Toolboxes for Data Scientists

2

2.1 Introduction

In this chapter, first we introduce some of the tools that data scientists use. The toolbox of any data scientist, as for any kind of programmer, is an essential ingredient for success and enhanced performance. Choosing the right tools can save a lot of time and thereby allow us to focus on data analysis.

The most basic tool to decide on is which programming language we will use. Many people use only one programming language in their entire life: the first and only one they learn. For many, learning a new language is an enormous task that, if at all possible, should be undertaken only once. The problem is that some languages are intended for developing high-performance or production code, such as C, C++, or Java, while others are more focused on prototyping code, among these the best known are the so-called scripting languages: Ruby, Perl, and Python. So, depending on the first language you learned, certain tasks will, at the very least, be rather tedious. The main problem of being stuck with a single language is that many basic tools simply will not be available in it, and eventually you will have either to reimplement them or to create a bridge to use some other language just for a specific task.

L. Igual and S. Seguí, *Introduction to Data Science*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-50017-1_2

In conclusion, you either have to be ready to change to the best language for each task and then glue the results together, or choose a very flexible language with a rich ecosystem (e.g., third-party open-source libraries). In this book we have selected Python as the programming language.

2.2 Why Python?

Python[1] is a mature programming language but it also has excellent properties for newbie programmers, making it ideal for people who have never programmed before. Some of the most remarkable of those properties are easy to read code, suppression of non-mandatory delimiters, dynamic typing, and dynamic memory usage. Python is an interpreted language, so the code is executed immediately in the Python console without needing the compilation step to machine language. Besides the Python console (which comes included with any Python installation) you can find other interactive consoles, such as IPython,[2] which give you a richer environment in which to execute your Python code.

Currently, Python is one of the most flexible programming languages. One of its main characteristics that makes it so flexible is that it can be seen as a multiparadigm language. This is especially useful for people who already know how to program with other languages, as they can rapidly start programming with Python in the same way. For example, Java programmers will feel comfortable using Python as it supports the object-oriented paradigm, or C programmers could mix Python and C code using *cython*. Furthermore, for anyone who is used to programming in functional languages such as Haskell or Lisp, Python also has basic statements for functional programming in its own core library.

In this book, we have decided to use Python language because, as explained before, it is a mature language programming, easy for the newbies, and can be used as a specific platform for data scientists, thanks to its large ecosystem of scientific libraries and its high and vibrant community. Other popular alternatives to Python for data scientists are R and MATLAB/Octave.

2.3 Fundamental Python Libraries for Data Scientists

The Python community is one of the most active programming communities with a huge number of developed toolboxes. The most popular Python toolboxes for any data scientist are NumPy, SciPy, Pandas, and Scikit-Learn.

[1] https://www.python.org/downloads/.
[2] http://ipython.org/install.html.

2.3.1 Numeric and Scientific Computation: NumPy and SciPy

NumPy[3] is the cornerstone toolbox for scientific computing with Python. NumPy provides, among other things, support for multidimensional arrays with basic operations on them and useful linear algebra functions. Many toolboxes use the NumPy array representations as an efficient basic data structure. Meanwhile, *SciPy* provides a collection of numerical algorithms and domain-specific toolboxes, including signal processing, optimization, statistics, and much more. Another core toolbox in SciPy is the plotting library *Matplotlib*. This toolbox has many tools for data visualization.

2.3.2 SCIKIT-Learn: Machine Learning in Python

Scikit-learn[4] is a machine learning library built from NumPy, SciPy, and Matplotlib. Scikit-learn offers simple and efficient tools for common tasks in data analysis such as classification, regression, clustering, dimensionality reduction, model selection, and preprocessing.

2.3.3 PANDAS: Python Data Analysis Library

Pandas[5] provides high-performance data structures and data analysis tools. The key feature of Pandas is a fast and efficient DataFrame object for data manipulation with integrated indexing. The DataFrame structure can be seen as a spreadsheet which offers very flexible ways of working with it. You can easily transform any dataset in the way you want, by reshaping it and adding or removing columns or rows. It also provides high-performance functions for aggregating, merging, and joining datasets. Pandas also has tools for importing and exporting data from different formats: comma-separated value (CSV), text files, Microsoft Excel, SQL databases, and the fast HDF5 format. In many situations, the data you have in such formats will not be complete or totally structured. For such cases, Pandas offers handling of missing data and intelligent data alignment. Furthermore, Pandas provides a convenient Matplotlib interface.

2.4 Data Science Ecosystem Installation

Before we can get started on solving our own data-oriented problems, we will need to set up our programming environment. The first question we need to answer concerns

[3] http://www.scipy.org/scipylib/download.html.
[4] http://www.scipy.org/scipylib/download.html.
[5] http://pandas.pydata.org/getpandas.html.

Python language itself. There are currently two different versions of Python: Python 2.X and Python 3.X. The differences between the versions are important, so there is no compatibility between the codes, i.e., code written in Python 2.X does not work in Python 3.X and vice versa. Python 3.X was introduced in late 2008; by then, a lot of code and many toolboxes were already deployed using Python 2.X (Python 2.0 was initially introduced in 2000). Therefore, much of the scientific community did not change to Python 3.0 immediately and they were stuck with Python 2.7. By now, almost all libraries have been ported to Python 3.0; but Python 2.7 is still maintained, so one or another version can be chosen. However, those who already have a large amount of code in 2.X rarely change to Python 3.X. In our examples throughout this book we will use Python 2.7.

