>> ser2 = pd.Series(np.random.rand(4), index=[5,6,7,8])
>>> ser2
5    0.411319
6    0.359946
7    0.987651
8    0.329173
dtype: float64
>>> pd.concat([ser1,ser2])
1    0.636584
2    0.345030
3    0.157537
4    0.070351
5    0.411319
6    0.359946
7    0.987651
8    0.329173
dtype: float64
```

By default, the **concat()** function works on axis = 0, having as returned object a Series. If you set the axis = 1, then the result will be a DataFrame.

```
>>> pd.concat([ser1,ser2],axis=1)
          0          1
1  0.636584        NaN
2  0.345030        NaN
3  0.157537        NaN
4  0.070351        NaN
5       NaN  0.411319
6       NaN  0.359946
7       NaN  0.987651
8       NaN  0.329173
```

From the result you can see that there is no overlap of data, therefore what you have just done is an outer join. This can be changed by setting the **join** option to 'inner'.

```
>>> pd.concat([ser1,ser3],axis=1,join='inner')
          0          0   1
1  0.636584   0.636584 NaN
2  0.345030   0.345030 NaN
3  0.157537   0.157537 NaN
4  0.070351   0.070351 NaN
```

A problem in this kind of operation is that the concatenated parts are not identifiable in the result. For example, you want to create a hierarchical index on the axis of concatenation. To do this you have to use the **keys** option.

```
>>> pd.concat([ser1,ser2], keys=[1,2])
1  1    0.636584
   2    0.345030
   3    0.157537
   4    0.070351
2  5    0.411319
   6    0.359946
   7    0.987651
   8    0.329173
dtype: float64
```

In the case of combinations between Series along the axis = 1 the keys become the column headers of the DataFrame.

```
>>> pd.concat([ser1,ser2], axis=1, keys=[1,2])
          1          2
1  0.636584        NaN
2  0.345030        NaN
3  0.157537        NaN
4  0.070351        NaN
5       NaN   0.411319
6       NaN   0.359946
7       NaN   0.987651
8       NaN   0.329173
```

So far you have seen the concatenation applied to the Series, but the same logic can be applied to the DataFrame.

```
>>> frame1 = pd.DataFrame(np.random.rand(9).reshape(3,3), index=[1,2,3],
columns=['A','B','C'])
>>> frame2 = pd.DataFrame(np.random.rand(9).reshape(3,3), index=[4,5,6],
columns=['A','B','C'])
>>> pd.concat([frame1, frame2])
          A          B          C
1  0.400663   0.937932   0.938035
2  0.202442   0.001500   0.231215
3  0.940898   0.045196   0.723390
4  0.568636   0.477043   0.913326
5  0.598378   0.315435   0.311443
6  0.619859   0.198060   0.647902

>>> pd.concat([frame1, frame2], axis=1)
          A          B          C          A          B          C
1  0.400663   0.937932   0.938035        NaN        NaN        NaN
2  0.202442   0.001500   0.231215        NaN        NaN        NaN
3  0.940898   0.045196   0.723390        NaN        NaN        NaN
4       NaN        NaN        NaN   0.568636   0.477043   0.913326
5       NaN        NaN        NaN   0.598378   0.315435   0.311443
6       NaN        NaN        NaN   0.619859   0.198060   0.647902
```

Combining

There is another situation in which there is combination of data that cannot be obtained either with merging or with concatenation. Take the case in which you want the two datasets to have indexes that overlap in their entirety or at least partially.

One applicable function to Series is **combine_first()**, which performs this kind of operation along with an data alignment.

```
>>> ser1 = pd.Series(np.random.rand(5),index=[1,2,3,4,5])
>>> ser1
1    0.942631
2    0.033523
3    0.886323
4    0.809757
5    0.800295
dtype: float64
>>> ser2 = pd.Series(np.random.rand(4),index=[2,4,5,6])
>>> ser2
2    0.739982
4    0.225647
5    0.709576
6    0.214882
dtype: float64
>>> ser1.combine_first(ser2)
1    0.942631
2    0.033523
3    0.886323
4    0.809757
5    0.800295
6    0.214882
dtype: float64
>>> ser2.combine_first(ser1)
1    0.942631
2    0.739982
3    0.886323
4    0.225647
5    0.709576
6    0.214882
dtype: float64
```

Instead, if you want a partial overlap, you can specify only the portion of the Series you want to overlap.

```
>>> ser1[:3].combine_first(ser2[:3])
1    0.942631
2    0.033523
3    0.886323
4    0.225647
5    0.709576
dtype: float64
```

Pivoting

In addition to assembling the data in order to unify the values collected from different sources, another fairly common operation is **pivoting**. In fact, arrangement of the values by row or by column is not always suited to your goals. Sometimes you would like to rearrange the data carrying column values on rows or vice versa.

Pivoting with Hierarchical Indexing

You have already seen that DataFrame can support hierarchical indexing. This feature can be exploited to rearrange the data in a DataFrame. In the context of pivoting you have two basic operations:

- stacking: rotates or pivots the data structure converting columns to rows
- unstacking: converts rows into columns

```
>>> frame1 = pd.DataFrame(np.arange(9).reshape(3,3),
...                       index=['white','black','red'],
...                       columns=['ball','pen','pencil'])
>>> frame1
       ball  pen  pencil
white     0    1       2
black     3    4       5
red       6    7       8
```

Using the **stack()** function on the DataFrame, you will get the pivoting of the columns in rows, thus producing a Series:

```
>>> frame1.stack()
white  ball      0
       pen       1
       pencil    2
black  ball      3
       pen       4
       pencil    5
red    ball      6
       pen       7
       pencil    8
dtype: int32
```

From this hierarchically indexed series, you can reassemble the DataFrame into a pivoted table by use of the unstack() function.

```
>>> ser5.unstack()
       ball  pen  pencil
white     0    1       2
black     3    4       5
red       6    7       8
```

You can also do the unstack on a different level, specifying the number of levels or its name as the argument of the function.

```
>>> ser5.unstack(0)
        white  black  red
ball        0      3    6
pen         1      4    7
pencil      2      5    8
```

Pivoting from "Long" to "Wide" Format

The most common way to store data sets is produced by the punctual registration of data that will fill a line of the text file, for example, CSV, or a table of a database. This happens especially when you have instrumental readings, calculation results iterated over time, or the simple manual input of a series of values. A similar case of these files is for example the logs file, which is filled line by line by accumulating data in it.

   The peculiar characteristic of this type of data set is to have entries on various columns, often duplicated in subsequent lines. Always remaining in tabular format of data, when you are in such cases you can refer them to as **long** or **stacked** format.

   To get a clearer idea about that, for example, consider the following DataFrame.

```
>>> longframe = pd.DataFrame({ 'color':['white','white','white',
...                                     'red','red','red',
...                                     'black','black','black'],
...                           'item':['ball','pen','mug',
...                                   'ball','pen','mug',
...                                   'ball','pen','mug'],
...                           'value': np.random.rand(9)})
>>> longframe
   color  item     value
0  white  ball  0.091438
1  white   pen  0.495049
2  white   mug  0.956225
3    red  ball  0.394441
4    red   pen  0.501164
5    red   mug  0.561832
6  black  ball  0.879022
7  black   pen  0.610975
8  black   mug  0.093324
```

   This mode of data recording, however, has some disadvantages. One, for example, is precisely the multiplicity and repetition of some fields. Considering the columns as keys, the data with this format will be difficult to read, especially in fully understanding the relationships between the key values and the rest of the columns.

   Instead of the long format, there is another way to arrange the data in a table that is called **wide**. This mode is easier to read, allowing easy connection with other tables, and it occupies much less space. So in general it is a more efficient way of storing the data, although less practical, especially if during the filling of the data.

   As a criterion, select a column, or a set of them, as the primary key; then, the values contained in it must be unique.

   In this regard, pandas gives you a function that allows you to make a transformation of a DataFrame from the long type to the wide type. This function is **pivot()** and it accepts as arguments the column, or columns, which will assume the role of key.

Starting from the previous example you choose to create a DataFrame in wide format by choosing the **color** column as the key, and **item** as a second key, the values of which will form the new columns of the data frame.

```
>>> wideframe = longframe.pivot('color','item')
>>> wideframe
         value
item       ball       mug       pen
color
black   0.879022  0.093324  0.610975
red     0.394441  0.561832  0.501164
white   0.091438  0.956225  0.495049
```

As you can now see, in this format, the DataFrame is much more compact and data contained in it are much more readable.

Removing

The last stage of data preparation is the removal of columns and rows. You have already seen this part in Chapter 4. However, for completeness, the description is reiterated here. Define a DataFrame by way of example.

```
>>> frame1 = pd.DataFrame(np.arange(9).reshape(3,3),
...                   index=['white','black','red'],
...                   columns=['ball','pen','pencil'])
>>> frame1
       ball  pen  pencil
white     0    1       2
black     3    4       5
red       6    7       8
```

In order to remove a column, you have to simply use the **del** command applied to the DataFrame with the column name specified.

```
>>> del frame1['ball']
>>> frame1
       pen  pencil
white    1       2
black    4       5
red      7       8
```

Instead, to remove an unwanted row, you have to use the **drop()** function with the label of the corresponding index as argument.

```
>>> frame1.drop('white')
       pen  pencil
black    4       5
red      7       8
```

Data Transformation

So far you have seen how to prepare data for analysis. This process in effect represents a reassembly of the data contained within a DataFrame, with possible additions by other DataFrame and removal of unwanted parts.

Now begin with the second stage of data manipulation: the **data transformation**. After you arrange the form of data and their disposal within the data structure, it is important to transform their values. In fact, in this section you will see some common issues and the steps required to overcome them using functions of the pandas library.

Some of these operations involve the presence of duplicate or invalid values, with possible removal or replacement. Other operations relate instead modifying the indexes. Other steps include handling and processing the numerical values of the data and also of strings.

Removing Duplicates

Duplicate rows might be present in a DataFrame for various reasons. In DataFrames of enormous size the detection of these rows can be very problematic. Also in this case, pandas provides us with a series of tools to analyze the duplicate data present in large data structures.

First, create a simple DataFrame with some duplicate rows.

```
>>> dframe = pd.DataFrame({ 'color': ['white','white','red','red','white'],
...                         'value': [2,1,3,3,2]})
>>> dframe
   color  value
0  white      2
1  white      1
2    red      3
3    red      3
4  white      2
```

The **duplicated()** function applied to a DataFrame can detect the rows which appear to be duplicated. It returns a Series of Booleans where each element corresponds to a row, with **True** if the row is duplicated (i.e., only the other occurrences, not the first), and with **False** if there are no duplicates in the previous elements.

```
>>> dframe.duplicated()
0    False
1    False
2    False
3     True
4     True
dtype: bool
```

The fact of having as the return value a Boolean Series can be useful in many cases, especially for the filtering. In fact, if you want to know what are the duplicate rows, just type the following:

```
>>> dframe[dframe.duplicated()]
   color  value
3    red      3
4  white      2
```

Generally, all duplicated rows are to be deleted from the DataFrame; to do that, pandas provides the **drop_duplicates()** function, which returns the DataFrame without duplicate rows.

```
>>> dframe[dframe.duplicated()]
   color  value
3    red      3
4  white      2
```

Mapping

The pandas library provides a set of functions which, as you shall see in this section, exploit mapping to perform some operations. The mapping is nothing more than the creation of a list of matches between two different values, with the ability to bind a value to a particular label or string.

To define a mapping there is no better object than dict objects.

```
map = {
   'label1' : 'value1,
   'label2' : 'value2,
   ...
}
```

The functions that you will see in this section perform specific operations but all of them are united from accepting a dict object with matches as an argument.

- **replace():** replaces values

- **map():** creates a new column

- **rename():** replaces the index values

Replacing Values via Mapping

Often in the data structure that you have assembled there are values that do not meet your needs. For example, the text may be in a foreign language, or may be a synonym of another value, or may not be expressed in the desired shape. In such cases, a replace operation of various values is often a necessary process.

Define, as an example, a DataFrame containing various objects and colors, including two colors that are not in English. Often during the assembly operations is likely to keep maintaining data with values in a form that is not desired.

```
>>> frame = pd.DataFrame({ 'item':['ball','mug','pen','pencil','ashtray'],
...                       'color':['white','rosso','verde','black','yellow'],
                          'price':[5.56,4.20,1.30,0.56,2.75]})
>>> frame
    color     item
0   white     ball
1     red      mug
2   green      pen
3   black   pencil
4  yellow  ashtray
```

Thus to be able to replace the incorrect values in new values is necessary to define a mapping of correspondences, containing as key to replace the old values and values as the new ones.

```
>>> newcolors = {
...     'rosso': 'red',
...     'verde': 'green'
... }
```

Now the only thing you can do is to use the **replace()** function with the mapping as an argument.

```
>>> frame.replace(newcolors)
     color      item  price
0    white      ball   5.56
1      red       mug   4.20
2    green       pen   1.30
3    black    pencil   0.56
4   yellow   ashtray   2.75
```

As you can see from the result, the two colors have been replaced with the correct values within the DataFrame. A common case, for example, is the replacement of the NaN values with another value, for example 0. Also here you can use the **replace(),** which performs its job very well.

```
>>> ser = pd.Series([1,3,np.nan,4,6,np.nan,3])
>>> ser
0      1
1      3
2    NaN
3      4
4      6
5    NaN
6      3
dtype: float64
>>> ser.replace(np.nan,0)
0      1
1      3
2      0
3      4
4      6
5      0
6      3
dtype: float64
```

Adding Values via Mapping

In the previous example, you have seen the case of the substitution of values through a mapping of correspondences. In this case you continue to exploit the mapping of values with another example. In this case you are exploiting mapping to add values in a column depending on the values contained in another. The mapping will always be defined separately.

```
>>> frame = pd.DataFrame({ 'item':['ball','mug','pen','pencil','ashtray'],
...                        'color':['white','red','green','black','yellow']})
>>> frame
    color     item
0   white     ball
1     red      mug
2   green      pen
3   black   pencil
4  yellow  ashtray
```

Let's suppose you want to add a column to indicate the price of the item shown in the DataFrame. Before you do this, it is assumed that you have a price list available somewhere, in which the price for each type of item is described. Define then a dict object that contains a list of prices for each type of item.

```
>>> price = {
...     'ball' : 5.56,
...     'mug' : 4.20,
...     'bottle' : 1.30,
...     'scissors' : 3.41,
...     'pen' : 1.30,
...     'pencil' : 0.56,
...     'ashtray' : 2.75
... }
```

The **map()** function applied to a Series or to a column of a DataFrame accepts a function or an object containing a dict with mapping. So in your case you can apply the mapping of the prices on the column item, making sure to add a column to the price data frame.

```
>>> frame['price'] = frame['item'].map(prices)
>>> frame
    color     item  price
0   white     ball   5.56
1     red      mug   4.20
2   green      pen   1.30
3   black   pencil   0.56
4  yellow  ashtray   2.75
```

Rename the Indexes of the Axes

In a manner very similar to what you saw for the values contained within the Series and the DataFrame, even the axis label can be transformed in a very similar way using the mapping. So to replace the label indexes, pandas provides the **rename()** function, which takes the mapping as argument, that is, a dict object.

```
>>> frame
    color     item  price
0   white     ball   5.56
1     red      mug   4.20
2   green      pen   1.30
3   black   pencil   0.56
4  yellow  ashtray   2.75
```

```
>>> reindex = {
...    0: 'first',
...    1: 'second',
...    2: 'third',
...    3: 'fourth',
...    4: 'fifth'}
>>> frame.rename(reindex)
        color     item  price
first    white     ball   5.56
second     red      mug   4.20
third    green      pen   1.30
fourth   black   pencil   0.56
fifth   yellow  ashtray   2.75
```

As you can see, by default, the indexes are renamed. If you want to rename columns you must use the **columns** option. Thus this time you assign various mapping explicitly to the two **index** and **columns** options.

```
>>> recolumn = {
...    'item':'object',
...    'price': 'value'}
>>> frame.rename(index=reindex, columns=recolumn)
        color   object  value
first    white     ball   5.56
second     red      mug   4.20
third    green      pen   1.30
fourth   black   pencil   0.56
fifth   yellow  ashtray   2.75
```

Also here, for the simplest cases in which you have a single value to be replaced, it can further explicate the arguments passed to the function of avoiding having to write and assign many variables.

```
>>> frame.rename(index={1:'first'}, columns={'item':'object'})
        color   object  price
0        white     ball   5.56
first      red      mug   4.20
2        green      pen   1.30
3        black   pencil   0.56
4       yellow  ashtray   2.75
```

So far you have seen that the **rename()** function returns a DataFrame with the changes, leaving unchanged the original DataFrame. If you want the changes to take effect on the object on which you call the function, you will set the **inplace** option to True.

```
>>> frame.rename(columns={'item':'object'}, inplace=True)
>>> frame
    color   object  price
0   white     ball   5.56
1     red      mug   4.20
2   green      pen   1.30
3   black   pencil   0.56
4  yellow  ashtray   2.75
```

Discretization and Binning

A more complex process of transformation that you will see in this section is **discretization**. Sometimes it can happen, especially in some experimental cases, to handle large quantities of data generated in sequence. To carry out an analysis of the data, however, it is necessary to transform this data into discrete categories, for example, by dividing the range of values of such readings in smaller intervals and counting the occurrence or statistics within each of them. Another case might be to have a huge amount of samples due to precise readings on a population. Even here, to facilitate analysis of the data it is necessary to divide the range of values into categories and then analyze the occurrences and statistics related to each of them.

In your case, for example, you may have a reading of an experimental value between 0 and 100. These data are collected in a list.

```
>>> results = [12,34,67,55,28,90,99,12,3,56,74,44,87,23,49,89,87]
```

You know that the experimental values have a range from 0 to 100; therefore you can uniformly divide this interval, for example, into four equal parts, i.e., bins. The first contains the values between 0 and 25, the second between 26 and 50, the third between 51 and 75, and the last between 76 and 100.

To do this binning with pandas, first you have to define an array containing the values of separation of bin:

```
>>> bins = [0,25,50,75,100]
```

Then there is a special function called **cut()** and apply it to the array of results also passing the bins.

```
>>> cat = pd.cut(results, bins)
>>> cat
   (0, 25]
  (25, 50]
  (50, 75]
  (50, 75]
  (25, 50]
 (75, 100]
 (75, 100]
   (0, 25]
   (0, 25]
  (50, 75]
  (50, 75]
  (25, 50]
 (75, 100]
   (0, 25]
  (25, 50]
 (75, 100]
 (75, 100]
Levels (4): Index(['(0, 25]', '(25, 50]', '(50, 75]', '(75, 100]'], dtype=object)
```

The object returned by the **cut()** function is a special object of **Categorical** type. You can consider it as an array of strings indicating the name of the bin. Internally it contains a **levels** array indicating the names of the different internal categories and a **labels** array that contains a list of numbers equal to the elements of **results** (i.e., the array subjected to binning). The number corresponds to the bin to which the corresponding element of **results** is assigned.

```
>>> cat.levels
Index([u'(0, 25]', u'(25, 50]', u'(50, 75]', u'(75, 100]'], dtype='object')
>>> cat.labels
array([0, 1, 2, 2, 1, 3, 3, 0, 0, 2, 2, 1, 3, 0, 1, 3, 3], dtype=int64)
```

Finally to know the occurrences for each bin, that is, how many results fall into each category, you have to use the value_**counts()** function.

```
>>> pd.value_counts(cat)
(75, 100]    5
(0, 25]      4
(25, 50]     4
(50, 75]     4
dtype: int64
```

As you can see, each class has the lower limit with a bracket and the upper limit with a parenthesis. This notation is consistent with mathematical notation that is used to indicate the intervals. If the bracket is square, the number belongs to the range (limit closed), and if it is round the number does not belong to the interval (limit open).

You can give names to various bins by calling them first in an array of strings and then assigning to the labels options inside the cut() function that you have used to create the Categorical object.

```
>>> bin_names = ['unlikely','less likely','likely','highly likely']
>>> pd.cut(results, bins, labels=bin_names)
      unlikely
   less likely
        likely
        likely
   less likely
 highly likely
 highly likely
      unlikely
      unlikely
        likely
        likely
   less likely
 highly likely
      unlikely
   less likely
 highly likely
 highly likely
Levels (4): Index(['unlikely', 'less likely', 'likely', 'highly likely'], dtype=object)
```

If the **cut()** function is passed as an argument to an integer instead of explicating the bin edges, this will divide the range of values of the array in many intervals as specified by the number.

The limits of the interval will be taken by the minimum and maximum of the sample data, namely, the array subjected to binning.

```
>>> pd.cut(results, 5)
 (2.904, 22.2]
  (22.2, 41.4]
  (60.6, 79.8]
  (41.4, 60.6]
  (22.2, 41.4]
    (79.8, 99]
    (79.8, 99]
 (2.904, 22.2]
 (2.904, 22.2]
  (41.4, 60.6]
  (60.6, 79.8]
  (41.4, 60.6]
    (79.8, 99]
  (22.2, 41.4]
  (41.4, 60.6]
    (79.8, 99]
    (79.8, 99]
Levels (5): Index(['(2.904, 22.2]', '(22.2, 41.4]', '(41.4, 60.6]',
                   '(60.6, 79.8]', '(79.8, 99]'], dtype=object)
```

In addition to **cut()**, pandas provides another method for binning: **qcut()**. This function divides the sample directly into quintiles. In fact, depending on the distribution of the data sample, using **cut()** rightly you will have a different number of occurrences for each bin. Instead **qcut()** will ensure that the number of occurrences for each bin is equal, but the edges of each bin to vary.

```
>>> quintiles = pd.qcut(results, 5)
>>> quintiles
    [3, 24]
   (24, 46]
 (62.6, 87]
 (46, 62.6]
   (24, 46]
   (87, 99]
   (87, 99]
    [3, 24]
    [3, 24]
 (46, 62.6]
 (62.6, 87]
   (24, 46]
 (62.6, 87]
    [3, 24]
 (46, 62.6]
   (87, 99]
 (62.6, 87]
Levels (5): Index(['[3, 24]', '(24, 46]', '(46, 62.6]', '(62.6, 87]',
                   '(87, 99]'], dtype=object)
```

```
>>> pd.value_counts(quintiles)
[3, 24]      4
(62.6, 87]   4
(87, 99]     3
(46, 62.6]   3
(24, 46]     3
dtype: int64
```

As you can see, in the case of quintiles, the intervals bounding the bin differ from those generated by the **cut()** function. Moreover, if you look at the occurrences for each bin will find that **qcut()** tried to standardize the occurrences for each bin, but in the case of quintiles, the first two bins have an occurrence in more because the number of results is not divisible by five.

Detecting and Filtering Outliers

During the data analysis, the need to detect the presence of abnormal values within a data structure often arises. By way of example, create a DataFrame with three columns from 1,000 completely random values:

```
>>> randframe = pd.DataFrame(np.random.randn(1000,3))
```

With the **describe()** function you can see the statistics for each column.

```
>>> randframe.describe()
                 0            1            2
count  1000.000000  1000.000000  1000.000000
mean      0.021609    -0.022926    -0.019577
std       1.045777     0.998493     1.056961
min      -2.981600    -2.828229    -3.735046
25%      -0.675005    -0.729834    -0.737677
50%       0.003857    -0.016940    -0.031886
75%       0.738968     0.619175     0.718702
max       3.104202     2.942778     3.458472
```

For example, you might consider outliers those that have a value greater than three times the standard deviation. To have only the standard deviation of each column of the DataFrame, use the **std()** function.

```
>>> randframe.std()
0    1.045777
1    0.998493
2    1.056961
dtype: float64
```

Now you apply the filtering of all the values of the DataFrame, applying the corresponding standard deviation for each column. Thanks to the **any()** function, you can apply the filter on each column.

```
>>> randframe[(np.abs(randframe) > (3*randframe.std())).any(1)]
            0         1         2
69  -0.442411 -1.099404  3.206832
576 -0.154413 -1.108671  3.458472
907  2.296649  1.129156 -3.735046
```

Permutation

The operations of permutation (random reordering) of a Series or the rows of a DataFrame are easy to do using the **numpy.random.permutation()** function.

For this example, create a DataFrame containing integers in ascending order.

```
>>> nframe = pd.DataFrame(np.arange(25).reshape(5,5))
>>> nframe
    0   1   2   3   4
0   0   1   2   3   4
1   5   6   7   8   9
2  10  11  12  13  14
3  15  16  17  18  19
4  20  21  22  23  24
```

Now create an array of five integers from 0 to 4 arranged in random order with the **permutation()** function. This will be the new order in which to set the values of a row of DataFrame.

```
>>> new_order = np.random.permutation(5)
>>> new_order
array([2, 3, 0, 1, 4])
```

Now apply it to the DataFrame on all lines, using the **take()** function.

```
>>> nframe.take(new_order)
    0   1   2   3   4
2  10  11  12  13  14
3  15  16  17  18  19
0   0   1   2   3   4
1   5   6   7   8   9
4  20  21  22  23  24
```

As you can see, the order of the rows has been changed; now the indices follow the same order as indicated in the **new_order** array.

You can submit even a portion of the entire DataFrame to a permutation. It generates an array that has a sequence limited to a certain range, for example, in our case from 2 to 4.

```
>>> new_order = [3,4,2]
>>> nframe.take(new_order)
    0   1   2   3   4
3  15  16  17  18  19
4  20  21  22  23  24
2  10  11  12  13  14
```

Random Sampling

You have just seen how to extract a portion of the DataFrame determined by subjecting it to permutation. Sometimes, when you have a huge DataFrame, you may have the need to sample it randomly, and the quickest way to do this is by using the **np.random.randint()** function.

```
>>> sample = np.random.randint(0, len(nframe), size=3)
>>> sample
array([1, 4, 4])
>>> nframe.take(sample)
    0   1   2   3   4
1   5   6   7   8   9
4  20  21  22  23  24
4  20  21  22  23  24
```

As you can see from this random sampling you can get the same sample even more times.

String Manipulation

Python is a popular language thanks to its ease of use in the processing of strings and text. Most operations can easily be made by using built-in functions provided by Python. For more complex cases of matching and manipulation, it is necessary the use of regular expressions.

Built-in Methods for Manipulation of Strings

In many cases you have composite strings in which you would like to separate the various parts and then assign them to the correct variables. The **split()** function allows us to separate parts of a text, taking as a reference point a separator, for example a comma.

```
>>> text = '16 Bolton Avenue , Boston'
>>> text.split(',')
['16 Bolton Avenue ', 'Boston']
```

As we can see in the first element, you have a string with a space character at the end. To overcome this problem and often a frequent problem, you have to use the **split()** function along with the **strip()** function that takes care of doing the trim of whitespace (including newlines).

```
>>> tokens = [s.strip() for s in text.split(',')]
>>> tokens
['16 Bolton Avenue', 'Boston']
```

The result is an array of strings. If the number of elements is small and always the same, a very interesting way to make assignments may be this:

```
>>> address, city = [s.strip() for s in text.split(',')]
>>> address
'16 Bolton Avenue'
>>> city
'Boston'
```

So far you have seen how to split a text into parts, but often you also need the opposite, namely concatenating various strings between them to form a more extended text.

The most intuitive and simple way is to concatenate the various parts of the text with the operator '+'.

```
>>> address + ',' + city
'16 Bolton Avenue, Boston'
```

This can be useful when you have only two or three strings to be concatenated. If the parts to be concatenated are much more, a more practical approach in this case will be to use the **join()** function assigned to the separator character, with which you want to join the various strings between them.

```
>>> strings = ['A+','A','A-','B','BB','BBB','C+']
>>> ';'.join(strings)
'A+;A;A-;B;BB;BBB;C+'
```

Another category of operations that can be performed on the string is the search for pieces of text in them, i.e., substrings. Python provides, in this respect, the keyword which represents the best way of detecting substrings.

```
>>> 'Boston' in text
True
```

However, there are two functions that could serve to this purpose: **index()** and **find()**.

```
>>> text.index('Boston')
19
>>> text.find('Boston')
19
```

In both cases, it returns the number of the corresponding character in the text where you have the substring. The difference in the behavior of these two functions can be seen, however, when the substring is not found:

```
>>> text.index('New York')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
>>> text.find('New York')
-1
```

In fact, the **index()** function returns an error message, and **find()** returns -1 if the substring is not found. In the same area, you can know how many times a character or combination of characters (substring) occurs within a text. The **count()** function provides you with this number.

```
>>> text.count('e')
2
>>> text.count('Avenue')
1
```

Another operation that can be performed on strings is the replacement or elimination of a substring (or a single character). In both cases you will use the **replace()** function, where if you are prompted to replace a substring with a blank character, the operation will be equivalent to the elimination of the substring from the text.

```
>>> text.replace('Avenue','Street')
'16 Bolton Street , Boston'
>>> text.replace('1','')
'16 Bolton Avenue, Boston'
```

Regular Expressions

The regular expressions provide a very flexible way to search and match string patterns within a text. A single expression, generically called **regex**, is a string formed according to the regular expression language. There is a built-in Python module called **re**, which is responsible for the operation of the regex.

So first of all, when you want to make use of regular expressions, you will need to import the module.

```
>>> import re
```

The **re** module provides a set of functions that can be divided into three different categories:

- pattern matching
- substitution
- splitting

Now you start with a few examples. For example, the regex for expressing a sequence of one or more whitespace characters is **\s+**. As you saw in the previous section, to split a text into parts through a separator character you used the **split()**. There is a **split()** function even for the **re** module that performs the same operations, only it is able to accept a regex pattern as the criterion of separation, which makes it considerably more flexible.

```
>>> text = "This is      an\t odd  \n text!"
>>> re.split('\s+', text)
['This', 'is', 'an', 'odd', 'text!']
```

But analyze more deeply the mechanism of **re** module. When you call the **re.split()** function, the regular expression is first compiled, then subsequently calls the **split()** function on the text argument. You can compile the regex function with the **re.compile()** function, thus obtaining a reusable object regex and so gaining in terms of CPU cycles.

This is especially true in the operations of iterative search of a substring in a set or an array of strings.

```
>>> regex = re.compile('\s+')
```

So if you make an **regex** object with the **compile()** function, you can apply **split()** directly to it in the following way.

```
>>> regex.split(text)
['This', 'is', 'an', 'odd', 'text!']
```

As regards matching a regex pattern with any other business substrings in the text, you can use the **findall()** function. It returns a list of all the substrings in the text that meet the requirements of the regex.

For example, if you want to find in a string all the words starting with "A" uppercase, or for example, with "a" regardless whether upper- or lowercase, you need to enter what follows:

```
>>> text = 'This is my address: 16 Bolton Avenue, Boston'
>>> re.findall('A\w+',text)
['Avenue']
>>> re.findall('[A,a]\w+',text)
['address', 'Avenue']
```

There are two other functions related to the function **findall()**: **match()** and **search()**. While **findall()** returns all matches within a list, the function **search()** returns only the first match. Furthermore, the object returned by this function is a particular object:

```
>>> re.search('[A,a]\w+',text)
<_sre.SRE_Match object at 0x0000000007D7ECC8>
```

This object does not contain the value of the substring that responds to the regex pattern, but its start and end positions within the string.

```
>>> search = re.search('[A,a]\w+',text)
>>> search.start()
11
>>> search.end()
18
>>> text[search.start():search.end()]
'address'
```

The **match()** function performs the matching only at the beginning of the string; if there is no match with the first character, it goes no further in research within the string. If you do not find any match then it will not return any objects.

```
>>> re.match('[A,a]\w+',text)
>>>
```

If **match()** has a response, then it returns an object identical to what you saw for the **search()** function.

```
>>> re.match('T\w+',text)
<_sre.SRE_Match object at 0x0000000007D7ECC8>
>>> match = re.match('T\w+',text)
>>> text[match.start():match.end()]
'This'
```

Data Aggregation

The last stage of data manipulation is data aggregation. For data aggregation you generally mean a transformation that produces a single integer from an array. In fact, you have already made many operations of data aggregation, for example, when we calculated the sum(), mean(), count(). In fact, these functions operate on a set of data and shall perform a calculation with a consistent result consisting of a single value. However, a more formal manner and the one with more control in data aggregation is that which includes the categorization of a set.

The categorization of a set of data carried out for grouping is often a critical stage in the process of data analysis. It is a process of transformation since after the division into different groups, you apply a function that converts or transforms the data in some way depending on the group they belong to. Very often the two phases of grouping and application of a function are performed in a single step.

Also for this part of the data analysis, pandas provides a tool very flexible and high performance: **GroupBy**.

Again, as in the case of join, those familiar with relational databases and the SQL language can find similarities. Nevertheless, languages such as SQL are quite limited when applied to operations on groups. In fact, given the flexibility of a programming language like Python, with all the libraries available, especially pandas, you can perform very complex operations on groups.

GroupBy

Now you will analyze in detail what the process of GroupBy is and how it works. Generally, it refers to its internal mechanism as a process called **SPLIT-APPLY-COMBINE**. So in its pattern of operation you may conceive this process as divided into three different phases expressed precisely by three operations:

- splitting: division into groups of datasets

- applying: application of a function on each group

- combining: combination of all the results obtained by different groups

Analyze better the three different phases (see Figure 6-1). In the first phase, that of splitting, the data contained within a data structure, such as a Series or a DataFrame, are divided into several groups, according to a given criterion, which is often linked to indexes or just certain values in a column. In the jargon of SQL, values contained in this column are reported as keys. Furthermore, if you are working with two-dimensional objects such as the DataFrame, the grouping criterion may be applied both to the line (axis = 0) for that column (axis = 1).

The second phase, that of applying, consists in applying a function, or better a calculation expressed precisely by a function, which will produce a new and single value, specific to that group.

The last phase, that of combining, will collect all the results obtained from each group and combine them together to form a new object.
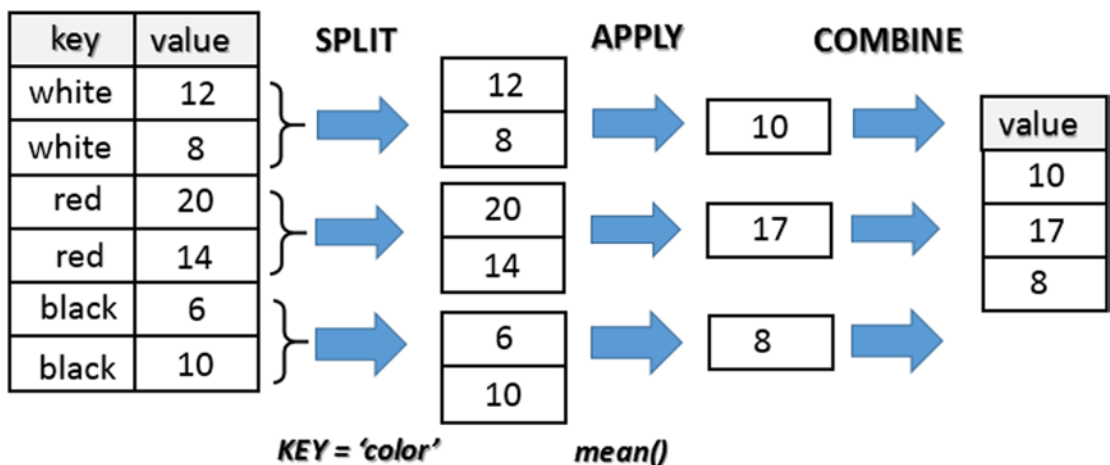
Figure 6-1. *The Split-Apply-Combine mechanism*

A Practical Example

You have just seen that the process of data aggregation in pandas is divided into various phases calls precisely split-apply-combine. With these pandas are not expressed explicitly with the functions as you would have expected, but by a **groupby()** function that generates an **GroupBy** object then that is the core of the whole process.

But to understand this mechanism, you must switch to a practical example. So, first, define a DataFrame containing both numeric and string values.

```
>>> frame = pd.DataFrame({ 'color': ['white','red','green','red','green'],
...                        'object': ['pen','pencil','pencil','ashtray','pen'],
...                        'price1' : [5.56,4.20,1.30,0.56,2.75],
...                        'price2' : [4.75,4.12,1.60,0.75,3.15]})
>>> frame
    color   object  price1  price2
0   white      pen    5.56    4.75
1     red   pencil    4.20    4.12
2   green   pencil    1.30    1.60
3     red  ashtray    0.56    0.75
4   green      pen    2.75    3.15
```

Suppose you want to calculate the average **price1** column using group labels listed in the column color. There are several ways to do this. You can for example access the **price1** column and call the **groupby()** function with the column color.

```
>>> group = frame['price1'].groupby(frame['color'])
>>> group
<pandas.core.groupby.SeriesGroupBy object at 0x00000000098A2A20>
```

The object that we got is a **GroupBy** object. In the operation that you just did there was not really any calculation; there was just a collection of all the information needed to calculate to be executed. What you have done is in fact a process of grouping, in which all rows having the same value of color are grouped into a single item.

To analyze in detail how the division into groups of rows of DataFrame was made, you call the attribute groups GroupBy object.

```
>>> group.groups
{'white': [0L], 'green': [2L, 4L], 'red': [1L, 3L]}
```

As you can see, each group is listed explicitly specifying the rows of the data frame assigned to each of them. Now it is sufficient to apply the operation on the group to obtain the results for each individual group.

```
>>> group.mean()
color
green    2.025
red      2.380
white    5.560
Name: price1, dtype: float64
```

```
>>> group.sum()
color
green    4.05
red      4.76
white    5.56
Name: price1, dtype: float64
```

Hierarchical Grouping

You have seen how to group the data according to the values of a column as a key choice. The same thing can be extended to multiple columns, i.e., make a grouping of multiple keys hierarchical.

```
>>> ggroup = frame['price1'].groupby([frame['color'],frame['object']])
>>> ggroup.groups
{('red', 'ashtray'): [3L], ('red', 'pencil'): [1L], ('green', 'pen'): [4L], ('green', '
], ('white', 'pen'): [0L]}

>>> ggroup.sum()
color  object
green  pen        2.75
       pencil     1.30
red    ashtray    0.56
       pencil     4.20
white  pen        5.56
Name: price1, dtype: float64
```

So far you have applied the grouping to a single column of data, but in reality it can be extended to multiple columns or the entire data frame. Also if you do not need to reuse the object GroupBy several times, it is convenient to combine in a single passing all of the grouping and calculation to be done, without defining any intermediate variable.

```
>>> frame[['price1','price2']].groupby(frame['color']).mean()
       price1  price2
color
green   2.025   2.375
red     2.380   2.435
white   5.560   4.750
>>> frame.groupby(frame['color']).mean()
       price1  price2
color
green   2.025   2.375
red     2.380   2.435
white   5.560   4.750
```

Group Iteration

The **GroupBy** object supports the operation of an iteration for generating a sequence of 2-tuples containing the name of the group together with the data portion.

```
>>> for name, group in frame.groupby('color'):
...     print name
...     print group
...
green
   color  object  price1  price2
2  green  pencil    1.30    1.60
4  green     pen    2.75    3.15
red
  color   object  price1  price2
1   red   pencil    4.20    4.12
3   red  ashtray    0.56    0.75
white
   color object  price1  price2
0  white    pen    5.56    4.75
```

In the example you have just seen, you only applied the print variable for illustration. In fact, you replace the printing operation of a variable with the function to be applied on it.

Chain of Transformations

From these examples you have seen that for each grouping, when subjected to some function calculation or other operations in general, regardless of how it was obtained and the selection criteria, the result will be a data structure Series (if we selected a single column data) or DataFrame, which then retains the index system and the name of the columns.

```
>>> result1 = frame['price1'].groupby(frame['color']).mean()
>>> type(result1)
<class 'pandas.core.series.Series'>
>>> result2 = frame.groupby(frame['color']).mean()
>>> type(result2)
<class 'pandas.core.frame.DataFrame'>
```

So it is possible to select a single column at any point in the various phases of this process. Here are three cases in which the selection of a single column in three different stages of the process applies. This example illustrates the great flexibility of this system of grouping provided by pandas.

```
>>> frame['price1'].groupby(frame['color']).mean()
color
green    2.025
red      2.380
white    5.560
Name: price1, dtype: float64
>>> frame.groupby(frame['color'])['price1'].mean()
color
```

```
green    2.025
red      2.380
white    5.560
Name: price1, dtype: float64
>>> (frame.groupby(frame['color']).mean())['price1']
color
green    2.025
red      2.380
white    5.560
Name: price1, dtype: float64
```

In addition, after an operation of aggregation the names of some columns may not be very meaningful in certain cases. In fact it is often useful to add a prefix to the column name that describes the type of business combination. Adding a prefix, instead of completely replacing the name, is very useful for keeping track of the source data from which they are derived aggregate values. This is important if you apply a process of transformation chain (a series of data frame is generated from each other) in which it is important to somehow keep some reference with the source data.

```
>>> means = frame.groupby('color').mean().add_prefix('mean_')
>>> means
       mean_price1  mean_price2
color
green        2.025        2.375
red          2.380        2.435
white        5.560        4.750
```

Functions on Groups

Although many methods have not been implemented specifically for use with GroupBy, they actually work correctly with data structures as the Series. You saw in the previous section how easy it is to get the Series by a GroupBy object, specifying the name of the column and then by applying the method to make the calculation. For example, you can use the calculation of quantiles with the **quantiles()** function.

```
>>> group = frame.groupby('color')
>>> group['price1'].quantile(0.6)
color
green    2.170
red      2.744
white    5.560
Name: price1, dtype: float64
```

You can also define their own aggregation functions. Define the function separately and then you pass as an argument to the **mark()** function. For example, you could calculate the range of the values of each group.

```
>>> def range(series):
...     return series.max() - series.min()
...
>>> group['price1'].agg(range)
color
green    1.45
```

```
red      3.64
white    0.00
Name: price1, dtype: float64
```

The agg() function() allows you to use aggregate functions on an entire DataFrame.

```
>>> group.agg(range)
       price1  price2
color
green    1.45    1.55
red      3.64    3.37
white    0.00    0.00
```

Also you can use more aggregate functions at the same time always with the **mark()** function passing an array containing the list of operations to be done, which will become the new columns.

```
>>> group['price1'].agg(['mean','std',range])
        mean        std  range
color
green  2.025  1.025305   1.45
red    2.380  2.573869   3.64
white  5.560       NaN   0.00
```

Advanced Data Aggregation

In this section you will be introduced to **transform()** and **apply()** functions, which will allow you to perform many kinds of group operations, some very complex.

Now suppose we want to bring together in the same DataFrame the following: (i) the DataFrame of origin (the one containing the data) and (ii) that obtained by the calculation of group aggregation, for example, the sum.

```
>>> frame = pd.DataFrame({ 'color':['white','red','green','red','green'],
...                        'price1':[5.56,4.20,1.30,0.56,2.75],
...                        'price2':[4.75,4.12,1.60,0.75,3.15]})
>>> frame
   color  price1  price2
0  white    5.56    4.75
1    red    4.20    4.12
2  green    1.30    1.60
3    red    0.56    0.75
4  green    2.75    3.15
>>> sums = frame.groupby('color').sum().add_prefix('tot_')
>>> sums
       tot_price1  tot_price2
color
green        4.05        4.75
red          4.76        4.87
white        5.56        4.75
```

```
>>> merge(frame,sums,left_on='color',right_index=True)
   color  price1  price2  tot_price1  tot_price2
0  white    5.56    4.75        5.56        4.75
1    red    4.20    4.12        4.76        4.87
3    red    0.56    0.75        4.76        4.87
2  green    1.30    1.60        4.05        4.75
4  green    2.75    3.15        4.05        4.75
```

So thanks to the **merge()**, you managed to add the results of a calculation of aggregation in each line of the data frame to start. But actually there is another way to do this type of operation. That is by using the **transform()**. This function performs the calculation of aggregation as you have seen before, but at the same time shows the values calculated based on the key value on each line of the data frame to start.

```
>>> frame.groupby('color').transform(np.sum).add_prefix('tot_')
   tot_price1  tot_price2
0        5.56        4.75
1        4.76        4.87
2        4.05        4.75
3        4.76        4.87
4        4.05        4.75
```

As you can see the transform() method is a more specialized function that has very specific requirements: the function passed as an argument must produce a single scalar value (aggregation) to be broadcasted.

The method to cover more general GroupBy is applicable to apply(). This method applies in its entirety the scheme split-apply-combine. In fact, this function divides the object into parts in order to be manipulated, invokes the passage of function on each piece, and then tries to chain together the various parts.

```
>>> frame = DataFrame( { 'color':['white','black','white','white','black','black'],
...                      'status':['up','up','down','down','down','up'],
...                      'value1':[12.33,14.55,22.34,27.84,23.40,18.33],
...                      'value2':[11.23,31.80,29.99,31.18,18.25,22.44]})
>>> frame
   color status  value1  value2
0  white     up   12.33   11.23
1  black     up   14.55   31.80
2  white   down   22.34   29.99
3  white   down   27.84   31.18
4  black   down   23.40   18.25

>>> frame.groupby(['color','status']).apply( lambda x: x.max())
              color status  value1  value2
color status
black down    black   down   23.40   18.25
      up      black     up   18.33   31.80
white down    white   down   27.84   31.18
      up      white     up   12.33   11.23
5  black        up   18.33   22.44
```

```
>>> frame.rename(index=reindex, columns=recolumn)
         color   object   value
first    white     ball    5.56
second     red      mug    4.20
third    green      pen    1.30
fourth   black   pencil    0.56
fifth   yellow  ashtray    2.75
>>> temp = date_range('1/1/2015', periods=10, freq= 'H')
>>> temp
<class 'pandas.tseries.index.DatetimeIndex'>
[2015-01-01 00:00:00, ..., 2015-01-01 09:00:00]
Length: 10, Freq: H, Timezone: None
>>> timeseries = Series(np.random.rand(10), index=temp)
>>> timeseries
2015-01-01 00:00:00    0.368960
2015-01-01 01:00:00    0.486875
2015-01-01 02:00:00    0.074269
2015-01-01 03:00:00    0.694613
2015-01-01 04:00:00    0.936190
2015-01-01 05:00:00    0.903345
2015-01-01 06:00:00    0.790933
2015-01-01 07:00:00    0.128697
2015-01-01 08:00:00    0.515943
2015-01-01 09:00:00    0.227647
Freq: H, dtype: float64

>>> timetable = DataFrame( {'date': temp, 'value1' : np.random.rand(10),
...                                       'value2' : np.random.rand(10)})
>>> timetable
                 date    value1    value2
0 2015-01-01 00:00:00  0.545737  0.772712
1 2015-01-01 01:00:00  0.236035  0.082847
2 2015-01-01 02:00:00  0.248293  0.938431
3 2015-01-01 03:00:00  0.888109  0.605302
4 2015-01-01 04:00:00  0.632222  0.080418
5 2015-01-01 05:00:00  0.249867  0.235366
6 2015-01-01 06:00:00  0.993940  0.125965
7 2015-01-01 07:00:00  0.154491  0.641867
8 2015-01-01 08:00:00  0.856238  0.521911
9 2015-01-01 09:00:00  0.307773  0.332822
```

We add to the DataFrame preceding a column that represents a set of text values that we will use as key values.

```
>>> timetable['cat'] = ['up','down','left','left','up','up','down','right','right','up']
>>> timetable
                date     value1    value2    cat
0 2015-01-01 00:00:00  0.545737  0.772712    up
1 2015-01-01 01:00:00  0.236035  0.082847  down
2 2015-01-01 02:00:00  0.248293  0.938431  left
3 2015-01-01 03:00:00  0.888109  0.605302  left
4 2015-01-01 04:00:00  0.632222  0.080418    up
5 2015-01-01 05:00:00  0.249867  0.235366    up
6 2015-01-01 06:00:00  0.993940  0.125965  down
7 2015-01-01 07:00:00  0.154491  0.641867 right
8 2015-01-01 08:00:00  0.856238  0.521911 right
9 2015-01-01 09:00:00  0.307773  0.332822    up
```

The example shown above, however, has duplicate key values.

Conclusions

In this chapter you saw the three basic parts which divide the data manipulation: preparation, processing, and data aggregation. Thanks to a series of examples you've got to know a set of library functions that allow pandas to perform these operations.

You saw how to apply these functions on simple data structures so that you can become familiar with how they work and understand its applicability to more complex cases.

Eventually you get knowledge of all the tools necessary to prepare a data set for the next phase of data analysis: data visualization.

In the next chapter, you will be presented with the Python library Matplotlib, which can convert the data structures in any chart.

■ ■ ■

Data Visualization with matplotlib

After discussing in the previous chapters about Python libraries that were responsible for data processing, now it is time for you to see a library that takes care of their visualization. This library is matplotlib.

Data visualization is an aspect too often underestimated in data analysis, but it is actually a very important factor because an incorrect or inefficient data representation can ruin an otherwise-excellent analysis. In this chapter you will discover the various aspects of the matplotlib library, how it is structured, and how to maximize the potential that it offers.

The matplotlib Library

Matplotlib is a Python library specializing in the development of two-dimensional charts (including 3D charts); in recent years, it has been widespread in scientific and engineering circles (http://matplolib.org).

Among all the features that have made it the most used tool in the graphical representation of data, there are a few that stand out:

- extreme simplicity in its use

- gradual development and interactive data visualization

- expressions and texts in LaTeX

- greater control over graphic elements

- export in many formats such as PNG, PDF, SVG, and EPS.

Matplotlib is designed to reproduce as much as possible an environment similar to Matlab in terms of both graphic view and syntactic form. This approach has proved successful as it has been able to exploit the experience of software (Matlab) that has been on the market for several years and is now widespread in all professional technical-scientific circles. So not only is matplotlib based on a scheme known and quite familiar to most experts in the field, but also it also exploits those optimizations that over the years have led to a **deducibility and simplicity in its use** which makes this library also an excellent choice for those approaching data visualization for the first time, especially those without any experience with applications such as Matlab or similar.

In addition to simplicity and deducibility, the matplotlib library has been able to inherit **interactivity** from Matlab as well. That is, the analyst is able to insert command after command to control the **gradual development of a graphical representation of data**. This mode is well suited to the more interactive approaches of Python as IPython QtConsole and IPython Notebook (see Chapter 2), thus providing an environment for data analysis that has little to envy from other tools such as Mathematica, IDL, or Matlab.

The genius of those who developed this beautiful library was to use and incorporate the good things currently available and in use in science. This is not only limited, as we have seen, to the operating mode of Matlab and similar, but also to models of textual formatting of scientific expressions and symbols represented by **LaTeX**. Because of its great capacity for display and presentation of scientific expressions, LaTeX has been an irreplaceable element in any scientific publication or documentation, where the need to visually represent expressions like integrals, summations, and derivatives is mandatory. Therefore matplotlib integrates this remarkable instrument in order to improve the representative capacity of charts.

In addition, you must not forget that matplotlib is not a separate application but a library of a programming language like Python. So it also takes full advantage of the potential that programming languages offer. Matplotlib looks like a graphics library that allows you to **programmatically manage the graphic elements** that make up a chart so that the graphical display can be controlled in its entirety. The ability to program the graphical representation allows management of the reproducibility of the data representation across multiple environments and especially when you make changes or when the data is updated.

Moreover, since matplotlib is a Python library, it allows you to exploit the full potential of other libraries available to any developer which implements with this language. In fact, with regard to data analysis, matplotlib normally cooperates with a set of other libraries such as NumPy and pandas, but many other libraries can be integrated without any problem.

Finally graphical representations obtained through encoding with this library can be exported in the most common graphic formats (such as **PNG** and **SVG**) and then be used in other applications, documentation, web pages, etc.

Installation

There are many options to install the matplotlib library. If you choose to use a distribution of packages like Anaconda or Enthought Canopy, the installation of the matplotlib package is very simple. For example, with the conda package manager, you have to enter

```
conda install matplotlib
```

If you want to directly install this package, the commands to insert vary depending on the operating system.

On Debian-Ubuntu Linux systems

```
sudo apt-get install python-matplotlib
```

On Fedora-Redhat Linux systems

```
sudo yum install python-matplotlib
```

On Windows or MacOS you should use **pip** for installing matplotlib.

IPython and IPython QtConsole

In order to get familiar with all the tools provided by the Python world I chose to use IPython both from a terminal and from the QtConsole. This is because IPython allows you to exploit the interactivity of its enhanced terminal and also, as you will see, IPython QtConsole also allows you to integrate graphics directly inside the console.

To run an IPython session simply run the following command:

```
ipython

Python 2.7.8 (default, Jul  2 2014, 15:12:11) [MSC v.1500 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 3.1.0 -- An enhanced Interactive Python.
?           -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help        -> Python's own help system.
object?    -> Details about 'object', use 'object??' for extra details.

