TRAINING & REFERENCE

# murach's Python for Data Analysis

(Chapter 1)

MIKE MURACH & ASSOCIATES, INC.
1-800-221-5528 • (559) 440-9071 • Fax: (559) 440-0963
**murachbooks@murach.com** • **www.murach.com**

# Section 1

## Get off to a fast start

This section will get you off to a fast start with Python for data analysis. In chapter 1, you will be introduced to Python for data analysis, and you'll learn how to use JupyterLab as your IDE. In chapter 2, you'll learn the Pandas essentials for data analysis. In chapter 3, you'll learn the Pandas essentials for data visualization. And in chapter 4, you'll learn the Seaborn essentials for enhanced data visualization.

When you complete those chapters, you'll have a subset of the skills that you need for doing analyses of your own. You'll also be able to skip to any chapter in section 2 whenever you need to learn more about a specific phase of data analysis.

# Chapter 1

# Introduction to Python for data analysis

This chapter starts by introducing you to data analysis with Python and by reviewing the Python coding skills that you'll need for data analysis. Then, it shows you how to use JupyterLab as your IDE. Last, this chapter introduces you to the case studies for this book because they are a critical part of the learning process.

When you finish this chapter, you'll have the background that you need for learning the essential skills for data analysis and visualization. And that's what you'll learn in chapters 2, 3, and 4 of this section.

# Introduction to data analysis

Before you start learning how to analyze and visualize data, you should have some perspective on what data analysis is and isn't. So that's where this chapter begins.

## What data analysis is

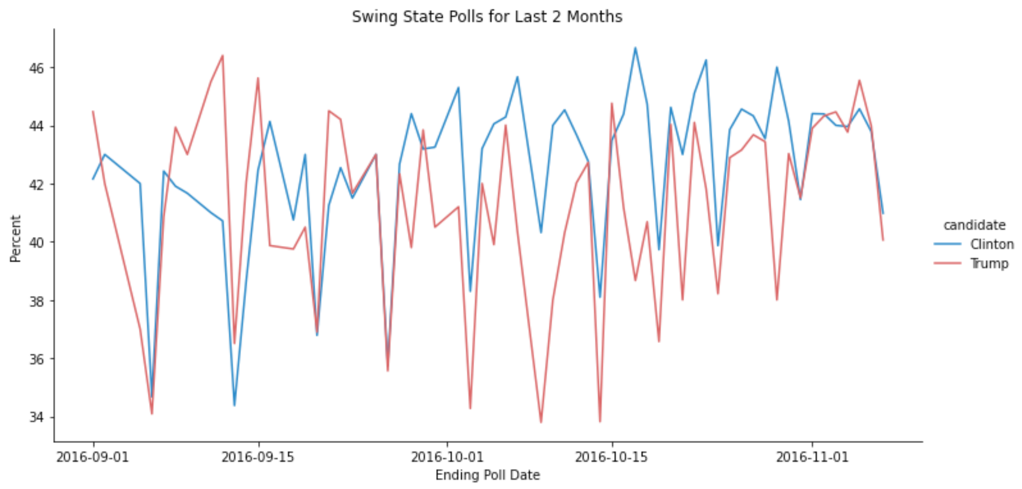
Figure 1-1 summarizes the components of *data analysis*. One of the key points here is that data analysis not only includes *data visualization* (or *data viz*), but also that data visualization often provides the best insights into the data. The goal of this book is to teach you a professional set of skills for data analysis and data viz.

*Data modeling*, or *predictive analysis*, is included within the context of data analysis. It involves the use of data for building models that can help predict what's going to happen in the future based on the data of the past.

*Data analytics* is often used as a synonym for data analysis and visualization. Similarly, *business analytics* refers to data analysis with the focus on business data, and *sports analytics* refers to data analysis with the focus on sports data. Of course, business and sports analytics are just applications of the skills that you'll learn in this book.

*Data science* is a field that starts with data analysis but also includes advanced skills like *data mining*, *machine learning*, *deep learning*, and *artificial intelligence* (*AI*). Although these advanced skills aren't included in this book, they all require a solid set of the essential skills for data analysis and data visualization. So that's where you need to start, and those are the skills that you'll learn from this book.

## Data visualization often provides the best insights into the data



## What data analysis includes

- Data analysis
- Data visualization (data viz)
- Data modeling (predictive analysis)

## Related terms

- Data analytics
- Business analytics
- Sports analytics

## Description

- In this book, you'll learn how to use Python for *data analysis* and *data visualization* (or *data viz*), and you'll be introduced to the skills for *predictive analysis*.
- *Data analytics* is often used as a synonym for data analysis. Similarly, *business analytics* refers to the analysis of business data, and *sports analytics* refers to the analysis of sports data.
- After you use this book to master data analysis and visualization, you'll be ready to learn advanced analytical skills like *data mining, machine learning, deep learning*, and *artificial intelligence* (*AI*).
- *Data science* is a term that includes both data analysis and advanced skills like data mining, machine learning, deep learning, and AI.

Figure 1-1 What data analysis is

## The five phases of data analysis and visualization

To give you some idea of what you're getting into, figure 1-2 presents the five phases of a data analysis and visualization project. Note, however, that you don't just start into the five phases of analysis without any planning. Instead, you need to *set the goals* for your analysis project and *define the target audience*. Then, when you have a clear view of what you're going to do, you start the five phases of the project.

In phase 1, you *get the data* for the project. That can be from a third-party website or from one of your own company's databases or spreadsheets. In this phase, you read (or *import*) the data into a *DataFrame*, which is a data structure that consists of columns and rows. You'll learn more about DataFrames in the next chapter.

In phase 2, you *clean the data*. That includes removing unnecessary rows and columns, fixing invalid or missing data, and changing data types. As you will discover, most real-world data is surprisingly "dirty" so this is often a time-consuming phase of analysis. But if you don't clean the data, it will affect the accuracy of your analysis.

In phase 3, you *prepare the data*. That includes adding new columns that are calculated or derived from the original data and shaping the data into the forms that are needed for the analysis. It may also include doing some early visualizations that will help you understand the data.

In phase 4, you *analyze the data*. This includes getting new views of the data by grouping and aggregating the data. It includes doing data visualizations because they often provide insights and show relationships that you can't get from tabular data. And it may include predictive analysis that tries to predict future results based on past results.

In phase 5, you *visualize the data* in a way that's appropriate for your target audience. That means you enhance your visuals so they get their points across as clearly and quickly as possible…even to those with no technical or analytical background.

Of course, the divisions between these phases aren't nearly as clear as this figure might make them seem. In fact, there is usually some overlap between the phases. For instance, when you clean or prepare the data, you're already looking ahead to the analyze phase. And when you analyze the data, you may discover that you need to do more cleaning or preparation.

Nevertheless, these phases are a good guide to the work that you'll do for most analyses, and they provide a useful way to divide the chapters in this book. But note that data visualization is presented in chapters 3 and 4 because it is so critical to effective data analysis. Then, chapters 5 through 9 in section 2 show how to do each of the first four phases: get, clean, prepare, and analyze the data.

## What you need to do before you start an analysis

### Set your goals

- The *goals of analysis* can be well-defined, like trying to answer specific questions, or more general, like trying to extract useful information from large volumes of data.

### Define your target audience

- If you're going to present your findings to other people like managers or clients, you also need to define your *target audience* before you start your analysis.

## The five phases of data analysis and visualization

### Get the data

- Find the data on a website or in one of your company's databases or spreadsheets.
- Read the data into a DataFrame or build a DataFrame from the data.

### Clean the data

- Remove unnecessary rows and columns.
- Handle invalid or missing values.
- Change object data types to datetime or numeric data types.

### Prepare the data

- Add columns that are derived from other columns.
- Shape the data into the forms that are needed for your analysis.
- Make preliminary visualizations to better understand the data.

### Analyze the data

- Get new views of the data by grouping and aggregating the data.
- Make visualizations that provide insights and show relationships.
- Model the data as part of predictive analysis.

### Visualize the data

- Enhance your visualizations so they're appropriate for your target audience.

## Description

- Before you start any analysis, you need to *set your goals* and *define the target audience*.
- You can divide a typical data analysis project into five phases like those above. In practice, though, there's usually some overlap between the phases.

Figure 1-2 The five phases of data analysis and visualization

## The IDEs for Python data analysis

In the appendixes for this book, you can learn how to install the *Anaconda distribution* of Python for both Windows and macOS. That distribution includes all the major components that you'll need for data analysis and visualization with Python, including the first two *Integrated Development Environments* (*IDEs*) in the table in figure 1-3. That's why we recommend that you install and use that distribution with this book.

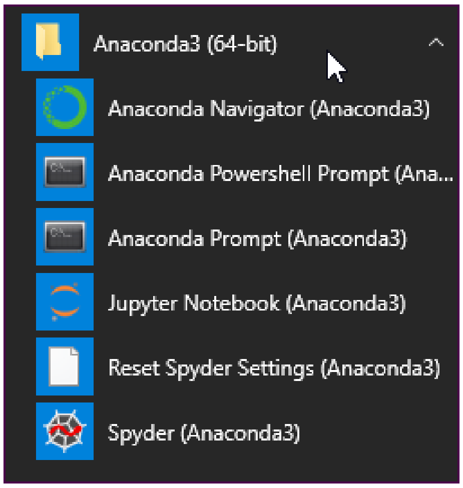
We also recommend that you use *JupyterLab* as your IDE, which is an enhanced version of *Jupyter Notebook*. As you'll see later in this chapter, you start JupyterLab from the Anaconda Navigator, which is why JupyterLab isn't included in the menu shown in this figure.

JupyterLab lets you organize your analyses in *Notebooks* with one Notebook for each analysis. Within each Notebook, you write small blocks of code in cells that you can execute, one at a time. You can also include text within the Notebooks to document what the code in the cells is doing. As you'll see in a moment, these features make JupyterLab an excellent IDE, and that's especially true when you're learning.

## Three of the IDEs you can use for Python data analysis

| IDE | Description |
|---|---|
| Jupyter Notebook | A web-based IDE that organizes each project in a Notebook that can include text that describes the operations |
| JupyterLab | An enhanced IDE for using Jupyter Notebooks |
| VS Code | A Microsoft IDE with support for operations like debugging and version control |

## The programs that are installed by the Anaconda distribution


Anaconda3 (64-bit)
Anaconda Navigator (Anaconda3)
Anaconda Powershell Prompt (Ana...
Anaconda Prompt (Anaconda3)
Jupyter Notebook (Anaconda3)
Reset Spyder Settings (Anaconda3)
Spyder (Anaconda3)

## Our recommendations for Python distributions and IDEs

- Use the Anaconda distribution of Python.
- Use JupyterLab as your IDE.

## Description

- *Jupyter Notebook* is an *Integrated Development Environment* (*IDE*) that helps you keep your code organized by dividing it into cells within *Notebooks*.
- *JupyterLab* is an enhanced version of Jupyter Notebook that provides features like split-screen editing of two different Notebooks.

Figure 1-3 The IDEs for Python data analysis

# The Python skills that you need for data analysis

This book assumes that you have some programming experience with Python. So what follows is a quick refresher on the main Python skills that you'll need when you use Python for data analysis and visualization.

Since all of this is typical Python code, you shouldn't have any trouble with it. But if you do have trouble, by all means get the companion book to this one, *Murach's Python Programming*. It presents all of the Python skills that you will need, so this book can focus on the skills for data analysis and visualization.

## How to install and import the Python modules for data analysis

When you use Python for data analysis, you use *modules* like the ones in the two tables in figure 1-4 for various aspects of your work. For instance, you'll use the Pandas module for much of the data analysis that you do. You'll use a module like Seaborn for data visualization.

Most of the modules that you will need for this book are included in the Anaconda distribution, so you won't need to install them. These modules are listed in the first table in this figure. However, you will need to install the modules that are listed in the second table.

To install a module, you can run a *conda command* from the *Anaconda Prompt*. This is illustrated by the first group of examples in this figure. Usually, the format for the first command will work. But sometimes, you will need to use the conda-forge channel as shown in the second command. This is because each channel holds different packages and not all packages are available from the default channel. For more information on how to display the Anaconda prompt and use the conda command to install the modules you need for this book, please see the appendixes.