Once we have chosen one of the Python versions, the next thing to decide is whether we want to install the data scientist Python ecosystem by individual toolboxes, or to perform a bundle installation with all the needed toolboxes (and a lot more). For newbies, the second option is recommended. If the first option is chosen, then it is only necessary to install all the mentioned toolboxes in the previous section, in exactly that order.

However, if a bundle installation is chosen, the Anaconda Python distribution[6] is then a good option. The Anaconda distribution provides integration of all the Python toolboxes and applications needed for data scientists into a single directory without mixing it with other Python toolboxes installed on the machine. It contains, of course, the core toolboxes and applications such as NumPy, Pandas, SciPy, Matplotlib, Scikit-learn, IPython, Spyder, etc., but also more specific tools for other related tasks such as data visualization, code optimization, and big data processing.

2.5 Integrated Development Environments (IDE)

For any programmer, and by extension, for any data scientist, the integrated development environment (IDE) is an essential tool. IDEs are designed to maximize programmer productivity. Thus, over the years this software has evolved in order to make the coding task less complicated. Choosing the right IDE for each person is crucial and, unfortunately, there is no "one-size-fits-all" programming environment. The best solution is to try the most popular IDEs among the community and keep whichever fits better in each case.

In general, the basic pieces of any IDE are three: the editor, the compiler, (or interpreter) and the debugger. Some IDEs can be used in multiple programming languages, provided by language-specific plugins, such as Netbeans[7] or Eclipse.[8] Others are only specific for one language or even a specific programming task. In

[6] http://continuum.io/downloads.
[7] https://netbeans.org/downloads/.
[8] https://eclipse.org/downloads/.

the case of Python, there are a large number of specific IDEs, both commercial (PyCharm,[9] WingIDE[10] …) and open-source. The open-source community helps IDEs to spring up, thus anyone can customize their own environment and share it with the rest of the community. For example, Spyder[11] (Scientific Python Development EnviRonment) is an IDE customized with the task of the data scientist in mind.

2.5.1 Web Integrated Development Environment (WIDE): Jupyter

With the advent of web applications, a new generation of IDEs for interactive languages such as Python has been developed. Starting in the academia and e-learning communities, web-based IDEs were developed considering how not only your code but also all your environment and executions can be stored in a server. One of the first applications of this kind of WIDE was developed by William Stein in early 2005 using Python 2.3 as part of his SageMath mathematical software. In SageMath, a server can be set up in a center, such as a university or school, and then students can work on their homework either in the classroom or at home, starting from exactly the same point they left off. Moreover, students can execute all the previous steps over and over again, and then change some particular *code cell* (a segment of the document that may content source code that can be executed) and execute the operation again. Teachers can also have access to student sessions and review the progress or results of their pupils.

Nowadays, such sessions are called notebooks and they are not only used in classrooms but also used to show results in presentations or on business dashboards. The recent spread of such notebooks is mainly due to IPython. Since December 2011, IPython has been issued as a browser version of its interactive console, called IPython notebook, which shows the Python execution results very clearly and concisely by means of cells. Cells can contain content other than code. For example, markdown (a wiki text language) cells can be added to introduce algorithms. It is also possible to insert Matplotlib graphics to illustrate examples or even web pages. Recently, some scientific journals have started to accept notebooks in order to show experimental results, complete with their code and data sources. In this way, experiments can become completely and absolutely replicable.

Since the project has grown so much, IPython notebook has been separated from IPython software and now it has become a part of a larger project: Jupyter[12]. Jupyter (for Julia, Python and R) aims to reuse the same WIDE for all these interpreted languages and not just Python. All old IPython notebooks are automatically imported to the new version when they are opened with the Jupyter platform; but once they

[9] https://www.jetbrains.com/pycharm/.
[10] https://wingware.com/.
[11] https://github.com/spyder-ide/spyder.
[12] http://jupyter.readthedocs.org/en/latest/install.html.

are converted to the new version, they cannot be used again in old IPython notebook versions.

In this book, all the examples shown use Jupyter notebook style.

2.6 Get Started with Python for Data Scientists

Throughout this book, we will come across many practical examples. In this chapter, we will see a very basic example to help get started with a data science ecosystem from scratch. To execute our examples, we will use Jupyter notebook, although any other console or IDE can be used.

The Jupyter Notebook Environment

Once all the ecosystem is fully installed, we can start by launching the Jupyter notebook platform. This can be done directly by typing the following command on your terminal or command line: `$ jupyter notebook`

If we chose the bundle installation, we can start the Jupyter notebook platform by clicking on the Jupyter Notebook icon installed by Anaconda in the start menu or on the desktop.

The browser will immediately be launched displaying the Jupyter notebook home-page, whose URL is http://localhost:8888/tree. Note that a special port is used; by default it is 8888. As can be seen in Fig. 2.1, this initial page displays a tree view of a directory. If we use the command line, the root directory is the same directory where we launched the Jupyter notebook. Otherwise, if we use the Anaconda launcher, the root directory is the current user directory. Now, to start a new notebook, we only need to press the New Notebooks Python 2 button at the top on the right of the home page.

As can be seen in Fig. 2.2, a blank notebook is created called `Untitled`. First of all, we are going to change the name of the notebook to something more appropriate. To do this, just click on the notebook name and rename it: `DataScience-GetStartedExample`.

Let us begin by importing those toolboxes that we will need for our program. In the first cell we put the code to import the *Pandas* library as `pd`. This is for convenience; every time we need to use some functionality from the Pandas library, we will write `pd` instead of `pandas`. We will also import the two core libraries mentioned above: the numpy library as `np` and the matplotlib library as `plt`.