In [1]:
```

Whereas if you want to run IPython QtConsole with the ability to display graphics within the line commands of the session:

```
ipython qtconsole --matplotlib inline
```

Immediately a window with a new open IPython session will appear on the screen as shown in Figure 7-1.
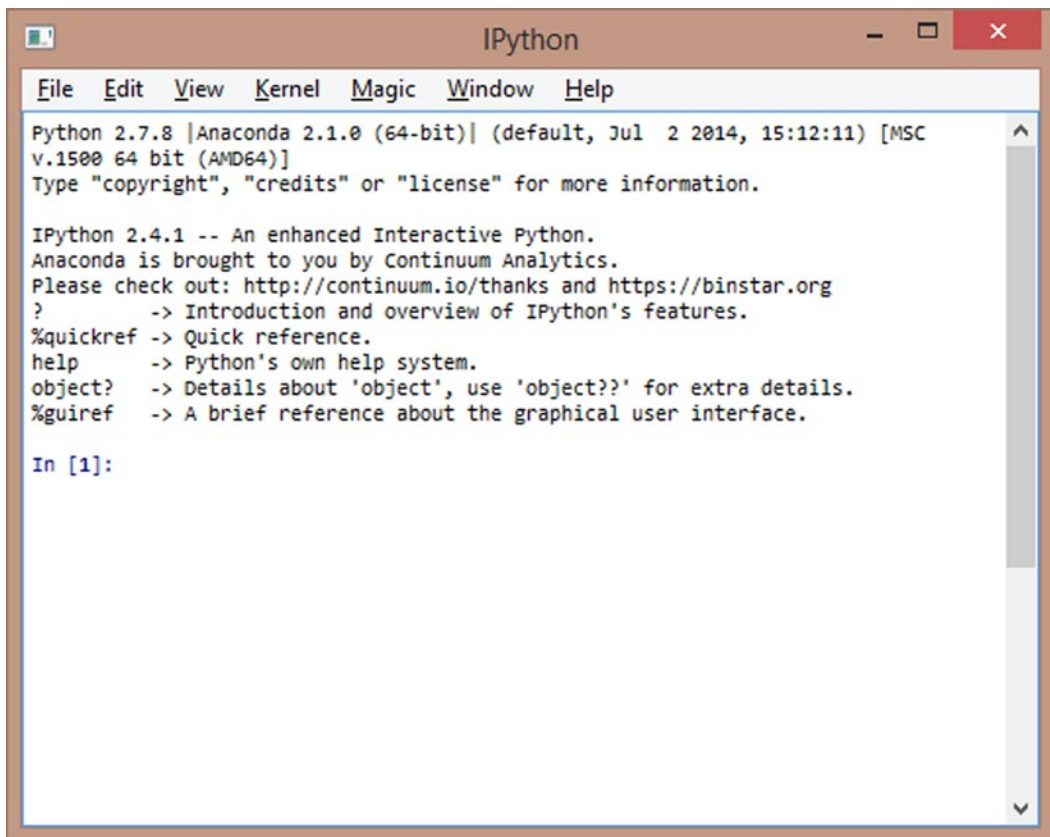
Figure 7-1. *The IPython QtConsole*

However, if you want to continue using a standard Python session you are free to do so. If you do not like working with IPython and want to continue to use Python from terminal all the examples in this chapter are still valid.

matplotlib Architecture

One of the key tasks that matplotlib must take is to provide a set of functions and tools that allow representation and manipulation of a **Figure** (the main object) along with all internal objects of which it is composed. However, matplotlib not only deals with graphics but also provides all the tools for the event handling and the ability to animate graphics. So, thanks to these additional features, matplotlib proves to be a tool capable of producing interactive charts based on the events triggered by pressing a key on the keyboard or mouse movement.

The architecture of matplotlib is logically structured in three layers placed at three different levels (see Figure 7-2). The communication is unidirectional, that is, each layer is able to communicate with the underlying layer, while the lower cannot do it with the top one.

The three layers are as follows:

- scripting

- artist

- backend
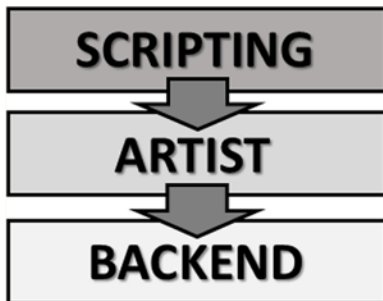
Figure 7-2. *The three layers of the matplotlib architecture*

Backend Layer

In the diagram of the matplotlib architecture the layer that works at the lowest level is the **Backend**. In this layer there are the matplotlib APIs, a set of classes that play the role of implementation of the graphic elements at a low level.

- **FigureCanvas** is the object that embodies the concept of drawing area.

- **Renderer** is the object that draws on FigureCanvas.

- **Event** is the object that handles user inputs (keyboard and mouse events).

Artist Layer

As an intermediate layer we have a layer called **Artist**. All that you can look inside a figure is an Artist object, that is, all the elements that go to make up a chart such as the title, axis labels, markers, etc., are instances of the Artist object. Each of these instances within the chart plays its role within a hierarchical structure (as shown in Figure 7-3).
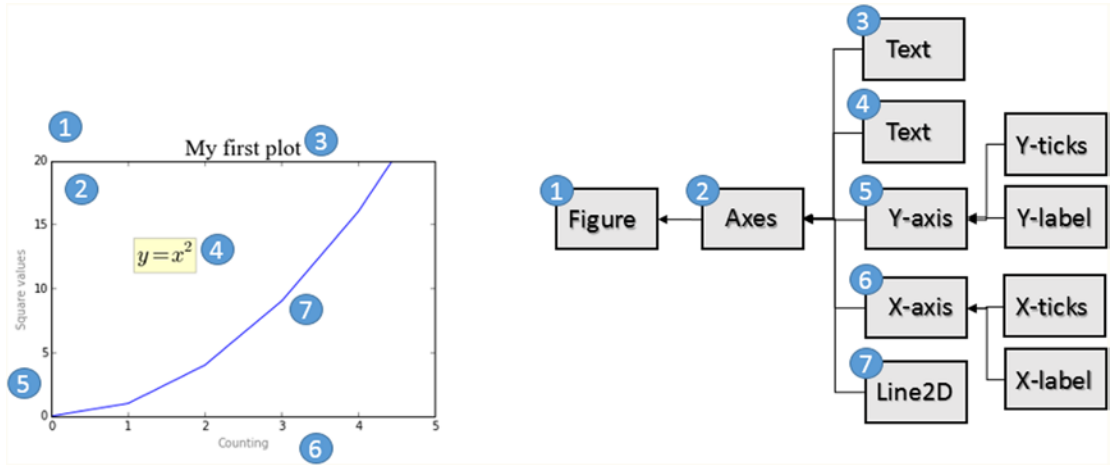
Figure 7-3. *Each element of a chart corresponds to an instance of Artist structured in a hierarchy*

There are two Artist classes: primitive and composite.

The **primitive artists** are individual objects that constitute the basic elements to form a graphical representation in a plot, for example a Line2D, or as a geometric figure such as a Rectangle or Circle, or even pieces of text.

The **composite artists** are those graphic elements present in a chart which are composed of several base elements, namely, the primitive artists. Composite artists are for example the Axis, Ticks, Axes, and figures (see Figure 7-4).

Generally, working at this level you will have to deal often with objects in higher hierarchy as Figure, Axes, and Axis. So it is important to fully understand what these objects are and what role they play within the graphical representation. In Figure 7-4 they are presented the three main Artist objects (composite artists) that are generally used in all implementations performed at this level.
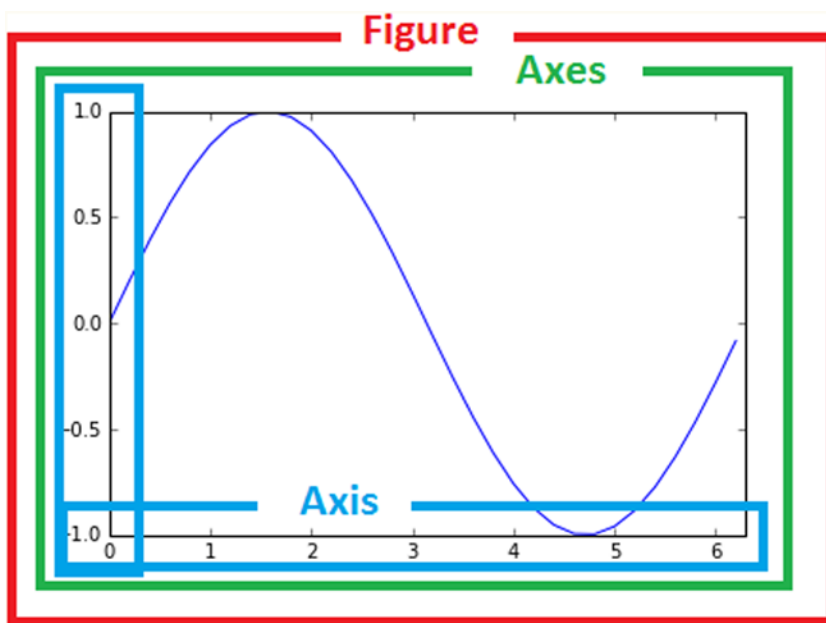
Figure 7-4. *The three main artist objects in the hierarchy of the Artist Layer*

Figure is the object with the highest level in the hierarchy. It corresponds to the entire graphical representation and generally can contain many Axes.

Axes is generally what you mean as plot or chart. Each Axes object belongs to only one Figure, and is characterized by two Artist Axis (three in the three-dimensional case). Also other objects such as the title, the x label and y label belong to this composite artist.

Axis objects that take into account the numerical values to be represented on Axes, define the limits, and manage the ticks (the mark on the axes) and tick labels (the label text represented on each tick). The position of the tick is adjusted by an object called **Locator** while formatting tick label is regulated by an object called **Formatter**.

Scripting Layer (pyplot)

Artist classes and their related functions (the matplotlib API) are tools particularly suitable for all developers, especially for those who work on web application servers or developing the GUI. But for purposes of calculation, and in particular for the analysis and visualization of data, the scripting layer is the one that suits best. This layer consists of an interface called **pyplot**.

pylab and pyplot

In general there is talk of **pylab** and **pyplot**. But what is the difference between these two packages? Pylab is a module that is installed along with matplotlib, while pyplot is an internal module of matplotlib. Often you will find references to one or the other approach.

```
from pylab import *
```

and

```
import matplotlib.pyplot as plt
import numpy as np
```

Pylab combines the functionality of pyplot with the capabilities of NumPy in a single namespace, and therefore you do not need to import NumPy separately. Furthermore, if you import pylab, pyplot, and NumPy functions can be called directly without any reference to a module (namespace), making the environment more similar to Matlab.

```
plot(x,y)
array([1,2,3,4])
```

instead of

```
plt.plot()
np.array([1,2,3,4]
```

The **pyplot** package provides the classic Python interface for programming the matplotlib library, has its own namespace, and requires the import of the NumPy package separately. This approach is the one chosen for this book, it is the main topic of this chapter, and it will be used for the rest of the book. In fact this choice is shared and approved by most Python developers.

pyplot

The pyplot module is a collection of command-style functions that allow you to use matplotlib so much like Matlab. Each pyplot function will operate or make some changes to the Figure object, for example, the creation of the Figure itself, the creation of a plotting area, representation of a line, decoration of the plot with a label, etc.

Pyplot also is **stateful**, that is it tracks the status of the current figure and its plotting area. The functions called act on the current figure.

A Simple Interactive Chart

To get familiar with the matplotlib library and in a particular way with Pyplot, you will start creating a simple interactive chart. Using matplotlib this operation is very simple; in fact, you can achieve it using only three lines of code.

But first you need to import the **pyplot** package and rename it as **plt**.

```
In [1]: import matplotlib.pyplot as plt
```

In Python programming language the constructors generally are not necessary; everything is already implicitly defined. In fact when you import the package, the **plt** object with all its graphics capabilities have already been instantiated and ready to use. In fact, you simply use the **plot()** function to pass the values to be plotted.

Thus, you can simply pass the values that you want to represent as a sequence of integers.

```
In [2]: plt.plot([1,2,3,4])
Out[2]: [<matplotlib.lines.Line2D at 0xa3eb438>]
```

As you can see a Line2D object has been generated. The object is a line that represents the linear trend of the points included in the chart.

Now it is all set. You just have to give the command to show the plot using the **show()** function.

```
In [3]: plt.show()
```

The result will be the one shown in Figure 7-5. It looks just a window, the **plotting window,** with a toolbar and the plot represented within it, just as with Matlab.
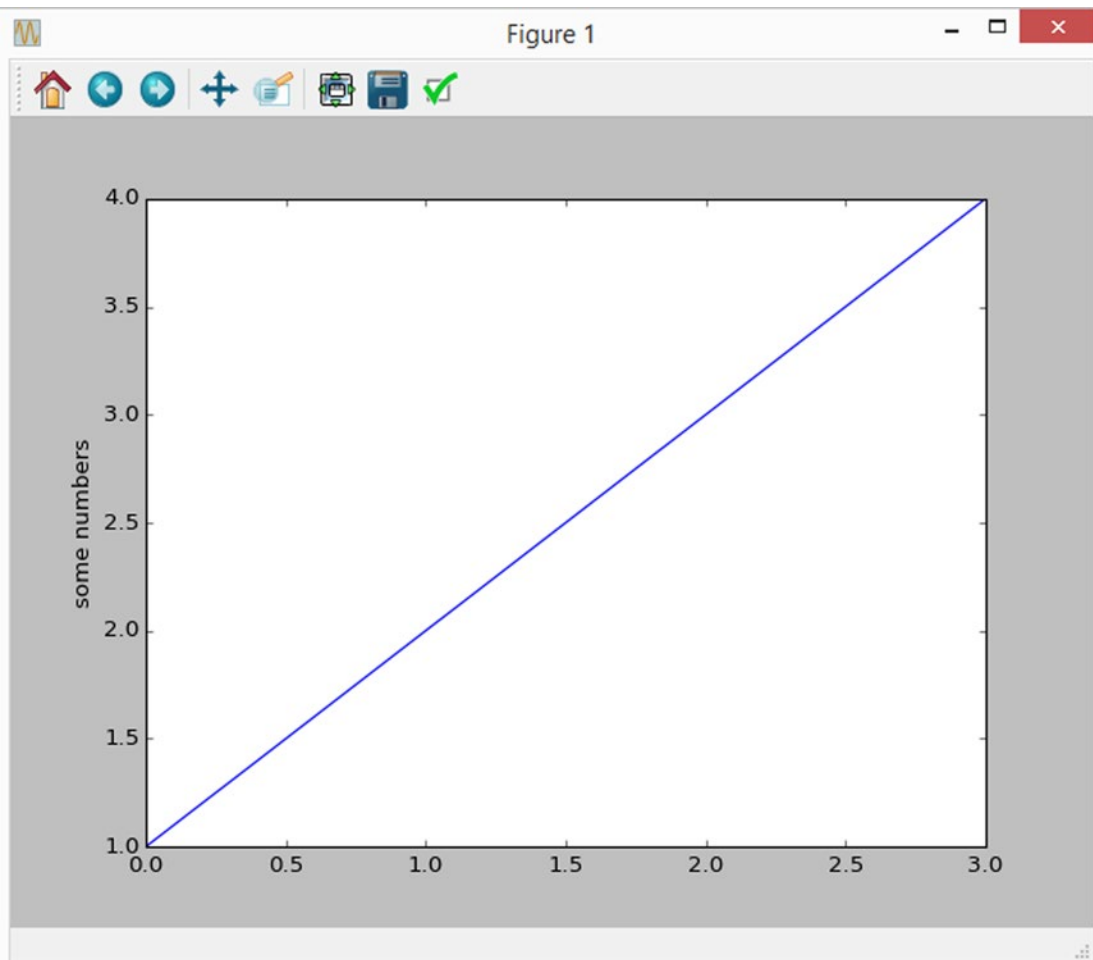
Figure 7-5. *The plotting window*

THE PLOTTING WINDOW

The plotting window is characterized by a toolbar at the top in which there are a series of buttons.

- 🏠 Reset the original view

- ⬅ ➡ Go to the previous/next view

- ✛ Pan axes with left mouse, zoom with right

- ✎ Zoom to rectangle

- 🖳 Configure subplots

- 💾 Save/Export the figure

- ✅ Edit curve lines and axes parameters

The code entered into the IPython console corresponds on the Python console to the following series of commands:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot([1,2,3,4])
[<matplotlib.lines.Line2D object at 0x0000000007DABFD0>]
>>> plt.show()
```

If you are using the IPython QtConsole you have noticed that after calling the plot() function the chart is displayed directly without explicitly invoking the show() function (see Figure 7-6).
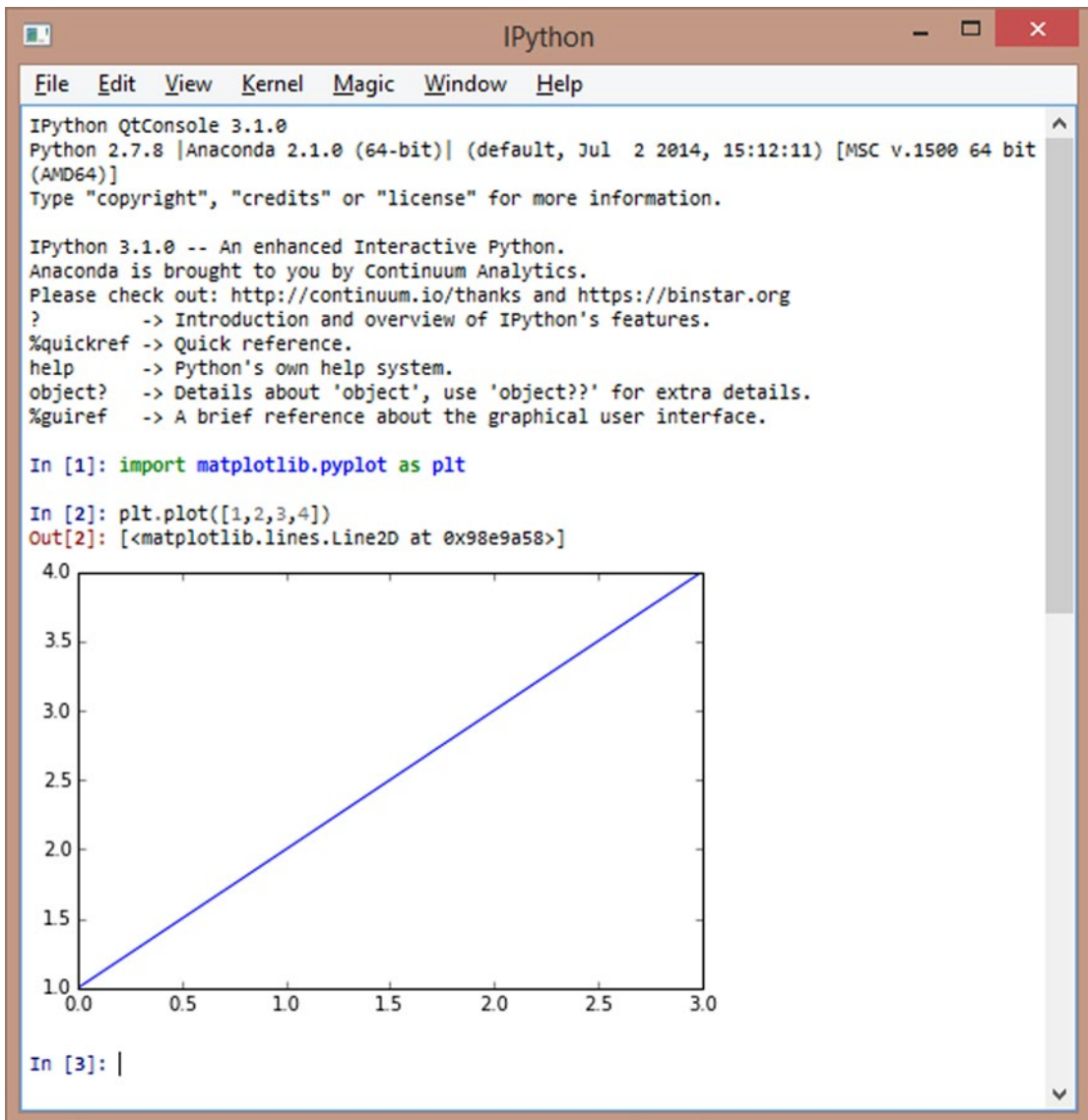
Figure 7-6. *The QtConsole shows the chart directly as output*

If you pass only a list of numbers or an array to the **plt.plot()** function, matplotlib assumes it is the sequence of y values of the chart, and it associates them to the natural sequence of values x: 0,1,2,3, ...

Generally a plot represents value pairs (x, y), so if you want to define a chart correctly, you must define two arrays, the first containing the values on the x axis and the second containing the values on the y axis. Moreover, the plot() function can accept a third argument which describes the specifics of how you want the point to be represented on the chart.

Set the Properties of the Plot

As you can see in Figure 7-6, the points were represented by a blue line. In fact, if you do not specify otherwise, the plot is represented taking into account a default configuration of the **plt.plot()** function:

- the size of the axes matches perfectly with the range of the input data

- there is neither title nor axis labels

- there is no legend

- A blue line connecting the points is drawn

Therefore you need to change this representation to have a real plot in which each pair of values (x, y) is represented by a red dot (see Figure 7-7).

If you're working on IPython, close the window to get back again to the active prompt for entering new commands. Then you have to call back the *show()* function to observe the changes made to the plot.

```
In [4]: plt.plot([1,2,3,4],[1,4,9,16],'ro')
Out[4]: [<matplotlib.lines.Line2D at 0x93e6898>]

In [4]: plt.show()
```

Instead, if you're working on IPython QtConsole you do not have to do any of these things but you can directly enter a new command (interactivity).

```
In [3]: plt.plot([1,2,3,4],[1,4,9,16],'ro')
```
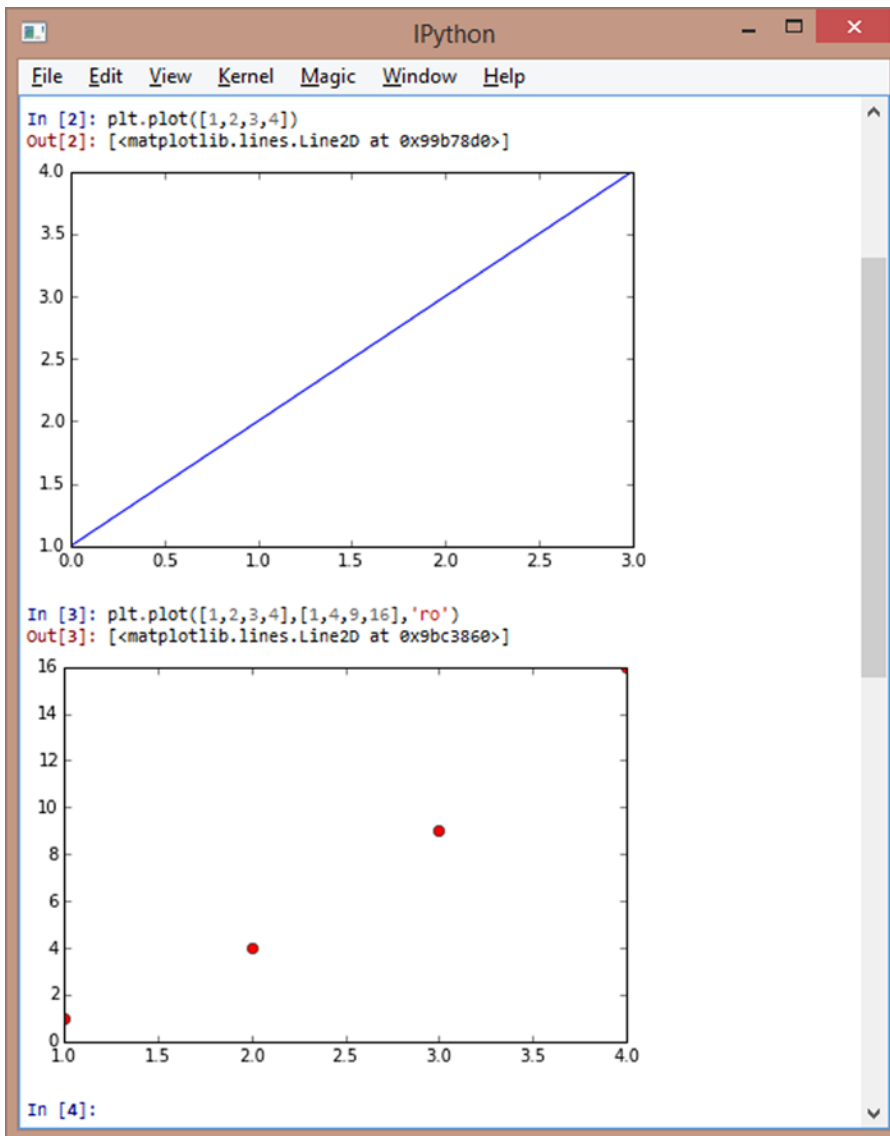
Figure 7-7. *The pairs of (x,y) values are represented in the plot by red circles*

■ **Note** At this point in the book, you have already a very clear idea about the difference between the various environments. To avoid confusion from this point I will consider IPython QtConsole as development environment.

You can define the range both on the x axis and on the y axis by defining the details of a list *[xmin, xmax, ymin, ymax]* and then passing it as an argument to the **axis()** function.

■ **Note** In IPython QtConsole, to generate a chart it is sometimes necessary to entr more rows of commands. To avoid generating a chart every time you press Enter (start a new line) along with losing the setting previously specified, you have to press Ctrl + Enter. When you want to finally generate the chart just press Enter twice.

The properties that you can set are several, and one of which is, for example, the title that can be entered through the **title()** function.

```
In [4]: plt.axis([0,5,0,20])
   ...: plt.title('My first plot')
   ...: plt.plot([1,2,3,4],[1,4,9,16],'ro')
Out[4]: [<matplotlib.lines.Line2D at 0x97f1c18>]
```

In Figure 7-8 you can see how the new settings made the plot more readable. In fact, the end points of the data set are now represented within the plot rather than at the edges. Also the title of the plot is now visible at the top.
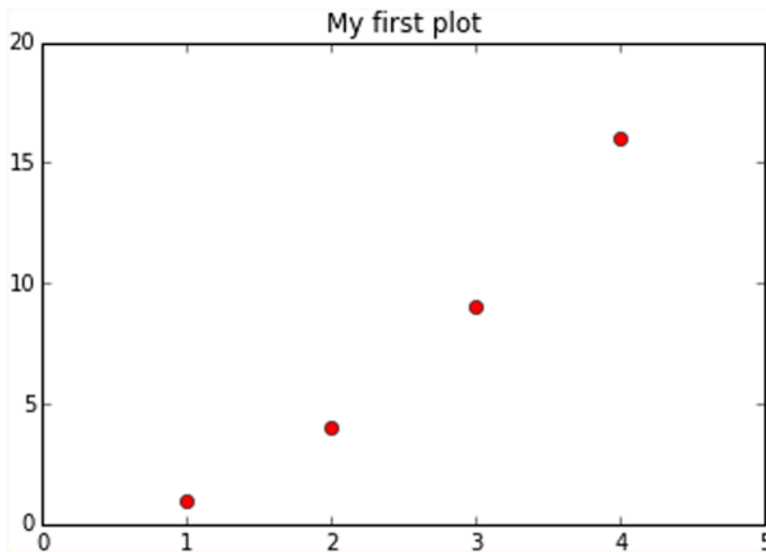
Figure 7-8. *The plot after the properties have been set*

matplotlib and NumPy

Even the Matplot library, despite being a fully graphical library, has its foundation the NumPy library. In fact, you have seen so far how to pass lists as arguments, both to represent the data and to set the extremes of the axes. Actually, these lists have been converted internally in NumPy arrays.

Therefore, you can directly enter NumPy arrays as input data. This array of data, which have been processed by pandas, can be directly used with matplotlib without further processing.

As an example now you see how it is possible to plot three different trends in the same plot (Figure 7-9). You can choose for this example the **sin()** function belonging to the **math** module. So you will need to import it. To generate points following a sinusoidal trend you will use the library NumPy. Generate a series of points on the x axis using the **arange()** function, while for the values on the y axis you will use the **map()** function to apply the **sin()** function on all the items of the array (without using a *for* loop).

```
In [5]: import math
In [6]: import numpy as np
In [7]: t = np.arange(0,2.5,0.1)
   ...: y1 = map(math.sin,math.pi*t)
   ...: y2 = map(math.sin,math.pi*t+math.pi/2)
   ...: y3 = map(math.sin,math.pi*t-math.pi/2)
In [8]: plt.plot(t,y1,'b*',t,y2,'g^',t,y3,'ys')
Out[8]:
[<matplotlib.lines.Line2D at 0xcbd2e48>,
 <matplotlib.lines.Line2D at 0xcbe10b8>,
 <matplotlib.lines.Line2D at 0xcbe15c0>]
```

■ **Note** If you are not using the IPython QtConsole set with matplotlib inline or you are implementing this code on a simple Python session, please insert the command **plt.show()** to the end of the code for obtaining the chart as shown in Figure 7-10.
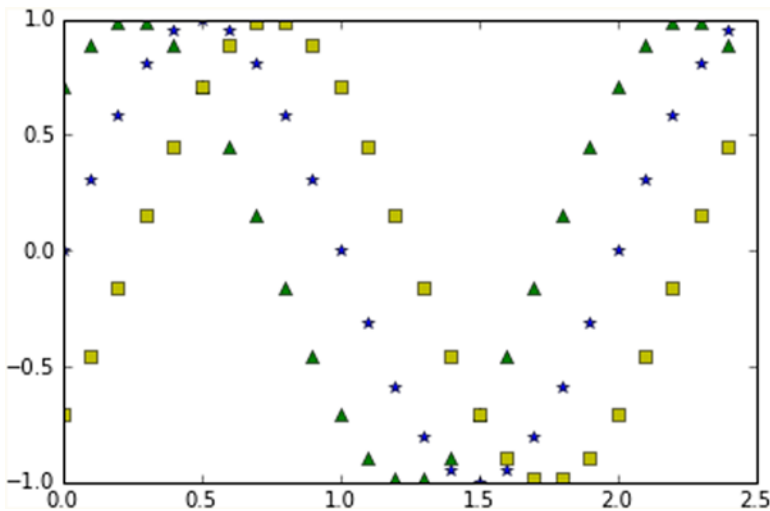
Figure 7-9. *Three sinusoidal trends phase-shifted by π / 4 represented by markers*

As you can see in Figure 7-9 the plot represents the three different temporal trends with three different colors and markers. In these cases, when the trend of a function is so obvious, the plot is perhaps not the most appropriate representation, but it is better to use the lines (see Figure 7-10). To differentiate the three trends as well as using different colors, you can use the pattern composed of different combinations of dots and dashes (- and .).

```
In [9]: plt.plot(t,y1,'b--',t,y2,'g',t,y3,'r-.')
Out[9]:
[<matplotlib.lines.Line2D at 0xd1eb550>,
 <matplotlib.lines.Line2D at 0xd1eb780>,
 <matplotlib.lines.Line2D at 0xd1ebd68>]
```

■ **Note**   If you are not using the IPython QtConsole set with matplotlib inline or you are implementing this code on a simple Python session, please insert the command **plt.show()** to the end of the code for obtaining the chart as shown in Figure 7-10.
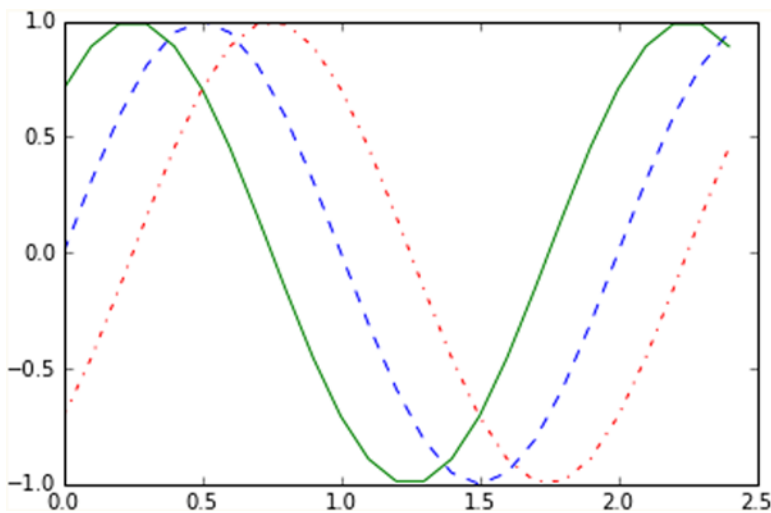
Figure 7-10. *This chart represents the three sinusoidal patterns with colored lines*

Using the kwargs

The objects that make up a chart have many attributes that characterize them. These attributes are all default values, but can be set through the use of **keyword args**, often referred as **kwargs**.

These keywords are passed as arguments to functions called. In reference documentation of the various functions of the matplotlib library, you will always find them referred to as **kwargs** in the last position. For example the plot() function that you are using in our examples is referred to in the following way.

```
matplotlib.pyplot.plot(*args, **kwargs)
```

For a practical example, the thickness of a line can be changed then if you set the **linewidth** keyword (see Figure 7-11).

```
In [10]: plt.plot([1,2,4,2,1,0,1,2,1,4],linewidth=2.0)
Out[10]: [<matplotlib.lines.Line2D at 0xc909da0>]
```
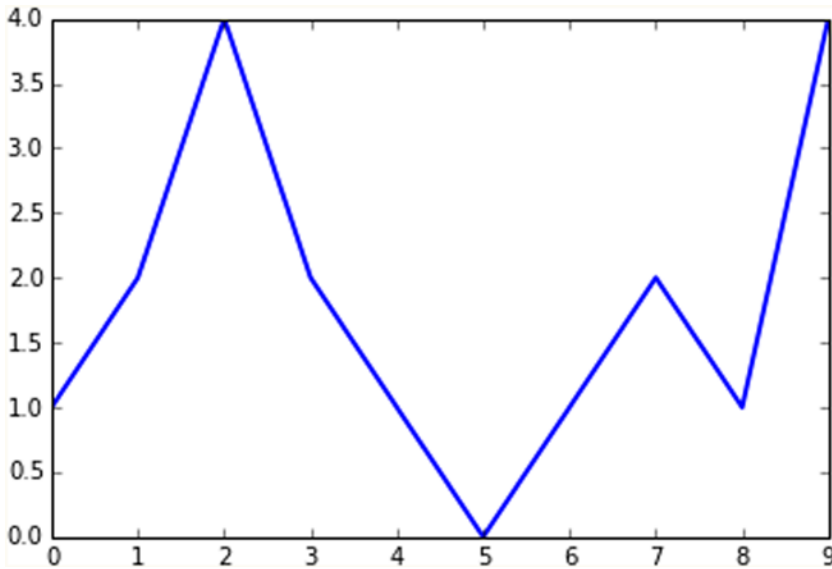

Figure 7-11. *The thickness of a line can be set directly from the plot() function*

Working with Multiple Figures and Axes

So far you have seen how all pyplot commands are routed to the display of a single figure. Actually, matplotlib allows management of multiple figures simultaneously, and within each figure, it offers the ability to view different plots defined as subplots.

So when you are working with pyplot, you must always keep in mind the concept of current Figure and current Axes (that is, the plot shown within the figure).

Now you will see an example where two subplots are represented in a single figure. The **subplot()** function, in addition to subdividing the figure in different drawing areas, is used at the same time in order to focus the commands on a specific subplot.

The argument passed to the subplot() function sets the mode of subdivision and which is the current subplot. The current subplot will be the only figure which will be affected by the commands. The argument of the subplot() function is composed of three integers. The first number defines how many parts the figure is split into vertically. The second number defines how many parts the figure is divided into horizontally. The third issue selects which is the current subplot on which you can direct commands.

Now you will display two sinusoidal trends (sine and cosine) and the best way to do that is to divide the canvas vertically in two horizontal subplots (as shown in Figure 7-12). So the numbers to pass as an argument are '211' and '212'.

```
In [11]: t = np.arange(0,5,0.1)
   ... : y1 = np.sin(2*np.pi*t)
   ... : y2 = np.sin(2*np.pi*t)
In [12]: plt.subplot(211)
   ...: plt.plot(t,y1,'b-.')
   ...: plt.subplot(212)
   ...: plt.plot(t,y2,'r--')
Out[12]: [<matplotlib.lines.Line2D at 0xd47f518>]
```
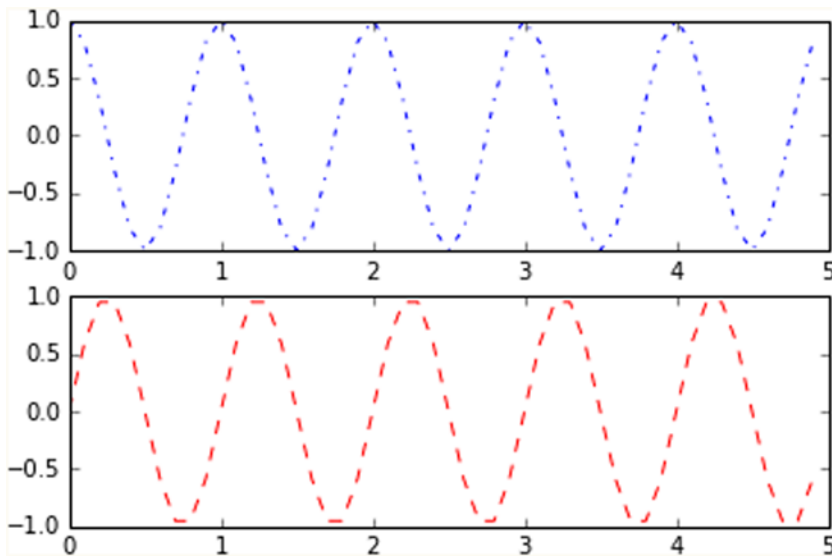

Figure 7-12. *The figure has been divided into two horizontal subplots*

Now you do the same thing by dividing the figure in two vertical subplots. The numbers to be passed as arguments to the *subplot()* function are '121' and '122' (as shown in Figure 7-13).

```
In [ ]: t = np.arange(0.,1.,0.05)
   ...: y1 = np.sin(2*np.pi*t)
   ...: y2 = np.cos(2*np.pi*t)
In [ ]: plt.subplot(121)
   ...: plt.plot(t,y1,'b-.')
   ...: plt.subplot(122)
   ...: plt.plot(t,y2,'r--')
Out[94]: [<matplotlib.lines.Line2D at 0xed0c208>]
```
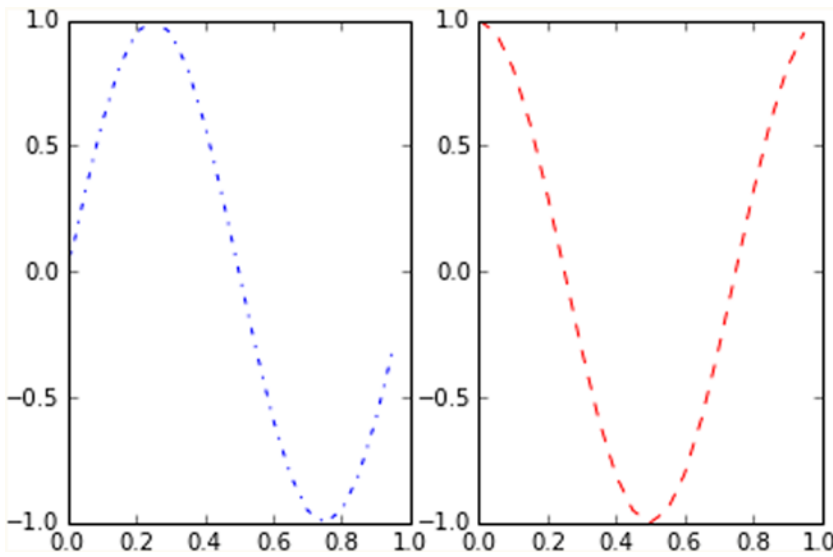
Figure 7-13. *The figure has been divided into two vertical subplots*

Adding Further Elements to the Chart

In order to make a chart more informative, many times it is not enough to represent the data using lines or markers and assign the range of values using two axes. In fact, there are many other elements that can be added to a chart in order to enrich it with additional information.

In this section you will see how to add additional elements to the chart as text labels, a legend, and so on.

Adding Text

You've already seen how you can add the title to a chart with the *title()* function. Two other textual indications very important to be added to a chart are the **axis labels**. This is possible through the use of two other specific functions such as **xlabel()** and **ylabel()**. These functions take as argument a string which will be the shown text.

■ **Note**   Command lines forming the code to represent your chart are growing in number. You do not need to rewrite all the commands each time, but using the arrow keys on the keyboard you can call up the list of commands previously passed and edit them by adding new rows (in the text are indicated in bold).

Now add two axis labels to the chart. They will describe which kind of value is assigned to each axis (as shown in Figure 7-14).

```
In [10]: plt.axis([0,5,0,20])
    ...: plt.title('My first plot')
    ...: plt.xlabel('Counting')
    ...: plt.ylabel('Square values')
    ...: plt.plot([1,2,3,4],[1,4,9,16],'ro')
Out[10]: [<matplotlib.lines.Line2D at 0x990f3c8>]
```

184
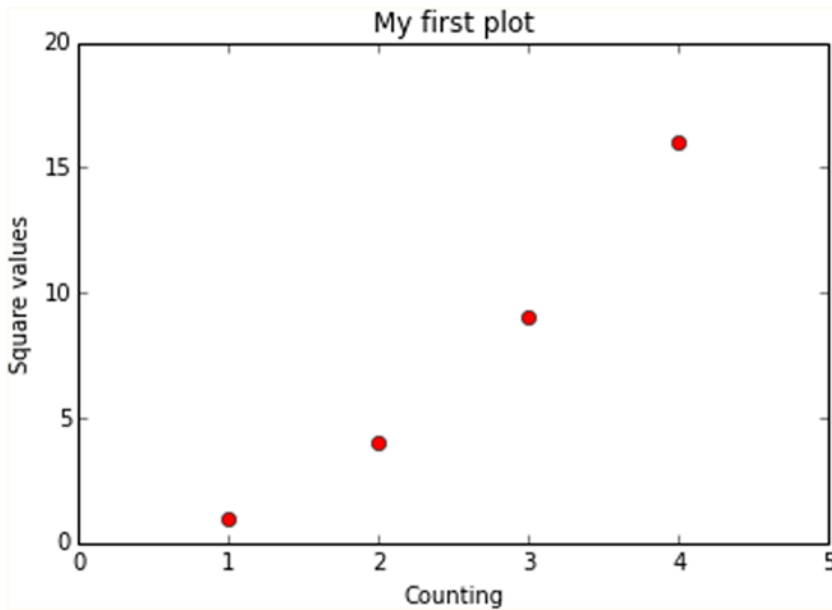
Figure 7-14. *A plot is more informative by adding axis labels*

Thanks to the keywords, you can change the characteristics of the text. For example, you can modify the title by changing the font and increasing the size of the characters. Also you can modify the color of the axis labels to bring out more the title of the plot (as shown in Figure 7-15).

```
In [ ]: plt.axis([0,5,0,20])
   ...: plt.title('My first plot',fontsize=20,fontname='Times New Roman')
   ...: plt.xlabel('Counting',color='gray')
   ...: plt.ylabel('Square values',color='gray')
   ...: plt.plot([1,2,3,4],[1,4,9,16],'ro')
Out[116]: [<matplotlib.lines.Line2D at 0x11f17470>]
```
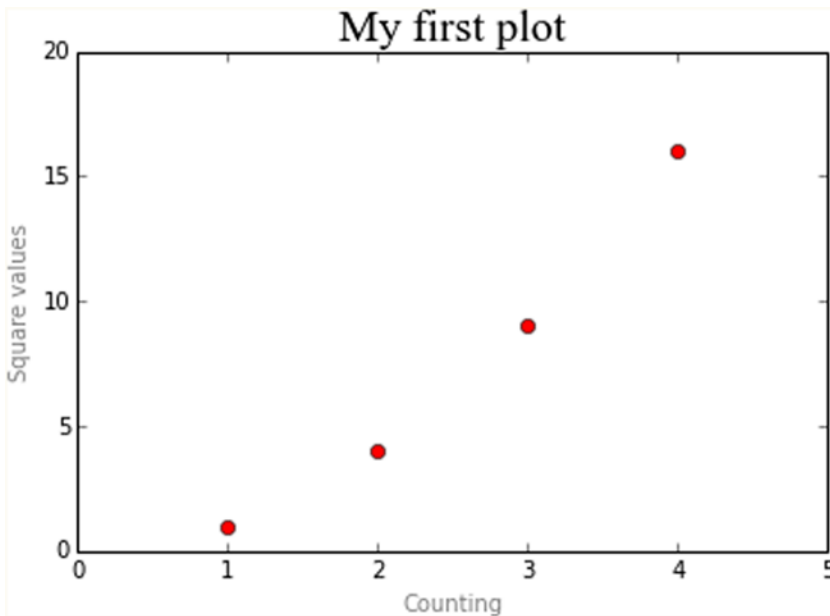
Figure 7-15. *The text can be modified by setting the keywords*

But matplotlib is not limited to this: pyplot allows you to add text to any position within a chart. This feature is performed by a specific function called **text()**.

text(x,y,s, fontdict=None, **kwargs)

The first two arguments are the coordinates of the location where you want to place the text. **s** is the string of text to be added, and **fontdict** (optional) is the font that you want the text to be represented. Finally, you can add the keywords.

Thus, add the label to each point of the plot. Because the first two arguments to the **text()** function are the coordinates of the graph, you have to use the coordinates of the four points of the plot shifted slightly on the y axis.

```
In [ ]: plt.axis([0,5,0,20])
   ...: plt.title('My first plot',fontsize=20,fontname='Times New Roman')
   ...: plt.xlabel('Counting',color='gray')
   ...: plt.ylabel('Square values',color='gray')
   ...: plt.text(1,1.5,'First')
   ...: plt.text(2,4.5,'Second')
   ...: plt.text(3,9.5,'Third')
   ...: plt.text(4,16.5,'Fourth')
   ...: plt.plot([1,2,3,4],[1,4,9,16],'ro')
Out[108]: [<matplotlib.lines.Line2D at 0x10f76898>]
```

As you can see in Figure 7-16 now each point of the plot has its label reporting a description.
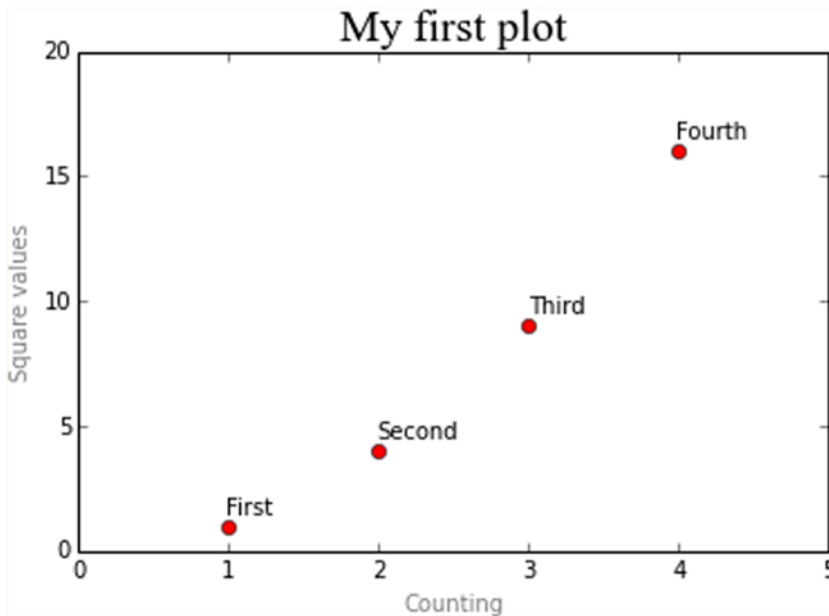
Figure 7-16. *Every point of the plot has an informative label*

Since matplotlib is a graphics library designed to be used in scientific circles, it must be able to exploit the full potential of scientific language, including mathematical expressions. Matplotlib offers the possibility to integrate LaTeX expressions allowing you to insert mathematical expressions within the chart.

To do this you can add a LaTeX expression to the text, enclosing it between two '**$**' characters. The interpreter will recognize them as LaTeX expressions and convert them in the corresponding graphic, which can be a mathematical expression, a formula, mathematical characters, or just Greek letters. Generally you have to precede the string containing LaTeX expressions with an '**r**', which indicates raw text in order to avoid unintended escape sequences.

Here, you can also make use of the keywords to further enrich the text to be shown in the plot. Therefore, as an example, you can add the formula describing the trend followed by the point of the plot and enclose it in a colored bounding box (see Figure 7-17).

```
In [ ]: plt.axis([0,5,0,20])
   ...: plt.title('My first plot',fontsize=20,fontname='Times New Roman')
   ...: plt.xlabel('Counting',color='gray')
   ...: plt.ylabel('Square values',color='gray')
   ...: plt.text(1,1.5,'First')
   ...: plt.text(2,4.5,'Second')
   ...: plt.text(3,9.5,'Third')
   ...: plt.text(4,16.5,'Fourth')
   ...: plt.text(1.1,12,r'$y = x^2$',fontsize=20,bbox={'facecolor':'yellow','alpha':0.2})
   ...: plt.plot([1,2,3,4],[1,4,9,16],'ro')

Out[130]: [<matplotlib.lines.Line2D at 0x13920860>]
```
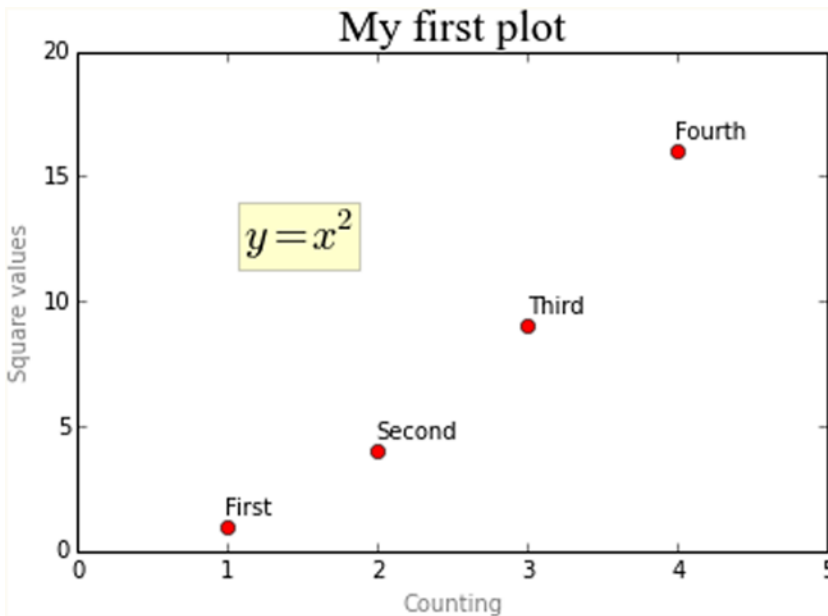
Figure 7-17. *Any mathematical expression can be seen in the context of a chart*

To get a complete view on the potential offered by LaTeX, you can consult the Appendix A of this book.

Adding a Grid

Another element you can add to a plot is a grid. Often its addition is necessary in order to better understand the position occupied by each point on the chart.