After you install a module, you need to *import* it into your Notebook before you can use it. To do that, you use the *import statement* as shown in the second group of examples. Within those statements, it's best to use the standard abbreviations that are listed in the first table. For instance, pd is used to refer to the Pandas module, np is used for NumPy, and sns is used for Seaborn. You can also use the from clause in an import statement to import just one submodule or method from a module. In this figure, for example, the second import statement imports the request submodule of the urllib module.

### Modules that are included with the Anaconda distribution

| Module | Abbreviation | Provides methods for |
|---|---|---|
| `pandas` | `pd` | Data analysis and visualization |
| `numpy` | `np` | Numerical computing |
| `seaborn` | `sns` | Data visualization |
| `datetime` | `dt` | Working with datetime objects |
| `urllib` | | Getting files from the web |
| `zipfile` | | Working with zip files |
| `sqlite3` | | Working with a SQLite database |
| `json` | | Working with JSON data |
| `sklearn` | | Regression analysis |

### The modules that you need to install for this book

| Module | Chapter | Provides methods for |
|---|---|---|
| `pyreadstat` | 5 | Reading Stata files |
| `geopandas` | 12 | Plotting geographic data |

### Two ways to install a module

#### Use the conda command from the Anaconda Prompt

```
conda install pandas --yes
```

#### Use the conda command with a different channel

```
conda install --channel conda-forge pyreadstat --yes
```

### How to import modules

#### How to import one module into the namespace specified by the as clause

```
import pandas as pd
```

#### How to import one submodule from a module

```
from urllib import request
```

### Description

- Most of the *modules* that you need for this book are installed as part of the Anaconda distribution. But you still need to import them before you can use them.
- To *install* a module that isn't included in the Anaconda distribution, you can run a *!pip command* from a Notebook cell or a *conda command* from the Anaconda prompt.
- To *import* a module, you use the Python *import statement*. This statement lets you import an entire module or just the submodules or methods that you're going to use.
- See the appendixes for more on how to install the modules that you'll need for this book that aren't included in the Anaconda distribution.

Figure 1-4 How to install and import the Python modules for data analysis

## How to call and chain methods

If you've used Python to develop applications, you already know how to call Python methods. But since this skill is so essential to using the modules for data analysis, figure 1-5 provides a quick review. In addition, it shows how to chain methods, which is another essential skill for data analysis.

After you import a module, you can *call* any of its methods, as shown by the first group of examples. Here, the first example shows how to call a method in a Pandas module. To do that, you code the abbreviation for the module (pd), a dot (period), the method name, and any *parameters* in parentheses. In this example, the read_csv() method is executed and its one parameter is the URL for a file on the FiveThirtyEight website. When this method is run, it reads the file into a Pandas DataFrame object named polls.

The second example in this group shows how to call a method from a Pandas object. To do that, you code the object name followed by a dot, the method name, and any parameters in parentheses. So in this case, the sort_values() method of the Pandas DataFrame object named polls is executed, and its one parameter specifies the column that the data should be sorted by.

The second group of examples in this figure shows how to use *dot notation* to *chain* methods. In the first statement, the head() method is chained to the sort_values() method. So after the data in the DataFrame named polls is sorted, the head() method displays the first five rows of the sorted data. This works because the sort_values() method returns a DataFrame object with the sorted data. Then, the head() method can be run on that object.

In the second statement in this group, the plot() method is chained to the query() method. So after the data is selected by the query() method, the plot() method plots the data for each poll.

When you code the parameters for a method, you need to recognize the difference between positional and keyword parameters. This is illustrated by the third group of examples in this figure. This starts with the *signature* for the sort_values() method. A signature is the part of the syntax for a method that lists its parameters.

Within a signature, the *positional parameters* are first and are identified by their position in the signature. In this case, the sort_values() method has one positional parameter named *by*. By contrast, each *keyword parameter* is followed by an equal sign and the default value for the parameter, if it has one. For instance, the sort_values() method has a keyword parameter named ascending that has a default value of True.

When you code the parameters for a method, you need to code the values for the positional parameters first and in the same sequence that the parameters are listed in the signature. This is illustrated by the statement after the signature. Here, the value of the positional parameter is the startdate column, and it's followed by the ascending and inplace keyword parameters. Because the other two keyword parameters aren't coded, their default values are used. As a result, the rows in the DataFrame will be sorted by the data in the startdate column in descending sequence, and the sorted result will replace the data in the polls DataFrame.

## How to call methods

### How to call a method in a module

```
import pandas as pd
polls_url = 'http://projects.fivethirtyeight.com/.../president_general_polls_2016.csv'
polls = pd.read_csv(poll_url)
```

### How to call a method from a DataFrame object

```
polls.sort_values('startdate')
```

## How to chain methods

### How to chain the sort_values() and head() methods

```
polls.sort_values('startdate').head()
```

### How to chain the query() and plot() methods

```
polls.query('state != "U.S."') \
    .plot(x='startdate', y=['Clinton_pct','Trump_pct'])
```

## How to call a method with positional and keyword parameters

### The signature for the sort_values() method

```
sort_values(by, axis=0, ascending=True, inplace=False,
                kind='quicksort', na_position='last')
```

### The sort_values() method with positional and keyword parameters

```
polls.sort_values('startdate', ascending=False, inplace=True)
```

## Description

- To *call* a *method* in a module, you code the module abbreviation, a dot, the method name, and the *parameters* (*arguments*) within parentheses. To call a method from an object, you code the object name, a dot, the method name, and the parameters within parentheses.
- To *chain methods*, you use *dot notation*.
- The *signature* of a method shows the parameters of the method. The parameters that have equal signs after them are *keyword parameters*; the others are *positional parameters*. The values after the equal signs in the keyword parameters are the default values.
- When methods accept both positional and keyword parameters, you need to code the positional parameters before the keyword parameters and in the sequence shown in the signature.

Figure 1-5 How to call and chain methods

## The coding basics for Python data analysis

As you learn to use the modules and methods for data analysis and visualization, you'll see that you need to code some parameters as *lists*, some as *tuples*, some as *dictionary objects*, and some as *slices*. That's why figure 1-6 provides a quick review of the coding for these structures.

As you can see in the first group of examples, a *list* is a sequence of items that is coded within brackets. A *tuple* is a sequence of items coded within parentheses. A *dictionary object* (or *dict*) is a sequence of key/value pairs that are connected by colons and coded within braces. And a *slice* is coded as a start value, a stop value, and an optional step value with the values separated by colons.

In the second group of examples, you can see how these structures are used in Python statements for data analysis. In the first statement, a list of four values is coded for the columns parameter of a drop() method. In the second statement, a tuple is coded for the xlim parameter of a line() method. In the third statement, a dictionary is coded for the columns parameter of a rename() method.

In the fourth statement in this group of examples, two slices are coded in a loc[ ] accessor. Here, the first slice accesses every tenth row from 0 to 100, and the second slice accesses every column from the state column to the grade column.

The third group of examples shows how you can use a *list comprehension* to create a list. This is a shorthand way to create a for loop that populates a new list. To start, you code a set of brackets. Then, within the brackets, you code an expression, the word *for*, the name that will be used for each list member, and a function that returns the members for the list or a list that has already been defined.

In this case, the expression is x which represents one member. And the function is the range() function, which returns one member for each of the values specified by the start, stop, and step parameters. Note that when you use the range() function, the stop value isn't included in the generated list. As a result, this list comprehension returns the even numbers from 1900 through 1918.

In the next chapter, you'll start learning how to use the Pandas methods that require structures like lists, tuples, dictionaries, and slices. So for now, you just need to know how to code these structures.

But if you're new to Python, you also need to know the rules for continuing statements over more than one line. This is illustrated by the last group of examples in this figure. With *implicit continuation*, you divide a statement after separators like parentheses, brackets, braces, and commas and after operators like plus and minus signs. With *explicit continuation*, you code a backward slash to divide a statement anywhere and continue it on the next line.

When you use implicit continuation, you need to realize that dividing a statement at the wrong point or with the wrong amount of indentation will raise a syntax error. Then, the easiest way to fix that is to switch to explicit continuation. In this book, most of the examples are coded so they don't require explicit continuation.

## The syntax for coding lists, slices, tuples, and dictionary objects

### A list is a sequence of items within brackets

```
[item1,item2,...]
```

### A tuple is coded like a list but in parentheses

```
(item1,item2,...)
```

### A dictionary is a sequence of key/value pairs within braces

```
{key1:value1, key2:value2,...}
```

### A slice sets the start and stop values and an optional step value

```
start:stop:step
```

## How to use lists, slices, tuples, and dictionary objects

### A list used as a keyword parameter

```
polls.drop(columns=['cycle','branch','matchup','forecastdate'], inplace=True)
```

### A tuple used as a keyword parameter

```
polls.plot.line(xlim=('2016-06','2016-11'))
```

### A dictionary used as a keyword parameter

```
polls.rename(columns={'adjpoll_clinton':'Clinton',
                      'adjpoll_trump':'Trump'})
```

### Two slices used in a loc[] accessor

```
polls.loc[0:100:10,'state':'grade']
```

## How to code a list comprehension

### The syntax

```
[expression for member in iterable]
```

### A list comprehension used to provide the list for a keyword parameter

```
xticks = [x for x in range(1900,1920,2)]
```

### The resulting list

```
[1900, 1902, 1904, 1906, 1908, 1910, 1912, 1914, 1916, 1918]
```

## Two ways to continue a statement over more than one line

### With implicit continuation

```
polls.sort_values(
    ['state','startdate'],
    ascending=False,
    inplace=True)
```

### With explicit continuation

```
polls.sort_values(['state','startdate'], \
                  ascending=False, \
                  inplace=True)
```

Figure 1-6 The coding basics for Python data analysis

## How to use JupyterLab as your IDE

Remember that JupyterLab is the IDE that we recommend for doing your analysis and visualization. If you've used other IDEs for developing applications, you shouldn't have much trouble learning how to use JupyterLab. But the topics that follow will help you get started.

### How to start JupyterLab and work with a Notebook

The first procedure in figure 1-7 shows how to start JupyterLab. To do that, you start Anaconda Navigator from the Start menu shown in figure 1-3. Then, you can click on the Launch button for JupyterLab in the Navigator. That starts a web server on your own computer and opens a browser tab for the JupyterLab IDE.

Within JupyterLab, you can use the other procedures in this figure to work with Notebooks. To start, you can open the File Browser by clicking on the File Browser icon in the upper left corner of the interface. This Browser makes it easy to find the Notebooks that you're looking for. So, to open a Notebook, you just browse to its file and double-click on it. That will display the Notebook in one tab within the Notebook panel.

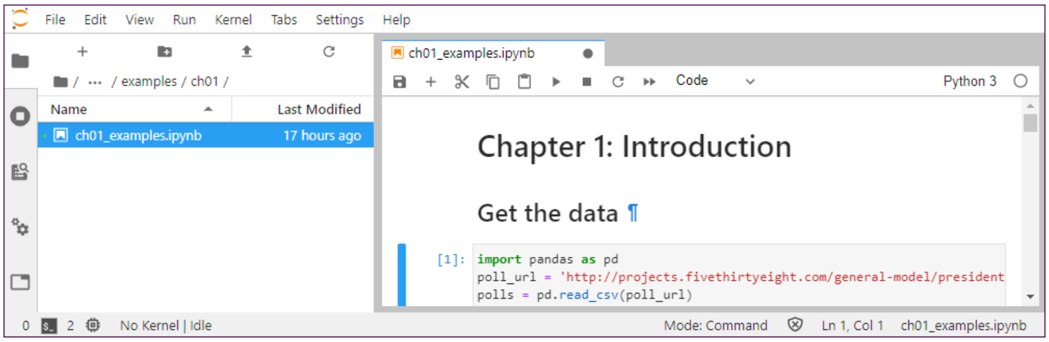
Note that the first time you start JupyterLab, the File Browser will display the high-level folders that are available by default from your computer's file system. Then, if you've installed the Notebooks for this book in the Documents folder as described in the appendixes, you can open that folder and then the python_analysis subfolder to see the folders with the examples, exercises, and solutions.

To start a new Notebook, you select File→New Launcher to open a new tab in JupyterLab. Then, you click on the Python 3 icon. To perform other file operations, you can use the items in the File menu or the popup menu that's displayed when you right-click on a file in the Browser.

As the screenshots in this figure show, each open Notebook is displayed in a separate tab of JupyterLab. Then, to switch from one Notebook to another, you can click on its tab. You can also save the current version of a Notebook by clicking on the Save icon that's on the left side of the toolbar for the tab.