In []:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

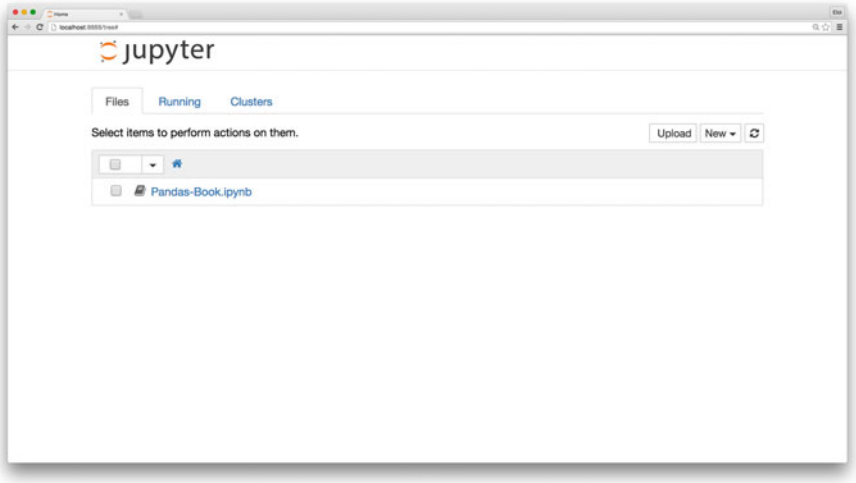


Fig. 2.1 IPython notebook home page, displaying a home tree directory

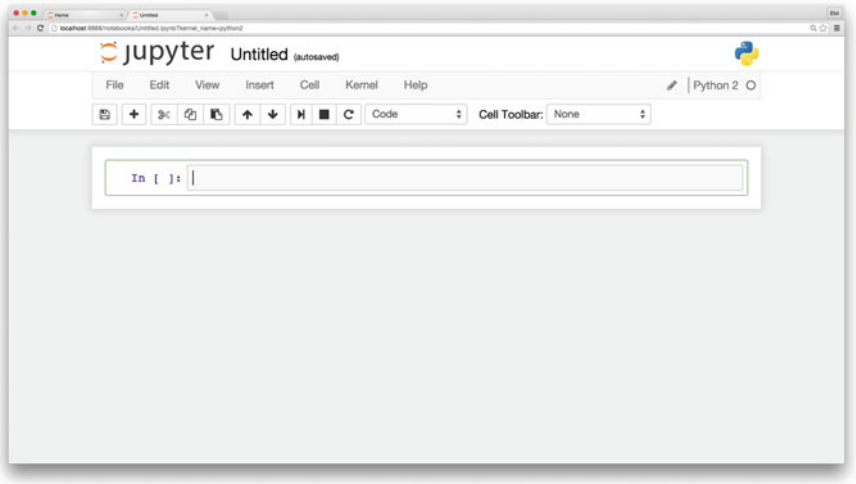


Fig. 2.2 An empty new notebook

To execute just one cell, we press the ▶I button or click on Cell > Run or press the keys Ctrl + Enter. While execution is underway, the header of the cell shows the * mark:

In [*]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

While a cell is being executed, no other cell can be executed. If you try to execute another cell, its execution will not start until the first cell has finished its execution.

Once the execution is finished, the header of the cell will be replaced by the next number of execution. Since this will be the first cell executed, the number shown will be 1. If the process of importing the libraries is correct, no output cell is produced.

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

For simplicity, other chapters in this book will avoid writing these imports.

The DataFrame Data Structure

The key data structure in Pandas is the `DataFrame` object. A DataFrame is basically a tabular data structure, with rows and columns. Rows have a specific index to access them, which can be any name or value. In Pandas, the columns are called `Series`, a special type of data, which in essence consists of a list of several values, where each value has an index. Therefore, the DataFrame data structure can be seen as a spreadsheet, but it is much more flexible. To understand how it works, let us see how to create a DataFrame from a common Python dictionary of lists. First, we will create a new cell by clicking Insert > Insert Cell Below or pressing the keys Ctrl + B. Then, we write in the following code:

In [2]:
```
data = {'year': [
         2010, 2011, 2012,
         2010, 2011, 2012,
         2010, 2011, 2012
     ],
     'team': [
         'FCBarcelona', 'FCBarcelona',
         'FCBarcelona', 'RMadrid',
         'RMadrid', 'RMadrid',
         'ValenciaCF', 'ValenciaCF',
         'ValenciaCF'
     ],
     'wins':   [30, 28, 32, 29, 32, 26, 21, 17, 19],
     'draws':  [6, 7, 4, 5, 4, 7, 8, 10, 8],
     'losses': [2, 3, 2, 4, 2, 5, 9, 11, 11]
     }
football = pd.DataFrame(data, columns = [
     'year', 'team', 'wins', 'draws', 'losses'
     ]
)
```

In this example, we use the pandas DataFrame object constructor with a dictionary of lists as argument. The value of each entry in the dictionary is the name of the column, and the lists are their values.

The DataFrame columns can be arranged at construction time by entering a keyword `columns` with a list of the names of the columns ordered as we want. If the

column keyword is not present in the constructor, the columns will be arranged in alphabetical order. Now, if we execute this cell, the result will be a table like this:

Out[2]:

	year	team	wins	draws	losses
0	2010	FCBarcelona	30	6	2
1	2011	FCBarcelona	28	7	3
2	2012	FCBarcelona	32	4	2
3	2010	RMadrid	29	5	4
4	2011	RMadrid	32	4	2
5	2012	RMadrid	26	7	5
6	2010	ValenciaCF	21	8	9
7	2011	ValenciaCF	17	10	11
8	2012	ValenciaCF	19	8	11

where each entry in the dictionary is a column. The index of each row is created automatically taking the position of its elements inside the entry lists, starting from 0. Although it is very easy to create DataFrames from scratch, most of the time what we will need to do is import chunks of data into a DataFrame structure, and we will see how to do this in later examples.