Adding a grid to a chart is a very simple operation: just add the **grid()** function, passing *True* as argument (see Figure 7-18).

```
In [ ]: plt.axis([0,5,0,20])
    ...: plt.title('My first plot',fontsize=20,fontname='Times New Roman')
    ...: plt.xlabel('Counting',color='gray')
    ...: plt.ylabel('Square values',color='gray')
    ...: plt.text(1,1.5,'First')
    ...: plt.text(2,4.5,'Second')
    ...: plt.text(3,9.5,'Third')
    ...: plt.text(4,16.5,'Fourth')
    ...: plt.text(1.1,12,r'$y = x^2$',fontsize=20,bbox={'facecolor':'yellow','alpha':0.2})
    ...: plt.grid(True)
    ...: plt.plot([1,2,3,4],[1,4,9,16],'ro')
Out[108]: [<matplotlib.lines.Line2D at 0x10f76898>]
```
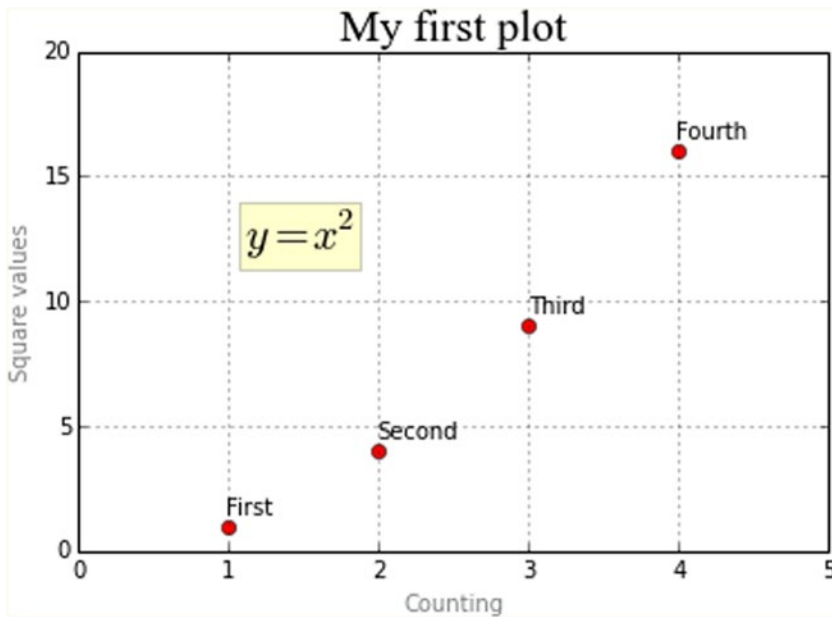
Figure 7-18. *A grid makes it easier to read the values of the data points represented in a chart*

Adding a Legend

Another very important component that should be present in any chart is the legend. pyplot also provides a specific function for this type of object: **legend()**.

Add a legend to your chart with the *legend()* function and a string indicating the words with which you want the series to be shown. In this example you will assign the 'First series' name to the input data array (See Figure 7-19).

```
In [ ]: plt.axis([0,5,0,20])
   ...: plt.title('My first plot',fontsize=20,fontname='Times New Roman')
   ...: plt.xlabel('Counting',color='gray')
   ...: plt.ylabel('Square values',color='gray')
   ...: plt.text(2,4.5,'Second')
   ...: plt.text(3,9.5,'Third')
   ...: plt.text(4,16.5,'Fourth')
   ...: plt.text(1.1,12,'$y = x^2$',fontsize=20,bbox={'facecolor':'yellow','alpha':0.2})
   ...: plt.grid(True)
   ...: plt.plot([1,2,3,4],[1,4,9,16],'ro')
   ...: plt.legend(['First series'])
Out[156]: <matplotlib.legend.Legend at 0x16377550>
```
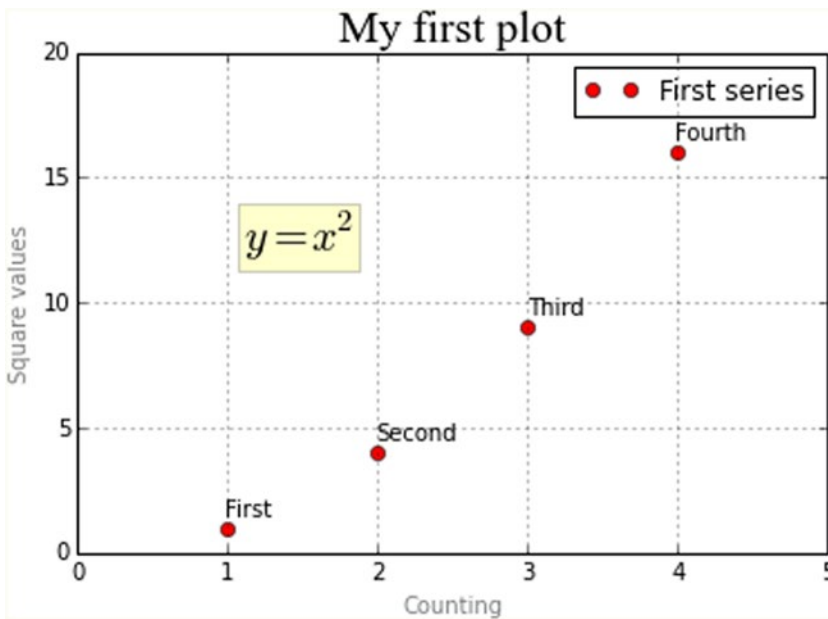

Figure 7-19. *A legend is added in the upper-right corner by default*

As you can see in Figure 7-19, the legend is added in the upper-right corner by default. Again if you want to change this behavior you will need to add a few kwargs. For example, the position occupied by the legend is set by assigning numbers from 0 to 10 to the **loc** kwarg. Each of these numbers characterizes one of the corners of the chart (see Table 7-1). A value of 1 is the default, that is, the upper-right corner. In the next example you will move the legend in the upper-left corner so it will not overlap with the points represented in the plot.

Table 7-1. *The Possible Values for the loc Keyword*

Location Code	Location String
0	best
1	upper-right
2	upper-left
3	lower-right
4	lower-left
5	right
6	center-left
7	center-right
8	lower-center
9	upper-center
10	center

Before you begin to modify the code to move the legend, I want to add a small notice. Generally, the legends are used to indicate the definition of a series to the reader via a label associated with a color and/or a marker that distinguishes it in the plot. So far, in the examples you have used a single series that was expressed by a single *plot()* function. Now, you have to focus on a more general case in which the same plot shows more series simultaneously. Each series in the chart will be characterized by a specific color and a specific marker (see Figure 7-20). In terms of code, instead, each series will be characterized by a call to the *plot()* function and the order in which they are defined will correspond to the order of the text labels passed as argument to the *legend()* function.

```
In [ ]: import matplotlib.pyplot as plt
   ...: plt.axis([0,5,0,20])
   ...: plt.title('My first plot',fontsize=20,fontname='Times New Roman')
   ...: plt.xlabel('Counting',color='gray')
   ...: plt.ylabel('Square values',color='gray')
   ...: plt.text(1,1.5,'First')
   ...: plt.text(2,4.5,'Second')
   ...: plt.text(3,9.5,'Third')
   ...: plt.text(4,16.5,'Fourth')
   ...: plt.text(1.1,12,'$y = x^2$',fontsize=20,bbox={'facecolor':'yellow','alpha':0.2})
   ...: plt.grid(True)
   ...: plt.plot([1,2,3,4],[1,4,9,16],'ro')
   ...: plt.plot([1,2,3,4],[0.8,3.5,8,15],'g^')
   ...: plt.plot([1,2,3,4],[0.5,2.5,4,12],'b*')
   ...: plt.legend(['First series','Second series','Third series'],loc=2)
Out[170]: <matplotlib.legend.Legend at 0x1828d7b8>
```
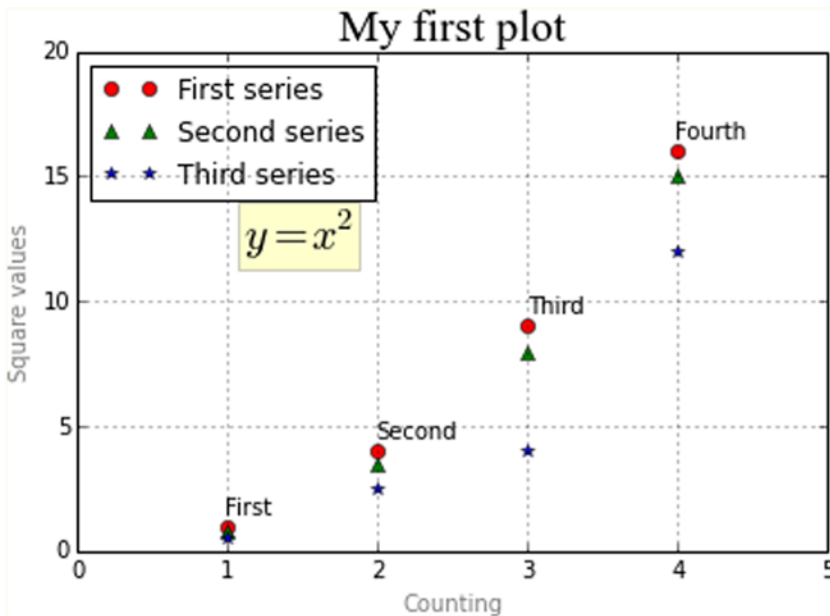
Figure 7-20. *A legend is necessary in every multiseries chart*

Saving Your Charts

In this section you will see how to save your chart in different ways depending on your purpose. If you need to reproduce your chart in different notebooks or Python sessions, or reuse them in future projects, it is a good practice to save the Python code. On the other hand, if you need to make reports or presentations, it can be very useful to save your chart as an image. Moreover, it is possible to save your chart as a HTML page, and this could be very useful when you need to share your work on Web.

Saving the Code

As you can see from the examples in the previous sections, the code concerning the representation of a single chart is growing into a fair number of rows. Once you think you've reached a good point in your development process you can choose to save all rows of code in a *.py* file that you can recall at any time.

You can use the magic command **save%** followed by the name of the file you want to save followed by the number of input prompts containing the row of code that you want to save. If all the code is written in only one prompt, as your case, you have to add only its number; otherwise if you want to save the code written in many prompts, for example from 10 to 20, you have to indicate this range with the two numbers separated by a '-', that is, 10-20.

In your case, you would to save, for instance, the Python code underlying the representation of your first chart contained into the input prompt with the number 171.

```
In [171]: import matplotlib.pyplot as plt
...
```

You need to insert the following command to save the code into a new .py file.

```
%save my_first_chart 171
```

After you launch the command, you will find the file *my_first_chart.py* in your working directory (See Listing 7-1).

Listing 7-1. my_first_chart.py

```
# coding: utf-8
import matplotlib.pyplot as plt
plt.axis([0,5,0,20])
plt.title('My first plot',fontsize=20,fontname='Times New Roman')
plt.xlabel('Counting',color='gray')
plt.ylabel('Square values',color='gray')
plt.text(1,1.5,'First')
plt.text(2,4.5,'Second')
plt.text(3,9.5,'Third')
plt.text(4,16.5,'Fourth')
plt.text(1.1,12,'$y = x^2$',fontsize=20,bbox={'facecolor':'yellow','alpha':0.2})
plt.grid(True)
plt.plot([1,2,3,4],[1,4,9,16],'ro')
plt.plot([1,2,3,4],[0.8,3.5,8,15],'g^')
plt.plot([1,2,3,4],[0.5,2.5,4,12],'b*')
plt.legend(['First series','Second series','Third series'],loc=2)
```

Later, when you open a new IPython session, you will have your chart and start to change the code at the point where you had saved it by entering the following command:

```
ipython qtconsole --matplotlib inline -m my_first_chart.py
```

or you can reload the entire code in a single prompt in the QtConsole using the magic command **%load**.

```
%load my_first_chart.py
```

or run it during a session with the magic command **%run**.

```
%run my_first_chart.py
```

■ **Note**   In my system, this command works only after launching the two previous commands.

Converting Your Session as an HTML File

Using Ipython QtConsole, you can convert all the code and graphics present in your current session in an HTML page. You have only to select *File* ➤ *Save to HTML / XHTML* in the menu (as shown in Figure 7-21).
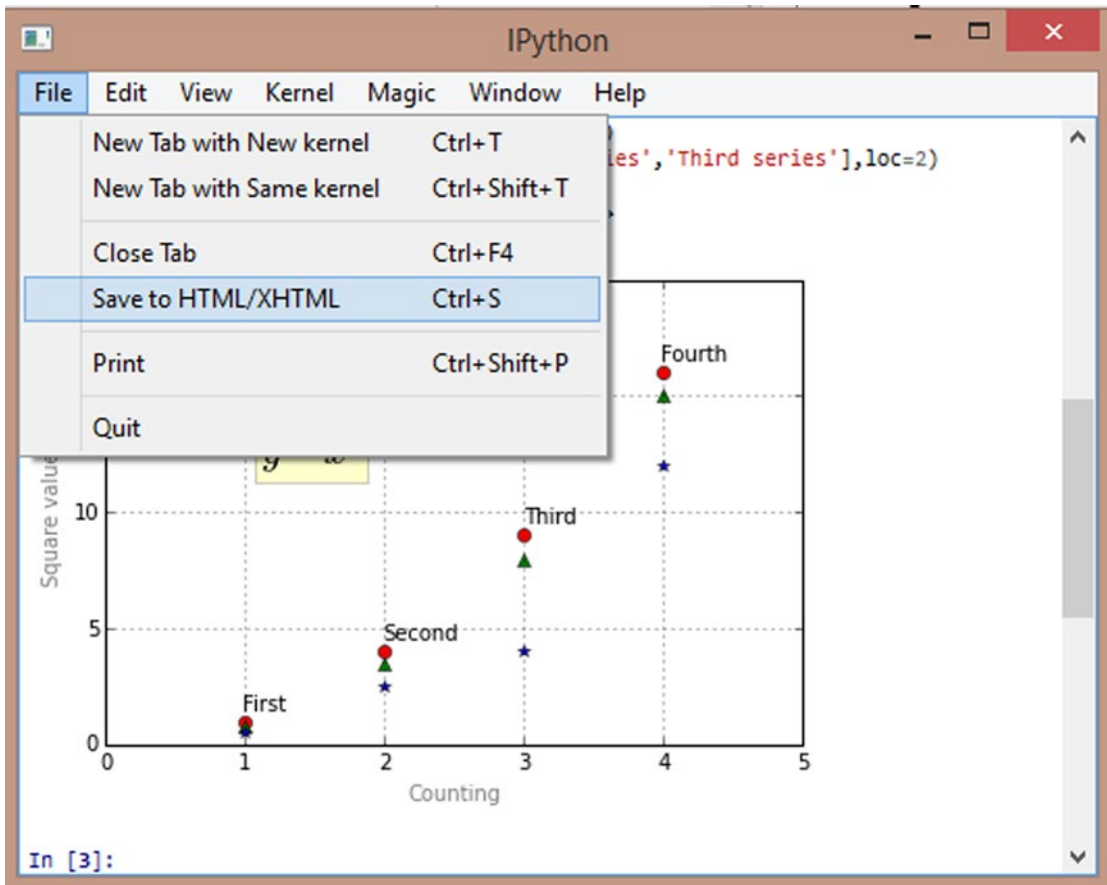
Figure 7-21. *You can save your current session as a web page*

You will be asked to save your session in two different formats: HTML and XHMTL. The difference between the two formats is based on the type of conversion of the images. If you select HTML as output file format, the images contained in your session will be converted to PNG format. If you select XHTML as output file format instead, the images will be converted to SVG format.

In this example, save your session as an HTML file and name it as *my_session.html* as shown in Figure 7-22.
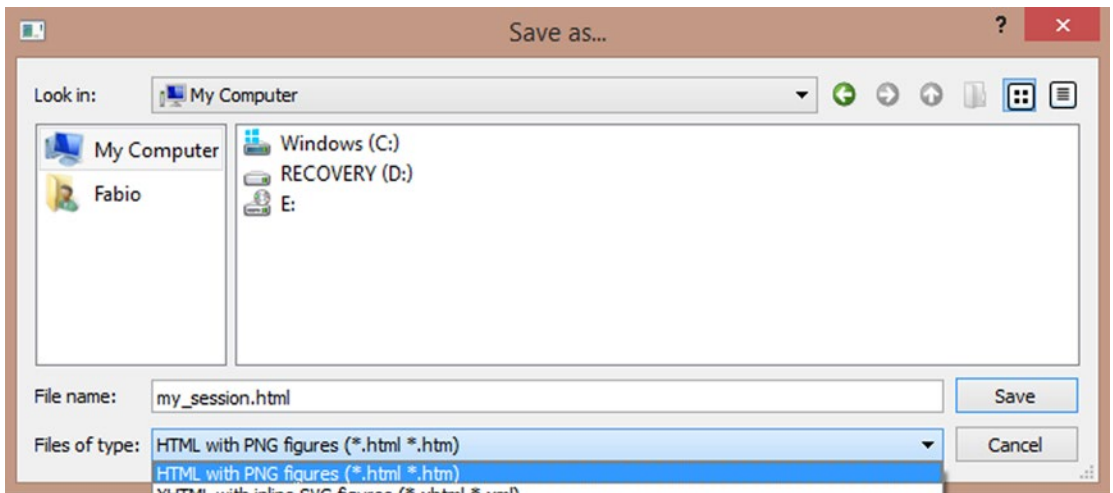
Figure 7-22. *You can select the type of file between HTML and XHTML*

At this point you will be asked if you want to save your images in an external directory or inline (Figure 7-23).
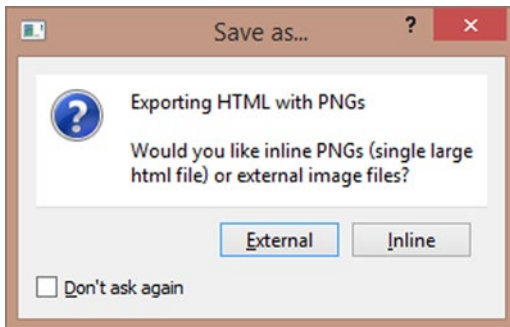
Figure 7-23. *You can choose between creating external image files and embedding the PNG format directly into the HTML page*

Choosing the external option, you will have images of your chart collected within a directory called *my_session_files*. Instead, choosing the other options, the graphic information concerning the image will be totally embedded into the HTML code.

Saving Your Chart Directly as an Image

If you are interested in saving only the figure of a chart as an image file, ignoring all the code you've written during the session, this is also possible. In fact, thanks to the **savefig()** function, you can directly save the chart in a PNG format, although you should take care to add this function to the end of the same series of commands (otherwise you'll get a blank PNG file).

```
In [ ]: plt.axis([0,5,0,20])
   ...: plt.title('My first plot',fontsize=20,fontname='Times New Roman')
   ...: plt.xlabel('Counting',color='gray')
   ...: plt.ylabel('Square values',color='gray')
   ...: plt.text(1,1.5,'First')
   ...: plt.text(2,4.5,'Second')
   ...: plt.text(3,9.5,'Third')
   ...: plt.text(4,16.5,'Fourth')
   ...: plt.text(1.1,12,'$y = x^2$',fontsize=20,bbox={'facecolor':'yellow','alpha':0.2})
   ...: plt.grid(True)
   ...: plt.plot([1,2,3,4],[1,4,9,16],'ro')
   ...: plt.plot([1,2,3,4],[0.8,3.5,8,15],'g^')
   ...: plt.plot([1,2,3,4],[0.5,2.5,4,12],'b*')
   ...: plt.legend(['First series','Second series','Third series'],loc=2)
   ...: plt.savefig('my_chart.png')
```

Executing the previous code, a new file will be created in your working directory. This file will be named *my_chart.png* containing the image of your chart.

Handling Date Values

One of the most common problems encountered when doing data analysis is handling data of the date-time type. Displaying them along an axis (normally the x axis) can be really problematic especially for the management of ticks (see Figure 7-24).

Take for example the display of a linear chart with a data set of eight points in which you have to represent date values on the x axis with the following format: day-month-year.

```
In [ ]: import datetime
   ...: import numpy as np
   ...: import matplotlib.pyplot as plt
   ...: events = [datetime.date(2015,1,23),datetime.date(2015,1,28),datetime.
        date(2015,2,3),datetime.date(2015,2,21),datetime.date(2015,3,15),datetime.
        date(2015,3,24),datetime.date(2015,4,8),datetime.date(2015,4,24)]
   ...: readings = [12,22,25,20,18,15,17,14]
   ...: plt.plot(events,readings)
Out[83]: [<matplotlib.lines.Line2D at 0x12666400>]
```
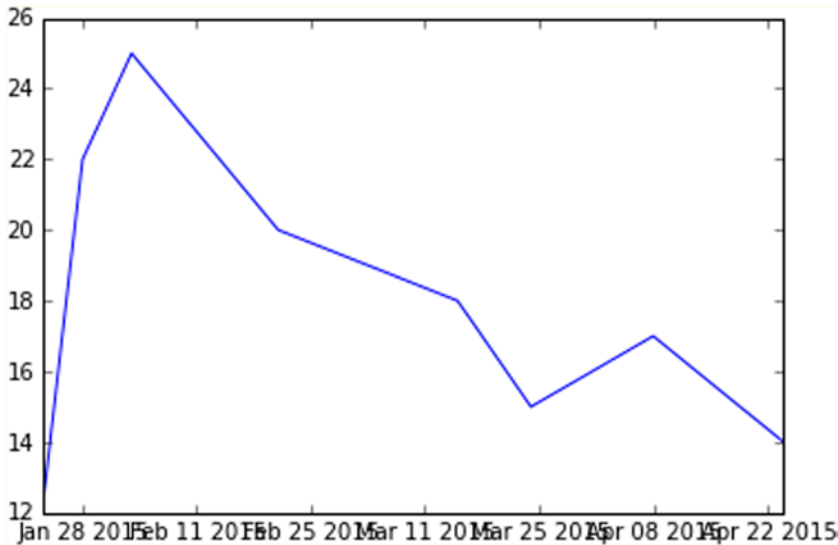
Figure 7-24. *If not handled, displaying date-time values can be problematic*

As you can see in Figure 7-24, the automatic management of ticks and especially the tick labels, it is a disaster. The dates expressed in this way are difficult to read, there are no clear time intervals elapsed between one point and another, and there is also overlap.

To manage the dates therefore it is advisable to define a time scale with appropriate objects. First you need to import matplotlib.dates, a module specialized for the management of this type of data. Then you define the scales of the times, as in this case, a scale of days and one of the months, through the functions **MonthLocator()** and **DayLocator()**. In these cases, the formatting is also very important, and to avoid overlap or unnecessary references, you have to limit the tick labels to the essential, which in this case is year-month. This format can be passed as an argument to **DateFormatter()** function.

After you defined the two scales, one for the days and one for the months, you can set two different kinds of ticks on the x axis, using the functions **set_major_locator()** and **set_minor_locator()** on the **xaxis** object. Instead, to set the text format of the tick labels referred to the months you have to use the **set_major_formatter()** function.

Making all these settings you finally obtain the plot as shown in Figure 7-25.

```
In [ ]: import datetime
   ...: import numpy as np
   ...: import matplotlib.pyplot as plt
   ...: import matplotlib.dates as mdates
   ...: months = mdates.MonthLocator()
   ...: days = mdates.DayLocator()
   ...: timeFmt = mdates.DateFormatter('%Y-%m')
   ...: events = [datetime.date(2015,1,23),datetime.date(2015,1,28),datetime.
        date(2015,2,3),datetime.date(2015,2,21),datetime.date(2015,3,15),datetime.
        date(2015,3,24),datetime.date(2015,4,8),datetime.date(2015,4,24)]
        readings = [12,22,25,20,18,15,17,14]
   ...: fig, ax = plt.subplots()
   ...: plt.plot(events,readings)
   ...: ax.xaxis.set_major_locator(months)
   ...: ax.xaxis.set_major_formatter(timeFmt)
   ...: ax.xaxis.set_minor_locator(days)
```
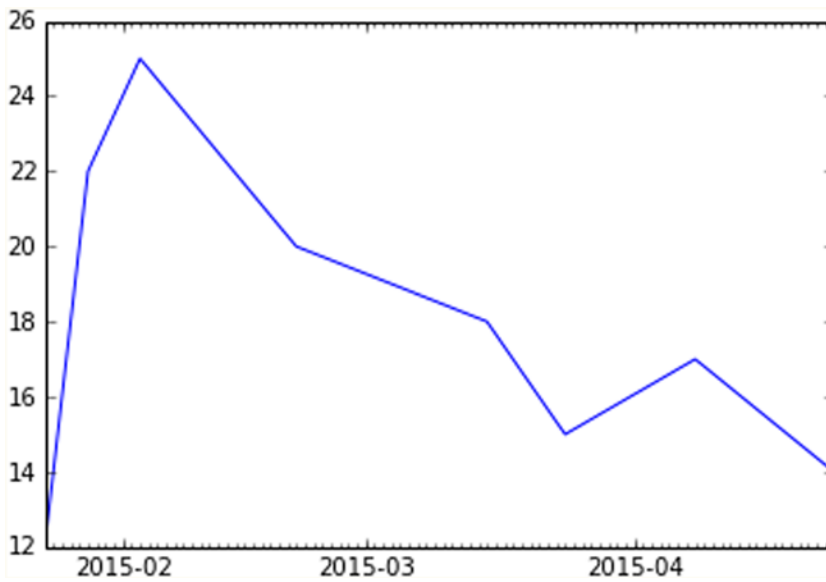
Figure 7-25. *Now the tick labels of the x axes refer only to the months, making the plot more readable*

Chart Typology

In the previous sections you have seen a number of examples relating to the architecture of the matplotlib library. Now that you are familiar with the use of the main graphic elements within a chart, it is time to see a series of examples treating different type of charts, starting from the most common ones such as linear charts, bar charts, and pie charts, up to a discussion about some that are more sophisticated but commonly used nonetheless.

This part of the chapter is very important since the purpose of this library is precisely the visualization of the results produced by the data analysis. Thus, knowing how to choose the type of chart to our data is a fundamental choice. Remember that even an excellent data analysis represented incorrectly can lead to a wrong interpretation of the experimental results.

Line Chart

Among all the types of chart the linear chart is the simplest. A line chart is a sequence of data points connected by a line. Each data point consists of a pair of values (x,y), which will be reported in the chart according to the scale of values of the two axes (x and y).

By way of example you can begin to plot the points generated by a mathematical function. Then, you can consider a generic mathematical function such as

$$y = sin\ (3 * x)\ /\ x$$

Therefore, if you want to create a sequence of data points, you need to create two NumPy arrays. First you create an array containing the x values to be referred to the x axis. In order to define a sequence of increasing values you will use the *np.arange()* function. Since the function is sinusoidal you should refer to values that are multiples and submultiples of the greek pi (*np.pi*). Then using these sequence of values you can obtain the y values applying the np.sin() function directly to these values (Thanks to NumPy!).

After all this, you have only to plot them calling the plot() function. You will obtain a line chart as shown in Figure 7-26.

```
In [ ]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: x = np.arange(-2*np.pi,2*np.pi,0.01)
   ...: y = np.sin(3*x)/x
   ...: plt.plot(x,y)
Out[393]: [<matplotlib.lines.Line2D at 0x22404358>]
```
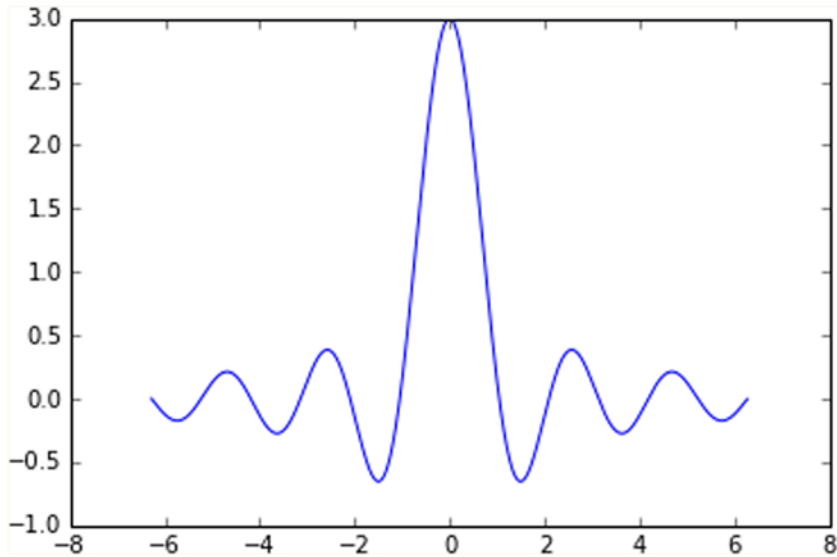

Figure 7-26. *A mathematical function represented in a line chart*

Now you can extend the case in which you want to display a family of functions such as

$$y = sin \ (n * x) \ / \ x$$

varying the parameter *n*.

```
In [ ]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: x = np.arange(-2*np.pi,2*np.pi,0.01)
   ...: y = np.sin(3*x)/x
   ...: y2 = np.sin(2*x)/x
   ...: y3 = np.sin(3*x)/x
   ...: plt.plot(x,y)
   ...: plt.plot(x,y2)
   ...: plt.plot(x,y3)
```

As you can see in Figure 7-27, a different color is automatically assigned to each line. All the plots are represented on the same scale; that is, the data points of each series refer to the same x axis and y axis. This is because each call of the *plot()* function takes into account the previous calls to same function, so the Figure applies the changes keeping memory of the previous commands until the Figure is not displayed (show() with Python and ENTER with IPython QtConsole).
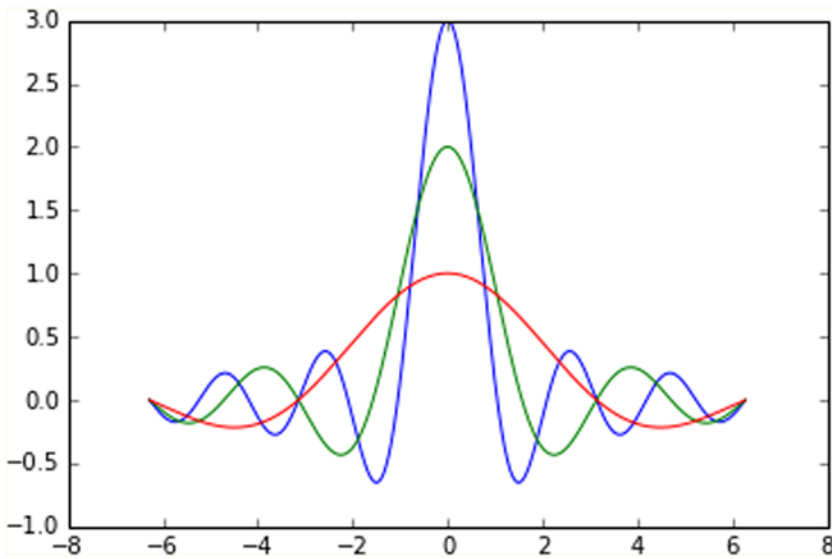
Figure 7-27. *Three different series are drawn with different colors in the same chart*

As we saw in the previous sections, regardless of the default settings, you can select the type of stroke, color, etc. As the third argument of the *plot()* function you can specify some codes that correspond to the color (see Table 7-2) and other codes that correspond to line styles, all included in the same string. Another possibility is to use two kwargs separately, **color** to define the color and **linestyle** to define the stroke (see Figure 7-28).

```
In [ ]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: x = np.arange(-2*np.pi,2*np.pi,0.01)
   ...: y = np.sin(3*x)/x
   ...: y2 = np.sin(2*x)/x
   ...: y3 = np.sin(3*x)/x
   ...: plt.plot(x,y,'k--',linewidth=3)
   ...: plt.plot(x,y2,'m-.')
   ...: plt.plot(x,y3,color='#87a3cc',linestyle='--')
```
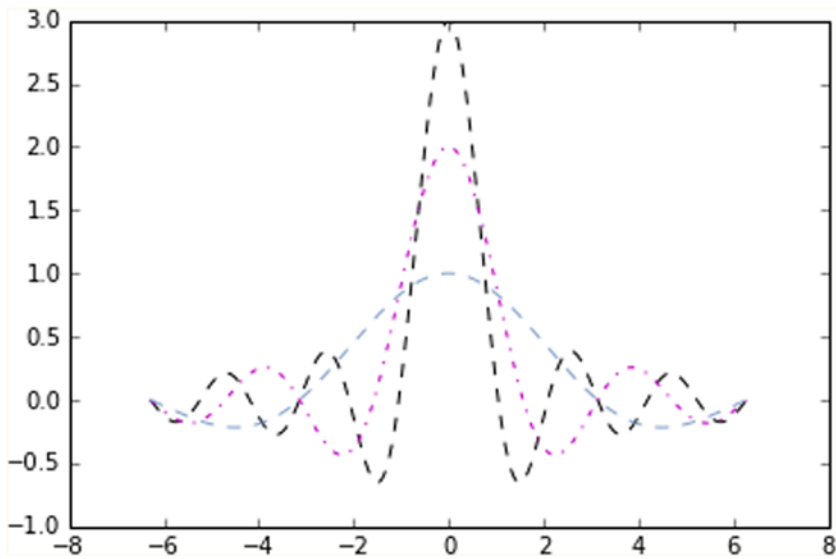
Figure 7-28. *You can define colors and line styles using character codes*

Table 7-2. *Color Codes*

Code	Color
b	blue
g	green
r	red
c	cyan
m	magenta
y	yellow
k	black
w	white

You have just defined a range from -2π to 2π on the x axis, but by default values on ticks are shown in numerical form. Therefore you need to replace the numerical values with multiple of π. You can also replace the ticks on the y axis. To do all this you have to use **xticks()** and **yticks()** functions, passing to each of them two lists of values. The first list contains values corresponding to the positions where the ticks are to be placed, and the second contains the tick labels. In this particular case, you have to use strings containing LaTeX format in order to correctly display the symbol π. Remember to define them within two '$' characters and add a 'r' as prefix.

```
In [ ]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: x = np.arange(-2*np.pi,2*np.pi,0.01)
   ...: y = np.sin(3*x)/x
   ...: y2 = np.sin(2*x)/x
   ...: y3 = np.sin(x)/x
   ...: plt.plot(x,y,color='b')
   ...: plt.plot(x,y2,color='r')
   ...: plt.plot(x,y3,color='g')
   ...: plt.xticks([-2*np.pi, -np.pi, 0, np.pi, 2*np.pi],
           [r'$-2\pi$',r'$-\pi$',r'$0$',r'$+\pi$',r'$+2\pi$'])
   ...: plt.yticks([-1,0,+1,+2,+3],
           [r'$-1$',r'$0$',r'$+1$',r'$+2$',r'$+3$'])
Out[423]:
([<matplotlib.axis.YTick at 0x26877ac8>,
  <matplotlib.axis.YTick at 0x271d26d8>,
  <matplotlib.axis.YTick at 0x273c7f98>,
  <matplotlib.axis.YTick at 0x273cc470>,
  <matplotlib.axis.YTick at 0x273cc9e8>],
 <a list of 5 Text yticklabel objects>)
```

At the end, you will get a clean and pleasant line chart showing Greek characters as in Figure 7-29.
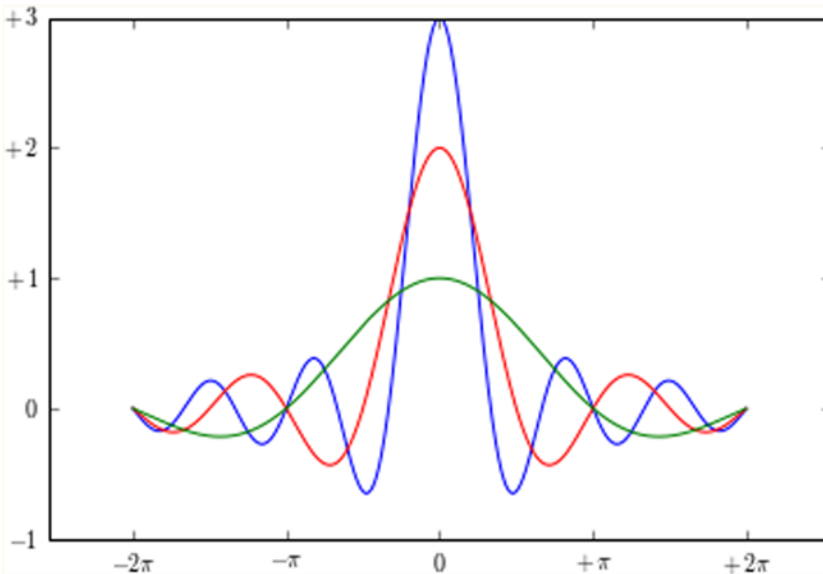
Figure 7-29. *The tick label can be improved adding text with LaTeX format*

In all the linear charts you have seen so far, you always have the x axis and y axis placed at the edge of the Figure (corresponding to the sides of the bounding border box). Another way of displaying axes is to have the two axes passing through the origin (0, 0), i.e., the two Cartesian axes.

To do this, you must first capture the Axes object through the **gca()** function. Then through this object, you can select each of the four sides making up the bounding box, specifying for each one its position: right, left, bottom, and top. Crop the sides that do not match any axis (right and bottom) using the **set_color()** function, indicating *none* as color. Then, the sides corresponding to the x and y axes are moved to pass through the origin (0,0) with the **set_position()** function.

```
In [ ]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: x = np.arange(-2*np.pi,2*np.pi,0.01)
   ...: y = np.sin(3*x)/x
   ...: y2 = np.sin(2*x)/x
   ...: y3 = np.sin(x)/x
   ...: plt.plot(x,y,color='b')
   ...: plt.plot(x,y2,color='r')
   ...: plt.plot(x,y3,color='g')
   ...: plt.xticks([-2*np.pi, -np.pi, 0, np.pi, 2*np.pi],
           [r'$-2\pi$',r'$-\pi$',r'$0$',r'$+\pi$',r'$+2\pi$'])
   ...: plt.yticks([-1,0,+1,+2,+3],
           [r'$-1$',r'$0$',r'$+1$',r'$+2$',r'$+3$'])
   ...: ax = plt.gca()
   ...: ax.spines['right'].set_color('none')
   ...: ax.spines['top'].set_color('none')
   ...: ax.xaxis.set_ticks_position('bottom')
   ...: ax.spines['bottom'].set_position(('data',0))
   ...: ax.yaxis.set_ticks_position('left')
   ...: ax.spines['left'].set_position(('data',0))
```

Now the chart will show the two axes crossing in the middle of the figure, that is, the origin of the Cartesian axes as shown in Figure 7-30.
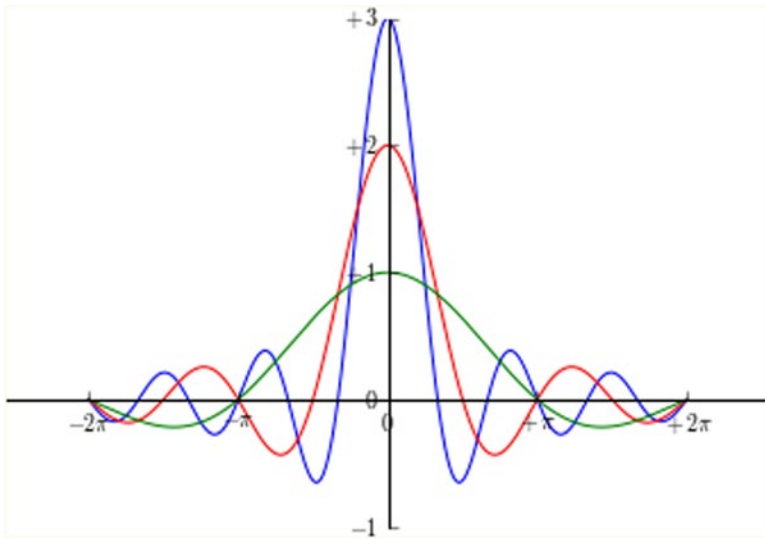
Figure 7-30. *The chart shows two Cartesian axes*

Often, it is very useful to be able to specify a particular point of the line using a notation and optionally adding an arrow to better indicate the position of the point. For example, this notation may be a LaTeX expression, such as the formula for the limit of the function sinx / x with x tends to 0.

In this regard matplotlib provides a function called **annotate()**, especially useful in these cases, even if the numerous kwargs needed to obtain a good result can make its settings quite complex. The first argument is the string to be represented containing the expression in LaTeX; then you can add the various kwargs. The point of the chart to note is indicated by a list containing the coordinates of the point [x, y] passed to the **xy** kwarg. The distance of the textual notation from the point to be highlighted is defined by the **xytext** kwarg and represented by means of a curved arrow whose characteristics are defined in the **arrowprops** kwarg.

```
In [ ]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: x = np.arange(-2*np.pi,2*np.pi,0.01)
   ...: y = np.sin(3*x)/x
   ...: y2 = np.sin(2*x)/x
   ...: y3 = np.sin(x)/x
   ...: plt.plot(x,y,color='b')
   ...: plt.plot(x,y2,color='r')
   ...: plt.plot(x,y3,color='g')
   ...: plt.xticks([-2*np.pi, -np.pi, 0, np.pi, 2*np.pi],
           [r'$-2\pi$',r'$-\pi$',r'$0$',r'$+\pi$',r'$+2\pi$'])
   ...: plt.yticks([-1,0,+1,+2,+3],
           [r'$-1$',r'$0$',r'$+1$',r'$+2$',r'$+3$'])
   ...: plt.annotate(r'$\lim_{x\to 0}\frac{\sin(x)}{x}= 1$',    xy=[0,1],xycoords='data',
       xytext=[30,30],fontsize=16,textcoords='offset points',arrowprops=dict(arrowstyle="->",
       connectionstyle="arc3,rad=.2"))
```

```
    ...: ax = plt.gca()
    ...: ax.spines['right'].set_color('none')
    ...: ax.spines['top'].set_color('none')
    ...: ax.xaxis.set_ticks_position('bottom')
    ...: ax.spines['bottom'].set_position(('data',0))
    ...: ax.yaxis.set_ticks_position('left')
    ...: ax.spines['left'].set_position(('data',0))
```

Running this code you will get the chart with the mathematical notation of the limit, which is the point shown by the arrow in Figure 7-31).
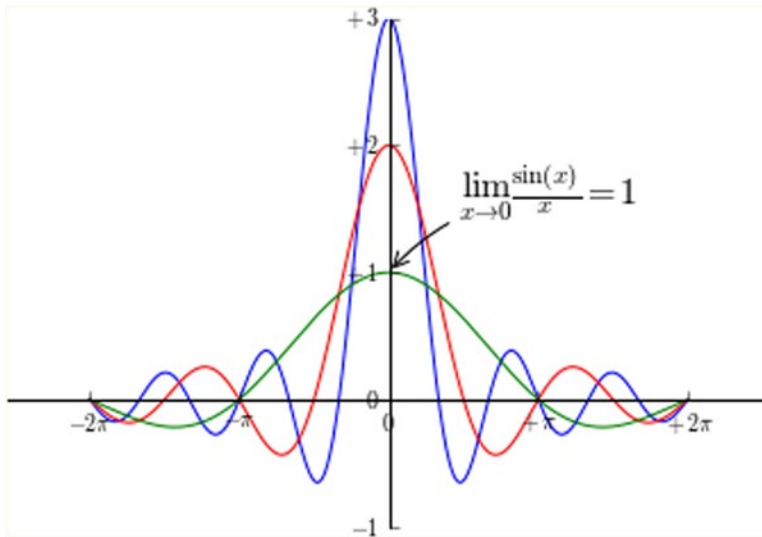
Figure 7-31. *Mathematical expressions can be added to a chart with annotate() function*

Line Charts with pandas

Moving to more practical cases, or at least more closely related to the data analysis, now is the time to see how easy it is, applying the matplotlib library with the dataframes of the pandas library. The visualization of the data in a dataframe as a linear chart is a very simple operation. It is sufficient to pass the dataframe as argument to the *plot()* function to obtain a multiseries linear chart (see Figure 7-32).

```
In [ ]: import matplotlib.pyplot as plt
    ...: import numpy as np
    ...: import pandas as pd
    ...: data = {'series1':[1,3,4,3,5],
                 'series2':[2,4,5,2,4],
                 'series3':[3,2,3,1,3]}
    ...: df = pd.DataFrame(data)
    ...: x = np.arange(5)
    ...: plt.axis([0,5,0,7])
    ...: plt.plot(x,df)
    ...: plt.legend(data, loc=2)
```
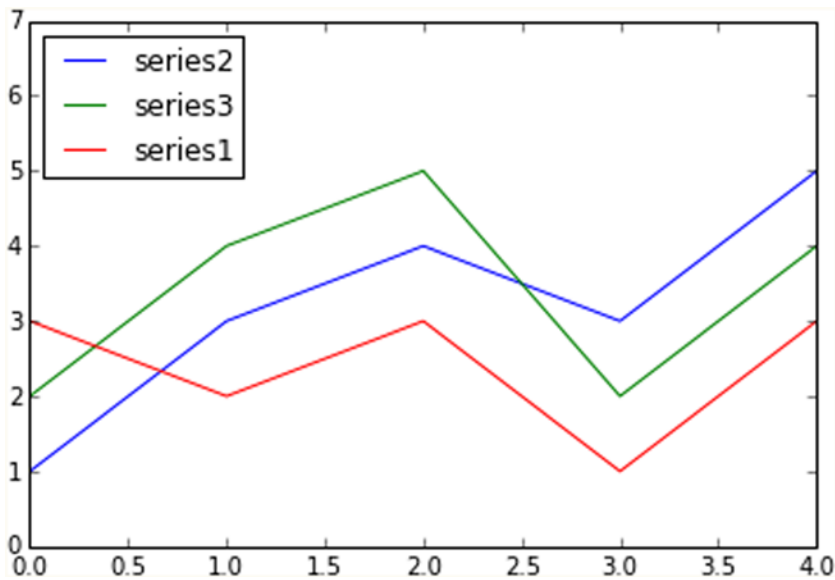
Figure 7-32. *The multiseries line chart displays the data within a pandas DataFrame*

Histogram

A histogram consists of adjacent rectangles erected on the x axis, split into discrete intervals called bins, and with an area proportional to the frequency of the occurrences for that bin. This kind of visualization is commonly used in statistical studies about distribution of samples.

In order to represent a histogram pyplot provides a special function called **hist()**. This graphic function also has a feature that other functions producing charts do not have. The *hist()* function, in addition to drawing the histogram, returns a tuple of values that are the results of the calculation of the histogram. In fact the *hist()* function can also implement the calculation of the histogram, that is, it is sufficient to provide a series of samples of values as an argument and the number of bins in which to be divided, and it will take care of dividing the range of samples in many intervals (bins), and then calculate the occurrences for each bin. The result of this operation, in addition to being shown in graphical form (see Figure 7-33), will be returned in the form of a tuple.

(n, bins, patches)

To see this operation there is no better explanation of a practical example. Then generate a population of 100 random values from 0 to 100 using the **random.randint()** function.

```
In [ ]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: pop = np.random.randint(0,100,100)
   ...: pop
Out[ ]:
array([32, 14, 55, 33, 54, 85, 35, 50, 91, 54, 44, 74, 77,  6, 77, 74,  2,
       54, 14, 30, 80, 70,  6, 37, 62, 68, 88,  4, 35, 97, 50, 85, 19, 90,
       65, 86, 29, 99, 15, 48, 67, 96, 81, 34, 43, 41, 21, 79, 96, 56, 68,
       49, 43, 93, 63, 26,  4, 21, 19, 64, 16, 47, 57,  5, 12, 28,  7, 75,
        6, 33, 92, 44, 23, 11, 61, 40,  5, 91, 34, 58, 48, 75, 10, 39, 77,
       70, 84, 95, 46, 81, 27,  6, 83,  9, 79, 39, 90, 77, 94, 29])
```

Now, create the histogram of these samples by passing as an argument of the **hist()**. function. For example, you want to divide the occurrences in 20 bins (if not specified, the default value is 10 bins) and to do that you have to use the kwarg bin (as shown in Figure 7-33).

```
In [ ]: n,bins,patches = plt.hist(pop,bins=20)
```
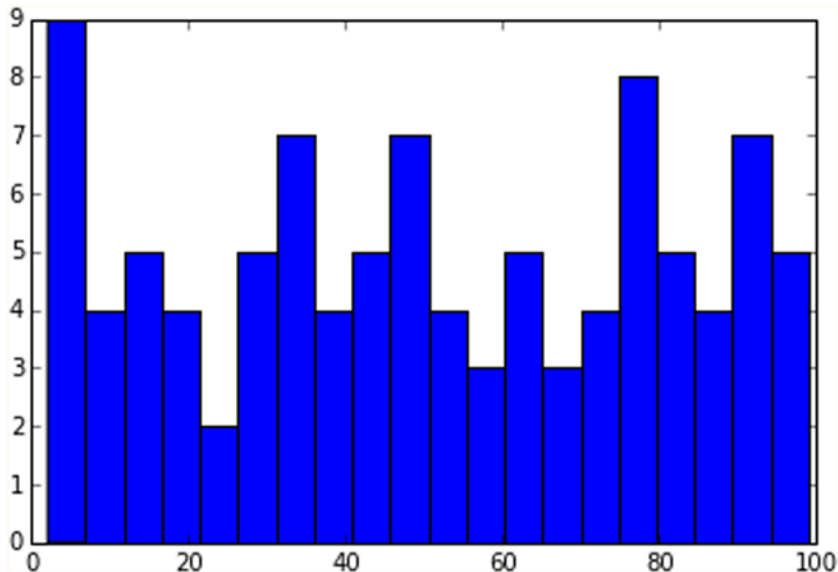

Figure 7-33. *The histogram shows the occurrences in each bin*

Bar Chart

Another very common type of chart is the bar chart. It is very similar to a histogram but in this case the x axis is not used to reference numerical values but categories. The realization of the bar chart is very simple with matplotlib using the **bar()** function.

```
In [ ]: import matplotlib.pyplot as plt
    ...: index = [0,1,2,3,4]
    ...: values = [5,7,3,4,6]
    ...: plt.bar(index,values)
Out[15]: <Container object of 5 artists>
```

With this few rows of code you will obtain a bar chart as shown in Figure 7-34.
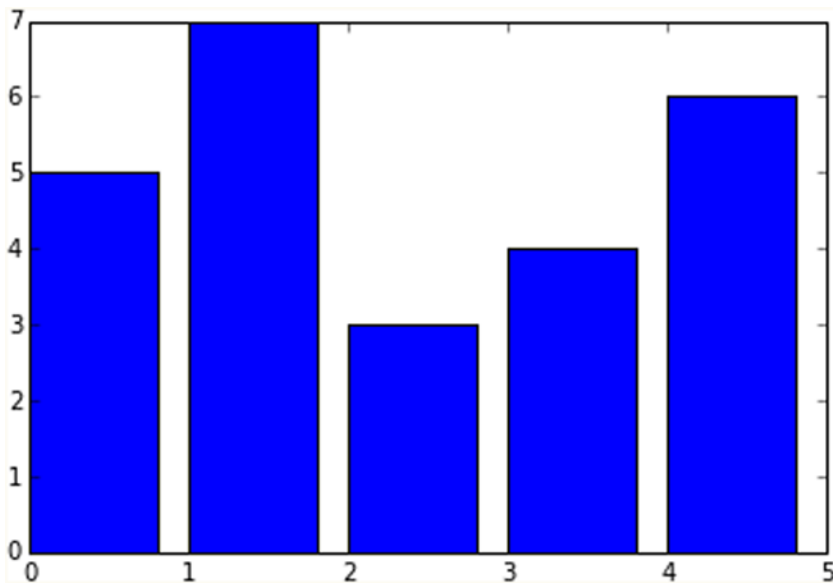
Figure 7-34. *The most simple bar chart with matplotlib*

If you look at Figure 7-34 you can see that the indices are drawn on the x axis at the beginning of each bar. Actually, because each bar corresponds to a category, it would be better if you specify the categories through the tick label, defined by a list of strings passed to the **xticks()** function. As for the location of these tick labels, you have to pass a list containing the values corresponding to their positions on the x axis as the first argument of the *xticks()* function. At the end you will get a bar chart as shown in Figure 7-35.

```
In [ ]: import numpy as np
   ...: index = np.arange(5)
   ...: values1 = [5,7,3,4,6]
   ...: plt.bar(index,values1)
   ...: plt.xticks(index+0.4,['A','B','C','D','E'])
```
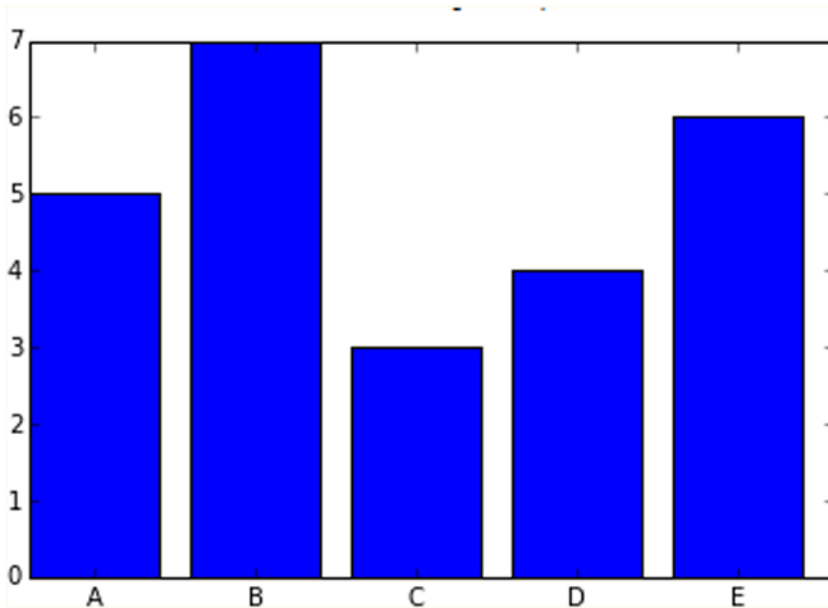
Figure 7-35. *A simple bar chart with categories on x axis*

Actually there are many other steps we can take to further refine the bar chart. Each of these finishes is set by adding a specific kwarg as an argument in the **bar()** function. For example, you can add the standard deviation values of the bar through the **yerr** kwarg along with a list containing the standard deviations. This kwarg is usually combined with another kwarg called **error_kw**, which, in turn, accepts other kwargs specialized for representing error bars. Two very specific kwargs used in this case are **eColor**, which specifies the color of the error bars, and **capsize**, which defines the width of the transverse lines that mark the ends of the error bars.