By default, JupyterLab also provides for *checkpoints*. That means that JupyterLab periodically saves a copy of each Notebook. Then, if something goes wrong and you want to restart from an earlier checkpoint, you can use File→Revert Notebook to Checkpoint to restore the DataFrame at that point.

### One Notebook in JupyterLab with the File Browser open

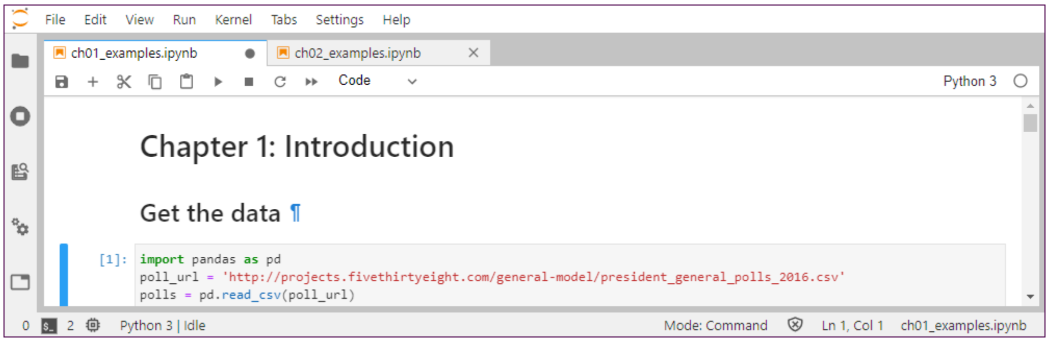


Chapter 1: Introduction

Get the data ¶

```
[1]: import pandas as pd
     poll_url = 'http://projects.fivethirtyeight.com/general-model/president
     polls = pd.read_csv(poll_url)
```

### Two Notebooks in JupyterLab with the File Browser closed



Chapter 1: Introduction

Get the data ¶

```
[1]: import pandas as pd
     poll_url = 'http://projects.fivethirtyeight.com/general-model/president_general_polls_2016.csv'
     polls = pd.read_csv(poll_url)
```

### How to start JupyterLab

- Use the Start menu to start the Anaconda Navigator, and then launch JupyterLab.

### How to work with Notebooks

- To open or close the File Browser, click the File Browser icon in the upper left corner.
- To open a Notebook, browse to the file you want to open and double-click on it.
- To start a new Notebook, select File→New Launcher to open a Launcher tab. Then, click the Python 3 icon.
- To save, close, or rename a Notebook, use the File menu. To save the active Notebook, click on the Save icon in the toolbar for the tab.
- To restore a Notebook to the last checkpoint, select File→Revert Notebook to Checkpoint.

### Description

- JupyterLab runs as a web application on a web server that's installed on your computer.
- By default, JuptyerLab periodically saves each Notebook as a *checkpoint*. If necessary, then, you can use the File menu to restore a Notebook at a previous checkpoint.

Figure 1-7 How to start JupyterLab and work with a Notebook

## How to edit and run the cells in a Notebook

When you use JupyterLab, each analysis is stored in a Notebook. In addition, each Notebook consists of *cells* that contain either one or more lines of code or the text for a heading that can be used to describe what the code in the cell or cells that follow are doing.

In figure 1-8, for example, the first cell contains text for the title of the Notebook, "Chapter 1: Introduction"; the second cell contains the text "Get the data"; and the third cell contains four lines of Python code. Then, when the code in that cell is run, the result is shown below the cell. This is an effective way to manage the code in an analysis because you can enter just a few lines of code in a cell, run the cell, and see the results right away. That's one reason why the Notebook approach has become so popular.

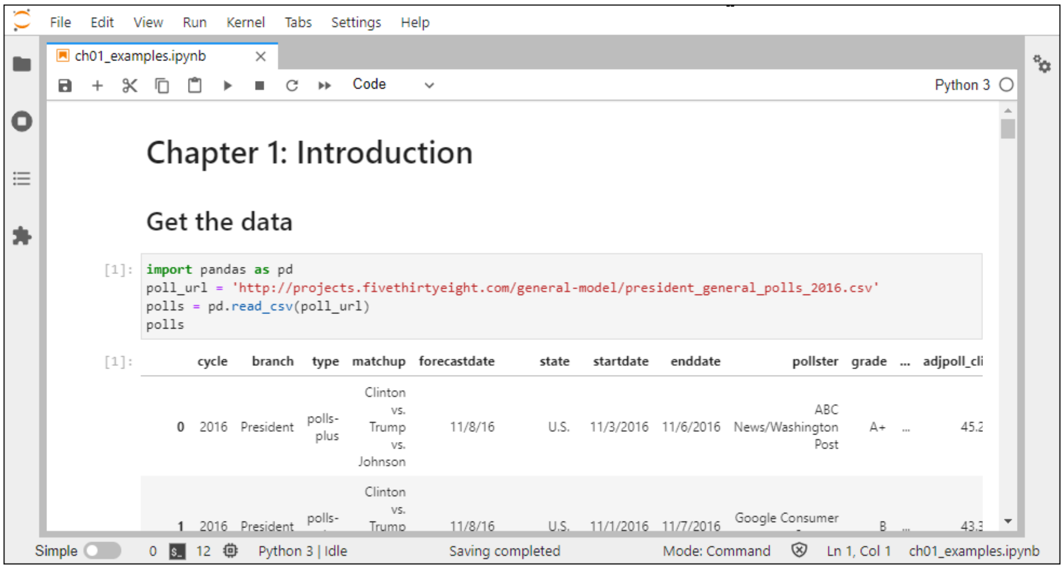
The procedures in this figure present the skills that you need to edit and run cells. If you want to do an operation on more than one cell, the first procedure shows how to select the cells. The key to this is to position the pointer to the left of a cell until it turns into a crosshair and then click. Then, the vertical blue line appears to show that the cell is selected.

Note that if you click when the cursor isn't a crosshair, the cell will still be selected but it will be collapsed so you can't see the code it contains. Then, you can click on the blue line to expand the cell. Once you've selected the cells, you can use the toolbar buttons or the items in the menus or popup menus to get the results you want.

When a cell is run, it's the Python interpreter, or *kernel*, that runs its statements. Then, to interrupt, restart, or shut down the kernel, you can use the items in the Kernel menu. For instance, one item lets you restart the kernel and clear all outputs. Another lets you restart the kernel and run all cells.

If you like to use the keyboard instead of the mouse to do operations, be sure to try Shift+Enter for running cells. This works when you want to run a cell right after you've entered its code. It also works when you want to step through the cells of a Notebook by running one cell at a time. Just be sure that the cursor is in a cell when you press Shift+Enter the first time.

## A cell and its output when the cell is run



Chapter 1: Introduction

Get the data

```
[1]: import pandas as pd
     poll_url = 'http://projects.fivethirtyeight.com/general-model/president_general_polls_2016.csv'
     polls = pd.read_csv(poll_url)
     polls
```

```
[1]:   cycle  branch     type    matchup   forecastdate  state  startdate  enddate    pollster          grade  ...  adjpoll_cli
                                 Clinton
                                   vs.
                         polls-  Trump                                                ABC
       0  2016  President  plus    vs.      11/8/16      U.S.   11/3/2016  11/6/2016  News/Washington   A+    ...  45.2
                                 Johnson                                              Post
                                 Clinton
                                   vs.
       1  2016  President polls- Trump      11/8/16      U.S.   11/1/2016  11/7/2016  Google Consumer   B     ...  43.3
```

Simple 0 12 Python 3 | Idle Saving completed Mode: Command Ln 1, Col 1 ch01_examples.ipynb

## How to select one or more cells

- To select one cell, position the pointer in the left margin of the cell so it becomes a crosshair, and then click so a blue line is displayed.
- To select more than one cell, select the first cell, hold down the Shift key, and select the last cell.

## How to copy, delete, merge, or move the selected cells

- Use the buttons in the toolbar or the items in the Edit or shortcut menu.

## How to add a cell after the current cell

- Use the + button in the toolbar.

## How to run the code in one cell

- Press Shift+Enter or click the Run button in the toolbar.

## How to run the code in selected cells or all cells

- Use the Run button in the toolbar or the items in the Run menu.

## How to interrupt, restart, or shutdown the kernel

- Use the items in the Kernel menu.

## Description

- When you run a cell, any output is displayed below the cell.
- The *kernel* is the Python interpreter that runs the code in the cells. You can use the Kernel menu to restart the kernel and clear all output or to restart and run all cells.

Figure 1-8 How to edit and run the cells in a Notebook

## How to use the Tab completion and tooltip features

As you enter or edit the code in a cell, JupyterLab provides two features that can help you work more productively. Both are illustrated in figure 1-9.

To activate the *Tab completion feature*, you press the Tab key after you enter an object name or an object name and a dot. Then, JupyterLab displays a drop-down list of all the attributes and methods that apply to the object. At that point, you can scroll down the list to find what you're looking for, or you can enter the first letter or two of the attribute or method that you're looking for to refine the entries in the list.

In the first example in this figure, you can see the two methods that start with the letters "so." They are the sort_index() and sort_values() methods. Then, you can click on the entry that you want. Or, you can scroll down to it and press the Enter key.

Be aware, however, that it may take several seconds for the drop-down list to appear if the method has many attributes and methods. So in some cases, you will want to enter the first letters of the attribute or method that you're looking for before you press the Tab key. That will speed up the display of the drop-down list.

To activate the *tooltip feature*, you press the Shift+Tab key after you enter an attribute or method name. This is illustrated by the second example in this figure. As you can see, this feature lets you scroll through the documentation for each attribute or method.

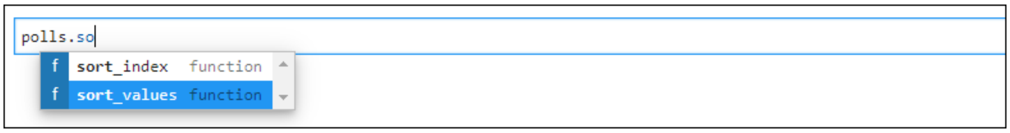
For each method, the documentation starts with the *signature*. As you have seen, the signature provides a list of the parameters for the method. This list starts with the positional parameters, which are indicated by keywords that aren't followed by equal signs. These positional parameters are followed by the keyword parameters, which are followed by equal signs and the default values.

After the signature, you can scroll down to get more information about each parameter in the signature. And after that, you can scroll down for even more information, like examples of how the method works and a summary of related methods. In short, this feature often provides all the information that you need so you don't have to go to other forms of documentation. That's especially true after you get used to working with the Pandas methods and accessors.

One shortcoming of the tooltips feature is that a tooltip disappears as soon as you type a character into the cell. However, you can get the tooltip back by pressing the Shift+Tab key again.

A more serious shortcoming is that you can't get tooltips for methods that are chained to other methods. You can only get a tooltip for the first method in a chain. In that case, though, you can get the information that you need by searching the Internet for the method.