Apart from DataFrame data structure creation, Panda offers a lot of functions to manipulate them. Among other things, it offers us functions for aggregation, manipulation, and transformation of the data. In the following sections, we will introduce some of these functions.

Open Government Data Analysis Example Using Pandas

To illustrate how we can use Pandas in a simple real problem, we will start doing some basic analysis of government data. For the sake of transparency, data produced by government entities must be open, meaning that they can be freely used, reused, and distributed by anyone. An example of this is the Eurostat, which is the home of European Commission data. Eurostat's main role is to process and publish comparable statistical information at the European level. The data in Eurostat are provided by each member state and it is free to reuse them, for both noncommercial and commercial purposes (with some minor exceptions).

Since the amount of data in the Eurostat database is huge, in our first study we are only going to focus on data relative to indicators of educational funding by the member states. Thus, the first thing to do is to retrieve such data from Eurostat. Since open data have to be delivered in a plain text format, CSV (or any other delimiter-separated value) formats are commonly used to store tabular data. In a delimiter-separated value file, each line is a data record and each record consists of one or more fields, separated by the delimiter character (usually a comma). Therefore, the data we will use can be found already processed at book's Github repository as `educ_figdp_1_Data.csv` file. Of course, it can also be downloaded as unprocessed tabular data from the Eurostat database site[13] following the path:

Tables by themes » Population and social conditions » Education and training » Education » Indicators on education finance » Public expenditure on education.

2.6.1 Reading

Let us start reading the data we downloaded. First of all, we have to create a new notebook called `Open Government Data Analysis` and open it. Then, after ensuring that the `educ_figdp_1_Data.csv` file is stored in the same directory as our notebook directory, we will write the following code to read and show the content:

In [1]:
```
edu = pd.read_csv('files/ch02/educ_figdp_1_Data.csv',
                  na_values = ':',
                  usecols = ["TIME","GEO","Value"])
edu
```

Out[1]:

	TIME	GEO	Value
0	2000	European Union ...	NaN
1	2001	European Union ...	NaN
2	2002	European Union ...	5.00
3	2003	European Union ...	5.03
...	...	...	...
382	2010	Finland	6.85
383	2011	Finland	6.76

384 rows × 5 columns

The way to read CSV (or any other separated value, providing the separator character) files in Pandas is by calling the `read_csv` method. Besides the name of the file, we add the `na_values` key argument to this method along with the character that represents "non available data" in the file. Normally, CSV files have a header with the names of the columns. If this is the case, we can use the `usecols` parameter to select which columns in the file will be used.

In this case, the DataFrame resulting from reading our data is stored in `edu`. The output of the execution shows that the `edu` DataFrame size is 384 rows × 3 columns. Since the DataFrame is too large to be fully displayed, three dots appear in the middle of each row.

Beside this, Pandas also has functions for reading files with formats such as Excel, HDF5, tabulated files, or even the content from the clipboard (`read_excel()`, `read_hdf()`, `read_table()`, `read_clipboard()`). Whichever function we use, the result of reading a file is stored as a DataFrame structure.

To see how the data looks, we can use the `head()` method, which shows just the first five rows. If we use a number as an argument to this method, this will be the number of rows that will be listed:

[13] http://ec.europa.eu/eurostat/data/database.

In [2]:
```
edu.head()
```

Out[2]:

	TIME	GEO	Value
0	2000	European Union ...	NaN
1	2001	European Union ...	NaN
2	2002	European Union ...	5.00
3	2003	European Union ...	5.03
4	2004	European Union ...	4.95

Similarly, it exists the `tail()` method, which returns the last five rows by default.

In [3]:
```
edu.tail()
```

Out[3]:

379	2007	Finland	5.90
380	2008	Finland	6.10
381	2009	Finland	6.81
382	2010	Finland	6.85
383	2011	Finland	6.76

If we want to know the names of the columns or the names of the indexes, we can use the DataFrame attributes `columns` and `index` respectively. The names of the columns or indexes can be changed by assigning a new list of the same length to these attributes. The values of any DataFrame can be retrieved as a Python array by calling its `values` attribute.

If we just want quick statistical information on all the numeric columns in a DataFrame, we can use the function `describe()`. The result shows the count, the mean, the standard deviation, the minimum and maximum, and the percentiles, by default, the 25th, 50th, and 75th, for all the values in each column or series.

In [4]:
```
edu.describe()
```

Out[4]:

	TIME	Value
count	384.000000	361.000000
mean	2005.500000	5.203989
std	3.456556	1.021694
min	2000.000000	2.880000
25%	2002.750000	4.620000
50%	2005.500000	5.060000
75%	2008.250000	5.660000
max	2011.000000	8.810000

Name: Value, dtype: float64

2.6.2 Selecting Data

If we want to select a subset of data from a DataFrame, it is necessary to indicate this subset using square brackets (`[ ]`) after the DataFrame. The subset can be specified in several ways. If we want to select only one column from a DataFrame, we only need to put its name between the square brackets. The result will be a Series data structure, not a DataFrame, because only one column is retrieved.