Another kwarg that you can use is **alpha**, which indicates the degree of transparency of the colored bar. Alpha is a value ranging from 0 to 1. When this value is 0 the object is completely transparent to become gradually more significant with the increase of the value, until arriving at 1, at which the color is fully represented.

As usual, the use of a legend is recommended, so in this case you should use a kwarg called **label** to identify the series that you are representing.

At the end you will get a bar chart with error bars as shown in Figure 7-36.

```
In [ ]: import numpy as np
   ...: index = np.arange(5)
   ...: values1 = [5,7,3,4,6]
   ...: std1 = [0.8,1,0.4,0.9,1.3]
   ...: plt.title('A Bar Chart')
   ...: plt.bar(index,values1,yerr=std1,error_kw={'ecolor':'0.1',
   ...:     'capsize':6},alpha=0.7,label='First')
   ...: plt.xticks(index+0.4,['A','B','C','D','E'])
   ...: plt.legend(loc=2)
```
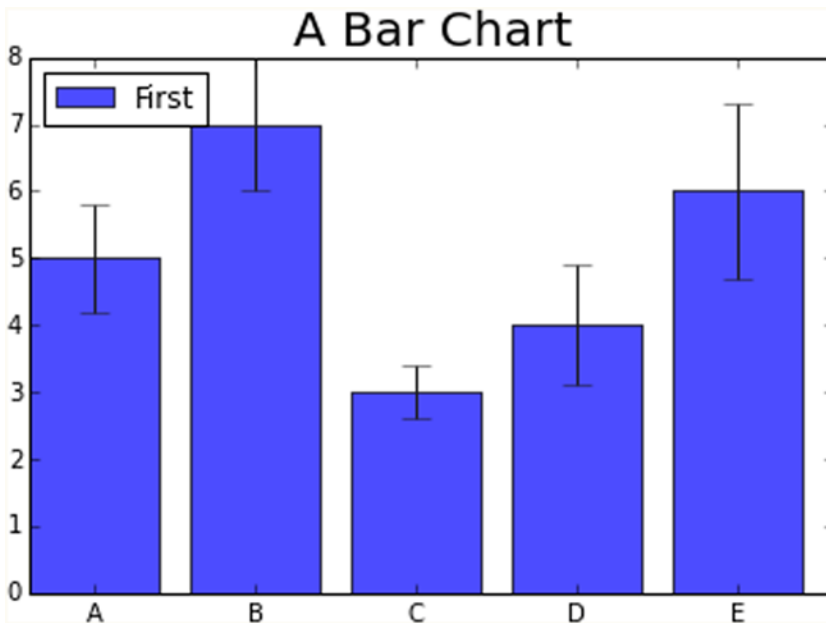
Figure 7-36. *A bar chart with error bars*

Horizontal Bar Chart

So far you have seen the bar chart oriented in vertical form. There are also bar chart oriented horizontally. This mode is implemented by a special function called **barh()**. The arguments and the kwargs valid for the bar() function remain so even for this function. The only care that you have to take into account is that the roles of the axes are reversed. Now, the categories are represented on the y axis and the numerical values are shown on the x axis (see Figure 7-37).

```
In [ ]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: index = np.arange(5)
   ...: values1 = [5,7,3,4,6]
   ...: std1 = [0.8,1,0.4,0.9,1.3]
   ...: plt.title('A Horizontal Bar Chart')
   ...: plt.barh(index,values1,xerr=std1,error_kw={'ecolor':'0.1','capsize':6},alpha=0.7,
       label='First')
   ...: plt.yticks(index+0.4,['A','B','C','D','E'])
   ...: plt.legend(loc=5)
```
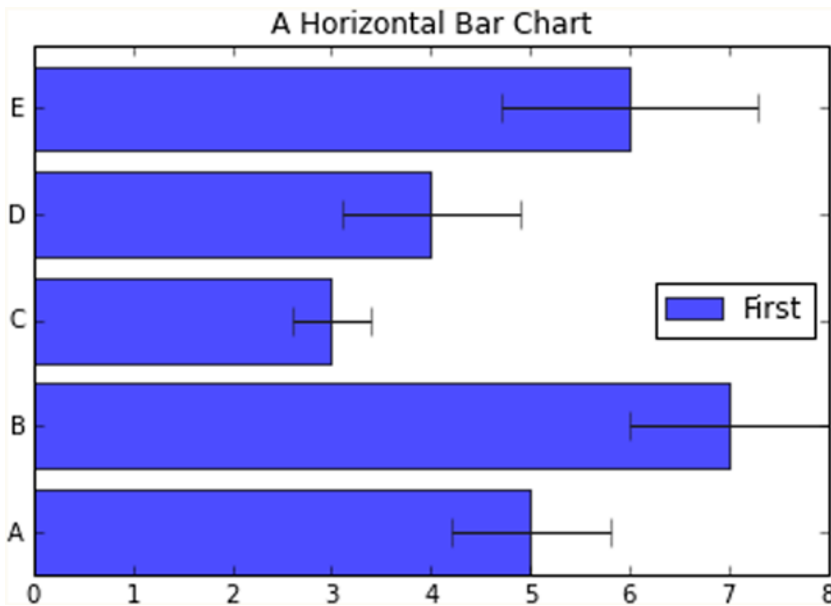
Figure 7-37. *A simple horizontal bar chart*

Multiserial Bar Chart

As line charts, bar charts also generally are used to simultaneously to display more series of values. But in this case it is necessary to make some clarifications on how to structure a multiseries bar chart. So far you have defined a sequence of indexes, each corresponding to a bar, to be assigned to the x axis. These indices should represent categories. In this case, however, we have more bars that must share the same category.

One approach used to overcome this problem is to divide the space occupied by an index (for convenience its width is 1) in as many parts as are the bars sharing that index and that we want to display. Moreover, it is advisable to add an additional space which will serve as gap to separate a category with respect to the next (as shown in Figure 7-38).

```
In [ ]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: index = np.arange(5)
   ...: values1 = [5,7,3,4,6]
   ...: values2 = [6,6,4,5,7]
   ...: values3 = [5,6,5,4,6]
   ...: bw = 0.3
   ...: plt.axis([0,5,0,8])
   ...: plt.title('A Multiseries Bar Chart',fontsize=20)
   ...: plt.bar(index,values1,bw,color='b')
   ...: plt.bar(index+bw,values2,bw,color='g')
   ...: plt.bar(index+2*bw,values3,bw,color='r')
   ...: plt.xticks(index+1.5*bw,['A','B','C','D','E'])
```
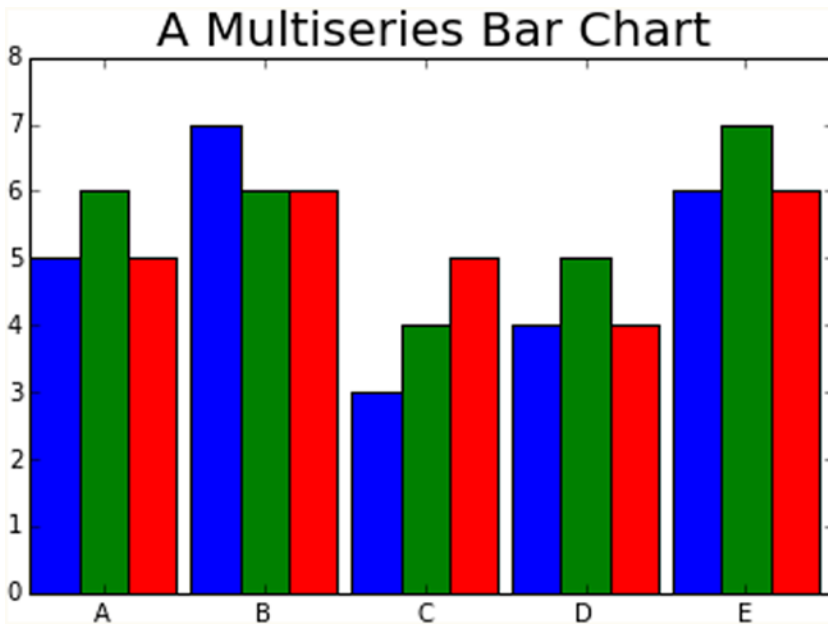
Figure 7-38. *A multiseries bar chart displaying three series*

Regarding the multiseries horizontal bar chart (Figure 7-39), things are very similar. You have to replace the *bar()* function with the corresponding *barh()* function and also remember to replace the *xticks()* function with the *yticks()* function. You need to reverse the range of values covered by the axes in the *axis()* function.

```
In [ ]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: index = np.arange(5)
   ...: values1 = [5,7,3,4,6]
   ...: values2 = [6,6,4,5,7]
   ...: values3 = [5,6,5,4,6]
   ...: bw = 0.3
   ...: plt.axis([0,8,0,5])
   ...: plt.title('A Multiseries Horizontal Bar Chart',fontsize=20)
   ...: plt.barh(index,values1,bw,color='b')
   ...: plt.barh(index+bw,values2,bw,color='g')
   ...: plt.barh(index+2*bw,values3,bw,color='r')
   ...: plt.yticks(index+0.4,['A','B','C','D','E'])
```
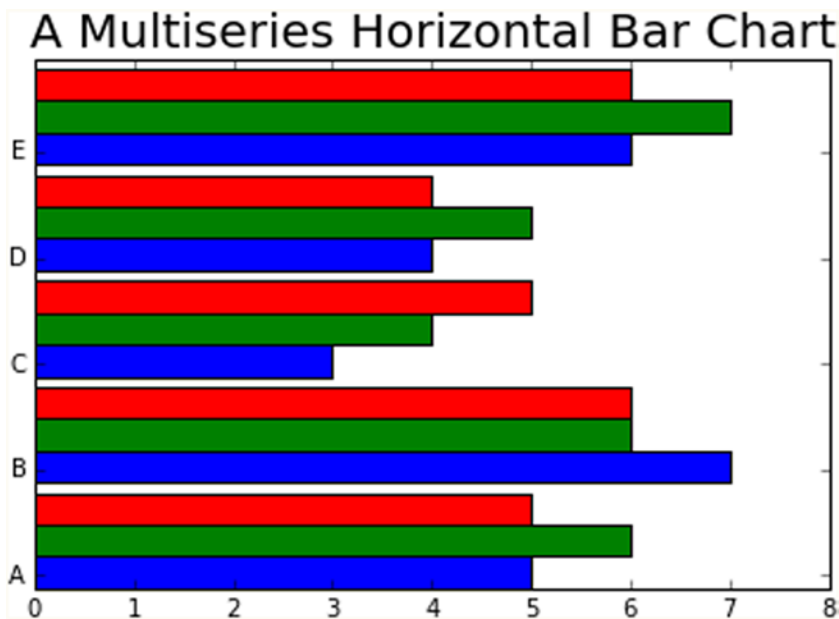
Figure 7-39. *A multiseries horizontal bar chart*

Multiseries Bar Chart with pandas DataFrame

As you saw in the line charts, the matplolib library also provides the ability to directly represent the DataFrame objects containing the results of the data analysis in the form of bar charts. And even here it does it quickly, directly, and automatically. The only thing you need to do is to use the *plot()* function applied to the DataFrame object specifying inside a kwarg called **kind** to which you have to assign the type of chart you want to represent, which in this case is **bar**. Thus, without specifying any other settings, you will get a bar chart as shown in Figure 7-40.

```
In [ ]: import matplotlib.pyplot as plt
    ...: import numpy as np
    ...: import pandas as pd
    ...: data = {'series1':[1,3,4,3,5],
                 'series2':[2,4,5,2,4],
                 'series3':[3,2,3,1,3]}
    ...: df = pd.DataFrame(data)
    ...: df.plot(kind='bar')
```
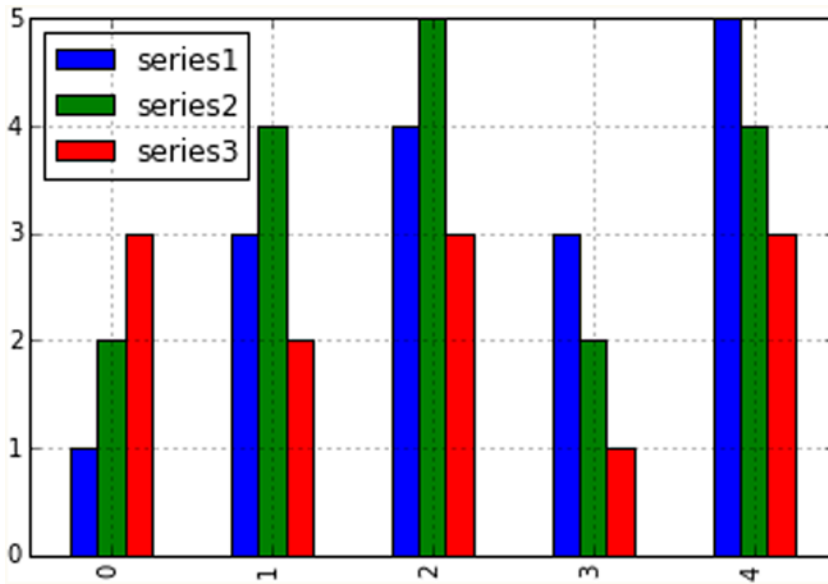
Figure 7-40. *The values in a DataFrame can be directly displayed as a bar chart*

However if you want to get more control, or if your case requires it, you can still extract portions of the DataFrame as NumPy arrays and use them as illustrated in the previous examples in this section. That is, by passing them separately as arguments to the matplotlib functions.

Moreover, regarding the horizontal bar chart the same rules can be applied, but remember to set **barh** as the value of the *kind* kwarg. You'll get a multiseries horizontal bar chart as shown in Figure 7-41.
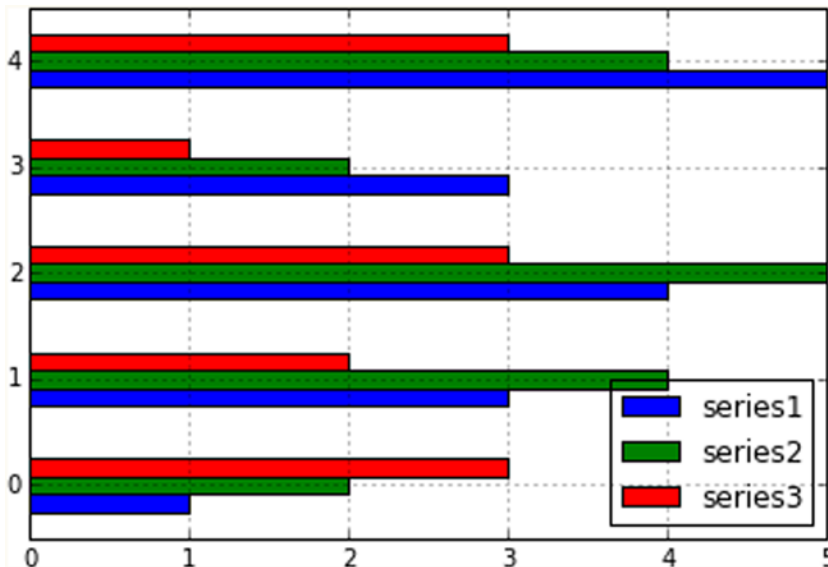
Figure 7-41. *A horizontal bar chart could be a valid alternative to visualize your DataFrame values*

Multiseries Stacked Bar Charts

Another form to represent a multiseries bar chart is in the stacked form, in which the bars are stacked one on the other. This is especially useful when you want to show the total value obtained by the sum of all the bars.

To transform a simple multiseries bar chart in a stacked one, you add the **bottom** kwarg to each *bar()* function. Each series must be assigned to the corresponding bottom kwarg. At the end you will obtain the stacked bar chart as shown in Figure 7-42.

```
In [ ]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: series1 = np.array([3,4,5,3])
   ...: series2 = np.array([1,2,2,5])
   ...: series3 = np.array([2,3,3,4])
   ...: index = np.arange(4)
   ...: plt.axis([0,4,0,15])
   ...: plt.bar(index,series1,color='r')
   ...: plt.bar(index,series2,color='b',bottom=series1)
   ...: plt.bar(index,series3,color='g',bottom=(series2+series1))
   ...: plt.xticks(index+0.4,['Jan15','Feb15','Mar15','Apr15'])
```
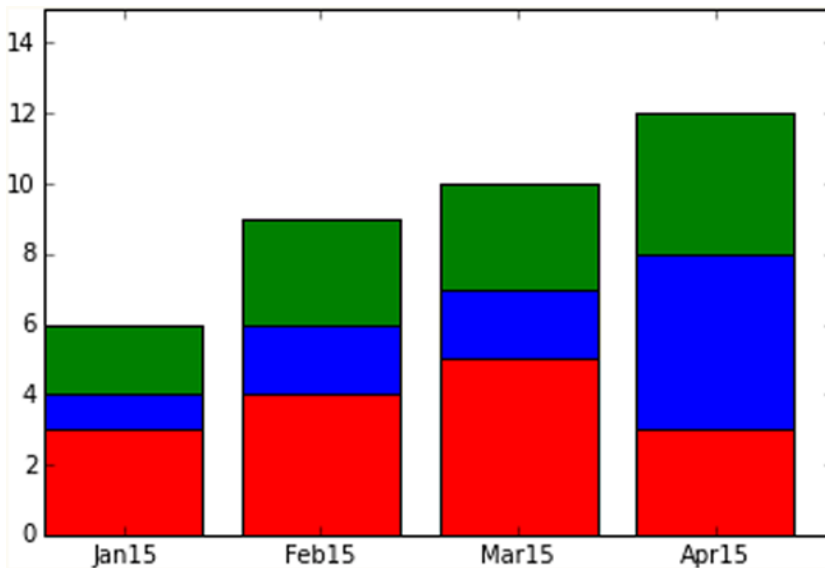

Figure 7-42. *A multiseries stacked bar*

Here too, in order to create the equivalent horizontal stacked bar chart, you need to replace the *bar()* function with *barh()* function, being careful to change other parameters as well. Indeed *xticks()* function should be replaced with the *yticks()* function because the labels of the categories now must be reported on the y axis. After doing all these changes you will obtain the horizontal stacked bar chart as shown in Figure 7-43.

```
In [ ]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: index = np.arange(4)
   ...: series1 = np.array([3,4,5,3])
```

215

```
...: series2 = np.array([1,2,2,5])
...: series3 = np.array([2,3,3,4])
...: plt.axis([0,15,0,4])
...: plt.title('A Multiseries Horizontal Stacked Bar Chart')
...: plt.barh(index,series1,color='r')
...: plt.barh(index,series2,color='g',left=series1)
...: plt.barh(index,series3,color='b',left=(series1+series2))
...: plt.yticks(index+0.4,['Jan15','Feb15','Mar15','Apr15'])
```


Figure 7-43. *A multiseries horizontal stacked bar chart*

So far the various series have been distinguished by using different colors. Another mode of distinction between the various series is for example, using hatches that allow to fill the various bars with strokes drawn in a different way. To do this, you have first to set the color of the bar as white and then you have to use the **hatch** kwarg to define how the hatch is to be set. The various hatches have codes distinguishable among these characters (|, /, -, \, *, -) corresponding to the line style filling the bar. The more the same symbol is replicated, the denser will be the lines forming the hatch. For example, /// is more dense than //, which is more dense than /. (See Figure 7-44).

```
In [ ]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: index = np.arange(4)
   ...: series1 = np.array([3,4,5,3])
   ...: series2 = np.array([1,2,2,5])
   ...: series3 = np.array([2,3,3,4])
   ...: plt.axis([0,15,0,4])
   ...: plt.title('A Multiseries Horizontal Stacked Bar Chart')
   ...: plt.barh(index,series1,color='w',hatch='xx')
   ...: plt.barh(index,series2,color='w',hatch='///', left=series1)
   ...: plt.barh(index,series3,color='w',hatch='\\\\\\',left=(series1+series2))
   ...: plt.yticks(index+0.4,['Jan15','Feb15','Mar15','Apr15'])
```

```
Out[453]:
([<matplotlib.axis.YTick at 0x2a9f0748>,
  <matplotlib.axis.YTick at 0x2a9e1f98>,
  <matplotlib.axis.YTick at 0x2ac06518>,
  <matplotlib.axis.YTick at 0x2ac52128>],
 <a list of 4 Text yticklabel objects>)
```
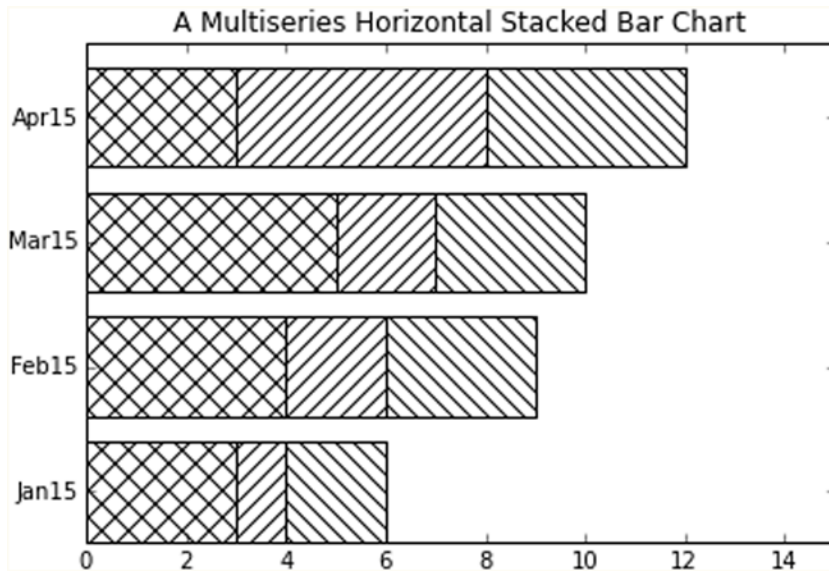

Figure 7-44. *The stacked bars can be distinguished by their hatches*

Stacked Bar Charts with pandas DataFrame

Also with regard to stacked bar charts, it is very simple to directly represent the values contained in the DataFrame object by using the plot() function. You need only to add as argument the stacked kwarg set to True (Figure 7-45).

```
In [ ]: import matplotlib.pyplot as plt
   ...: import pandas as pd
   ...: data = {'series1':[1,3,4,3,5],
                'series2':[2,4,5,2,4],
                'series3':[3,2,3,1,3]}
   ...: df = pd.DataFrame(data)
   ...: df.plot(kind='bar', stacked=True)
Out[5]: <matplotlib.axes._subplots.AxesSubplot at 0xcda8f98>
```
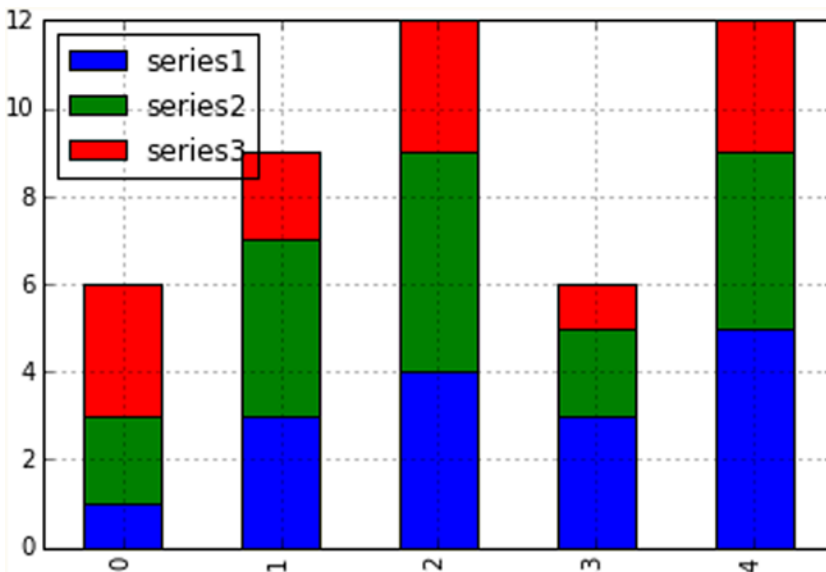
Figure 7-45. *The values of a DataFrame can be directly displayed as a stacked bar chart*

Other Bar Chart Representations

Another type of very useful representation is that of a bar chart for comparison, where two series of values sharing the same categories are compared by placing the bars in opposite directions along the y axis. In order to do this, you have to put the y values of one of the two series in a negative form. Also in this example, you will see the possibility of coloring the edge of the bars and their inner color in a different way. In fact, you can do this by setting the two different colors on two specific kwargs: **facecolor** and **edgecolor**.

Furthermore, in this example, you will see how to add the y value with a label at the end of each bar. This could be useful to increase the readability of the bar chart. You can do this using a *for* loop in which the *text()* function will show the y value. You can adjust the label position with the two kwargs **ha** and **va**, which control the horizontal and vertical alignment, respectively. The result will be the chart shown in the Figure 7-46.

```
In [ ]: import matplotlib.pyplot as plt
   ...: x0 = np.arange(8)
   ...: y1 = np.array([1,3,4,6,4,3,2,1])
   ...: Y2 = np.array([1,2,5,4,3,3,2,1])
   ...: plt.ylim(-7,7)
   ...: plt.bar(x0,y1,0.9,facecolor='r',edgecolor='w')
   ...: plt.bar(x0,-y2,0.9,facecolor='b',edgecolor='w')
   ...: plt.xticks(())
   ...: plt.grid(True)
   ...: for x, y in zip(x0, y1):
          plt.text(x + 0.4, y + 0.05, '%d' % y, ha='center', va= 'bottom')
   ...:
   ...: for x, y in zip(x0, y2):
          plt.text(x + 0.4, -y - 0.05, '%d' % y, ha='center', va= 'top')
```
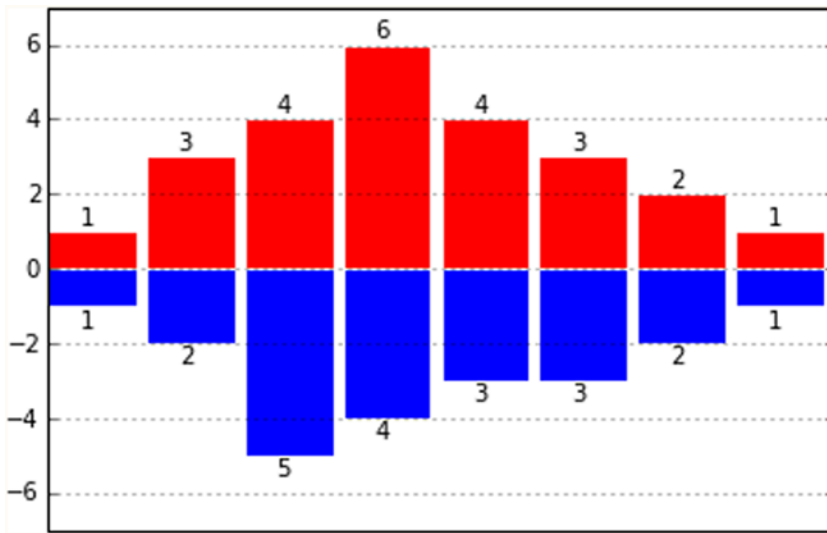
Figure 7-46. *Two series can be compared using this kind of bar chart*

Pie Charts

An alternative way to display data to the bar charts is the pie chart, easily obtainable using the **pie()** function.

Even for this type of function, you pass as the main argument a list containing the values to be displayed. I chose the percentages (their sum is 100) but actually you can use any kind of value. It will be the *pie()* function to inherently calculate the percentage occupied by each value.

Also with this type of representation, you need to define some key features making use of the kwargs. For example, if you want to define the sequence of the colors, which will be assigned to the sequence of input values correspondingly, you have to use the **colors** kwarg. Therefore you have to assign a list of strings, each containing the name of the desired color. Another important feature is to add labels to each slice of the pie. To do this, you have to use the **labels** kwarg to which you will assign a list of strings containing the labels to be displayed in sequence.

In addition, in order to draw the pie chart in a perfectly spherical way you have to add the *axis()* function at the end, specifying the string *'equal'* as an argument. You will get a pie chart as shown in Figure 7-47.

```
In [ ]: import matplotlib.pyplot as plt
   ...: labels = ['Nokia','Samsung','Apple','Lumia']
   ...: values = [10,30,45,15]
   ...: colors = ['yellow','green','red','blue']
   ...: plt.pie(values,labels=labels,colors=colors)
   ...: plt.axis('equal')
```
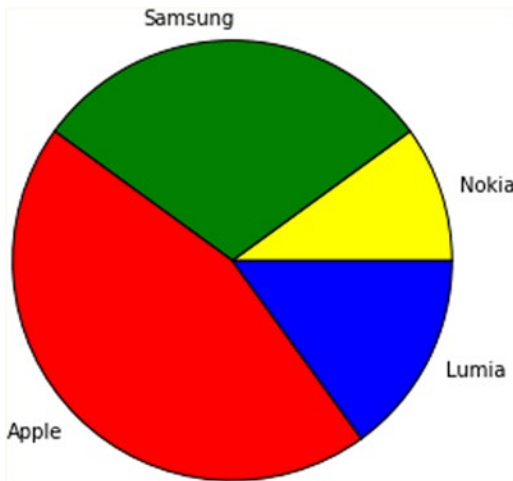
Figure 7-47. *A very simple pie chart*

To add complexity to the pie chart, you can draw it with a slice extracted from the pie. This is usually done when you want to focus on a specific slice. In this case, for example, you would highlight the slice referring to Nokia. In order to do this there is a special kwarg named **explode**. It is nothing but a sequence of float values of 0 or 1, where 1 corresponds to the fully extended slice and 0 corresponds to slices completely in the pie. All intermediate values correspond to an intermediate degree of extraction (see Figure 7-48).

You can also add a title to the pie chart with the *title()* function. You can also adjust the angle of rotation of the pie by adding the **startangle** kwarg that takes an integer value between 0 and 360, which are the degrees of rotation precisely (0 is the default value).

The modified chart should appear as in Figure 7-48.

```
In [ ]: import matplotlib.pyplot as plt
   ...: labels = ['Nokia','Samsung','Apple','Lumia']
   ...: values = [10,30,45,15]
   ...: colors = ['yellow','green','red','blue']
   ...: explode = [0.3,0,0,0]
   ...: plt.title('A Pie Chart')
   ...: plt.pie(values,labels=labels,colors=colors,explode=explode,startangle=180)
   ...: plt.axis('equal')
```
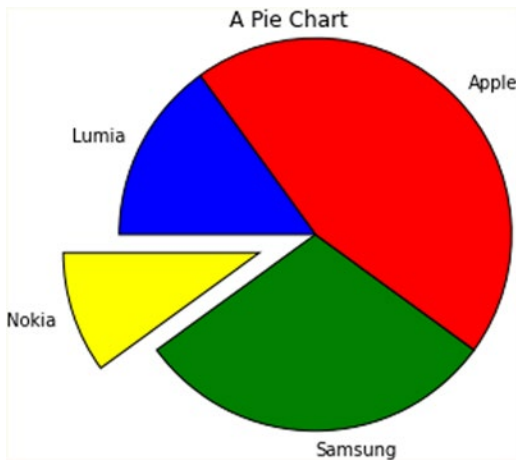
Figure 7-48. *A more advanced pie chart*

But the possible additions that you can insert in a pie chart do not end here. For example, a pie chart does not have axes with ticks and so it is difficult to imagine the perfect percentage represented by each slice. To overcome this, you will use the **autopct** kwarg that adds to the center of each slice a text label which shows the corresponding value.

If you wish to make it an even more appealing image, you can add a shadow with the **shadow** kwarg setting it to True. In the end you will get a pie chart as shown in Figure 7-49.

```
In [ ]: import matplotlib.pyplot as plt
   ...: labels = ['Nokia','Samsung','Apple','Lumia']
   ...: values = [10,30,45,15]
   ...: colors = ['yellow','green','red','blue']
   ...: explode = [0.3,0,0,0]
   ...: plt.title('A Pie Chart')
   ...: plt.pie(values,labels=labels,colors=colors,explode=explode,
       shadow=True,autopct='%1.1f%%',startangle=180)
   ...: plt.axis('equal')
```
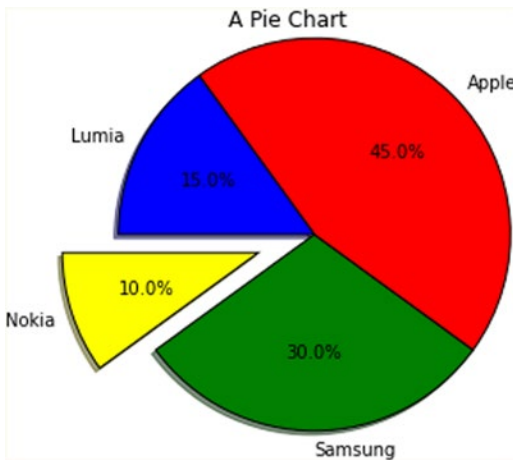
Figure 7-49. *An even more advanced pie chart*

Pie Charts with pandas DataFrame

Even for the pie chart, you can represent the values contained within a DataFrame object. In this case, however, the pie chart can represent only one series at a time, so in this example you will display only the values of the first series specifying *df['series1']*. You have to specify the type of chart you want to represent through the *kind* kwarg in the *plot()* function, which in this case is *'pie'*. Furthermore, because you want to represent a pie chart as perfectly circular, it is necessary that you add the **figsize** kwarg. At the end you will obtain a pie chart as shown in Figure 7-50.

```
In [ ]: import matplotlib.pyplot as plt
   ...: import pandas as pd
   ...: data = {'series1':[1,3,4,3,5],
               'series2':[2,4,5,2,4],
               'series3':[3,2,3,1,3]}
   ...: df = pd.DataFrame(data)
   ...: df['series1'].plot(kind='pie',figsize=(6,6))
Out[14]: <matplotlib.axes._subplots.AxesSubplot at 0xe1ba710>
```
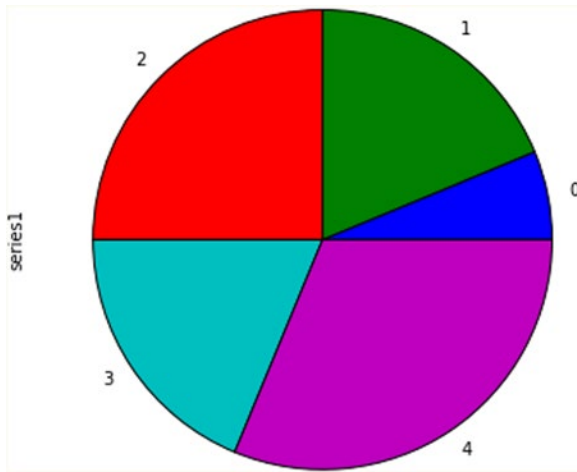
Figure 7-50. *The values in a pandas DataFrame can be directly drawn as a pie chart*

Advanced Charts

In addition to the more classical charts such as bar charts or pie charts, it is easy to have the need to represent your results in an alternative way. On the Internet and in various publications there are many examples in which many alternative graphics solutions are discussed and proposed, some really brilliant and captivating. This section will only show some graphic representations; a more detailed discussion about this topic is beyond the purpose of this book. You can use this section as an introduction to a world that is constantly expanding: data visualization.

Contour Plot

A quite common type of chart in the scientific world is the **contour plot** or **contour map**. This visualization is in fact suitable for displaying three-dimensional surfaces through a contour map composed of curves closed showing the points on the surface that are located at the same level, or that have the same z value.

Although visually the contour plot is a very complex structure, its implementation is not so difficult, thanks to the matplotlib library. First, you need the function $z = f(x, y)$ for generating a three-dimensional surface. Then, once you have defined a range of values x, y that will define the area of the map to be displayed, you can calculate the z values for each pair (x, y), applying the function $f(x, y)$ just defined in order to obtain a matrix of z values. Finally, thanks to the *contour()* function you can generate the contour map of the surface. It is often desirable to add also a color map along with a contour map. That is, the areas delimited by the curves of level are filled by a color gradient, defined by a color map. For example, as in Figure 7-51, you may indicate negative values with increasingly dark shades of blue, and move to yellow and then red with the increase of the positive values.

```
In [ ]: import matplotlib.pyplot as plt
    ...: import numpy as np
    ...: dx = 0.01; dy = 0.01
    ...: x = np.arange(-2.0,2.0,dx)
    ...: y = np.arange(-2.0,2.0,dy)
    ...: X,Y = np.meshgrid(x,y)
    ...: def f(x,y):
```

223

```
        return (1 - y**5 + x**5)*np.exp(-x**2-y**2)
...: C = plt.contour(X,Y,f(X,Y),8,colors='black')
...: plt.contourf(X,Y,f(X,Y),8)
...: plt.clabel(C, inline=1, fontsize=10)
```

The standard color gradient (color map) is represented in Figure 7-51. Actually you choose among a large number of color maps available just specifying them with the **cmap** kwarg.

Furthermore, when you have to deal with this kind of representation, adding a color scale as a reference to the side of the graph is almost a must. This is possible by simply adding the function **colorbar()** at the end of the code. In Figure 7-52 you can see another example of color map which starts from black, to pass through red, then yellow until reaching white for the highest values. This color map is **plt.cm.hot**.
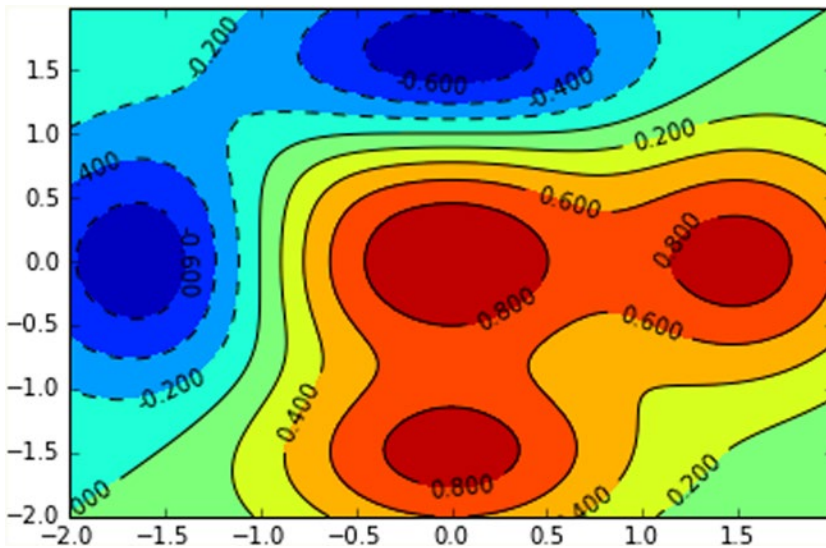
Figure 7-51. *A contour map can describe the z values of a surface*

```
In [ ]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: dx = 0.01; dy = 0.01
   ...: x = np.arange(-2.0,2.0,dx)
   ...: y = np.arange(-2.0,2.0,dy)
   ...: X,Y = np.meshgrid(x,y)
   ...:
   ...: C = plt.contour(X,Y,f(X,Y),8,colors='black')
   ...: plt.contourf(X,Y,f(X,Y),8,cmap=plt.cm.hot)
   ...: plt.clabel(C, inline=1, fontsize=10)
   ...: plt.colorbar()
```
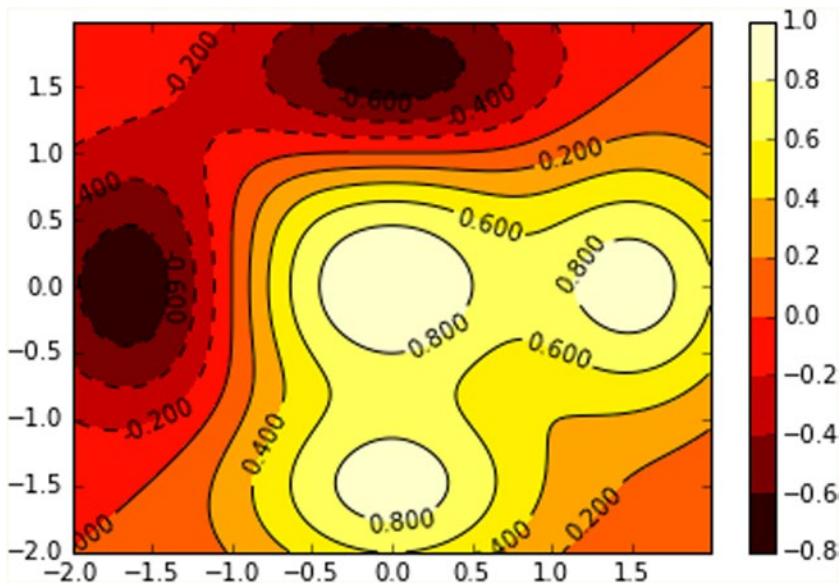
Figure 7-52. *The "hot" color map gradient gives an attractive look to the contour map*

Polar Chart

Another type of advanced chart that is having some success is the **polar chart**. This type of chart is characterized by a series of sectors which extend radially; each of these areas will occupy a certain angle. Thus you can display two different values assigning them to the magnitudes that characterize the polar chart: the extension of the radius r and the angle θ occupied by the sector. These in fact are the polar coordinates (r, θ), an alternative way of representing functions at the coordinate axes. From the graphical point of view you could imagine it as a kind of chart that has characteristics both of the pie chart and of the bar chart. In fact as the pie chart, the angle of each sector gives percentage information represented by that category with respect to the total. As for the bar chart, the radial extension is the numerical value of that category.

So far we have used the standard set of colors using single characters as the color code (e.g., 'r' to indicate red). In fact you can use any sequence of colors you want. You have to define a list of string values which contain RGB codes in the #rrggbb format corresponding to the colors you want.

Oddly, for getting a polar chart you have to use the *bar()* function in which you pass the list containing the angles θ and a list of the radial extension of each sector. The result will be a polar chart as shown in Figure 7-53.

```
In [ ]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: N = 8
   ...: theta = np.arange(0.,2 * np.pi, 2 * np.pi / N)
   ...: radii = np.array([4,7,5,3,1,5,6,7])
   ...: plt.axes([0.025, 0.025, 0.95, 0.95], polar=True)
   ...: colors = np.array(['#4bb2c5', '#c5b47f', '#EAA228', '#579575', '#839557', '#958c12',
       '#953579', '#4b5de4'])
   ...: bars = plt.bar(theta, radii, width=(2*np.pi/N), bottom=0.0, color=colors)
```
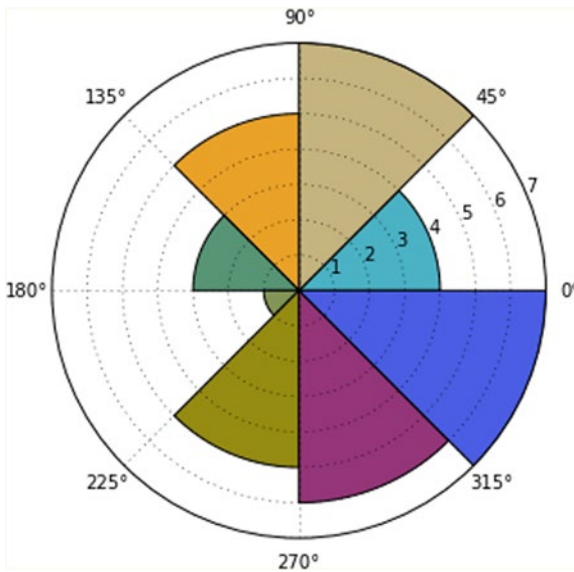
Figure 7-53. *A polar chart*

In this example you have defined the sequence of colors using the format #rrggbb, but you can specify a sequence of colors as strings with their actual name (see Figure 7-54).

```
In [ ]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: N = 8
   ...: theta = np.arange(0.,2 * np.pi, 2 * np.pi / N)
   ...: radii = np.array([4,7,5,3,1,5,6,7])
   ...: plt.axes([0.025, 0.025, 0.95, 0.95], polar=True)
   ...: colors = np.array(['lightgreen', 'darkred', 'navy', 'brown', 'violet', 'plum',
        'yellow', 'darkgreen'])
   ...: bars = plt.bar(theta, radii, width=(2*np.pi/N), bottom=0.0, color=colors)
```
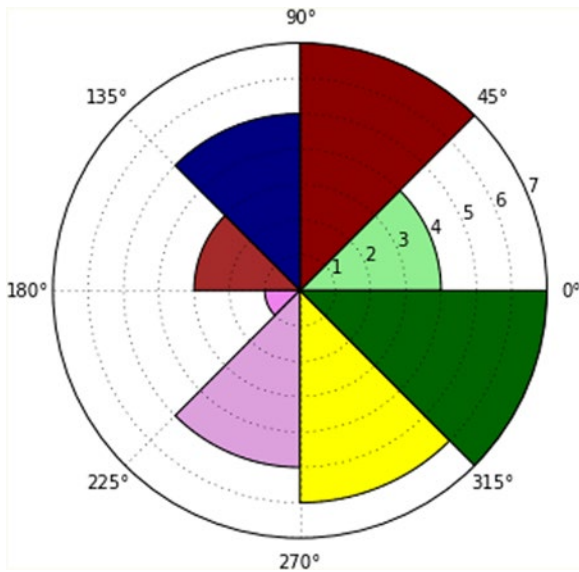
Figure 7-54. *A polar chart with another sequence of colors*

mplot3d

The **mplot3d** toolkit is included with all standard installations of matplotlib and allows you to extend the capabilities of visualization to 3D data. If the figure is displayed in a separate window you can rotate the axes of the three-dimensional representation with the mouse.

With this package you are still using the Figure object, only that instead of the Axes object you will define a new kind object, **Axes3D,** introduced by this toolkit. Thus, you need to add a new import to the code, if you want to use the Axes3D object.

```
from mpl_toolkits.mplot3d import Axes3D
```

3D Surfaces

In the previous section you used the contour plot to represent the three-dimensional surfaces through the level lines. Using the mplot3D package, surfaces can be drawn directly in 3D. In this example you will use again the same function $z = f(x, y)$ you have used in the contour map.

Once you have calculated the meshgrid you can view the surface with the *plot_surface()* function. A three-dimensional surface of blue color will appear as in Figure 7-55.

```
In [ ]: from mpl_toolkits.mplot3d import Axes3D
   ...: import matplotlib.pyplot as plt
   ...: fig = plt.figure()
   ...: ax = Axes3D(fig)
   ...: X = np.arange(-2,2,0.1)
   ...: Y = np.arange(-2,2,0.1)
   ...: X,Y = np.meshgrid(X,Y)
   ...: def f(x,y):
   ...:     return (1 - y**5 + x**5)*np.exp(-x**2-y**2)
   ...: ax.plot_surface(X,Y,f(X,Y), rstride=1, cstride=1)
```

227
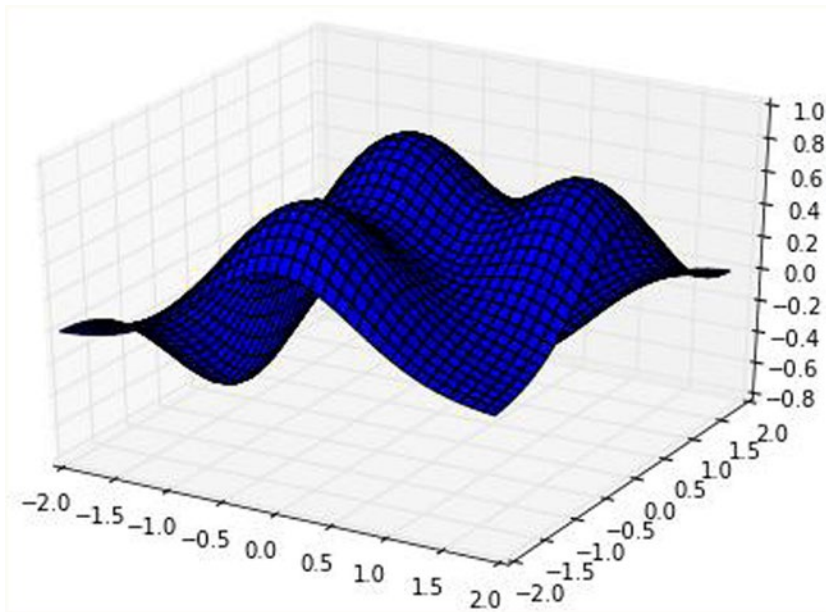
Figure 7-55. *A 3D surface can be represented with the mplot3d toolkit*

A 3D surface stands out most by changing the color map, for example by setting the **cmap** kwarg. You can also rotate the surface using the **view_init()** function. In fact, this function adjusts the view point from which you see the surface, changing the two kwarg **elev** and **azim**. Through their combination you can get the surface displayed from any angle. The first kwarg adjusts the height at which the surface is seen, while azim adjusts the angle of rotation of the surface.