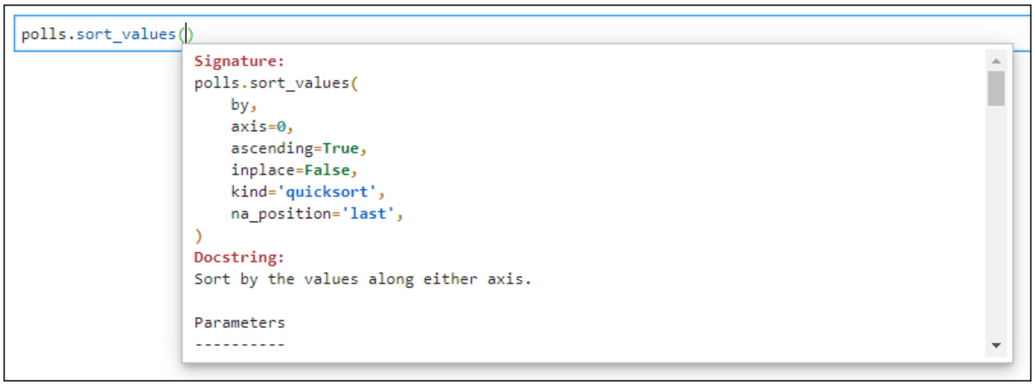
## The Tab completion feature is activated when you press the Tab key



```
polls.so
f sort_index   function
f sort_values  function
```

## The tooltip feature is activated when you press the Shift+Tab key

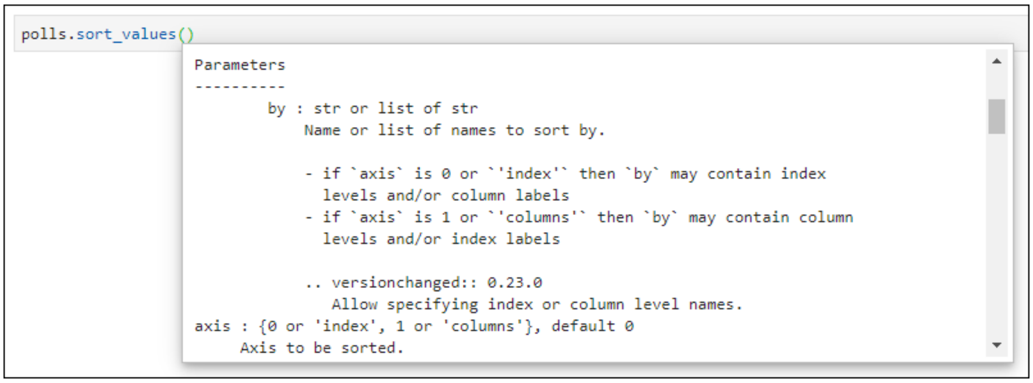
### The start of the tooltip for the sort_values() method



```
polls.sort_values()
Signature:
polls.sort_values(
    by,
    axis=0,
    ascending=True,
    inplace=False,
    kind='quicksort',
    na_position='last',
)
Docstring:
Sort by the values along either axis.

Parameters
----------
```

### More of the tooltip after scrolling down to the start of the parameters



```
polls.sort_values()
Parameters
----------
        by : str or list of str
            Name or list of names to sort by.

            - if `axis` is 0 or `'index'` then `by` may contain index
              levels and/or column labels
            - if `axis` is 1 or `'columns'` then `by` may contain column
              levels and/or index labels

            .. versionchanged:: 0.23.0
               Allow specifying index or column level names.
axis : {0 or 'index', 1 or 'columns'}, default 0
     Axis to be sorted.
```

## Description

- To activate the *Tab completion feature*, press the Tab key after you enter an object name, the dot after an object name, or the dot and one or more characters after an object name.
- The *tooltip feature* displays the *signature* for the method, a summary of the parameters for the method, and more.
- To activate the tooltip feature, press the Shift+Tab key with the cursor after a method name or anywhere within the parentheses for a method. However, this only works for the first method in a chain.

Figure 1-9 How to use the Tab completion and tooltip features

## How syntax and runtime errors work

When you run one or more cells, either a syntax error or a runtime error may occur. A *syntax error* occurs when you enter a statement that doesn't have the proper syntax. A *runtime error* occurs when the Python interpreter can't run a statement even though its syntax is okay.

This is illustrated by figure 1-10. In the first example, the syntax error occurred because the right bracket is missing from what should be a pair of brackets. In this case, the error message is:

```
SyntaxError: closing parenthesis ')' does not match opening
parenthesis '['
```

In the second example, the error occurs because the name of the first column that the code refers to should be "startdate", not "stardate". As a result, the interpreter can't find the column that it is looking for. This time the message is:

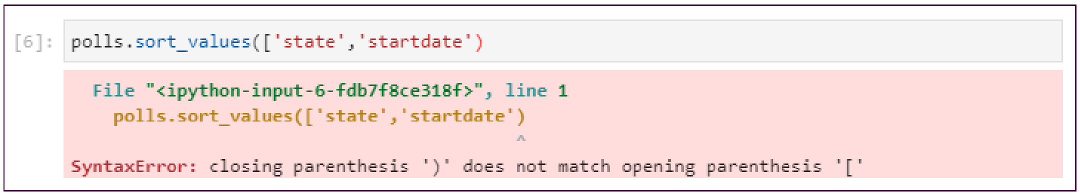
```
KeyError: 'stardate'
```

This shows that unlike some IDEs that you may have used, JupyterLab doesn't identify syntax errors as you enter them. In fact, it doesn't find them until you run the code in the cell. And then, it only finds the first syntax error in the cell. To find the other syntax errors, you have to fix the first error and run the cell again.

Once the syntax errors have been fixed, you run the cell again. Then, if the interpreter finds a runtime error, it displays the error message for it. Here again, it finds only one runtime error at a time. So you need to fix each error and run the cell again until the code works.

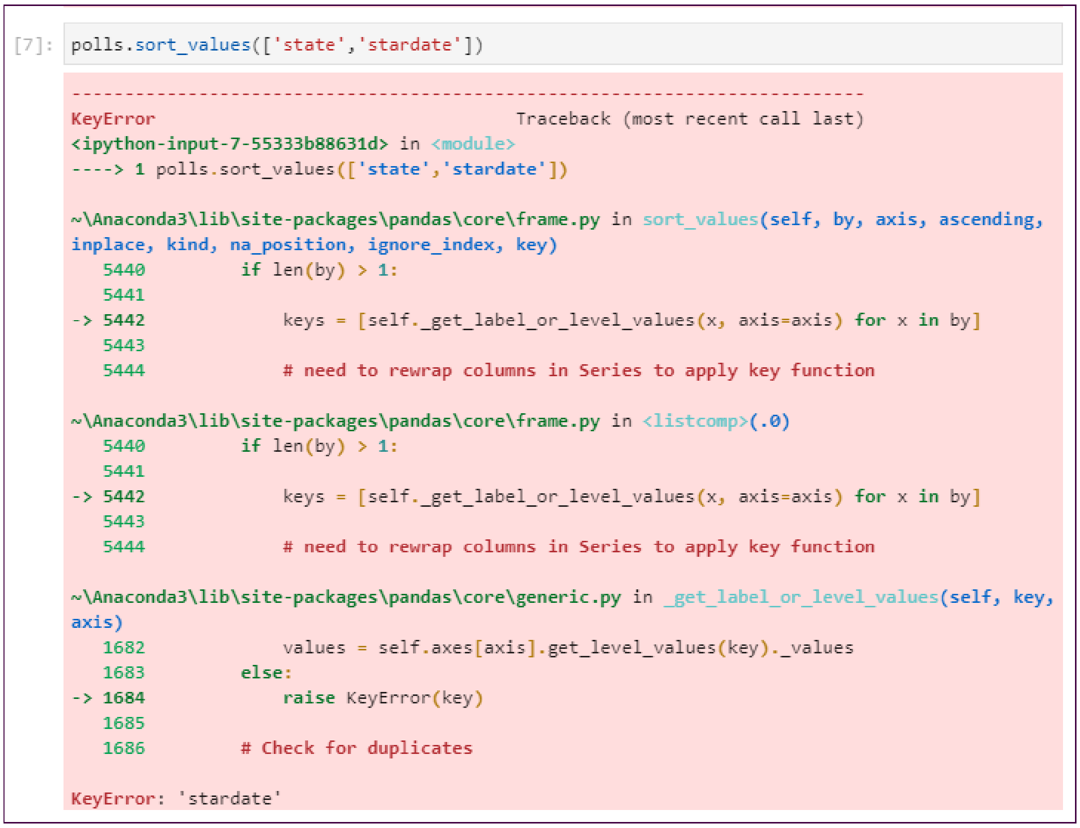
If you run several cells at the same time, you should also know that the process will stop as soon as the first syntax or runtime error occurs. If, for example, you run the code for eight cells and a syntax error occurs in the third cell, the code in the last five cells won't be run.

### A syntax error in a Notebook


```
[6]: polls.sort_values(['state','startdate')

  File "<ipython-input-6-fdb7f8ce318f>", line 1
    polls.sort_values(['state','startdate')
                                          ^
SyntaxError: closing parenthesis ')' does not match opening parenthesis '['
```

### A runtime error in a Notebook


```
[7]: polls.sort_values(['state','stardate'])

---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-7-55333b88631d> in <module>
----> 1 polls.sort_values(['state','stardate'])

~\Anaconda3\lib\site-packages\pandas\core\frame.py in sort_values(self, by, axis, ascending,
inplace, kind, na_position, ignore_index, key)
   5440         if len(by) > 1:
   5441
-> 5442             keys = [self._get_label_or_level_values(x, axis=axis) for x in by]
   5443
   5444             # need to rewrap columns in Series to apply key function

~\Anaconda3\lib\site-packages\pandas\core\frame.py in <listcomp>(.0)
   5440         if len(by) > 1:
   5441
-> 5442             keys = [self._get_label_or_level_values(x, axis=axis) for x in by]
   5443
   5444             # need to rewrap columns in Series to apply key function

~\Anaconda3\lib\site-packages\pandas\core\generic.py in _get_label_or_level_values(self, key,
axis)
   1682             values = self.axes[axis].get_level_values(key)._values
   1683         else:
-> 1684             raise KeyError(key)
   1685
   1686         # Check for duplicates

KeyError: 'stardate'
```

### Description

- A *syntax error* occurs when the code violates one of the rules of Python coding so the source code can't be interpreted.
- A *runtime error* occurs when a Python statement can't be executed.
- When you run the code in one or more cells, JupyterLab detects one syntax error at a time.
- When all syntax errors have been fixed, JupyterLab detects one runtime error at a time.

Figure 1-10 How syntax and runtime errors work

## How to use Markdown language

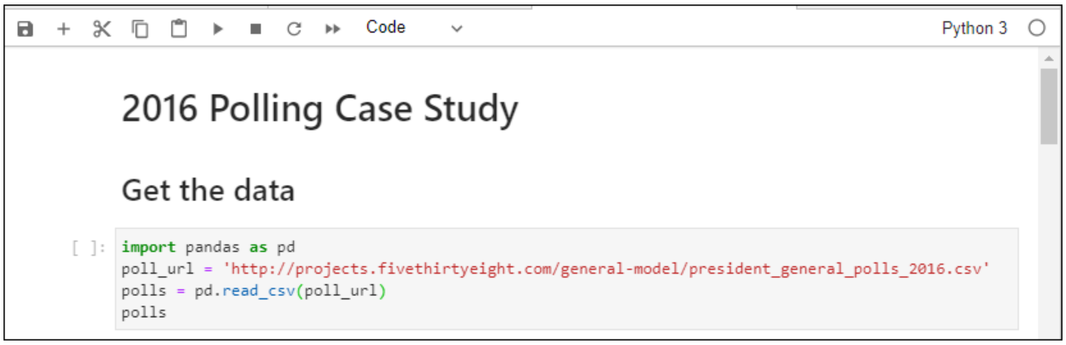
To enter a heading into a cell, you use *Markdown language* as shown in figure 1-11. With the cursor in a cell, you use the drop-down menu in the toolbar to change from Code to Markdown. Then, you enter the text for the heading preceded by from one to five hash symbols (#). Those signs will determine what level of heading is used for the text. To display the text in the cell, you run the cell.

In the second screenshot, you can see that one # is used for the top heading and two are used for the next heading. That's why these headings appear the way they do in the first screenshot.

Later, if you want to change the text in a cell, you can use the second procedure in this figure. Just double-click in the cell, modify the markdown, and run the cell.

Although the examples in this book use Markdown language only for headings, you can also use it to apply boldface, italics, numbered lists, bulleted lists, and more. For instance, the text is boldfaced if it's preceded and followed by two asterisks (**). To find out more, you can select Markdown Reference from JupyterLab's Help menu. But for most Notebooks, you shouldn't need to use Markdown language other than for headings.

### A Notebook with headings

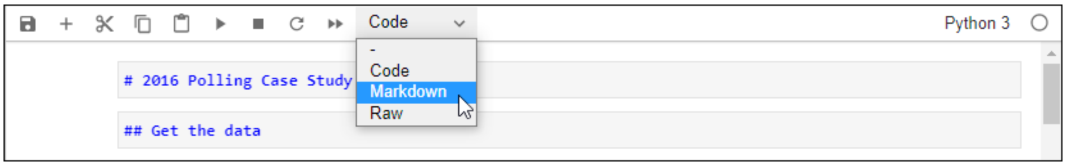


2016 Polling Case Study

Get the data

```
import pandas as pd
poll_url = 'http://projects.fivethirtyeight.com/general-model/president_general_polls_2016.csv'
polls = pd.read_csv(poll_url)
polls
```

### The Markdown language for the headings



```
# 2016 Polling Case Study
```

```
## Get the data
```

### How to create a heading by using Markdown language

- With the cursor in a new cell, change the drop-down list from Code to Markdown.
- Type the text for a heading into the cell preceded by from one to five # signs. The number of signs determines the level of the heading.
- Run the cell to convert the Markdown language to the heading.