In [5]:
```
edu['Value']
```

Out[5]:
```
0    NaN
1    NaN
2   5.00
3   5.03
4   4.95
...  ...
380 6.10
381 6.81
382 6.85
383 6.76
Name: Value, dtype: float64
```

If we want to select a subset of rows from a DataFrame, we can do so by indicating a range of rows separated by a colon (`:`) inside the square brackets. This is commonly known as a *slice* of rows:

In [6]:
```
edu[10:14]
```

Out[6]:

	TIME	GEO	Value
10	2010	European Union (28 countries)	5.41
11	2011	European Union (28 countries)	5.25
12	2000	European Union (27 countries)	4.91
13	2001	European Union (27 countries)	4.99

This instruction returns the slice of rows from the 10th to the 13th position. Note that the slice does not use the index labels as references, but the position. In this case, the labels of the rows simply coincide with the position of the rows.

If we want to select a subset of columns and rows using the labels as our references instead of the positions, we can use `ix` indexing:

In [7]:
```
edu.ix[90:94, ['TIME','GEO']]
```

Out[7]:

	TIME	GEO
90	2006	Belgium
91	2007	Belgium
92	2008	Belgium
93	2009	Belgium
94	2010	Belgium

This returns all the rows between the indexes specified in the slice before the comma, and the columns specified as a list after the comma. In this case, `ix` references the index labels, which means that `ix` does not return the 90th to 94th rows, but it returns all the rows between the row labeled 90 and the row labeled 94; thus if the index 100 is placed between the rows labeled as 90 and 94, this row would also be returned.

2.6.3 Filtering Data

Another way to select a subset of data is by applying Boolean indexing. This indexing is commonly known as a *filter*. For instance, if we want to filter those values less than or equal to 6.5, we can do it like this:

In [8]:

```
edu[edu['Value'] > 6.5].tail()
```

Out[8]:

	TIME	GEO	Value
218	2002	Cyprus	6.60
281	2005	Malta	6.58
94	2010	Belgium	6.58
93	2009	Belgium	6.57
95	2011	Belgium	6.55

Boolean indexing uses the result of a Boolean operation over the data, returning a mask with True or False for each row. The rows marked True in the mask will be selected. In the previous example, the Boolean operation `edu['Value'] > 6.5` produces a Boolean mask. When an element in the "Value" column is greater than 6.5, the corresponding value in the mask is set to True, otherwise it is set to False. Then, when this mask is applied as an index in `edu[edu['Value'] > 6.5]`, the result is a filtered DataFrame containing only rows with values higher than 6.5. Of course, any of the usual Boolean operators can be used for filtering: $<$ (less than), $<=$ (less than or equal to), $>$ (greater than), $>=$ (greater than or equal to), $==$ (equal to), and $!=$ (not equal to).

2.6.4 Filtering Missing Values

Pandas uses the special value `NaN` (not a number) to represent missing values. In Python, `NaN` is a special floating-point value returned by certain operations when

Table 2.1 List of most common aggregation functions

Function	Description
count()	Number of non-null observations
sum()	Sum of values
mean()	Mean of values
median()	Arithmetic median of values
min()	Minimum
max()	Maximum
prod()	Product of values
std()	Unbiased standard deviation
var()	Unbiased variance

one of their results ends in an undefined value. A subtle feature of `NaN` values is that two `NaN` are never equal. Because of this, the only safe way to tell whether a value is missing in a DataFrame is by using the `isnull()` function. Indeed, this function can be used to filter rows with missing values:

In [9]:
```
edu[edu["Value"].isnull()].head()
```

Out[9]:

	TIME	GEO	Value
0	2000	European Union (28 countries)	NaN
1	2001	European Union (28 countries)	NaN
36	2000	Euro area (18 countries)	NaN
37	2001	Euro area (18 countries)	NaN
48	2000	Euro area (17 countries)	NaN

2.6.5 Manipulating Data

Once we know how to select the desired data, the next thing we need to know is how to manipulate data. One of the most straightforward things we can do is to operate with columns or rows using aggregation functions. Table 2.1 shows a list of the most common aggregation functions. The result of all these functions applied to a row or column is always a number. Meanwhile, if a function is applied to a DataFrame or a selection of rows and columns, then you can specify if the function should be applied to the rows for each column (setting the `axis=0` keyword on the invocation of the function), or it should be applied on the columns for each row (setting the `axis=1` keyword on the invocation of the function).

In [10]:
```
edu.max(axis = 0)
```

Out[10]:
```
TIME          2011
GEO          Spain
Value         8.81
dtype: object
```

Note that these are functions specific to Pandas, not the generic Python functions. There are differences in their implementation. In Python, `NaN` values propagate through all operations without raising an exception. In contrast, Pandas operations exclude `NaN` values representing missing data. For example, the pandas `max` function excludes `NaN` values, thus they are interpreted as missing values, while the standard Python `max` function will take the mathematical interpretation of `NaN` and return it as the maximum:

In [11]:
```
print "Pandas max function:", edu['Value'].max()
print "Python max function:", max(edu['Value'])
```

Out[11]:
```
Pandas max function: 8.81
Python max function: nan
```

Beside these aggregation functions, we can apply operations over all the values in rows, columns or a selection of both. The rule of thumb is that an operation between columns means that it is applied to each row in that column and an operation between rows means that it is applied to each column in that row. For example we can apply any binary arithmetical operation (+,-,*,/) to an entire row:

In [12]:
```
s = edu["Value"]/100
s.head()
```

Out[12]:
```
0          NaN
1          NaN
2       0.0500
3       0.0503
4       0.0495
Name: Value, dtype: float64
```

However, we can apply any function to a DataFrame or Series just setting its name as argument of the `apply` method. For example, in the following code, we apply the `sqrt` function from the NumPy library to perform the square root of each value in the `Value` column.