For instance, you can change the color map using *plt.cm.hot* and moving the view point to elev=30 and azim=125. The result is shown in Figure 7-56.

```
In [ ]: from mpl_toolkits.mplot3d import Axes3D
   ...: import matplotlib.pyplot as plt
   ...: fig = plt.figure()
   ...: ax = Axes3D(fig)
   ...: X = np.arange(-2,2,0.1)
   ...: Y = np.arange(-2,2,0.1)
   ...: X,Y = np.meshgrid(X,Y)
   ...: def f(x,y):
            return (1 - y**5 + x**5)*np.exp(-x**2-y**2)
   ...: ax.plot_surface(X,Y,f(X,Y), rstride=1, cstride=1, cmap=plt.cm.hot)
   ...: ax.view_init(elev=30,azim=125)
```
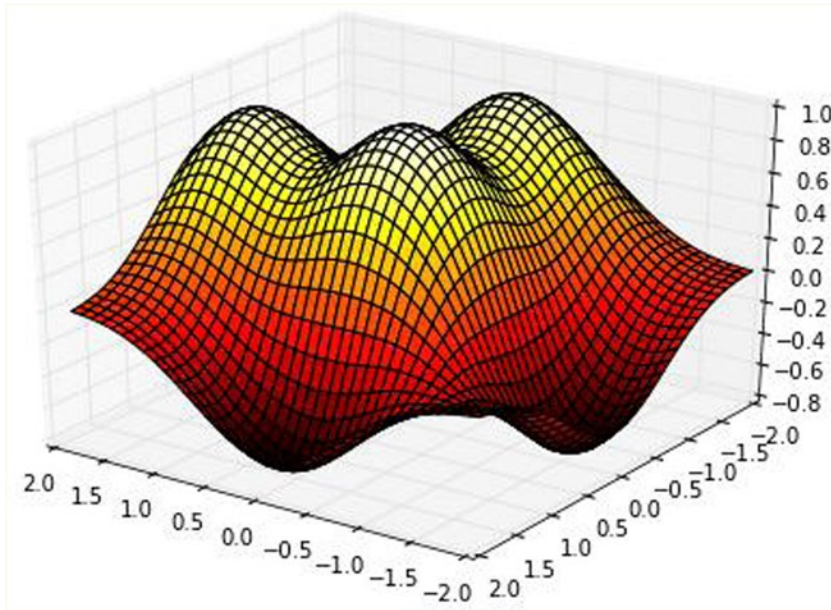
Figure 7-56. *The 3D surface can be rotated and observed from a higher viewpoint*

Scatter Plot in 3D

However the mode most used among all 3D views remains the 3D scatter plot. With this type of visualization you can identify if the points follow particular trends, but above all if they tend to cluster.

In this case you will use the *scatter()* function as the 2D case but applied on Axes3D object. Doing this you can visualize different series, expressed by the calls to the *scatter()* function, all together in the same 3D representation (see Figure 7-57).

```
In [ ]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: from mpl_toolkits.mplot3d import Axes3D
   ...: xs = np.random.randint(30,40,100)
   ...: ys = np.random.randint(20,30,100)
   ...: zs = np.random.randint(10,20,100)
   ...: xs2 = np.random.randint(50,60,100)
   ...: ys2 = np.random.randint(30,40,100)
   ...: zs2 = np.random.randint(50,70,100)
   ...: xs3 = np.random.randint(10,30,100)
   ...: ys3 = np.random.randint(40,50,100)
   ...: zs3 = np.random.randint(40,50,100)
   ...: fig = plt.figure()
   ...: ax = Axes3D(fig)
   ...: ax.scatter(xs,ys,zs)
   ...: ax.scatter(xs2,ys2,zs2,c='r',marker='^')
   ...: ax.scatter(xs3,ys3,zs3,c='g',marker='*')
   ...: ax.set_xlabel('X Label')
   ...: ax.set_ylabel('Y Label')
   ...: ax.set_zlabel('Z Label')
Out[34]: <matplotlib.text.Text at 0xe1c2438>
```
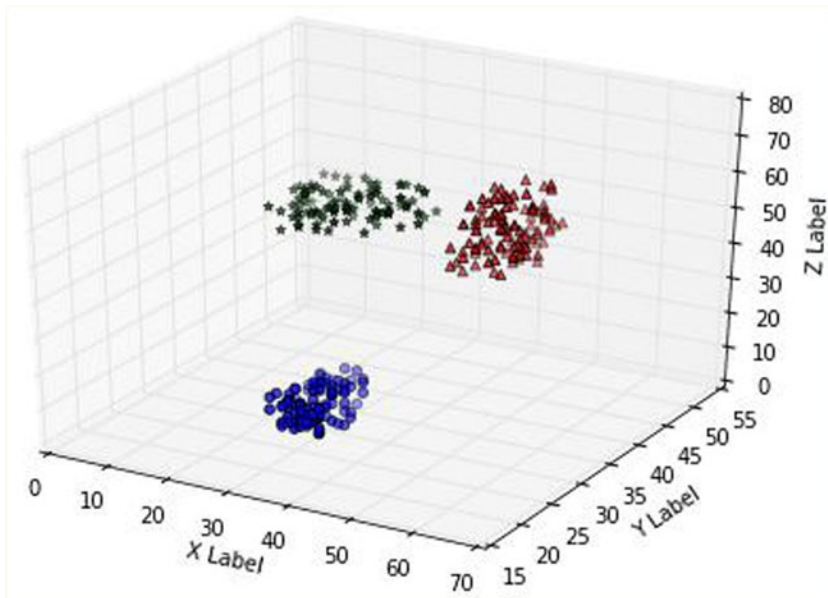
Figure 7-57. *This 3D scatter plot shows three different clusters*

Bar Chart 3D

Another type of 3D plot widely used in the data analysis is the 3D bar chart. Also in this case using the *bar()* function applied to the object Axes3D. If you define multiple series, you can accumulate several calls to the bar() function in the same 3D visualization (See Figure 7-58).

```
In [ ]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: from mpl_toolkits.mplot3d import Axes3D
   ...: x = np.arange(8)
   ...: y = np.random.randint(0,10,8)
   ...: y2 = y + np.random.randint(0,3,8)
   ...: y3 = y2 + np.random.randint(0,3,8)
   ...: y4 = y3 + np.random.randint(0,3,8)
   ...: y5 = y4 + np.random.randint(0,3,8)
   ...: clr = ['#4bb2c5', '#c5b47f', '#EAA228', '#579575', '#839557', '#958c12', '#953579',
'#4b5de4']
   ...: fig = plt.figure()
   ...: ax = Axes3D(fig)
   ...: ax.bar(x,y,0,zdir='y',color=clr)
   ...: ax.bar(x,y2,10,zdir='y',color=clr)
   ...: ax.bar(x,y3,20,zdir='y',color=clr)
   ...: ax.bar(x,y4,30,zdir='y',color=clr)
   ...: ax.bar(x,y5,40,zdir='y',color=clr)
   ...: ax.set_xlabel('X Axis')
   ...: ax.set_ylabel('Y Axis')
   ...: ax.set_zlabel('Z Axis')
   ...: ax.view_init(elev=40)
```
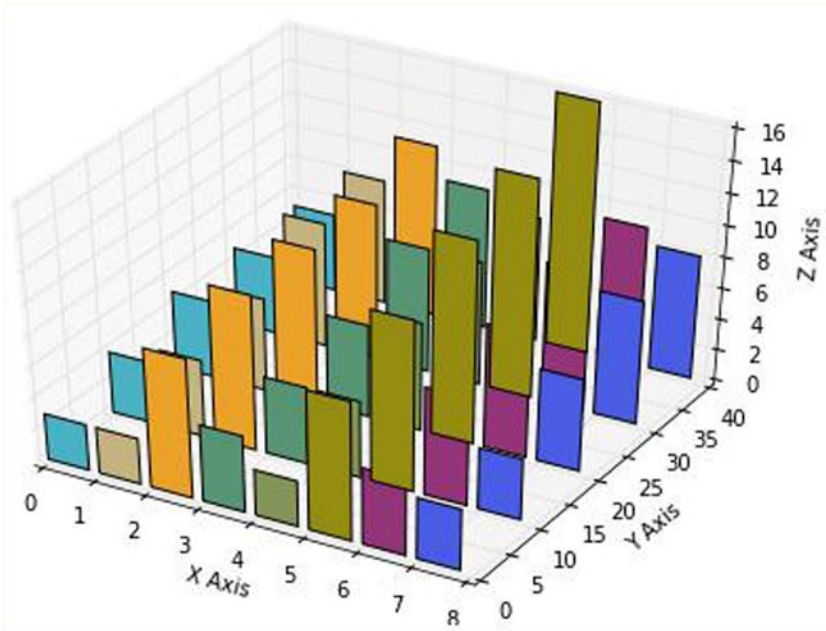
Figure 7-58. *A bar chart 3D*

Multi-Panel Plots

So far you've had the chance to see different ways of representing data through a chart. You saw the chance to see more charts in the same figure by separating it with subplots. In this section you will further deepen your understanding of this topic, analyzing more complex cases.

Display Subplots within Other Subplots

Now an even more advanced method will be explained: the ability to view charts within others, enclosed within frames. Since we are talking of frames, i.e., Axes objects, you will need to separate the main Axes (i.e., the general chart) from the frame you want to add that will be another instance of Axes. To do this you use the *figures()* function to get the Figure object on which you will define two different Axes objects using the **add_axes()** function. See the result of this example in Figure 7-59.

```
In [ ]: import matplotlib.pyplot as plt
   ...: fig = plt.figure()
   ...: ax = fig.add_axes([0.1,0.1,0.8,0.8])
   ...: inner_ax = fig.add_axes([0.6,0.6,0.25,0.25])
```
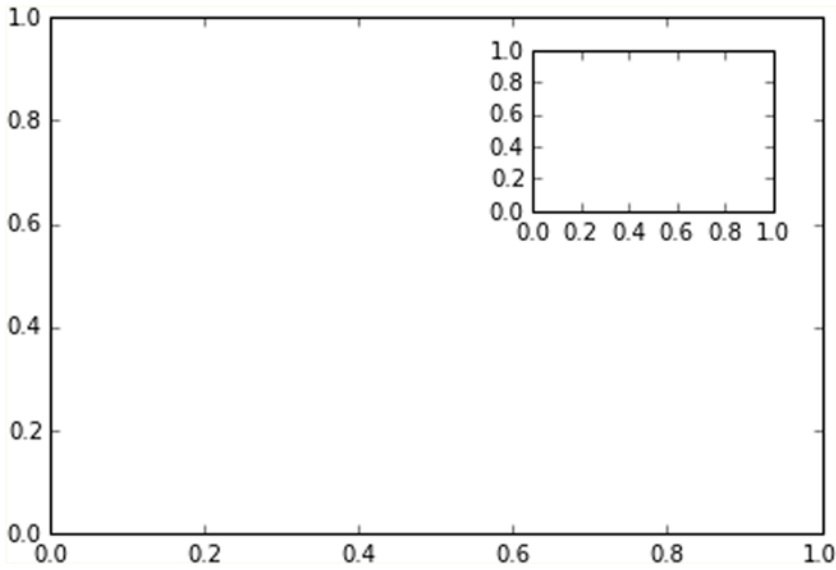
Figure 7-59. *A subplot is displayed between another plot*

To better understand the effect of this mode of display, you can fill the previous Axes with real data as shown in Figure 7-60.

```
In [ ]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: fig = plt.figure()
   ...: ax = fig.add_axes([0.1,0.1,0.8,0.8])
   ...: inner_ax = fig.add_axes([0.6,0.6,0.25,0.25])
   ...: x1 = np.arange(10)
   ...: y1 = np.array([1,2,7,1,5,2,4,2,3,1])
   ...: x2 = np.arange(10)
   ...: y2 = np.array([1,3,4,5,4,5,2,6,4,3])
   ...: ax.plot(x1,y1)
   ...: inner_ax.plot(x2,y2)
Out[95]: [<matplotlib.lines.Line2D at 0x14acf6d8>]
```
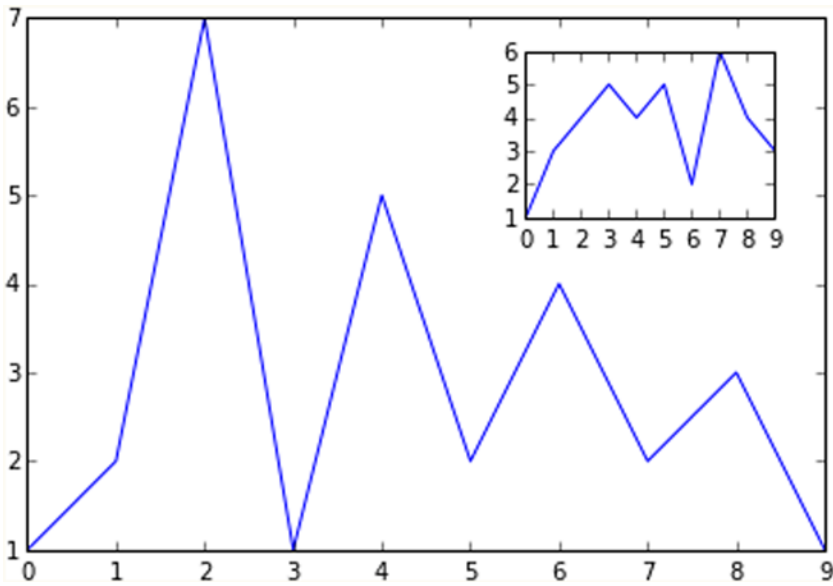
Figure 7-60. *A more realistic visualization of a subplot within another plot*

Grids of Subplots

You have already seen the creation of subplots and it is quite simple using the *subplots()* function to add them by dividing a plot into sectors. Matplotlib allows you to manage even more complex cases using another function called **GridSpec()**. This subdivision allows splitting the drawing area into a grid of sub-areas, to which you can assign one or more of them to each subplot, so that in the end you can obtain subplots with different sizes and orientations, as you can see in Figure 7-61.

```
In [ ]: import matplotlib.pyplot as plt
   ...: gs = plt.GridSpec(3,3)
   ...: fig = plt.figure(figsize=(6,6))
   ...: fig.add_subplot(gs[1,:2])
   ...: fig.add_subplot(gs[0,:2])
   ...: fig.add_subplot(gs[2,0])
   ...: fig.add_subplot(gs[:2,2])
   ...: fig.add_subplot(gs[2,1:])
Out[97]: <matplotlib.axes._subplots.AxesSubplot at 0x12717438>
```
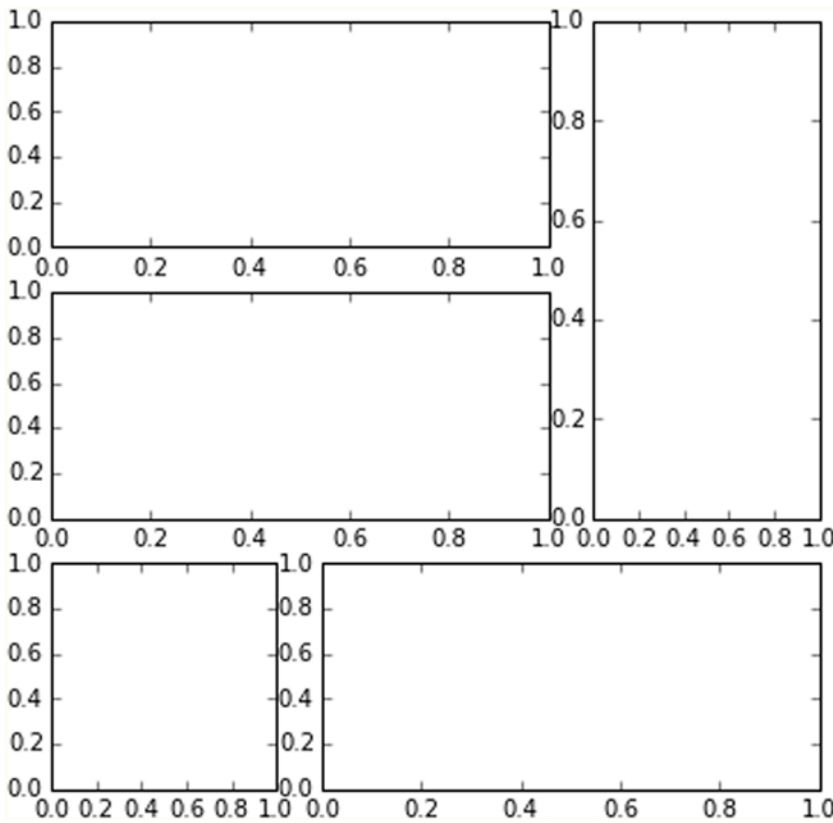
Figure 7-61. *Subplots with different sizes can be defined on a grid of sub-areas*

Now that it's clear to you how to manage the grid by assigning the various sectors to subplot, it's time to see how to use these subplots. In fact, you can use the Axes object returned by each *add_subplot()* function to call the *plot()* function to draw the corresponding plot (see Figure 7-62).

```
In [ ]: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...: gs = plt.GridSpec(3,3)
   ...: fig = plt.figure(figsize=(6,6))
   ...: x1 = np.array([1,3,2,5])
   ...: y1 = np.array([4,3,7,2])
   ...: x2 = np.arange(5)
   ...: y2 = np.array([3,2,4,6,4])
   ...: s1 = fig.add_subplot(gs[1,:2])
   ...: s1.plot(x,y,'r')
   ...: s2 = fig.add_subplot(gs[0,:2])
```

```
...: s2.bar(x2,y2)
...: s3 = fig.add_subplot(gs[2,0])
...: s3.barh(x2,y2,color='g')
...: s4 = fig.add_subplot(gs[:2,2])
...: s4.plot(x2,y2,'k')
...: s5 = fig.add_subplot(gs[2,1:])
...: s5.plot(x1,y1,'b^',x2,y2,'yo')
```
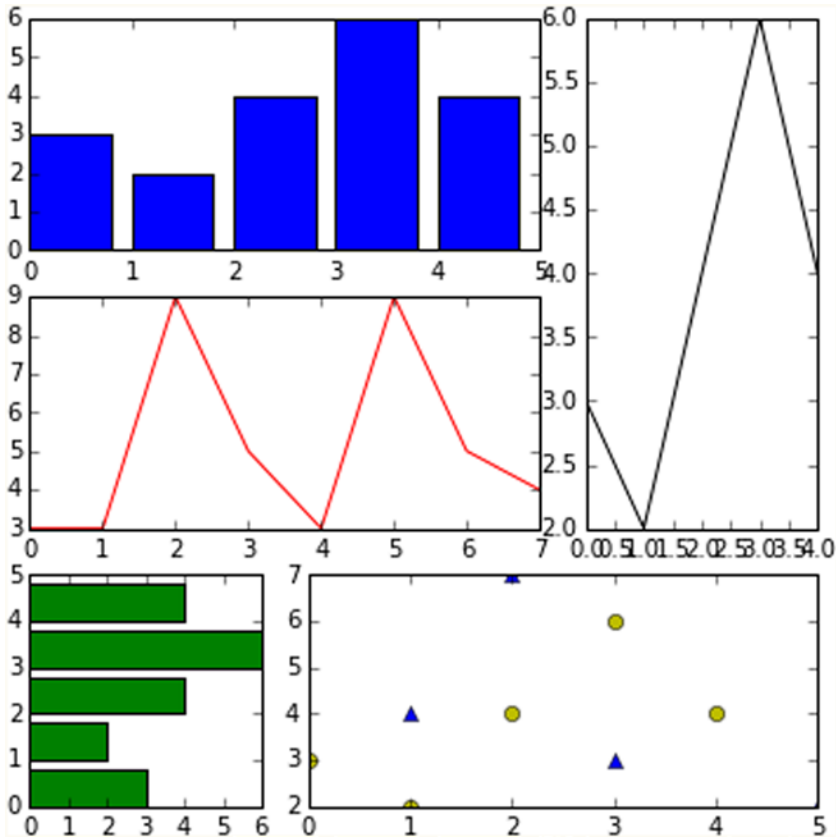

Figure 7-62. *A grid of subplots can display many plots at the same time*

Conclusions

In this chapter you received all the fundamental aspects of the matplotlib library, and through a series of examples you have mastered the basic tools for handling data visualization. You have become familiar with various examples of how to develop different types of charts with a few lines of code.

With this chapter, we conclude the part about the libraries that provide the basic tools to perform data analysis. In the next chapter, you will begin to treat topics most closely related to data analysis.

235

■ ■ ■

Machine Learning with scikit-learn

In the chain of processes that make up the data analysis, the construction phase of predictive models and their validation are done by a powerful library called **scikit-learn**. In this chapter you will see some examples that will illustrate the basic construction of predictive models with some different methods.

The scikit-learn Library

scikit-learn is a Python module that integrates many of machine learning algorithms. This library was developed initially by Cournapeu in 2007, but the first real release was in 2010.

This library is part of the SciPy (Scientific Python) group, a set of libraries created for scientific computing and especially for data analysis, many of which are discussed in this book. Generally these libraries are defined as **SciKits**, hence the first part of the name of this library. The second part of the library's name is derived from **Machine Learning**, the discipline pertaining to this library.

Machine Learning

Machine Learning is a discipline that deals with the study of methods for pattern recognition in data sets undergoing data analysis. In particular, it deals with the development of algorithms that learn from data and make predictions. Each methodology is based on building a specific model.

There are very many methods that belong to the learning machine, each with its unique characteristics, which are specific to the nature of the data and the predictive model that you want to build. The choice of which method is to be applied is called **learning problem**.

The data to be subjected to a pattern in the learning phase can be arrays composed by a single value per element, or by a multivariate value. These values are often referred to as **features** or **attributes**.

Supervised and Unsupervised Learning

Depending on the type of the data and the model to be built, you can separate the learning problems into two broad categories:

Supervised learning. They are the methods in which the training set contains additional attributes that you want to predict (**target**). Thanks to these values, you can instruct the model to provide similar values when you have to submit new values (**test set**).

- **Classification**: the data in the training set belong to two or more classes or categories; then, the data, already being labeled, allow us to teach the system to recognize the characteristics that distinguish each class. When you will need to consider a new value unknown to the system, the system will evaluate its class according to its characteristics.

- **Regression**: when the value to be predicted is a continuous variable. The simplest case to understand is when you want to find the line which describes the trend from a series of points represented in a scatterplot.

Unsupervised learning. These are the methods in which the training set consists of a series of input values x without any corresponding target value.

- **Clustering**: the goal of these methods is to discover groups of similar examples in a dataset.
- **Dimensionality reduction**: reduction of a high-dimensional dataset to one with only two or three dimensions is useful not just for data visualization, but for converting data of very high dimensionality into data of much lower dimensionality such that each of the lower dimensions conveys much more information.

In addition to these two main categories, there is a further group of methods which have the purpose of validation and evaluation of the models.

Training Set and Testing Set

Machine learning enables learning some properties by a model from a data set and applying them to new data. This is because a common practice in machine learning is to evaluate an algorithm. This valuation consists of splitting the data into two parts, one called the **training set**, with which we will learn the properties of the data, and the other called the **testing set**, on which to test these properties.

Supervised Learning with scikit-learn

In this chapter you will see a number of examples of **supervised learning**.

- Classification, using the Iris Dataset
  - K-Nearest Neighbors Classifier
  - Support Vector Machines (SVC)
- Regression, using the Diabetes Dataset
  - Linear Regression
  - Support Vector Machines (SVR)

Supervised learning consists of learning possible patterns between two or more features reading values from a training set; the learning is possible because the training set contains known results (**target** or **labels**). All models in scikit-learn are referred to as **supervised estimators**, using the **fit(x, y)** function that makes their training. *x* comprises the features observed, while *y* indicates the target. Once the estimator has carried out the training, it will be able to predict the value of *y* for any new observation *x* not labeled. This operation will make it through the **predict(x)** function.

The Iris Flower Dataset

The **Iris Flower Dataset** is a particular dataset used for the first time by Sir Ronald Fisher in 1936. It is often also called Anderson Iris Dataset, after the person who collected the data directly measuring the size of the various parts of the iris flowers. In this dataset, data from three different species of iris (Iris silky, virginica Iris, and Iris versicolor) are collected and exactly these data correspond to the length and width of the sepals and the length and width of the petals (see Figure 8-1).
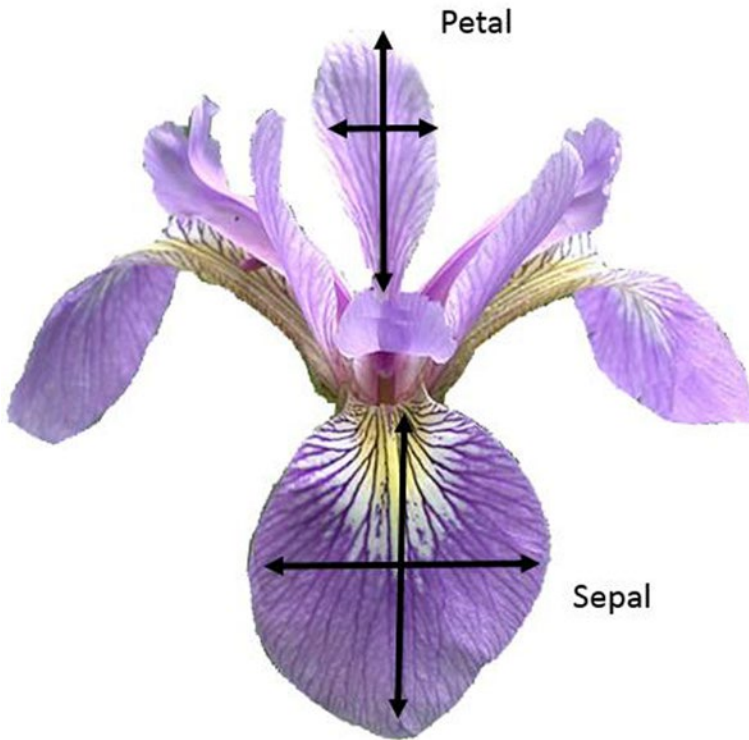
Figure 8-1. *Iris versicolor and the petal and sepal width and length*

This dataset is currently being used as a good example to utilize for many types of analysis, in particular as regards the problems of **classification** that can be approached by means of machine learning methodologies. It is no coincidence then that this dataset is provided along with the scikit-learn library as a 150x4 NumPy array.

Now you will study this dataset in detail importing it in the IPython QtConsole or in a normal Python session.

```
In [ ]: from sklearn import datasets
   ...: iris = datasets.load_iris()
```

In this way you loaded all the data and metadata concerning the Iris Dataset in the *iris* variable. In order to see the values of the data collected in it, it is sufficient to call the attribute **data** of the variable *iris*.

```
In [ ]: iris.data
Out[ ]:
array([[ 5.1,  3.5,  1.4,  0.2],
       [ 4.9,  3. ,  1.4,  0.2],
       [ 4.7,  3.2,  1.3,  0.2],
       [ 4.6,  3.1,  1.5,  0.2],
       ...
```

As you can see you will get an array of 150 elements, each containing 4 numeric values: the size of sepals and petals respectively.

To know instead what kind of flower belongs each item you will refer to the **target** attribute.

```
In [ ]: iris.target
Out[ ]:
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

You obtain 150 items with three possible integer values (0, 1 and 2) which correspond to the three species of iris. To know the correspondence between the species and number you have to call the **target_names** attribute.

```
In [ ]: iris.target_names
Out[ ]:
array(['setosa', 'versicolor', 'virginica'],
      dtype='|S10')
```

To better understand this data set you can use the matplotlib library, using the techniques you learned in Chapter 7. Therefore create a scatterplot that displays the three different species in three different colors. X-axis will represent the length of the sepal while the y-axis will represent the width of the sepal.

```
 In [ ]: import matplotlib.pyplot as plt
   ...: import matplotlib.patches as mpatches
   ...: from sklearn import datasets
   ...:
   ...: iris = datasets.load_iris()
   ...: x = iris.data[:,0]  #X-Axis - sepal length
   ...: y = iris.data[:,1]  #Y-Axis - sepal length
   ...: species = iris.target     #Species
   ...:
   ...: x_min, x_max = x.min() - .5,x.max() + .5
   ...: y_min, y_max = y.min() - .5,y.max() + .5
   ...:
   ...: #SCATTERPLOT
   ...: plt.figure()
   ...: plt.title('Iris Dataset - Classification By Sepal Sizes')
   ...: plt.scatter(x,y, c=species)
   ...: plt.xlabel('Sepal length')
   ...: plt.ylabel('Sepal width')
   ...: plt.xlim(x_min, x_max)
   ...: plt.ylim(y_min, y_max)
   ...: plt.xticks(())
   ...: plt.yticks(())
```

As a result you get the scatterplot as shown in Figure 8-2. In blue are represented the Iris setosa, flowers, green ones are the Iris versicolor, and red ones are the Iris virginica.

Figure 8-2. *The different species of irises are shown with different colors*

From Figure 8-2 you can see how the Iris setosa features differ from the other two, forming a cluster of blue dots separate from the others.

Try to follow the same procedure, but this time using the other two variables, that is the measure of the length and width of the petal. You can use the same code above only by changing some values.

```
In [ ]: import matplotlib.pyplot as plt
   ...: import matplotlib.patches as mpatches
   ...: from sklearn import datasets
   ...:
   ...: iris = datasets.load_iris()
   ...: x = iris.data[:,2]  #X-Axis - petal length
   ...: y = iris.data[:,3]  #Y-Axis - petal length
   ...: species = iris.target     #Species
   ...:
   ...: x_min, x_max = x.min() - .5,x.max() + .5
   ...: y_min, y_max = y.min() - .5,y.max() + .5
   ...: #SCATTERPLOT
   ...: plt.figure()
   ...: plt.title('Iris Dataset - Classification By Petal Sizes', size=14)
   ...: plt.scatter(x,y, c=species)
   ...: plt.xlabel('Petal length')
   ...: plt.ylabel('Petal width')
   ...: plt.xlim(x_min, x_max)
   ...: plt.ylim(y_min, y_max)
   ...: plt.xticks(())
   ...: plt.yticks(())
```

The result is the scatterplot shown in Figure 8-3. In this case the division between the three species is much more evident. As you can see you have three different clusters.
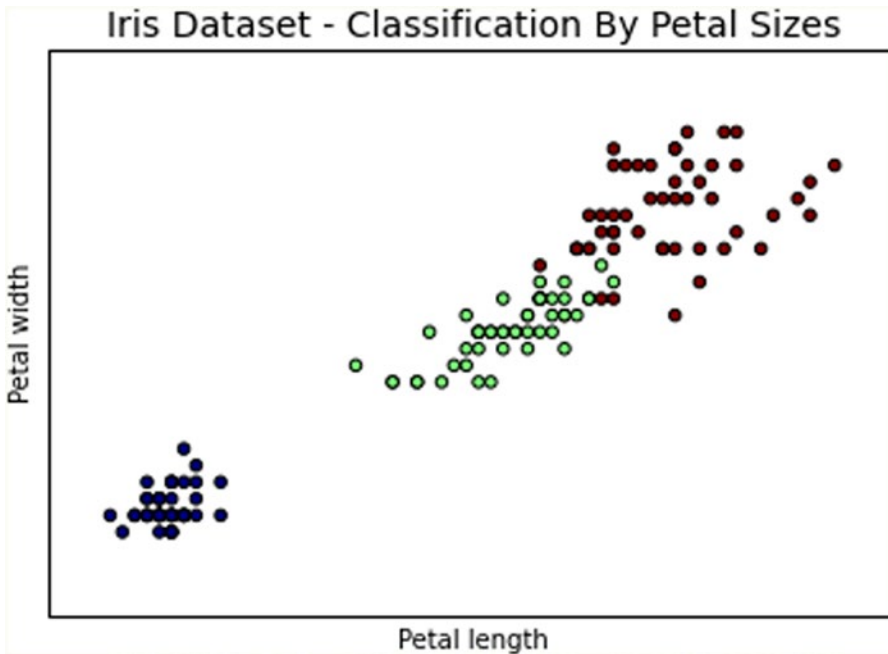
Figure 8-3. *The different species of irises are shown with different colors*

The PCA Decomposition

You have seen how the three species could be characterized taking into account four measurements of the petals and sepals size. We represented two scatterplots, one for the petals and one for sepals, but how can you can unify the whole thing? Four dimensions are a problem that even a Scatterplot 3D is not able to solve.

In this regard a special technique called **Principal Component Analysis (PCA)** has been developed. This technique allows you to reduce the number of dimensions of a system keeping all the information for the characterization of the various points, the new dimensions generated are called **principal components**. In our case, so you can reduce the system from 4 to 3 dimensions and then plot the results within a 3D scatterplot. In this way you can use measures both of sepals and of petals for characterizing the various species of iris of the test elements in the dataset.

The Scikit-learn function which allows you to do the dimensional reduction, is the **fit_transform()** function which belongs to the **PCA** object. In order to use it, first you need to import the PCA module **sklearn.decomposition**. Then you have to define the object constructor using PCA() defining the number of new dimensions (principal components) as value of the **n_components** option. In your case it is 3. Finally you have to call the **fit_transform()** function passing the four-dimensional Iris Dataset as argument.

```
from sklearn.decomposition import PCA
x_reduced = PCA(n_components=3).fit_transform(iris.data)
```

In addition, in order to visualize the new values you will use a scatterplot 3D using the **mpl_toolkits. mplot3d** module of matplotlib. If you don't remember how to do it, see the Scatterplot 3D section in Chapter 7.

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn import datasets
from sklearn.decomposition import PCA

iris = datasets.load_iris()
x = iris.data[:,1]  #X-Axis - petal length
y = iris.data[:,2]  #Y-Axis - petal length
species = iris.target      #Species
x_reduced = PCA(n_components=3).fit_transform(iris.data)

#SCATTERPLOT 3D
fig = plt.figure()
ax = Axes3D(fig)
ax.set_title('Iris Dataset by PCA', size=14)
ax.scatter(x_reduced[:,0],x_reduced[:,1],x_reduced[:,2], c=species)
ax.set_xlabel('First eigenvector')
ax.set_ylabel('Second eigenvector')
ax.set_zlabel('Third eigenvector')
ax.w_xaxis.set_ticklabels(())
ax.w_yaxis.set_ticklabels(())
ax.w_zaxis.set_ticklabels(())
```

The result will be a scatterplot as shown in Figure 8-4. The three species of iris are well characterized with respect to each other to form a cluster.
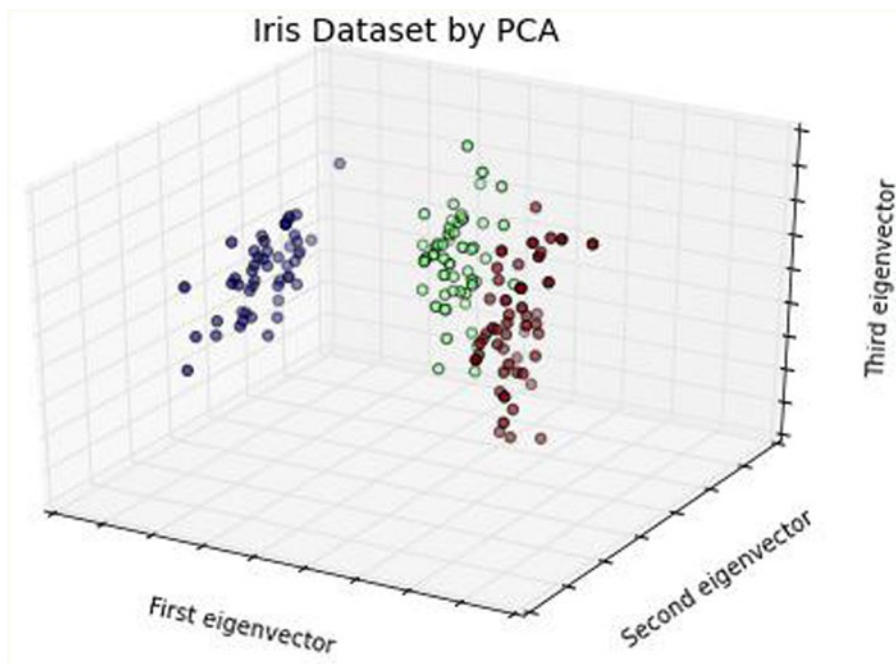
Figure 8-4. *3D scatterplot with three clusters representative of each species of iris*

243

K-Nearest Neighbors Classifier

Now, you will perform a **classification,** and to do this operation with the scikit-learn library you need a **classifier**.

Given a new measurement of an iris flower, the task of the classifier is to figure out to which of the three species it belongs. The simplest possible classifier is the **nearest neighbor**. This algorithm will search within the training set the observation that most closely approaches the new test sample.

A very important thing to consider at this point are the concepts of **training set** and **testing set** (already seen in Chapter 1). Indeed, if you have only a single dataset of data, it is important not to use the same data both for the test and for the training. In this regard, the elements of the dataset are divided into two parts, one dedicated to train the algorithm and the other to perform its validation.

Thus, before proceeding further you have to divide your *Iris* Dataset into two parts. However, it is wise to randomly mix the array elements and then make the division. In fact, often the data may have been collected in a particular order, and in your case the Iris Dataset contains items sorted by species. So to make a blending of elements of the data set you will use a NumPy function called **random.permutation()**. The mixed dataset consists of 150 different observations; the first 140 will be used as training set, the remaining 10 as test set.

```
import numpy as np
from sklearn import datasets
np.random.seed(0)
iris = datasets.load_iris()
x = iris.data
y = iris.target
i = np.random.permutation(len(iris.data))
x_train = x[i[:-10]]
y_train = y[i[:-10]]
x_test = x[i[-10:]]
y_test = y[i[-10:]]
```

Now you can apply the K-Nearest Neighbor algorithm. Import the KneighborsClassifier, call the constructor of the classifier, and then train it with the fit() function.

```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier()
knn.fit(x_train,y_train)
Out[86]:
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
          metric_params=None, n_neighbors=5, p=2, weights='uniform')
```

Now that you have a predictive model which consists of the *knn* classifier, trained by 140 observations, you will find out how it is valid. The classifier should correctly predict the species of iris of the 10 observations of the test set. In order to obtain the prediction you have to use the **predict()** function, which will be applied directly to the predictive model *knn*. Finally, you will compare the results predicted with the actual observed contained in *y_test*.

```
knn.predict(x_test)
Out[100]: array([1, 2, 1, 0, 0, 0, 2, 1, 2, 0])
y_test
Out[101]: array([1, 1, 1, 0, 0, 0, 2, 1, 2, 0])
```

You can see that you obtained a 10% error. Now you can visualize all this using decision boundaries in a space represented by the 2D scatterplot of sepals.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn import datasets
from sklearn.neighbors import KNeighborsClassifier
iris = datasets.load_iris()
x = iris.data[:,:2]      #X-Axis - sepal length-width
y = iris.target          #Y-Axis - species

x_min, x_max = x[:,0].min() - .5,x[:,0].max() + .5
y_min, y_max = x[:,1].min() - .5,x[:,1].max() + .5

#MESH
cmap_light = ListedColormap(['#AAAAFF','#AAFFAA','#FFAAAA'])
h = .02
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
knn = KNeighborsClassifier()
knn.fit(x,y)
Z = knn.predict(np.c_[xx.ravel(),yy.ravel()])
Z = Z.reshape(xx.shape)
plt.figure()
plt.pcolormesh(xx,yy,Z,cmap=cmap_light)

#Plot the training points
plt.scatter(x[:,0],x[:,1],c=y)
plt.xlim(xx.min(),xx.max())
plt.ylim(yy.min(),yy.max())

Out[120]: (1.5, 4.900000000000003)
```

You get a subdivision of the scatterplot in decision boundaries, as shown in Figure 8-5.
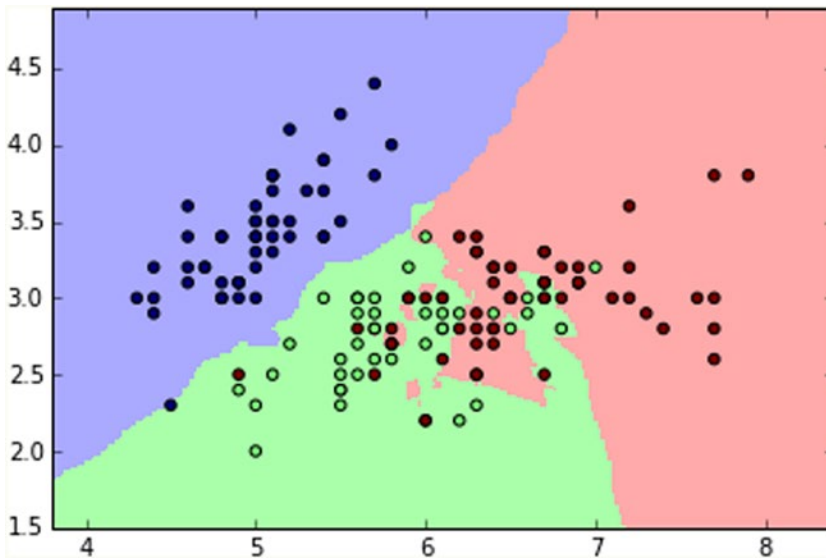
Figure 8-5. *The three decision boundaries are represented by three different colors*

You can do the same thing considering the size of the petals.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn import datasets
from sklearn.neighbors import KNeighborsClassifier
iris = datasets.load_iris()
x = iris.data[:,2:4]      #X-Axis - petals length-width
y = iris.target           #Y-Axis - species

x_min, x_max = x[:,0].min() - .5,x[:,0].max() + .5
y_min, y_max = x[:,1].min() - .5,x[:,1].max() + .5

#MESH
cmap_light = ListedColormap(['#AAAAFF','#AAFFAA','#FFAAAA'])
h = .02
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
knn = KNeighborsClassifier()
knn.fit(x,y)
Z = knn.predict(np.c_[xx.ravel(),yy.ravel()])
Z = Z.reshape(xx.shape)
plt.figure()
plt.pcolormesh(xx,yy,Z,cmap=cmap_light)

#Plot the training points
plt.scatter(x[:,0],x[:,1],c=y)
plt.xlim(xx.min(),xx.max())
plt.ylim(yy.min(),yy.max())

Out[126]: (-0.40000000000000002, 2.9800000000000031)
```

As shown in Figure 8-6, you will have the corresponding decision boundaries regarding the characterization of iris flowers taking into account the size of the petals.
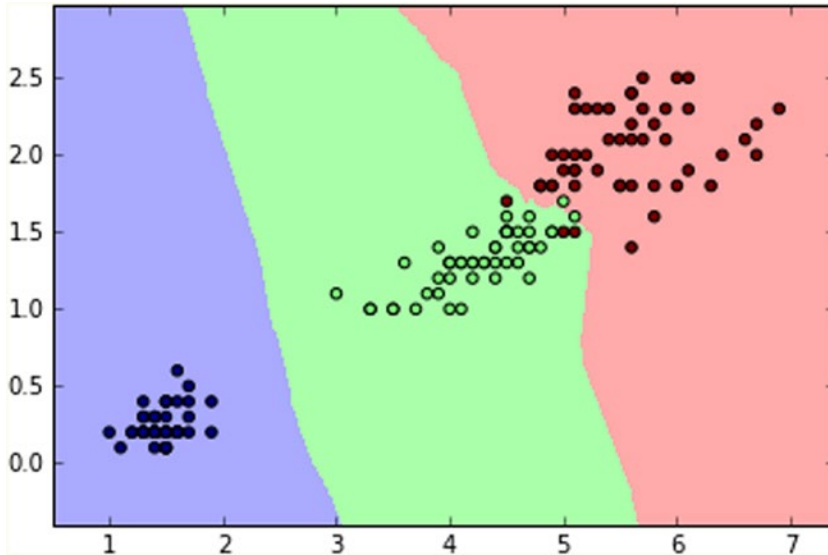
Figure 8-6. *The decision boundaries on a 2D scatterplot describing the petal sizes*

Diabetes Dataset

Among the various datasets available within the scikit-learn library, there is the diabetes dataset. This dataset was used for the first time in 2004 (*Annals of Statistics*, by Efron, Hastie, Johnston, and Tibshirani). Since then it has become an example widely used to study various predictive models and their effectiveness.

To upload the data contained in this dataset, before you have to import the **datasets** module of the scikit-learn library and then you call the **load_diabetes()** function to load the data set into a variable that will be called just *diabetes*.

```
In [ ]: from sklearn import datasets
   ...: diabetes = datasets.load_diabetes()
```

This dataset contains physiological data collected on 442 patients and as corresponding target an indicator of the disease progression after a year. The physiological data occupy the first 10 columns with values that indicate respectively:

- Age

- Sex

- Body Mass Index

- Blood Pressure

- S1, S2, S3, S4, S5, S6 (six blood serum measurements)

247

These measurements can be obtained by calling the **data** attribute. But going to check the values in the dataset you will find values very different from what you would have expected. For example, we look at the 10 values for the first patient.

```
diabetes.data[0]
Out[ ]:
array([ 0.03807591,  0.05068012,  0.06169621,  0.02187235, -0.0442235 ,
       -0.03482076, -0.04340085, -0.00259226,  0.01990842, -0.01764613])
```

These values are in fact the result of a processing. Each of the ten values was mean centered and subsequently scaled by the standard deviation times the number of the samples. Checking will reveal that the sum of squares of each column is equal to 1. Try doing this calculation with the age measurements; you will obtain a value very close to 1.

```
np.sum(diabetes.data[:,0]**2)
Out[143]: 1.0000000000000746
```

Even though these values are normalized and therefore difficult to read, they continue to express the 10 physiological characteristics and therefore have not lost their value or statistical information.

As for the indicators of the progress of the disease, that is, the values that must correspond to the results of your predictions, these are obtainable by means of the **target** attribute.

```
diabetes.target
Out[146]:
array([ 151.,    75.,   141.,   206.,   135.,    97.,   138.,    63.,   110.,
        310.,   101.,    69.,   179.,   185.,   118.,   171.,   166.,   144.,
         97.,   168.,    68.,    49.,    68.,   245.,   184.,   202.,   137
        . . .
```

You obtain a series of 442 integer values between 25 and 346.

Linear Regression: The Least Square Regression

Linear regression is a procedure that uses data contained in the training set to build a linear model. The most simple is based on the equation of a rect with the two parameters *a* and *b* to characterize it. These parameters will be calculated so as to make the sum of squared residuals as small as possible.

$$y = a*x + c$$

In this expression, *x* is the training set, *y* is the target, *b* is the slope, and *c* is the intercept of the rect represented by the model. In scikit-learn, to use the predictive model for the linear regression you must import **linear_model** module and then use the manufacturer **LinearRegression()** constructor for creating the predictive model, which you call **linreg**.

```
from sklearn import linear_model
linreg = linear_model.LinearRegression()
```

For practicing with an example of linear regression you can use the diabetes dataset described earlier. First you will need to break the 442 patients into a training set (composed of the first 422 patients) and a test set (the last 20 patients).

```
from sklearn import datasets
diabetes = datasets.load_diabetes()
x_train = diabetes.data[:-20]
y_train = diabetes.target[:-20]
x_test = diabetes.data[-20:]
y_test = diabetes.target[-20:]
```

Now, apply the training set to the predictive model through the use of **fit()** function.

```
linreg.fit(x_train,y_train)
Out[ ]: LinearRegression(copy_X=True, fit_intercept=True, normalize=False)
```

Once the model is trained you can get the ten *b* coefficients calculated for each physiological variable, using the **coef_** attribute of the predictive model.

```
linreg.coef_
Out[164]:
array([  3.03499549e-01,  -2.37639315e+02,   5.10530605e+02,
         3.27736980e+02,  -8.14131709e+02,   4.92814588e+02,
         1.02848452e+02,   1.84606489e+02,   7.43519617e+02,
         7.60951722e+01])
```

If you apply the test set to the **linreg** prediction model you will get a series of targets to be compared with the values actually observed.

```
linreg.predict(x_test)
Out[ ]:
array([ 197.61846908,  155.43979328,  172.88665147,  111.53537279,
        164.80054784,  131.06954875,  259.12237761,  100.47935157,
        117.0601052 ,  124.30503555,  218.36632793,   61.19831284,
        132.25046751,  120.3332925 ,   52.54458691,  194.03798088,
        102.57139702,  123.56604987,  211.0346317 ,   52.60335674])
```

```
y_test
Out[ ]:
array([ 233.,   91.,  111.,  152.,  120.,   67.,  310.,   94.,  183.,
         66.,  173.,   72.,   49.,   64.,   48.,  178.,  104.,  132.,
        220.,   57.])
```

However, a good indicator of what prediction should be perfect is the **variance**. The more the variance is close to 1 the more the prediction is perfect.

```
linreg.score(x_test, y_test)
Out[ ]: 0.58507530226905713
```

Now you will start with the linear regression taking into account a single physiological factor, for example, you can start from the age.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import linear_model
from sklearn import datasets
```

```
diabetes = datasets.load_diabetes()
x_train = diabetes.data[:-20]
y_train = diabetes.target[:-20]
x_test = diabetes.data[-20:]
y_test = diabetes.target[-20:]

x0_test = x_test[:,0]
x0_train = x_train[:,0]
x0_test = x0_test[:,np.newaxis]
x0_train = x0_train[:,np.newaxis]
linreg = linear_model.LinearRegression()
linreg.fit(x0_train,y_train)
y = linreg.predict(x0_test)
plt.scatter(x0_test,y_test,color='k')
plt.plot(x0_test,y,color='b',linewidth=3)
```

```
Out[230]: [<matplotlib.lines.Line2D at 0x380b1908>]
```

Figure 8-7 shows the blue line representing the linear correlation between the ages of patients and the disease progression.
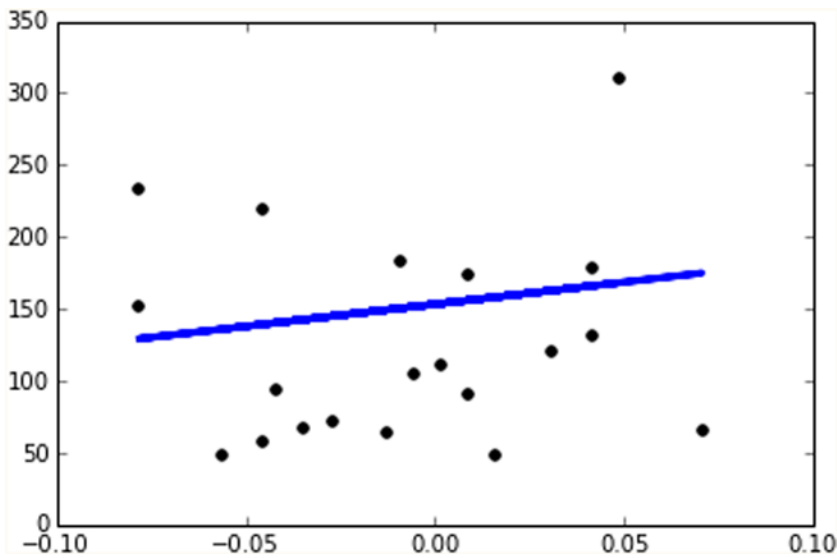
Figure 8-7. *A linear regression represents a linear correlation between a feature and the targets*

Actually, you have 10 physiological factors within the diabetes dataset. Therefore, to have a more complete picture of all the training set, you can make a linear regression for every physiological feature, creating 10 models and seeing the result for each of them through a linear chart.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import linear_model
from sklearn import datasets
diabetes = datasets.load_diabetes()
x_train = diabetes.data[:-20]
y_train = diabetes.target[:-20]
x_test = diabetes.data[-20:]
y_test = diabetes.target[-20:]
plt.figure(figsize=(8,12))
for f in range(0,10):
   xi_test = x_test[:,f]
   xi_train = x_train[:,f]
   xi_test = xi_test[:,np.newaxis]
   xi_train = xi_train[:,np.newaxis]
   linreg.fit(xi_train,y_train)
   y = linreg.predict(xi_test)
   plt.subplot(5,2,f+1)
   plt.scatter(xi_test,y_test,color='k')
   plt.plot(xi_test,y,color='b',linewidth=3)
```

Figure 8-8 shows ten linear charts, each of which represents the correlation between a physiological factor and the progression of diabetes.
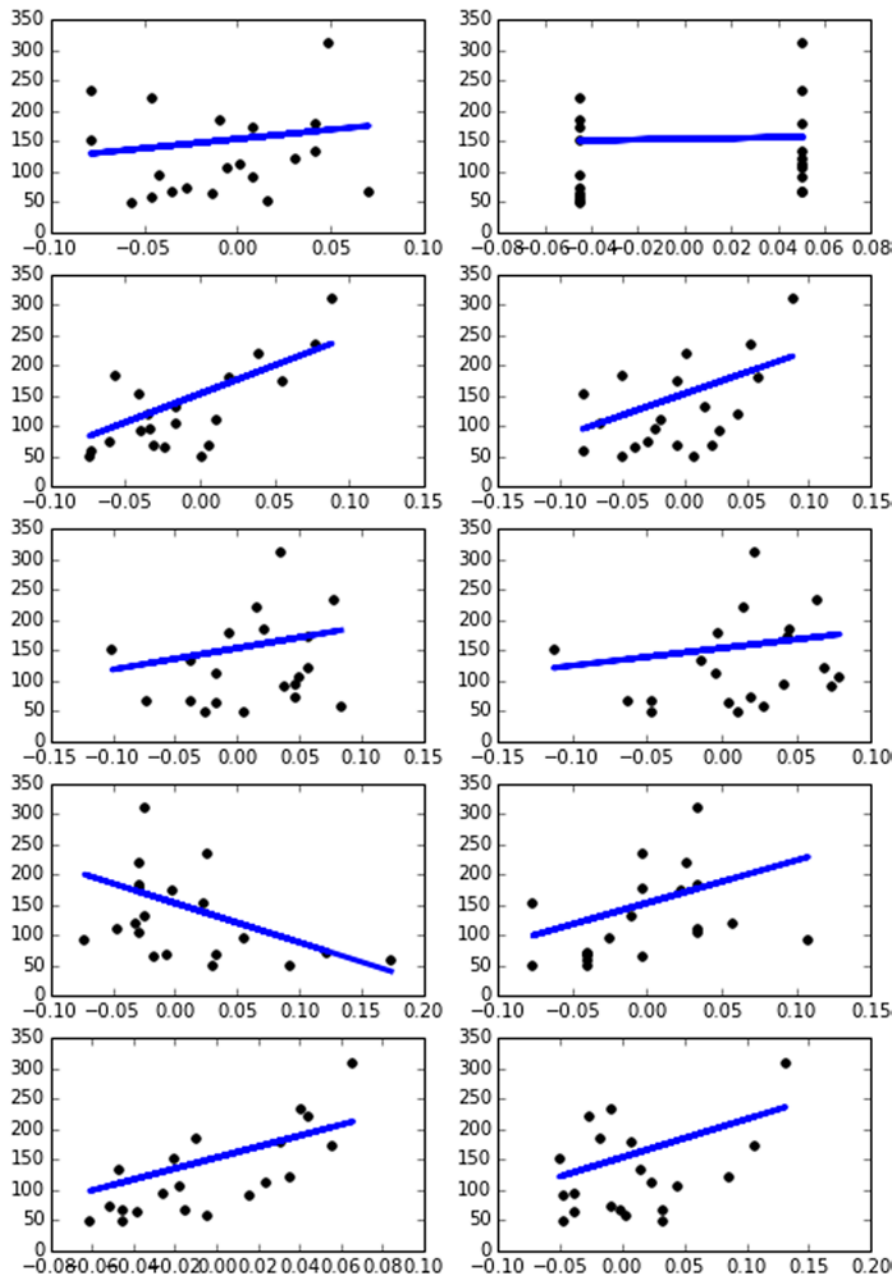
Figure 8-8. *Ten linear charts showing the correlations between physiological factors and the progression of diabetes*

Support Vector Machines (SVMs)

Support Vector Machines are a number of machine learning techniques that were first developed in the AT&T laboratories by Vapnik and colleagues in the early 90s. The basis of this class of procedures is in fact an algorithm called **Support Vector**, which is a generalization of a previous algorithm called Generalized Portrait, developed in Russia in 1963 by Vapnik as well.