### How to modify a heading in a Notebook cell

- Double-click in the cell to display the Markdown language, modify it, and run the cell.

### Description

- You can use *Markdown language* to create cells that contain text that identifies the contents or purposes of the cells that follow.
- To create headings and subheadings, you precede the text by from one to five hash symbols (#).
- You can also use Markdown language to apply boldface, italics, numbered lists, bulleted lists and more, but for most analyses, you shouldn't need to do that.
- For more information about Markdown language, select Markdown Reference from JupyterLab's Help menu, which is shown in the next figure.

Figure 1-11 How to use Markdown language

## How to get reference information

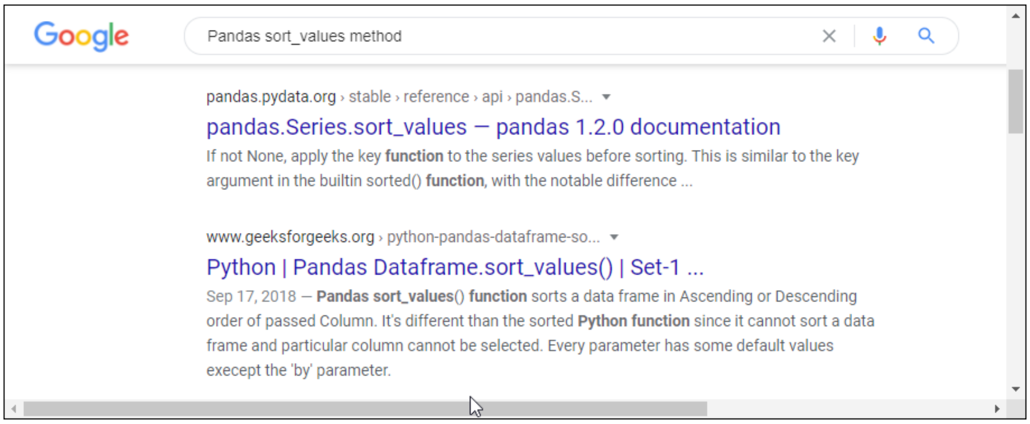
Figure 1-12 shows how to get the reference information that you need. Often, the fastest and best way to get that information is just to search the Internet for it. That way, you'll get links to all the official documentation for Pandas, Seaborn, and the other modules that you'll be using. But you'll also get links to a wider variety of information, including tutorials and videos. This is illustrated by the first screenshot in this figure.

However, you should also know that you can use the JupyterLab Help menu to get reference information. As you can see in the second example in this figure, that menu provides access to information about JupyterLab, Markdown language, Python, Pandas, and more. When you access any of this information, it's displayed in a new tab within JupyterLab.

Then, you can click on the links in the left sidebar or at the top of the tab to go to other information. You can also use the Search box at the top of the left sidebar (not shown) to search for information. For instance, the screenshot in this figure shows the result of a search for the sort_values() method, which is the same information that you can get by searching the Internet.

Remember too that tooltips provide much of the same information. For instance, the tooltip for the sort_values() method in figure 1-9 provides the same type of information that's shown in the second screenshot in this figure. That includes the signature for the method as well as a parameter summary. So, as you get more familiar with the methods for data analysis, you'll be able to use the tooltips more and reference information less.

## A Google search for the Pandas sort_values() method



pandas.pydata.org › stable › reference › api › pandas.S...

pandas.Series.sort_values — pandas 1.2.0 documentation

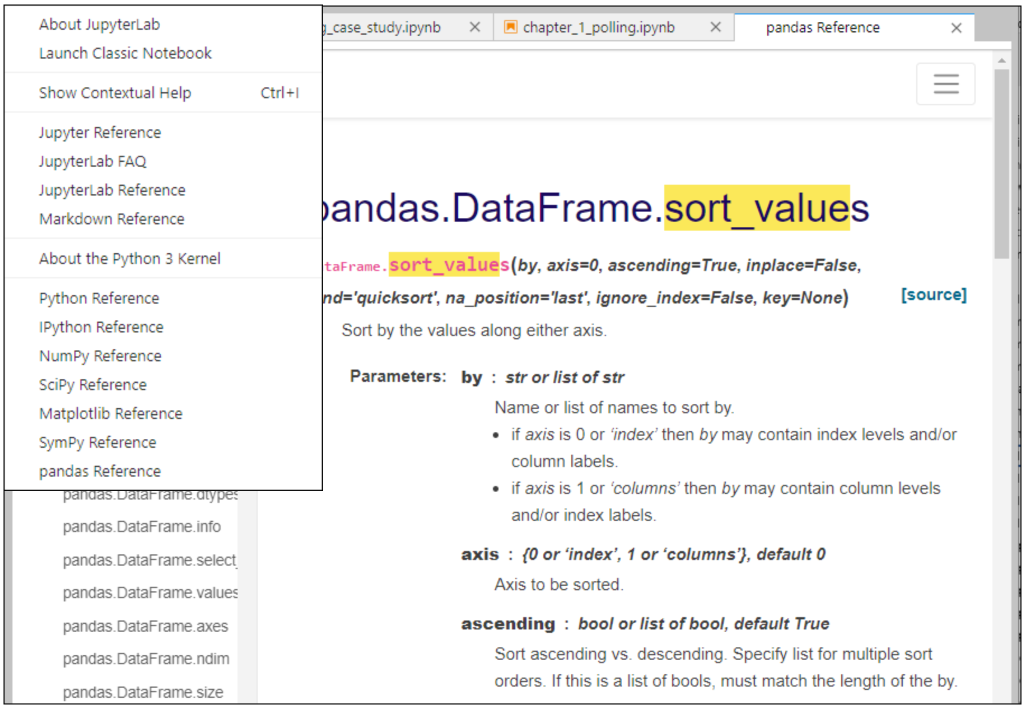
If not None, apply the key **function** to the series values before sorting. This is similar to the key argument in the builtin sorted() **function**, with the notable difference ...

www.geeksforgeeks.org › python-pandas-dataframe-so...

Python | Pandas Dataframe.sort_values() | Set-1 ...

Sep 17, 2018 — **Pandas sort_values**() **function** sorts a data frame in Ascending or Descending order of passed Column. It's different than the sorted **Python function** since it cannot sort a data frame and particular column cannot be selected. Every parameter has some default values execept the 'by' parameter.

## The JupyterLab Help menu and a page in the Pandas reference



pandas.DataFrame.sort_values

taFrame.sort_values(*by*, *axis=0*, *ascending=True*, *inplace=False*, *nd='quicksort'*, *na_position='last'*, *ignore_index=False*, *key=None*) [source]

Sort by the values along either axis.

**Parameters:** **by** : *str or list of str*

Name or list of names to sort by.

- if *axis* is 0 or *'index'* then *by* may contain index levels and/or column labels.
- if *axis* is 1 or *'columns'* then *by* may contain column levels and/or index labels.

**axis** : *{0 or 'index', 1 or 'columns'}, default 0*

Axis to be sorted.

**ascending** : *bool or list of bool, default True*

Sort ascending vs. descending. Specify list for multiple sort orders. If this is a list of bools, must match the length of the by.

## Description

- Often, the fastest way to get the reference information that you need is to search the Internet. That will lead you to the official Pandas and Seaborn documentation as well as to tutorials and more.
- You can also get reference information by using JupyterLab's Help menu. That will lead you to reference information for JupyterLab, Python, Pandas, and more.
- When you use JupyterLab, the reference information is displayed in a new tab.

Figure 1-12 How to get reference information

## Two more skills for working with JupyterLab

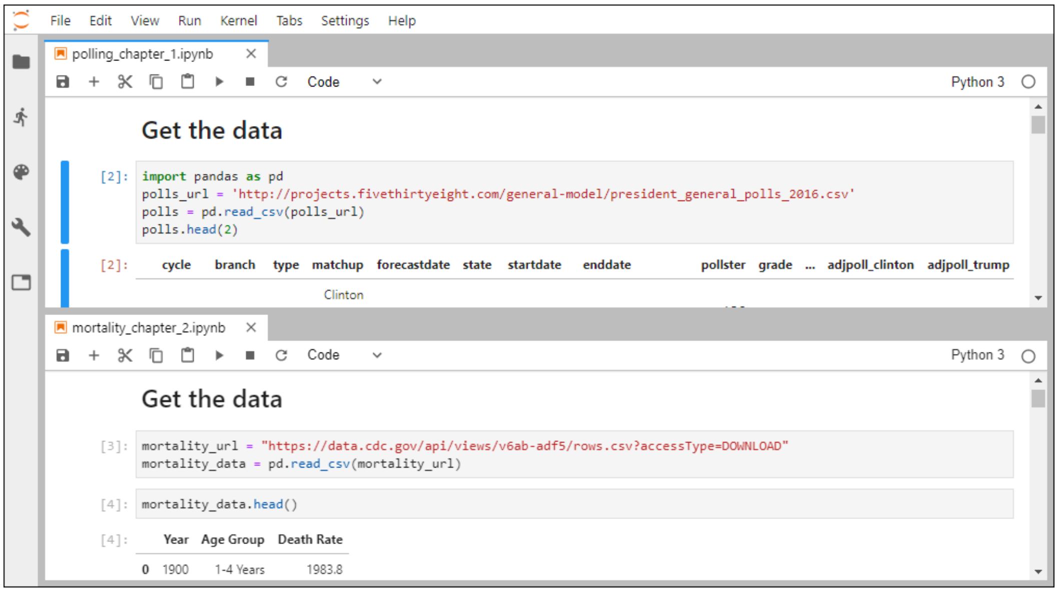
At this point, you've learned the essential skills for entering and running the cells that contain your Python code as well as for the cells that contain Markdown language. Now, you'll learn two more skills that can come in handy from time to time...although you can easily get by without either of them.

### How to split the screen between two Notebooks

Figure 1-13 shows how to split the JupyterLab screen between two Notebooks. In this example, the screen is split horizontally, but you can also split it vertically. The benefit of using split screens is that you can easily compare the code in two different Notebooks and copy code from one to the other.

To split the screen vertically, just drag the tab of one of the open Notebooks down and to the right. To split the screen horizontally, drag the tab down. To restore a tab, drag it back to the tab bar. It's that easy.

## JupyterLab with two Notebooks in a horizontally split screen



Get the data

```
[2]: import pandas as pd
     polls_url = 'http://projects.fivethirtyeight.com/general-model/president_general_polls_2016.csv'
     polls = pd.read_csv(polls_url)
     polls.head(2)
```

```
[2]:    cycle   branch   type  matchup  forecastdate  state  startdate   enddate     pollster  grade  ...  adjpoll_clinton  adjpoll_trump
                                Clinton
```

Get the data

```
[3]: mortality_url = "https://data.cdc.gov/api/views/v6ab-adf5/rows.csv?accessType=DOWNLOAD"
     mortality_data = pd.read_csv(mortality_url)
```

```
[4]: mortality_data.head()
```

```
[4]:    Year  Age Group  Death Rate
     0  1900  1-4 Years      1983.8
```

### How to split the screen

- To split the screen vertically between two open Notebooks, drag the second tab down and to the right and drop it.
- To split the screen horizontally between two open Notebooks, drag the second tab down and drop it.

### How to restore the screen

- Drag the tab of the bottom or right Notebook until it's next to the tab of the other Notebook and drop it.
- If you have trouble restoring the screen by dragging the split tab, you can close one of the Notebooks and then reopen it.

### Description

- When you use JupyterLab, you can use split screen to make it easy to compare the code in two Notebooks and copy code from one Notebook to the other.
- By contrast, when you use Jupyter Notebook, each Notebook is in a separate browser tab. So to display two Notebooks at the same time, you need to open two browsers and arrange them side by side…a cumbersome process.

Figure 1-13 How to split the screen between two Notebooks

## How to use Magic Commands

Figure 1-14 shows how to use *Magic Commands*. These are commands that aren't in the Python language. Instead, they are provided by the kernel. Although you may never need to use them, the table in this figure lists a few that can come in handy.

As the first example shows, the %time command returns the time that it takes to run a Python statement. In this case, it took the read_csv() method 7.37 seconds to import the code in the file. This command can be helpful when you're trying to improve the performance of your code.