In [13]:
```
s = edu["Value"].apply(np.sqrt)
s.head()
```

Out[13]:
```
0            NaN
1            NaN
2       2.236068
3       2.242766
4       2.224860
Name: Value, dtype: float64
```

If we need to design a specific function to apply it, we can write an in-line function, commonly known as a λ-function. A λ-function is a function without a name. It is only necessary to specify the parameters it receives, between the `lambda` keyword and the colon (`:`). In the next example, only one parameter is needed, which will be the value of each element in the `Value` column. The value the function returns will be the square of that value.

In [14]:
```
s = edu["Value"].apply(lambda d: d**2)
s.head()
```

Out[14]:
```
0          NaN
1          NaN
2      25.0000
3      25.3009
4      24.5025
Name: Value, dtype: float64
```

Another basic manipulation operation is to set new values in our DataFrame. This can be done directly using the assign operator (=) over a DataFrame. For example, to add a new column to a DataFrame, we can assign a Series to a selection of a column that does not exist. This will produce a new column in the DataFrame after all the others. You must be aware that if a column with the same name already exists, the previous values will be overwritten. In the following example, we assign the Series that results from dividing the `Value` column by the maximum value in the same column to a new column named `ValueNorm`.

In [15]:
```
edu['ValueNorm'] = edu['Value']/edu['Value'].max()
edu.tail()
```

Out[15]:

	TIME	GEO	Value	ValueNorm
379	2007	Finland	5.90	0.669694
380	2008	Finland	6.10	0.692395
381	2009	Finland	6.81	0.772985
382	2010	Finland	6.85	0.777526
383	2011	Finland	6.76	0.767310

Now, if we want to remove this column from the DataFrame, we can use the `drop` function; this removes the indicated rows if `axis=0`, or the indicated columns if `axis=1`. In Pandas, all the functions that change the contents of a DataFrame, such as the drop function, will normally return a copy of the modified data, instead of overwriting the DataFrame. Therefore, the original DataFrame is kept. If you do not want to keep the old values, you can set the keyword `inplace` to `True`. By default, this keyword is set to `False`, meaning that a copy of the data is returned.

In [16]:
```
edu.drop('ValueNorm', axis = 1, inplace = True)
edu.head()
```

Out[16]:

	TIME	GEO	Value
0	2000	European Union (28 countries)	NaN
1	2001	European Union (28 countries)	NaN
2	2002	European Union (28 countries)	5
3	2003	European Union (28 countries)	5.03
4	2004	European Union (28 countries)	4.95

Instead, if what we want to do is to insert a new row at the bottom of the DataFrame, we can use the Pandas `append` function. This function receives as argument the new row, which is represented as a dictionary where the keys are the name of the columns and the values are the associated value. You must be aware to setting the `ignore_index` flag in the `append` method to `True`, otherwise the index 0 is given to this new row, which will produce an error if it already exists:

In [17]:

```
edu = edu.append({"TIME": 2000,"Value": 5.00,"GEO": 'a'},
                 ignore_index = True)
edu.tail()
```

Out[17]:

	TIME	GEO	Value
380	2008	Finland	6.1
381	2009	Finland	6.81
382	2010	Finland	6.85
383	2011	Finland	6.76
384	2000	a	5

Finally, if we want to remove this row, we need to use the `drop` function again. Now we have to set the axis to 0, and specify the index of the row we want to remove. Since we want to remove the last row, we can use the max function over the indexes to determine which row is.

In [18]:

```
edu.drop(max(edu.index), axis = 0, inplace = True)
edu.tail()
```

Out[18]:

	TIME	GEO	Value
379	2007	Finland	5.9
380	2008	Finland	6.1
381	2009	Finland	6.81
382	2010	Finland	6.85
383	2011	Finland	6.76

The `drop()` function is also used to remove missing values by applying it over the result of the `isnull()` function. This has a similar effect to filtering the `NaN` values, as we explained above, but here the difference is that a copy of the DataFrame without the `NaN` values is returned, instead of a view.

In [19]:

```
eduDrop = edu.drop(edu["Value"].isnull(), axis = 0)
eduDrop.head()
```

Out[19]:

	TIME	GEO	Value
2	2002	European Union (28 countries)	5.00
3	2003	European Union (28 countries)	5.03
4	2004	European Union (28 countries)	4.95
5	2005	European Union (28 countries)	4.92
6	2006	European Union (28 countries)	4.91

To remove NaN values, instead of the generic drop function, we can use the specific `dropna()` function. If we want to erase any row that contains an NaN value, we have to set the `how` keyword to `any`. To restrict it to a subset of columns, we can specify it using the `subset` keyword. As we can see below, the result will be the same as using the `drop` function:

In [20]:

```
eduDrop = edu.dropna(how = 'any', subset = ["Value"])
eduDrop.head()
```

Out[20]:

	TIME	GEO	Value
2	2002	European Union (28 countries)	5.00
3	2003	European Union (28 countries)	5.03
4	2004	European Union (28 countries)	4.95
5	2005	European Union (28 countries)	4.92
6	2006	European Union (28 countries)	4.91

If, instead of removing the rows containing NaN, we want to fill them with another value, then we can use the `fillna()` method, specifying which value has to be used. If we want to fill only some specific columns, we have to set as argument to the `fillna()` function a dictionary with the name of the columns as the key and which character to be used for filling as the value.