In simple words the SVM classifiers are binary or discriminating models, working on two classes of differentiation. Their main task is basically to discriminate against new observations between two classes. During the learning phase, these classifiers project the observations in a multidimensional space called **decisional space** and build a separation surface called **decision boundary** that divides this space into two areas of belonging. In the simplest case, that is, the linear case, the decision boundary will be represented by a plane (in 3D) or by a straight line (in 2D). In more complex cases the separation surfaces are curved shapes with increasingly articulated shapes.

The SVM can be used both in regression with the **SVR (Support Vector Regression)** and in classification with the **SVC (Support Vector Classification)**.

Support Vector Classification (SVC)

If you wish to better understand how this algorithm works, you can start by referring to the simplest case, that is the linear 2D case, where the decision boundary will be a straight line separating into two parts the decisional area. Take for example a simple training set where some points are assigned to two different classes. The training set will consist of 11 points (observations) with two different attributes that will have values between 0 and 4. These values will be contained within a NumPy array called *x*. Their belonging to one of two classes will be defined by 0 or 1 values contained in another array called *y*.

Visualize distribution of these points in space with a scatterplot which will then be defined as decision space (see Figure 8-9).

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm
x = np.array([[1,3],[1,2],[1,1.5],[1.5,2],[2,3],[2.5,1.5],
    [2,1],[3,1],[3,2],[3.5,1],[3.5,3]])
y = [0]*6 + [1]*5
plt.scatter(x[:,0],x[:,1],c=y,s=50,alpha=0.9)
Out[360]: <matplotlib.collections.PathCollection at 0x545634a8>
```
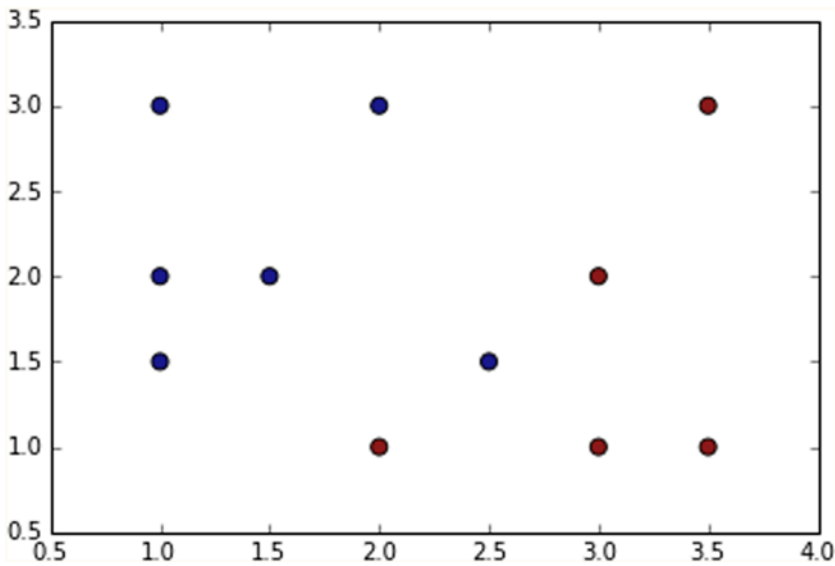
Figure 8-9. *The scatterplot of the training set displays the decision space*

Now that you have defined the training set, you can apply the SVC (Support Vector Classification) algorithm. This algorithm will create a line (decision boundary) in order to divide the decision area into two parts (see Figure 8-10), and this straight line will be placed so as to maximize its distance of closest observations contained in the training set. This condition should produce two different portions in which all points of a same class should be contained.

Then you apply the SVC algorithm to the training set and to do so first you define the model with the **SVC()** constructor defining the kernel as linear. (A kernel is a class of algorithms for pattern analysis.) Then you will use the **fit()** function with the training set as argument. Once the model is trained you can plot the decision boundary with the **decision_function()** function. Then you draw the scatterplot giving a different color to the two portions of the decision space.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm
x = np.array([[1,3],[1,2],[1,1.5],[1.5,2],[2,3],[2.5,1.5],
    [2,1],[3,1],[3,2],[3.5,1],[3.5,3]])
y = [0]*6 + [1]*5
svc = svm.SVC(kernel='linear').fit(x,y)
X,Y = np.mgrid[0:4:200j,0:4:200j]
Z = svc.decision_function(np.c_[X.ravel(),Y.ravel()])
Z = Z.reshape(X.shape)
plt.contourf(X,Y,Z > 0,alpha=0.4)
plt.contour(X,Y,Z,colors=['k'], linestyles=['-'],levels=[0])
plt.scatter(x[:,0],x[:,1],c=y,s=50,alpha=0.9)

Out[363]: <matplotlib.collections.PathCollection at 0x54acae10>
```
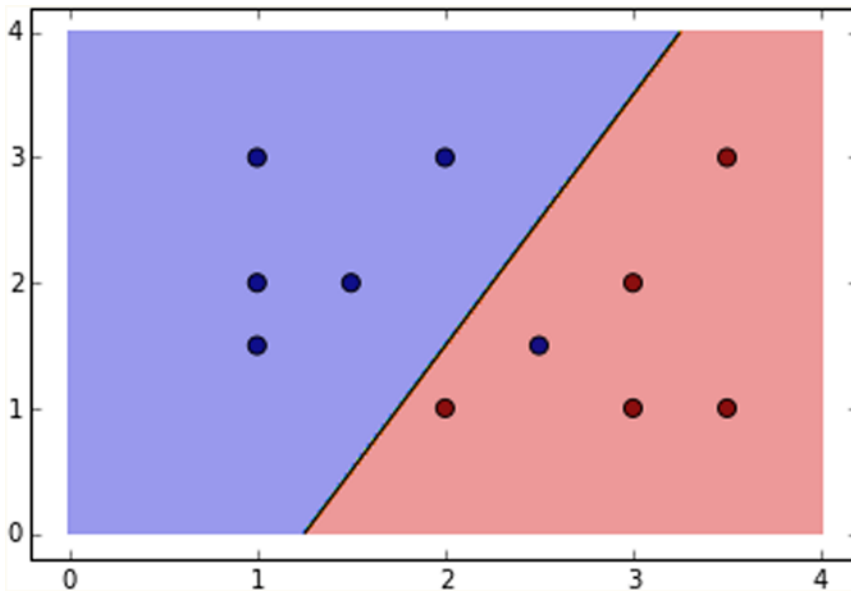
Figure 8-10. *The decision area is split into two portions*

In Figure 8-10 you can see the two portions containing the two classes. It can be said that the division is successful except for a blue dot in the red portion.

So once the model has been trained, it is simple to understand how the predictions operate. Graphically, depending on the position occupied by the new observation, you will know its corresponding membership in one of the two classes.

Instead, from a more programmatic point of view, the **predict()** function will return the number of the corresponding class of belonging (0 for class in blue, 1 for the class in red).

```
svc.predict([1.5,2.5])
Out[56]: array([0])

svc.predict([2.5,1])
Out[57]: array([1])
```

A related concept with the SVC algorithm is **regularization**. It is set by the parameter **C**: a small value of C means that the margin is calculated using many or all of the observations around the line of separation (greater regularization), while a large value of C means that the margin is calculated on the observations near to the line separation (lower regularization). Unless otherwise specified, the default value of C is equal to 1. You can highlight points that participated in the margin calculation, identifying them through the **support_vectors** array.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm
x = np.array([[1,3],[1,2],[1,1.5],[1.5,2],[2,3],[2.5,1.5],
    [2,1],[3,1],[3,2],[3.5,1],[3.5,3]])
y = [0]*6 + [1]*5
svc = svm.SVC(kernel='linear',C=1).fit(x,y)
X,Y = np.mgrid[0:4:200j,0:4:200j]
Z = svc.decision_function(np.c_[X.ravel(),Y.ravel()])
```

```
Z = Z.reshape(X.shape)
plt.contourf(X,Y,Z > 0,alpha=0.4)
plt.contour(X,Y,Z,colors=['k','k','k'], linestyles=['--','-','--'],levels=[-1,0,1])
plt.scatter(svc.support_vectors_[:,0],svc.support_vectors_[:,1],s=120,facecolors='none')
plt.scatter(x[:,0],x[:,1],c=y,s=50,alpha=0.9)
```

```
Out[23]: <matplotlib.collections.PathCollection at 0x177066a0>
```

These points are represented by rimmed circles in the scatterplot. Furthermore, they will be within an evaluation area in the vicinity of the separation line (see the dashed lines in Figure 8-11).
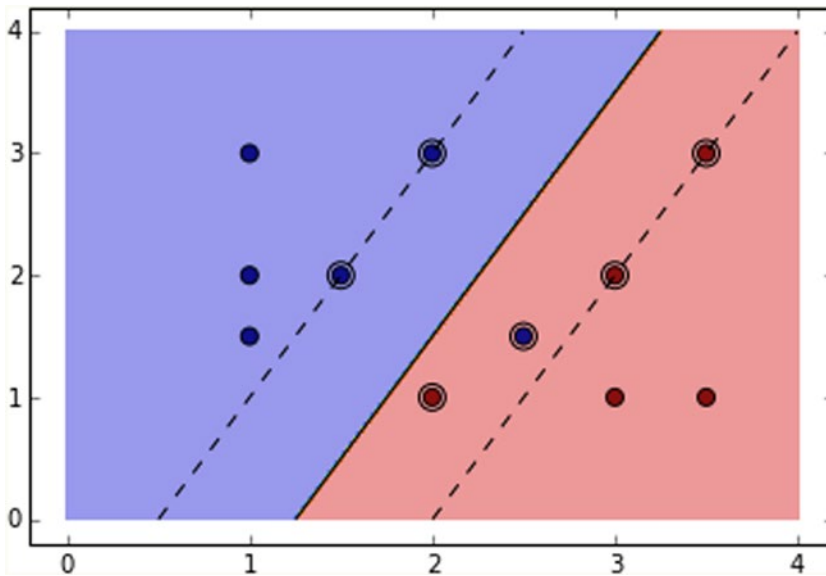
Figure 8-11. *The number of points involved in the calculation depends on the C parameter*

To see the effect on the decision boundary, you can restrict the value to C = 0.1. Let's see how many points will be taken into consideration.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm
x = np.array([[1,3],[1,2],[1,1.5],[1.5,2],[2,3],[2.5,1.5],
     [2,1],[3,1],[3,2],[3.5,1],[3.5,3]])
y = [0]*6 + [1]*5
svc = svm.SVC(kernel='linear',C=0.1).fit(x,y)
X,Y = np.mgrid[0:4:200j,0:4:200j]
Z = svc.decision_function(np.c_[X.ravel(),Y.ravel()])
Z = Z.reshape(X.shape)
plt.contourf(X,Y,Z > 0,alpha=0.4)
plt.contour(X,Y,Z,colors=['k','k','k'], linestyles=['--','-','--'],levels=[-1,0,1])
plt.scatter(svc.support_vectors_[:,0],svc.support_vectors_[:,1],s=120,facecolors='none')
plt.scatter(x[:,0],x[:,1],c=y,s=50,alpha=0.9)
```

```
Out[24]: <matplotlib.collections.PathCollection at 0x1a01ecc0>
```

The points taken into consideration are increased and consequently the separation line (decision boundary) has changed orientation. But now there are two points that are in the wrong decision areas (see Figure 8-12).
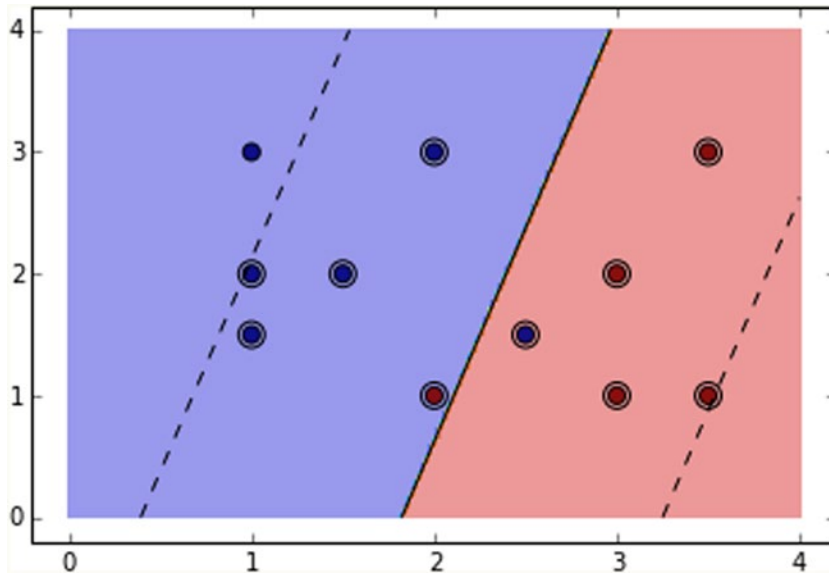
Figure 8-12. *The number of points involved in the calculation grows with decreasing of C*

Nonlinear SVC

So far you have seen the SVC linear algorithm defining a line of separation that was intended to split the two classes. There are also more complex SVC algorithms able to establish curves (2D) or curved surfaces (3D) based on the same principles of maximizing the distances between the points closest to the surface. Let's see the system using a polynomial kernel.

As the name implies, you can define a polynomial curve that separates the area decision in two portions. The degree of the polynomial can be defined by the **degree** option. Even in this case C is the coefficient of regularization. So try to apply an SVC algorithm with a polynomial kernel of third degree and with a C coefficient equal to 1.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm
x = np.array([[1,3],[1,2],[1,1.5],[1.5,2],[2,3],[2.5,1.5],
     [2,1],[3,1],[3,2],[3.5,1],[3.5,3]])
y = [0]*6 + [1]*5
svc = svm.SVC(kernel='poly',C=1, degree=3).fit(x,y)
X,Y = np.mgrid[0:4:200j,0:4:200j]
Z = svc.decision_function(np.c_[X.ravel(),Y.ravel()])
Z = Z.reshape(X.shape)
plt.contourf(X,Y,Z > 0,alpha=0.4)
plt.contour(X,Y,Z,colors=['k','k','k'], linestyles=['--','-','--'],levels=[-1,0,1])
```

```
plt.scatter(svc.support_vectors_[:,0],svc.support_vectors_[:,1],s=120,facecolors='none')
plt.scatter(x[:,0],x[:,1],c=y,s=50,alpha=0.9)
```

```
Out[34]: <matplotlib.collections.PathCollection at 0x1b6a9198>
```

As you can see, you get the situation shown in Figure 8-13.
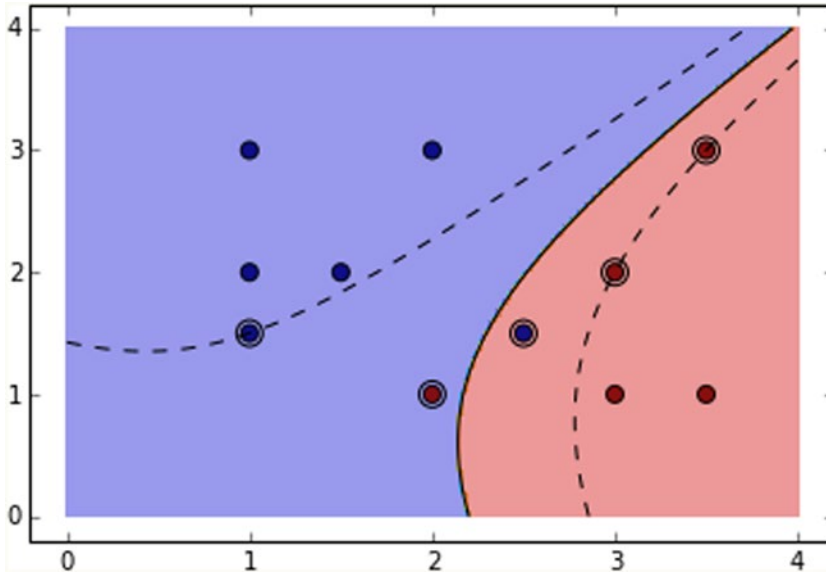
Figure 8-13. *The decision space using a SCV with a polynomial kernel*

Another type of nonlinear kernel is the **Radial Basis Function (RBF)**. In this case the separation curves tend to define the zones radially with respect to the observation points of the training set.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm
x = np.array([[1,3],[1,2],[1,1.5],[1.5,2],[2,3],[2.5,1.5],
    [2,1],[3,1],[3,2],[3.5,1],[3.5,3]])
y = [0]*6 + [1]*5
svc = svm.SVC(kernel='rbf', C=1, gamma=3).fit(x,y)
X,Y = np.mgrid[0:4:200j,0:4:200j]
Z = svc.decision_function(np.c_[X.ravel(),Y.ravel()])
Z = Z.reshape(X.shape)
plt.contourf(X,Y,Z > 0,alpha=0.4)
plt.contour(X,Y,Z,colors=['k','k','k'], linestyles=['--','-','--'],levels=[-1,0,1])
plt.scatter(svc.support_vectors_[:,0],svc.support_vectors_[:,1],s=120,facecolors='none')
plt.scatter(x[:,0],x[:,1],c=y,s=50,alpha=0.9)
```

```
Out[43]: <matplotlib.collections.PathCollection at 0x1cb8d550>
```

In Figure 8-14 we can see the two portions of the decision with all points of the training set correctly positioned.
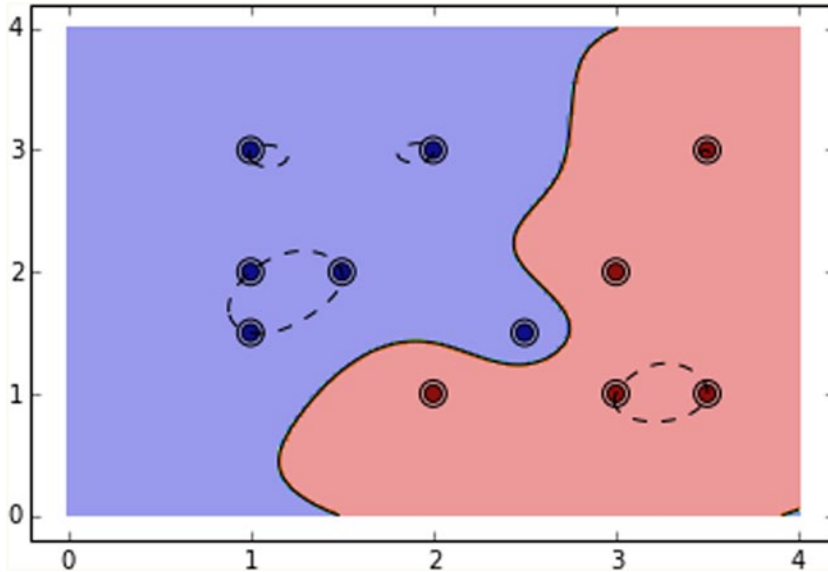
Figure 8-14. *The decisional area using a SVC with rgb kernel*

Plotting Different SVM Classifiers Using the Iris Dataset

The SVM example that you just saw is based on a very simple dataset. Use a more complex datasets for a classification problem with SVC. Use the previously used dataset: the Iris Dataset.

The SVC algorithm used before learned from a training set containing only two classes. In this case you will extend the case to three classifications, as three are the classes of the Iris Dataset is split, corresponding to the three different species of flowers.

In this case the decision boundaries intersect each other subdividing the decision area (in the case 2D) or the decision volume (3D) in several portions.

Both linear models have linear decision boundaries (intersecting hyperplanes) while models with nonlinear kernels (polynomial or Gaussian RBF) have nonlinear decision boundaries more flexible with figures that are dependent on the type of kernel and its parameters.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm, datasets

iris = datasets.load_iris()
x = iris.data[:,:2]
y = iris.target
h = .05
svc = svm.SVC(kernel='linear',C=1.0).fit(x,y)
x_min,x_max = x[:,0].min() - .5, x[:,0].max() + .5
y_min,y_max = x[:,1].min() - .5, x[:,1].max() + .5
```

```
h = .02
X, Y = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min,y_max,h))

Z = svc.predict(np.c_[X.ravel(),Y.ravel()])
Z = Z.reshape(X.shape)
plt.contourf(X,Y,Z,alpha=0.4)
plt.contour(X,Y,Z,colors='k')
plt.scatter(x[:,0],x[:,1],c=y)
```

```
Out[49]: <matplotlib.collections.PathCollection at 0x1f2bd828>
```

In Figure 8-15, the decision space is divided into three portions separated by decisional boundaries.
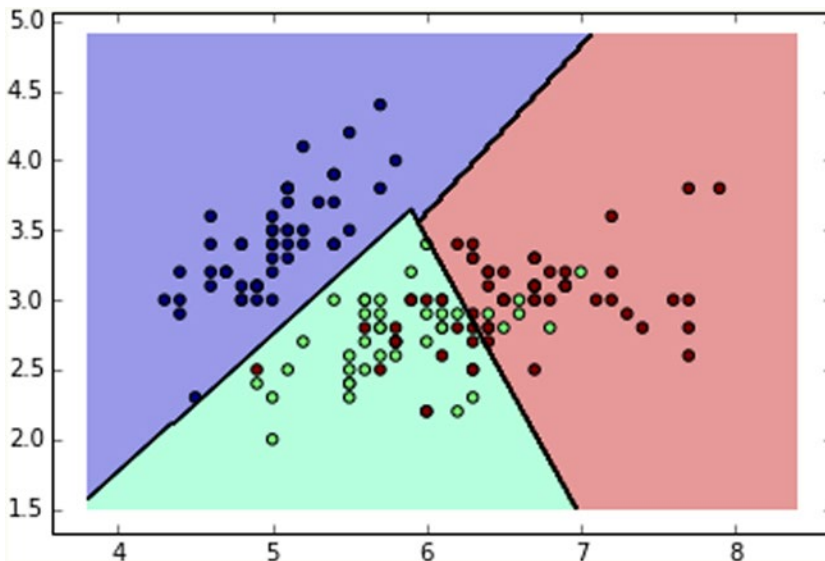
Figure 8-15. *The decisional boundaries split the decisional area into three different portions*

Now it's time to apply a nonlinear kernel for generating nonlinear decision boundaries, such as the polynomial kernel.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm, datasets

iris = datasets.load_iris()
x = iris.data[:,:2]
y = iris.target
h = .05
svc = svm.SVC(kernel='poly',C=1.0,degree=3).fit(x,y)
x_min,x_max = x[:,0].min() - .5, x[:,0].max() + .5
y_min,y_max = x[:,1].min() - .5, x[:,1].max() + .5

h = .02
X, Y = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min,y_max,h))
```

```
Z = svc.predict(np.c_[X.ravel(),Y.ravel()])
Z = Z.reshape(X.shape)
plt.contourf(X,Y,Z,alpha=0.4)
plt.contour(X,Y,Z,colors='k')
plt.scatter(x[:,0],x[:,1],c=y)
```

```
Out[50]: <matplotlib.collections.PathCollection at 0x1f4cc4e0>
```

Figure 8-16 shows how the polynomial decision boundaries split the area in a very different way compared to the linear case.
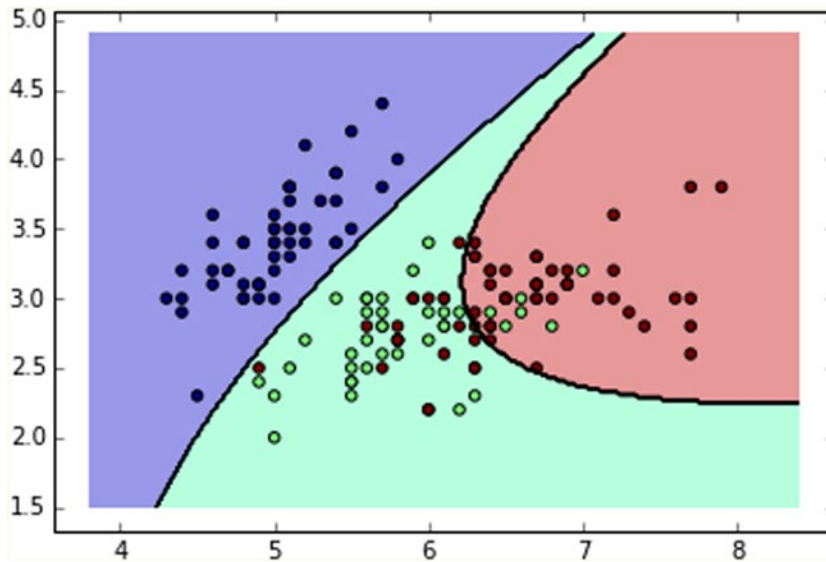
Figure 8-16. *In the polynomial case the blue portion is not directly connected with the red portion*

Now you can apply the RBF kernel to see the difference in the distribution of areas.

```
svc = svm.SVC(kernel='rbf', gamma=3, C=1.0).fit(x,y)
```

Figure 8-17 shows how the RBF kernel generates radial areas.

261
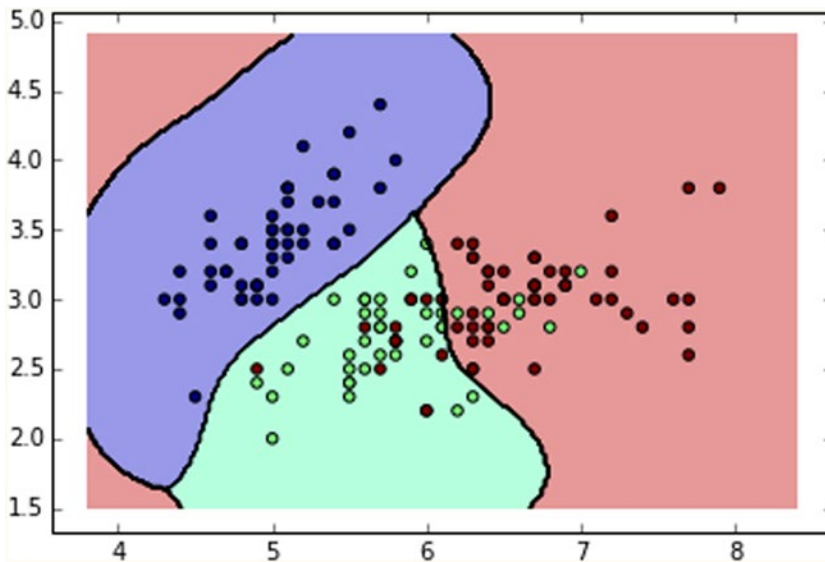
Figure 8-17. *The kernel RBF defines radial decisional areas*

Support Vector Regression (SVR)

The SVC method can be extended to solve even regression problems. This method is called **Support Vector Regression**.

The model produced by SVC actually does not depend on the complete training set, but uses only a subset of elements, i.e., those closest to the decisional boundary. In a similar way, the model produced by SVR also depends only on a subset of the training set.

We will demonstrate how the SVR algorithm will use the diabetes dataset that you have already seen in this chapter. By way of example you will refer only to the third physiological data. We will perform three different regressions, a linear and two nonlinear (polynomial). The linear case will produce a straight line as linear predictive model very similar to the linear regression seen previously, whereas polynomial regressions will be built of the second and third degrees. The SVR() function is almost identical to the SVC()function seen previously.

The only aspect to consider is that the test set of data must be sorted in ascending order.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm
from sklearn import datasets
diabetes = datasets.load_diabetes()
x_train = diabetes.data[:-20]
y_train = diabetes.target[:-20]
x_test = diabetes.data[-20:]
y_test = diabetes.target[-20:]

x0_test = x_test[:,2]
x0_train = x_train[:,2]
x0_test = x0_test[:,np.newaxis]
x0_train = x0_train[:,np.newaxis]
```

```
x0_test.sort(axis=0)
x0_test = x0_test*100
x0_train = x0_train*100

svr = svm.SVR(kernel='linear',C=1000)
svr2 = svm.SVR(kernel='poly',C=1000,degree=2)
svr3 = svm.SVR(kernel='poly',C=1000,degree=3)
svr.fit(x0_train,y_train)
svr2.fit(x0_train,y_train)
svr3.fit(x0_train,y_train)
y = svr.predict(x0_test)
y2 = svr2.predict(x0_test)
y3 = svr3.predict(x0_test)
plt.scatter(x0_test,y_test,color='k')
plt.plot(x0_test,y,color='b')
plt.plot(x0_test,y2,c='r')
plt.plot(x0_test,y3,c='g')
```

```
Out[155]: [<matplotlib.lines.Line2D at 0x262e10b8>]
```

The three regression curves will be represented with three different colors. The linear regression will be blue; the polynomial of second degree that is, a parabola, will be red; and the polynomial of third degree will be green (see Figure 8-18).
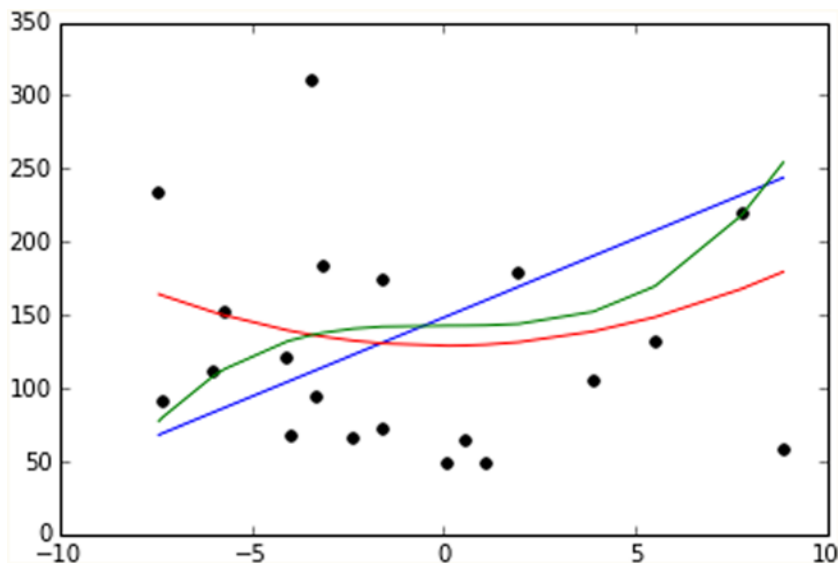
Figure 8-18. *The three regression curves produce very different trends starting from the training set*

Conclusions

In this chapter you saw the simplest cases of regression and classification problems solved using the scikit-learn library. Many concepts of the validation phase for a predictive model were presented in a practical way through some practical examples.

In the next chapter you will see a complete case in which all steps of data analysis are discussed through a single practical example. Everything will be implemented on IPython Notebook, an interactive and innovative environment well suited for sharing every step of the data analysis with the form of an interactive documentation useful as a report or as a web presentation.

An Example—Meteorological Data

One type of data easier to find on the net is meteorological data. Many sites provide historical data on many meteorological parameters such as pressure, temperature, humidity, rain, etc. You only need to specify the location and the date to get a file with data sets of measurements collected by weather stations. These data are a source of wide range of information and as you read in the first chapter of this book, the purpose of data analysis is to transform the raw data into information and then convert it into knowledge.

In this chapter you will see a simple example of how to use meteorological data. This example will be useful to get a general idea of how to apply many of the things seen in the previous chapters.

A Hypothesis to Be Tested: The Influence of the Proximity of the Sea

At the time of writing of this chapter, I find myself at the beginning of summer and high temperatures begin to be felt, especially for all those live in large cities. On the weekend, many people travel mountain villages or cities close to the sea, to enjoy a little refreshment and get away from the sultry weather of cities inland. This always made me ask, what effect does the proximity of the sea have on the climate?

This simple question can be a good starting point for the data analysis. I don't want to pass this chapter off as something scientific, just a way for someone passionate about data analysis to put his knowledge into practice in order to answer this question: **what influence might the presence of the sea have on the climate of a locality?**

The System in the Study: The Adriatic Sea and the Po Valley

Now that the problem has been defined, it is necessary to look for a system that is well suited to the study of the data and to provide a setting suitable to answer this question.

First you need a sea. Well, I'm in Italy and I have many seas to choose from, since Italy is a peninsula surrounded by seas. Why limit myself to Italy? Well, the problem involves a behavior typical of the Italians, that is, to take refuge in places close to the sea during the summer to avoid the heat of the hinterland. Not knowing if this behavior is the same for the people of other nations, I will only consider Italy as a system of study.

But what areas of Italy might you consider? Well, I said that Italy is a peninsula and that as far as the finding of a sea there is no problem, but what about the assessment of the effects of the sea at various distances? This creates a lot of problems. In fact, Italy is also rich in mountainous areas and also doesn't have very much territory uniformly extended for many kilometers inland. So, for the assessment of the effects of the sea I want to exclude the mountains, as they may introduce very many other factors, such as altitude for example.

A part of Italy that is well suited to this assessment is the Po Valley. This plain starts from the Adriatic Sea and spreads inland for hundreds of kilometers (see Figure 9-1). It is surrounded by mountains but its great width prevents their effects. It is also rich in towns and so it will be easy to choose a set of cities increasingly distant from the sea, to cover a distance of almost 400 km in this evaluation.

Figure 9-1. *An image of the Po Valley and the Adriatic Sea (Google Map)*

The first step is to choose a set of ten cities that will serve as a reference standard. These cities will have to be taken in order to cover the entire range of the plain (see Figure 9-2).
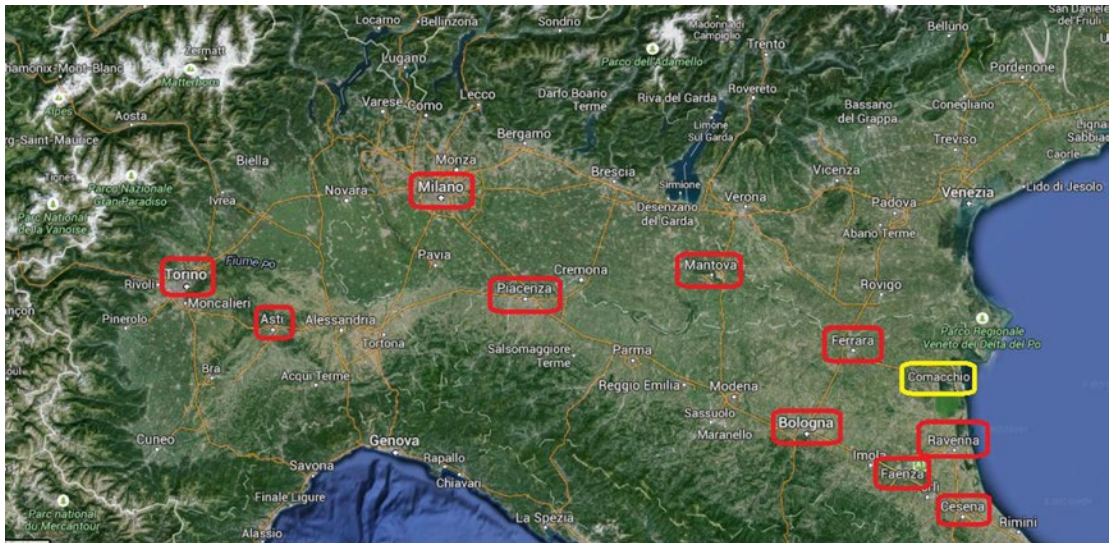
Figure 9-2. *The ten cities chosen as sample (there is another one used as a reference for take distances from the sea)*

From Figure 9-2 you can see the ten cities that were chosen to analyze weather data: five cities within the first 100 km and the other five distributed in the remaining 300 km.

Here is the list of cities chosen:

- Ferrara

- Torino

- Mantova

- Milano

- Ravenna

- Asti

- Bologna

- Piacenza

- Cesena

- Faenza

Now you have to determine the distances of these cities from the sea. You can follow many procedures to obtain these values. In your case, you can use the service provided by the site **TheTimeNow** (`http://www.thetimenow.com/distance-calculator.php`) available in many languages (See Figure 9-3).

Figure 9-3. *The service at the site TheTimeNow allows you to calculate distances between two cities*

Thanks to this service that calculates the distance between two cities, it is possible to calculate the approximate distance of the cities chosen from the sea. You can do this by selecting a city on the coast as destination. For many of them you can choose the city of Comacchio as reference to calculate the distance from the sea (see Figure 9-2). Once you have taken the distances from the ten cities, you will get the values shown in Table 9-1.

Table 9-1. *The Distances from the Sea of the Ten Cities*

City	Distance (km)	Note
Ravenna	8	Measured with Google-Earth
Cesena	14	Measured with Google-Earth
Faenza	37	Distance Faenza-Ravenna+8 km
Ferrara	47	Distance Ferrara-Comacchio
Bologna	71	Distance Bologna-Comacchio
Mantova	121	Distance Mantova-Comacchio
Piacenza	200	Distance Piacenza-Comacchio
Milano	250	Distance Milano-Comacchio
Asti	315	Distance Asti-Comacchio
Torino	357	Distance Torino-Comacchio

Data Source

Once the system under study has been defined, you need to establish a data source from which to obtain the needed data. Browsing the net here and there, you can discover many sites that provide meteorological data measured from various locations around the world. One such site is **Open Weather Map**, available at http://openweathermap.org/ (see Figure 9-4).
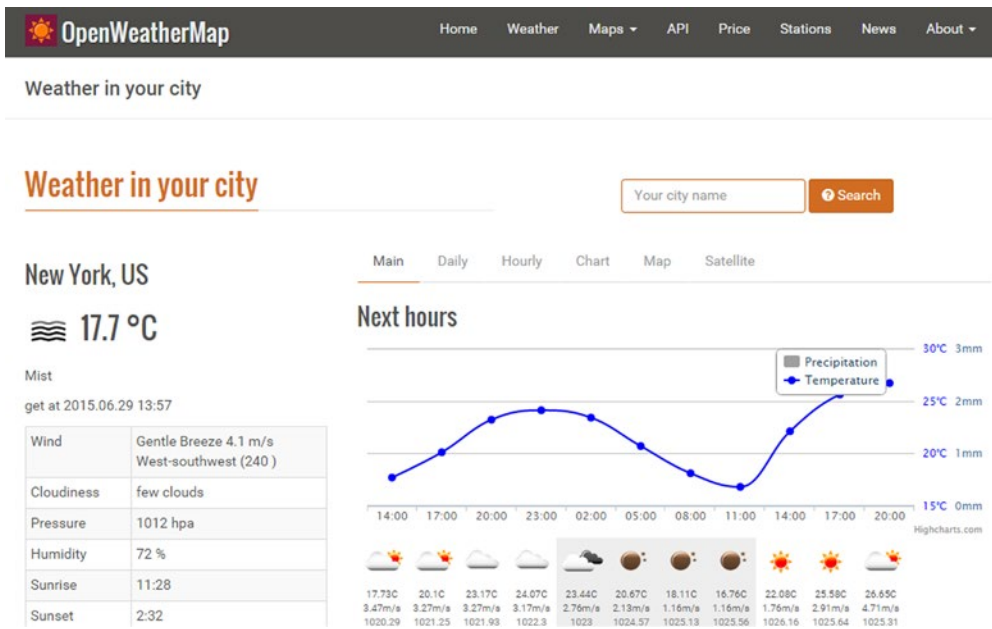
Figure 9-4. *Open Weather Map*

This site provides the ability to capture data by specifying the city through a request via URL.

http://api.openweathermap.org/data/2.5/history/city?q=Atlanta,US

This request will return a JSON file containing the meteorological data from Atlanta (see Figure 9-5). This JSON file will be submitted to data analysis using the Python pandas library.
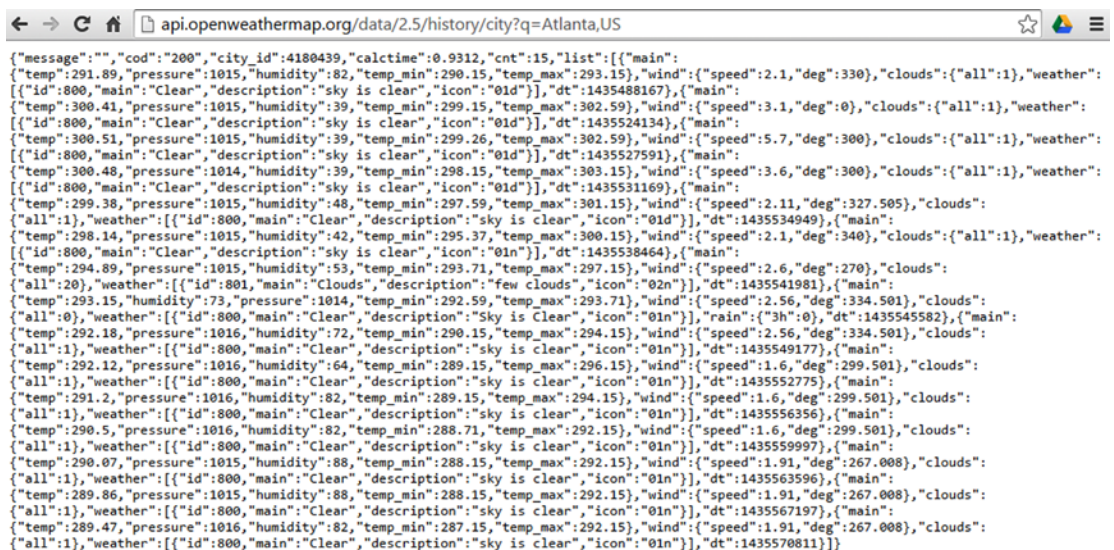
Figure 9-5. *The JSON file containing the meteorological data on urban issues*

Data Analysis on IPython Notebook

This chapter will address data analysis using IPython Notebook. This will allow you to enter and study portions of code gradually.

To start the IPython Notebook application, launch the following command from the command line:

```
ipython notebook
```

After the service is started, create a new Notebook.

Let's start by importing the necessary libraries:

```
import numpy as np
import pandas as pd
import datetime
```

Then you can import the data of the ten cities chosen. Since the server could be busy, it is recommended to load data from a city at a time. Otherwise, you may get an error message because the server can not satisfy all requests simultaneously.

```
ferrara = pd.read_json('http://api.openweathermap.org/data/2.5/history/city?q=Ferrara,IT')
torino = pd.read_json('http://api.openweathermap.org/data/2.5/history/city?q=Torino,IT')
mantova = pd.read_json('http://api.openweathermap.org/data/2.5/history/city?q=Mantova,IT')
milano = pd.read_json('http://api.openweathermap.org/data/2.5/history/city?q=Milano,IT')
ravenna = pd.read_json('http://api.openweathermap.org/data/2.5/history/city?q=Ravenna,IT')
asti = pd.read_json('http://api.openweathermap.org/data/2.5/history/city?q=Asti,IT')
bologna = pd.read_json('http://api.openweathermap.org/data/2.5/history/city?q=Bologna,IT')
piacenza = pd.read_json('http://api.openweathermap.org/data/2.5/history/city?q=Piacenza,IT')
cesena = pd.read_json('http://api.openweathermap.org/data/2.5/history/city?q=Cesena,IT')
faenza = pd.read_json('http://api.openweathermap.org/data/2.5/history/city?q=Faenza,IT')
```

Now JSON data are collected within ten data structures referred as DataFrame by using the **read_json()** function. The JSON structure is automatically converted to the tabular structure of the DataFrame. With this format, the data will be studied and developed in a much simpler way (see Figure 9-6).

```
In [153]: faenza
```

Out[153]:

	calctime	city_id	cnt	cod	list	message
0	0.1222	3177300	19	200	{u'clouds': {u'all': 0}, u'rain': {u'1h': 41.9...	
1	0.1222	3177300	19	200	{u'clouds': {u'all': 0}, u'rain': {u'1h': 41.9...	
2	0.1222	3177300	19	200	{u'clouds': {u'all': 0}, u'rain': {u'1h': 41.9...	
3	0.1222	3177300	19	200	{u'clouds': {u'all': 0}, u'rain': {u'1h': 41.9...	
4	0.1222	3177300	19	200	{u'clouds': {u'all': 0}, u'rain': {u'1h': 41.9...	
5	0.1222	3177300	19	200	{u'clouds': {u'all': 0}, u'rain': {u'1h': 41.9...	
6	0.1222	3177300	19	200	{u'clouds': {u'all': 0}, u'rain': {u'1h': 41.9...	
7	0.1222	3177300	19	200	{u'clouds': {u'all': 0}, u'rain': {u'1h': 41.9...	
8	0.1222	3177300	19	200	{u'clouds': {u'all': 0}, u'rain': {u'1h': 41.9...	
9	0.1222	3177300	19	200	{u'clouds': {u'all': 0}, u'rain': {u'1h': 41.9...	
10	0.1222	3177300	19	200	{u'clouds': {u'all': 0}, u'rain': {u'1h': 18.9...	
11	0.1222	3177300	19	200	{u'clouds': {u'all': 40}, u'rain': {u'1h': 41....	
12	0.1222	3177300	19	200	{u'clouds': {u'all': 40}, u'rain': {u'1h': 41....	
13	0.1222	3177300	19	200	{u'clouds': {u'all': 0}, u'rain': {u'1h': 41.9...	
14	0.1222	3177300	19	200	{u'clouds': {u'all': 0}, u'rain': {u'1h': 41.9...	
15	0.1222	3177300	19	200	{u'clouds': {u'all': 0}, u'rain': {u'1h': 41.9...	
16	0.1222	3177300	19	200	{u'clouds': {u'all': 0}, u'rain': {u'1h': 41.9...	
17	0.1222	3177300	19	200	{u'clouds': {u'all': 0}, u'rain': {u'1h': 41.9...	
18	0.1222	3177300	19	200	{u'clouds': {u'all': 0}, u'rain': {u'1h': 41.9...	

Figure 9-6. The tabular structure resulting from direct conversion of JSON files into a pandas DataFrame

As you can see, this is not really ready to be used for your analysis. In fact, all the meteorological data are enclosed within the **list** column. This column still continues to maintain a JSON structure inside and then you need to implement a procedure that lets you convert this tree-structured data into a tabular structure, much more suited to the data analysis using the pandas library.

```
def prepare(city_list,city_name):
    temp = [ ]
    humidity = [ ]
    pressure = [ ]
    description = [ ]
    dt = [ ]
    wind_speed = [ ]
    wind_deg = [ ]
```

```
    for row in city_list:
        temp.append(row['main']['temp']-273.15)
        humidity.append(row['main']['humidity'])
        pressure.append(row['main']['pressure'])
        description.append(row['weather'][0]['description'])
        dt.append(row['dt'])
        wind_speed.append(row['wind']['speed'])
        wind_deg.append(row['wind']['deg'])
    headings = ['temp','humidity','pressure','description','dt','wind_speed','wind_deg']
    data = [temp,humidity,pressure,description,dt,wind_speed,wind_deg]
    df = pd.DataFrame(data,index=headings)
    city = df.T
    city['city'] = city_name
    city['day'] = city['dt'].apply(datetime.datetime.fromtimestamp)
    return city
```

This function takes two arguments, the DataFrame and name of the corresponding city, and returns a new DataFrame. Among all the parameters described in the JSON structure within the **list** column, you choose those you deem most appropriate for the study of your system.

- Temperature

- Humidity

- Pressure

- Description

- Wind Speed

- Wind Degree

All these properties will be related to the time of acquisition expressed from the **dt** column, which contains timestamp as type of data. This value is difficult to read, so you will convert it into a datetime format that allows you to express the date and time in a manner more familiar to you. The new column will be called **day**.

```
city['day'] = city['dt'].apply(datetime.datetime.fromtimestamp)
```

Temperature is expressed in degrees Kelvin, and so you will need to convert these values in Celsius scale by subtracting 273.15 from each value.

Finally you add the name of the city to each row of the data frame by adding the **city** column.

Then execute the function just defined on all ten of the corresponding DataFrames.

```
df_ferrara = prepare(ferrara.list,'Ferrara')
df_milano = prepare(milano.list,'Milano')
df_mantova = prepare(mantova.list,'Mantova')
df_ravenna = prepare(ravenna.list,'Ravenna')
df_torino = prepare(torino.list,'Torino')
df_asti = prepare(asti.list,'Asti')
df_bologna = prepare(bologna.list,'Bologna')
df_piacenza = prepare(piacenza.list,'Piacenza')
df_cesena = prepare(cesena.list,'Cesena')
df_faenza = prepare(faenza.list,'Faenza')
```

Now another feature to be added is the distance of the city from the sea.