Similarly, as the second example shows, the %%time command can be coded at the start of a cell that contains more than one statement. Then, the time to run all of the statements is displayed. This lets you time something like a for loop. In the example in this figure, it took 9.92 milliseconds to execute the two statements that follow the Magic Command.

In contrast to the timing commands, the %whos command returns a table of the active variables, plus the data type and some other information for each of the variables. This is illustrated by the third example in this figure. This information can be helpful when you're debugging because some methods can only be applied to specific data types. In this case, the command shows that only three variables have been created, and their data types are module, str (string), and DataFrame.

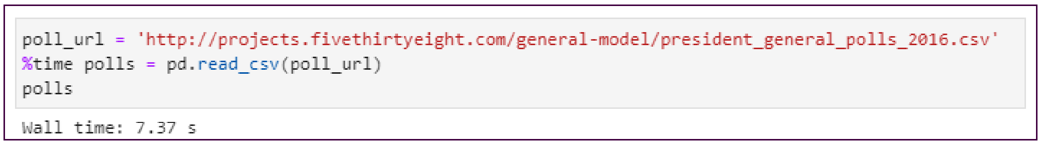
The fourth example shows that you can also use the Python type() function to get the data type of a variable. Here, the first type() function shows that the poll_url variable is of the str type. And the second type() function shows that the polls variable is a DataFrame. If you just want to find out what the data type of one variable is, this function is all that you need. But as you've seen, the %whos Magic Command displays the data types for all of the active variables in a single command.

The other useful Magic Command is the %magic command. But beware, it returns the documentation for all of the Magic commands, which can be overwhelming. Perhaps a better alternative is to search the Internet for something like "best Magic Commands for Python."

## Four of the most useful Magic Commands

| Command | Description |
|---|---|
| `%time` | Displays the time that it takes for a statement to run. |
| `%%time` | Displays the time that it takes for all the statements in a cell to run. |
| `%whos` | Displays the variables that are in the namespace along with their data types. |
| `%magic` | Displays a reference for all of the Magic commands. |

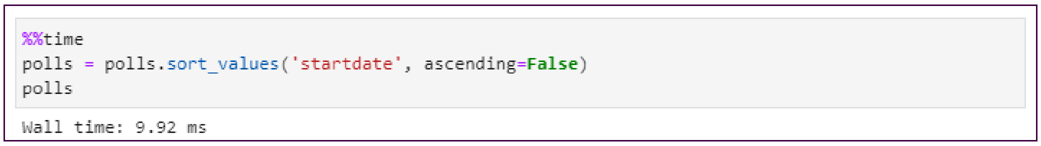
### How the %time command works



```
poll_url = 'http://projects.fivethirtyeight.com/general-model/president_general_polls_2016.csv'
%time polls = pd.read_csv(poll_url)
polls

Wall time: 7.37 s
```

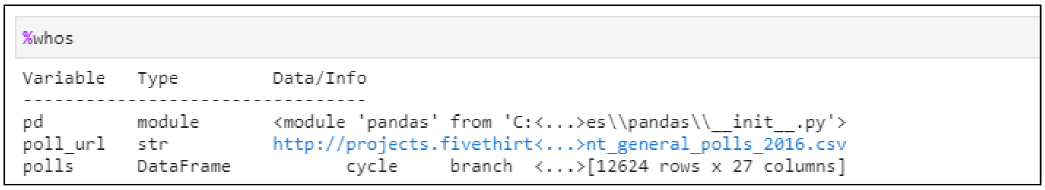
### How the %%time command works



```
%%time
polls = polls.sort_values('startdate', ascending=False)
polls

Wall time: 9.92 ms
```

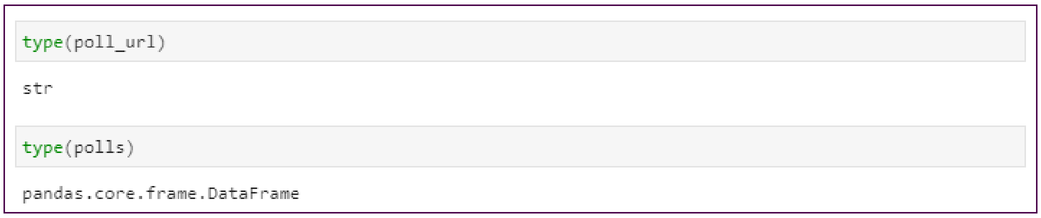
### How the %whos command works



```
%whos

Variable   Type         Data/Info
---------------------------------
pd         module       <module 'pandas' from 'C:<...>es\\pandas\\__init__.py'>
poll_url   str          http://projects.fivethirt<...>nt_general_polls_2016.csv
polls      DataFrame           cycle     branch  <...>[12624 rows x 27 columns]
```

## How to use the Python type() function to check the data type of a variable



```
type(poll_url)

str

type(polls)

pandas.core.frame.DataFrame
```

## Description

- Magic Commands provide some useful functions that aren't provided by the Python functions.
- In general, the commands that start with % apply to one statement, and the commands that start with %% apply to all the statements in the cell.

Figure 1-14 How to use Magic Commands

## Introduction to the case studies

Section 4 of this book presents four case studies that show how the skills that you learn in this book are applied to real-world analyses. The next four figures introduce these case studies because the chapters in section 2 often present examples taken from them. Although those examples should be self-explanatory, this introduction should help you see the examples in a larger context.

Beyond that, this introduction is intended to encourage you to look through the case studies as you progress through this book. In fact, before you're through with this book, you should understand every line of code in each of the case studies. Since these are real-world analyses, that will show that you have mastered data analysis at a professional level.

## The Polling case study

Figure 1-15 introduces the Polling case study. The data for this case study comes from the FiveThirtyEight.com website, a well-known analytical site for politics, economics, and sports. This data is for the polls that were taken for the 2016 presidential election in the United States. You may remember that most predictions were for Hillary Clinton to win, but somehow Donald Trump won.

As you can see in this figure, when the polling data is imported into a Pandas DataFrame, it consists of 12,624 rows. That's because there are three rows for each poll that was taken. The DataFrame also consists of 27 columns, but many of them aren't needed for the analysis. So, after the data is cleaned, the DataFrame is down to 4,116 rows and 10 columns.

When the data is prepared, voter_type, state_gap, and swing columns are added to the cleaned DataFrame. Then, a second DataFrame is prepared that shapes the data in a new way. This is illustrated by the third table in this figure. It consists of 8,232 rows and just 9 columns. In this DataFrame, there are two rows for each poll: one with the results for Clinton, the other with the results for Trump. Next, the month_bin and month_pct_avg columns are added to help improve the visualizations for this data.

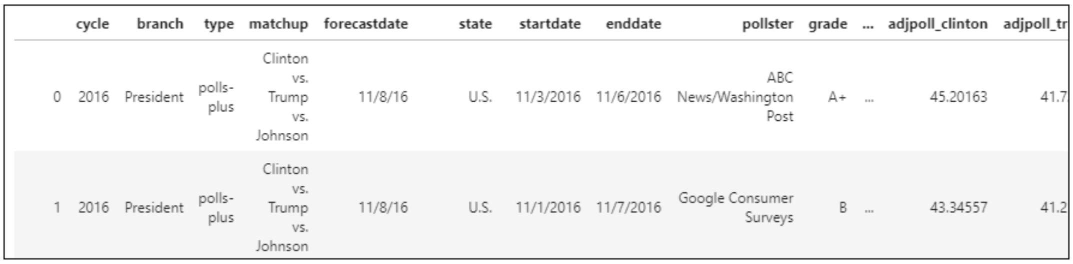
The visualization in this figure is one of the several in this case study. In this case, the Seaborn relplot() method was used to plot the visualization from the data in the prepared DataFrame that's shown above it. This visualization shows the results of the polls for the swing states during the last 2 months before the election. This shows that those polls were always close but with some wide variations. And this shows that the polls got very close in the last few weeks before the election.

This complete case study in presented in chapter 12. There, you can see how the skills in this book are used to get, clean, prepare, analyze, and visualize the data. And that will give you the perspective that you need for applying those skills to your own analyses.

## The URL for the Polling data

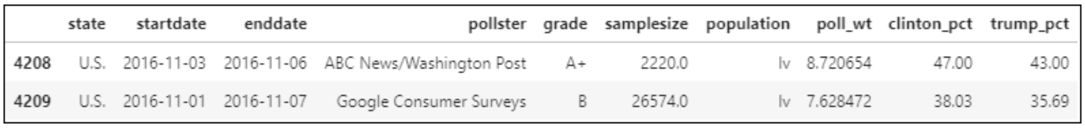
```
http://projects.fivethirtyeight.com/general-model/president_general_polls_2016.csv
```

## The imported DataFrame (12,624 rows and 27 columns)



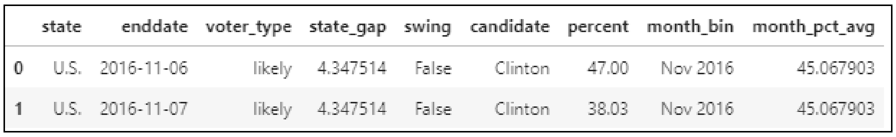
| | cycle | branch | type | matchup | forecastdate | state | startdate | enddate | pollster | grade | ... | adjpoll_clinton | adjpoll_tr |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 2016 | President | polls-plus | Clinton vs. Trump vs. Johnson | 11/8/16 | U.S. | 11/3/2016 | 11/6/2016 | ABC News/Washington Post | A+ | ... | 45.20163 | 41.7 |
| 1 | 2016 | President | polls-plus | Clinton vs. Trump vs. Johnson | 11/8/16 | U.S. | 11/1/2016 | 11/7/2016 | Google Consumer Surveys | B | ... | 43.34557 | 41.2 |

## The cleaned DataFrame (4,116 rows and 10 columns)



| | state | startdate | enddate | pollster | grade | samplesize | population | poll_wt | clinton_pct | trump_pct |
|---|---|---|---|---|---|---|---|---|---|---|
| 4208 | U.S. | 2016-11-03 | 2016-11-06 | ABC News/Washington Post | A+ | 2220.0 | lv | 8.720654 | 47.00 | 43.00 |
| 4209 | U.S. | 2016-11-01 | 2016-11-07 | Google Consumer Surveys | B | 26574.0 | lv | 7.628472 | 38.03 | 35.69 |

## The prepared DataFrame (8,232 rows and 9 columns)



| | state | enddate | voter_type | state_gap | swing | candidate | percent | month_bin | month_pct_avg |
|---|---|---|---|---|---|---|---|---|---|
| 0 | U.S. | 2016-11-06 | likely | 4.347514 | False | Clinton | 47.00 | Nov 2016 | 45.067903 |
| 1 | U.S. | 2016-11-07 | likely | 4.347514 | False | Clinton | 38.03 | Nov 2016 | 45.067903 |

## A Seaborn plot of the swing state polls in the 2 months before the election

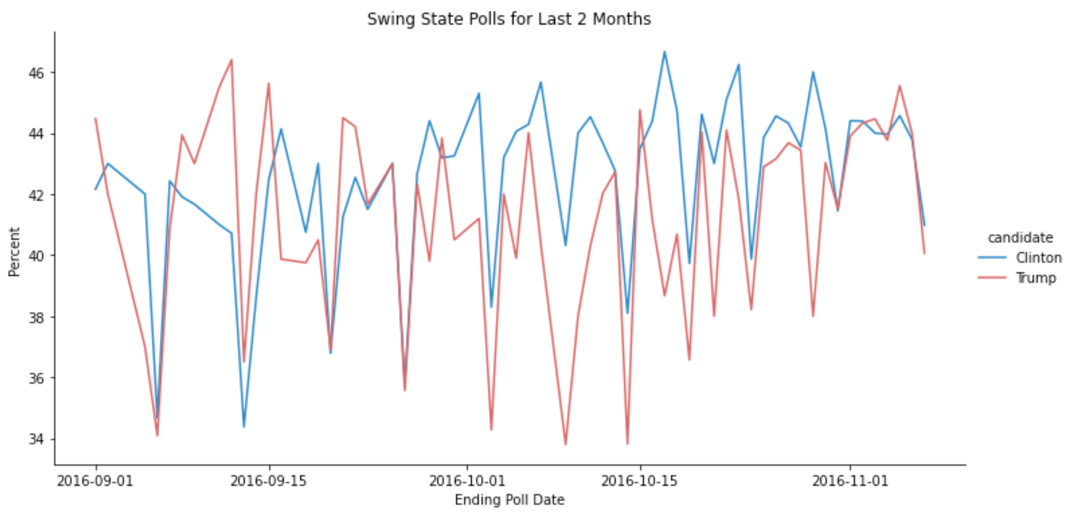


Figure 1-15 Introduction to the Polling case study

## The Forest Fires case study

Figure 1-16 introduces the Forest Fires case study. This study analyzes the data for forest fires from 1992 through 2015. This study gets the data from the website for the US Forest Service, which is part of the US Department of Agriculture. The data is available in several forms, but this case study gets it as a SQLite database.