In [21]:

```
eduFilled = edu.fillna(value = {"Value": 0})
eduFilled.head()
```

Out[21]:

	TIME	GEO	Value
0	2000	European Union (28 countries)	0.00
1	2001	European Union (28 countries)	0.00
2	2002	European Union (28 countries)	5.00
3	2003	European Union (28 countries)	4.95
4	2004	European Union (28 countries)	4.95

2.6.6 Sorting

Another important functionality we will need when inspecting our data is to sort by columns. We can sort a DataFrame using any column, using the `sort` function. If we want to see the first five rows of data sorted in descending order (i.e., from the largest to the smallest values) and using the `Value` column, then we just need to do this:

In [22]:
```
edu.sort_values(by = 'Value', ascending = False,
                inplace = True)
edu.head()
```

Out[22]:

	TIME	GEO	Value
130	2010	Denmark	8.81
131	2011	Denmark	8.75
129	2009	Denmark	8.74
121	2001	Denmark	8.44
122	2002	Denmark	8.44

Note that the `inplace` keyword means that the DataFrame will be overwritten, and hence no new DataFrame is returned. If instead of `ascending = False` we use `ascending = True`, the values are sorted in ascending order (i.e., from the smallest to the largest values).

If we want to return to the original order, we can sort by an index using the `sort_index` function and specifying `axis=0`:

In [23]:
```
edu.sort_index(axis = 0, ascending = True, inplace = True)
edu.head()
```

Out[23]:

	TIME	GEO	Value
0	2000	European Union ...	NaN
1	2001	European Union ...	NaN
2	2002	European Union ...	5.00
3	2003	European Union ...	5.03
4	2004	European Union ...	4.95

2.6.7 Grouping Data

Another very useful way to inspect data is to group it according to some criteria. For instance, in our example it would be nice to group all the data by country, regardless of the year. Pandas has the `groupby` function that allows us to do exactly this. The value returned by this function is a special grouped DataFrame. To have a proper DataFrame as a result, it is necessary to apply an aggregation function. Thus, this function will be applied to all the values in the same group.

For example, in our case, if we want a DataFrame showing the mean of the values for each country over all the years, we can obtain it by grouping according to country and using the mean function as the aggregation method for each group. The result would be a DataFrame with countries as indexes and the mean values as the column:

In [24]:
```
group = edu[["GEO", "Value"]].groupby('GEO').mean()
group.head()
```

Out[24]:

	Value
GEO	
Austria	5.618333
Belgium	6.189091
Bulgaria	4.093333
Cyprus	7.023333
Czech Republic	4.16833

2.6.8 Rearranging Data

Up until now, our indexes have been just a numeration of rows without much meaning. We can transform the arrangement of our data, redistributing the indexes and columns for better manipulation of our data, which normally leads to better performance. We can rearrange our data using the `pivot_table` function. Here, we can specify which columns will be the new indexes, the new values, and the new columns.

For example, imagine that we want to transform our DataFrame to a spreadsheet-like structure with the country names as the index, while the columns will be the years starting from 2006 and the values will be the previous `Value` column. To do this, first we need to filter out the data and then pivot it in this way:

In [25]:

```
filtered_data = edu[edu["TIME"] > 2005]
pivedu = pd.pivot_table(filtered_data, values = 'Value',
                        index = ['GEO'],
                        columns = ['TIME'])
pivedu.head()
```

Out[25]:

TIME	2006	2007	2008	2009	2010	2011
GEO						
Austria	5.40	5.33	5.47	5.98	5.91	5.80
Belgium	5.98	6.00	6.43	6.57	6.58	6.55
Bulgaria	4.04	3.88	4.44	4.58	4.10	3.82
Cyprus	7.02	6.95	7.45	7.98	7.92	7.87
Czech Republic	4.42	4.05	3.92	4.36	4.25	4.51

Now we can use the new index to select specific rows by label, using the `ix` operator:

In [26]:

```
pivedu.ix[['Spain','Portugal'], [2006,2011]]
```

Out[26]:

TIME	2006	2011
GEO		
Spain	4.26	4.82
Portugal	5.07	5.27

Pivot also offers the option of providing an argument `aggr_function` that allows us to perform an aggregation function between the values if there is more

than one value for the given row and column after the transformation. As usual, you can design any custom function you want, just giving its name or using a λ-function.

2.6.9 Ranking Data

Another useful visualization feature is to rank data. For example, we would like to know how each country is ranked by year. To see this, we will use the pandas `rank` function. But first, we need to clean up our previous pivoted table a bit so that it only has real countries with real data. To do this, first we drop the Euro area entries and shorten the Germany name entry, using the `rename` function and then we drop all the rows containing any NaN, using the `dropna` function.

Now we can perform the ranking using the `rank` function. Note here that the parameter `ascending=False` makes the ranking go from the highest values to the lowest values. The Pandas rank function supports different tie-breaking methods, specified with the `method` parameter. In our case, we use the `first` method, in which ranks are assigned in the order they appear in the array, avoiding gaps between ranking.

In [27]:
```
pivedu = pivedu.drop([
                      'Euro area (13 countries)',
                      'Euro area (15 countries)',
                      'Euro area (17 countries)',
                      'Euro area (18 countries)',
                      'European Union (25 countries)',
                      'European Union (27 countries)',
                      'European Union (28 countries)'
                     ],
                     axis = 0)
pivedu = pivedu.rename(index = {'Germany (until 1990 former territory
    of the FRG)': 'Germany'})
pivedu = pivedu.dropna()
pivedu.rank(ascending = False, method = 'first').head()
```

Out[27]:

TIME	2006	2007	2008	2009	2010	2011
GEO						
Austria	10	7	11	7	8	8
Belgium	5	4	3	4	5	5
Bulgaria	21	21	20	20	22	21
Cyprus	2	2	2	2	2	3
Czech Republic	19	20	21	21	20	18

If we want to make a global ranking taking into account all the years, we can sum up all the columns and rank the result. Then we can sort the resulting values to retrieve the top five countries for the last 6 years, in this way:

In [28]:
```
totalSum = pivedu.sum(axis = 1)
totalSum.rank(ascending = False, method = 'dense')
        .sort_values().head()
```

Out[28]:
```
GEO
Denmark                1
Cyprus                 2
Finland                3
Malta                  4
Belgium                5
dtype: float64
```

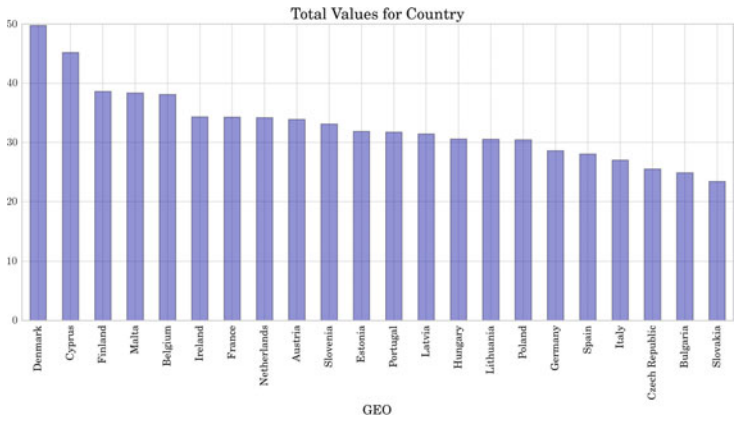
Notice that the `method` keyword argument in the in the `rank` function specifies how items that compare equals receive ranking. In the case of `dense`, items that compare equals receive the same ranking number, and the next not equal item receives the immediately following ranking number.

2.6.10 Plotting

Pandas DataFrames and Series can be plotted using the `plot` function, which uses the library for graphics Matplotlib. For example, if we want to plot the accumulated values for each country over the last 6 years, we can take the Series obtained in the previous example and plot it directly by calling the `plot` function as shown in the next cell:

In [29]:
```
totalSum = pivedu.sum(axis = 1)
                 .sort_values(ascending = False)
totalSum.plot(kind = 'bar', style = 'b', alpha = 0.4,
              title = "Total Values for Country")
```

Out[29]:



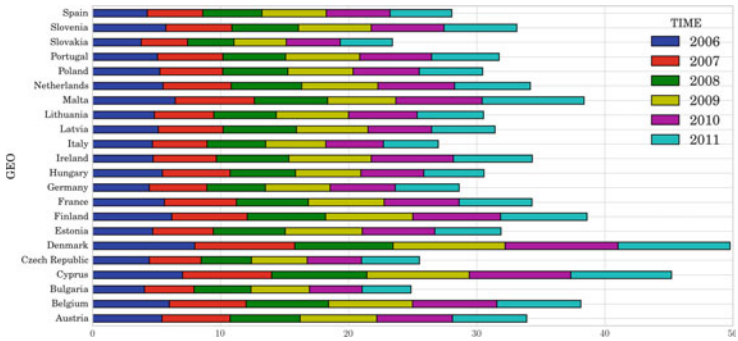
Note that if we want the bars ordered from the highest to the lowest value, we need to sort the values in the Series first. The parameter `kind` used in the `plot` function defines which kind of graphic will be used. In our case, a bar graph. The parameter `style` refers to the style properties of the graphic, in our case, the color

of bars is set to `b` (blue). The alpha channel can be modified adding a keyword parameter `alpha` with a percentage, producing a more translucent plot. Finally, using the `title` keyword the name of the graphic can be set.

It is also possible to plot a DataFrame directly. In this case, each column is treated as a separated Series. For example, instead of printing the accumulated value over the years, we can plot the value for each year.

In [30]:
```
my_colors = ['b', 'r', 'g', 'y', 'm', 'c']
ax = pivedu.plot(kind = 'barh',
                 stacked = True,
                 color = my_colors)
ax.legend(loc = 'center left', bbox_to_anchor = (1, .5))
```

Out[30]:



In this case, we have used a horizontal bar graph (`kind='barh'`) stacking all the years in the same country bar. This can be done by setting the parameter `stacked` to `True`. The number of default colors in a plot is only 5, thus if you have more than 5 Series to show, you need to specify more colors or otherwise the same set of colors will be used again. We can set a new set of colors using the keyword `color` with a list of colors. Basic colors have a single-character code assigned to each, for example, "b" is for blue, "r" for red, "g" for green, "y" for yellow, "m" for magenta, and "c" for cyan. When several Series are shown in a plot, a legend is created for identifying each one. The name for each Series is the name of the column in the DataFrame. By default, the legend goes inside the plot area. If we want to change this, we can use the `legend` function of the axis object (this is the object returned when the plot function is called). By using the `loc` keyword, we can set the relative position of the legend with respect to the plot. It can be a combination of right or left and upper, lower, or center. With `bbox_to_anchor` we can set an absolute position with respect to the plot, allowing us to put the legend outside the graph.

2.7 Conclusions

This chapter has been a brief introduction to the most essential elements of a programming environment for data scientists. The tutorial followed in this chapter is just a starting point for more advanced projects and techniques. As we will see in the following chapters, Python and its ecosystem is a very empowering choice for developing data science projects.

Acknowledgements This chapter was co-written by Eloi Puertas and Francesc Dantí.

http://www.springer.com/978-3-319-50016-4

Introduction to Data Science
A Python Approach to Concepts, Techniques and Applications
Igual, L.; Seguí, S.
2017, XIV, 218 p. 73 illus., 67 illus. in color., Softcover
ISBN: 978-3-319-50016-4