```
df_ravenna['dist'] = 8
df_cesena['dist'] = 14
df_faenza['dist'] = 37
df_ferrara['dist'] = 47
df_bologna['dist'] = 71
df_mantova['dist'] = 121
df_piacenza['dist'] = 200
df_milano['dist'] = 250
df_asti['dist'] = 315
df_torino['dist'] = 357
```

At the end, each city will be described through a DataFrame of the pandas library. Objects in this class will have a structure such as that shown in Figure 9-7.

In [154]: df_faenza

Out[154]:

	temp	humidity	pressure	description	dt	wind_speed	wind_deg	city	day	dist
0	25.44	69	1018	very heavy rain	1435390925	1.29	14.5002	Faenza	2015-06-27 09:42:05	37
1	26.38	73	1017	very heavy rain	1435394243	2.1	100	Faenza	2015-06-27 10:37:23	37
2	27.7	69	1017	very heavy rain	1435399019	3.1	120	Faenza	2015-06-27 11:56:59	37
3	29.04	61	1016	very heavy rain	1435402422	3.1	110	Faenza	2015-06-27 12:53:42	37
4	29.11	69	1016	very heavy rain	1435406058	3.6	110	Faenza	2015-06-27 13:54:18	37
5	29.33	65	1015	very heavy rain	1435409704	5.7	110	Faenza	2015-06-27 14:55:04	37
6	29.2	65	1014	very heavy rain	1435416898	5.1	110	Faenza	2015-06-27 16:54:58	37
7	28.88	65	1014	very heavy rain	1435420542	6.2	120	Faenza	2015-06-27 17:55:42	37
8	27.53	65	1014	very heavy rain	1435424296	6.7	120	Faenza	2015-06-27 18:58:16	37
9	26.12	73	1013	very heavy rain	1435427936	6.2	120	Faenza	2015-06-27 19:58:56	37
10	22.32	94	1015	very heavy rain	1435438357	2.6	90	Faenza	2015-06-27 22:52:37	37
11	21.38	83	1016	very heavy rain	1435442241	5.7	90	Faenza	2015-06-27 23:57:21	37
12	21.16	83	1016	very heavy rain	1435445863	2.1	210	Faenza	2015-06-28 00:57:43	37
13	20	94	1016	very heavy rain	1435453232	1.39	218.504	Faenza	2015-06-28 03:00:32	37
14	19.48	93	1016	very heavy rain	1435456487	2.1	330	Faenza	2015-06-28 03:54:47	37
15	19.16	93	1016	very heavy rain	1435460042	1.5	320	Faenza	2015-06-28 04:54:02	37
16	18.62	93	1016	very heavy rain	1435463880	0.5	300	Faenza	2015-06-28 05:58:00	37
17	19.34	88	1016	very heavy rain	1435467179	0.5	270	Faenza	2015-06-28 06:52:59	37
18	21.05	88	1016	very heavy rain	1435470851	2.1	260	Faenza	2015-06-28 07:54:11	37

Figure 9-7. *The DataFrame structure corresponding to a city*

If you noted, now the tabular data structure shows all measurements orderly and in an easily readable way. Furthermore, you can see that each row shows the measured values for each hour of the day, covering a timeline of about 20 hours in the past. In the case shown in Figure 9-7 you can however note that there are only 18 rows. In fact, observing other cities it is possible that some meteorological measurement systems have sometimes failed in their measure, leaving holes during the acquisition. However, if the data collected are 18, as in this case, they are sufficient to describe the trend of meteorological properties during the day.

However, it is good practice to check the size of all ten data frames. If a city provides insufficient data to describe the daily trend, you will need to replace it with another city.

```
print df_ferrara.shape
print df_milano.shape
print df_mantova.shape
print df_ravenna.shape
print df_torino.shape
print df_asti.shape
print df_bologna.shape
print df_piacenza.shape
print df_cesena.shape
print df_faenza.shape
```

This will give the following result:

```
(20, 9)
(18, 9)
(20, 9)
(18, 9)
(20, 9)
(20, 9)
(20, 9)
(20, 9)
(20, 9)
(19, 9)
```

As you can see the choice of ten cities has proved to be optimal, since the control units have provided enough data to continue in the data analysis.

Now that you have ten DataFrames containing data corresponding to the current day, you should save them in some way. Remember that if tomorrow (or even a few hours) the same code is executed again as described, all results will change! So it is good practice to save the data contained in the DataFrames as CSV files so they can be reused later.

```
df_ferrara.to_csv('ferrara_270615.csv')
df_milano.to_csv('milano_270615.csv')
df_mantova.to_csv('mantova_270615.csv')
df_ravenna.to_csv('ravenna_270615.csv')
df_torino.to_csv('torino_270615.csv')
df_asti.to_csv('asti_270615.csv')
df_bologna.to_csv('bologna_270615.csv')
df_piacenza.to_csv('piacenza_270615.csv')
df_cesena.to_csv('cesena_270615.csv')
df_faenza.to_csv('faenza_270615.csv')
```

By launching these commands now you have ten CSV files in your working directory.

By now if you want to refer to the data used in this chapter you have to load the ten CSV files that I saved at the time of writing this chapter.

```
df_ferrara.read_csv('ferrara_270615.csv')
df_milano.read_csv('milano_270615.csv')
df_mantova.read_csv('mantova_270615.csv')
df_ravenna.read_csv('ravenna_270615.csv')
df_torino.read_csv('torino_270615.csv')
df_asti.read_csv('asti_270615.csv')
df_bologna.read_csv('bologna_270615.csv')
df_piacenza.read_csv('piacenza_270615.csv')
df_cesena.read_csv('cesena_270615.csv')
df_faenza.read_csv('faenza_270615.csv')
```

Now a normal way to approach to the analysis of the data you have just collected is to use data visualization. You saw that the matplotlib library gives us a set of tools to generate charts on which to display data. In fact, data visualization helps you a lot during the data analysis to discover some features of the system you are studying.

Then you activate the necessary libraries:

```
%matplotlib inline
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
```

For example, a simple way to choose as the first approach is to analyze the trend of the temperature during the day. Take as first example the city of Milan.

```
y1 = df_milano['temp']
x1 = df_milano['day']
fig, ax = plt.subplots()
plt.xticks(rotation=70)
hours = mdates.DateFormatter('%H:%M')
ax.xaxis.set_major_formatter(hours)
ax.plot(x1,y1,'r')
```

Executing this code, you will get the graph shown in Figure 9-8. As you can see, the trend of temperature follows a nearly sinusoidal pattern characterized by a temperature that grows in the morning to reach the maximum value during the heat of the afternoon between 2:00 and 6:00 p.m. and then decreases up to a minimum value corresponding to just before dawn, that is, at 6:00 a.m.
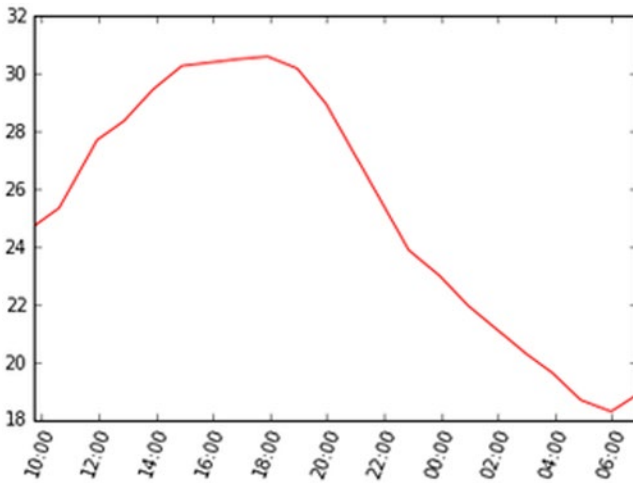
Figure 9-8. *Temperature trend of Milan during the day*

Since the purpose of your analysis is to try to interpret whether it is possible to assess how and if the sea is able to influence this trend, this time you evaluate the trends of different cities simultaneously. This is the only way to see if the analysis is going in the right direction. Thus, choose the three cities closest to the sea and the three cities furthest.

```
y1 = df_ravenna['temp']
x1 = df_ravenna['day']
y2 = df_faenza['temp']
x2 = df_faenza['day']
y3 = df_cesena['temp']
x3 = df_cesena['day']
y4 = df_milano['temp']
x4 = df_milano['day']
y5 = df_asti['temp']
x5 = df_asti['day']
y6 = df_torino['temp']
x6 = df_torino['day']
fig, ax = plt.subplots()
plt.xticks(rotation=70)
hours = mdates.DateFormatter('%H:%M')
ax.xaxis.set_major_formatter(hours)
plt.plot(x1,y1,'r',x2,y2,'r',x3,y3,'r')
plt.plot(x4,y4,'g',x5,y5,'g',x6,y6,'g')
```

This code will produce the chart as shown in Figure 9-9. The temperature of the three cities closest to the sea will be shown in red, while the temperature of the three cities most distant will be drawn in green.
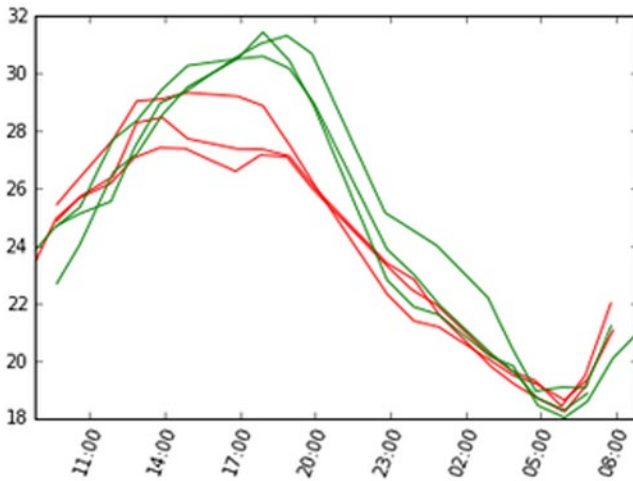
Figure 9-9. *The trend of the temperatures of six different cities (red: the closest to the sea; green: the furthest)*

Well, looking at Figure 9-9, the results seem promising. In fact, the three closest cities have maximum temperatures much lower than those furthest away, while as regards the minimum temperatures it seems that there is no difference for any of the cities.

In order to go deep into this aspect, you will collect all the maximum and minimum temperatures of the ten cities to display a line chart that sets forth the maximum and minimum temperatures in relation to the distance from the sea.

```
dist = [df_ravenna['dist'][0],
    df_cesena['dist'][0],
    df_faenza['dist'][0],
    df_ferrara['dist'][0],
    df_bologna['dist'][0],
    df_mantova['dist'][0],
    df_piacenza['dist'][0],
    df_milano['dist'][0],
    df_asti['dist'][0],
    df_torino['dist'][0]
]
temp_max = [df_ravenna['temp'].max(),
    df_cesena['temp'].max(),
    df_faenza['temp'].max(),
    df_ferrara['temp'].max(),
    df_bologna['temp'].max(),
    df_mantova['temp'].max(),
    df_piacenza['temp'].max(),
    df_milano['temp'].max(),
    df_asti['temp'].max(),
    df_torino['temp'].max()
]
```

```
temp_min = [df_ravenna['temp'].min(),
     df_cesena['temp'].min(),
     df_faenza['temp'].min(),
     df_ferrara['temp'].min(),
     df_bologna['temp'].min(),
     df_mantova['temp'].min(),
     df_piacenza['temp'].min(),
     df_milano['temp'].min(),
     df_asti['temp'].min(),
     df_torino['temp'].min()
]
```

Thus, start by representing the maximum temperatures.

```
plt.plot(dist,temp_max,'ro')
```

The result is shown in Figure 9-10.
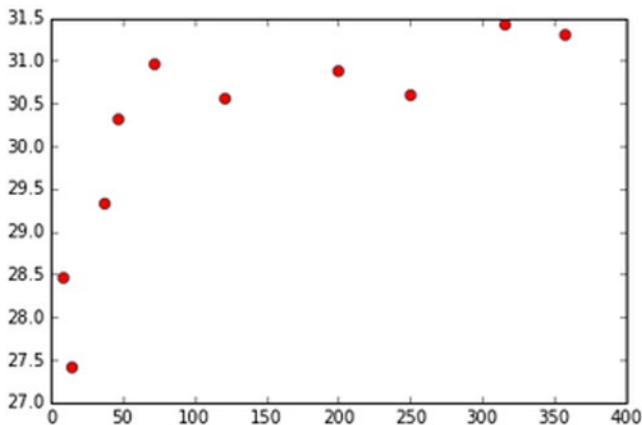
Figure 9-10. *Trend of maximum temperature in relation to the distance from the sea*

As shown in Figure 9-10, you can affirm that the hypothesis that the presence of the sea somehow influences meteorological parameters is true (at least in the day today ☺).

Furthermore, observing the chart you can see that the effect of the sea, however, decreases rapidly, and after about 60-70 km the maximum temperatures reach a plateau.

An interesting thing would be to represent the two different trends with two straight lines obtained by linear regression. In this connection, you can use the SVR method provided by the scikit-learn library.

```
from sklearn.svm import SVR
svr_lin1 = SVR(kernel='linear', C=1e3)
svr_lin2 = SVR(kernel='linear', C=1e3)
svr_lin1.fit(x1, y1)
svr_lin2.fit(x2, y2)
xp1 = np.arange(10,100,10).reshape((9,1))
xp2 = np.arange(50,400,50).reshape((7,1))
yp1 = svr_lin1.predict(xp1)
yp2 = svr_lin2.predict(xp2)
```

278

```
plt.plot(xp1, yp1, c='r', label='Strong sea effect')
plt.plot(xp2, yp2, c='b', label='Light sea effect')
plt.axis((0,400,27,32))
plt.scatter(x, y, c='k', label='data')
```

This code will produce the chart as shown in Figure 9-11.
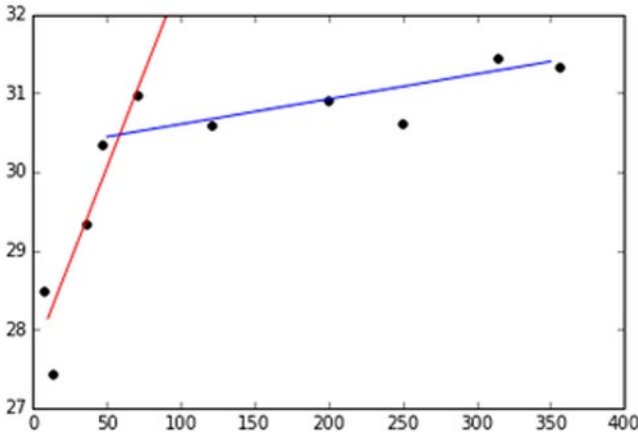
Figure 9-11. *The two trends described by the maximum temperatures in relation to the distance*

As you can see, the increase of the temperature for the first 60 km is very rapid, rising from 28 to 31 degrees, then increasing very mildly (if at all) over longer distances. The two trends are described by two straight lines that have the expression

$$x = ax + b$$

where *a* is the slope and the *b* is the intercept.

```
print svr_lin1.coef_
print svr_lin1.intercept_
print svr_lin2.coef_
print svr_lin2.intercept_
```

```
[[-0.04794118]]
[ 27.65617647]
[[-0.00317797]]
[ 30.2854661]
```

You might consider the intersection point of the two lines as the point between the area where the sea exerts its influence and the area where it doesn't, or at least not so strongly.

```
from scipy.optimize import fsolve

def line1(x):
    a1 = svr_lin1.coef_[0][0]
    b1 = svr_lin1.intercept_[0]
    return -a1*x + b1
```

279

```
def line2(x):
    a2 = svr_lin2.coef_[0][0]
    b2 = svr_lin2.intercept_[0]
    return -a2*x + b2
def findIntersection(fun1,fun2,x0):
 return fsolve(lambda x : fun1(x) - fun2(x),x0)

result = findIntersection(line1,line2,0.0)
print "[x,y] = [ %d , %d ]" % (result,line1(result))
x = numpy.linspace(0,300,31)
plt.plot(x,line1(x),x,line2(x),result,line1(result),'ro')
```

Executing the code, you can find the point of intersection

$$[x,y] = [\,58,\,30\,]$$

also represented in the chart shown in Figure 9-12.
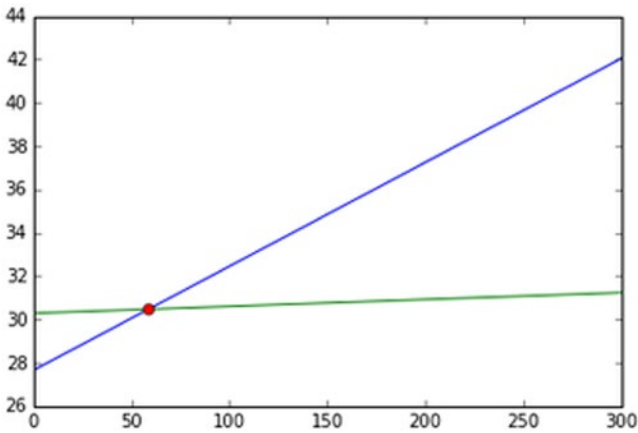
Figure 9-12. *The point of intersection between two straight lines obtained by linear regression*

So you can say that the average distance (measured today) in which the effects of the sea vanish is 58 km. Now, you can analyze instead the minimum temperatures.

```
plt.axis((0,400,15,25))
plt.plot(dist,temp_min,'bo')
```

This way, you'll obtain the chart shown in Figure 9-13.
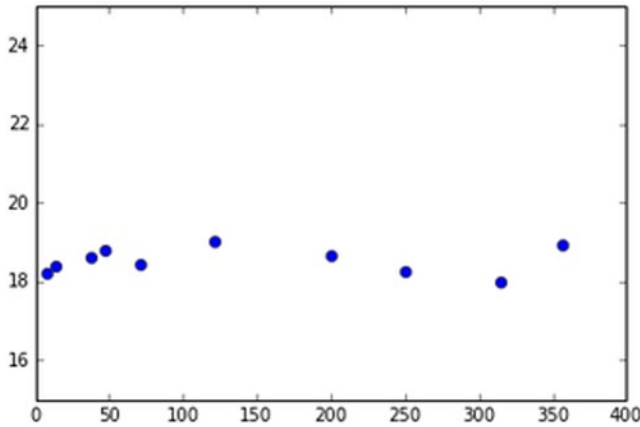
Figure 9-13. *The minimum temperatures are almost independent of the distance from the sea*

In this case it appears very clear that the sea has no effect on minimum temperatures recorded during the night, or rather, around six in the morning. If I remember well, when I was a child I was taught that the sea mitigated the cold temperatures, or that the sea released the heat absorbed during the day. This does not seem to be the case. We are in the summer and we are in Italy, it would be really interesting to see if this other hypothesis is fulfilled in winter or somewhere else.

Another meteorological measure contained within the ten DataFrames is the humidity. So even for this measure, you can see the trend of the humidity during the day for three cities closest to the sea and for the three most distant.

```
y1 = df_ravenna['humidity']
x1 = df_ravenna['day']
y2 = df_faenza['humidity']
x2 = df_faenza['day']
y3 = df_cesena['humidity']
x3 = df_cesena['day']
y4 = df_milano['humidity']
x4 = df_milano['day']
y5 = df_asti['humidity']
x5 = df_asti['day']
y6 = df_torino['humidity']
x6 = df_torino['day']
fig, ax = plt.subplots()
plt.xticks(rotation=70)
hours = mdates.DateFormatter('%H:%M')
ax.xaxis.set_major_formatter(hours)
plt.plot(x1,y1,'r',x2,y2,'r',x3,y3,'r')
plt.plot(x4,y4,'g',x5,y5,'g',x6,y6,'g')
```

This piece of code will show the chart shown in Figure 9-14.
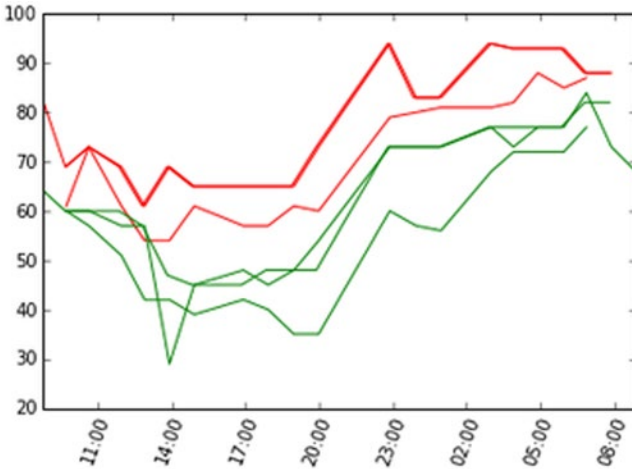
Figure 9-14. *The trend of the humidity during the day for three cities nearest the sea (shown in red) and the three cities furthest away (indicated in green)*

At first glance it would seem that the cities closest to the sea are more humid than those furthest away and that this difference in moisture (about 20%) extends throughout the day. Let's see if it remains true when we report the maximum and minimum humidity with respect to the distances from the sea.

```
hum_max = [df_ravenna['humidity'].max(),
    df_cesena['humidity'].max(),
    df_faenza['humidity'].max(),
    df_ferrara['humidity'].max(),
    df_bologna['humidity'].max(),
    df_mantova['humidity'].max(),
    df_piacenza['humidity'].max(),
    df_milano['humidity'].max(),
    df_asti['humidity'].max(),
    df_torino['humidity'].max()
]
plt.plot(dist,hum_max,'bo')
```

The maximum humidities of ten cities according to their distance from the sea are represented in the chart in Figure 9-15.
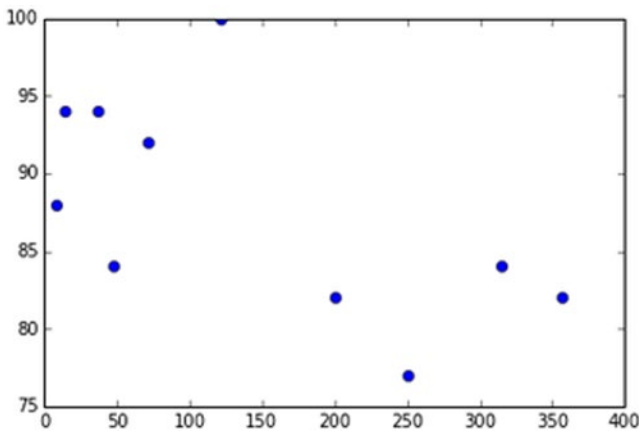
Figure 9-15. *The trend of the maximum humidities function with respect to the distance from the sea*

```
hum_min = [df_ravenna['humidity'].min(),
    df_cesena['humidity'].min(),
    df_faenza['humidity'].min(),
    df_ferrara['humidity'].min(),
    df_bologna['humidity'].min(),
    df_mantova['humidity'].min(),
    df_piacenza['humidity'].min(),
    df_milano['humidity'].min(),
    df_asti['humidity'].min(),
    df_torino['humidity'].min()
]
plt.plot(dist,hum_min,'bo')
```

The minimum humidities of ten cities according to their distance from the sea are represented in the chart in Figure 9-16.
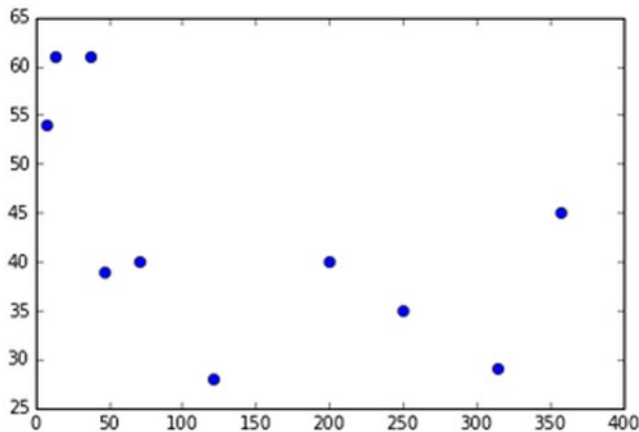
Figure 9-16. *The trend of the minimum humidity as a function of distance from the sea*

Looking at Figures 9-14 and 9-15, you can certainly see that the humidity is higher, both minimum and maximum, for the city closest to the sea. However in my opinion it is not possible to say that there is a linear relationship or of some other kind of relation to draw a curve. Collected points (10) are too few to highlight a trend in this case.

The RoseWind

Among the various meteorological data measured that we have collected for each city there are those relating to the wind:

- Wind Degree (Direction)
- Wind Speed

If you analyze the DataFrame, you will notice that the wind speed as well as being related to the hours of the day is also relative to the 360-degree direction. For instance, each measurement also shows the direction in which this wind blows (see Figure 9-17).

```
df_ravenna[['wind_deg','wind_speed','day']]
```

	wind_deg	wind_speed	day
0	159.5	2.01	2015-06-27 09:42:05
1	100	2.1	2015-06-27 10:37:24
2	80	4.6	2015-06-27 11:57:01
3	90	4.6	2015-06-27 12:53:43
4	80	6.2	2015-06-27 13:54:20
5	80	6.7	2015-06-27 14:55:06
6	90	6.7	2015-06-27 16:55:00
7	90	5.7	2015-06-27 17:55:43
8	90	4.6	2015-06-27 18:58:17
9	97	2.06	2015-06-27 19:58:58
10	89	2.06	2015-06-27 22:52:39
11	88.0147	2.86	2015-06-27 23:57:25
12	107.004	2.01	2015-06-28 00:57:46
13	107.004	2.01	2015-06-28 03:00:34
14	132.503	1.06	2015-06-28 03:54:49
15	132.503	1.06	2015-06-28 04:54:04
16	132.503	1.06	2015-06-28 05:58:15
17	251	1.54	2015-06-28 06:52:59

Figure 9-17. *The wind data contained within the DataFrame*

To make a better analysis of this kind of data, it is necessary to visualize them, but in this case a linear chart in Cartesian coordinates is not the most optimal.

In fact, if you use the classic scatter plot with the points contained within a single DataFrame

```
plt.plot(df_ravenna['wind_deg'],df_ravenna['wind_speed'],'ro')
```

you get a chart like the one shown in Figure 9-18, which certainly is not very effective.
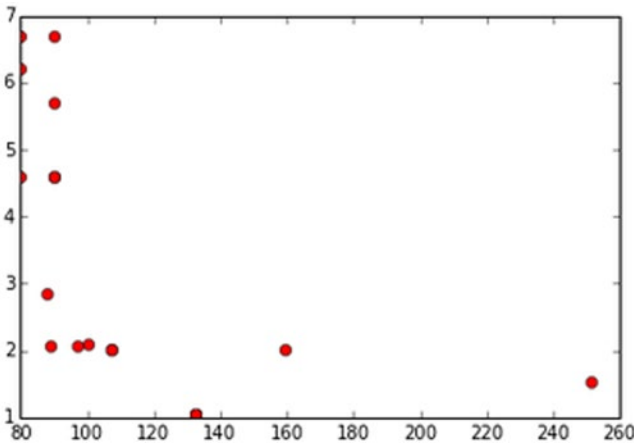
Figure 9-18. *A scatter plot representing a distribution of 360 degrees*

To represent a distribution of points on 360 degrees is best to use another type of visualization: the **polar chart**. You have already seen this king of chart in chapter 8.

First you need to create a histogram, i.e., the data will be distributed over the interval of 360 degrees divided into eight bins, each 45 degrees.

```
hist, bins = np.histogram(df_ravenna['wind_deg'],8,[0,360])
print hist
print bins
```

The values returned are occurrences within each bin expressed by an array called **hist**,

```
[ 0  5 11  1  0  1  0  0]
```

and an array called **bins**, which defines the edges of each bin within the range of 360 degrees.

```
[   0.   45.   90.  135.  180.  225.  270.  315.  360.]
```

These arrays will be useful to correctly define the polar chart to be drawn. For this purpose you have to create a function in part by using the code contained in Chapter 8. You will call this function **showRoseWind()**, and it will need three different arguments: **values** is the array containing the values to be displayed, which in our case will be the **hist** array; as the second argument **city_name** is a string which contains the name of the city to be shown as chart title; and finally **max_value** is an integer that will establish the max value for presenting the blue color.

The definition of a function of this kind is very useful as well to avoid rewriting the same code many times, and also to produce a more modular code, which allows you to focus the concepts related to a particular operation within a function.

```
def showRoseWind(values,city_name,max_value):
    N = 8
    theta = np.arange(0.,2 * np.pi, 2 * np.pi / N)
    radii = np.array(values)
    plt.axes([0.025, 0.025, 0.95, 0.95], polar=True)
    colors = [(1-x/max_value, 1-x/max_value, 0.75) for x in radii]
    plt.bar(theta, radii, width=(2*np.pi/N), bottom=0.0, color=colors)
    plt.title(city_name,x=0.2, fontsize=20)
```

One thing you have changed is the color map in colors. In fact, in this case you have made sure that the closer to blue the color of the slice is, the greater the value it represents.

Once you define a function you can do is use it:

```
showRoseWind(hist,'Ravenna',max(hist))
```

Executing this code, you will obtain a polar chart like the one shown in Figure 9-19.
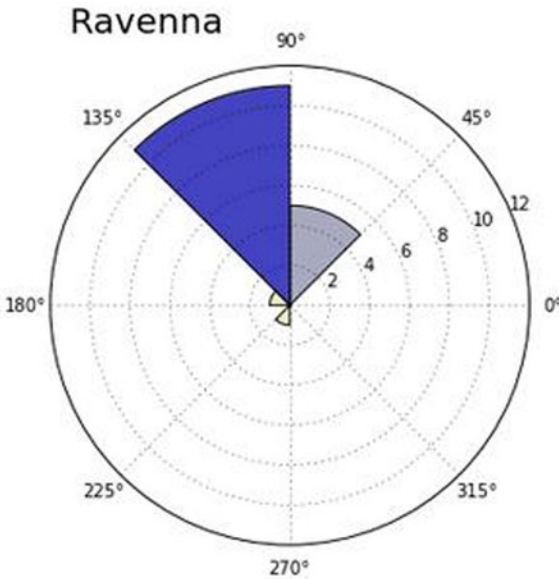
Figure 9-19. *The polar chart is able to represent the distribution of values within a range of 360 degrees*

As you can see in Figure 9-19, you have a range of 360 degrees divided into eight areas of 45 degrees each (bin), in which a scale of values is represented radially. In this case, in each of the eight areas a slice is represented with a variable length that corresponds precisely to the corresponding value. The more radially extended the slice is, the greater is the value to be represented. In order to increase the readability of the chart, a color scale has been entered corresponding to the extension of its slice. The wider the slice is, the more the color tends to a deep blue.

The polar chart just obtained provides you with information about how the wind direction will be distributed radially. In this case the wind has blown purely towards the southwest West for most of the day.

Once you have defined the **showRoseWind** function, it is really very easy to observe the situation of the winds with respect to any of the ten sample cities.

```
hist, bin = np.histogram(df_ferrara['wind_deg'],8,[0,360])
print hist
showRoseWind(hist,'Ferrara', 15.0)
```

Figure 9-20 shows all the polar charts of the ten cities.
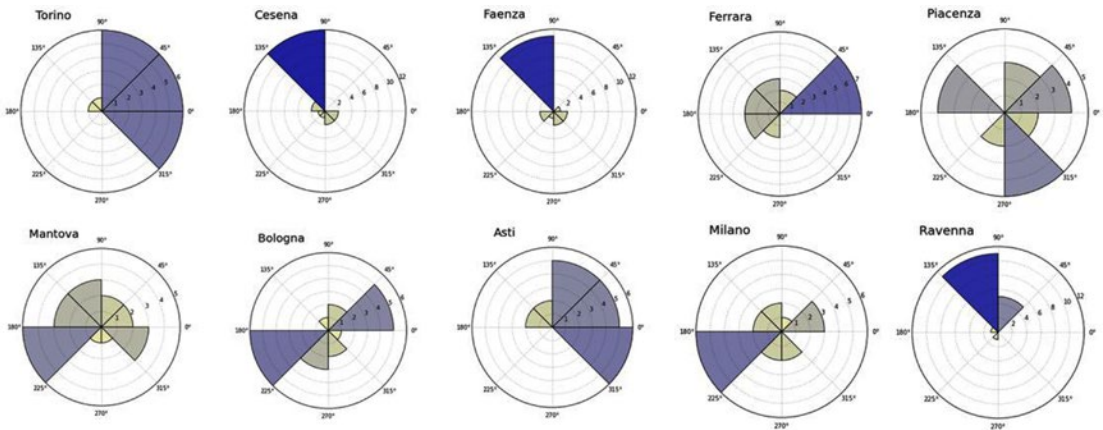
Figure 9-20. *The polar charts display the distribution of the wind direction*

Calculating the Distribution of the Wind Speed Means

Even the other quantity that relates the speed of the winds can be represented as a distribution on 360 degrees.

Now define a feature called **RoseWind_Speed** that will allow you to calculate the mean wind speeds for each of the eight bins into which 360 degrees are divided.

```
def RoseWind_Speed(df_city):
   degs = np.arange(45,361,45)
   tmp = []
   for deg in degs:
      tmp.append(df_city[(df_city['wind_deg']>(deg-46)) & (df_city['wind_deg']<deg)]
      ['wind_speed'].mean())
   return np.array(tmp)
```

This function returns a NumPy array containing the eight means of wind speeds. This array will be used as the first argument to the previous **ShowRoseWind()** function used for the representation of polar chart.

```
showRoseWind_Speed(RoseWind_Speed(df_ravenna),'Ravenna')
```

Indeed, Figure 9-21 represents the RoseWind corresponding to the wind speeds distributed around the 360 degrees.
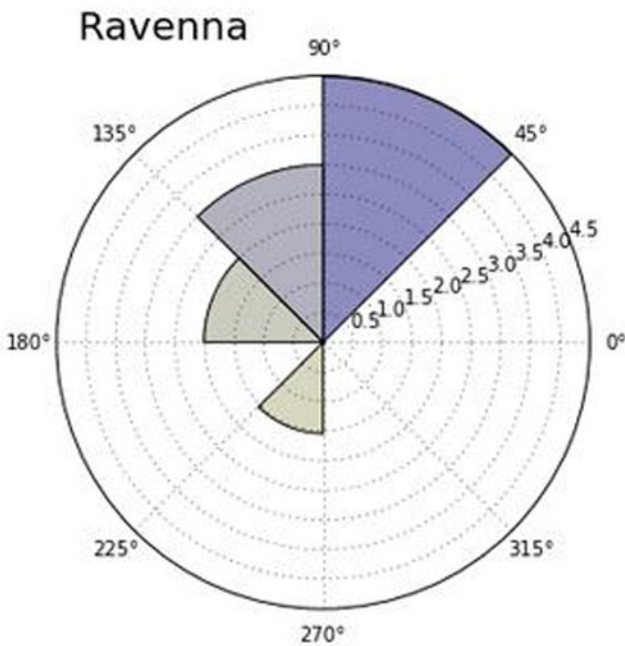
Figure 9-21. *This polar chart represents the distribution of wind speeds within 360 degrees*

Conclusions

The purpose of this chapter was mainly to see how you can get information from raw data. Some of this information will not lead to large conclusions, while others will lead to the confirmation of the hypothesis, thus increasing your state of knowledge. These are the cases in which the data analysis has led to a success.

In the next chapter, you will see another case relating to real data obtained from open data source. We'll also see how you can further enhance the graphical representation of the data using the D3 JavaScript library. This library, although not Python, can be easily integrated into Python.

■ ■ ■ ■

Embedding the JavaScript D3 Library in IPython Notebook

In this chapter you will see how to extend the capabilities of the graphical representation including the JavaScript D3 library within your IPython Notebooks. This library has enormous potential graphics and allows you to build graphical representations that even the matplotlib library cannot represent.

In the course of the various examples you will see how you can implement JavaScript code in a totally Python environment, using the large capacity of integrative IPython Notebook. Also you'll see different ways to use the data contained within Pandas dataframes Pandas in representations based on JavaScript code.

The Open Data Source for Demographics

In this chapter you will use demographic data as dataset on which to perform the analysis. A good starting point is the one suggested in the Web article "Embedding Interactive Charts on an IPython Notebook" written by Agustin Barto (`http://www.machinalis.com/blog/embedding-interactive-charts-on-an-ipython-nb/`). This article suggested the site of the **United States Census Bureau** (`http://www.census.gov`) as the data source for demographics (see Figure 10-1).
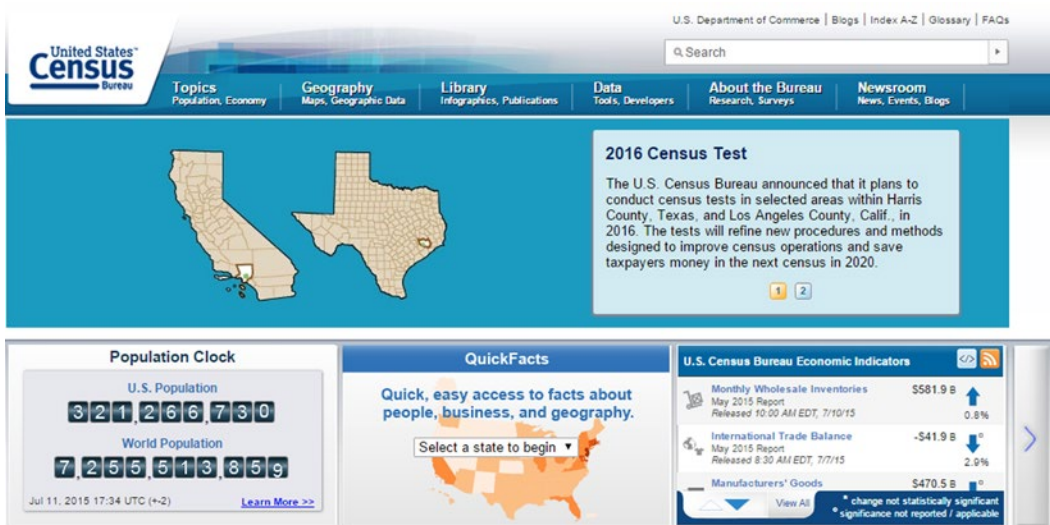
Figure 10-1.   This is the home page of the United States Census Bureau site

The United States Census Bureau is part of the United States Department of Commerce, and is officially in charge of collecting demographic data on the US population and making statistics on it. Its site provides a large amount of data as CSV files, which, as you have seen in previous chapters, are easily imported in the form of Pandas dataframes.

For the purposes of this chapter, you are interested to know all the data on the estimate of the population of all the states and counties in the United States. A CSV file that contains all of this information is *CO-EST2014-alldata.csv*.

So first, open an IPython Notebook and in the first frame, import all of the Python library that could later be needed in any page of IPython Notebook.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
```

Now that you have all the necessary libraries, you can start with importing data from Census.gov in your Notebook. So you need to upload the **CO-EST2014-alldata.csv** file directly in the form of a Pandas dataframe. The **Urllib2** library allows you to specify a URL of a file within the **pd.read_csv()** function, as you have seen in previous chapters. This function will convert tabular data contained in a CSV file to a Pandas dataframe, which you will name **pop2014**. Using the **dtype** option, you can force the interpretation of some fields that could be interpreted as numbers, as strings instead.

```
from urllib2 import urlopen

pop2014 = pd.read_csv(
    urlopen('http://www.census.gov/popest/data/counties/totals/2014/files/CO-EST2014-
    alldata.csv'),
    encoding='latin-1',
    dtype={'STATE': 'str', 'COUNTY': 'str'}
)
```

Once you have acquired and collected data in the pop2014 dataframe, you can see how they are structured by simply writing:

```
pop2014
```

obtaining an image like that shown in Figure 10-2.

	SUMLEV	REGION	DIVISION	STATE	COUNTY	STNAME	CTYNAME	CENSUS2010POP	ESTIMATESBASE2010	POPESTIMATE2010	...
0	40	3	6	01	000	Alabama	Alabama	4779736	4780127	4785822	...
1	50	3	6	01	001	Alabama	Autauga County	54571	54571	54684	...
2	50	3	6	01	003	Alabama	Baldwin County	182265	182265	183216	...
3	50	3	6	01	005	Alabama	Barbour County	27457	27457	27336	...
4	50	3	6	01	007	Alabama	Bibb County	22915	22919	22879	...
5	50	3	6	01	009	Alabama	Blount County	57322	57322	57344	...
6	50	3	6	01	011	Alabama	Bullock County	10914	10915	10886	...
7	50	3	6	01	013	Alabama	Butler County	20947	20946	20945	...
8	50	3	6	01	015	Alabama	Calhoun County	118572	118586	118443	...
9	50	3	6	01	017	Alabama	Chambers County	34215	34170	34111	...
10	50	3	6	01	019	Alabama	Cherokee County	25989	25986	25968	...

Figure 10-2. *The pop2014 dataframe contains all demographics for the years from 2010 to 2014*

Carefully analyzing the nature of the data, we can see how they are organized within the dataframe. The SUMLEV column contains the geographic level of the data; for example, you will have a value of 40 in respect of a state, while a value of 50 indicates data covering a single county. Then the data with SUMLEV equal to 40 contain the population and the estimates produced by the sum of all the peoples of the fields of level 50 that belong to the state.

The columns REGION, DIVISION, STATE, and COUNTY contain all hierarchical subdivisions of all areas in which the US territory has been divided. STNAME and CTYNAME indicate the name of the state and the county, respectively. In the following columns there are all the data on population. CENSUS2010POP is the column that contains the actual data on the population, that is, the data that were collected by a census made in the United States every ten years. Following that are other columns with the population estimates calculated for each year. In our example, you can see 2010 (2011, 2012, 2013, and 2014 are also in the dataframe but not shown in Figure 10-2).

You will use these values of population estimates as data to be represented in the examples discussed in this chapter.

Therefore, the pop2014 dataframe contains a large number of columns and rows that you are not interested in, and so it is convenient to eliminate unnecessary information. First, you are interested in the values of the people who relate to entire states, and so you extract only the rows with SUMLEV equals 40. Collect these data within the **pop2014_by_state** dataframe.

```
pop2014_by_state = pop2014[pop2014.SUMLEV == 40]
```

We get a dataframe as shown in Figure 10-3.

	SUMLEV	REGION	DIVISION	STATE	COUNTY	STNAME	CTYNAME	CENSUS2010POP	ESTIMATESBASE2010	POPESTIMATE
0	40	3	6	01	000	Alabama	Alabama	4779736	4780127	4785822
68	40	4	9	02	000	Alaska	Alaska	710231	710249	713856
98	40	4	8	04	000	Arizona	Arizona	6392017	6392310	6411999
114	40	3	7	05	000	Arkansas	Arkansas	2915918	2915958	2922297
190	40	4	9	06	000	California	California	37253956	37254503	37336011
249	40	4	8	08	000	Colorado	Colorado	5029196	5029324	5048575
314	40	1	1	09	000	Connecticut	Connecticut	3574097	3574096	3579345
323	40	3	5	10	000	Delaware	Delaware	897934	897936	899731
327	40	3	5	11	000	District of Columbia	District of Columbia	601723	601767	605210
329	40	3	5	12	000	Florida	Florida	18801310	18804623	18852220

Figure 10-3. *The pop2014_by_state dataframe contains all demographics related to the states*

However, the dataframe just obtained still contains too many columns with unnecessary information. Given the high number of columns, instead of carrying out their removal with the **drop()** function, it is more convenient to perform an extraction.

```
states = pop2014_by_state[['STNAME','POPESTIMATE2011', 'POPESTIMATE2012', 'POPESTIMATE2013',
'POPESTIMATE2014']]
```

Now that you have the essential information needed, you can evaluate the idea of starting to make graphical representations. For example, you could evaluate what are the five most populated states in the United States.

```
states.sort(['POPESTIMATE2014'], ascending=False)[:5]
```

Putting them in descending order, you will receive a dataframe is as shown in Figure 10-4.

	STNAME	POPESTIMATE2011	POPESTIMATE2012	POPESTIMATE2013	POPESTIMATE2014
190	California	37701901	38062780	38431393	38802500
2566	Texas	25657477	26094422	26505637	26956958
329	Florida	19107900	19355257	19600311	19893297
1860	New York	19521745	19607140	19695680	19746227
608	Illinois	12858725	12873763	12890552	12880580

Figure 10-4. *The five most populous states in the United States*

For example, you can consider the idea of using a bar chart to represent the five most populous states in descending order. This work is easily achieved with matplotlib, but in this chapter, you will take advantage of this simple representation to see how you can do the same representation in our IPython Notebook using the JavaScript **D3** library.

The JavaScript D3 Library

D3 is a JavaScript library that allows direct inspection and manipulation of the DOM object (HTML 5), but it is intended solely for data visualization and it really does its job excellently. In fact, the name D3 is derived from the three D's contained in "data-driven documents." D3 is a library entirely developed by Mike Bostock.

This library is proving to be very versatile and powerful, thanks to the technologies upon which it is based: JavaScript, SVG, and CSS. D3 combines powerful visualization components with a data-driven approach to the DOM manipulation. In so doing, D3 takes full advantage of the capabilities of the modern browser.

Given that even IPython Notebooks are Web objects, and use the same technologies that are the basis of the current browser, the idea of using this library, although JavaScript, within the notebook is not as preposterous as it may seem at first.

For those not familiar with the JavaScript D3 library and wish to know more about this topic I recommend reading another book: *Create Web Charts with D3* by F. Nelli (Apress, 2014).

Indeed, IPython notebook has the magic function **%% javascript** to integrate JavaScript code within the Python code.

But the JavaScript code, in a manner similar to Python, requires the import of some libraries to be executed. The libraries are available online and must be loaded each time you launch the execution. In HTML importing library has a particular construct:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/d3/3.5.5/d3.min.js"></script>
```

but this is an HTML tag and then to make the import within an IPython Notebook you should instead use this different construct:

```
%%javascript
require.config({
    paths: {
        d3: '//cdnjs.cloudflare.com/ajax/libs/d3/3.5.5/d3.min'
    }
});
```

Using **require.config()**, you can import all the necessary JavaScript libraries.

In addition, if you are familiar with HTML code you will know for sure that you need to define CSS styles if you want to strengthen the capacity of visualization of an HTML page. In parallel, also in IPython Notebook, you can define a set of CSS styles. To do this you can write HTML code, thanks to the **HTML()** function belonging to the IPython.core.display module. Therefore, make appropriate CSS definitions as follows:

```
from IPython.core.display import display, Javascript, HTML

display(HTML("""
<style>

.bar {
   fill: steelblue;
}

.bar:hover{
   fill: brown;
}
```

```
.axis {
   font: 10px sans-serif;
}

.axis path,

.axis line {
   fill: none;
   stroke: #000;
}

.x.axis path {
   display: none;
}

</style>
<div id="chart_d3" />
"""))
```

At the bottom of the previous code, you can notice the <div> HTML tag identified as chart_d3. This tag identifies the location where it will be represented the display D3.

Now you have to write the JavaScript code making use of the functions provided by the D3 library. Using the Template object provided by the **Jinja2** library, you can define a dynamic JavaScript code where you can replace the text depending on the values contained in a data frame Pandas.

If it still does not have a Jinja2 library installed in your system, you can always install it with Anaconda.

```
conda install jinja2
```

or using

```
pip install jinja2
```

After you have installed this library you can define the template.

```
import jinja2

myTemplate = jinja2.Template("""

require(["d3"], function(d3){

   var data = []

   {% for row in data %}
   data.push({ 'state': '{{ row[1] }}', 'population': {{ row[5] }}  });
   {% endfor %}

d3.select("#chart_d3 svg").remove()

    var margin = {top: 20, right: 20, bottom: 30, left: 40},
        width = 800 - margin.left - margin.right,
        height = 400 - margin.top - margin.bottom;
```

```
    var x = d3.scale.ordinal()
        .rangeRoundBands([0, width], .25);

    var y = d3.scale.linear()
        .range([height, 0]);

    var xAxis = d3.svg.axis()
        .scale(x)
        .orient("bottom");

    var yAxis = d3.svg.axis()
        .scale(y)
        .orient("left")
        .ticks(10)
        .tickFormat(d3.format('.1s'));

    var svg = d3.select("#chart_d3").append("svg")
        .attr("width", width + margin.left + margin.right)
        .attr("height", height + margin.top + margin.bottom)
        .append("g")
        .attr("transform", "translate(" + margin.left + "," + margin.top + ")");

    x.domain(data.map(function(d) { return d.state; }));
    y.domain([0, d3.max(data, function(d) { return d.population; })]);

    svg.append("g")
        .attr("class", "x axis")
        .attr("transform", "translate(0," + height + ")")
        .call(xAxis);

    svg.append("g")
        .attr("class", "y axis")
        .call(yAxis)
        .append("text")
        .attr("transform", "rotate(-90)")
        .attr("y", 6)
        .attr("dy", ".71em")
        .style("text-anchor", "end")
        .text("Population");

    svg.selectAll(".bar")
        .data(data)
        .enter().append("rect")
        .attr("class", "bar")
        .attr("x", function(d) { return x(d.state); })
        .attr("width", x.rangeBand())
        .attr("y", function(d) { return y(d.population); })
        .attr("height", function(d) { return height - y(d.population); });
});
""");
```

But you have not yet finished. Now it is the time to launch the representation of this D3 chart you have just defined. You also need to write the commands needed to pass data contained in the Pandas dataframe to the template, so they can be directly integrated into the JavaScript code written previously. The representation of JavaScript code, or rather the template just defined, will be executed by launching the **render()** function.

```
display(Javascript(myTemplate.render(
    data=states.sort(['POPESTIMATE2012'], ascending=False)[:10].itertuples()
)))
```

Once it has been launched, in the previous frame in which the <div> was placed, the bar chart. will appear as shown in Figure 10-5. This figure shows all the population estimates for the year 2014.
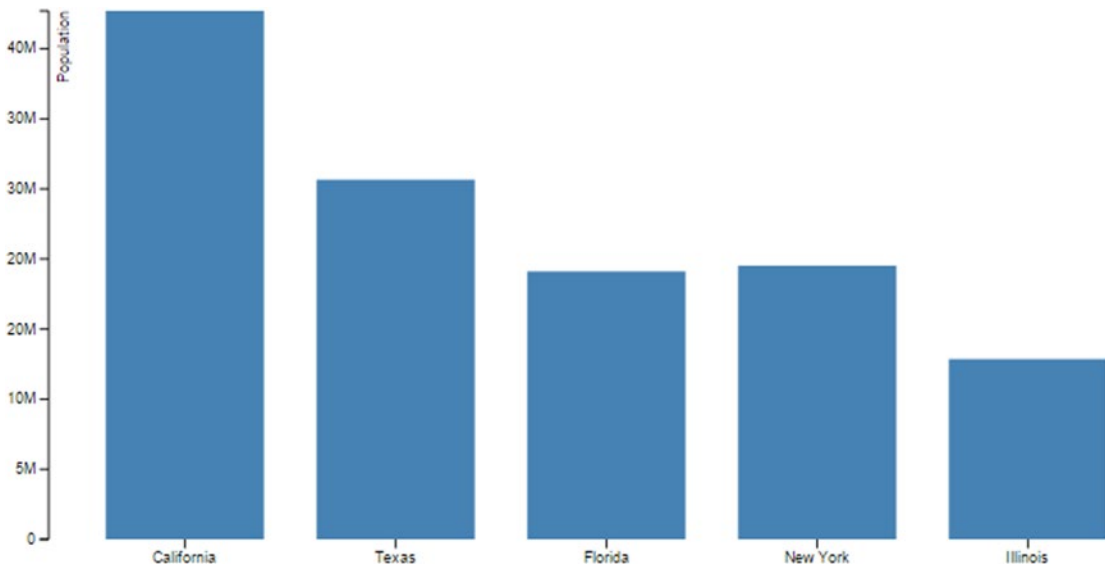
Figure 10-5. *The five most populous states of the United States represented by a bar chart relative to 2014*

Drawing a Clustered Bar Chart

So far you have relied broadly to what had been described in the fantastic article written by Barto. However, the type of data that you extracted has given you the trend of population estimates in the last four years for all states in the United States. So a more useful chart for visualizing data would be to show the trend of the population of each state over time.

To do that, a good choice could be to use a clustered bar chart, where each cluster is going to become one of the five most populous states, in which each cluster will have four bars to represent the population in a given year.

At this point you can modify the previous code or write code again in your IPython Notebook.

```
display(HTML("""
<style>

.bar2011 {
    fill: steelblue;
}

.bar2012 {
    fill: red;
}

.bar2013 {
    fill: yellow;
}

.bar2014 {
    fill: green;
}
.axis {
    font: 10px sans-serif;
}

.axis path,

.axis line {
    fill: none;
    stroke: #000;
}

.x.axis path {
    display: none;
}

</style>
<div id="chart_d3" />
"""))
```

You have to modify the template as well, adding the other three sets of data, also corresponding to the years 2011, 2012, and 2013. These years will be represented with a different color on the clustered bar chart.