After the data is imported from the database, the DataFrame consists of just 8 columns but 1,880,465 rows. The first two rows of this DataFrame are shown in this figure. Since the SQL statement that's used to import this data selects the columns for the analysis, this DataFrame should require minimal cleaning.

In fact, the primary cleaning is to drop the rows for all fires that are less than 10 acres. But besides that, the column names are changed from upper to lowercase, and the fire names are changed to title case. That just makes it easier to work with the data.

Then, to prepare the DataFrame, just two columns are added: one for the month of the fire and one for the number of days that it took to contain the fire. So, the cleaned and prepared DataFrame consists of 247,123 rows and 10 columns.

The two visualizations in this figure illustrate the types of analysis that are done by this case study. The first one uses Pandas to plot the total number of acres that were burned in 2015 for the states with the top 10 totals. As you can see, Alaska had the most acres with more than 3,000,000 acres burned. And Idaho and California were second and third, both with more than 1,000,000 acres burned.

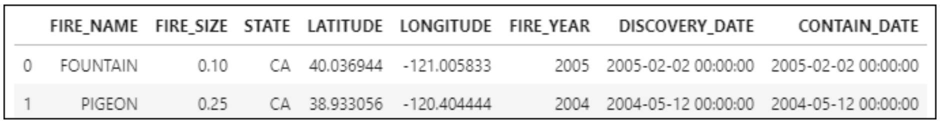
The second visualization uses Seaborn to plot the locations of the California fires in 2015 that were greater than or equal to 500 acres. Here, the size and darkness of each dot indicates the size of the fire: the larger and darker the dot, the larger the fire. To lay this plot on an outline of the state of California also required the installation and use of the GeoPandas module.

This complete case study is presented in chapter 13. There, you can see how the skills in this book are used to get, clean, prepare, analyze, and visualize the data. You'll also see how a module like GeoPandas can be used with Seaborn to create plots that are placed over the outlines of countries and states.

### The URL for the Forest Fires data

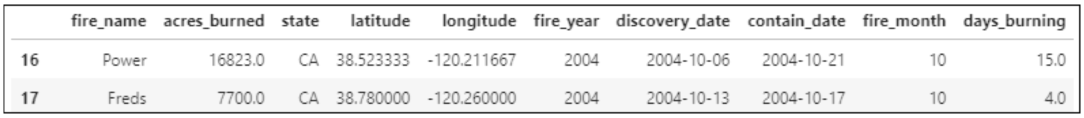
```
https://www.fs.usda.gov/rds/archive/products/RDS-2013-0009.4/RDS-2013-0009.4_SQLITE.zip
```

### The imported DataFrame (1,880,465 rows and 8 columns)



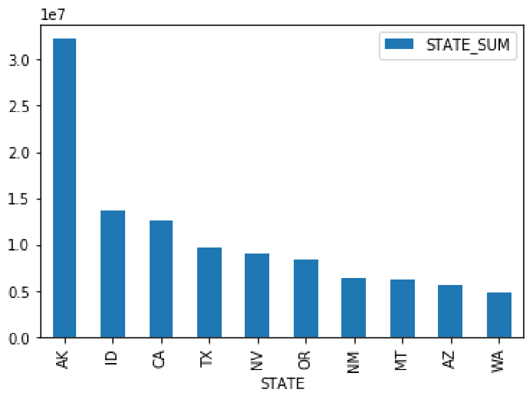
| | FIRE_NAME | FIRE_SIZE | STATE | LATITUDE | LONGITUDE | FIRE_YEAR | DISCOVERY_DATE | CONTAIN_DATE |
|---|---|---|---|---|---|---|---|---|
| 0 | FOUNTAIN | 0.10 | CA | 40.036944 | -121.005833 | 2005 | 2005-02-02 00:00:00 | 2005-02-02 00:00:00 |
| 1 | PIGEON | 0.25 | CA | 38.933056 | -120.404444 | 2004 | 2004-05-12 00:00:00 | 2004-05-12 00:00:00 |

### The prepared DataFrame (247,123 rows and 10 columns)



| | fire_name | acres_burned | state | latitude | longitude | fire_year | discovery_date | contain_date | fire_month | days_burning |
|---|---|---|---|---|---|---|---|---|---|---|
| 16 | Power | 16823.0 | CA | 38.523333 | -120.211667 | 2004 | 2004-10-06 | 2004-10-21 | 10 | 15.0 |
| 17 | Freds | 7700.0 | CA | 38.780000 | -120.260000 | 2004 | 2004-10-13 | 2004-10-17 | 10 | 4.0 |

### A Pandas plot of the total acres burned in the top 10 fire states in 2015



### A GeoPandas and Seaborn plot of the California fires over 500 acres in 2015

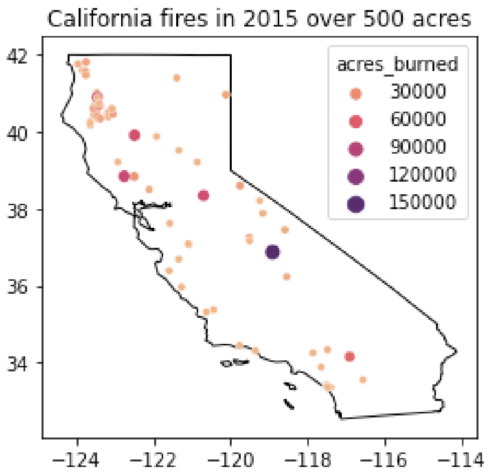


Figure 1-16 Introduction to the Forest Fires case study

## The Social Survey case study

Figure 1-17 introduces the Social Survey case study. This study analyzes the data compiled by the National Opinion Research Center in Chicago. This data was compiled from 1972 to the present, and the questions varied from year to year. The result is that the data consists of 64,814 rows that represent the people who responded to the questions, and 6,110 columns that represent the questions that were asked.

This data is provided in a Stata file, which consists of two parts: metadata that provides information about the data, and the data itself. The trouble is that this dataset is so large that a computer needs about 3 gigabytes of memory to import it into a DataFrame. But that means that many computers won't be able to import the data.

For that reason, you need to use the metadata and the documentation that comes with the Stata file to figure out which portions of the data you want to import. For instance, this figure shows one of the DataFrames that this case study prepares for analysis. It consists of just 128 rows and 5 columns with the focus on the work status column (wrkstat) in the Stata file. Then, the plot that follows shows the changes in work status from 1972 to the present.

This example is typical of the approach that this case study takes to getting useful information from this dataset. First, you figure out which columns (questions) you want to work with. Then, you import and analyze those columns. As you will see when you read chapter 14, this data can lead the way to many insights about social behavior.

### The URL for the Social Survey data

```
http://gss.norc.org/Documents/stata/gss_stata_with_codebook.zip
```

### The starting dataset

- 64,814 rows and 6,110 columns
- This dataset is so large that importing it requires 3 gigabytes of memory. So instead of importing the entire file, you should import just the subsets of data that you need for your analyses.

### One of the DataFrames that's prepared for analysis (128 rows, 5 columns)

| | year | wrkstat | counts | countsTotal | percent |
|---|---|---|---|---|---|
| 0 | 1972 | working fulltime | 750 | 1061 | 0.706880 |
| 1 | 1972 | working parttime | 121 | 1061 | 0.114043 |
| 2 | 1972 | unempl, laid off | 46 | 1061 | 0.043355 |
| 3 | 1972 | retired | 144 | 1061 | 0.135721 |
| 4 | 1973 | working fulltime | 651 | 974 | 0.668378 |

### A Seaborn plot derived from the DataFrame

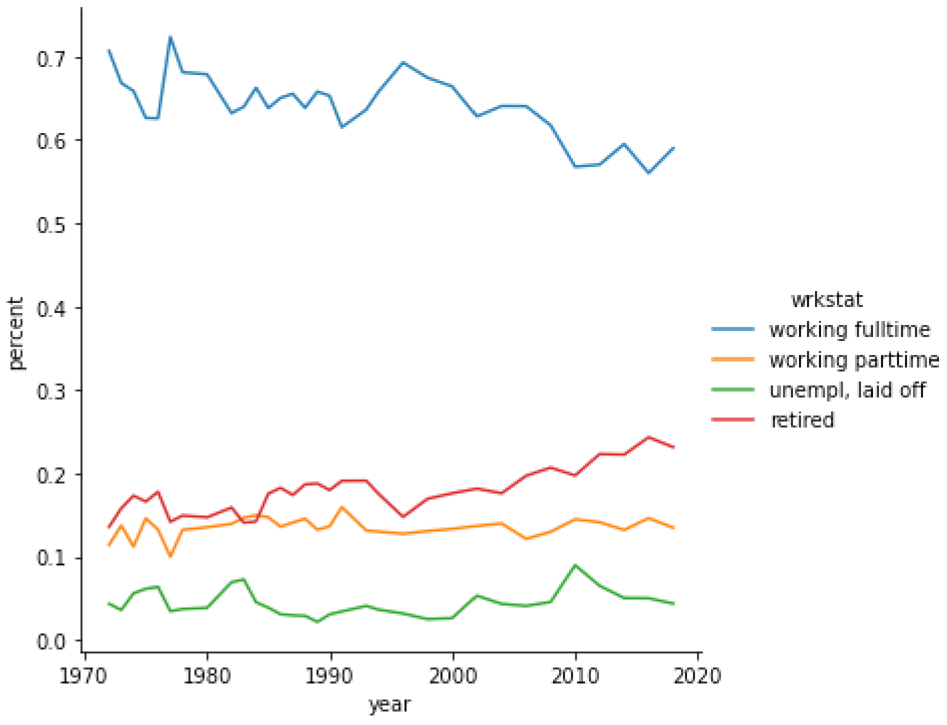


Figure 1-17 Introduction to the Social Survey case study

## The Sports Analytics case study

Figure 1-18 introduces the Sports Analytics case study. This case study analyzes the shots that Stephen Curry of the Golden State Warriors took from 2009 through 2019. Since this data is provided in JSON format that's several levels deep, it can't just be imported into a DataFrame. Instead, you need to use the DataFrame() constructor to build the DataFrame from the available data.

After the DataFrame is constructed, it contains one row for every shot that Curry took during that time period with 24 columns of data. Since most of those columns aren't needed for the shot analysis, all but six are dropped when the data is cleaned. But then six summary columns are added when the data is prepared. These are the last six columns in the second DataFrame in this figure.

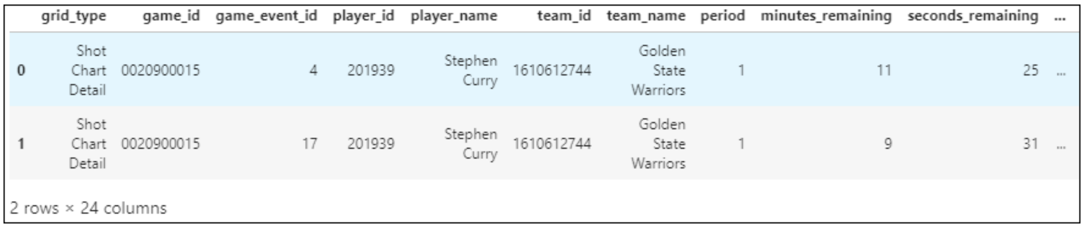
The analysis in this case study shows how Curry's statistics like shots and points per season changed as his career progressed. But beyond that, it focuses on how his shot selection changed from one year to the next. In this figure, for example, you can see a scatter plot of the shot locations for his missed and made shots in two different seasons: his rookie season and the first season in which he was the Most Valuable Player.

This complete case study is presented in chapter 15. There, you can see how the skills in this book are used to get, clean, prepare, analyze, and visualize the data. You can also see how the diagram of the basketball floor can be added to plots.

## The URL for the Sports Analytics data

```
https://www.murach.com/python_analysis/shots.json
```

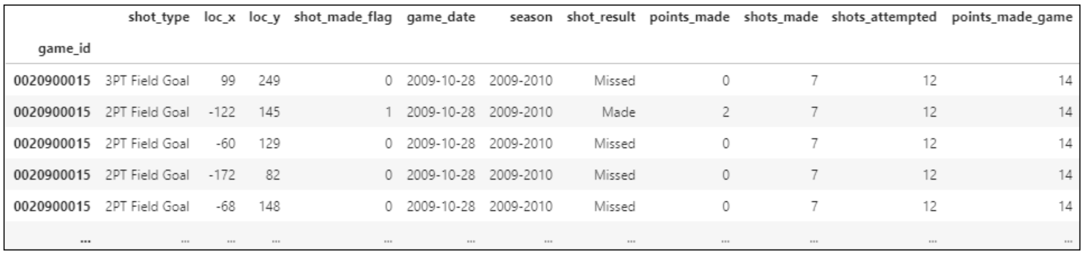
## The imported DataFrame (11,846 rows and 24 columns)



| | grid_type | game_id | game_event_id | player_id | player_name | team_id | team_name | period | minutes_remaining | seconds_remaining | ... |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | Shot Chart Detail | 0020900015 | 4 | 201939 | Stephen Curry | 1610612744 | Golden State Warriors | 1 | 11 | 25 | ... |
| **1** | Shot Chart Detail | 0020900015 | 17 | 201939 | Stephen Curry | 1610612744 | Golden State Warriors | 1 | 9 | 31 | ... |

2 rows × 24 columns

## The prepared DataFrame (11,753 rows and 12 columns)



| | shot_type | loc_x | loc_y | shot_made_flag | game_date | season | shot_result | points_made | shots_made | shots_attempted | points_made_game |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **game_id** | | | | | | | | | | | |
| **0020900015** | 3PT Field Goal | 99 | 249 | 0 | 2009-10-28 | 2009-2010 | Missed | 0 | 7 | 12 | 14 |
| **0020900015** | 2PT Field Goal | -122 | 145 | 1 | 2009-10-28 | 2009-2010 | Made | 2 | 7 | 12 | 14 |
| **0020900015** | 2PT Field Goal | -60 | 129 | 0 | 2009-10-28 | 2009-2010 | Missed | 0 | 7 | 12 | 14 |
| **0020900015** | 2PT Field Goal | -172 | 82 | 0 | 2009-10-28 | 2009-2010 | Missed | 0 | 7 | 12 | 14 |
| **0020900015** | 2PT Field Goal | -68 | 148 | 0 | 2009-10-28 | 2009-2010 | Missed | 0 | 7 | 12 | 14 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |

## A Seaborn plot for how Curry's shot selection changed

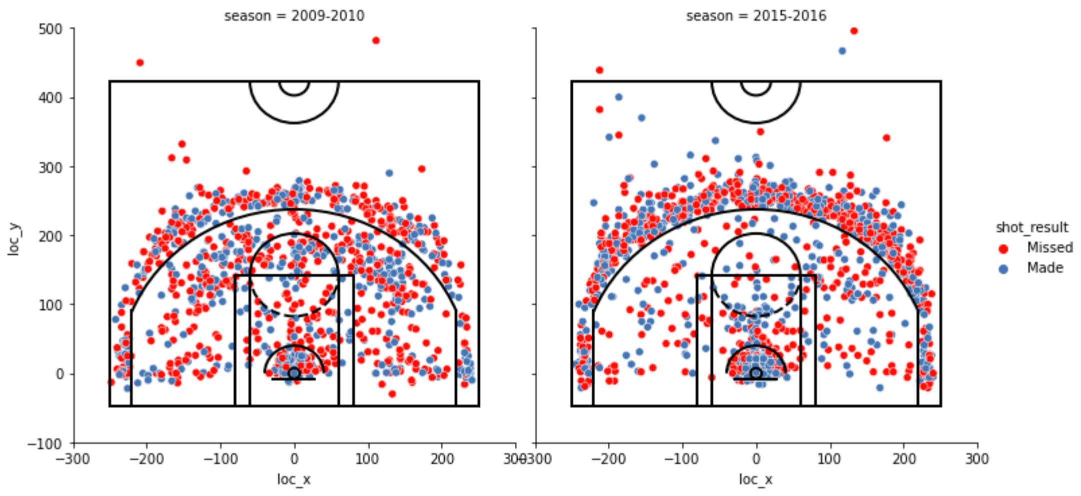


Figure 1-18 Introduction to the Sports Analytics case study

## Perspective

Now that you've completed this chapter, you should be able to use JupyterLab to work with Notebooks. You should also realize that to master data analysis and visualization, you need to learn how to use the methods of modules like Pandas and Seaborn to get, clean, prepare, analyze, and visualize the data. With that as background, you're ready to learn the essential Pandas and Seaborn skills, and you'll start that in the next chapter.

## Terms

data analysis
data visualization (data viz)
data modeling
predictive analysis
data analytics
business analytics
sports analytics
data science
get the data
import data
DataFrame
clean the data
prepare the data
analyze the data
visualize the data
Anaconda distribution
Integrated Development Environment (IDE)
JupyterLab
Jupyter Notebook
Notebook
module
install a module
!pip command
conda command
Anaconda Prompt
import a module
import statement
call a method
parameter
dot notation
chain methods
signature
positional parameter
keyword parameter
list
tuple
dictionary (dict)
slice
list comprehension
implicit continuation
explicit continuation
checkpoint
Notebook cell
kernel
Tab completion feature
tooltip feature
syntax error
runtime error
Markdown language
Magic Command

## Summary

- The term *data analysis* (or *data analytics*) includes *data visualization* and *predictive analysis*. The term *data science* not only includes data analysis but also data mining, machine learning, deep learning, and artificial intelligence (AI).
- A data analysis project can be divided into five phases: *get*, *clean*, *prepare*, *analyze*, and *visualize* the data. Often, though, there's overlap between the phases.
- Three of the most popular IDEs for data analysis are *Jupyter Notebook*, *JupyterLab*, and *VS Code*. For this book, we recommend that you use JupyterLab as your IDE.
- The *Anaconda distribution* of Python includes JupyterLab as well as the Pandas and Seaborn *modules* that you'll use for data analysis and visualization.
- When you use Python for data analysis, you need to know how to *install* and *import* any *modules* that aren't included in your Python distribution. You need to know how to call and chain methods. And you need to know how to code *lists*, *tuples*, *dictionaries*, *slices*, and *list comprehensions*.
- A Jupyter Notebook consists of *cells* that contain either Python code or *Markdown language* that provides for five levels of headings. When you run a cell that contains code, any results are shown below the cell.
- The *kernel* for a Notebook is the Python interpreter that runs the code. You can use the Kernel menu in JupyterLab to restart the kernel and clear all output or run all of the cells.
- Two of the JupyterLab features that you'll want to use are the *Tab completion* and *tooltip features*.
- If a cell has a *syntax error* or a *runtime error*, an error message is displayed below the cell when you try to run it, but only one error is detected at a time.
- You can use the Help menu in JupyterLab to get reference information that is displayed in one tab of the IDE. Often, though, it's faster and easier to search for the information that you need on the Internet.
- JupyterLab lets you split the screen between two open Notebooks. That makes it easier to copy code from one Notebook to the other.
- You can code *Magic Commands* in the cells of a Notebook for special purposes like timing how long it takes to run a statement and displaying a table of the active variables and their data types.

## Before you do the exercises for this chapter

Before you do any of the exercises in this book, you need to install the Anaconda distribution of Python. You also need to install the JupyterLab Notebooks for this book. For details, see the appendix for your operating system.

## Exercise 1-1 Get started with JupyterLab

This exercise is designed to get you started with JupyterLab. It is not designed to teach you coding skills.

### Start JupyterLab and set its theme

1. Start Anaconda Navigator. Then, use the Navigator to launch JupyterLab, and note that JupyterLab runs in your browser.
2. Use the Settings menu to determine if the theme for JupyterLab is set to Dark or Light. Then, change the theme to see if you like it better. If not, change it back. The screenshots in this book use JupyterLab Light, and that's the one we recommend.

### Open the Polling Notebook and run the first three cells

3. Use the File Browser shown in figure 1-7 to open the Notebook named ex_1-1_polling that should be in this folder:
   ```
   python_analysis/exercises/ch01
   ```
4. Use the Kernel menu to restart the kernel and clear all outputs.
5. Click in the first cell that contains code and run the cell by clicking on the Run button in the toolbar. This code uses the Pandas read_csv() method to import polling data from the FiveThirtyEight website into a DataFrame named polls.
6. With the cursor in the next cell, press Shift+Enter to run the statement in that cell and see how it displays the data in the polls DataFrame. Next, change the code to:
   ```
   polls.head()
   ```
   Then, run the cell and note that the head() method displays just the first five rows in the DataFrame.
7. Run the sort_values() method in the third cell. Note that this method has one positional parameter (state) and one keyword parameter.

### Fix syntax and runtime errors

8. Still in the third cell, delete the e in state to make it stat, and delete the right parenthesis at the end of the sort_values() method.
9. Run that cell and note that the error message says that a syntax error occurred. Fix that by replacing the right parentheses, and run the cell again.
10. This time, there's a runtime error because there's no column named "stat". Fix that, and run the cell again. Now, the sort should work.

### Use Markdown language, Tab completion, and tooltips

11. Add a new cell after the one for the sort_values() method. Then, with figure 1-11 as a guide, use Markdown language to create a subheading that says: "Use Tab completion and tooltips".
12. Add a cell after the one you just created. Then, with figure 1-9 as a guide, enter:
    ```
    polls.s
    ```
    and press the Tab key. That should display a completion list, although it may take a while for it to be displayed. Then, type the letter *o* to refine the completion list, and select sort_values from the list.
13. Enter a set of parentheses after the sort_values method name, and with the cursor in the parentheses, press Shift+Tab to display the tooltip. Note that the signature lists one positional parameter (by) and five keyword parameters. Then, scroll through the tooltip to see all that it offers.
14. Finish the sort_value() method so it looks like this:
    ```
    sort_values('state', ascending=False)
    ```
    Next, run the statement to see how it displays the results. Then, chain a head() method to the sort_values() method, and run it to see the results.
15. Try the tooltip feature for the sort_values() method that you just coded, and see that it still works. But try it for the head() method, and you'll see that it only works for the first method in a chain.

### Run the rest of the cells

16. Run the rest of the cells in the Notebook. To do that, press Shift+Enter for each cell or click on the Run button in the toolbar. (If the first cell for plotting doesn't display a plot when you run it, run it a second time.)
17. Review the code in each of the cells that you've just run. Note the use of a dictionary in the rename() method and the use of lists in some of the other statements. Note too that the plot() method in the last cell of this Notebook is chained to the query() method.

### Use two Magic Commands and the Python type() function

18. Add the %%time Magic Command as the first line in the cell for the chained plot() method. Then, run the cell to see how long it takes.
19. In the new cell at the end of the Notebook, run the %whos command and note the data types for all the variables that have been created by this Notebook.
20. In the next cell, run the Python type() function for the polls DataFrame, which is another way to identify the data type for a variable.

### Start a new Notebook, use a split screen, and copy code into it

21. Use File→New Launcher to open a new tab in JupyterLab, and click on the first Python 3 icon to start a new Notebook.
22. Right-click on the tab for the new file, which will say "Untitled.ipynb", select Rename in the popup menu, and change the name to ex_1-1_practice.
23. Select the first four cells in the chapter_1_exercise Notebook (the two heading cells and the two code cells), right-click on them, and select Copy Cells from the popup menu. Then, go to the tab for the new Notebook, right-click on the empty first cell, and select Paste Cells Below.
24. After you paste the cells into the new Notebook, the first cell in the notebook will be empty. To delete that cell, right-click on it and select Delete Cells from the popup menu.
25. With figure 1-13 as a guide, split the tabs for the two Notebooks vertically. Then, copy the next two or three cells from the first Notebook to the new Notebook.
26. Restore the split screens to a single screen.

### Experiment on your own

27. If you have the time and interest, experiment with some of the other ways that JupyterLab provides for getting the results you want. Otherwise, close both Notebooks.

# How to become a data analyst

*Mike Murach, Publisher*