```
import jinja2

myTemplate = jinja2.Template("""

require(["d3"], function(d3){

    var data = []
    var data2 = []
    var data3 = []
    var data4 = []
```

```
    {% for row in data %}
    data.push({ 'state': '{{ row[1] }}', 'population': {{ row[2] }}  });
    data2.push({ 'state': '{{ row[1] }}', 'population': {{ row[3] }}  });
    data3.push({ 'state': '{{ row[1] }}', 'population': {{ row[4] }}  });
    data4.push({ 'state': '{{ row[1] }}', 'population': {{ row[5] }}  });
    {% endfor %}

d3.select("#chart_d3 svg").remove()

    var margin = {top: 20, right: 20, bottom: 30, left: 40},
        width = 800 - margin.left - margin.right,
        height = 400 - margin.top - margin.bottom;

    var x = d3.scale.ordinal()
        .rangeRoundBands([0, width], .25);

    var y = d3.scale.linear()
        .range([height, 0]);

    var xAxis = d3.svg.axis()
        .scale(x)
        .orient("bottom");

    var yAxis = d3.svg.axis()
        .scale(y)
        .orient("left")
        .ticks(10)
        .tickFormat(d3.format('.1s'));

    var svg = d3.select("#chart_d3").append("svg")
        .attr("width", width + margin.left + margin.right)
        .attr("height", height + margin.top + margin.bottom)
        .append("g")
        .attr("transform", "translate(" + margin.left + "," + margin.top + ")");

    x.domain(data.map(function(d) { return d.state; }));
    y.domain([0, d3.max(data, function(d) { return d.population; })]);

    svg.append("g")
        .attr("class", "x axis")
        .attr("transform", "translate(0," + height + ")")
        .call(xAxis);

    svg.append("g")
        .attr("class", "y axis")
        .call(yAxis)
        .append("text")
        .attr("transform", "rotate(-90)")
        .attr("y", 6)
        .attr("dy", ".71em")
        .style("text-anchor", "end")
        .text("Population");
```

```
    svg.selectAll(".bar2011")
        .data(data)
        .enter().append("rect")
        .attr("class", "bar2011")
        .attr("x", function(d) { return x(d.state); })
        .attr("width", x.rangeBand()/4)
        .attr("y", function(d) { return y(d.population); })
        .attr("height", function(d) { return height - y(d.population); });

    svg.selectAll(".bar2012")
        .data(data2)
        .enter().append("rect")
        .attr("class", "bar2012")
        .attr("x", function(d) { return (x(d.state)+x.rangeBand()/4); })
        .attr("width", x.rangeBand()/4)
        .attr("y", function(d) { return y(d.population); })
        .attr("height", function(d) { return height - y(d.population); });

    svg.selectAll(".bar2013")
        .data(data3)
        .enter().append("rect")
        .attr("class", "bar2013")
        .attr("x", function(d) { return (x(d.state)+2*x.rangeBand()/4); })
        .attr("width", x.rangeBand()/4)
        .attr("y", function(d) { return y(d.population); })
        .attr("height", function(d) { return height - y(d.population); });

    svg.selectAll(".bar2014")
        .data(data4)
        .enter().append("rect")
        .attr("class", "bar2014")
        .attr("x", function(d) { return (x(d.state)+3*x.rangeBand()/4); })
        .attr("width", x.rangeBand()/4)
        .attr("y", function(d) { return y(d.population); })
        .attr("height", function(d) { return height - y(d.population); });

});
""");
```

Because now the series of data to be passed from the dataframe to the template are four, you have to refresh the data and the changes that you have just made to the code. So you will need to rerun the code of the render() function.

```
display(Javascript(myTemplate.render(
    data=states.sort(['POPESTIMATE2014'], ascending=False)[:5].itertuples()
)))
```

Once you have launched the render() function again, you get a chart like the one shown in Figure 10-6.
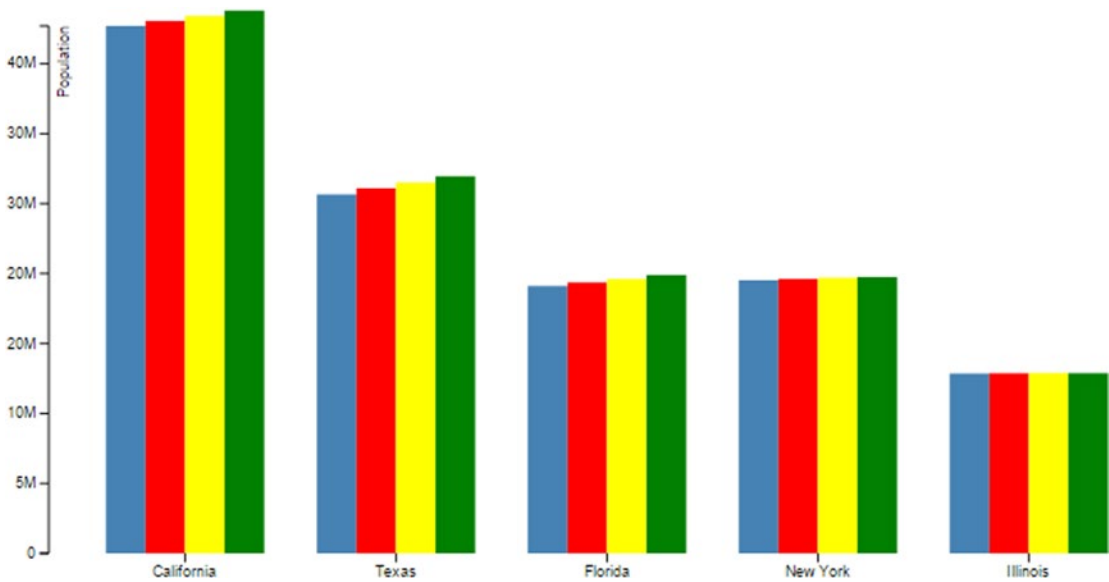
Figure 10-6. *A clustered bar chart representing the populations of the five most populous states from 2011 to 2014*

The Choropleth Maps

In the previous sections you saw how to use the JavaScript code and the D3 library to represent the bar chart. Well, these achievements would have been easy even with matplotlib and perhaps implemented in an even better way. The purpose of the previous code was only for educational purposes.

Something quite different is the use of much more complex views unobtainable by matplotlib. So now we will put in place the true potential made available by the D3 library. The **choropleth maps** are a very complex type of representation.

The choropleth maps are geographical representations where the land areas are divided into portions characterized by different colors. The colors and the boundaries between a portion geographical and another are themselves representations of data.

This type of representation is very useful to represent the results of an analysis of data carried out on demographic or economic information, and this is also the case for data that have a correlation to their geographical distribution.

The representation of choropleth is based on a particular file called JSON TopoJSON. This type of file contains all the inside information to represent a choropleth map such as that of the United States (see Figure 10-7).
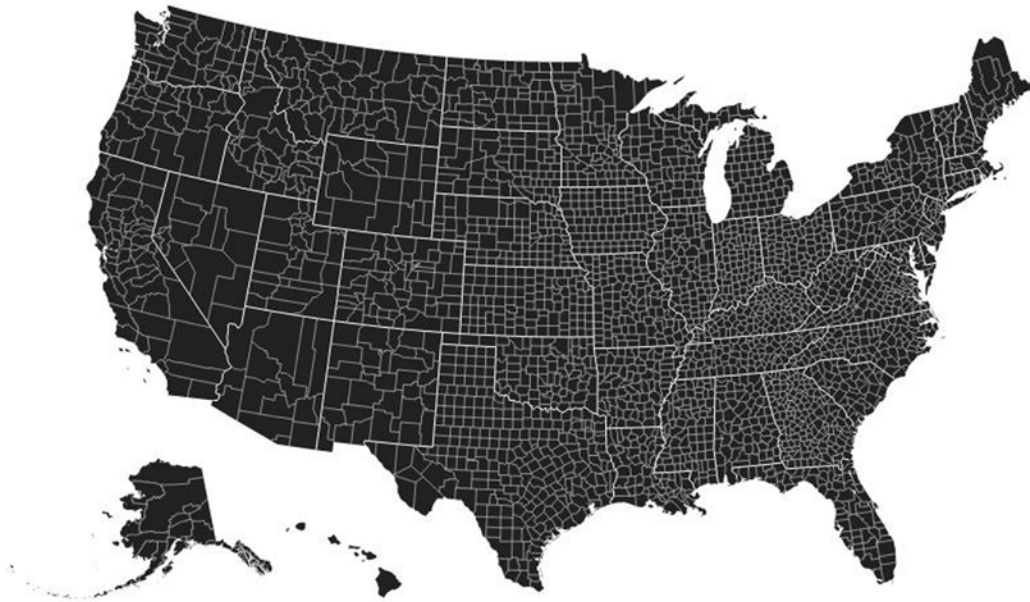
Figure 10-7. *The representation of a choropleth map of US territory with no value related to each county or state*

A good link where to find such material is the US Atlas TopoJSON (https://github.com/mbostock/us-atlas) but a lot of literature about it is available online.

Now a representation of this kind is not only possible but even customizable. Thanks to the D3 library, you can correlate the coloration of geographic portions based on the value of particular columns contained within a data frame.

First, let's start with an example already on the Internet, in the D3 library, http://bl.ocks.org/mbostock/4060606, but fully developed in HTML. So now you will learn how to adapt a D3 example in HTML in an IPython Notebook.

If you look at the code shown in the web page of the example you can see that the necessary JavaScript libraries are three. This time, in addition to the D3 library, we need to import both queue and TopoJSON libraries.

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/d3/3.5.5/d3.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/queue-async/1.0.7/queue.min.js">
</script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/topojson/1.6.19/topojson.min.js">
</script>
```

So you have to use the require.config() as you did in the previous sections.

```
%%javascript
require.config({
    paths: {
        d3: '//cdnjs.cloudflare.com/ajax/libs/d3/3.5.5/d3.min',
        queue: '//cdnjs.cloudflare.com/ajax/libs/queue-async/1.0.7/queue.min',
        topojson: '//cdnjs.cloudflare.com/ajax/libs/topojson/1.6.19/topojson.min'
    }
});
```

As regards the part of CSS is shown again all within the HTML() function.

```
from IPython.core.display import display, Javascript, HTML

display(HTML("""
<style>

.counties {
  fill: none;
}

.states {
  fill: none;
  stroke: #fff;
  stroke-linejoin: round;
}

.q0-9 { fill:rgb(247,251,255); }
.q1-9 { fill:rgb(222,235,247); }
.q2-9 { fill:rgb(198,219,239); }
.q3-9 { fill:rgb(158,202,225); }
.q4-9 { fill:rgb(107,174,214); }
.q5-9 { fill:rgb(66,146,198); }
.q6-9 { fill:rgb(33,113,181); }
.q7-9 { fill:rgb(8,81,156); }
.q8-9 { fill:rgb(8,48,107); }

</style>
<div id="choropleth" />
"""))
```

Here is the new template that mirrors the code shown in the example of Bostock with some changes in this regard:

```
import jinja2

choropleth = jinja2.Template("""

require(["d3","queue","topojson"], function(d3,queue,topojson){

//   var data = []

//   {% for row in data %}
//   data.push({ 'state': '{{ row[1] }}', 'population': {{ row[2] }} });
//   {% endfor %}

d3.select("#choropleth svg").remove()

var width = 960,
    height = 600;
```

```
var rateById = d3.map();

ar quantize = d3.scale.quantize()
    .domain([0, .15])
    .range(d3.range(9).map(function(i) { return "q" + i + "-9"; }));

var projection = d3.geo.albersUsa()
    .scale(1280)
    .translate([width / 2, height / 2]);

var path = d3.geo.path()
    .projection(projection);

//row to modify
var svg = d3.select("#choropleth").append("svg")
    .attr("width", width)
    .attr("height", height);

queue()
    .defer(d3.json, "us.json")
    .defer(d3.tsv, "unemployment.tsv", function(d) { rateById.set(d.id, +d.rate); })
    .await(ready);

function ready(error, us) {
  if (error) throw error;

  svg.append("g")
      .attr("class", "counties")
    .selectAll("path")
      .data(topojson.feature(us, us.objects.counties).features)
    .enter().append("path")
      .attr("class", function(d) { return quantize(rateById.get(d.id)); })
      .attr("d", path);

  svg.append("path")
      .datum(topojson.mesh(us, us.objects.states, function(a, b) { return a !== b; }))
      .attr("class", "states")
      .attr("d", path);
}
});
""");
```

Now you launch the representation, this time without any value for the template, since all values are contained within the file JSON and TSV.

```
display(Javascript(choropleth.render()))
```

The results are identical to those shown in the example of Bostock (see Figure 10-8).
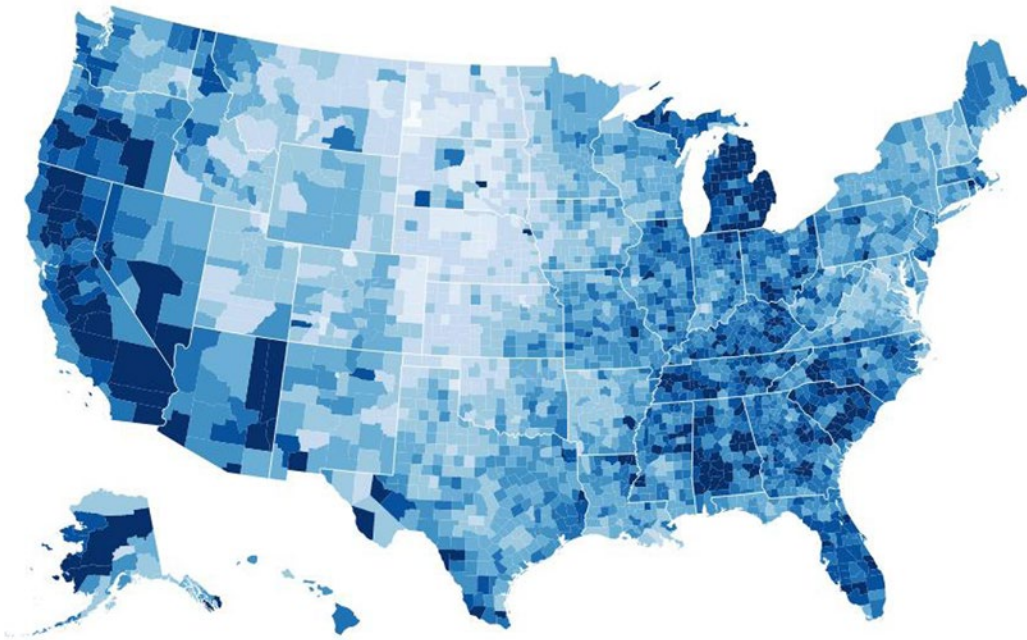
Figure 10-8. *The choropleth map of the United States with the coloring of the counties based on the values contained in the file TSV*

The Choropleth Map of the US Population in 2014

Now that you have seen how to extract demographic information from the US Census Bureau and that you can achieve the choropleth map, you can unify both things to represent a choropleth map with a degree of coloration that will represent the population values. The more populous the county, the deeper blue it will be. In counties with low population levels, the hue will tend toward white.

In the first section of the chapter, you extracted information on the states by the pop2014 dataframe. This was done by selecting the rows of the dataframe with SUMLEV values equal to 40. In this example, instead you need the values of the populations of each county and so you have to take out a new dataframe by taking pop2014 using only lines with SUMLEV of 50.

As regards the counties you must instead select the rows to level 50.

```
pop2014_by_county = pop2014[pop2014.SUMLEV == 50]
pop2014_by_county
```

You get a dataframe that contains all US counties like that in Figure 10-9.

	SUMLEV	REGION	DIVISION	STATE	COUNTY	STNAME	CTYNAME	CENSUS2010POP	ESTIMATESBASE2010	POPESTIMATE2010	...
1	50	3	6	01	001	Alabama	Autauga County	54571	54571	54684	...
2	50	3	6	01	003	Alabama	Baldwin County	182265	182265	183216	...
3	50	3	6	01	005	Alabama	Barbour County	27457	27457	27336	...
4	50	3	6	01	007	Alabama	Bibb County	22915	22919	22879	...
5	50	3	6	01	009	Alabama	Blount County	57322	57322	57344	...
6	50	3	6	01	011	Alabama	Bullock County	10914	10915	10886	...
7	50	3	6	01	013	Alabama	Butler County	20947	20946	20945	...
8	50	3	6	01	015	Alabama	Calhoun County	118572	118586	118443	...
9	50	3	6	01	017	Alabama	Chambers County	34215	34170	34111	...
10	50	3	6	01	019	Alabama	Cherokee County	25989	25986	25968	...

Figure 10-9. *The pop2014_by_county dataframe contains all demographics of all US counties*

You must use your data instead of TSV previously used. Inside it, there are the ID numbers corresponding to the various counties. To know their name a file exists in the Web; therefore you can download it and turn it into a dataframe.

```
from urllib2 import urlopen

USJSONnames = pd.read_table(urlopen('http://bl.ocks.org/mbostock/raw/4090846/us-county-
names.tsv'))
USJSONnames
```

Thanks to this file, you see the codes with the corresponding counties (see Figure 10-10).

	id	name
0	1000	Alabama
1	1001	Autauga
2	1003	Baldwin
3	1005	Barbour
4	1007	Bibb
5	1009	Blount
6	1011	Bullock
7	1013	Butler
8	1015	Calhoun

Figure 10-10. *The codes contained within the file TSV are the codes of the counties*

If you take for example a county as 'Baldwin'

```
USJSONnames[USJSONnames['name'] == 'Baldwin']
```

You can see that there are actually two counties with the same name, but in reality they are identified by two different identifiers (Figure 10-11).

	id	name
2	1003	Baldwin
399	13009	Baldwin

Figure 10-11. *There are two Baldwin Counties*

You get a table and find out that there are two counties and two different codes. Now you see in your dataframe with data taken from the data source census.gov (see Figure 10-12).

```
pop2014_by_county[pop2014_by_county['CTYNAME'] == 'Baldwin County']
```

	SUMLEV	REGION	DIVISION	STATE	COUNTY	STNAME	CTYNAME	CENSUS2010POP	ESTIMATESBASE2010	POPESTIMATE2010	...	RINTI
2	50	3	6	01	003	Alabama	Baldwin County	182265	182265	183216	...	1.471
402	50	3	5	13	009	Georgia	Baldwin County	45720	45835	45685	...	0.971

2 rows × 84 columns

Figure 10-12. *The ID codes in the TSV files correspond to the combination of the values contained in the STATE and COUNTY columns*

You can recognize that there is a match. The ID contained in TOPOJSON matches the numbers in the STATE and COUNTY columns if combined together, but removing the 0 when it is the digit at the beginning of the code. So now you can reconstruct all the data needed to replicate the TSV example of choropleth from the **counties** dataframe. The file will be saved as **population.csv**.

```
counties = pop2014_by_county[['STATE','COUNTY','POPESTIMATE2014']]
counties.is_copy = False
counties['id'] = counties['STATE'].str.lstrip('0') + "" + counties['COUNTY']
del counties['STATE']
del counties['COUNTY']
counties.columns = ['pop','id']
counties = counties[['id','pop']]
counties.to_csv('population.csv')
```

Now again you rewrite the contents of the HTML() function specifying a new <div> tag with the id as choropleth2.

```python
from IPython.core.display import display, Javascript, HTML

display(HTML("""
<style>

.counties {
  fill: none;
}

.states {
  fill: none;
  stroke: #fff;
  stroke-linejoin: round;
}

.q0-9 { fill:rgb(247,251,255); }
.q1-9 { fill:rgb(222,235,247); }
.q2-9 { fill:rgb(198,219,239); }
.q3-9 { fill:rgb(158,202,225); }
.q4-9 { fill:rgb(107,174,214); }
.q5-9 { fill:rgb(66,146,198); }
.q6-9 { fill:rgb(33,113,181); }
.q7-9 { fill:rgb(8,81,156); }
.q8-9 { fill:rgb(8,48,107); }

</style>
<div id="choropleth2" />
"""))
```

Finally, you have to define a new Template object.

```python
choropleth2 = jinja2.Template("""

require(["d3","queue","topojson"], function(d3,queue,topojson){

    var data = []

d3.select("#choropleth2 svg").remove()

var width = 960,
    height = 600;

var rateById = d3.map();

var quantize = d3.scale.quantize()
    .domain([0, 1000000])
    .range(d3.range(9).map(function(i) { return "q" + i + "-9"; }));

var projection = d3.geo.albersUsa()
    .scale(1280)
    .translate([width / 2, height / 2]);
```

```
var path = d3.geo.path()
    .projection(projection);

var svg = d3.select("#choropleth2").append("svg")
    .attr("width", width)
    .attr("height", height);

queue()
    .defer(d3.json, "us.json")
    .defer(d3.csv,"population.csv", function(d) { rateById.set(d.id, +d.pop); })
    .await(ready);

function ready(error, us) {
  if (error) throw error;

  svg.append("g")
      .attr("class", "counties")
    .selectAll("path")
      .data(topojson.feature(us, us.objects.counties).features)
    .enter().append("path")
      .attr("class", function(d) { return quantize(rateById.get(d.id)); })
      .attr("d", path);

  svg.append("path")
      .datum(topojson.mesh(us, us.objects.states, function(a, b) { return a !== b; }))
      .attr("class", "states")
      .attr("d", path);
}

});

""");
```

Finally, you can execute the render() function for getting the chart.

```
display(Javascript(choropleth2.render()))
```

The Choropleth map will be shown with the counties differently colored depending on their population as shown in Figure 10-13.
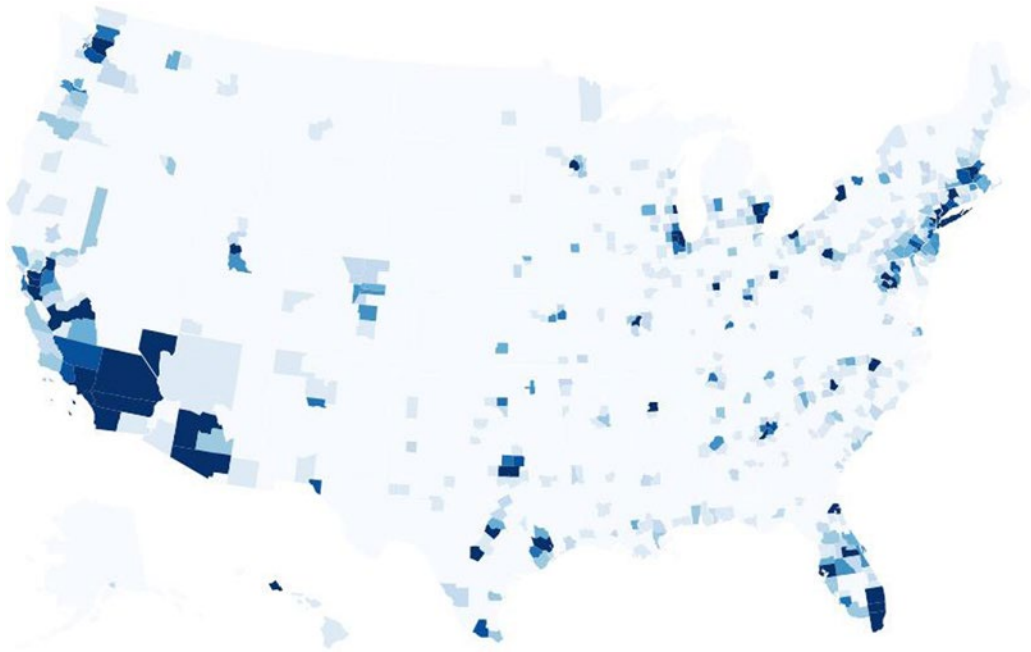
Figure 10-13. *The Choropleth map of the United States shows the density of the population of all counties*

Conclusions

With this chapter you have seen how it is possible to further extend the ability to display data using a JavaScript library called D3. The choropleth maps are just one of many examples of display advanced graphics that are often used to represent the data. This is also a very good example to see that working on IPython Notebook (Jupyter), you can integrate more technologies; in other words, the world does not revolve around Python alone, but Python can provide additional capabilities for our work.

In the next and final chapter you will see how to apply the data analysis also to images. You'll see how easy it is to build a model that is able to recognize handwritten numbers.

■ ■ ■

Recognizing Handwritten Digits

So far you have seen how to apply the techniques of data analysis to Pandas dataframes containing numbers or strings. Indeed, the data analysis is not limited to this, but also images and sounds can be analyzed and classified.

In this short but no-less-important chapter you'll face handwriting recognition, especially about the digits.

Handwriting Recognition

The recognition of handwritten text is a problem that can be traced back to the first automatic machines that had the need to recognize individual characters among the handwritten documents. You can think, for example, of the ZIP code on the letters at the post office and the automation needed to recognize the five digits. Their perfect recognition is necessary in order to sort mail automatically and efficiently. Included among the other applications that may come to mind is OCR (Optical Character Recognition) software, that is, software that must read handwritten text, or pages of printed books for general electronic documents in which each character is well defined.

But the problem of handwriting recognition goes further back in time, more precisely in the early 20th century (1920s), when Emanuel Goldberg (1881–1970) began his studies regarding this issue, suggesting that a statistical approach would be an optimal choice.

To address this issue, the scikit-learn library gives us a good example in order to better understand this technique, the issues involved, and the possibility of making predictions.

Recognizing Handwritten Digits with scikit-learn

The scikit-learn library (http://scikit-learn.org/) enables you to approach this type of data analysis in a way that is slightly different from what you've used throughout the book. The data to be analyzed is closely related to numerical values or strings, but can also involve images and sounds.

Therefore, it is clear that the problem you have to face in this chapter can be considered a prediction of a numeric value, reading and interpreting an image that shows a handwritten font.

So even in this case you will have an **estimator** with the task of learning through a **fit()** function, and once it has reached a degree of predictive capability (a model sufficiently valid) , it will produce a prediction with the **predict()** function. Then we will discuss the training set and validation set, compounds this time from a series of images.

Now open a new IPython Notebook session from command line, entering the following command:

```
ipython notebook
```

then create a new Notebook clicking on New ➤ Python 2 as shown in Figure 11-1.

Figure 11-1. *The home page of the IPython Notebook (Jupyter)*

An estimator that is useful in this case is **sklearn.svm.SVC**, which uses the technique of **Support Vector Classification (SVC)**.

Thus, you have to import the **svm** module of the scikit-learn library. You can create an estimator of SVC type and then choose an initial setting, setting the two values **C** and **gamma** with the generic values. These values can then be adjusted in a different way in the course of the analysis.

```
from sklearn import svm
svc = svm.SVC(gamma=0.001, C=100.)
```

The Digits Dataset

As we saw in Chapter 8, the scikit-learn library provides numerous datasets useful for testing many problems of data analysis and prediction of the results. Also in this case there is a dataset of images called **Digits**.

This dataset consists of 1,797 images of size 8x8 pixels. Each image is a handwritten digit shown in the image in a grayscale (see Figure 11-2).
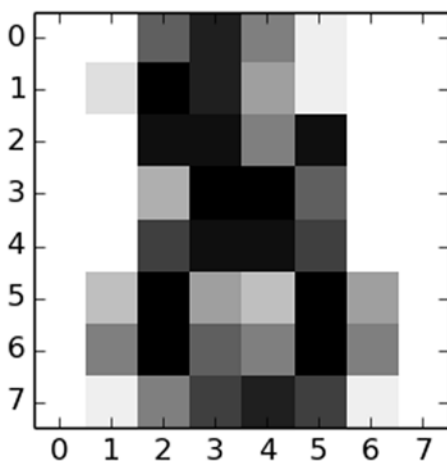
Figure 11-2. *One of 1,797 handwritten number images that make up the dataset digit*

Thus, you can load the Digits dataset in your Notebook.

```
from sklearn import datasets
digits = datasets.load_digits()
```

After loading the dataset, you can analyze the content. First, you can read a lot of information about the datasets that are contained within, calling the attribute DESCR.

```
print digits.DESCR
```

For a textual description of the dataset, the authors who contributed to its creation and the references will appear as shown in Figure 11-3.

```
print digits.DESCR

 Optical Recognition of Handwritten Digits Data Set

Notes
-----
Data Set Characteristics:
    :Number of Instances: 5620
    :Number of Attributes: 64
    :Attribute Information: 8x8 image of integer pixels in the range 0..16.
    :Missing Attribute Values: None
    :Creator: E. Alpaydin (alpaydin '@' boun.edu.tr)
    :Date: July; 1998

This is a copy of the test set of the UCI ML hand-written digits datasets
http://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits

The data set contains images of hand-written digits: 10 classes where
each class refers to a digit.

Preprocessing programs made available by NIST were used to extract
normalized bitmaps of handwritten digits from a preprinted form. From a
total of 43 people, 30 contributed to the training set and different 13
to the test set. 32x32 bitmaps are divided into nonoverlapping blocks of
4x4 and the number of on pixels are counted in each block. This generates
an input matrix of 8x8 where each element is an integer in the range
0..16. This reduces dimensionality and gives invariance to small
distortions.

For info on NIST preprocessing routines, see M. D. Garris, J. L. Blue, G.
T. Candela, D. L. Dimmick, J. Geist, P. J. Grother, S. A. Janet, and C.
L. Wilson, NIST Form-Based Handprint Recognition System, NISTIR 5469,
1994.
```

Figure 11-3. *Each dataset in the scikit-learn library has a field containing all the information*

Regarding the images of the handwritten digits, these are contained within a **digits.images** array. Each element of this array is an image that is represented by an 8x8 matrix of numerical values that correspond to a grayscale from white, with a value of 0, to black, with the value 15.

```
digits.images[0]
array([[  0.,   0.,   5.,  13.,   9.,   1.,   0.,   0.],
       [  0.,   0.,  13.,  15.,  10.,  15.,   5.,   0.],
       [  0.,   3.,  15.,   2.,   0.,  11.,   8.,   0.],
       [  0.,   4.,  12.,   0.,   0.,   8.,   8.,   0.],
       [  0.,   5.,   8.,   0.,   0.,   9.,   8.,   0.],
       [  0.,   4.,  11.,   0.,   1.,  12.,   7.,   0.],
       [  0.,   2.,  14.,   5.,  10.,  12.,   0.,   0.],
       [  0.,   0.,   6.,  13.,  10.,   0.,   0.,   0.]])
```

You can visually check the contents of this using the matplotlib library.

```
import matplotlib.pyplot as plt
%matplotlib inline
plt.imshow(digits.images[0], cmap=plt.cm.gray_r, interpolation='nearest')
```

By launching this command, you will obtain a grayscale image as shown in Figure 11-4.
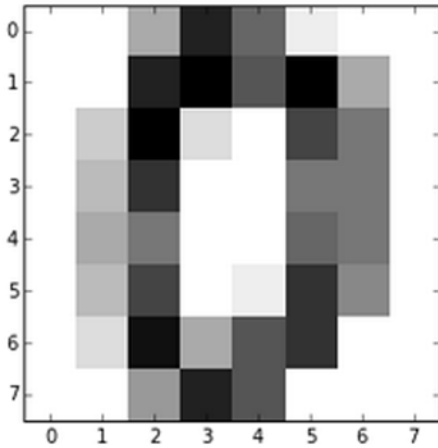
Figure 11-4. *One of the 1,797 handwritten digits*

As for the numerical values which are represented by the images, i.e., the targets, they are contained within the **digit.targets** array.

```
digits.target
array([0, 1, 2, ..., 8, 9, 8])
```

It was reported that the dataset is a training set consisting of 1,797 images. You can check if it is true.

```
digits.target.size
1797
```

Learning and Predicting

Now that you have loaded the Digits datasets in your notebook and you have defined an SVC estimator, you can start with the learning.

As you've already seen in Chapter 8, once you defined a predictive model, you must instruct it with a training set, a set of data in which you already know the belonging class. Given the large quantity of elements contained within the digits dataset, you will certainly obtain a very effective model, i.e., one which is capable of being able to recognize with good certainty the handwritten number.

The dataset consists of 1,797 elements, and so we can consider the first 1,791 as a training set and will use the last 6 as validation set.

You can see in detail these 6 handwritten digits, using again the matplotlib library:

```
import matplotlib.pyplot as plt
%matplotlib inline

plt.subplot(321)
plt.imshow(digits.images[1791], cmap=plt.cm.gray_r, interpolation='nearest')
plt.subplot(322)
plt.imshow(digits.images[1792], cmap=plt.cm.gray_r, interpolation='nearest')
plt.subplot(323)
plt.imshow(digits.images[1793], cmap=plt.cm.gray_r, interpolation='nearest')
plt.subplot(324)
plt.imshow(digits.images[1794], cmap=plt.cm.gray_r, interpolation='nearest')
plt.subplot(325)
plt.imshow(digits.images[1795], cmap=plt.cm.gray_r, interpolation='nearest')
plt.subplot(326)
plt.imshow(digits.images[1796], cmap=plt.cm.gray_r, interpolation='nearest')
```

It will produce an image with 6 digits as in Figure 11-5.
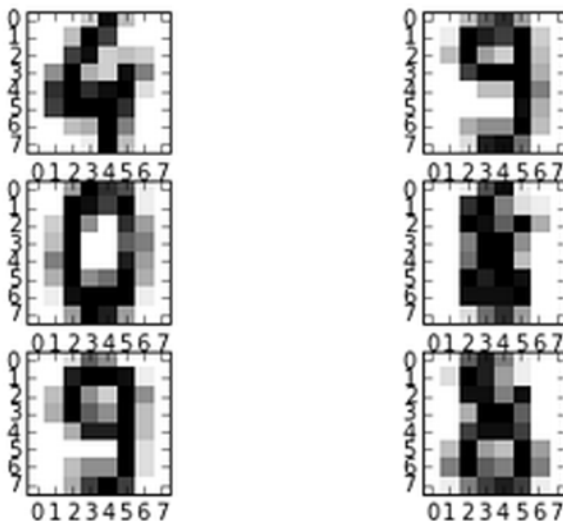
Figure 11-5. *The six digits of the validation set*

Now you can do the learning of the **svc** estimator that you defined earlier.

```
svc.fit(digits.data[1:1790], digits.target[1:1790])
```

After a short time, the trained estimator will appear with a text output.

```
SVC(C=100.0, cache_size=200, class_weight=None, coef0=0.0, degree=3,
  gamma=0.001, kernel='rbf', max_iter=-1, probability=False,
  random_state=None, shrinking=True, tol=0.001, verbose=False)
```

Now you have to test your estimator, making it to interpret the 6 digits of the validation set.

```
svc.predict(digits.data[1791:1976])
```

and you will obtain these results

```
array([4, 9, 0, 8, 9, 8])
```

if you compare them with the actual digits

```
digits.target[1791:1976]
```

```
array([4, 9, 0, 8, 9, 8])
```

You can see that the svc estimator has been learnt in a correct way. It is able to recognize the handwritten digits, interpreting correctly all the 6 digits of the validation set.

Conclusions

In this short chapter you have seen how many application possibilities this analysis of data has. It is not limited to the analysis of numerical or textual data but also can analyze images, as may be the handwritten digits read by a camera or a scanner.

Furthermore, you have seen that predictive models can provide truly optimal results thanks to machine learning techniques which are easily implemented thanks to the scikit-learn library.

■ ■ ■

Writing Mathematical Expressions with LaTeX

LaTeX is extensively used in Python. In this appendix there are many examples that can be useful to represent LaTeX expressions inside Python implementations. This same information can be found at the link http://matplotlib.org/users/mathtext.html.

With matplotlib

You can enter the LaTeX expression directly as an argument of various functions that can accept it. For example, the title() function that draws a chart title.

```
import matplotlib.pyplot as plt
%matplotlib inline
plt.title(r'$\alpha > \beta$')
```

With IPython Notebook in a Markdown Cell

You can enter the LaTeX expression between two '$$'.

```
$$c = \sqrt{a^2 + b^2}$$
```

$c = \sqrt{a^2 + b^2}$

With IPython Notebook in a Python 2 Cell

You can enter the LaTeX expression within the Math() function.

```
from IPython.display import display, Math, Latex
display(Math(r'F(k) = \int_{-\infty}^{\infty} f(x) e^{2\pi i k} dx'))
```

Subscripts and Superscripts

To make subscripts and superscripts, use the '_' and '^' symbols:
    r'$\alpha_i > \beta_i$'

$$\alpha_i > \beta_i$$

This could be very useful when you have to write summations:
    r'$\sum_{i=0}^\infty x_i$'

$$\sum_{i=0}^{\infty} x_i$$

Fractions, Binomials, and Stacked Numbers

Fractions, binomials, and stacked numbers can be created with the \frac{}{}, \binom{}{}, and \stackrel{}{}
commands, respectively:
    r'$\frac{3}{4} \binom{3}{4} \stackrel{3}{4}$'

$$\frac{3}{4} \binom{3}{4} \stackrel{3}{4}$$

Fractions can be arbitrarily nested:

$$\frac{5 - \frac{1}{x}}{4}$$

Note that special care needs to be taken to place parentheses and brackets around fractions. You have to
insert \left and \right preceding the bracket in order to inform the parser that those brackets encompass the
entire object:

$$\left(\frac{5 - \frac{1}{x}}{4} \right)$$

Radicals

Radicals can be produced with the \sqrt[]{} command.
    r'$\sqrt{2}$'

$$\sqrt{2}$$

Fonts

The default font is italics for mathematical symbols. To change fonts, for example with trigonometric functions as sin:

$$s(t) = A sin(2\omega t)$$

The choices available with all fonts are

```
from IPython.display import display, Math, Latex
display(Math(r'\mathrm{Roman}'))
display(Math(r'\mathit{Italic}'))
display(Math(r'\mathtt{Typewriter}'))
display(Math(r'\mathcal{CALLIGRAPHY}'))
```

Roman

Italic

Typewriter

$\mathcal{CALLIGRAPHY}$

Accents

An accent command may precede any symbol to add an accent above it. There are long and short forms for some of them.

\acute a or \'a	$\acute{a}$
\bar a	$\bar{a}$
\breve a	$\breve{a}$
\ddot a or \"a	$\ddot{a}$
\dot a or \.a	$\dot{a}$
\grave a or \`a	$\grave{a}$
\hat a or \^a	$\hat{a}$
\tilde a or \~a	$\tilde{a}$
\vec a	$\vec{a}$
\overline{abc}	$\overline{abc}$

Symbols
You can also use a large number of the TeX symbols.

Lowercase Greek

α \alpha	β \beta	χ \chi	δ \delta	F \digamma
ϵ \epsilon	η \eta	γ \gamma	ι \iota	κ \kappa
λ \lambda	μ \mu	ν \nu	ω \omega	ϕ \phi
π \pi	ψ \psi	ρ \rho	σ \sigma	τ \tau
θ \theta	υ \upsilon	ε \varepsilon	$\varkappa$ \varkappa	φ \varphi
ϖ \varpi	ϱ \varrho	ς \varsigma	ϑ \vartheta	ξ \xi
ζ \zeta				

Uppercase Greek

Δ \Delta	Γ \Gamma	Λ \Lambda	Ω \Omega	Φ \Phi	Π \Pi
Ψ \Psi	Σ \Sigma	Θ \Theta	Υ \Upsilon	Ξ \Xi	$\mho$ \mho
∇ \nabla					

Hebrew

$\aleph$ \aleph	$\beth$ \beth	$\daleth$ \daleth	$\gimel$ \gimel

Delimiters

$/$ /	$[$ [	$\Downarrow$ \Downarrow	$\Uparrow$ \Uparrow	$\|$ \Vert	$\backslash$ \backslash
$\downarrow$ \downarrow	$\langle$ \langle	$\lceil$ \lceil	$\lfloor$ \lfloor	$\llcorner$ \llcorner	$\lrcorner$ \lrcorner
$\rangle$ \rangle	$\rceil$ \rceil	$\rfloor$ \rfloor	$\ulcorner$ \ulcorner	$\uparrow$ \uparrow	$\urcorner$ \urcorner
$\vert$ \vert	$\{$ \{	$\|$ |	$\}$ \}	$]$]	$|$	

Big Symbols

⋂ \bigcap	⋃ \bigcup	⊙ \bigodot	⊕ \bigoplus	⊗ \bigotimes
⊎ \biguplus	⋁ \bigvee	⋀ \bigwedge	∐ \coprod	∫ \int
∮ \oint	∏ \prod	∑ \sum		

Standard Function Names

Pr \Pr	arccos \arccos	arcsin \arcsin	arctan \arctan
arg \arg	cos \cos	cosh \cosh	cot \cot
coth \coth	csc \csc	deg \deg	det \det
dim \dim	exp \exp	gcd \gcd	hom \hom
inf \inf	ker \ker	lg \lg	lim \lim
liminf \liminf	limsup \limsup	ln \ln	log \log
max \max	min \min	sec \sec	sin \sin
sinh \sinh	sup \sup	tan \tan	tanh \tanh

Binary Operation and Relation Symbols

≎ \Bumpeq	⋓ \Cap	⋒ \Cup
≑ \Doteq	⋈ \Join	⋐ \Subset
⋑ \Supset	⊩ \Vdash	⊪ \Vvdash
≈ \approx	≊ \approxeq	∗ \ast
≍ \asymp	∍ \backepsilon	∽ \backsim
≌ \backsimeq	$\overline{wedge}$ \barwedge	∵ \because
⨃ \between	○ \bigcirc	▽ \bigtriangledown

△ \bigtriangleup	◀ \blacktriangleleft	▶ \blacktriangleright
⊥ \bot	⋈ \bowtie	⊡ \boxdot
⊟ \boxminus	⊞ \boxplus	⊠ \boxtimes
● \bullet	≏ \bumpeq	∩ \cap
․ \cdot	○ \circ	≗ \circeq
≔ \coloneq	≅ \cong	∪ \cup
⋞ \curlyeqprec	⋟ \curlyeqsucc	⋎ \curlyvee
⋏ \curlywedge	† \dag	⊣ \dashv
‡ \ddag	◇ \diamond	÷ \div
⊛ \divideontimes	ė𝑞 \doteq	ė𝑞𝑑𝑜𝑡 \doteqdot
𝑝𝑙𝑢𝑠 \dotplus	⊼ \doublebarwedge	⊺ \eqcirc
≕ \eqcolon	≂ \eqsim	⪖ \eqslantgtr
⪕ \eqslantless	≡ \equiv	≒ \fallingdotseq
⌢ \frown	≥ \geq	≧ \geqq
⩾ \geqslant	≫ \gg	⋙ \ggg
⪊ \gnapprox	≩ \gneqq	⋧ \gnsim
⪆ \gtrapprox	⋗ \gtrdot	⋛ \gtreqless
⪌ \gtreqqless	≷ \gtrless	≳ \gtrsim
∈ \in	⊺ \intercal	⋋ \leftthreetimes
≤ \leq	≦ \leqq	⩽ \leqslant
⪅ \lessapprox	⋖ \lessdot	⪋ \lesseqgtr
⪑ \lesseqqgtr	≶ \lessgtr	≲ \lesssim
≪ \ll	⋘ \lll	⪉ \lnapprox
≨ \lneqq	⋦ \lnsim	⋉ \ltimes

∣ \mid	⊨ \models	∓ \mp
⊭ \nVDash	⊯ \nVdash	≉ \napprox
≇ \ncong	≠ \ne	≠ \neq
≠ \neq	≢ \nequiv	≱ \ngeq
≯ \ngtr	∋ \ni	≰ \nleq
≮ \nless	∤ \nmid	∉ \notin
∦ \nparallel	⊀ \nprec	≁ \nsim
⊄ \nsubset	⊈ \nsubseteq	⊁ \nsucc
⊅ \nsupset	⊉ \nsupseteq	⋪ \ntriangleleft
⋬ \ntrianglelefteq	⋫ \ntriangleright	⋭ \ntrianglerighteq
⊭ \nvDash	⊬ \nvdash	⊙ \odot
⊖ \ominus	⊕ \oplus	⊘ \oslash
⊗ \otimes	∥ \parallel	⊥ \perp
⋔ \pitchfork	± \pm	≺ \prec
⪷ \precapprox	≼ \preccurlyeq	⪯ \preceq
⪹ \precnapprox	⋨ \precnsim	≾ \precsim
∝ \propto	⋋ \rightthreetimes	≓ \risingdotseq
⋊ \rtimes	∼ \sim	≃ \simeq
╱ \slash	⌣ \smile	⊓ \sqcap
⊔ \sqcup	⊏ \sqsubset	⊏ \sqsubset
⊑ \sqsubseteq	⊐ \sqsupset	⊐ \sqsupset
⊒ \sqsupseteq	⋆ \star	⊂ \subset
⊆ \subseteq	⫅ \subseteqq	⊊ \subsetneq
⫋ \subsetneqq	≻ \succ	⪸ \succapprox
≽ \succcurlyeq	⪰ \succeq	⪺ \succnapprox
⋩ \succnsim	≿ \succsim	⊃ \supset

⊇ \supseteq	⊇̲ \supseteqq	⊋ \supsetneq
⊋̲ \supsetneqq	∴ \therefore	× \times
⊤ \top	◁ \triangleleft	⊴ \trianglelefteq
≜ \triangleq	▷ \triangleright	⊵ \trianglerighteq
⊎ \uplus	⊨ \vDash	∝ \varpropto
◁ \vartriangleleft	▷ \vartriangleright	⊢ \vdash
∨ \vee	⊻ \veebar	∧ \wedge
≀ \wr		

Arrow Symbols

⇓ \Downarrow	⇐ \Leftarrow
⇔ \Leftrightarrow	⇚ \Lleftarrow
⟸ \Longleftarrow	⟺ \Longleftrightarrow
⟹ \Longrightarrow	↰ \Lsh
⤢ \Nearrow	⤡ \Nwarrow
⇒ \Rightarrow	⇛ \Rrightarrow
↱ \Rsh	⤡ \Searrow
⤢ \Swarrow	⇑ \Uparrow
⇕ \Updownarrow	↺ \circlearrowleft
↻ \circlearrowright	↶ \curvearrowleft
↷ \curvearrowright	⇠ \dashleftarrow
⇢ \dashrightarrow	↓ \downarrow
⇊ \downdownarrows	⇃ \downharpoonleft
⇂ \downharpoonright	↩ \hookleftarrow
↪ \hookrightarrow	↝ \leadsto
← \leftarrow	↢ \leftarrowtail
↽ \leftharpoondown	↼ \leftharpoonup

(*continued*)

⇐ \leftleftarrows	↔ \leftrightarrow
⇆ \leftrightarrows	⇋ \leftrightharpoons
↭ \leftrightsquigarrow	↜ \leftsquigarrow
⟵ \longleftarrow	⟷ \longleftrightarrow
⟼ \longmapsto	⟶ \longrightarrow
↫ \looparrowleft	↬ \looparrowright
↦ \mapsto	⊸ \multimap
⇍ \nLeftarrow	⇎ \nLeftrightarrow
⇏ \nRightarrow	↗ \nearrow
↚ \nleftarrow	↮ \nleftrightarrow
↛ \nrightarrow	↖ \nwarrow
→ \rightarrow	↣ \rightarrowtail
⇁ \rightharpoondown	⇀ \rightharpoonup
⇌ \rightleftarrows	⇌ \rightleftarrows
⇌ \rightleftharpoons	⇌ \rightleftharpoons
⇉ \rightrightarrows	⇉ \rightrightarrows
↝ \rightsquigarrow	↘ \searrow
↙ \swarrow	→ \to
↞ \twoheadleftarrow	↠ \twoheadrightarrow
↑ \uparrow	↕ \updownarrow
↕ \updownarrow	↿ \upharpoonleft
↾ \upharpoonright	⇈ \upuparrows

Miscellaneous Symbols

$ \\$	Å \AA	⅁ \Finv
⅁ \Game	ℑ \Im	¶ \P
ℜ \Re	§ \S	∠ \angle
‵ \backprime	★ \bigstar	■ \blacksquare
▲ \blacktriangle	▼ \blacktriangledown	⋯ \cdots
✓ \checkmark	® \circledR	Ⓢ \circledS

(*continued*)

325

♣ \clubsuit	С \complement	© \copyright
⋰ \ddots	◇ \diamondsuit	ℓ \ell
∅ \emptyset	ð \eth	∃ \exists
♭ \flat	∀ \forall	ℏ \hbar
♡ \heartsuit	ℏ \hslash	∭ \iiint
∬ \iint	∬ \iint	ɀ \imath
∞ \infty	ȷ \jmath	… \ldots
∡ \measuredangle	♮ \natural	¬ \neg
∄ \nexists	∰ \oiiint	∂ \partial
′ \prime	♯ \sharp	♠ \spadesuit
∢ \sphericalangle	β \ss	▽ \triangledown
∅ \varnothing	△ \vartriangle	⋮ \vdots
℘ \wp	¥ \yen	

APPENDIX B

■ ■ ■ ■

Open Data Sources

Political and Government Data

Data.gov

http://data.gov

This is the resource for most government-related data.

Socrata

http://www.socrata.com/resources/

Socrata is a good place to explore government-related data. Furthermore, it provides some visualization tools for exploring data.

US Census Bureau

http://www.census.gov/data.html

This site provides information about US citizens covering population data, geographic data, and education.

UN3ta

https://data.un.org/

UNdata is an Internet-based data service which brings UN statistical databases.

European Union Open Data Portal

http://open-data.europa.eu/en/data/

This site provides a lot of data from European Union institutions.

Data.gov.uk

http://data.gov.uk/

This site of the UK Government includes the British National Bibliography: metadata on all UK books and publications since 1950.

The CIA World Factbook

https://www.cia.gov/library/publications/the-world-factbook/

This site of the Central Intelligence Agency provides a lot of information on history, population, economy, government, infrastructure, and military of 267 countries.

Health Data

Healthdata.gov

https://www.healthdata.gov/

This site provides medical data about epidemiology and population statistics.

NHS Health and Social Care Information Centre

http://www.hscic.gov.uk/home

Health data sets from the UK National Health Service.

Social Data

Facebook Graph

https://developers.facebook.com/docs/graph-api

Facebook provides this API which allows you to query the huge amount of information that users are sharing with the world.

Topsy

http://topsy.com/

Topsy provides a searchable database of public tweets going back to 2006 as well as several tools to analyze the conversations.

Google Trends

http://www.google.com/trends/explore

Statistics on search volume (as a proportion of total search) for any given term, since 2004.

Likebutton

http://likebutton.com/

Mines Facebook's public data—globally and from your own network—to give an overview of what people "Like" at the moment.

Miscellaneous and Public Data Sets

Amazon Web Services public datasets

http://aws.amazon.com/datasets

The public data sets on Amazon Web Services provide a centralized repository of public data sets. An interesting dataset is the 1000 Genome Project, an attempt to build the most comprehensive database of human genetic information. Also a NASA database of satellite imagery of Earth is available.

DBPedia

http://wiki.dbpedia.org

Wikipedia contains millions of pieces of data, structured and unstructured, on every subject. DBPedia is an ambitious project to catalogue and create a public, freely distributable database allowing anyone to analyze this data.

Freebase

http://www.freebase.com/

This community database provides information about several topics, with over 45 million entries.

Gapminder

http://www.gapminder.org/data/

This site provides data coming from the World Health Organization and World Bank covering economic, medical, and social statistics from around the world.

Financial Data

Google Finance

https://www.google.com/finance

Forty years' worth of stock market data, updated in real time.

Climatic Data

National Climatic Data Center

http://www.ncdc.noaa.gov/data-access/quick-links#loc-clim

Huge collection of environmental, meteorological, and climate data sets from the US National Climatic Data Center. The world's largest archive of weather data.

WeatherBase

http://www.weatherbase.com/

This site provides climate averages, forecasts, and current conditions for over 40,000 cities worldwide.

Wunderground

http://www.wunderground.com/

This site provides climatic data from satellites and weather stations, allowing you to get all information about the temperature, wind, and other climatic measurements.

Sports Data

Pro-Football-Reference

http://www.pro-football-reference.com/

This site provides data about football and several other sports.

Publications, Newspapers, and Books

New York Times

http://developer.nytimes.com/docs

Searchable, indexed archive of news articles going back to 1851.

Google Books Ngrams

http://storage.googleapis.com/books/ngrams/books/datasetsv2.html

This source searches and analyzes the full text of any of the millions of books digitized as part of the Google Books project.

Musical Data

Million Song Data Set

http://aws.amazon.com/datasets/6468931156960467

Metadata on over a million songs and pieces of music. Part of Amazon Web Services.

Index