CHAPTER 10
First Steps with Python for Excel Users

Created in 1991 by Guido van Rossum, Python is a programming language that, like R, is free and open source. At the time, van Rossum was reading the scripts from *Monty Python's Flying Circus* and decided to name the language after the British comedy. Unlike R, which was designed explicitly for data analysis, Python was developed as a general-purpose language meant to do things like interact with operating systems, handle processing errors, and so forth. This has some important implications for how Python "thinks" and works with data. For example, you saw in Chapter 7 that R has a built-in tabular data structure. This isn't the case in Python; we'll need to rely more heavily on external packages to work with data.

That's not necessarily a problem: Python, like R, has thousands of packages maintained by a thriving contributor community. You'll find Python used for everything from mobile app development to embedded devices to, yes, data analytics. Its diverse user base is growing rapidly, and Python has become one of the most popular programming languages not just for analytics but for computing generally.

Python was conceived as a general-purpose programming language, while R was bred specifically with statistical analysis in mind.

Downloading Python

The Python Software Foundation (*https://python.org*) maintains the "official" Python source code. Because Python is open source, anyone is available to take, add to, and redistribute Python code. Anaconda is one such Python distribution and is the suggested installation for this book. It's maintained by a for-profit company of the same name and is available in paid tiers; we'll be using the free Individual Edition. Python is now on its third version, Python 3. You can download the latest release of Python 3 at Anaconda's website (*https://oreil.ly/3RYeQ*).

Python 2 and Python 3

Python 3, released in 2008, made significant changes to the language and importantly was not backward compatible with code from Python 2. This means that code written for Python 2 may not necessarily run on Python 3, and vice versa. At the time of writing, Python 2 has been officially retired, although you may encounter some references and code remnants in your Python journey.

In addition to a simplified installation of Python, Anaconda comes with extras, including some popular packages which we'll use later in the book. It also ships with a web application that we'll use to work with Python: the Jupyter Notebook.

Getting Started with Jupyter

As mentioned in Chapter 6, R was modeled after the S program for EDA. Because of the iterative nature of EDA, the expected workflow of the language is to execute and explore the output of selected lines of code. This makes it easy to conduct data analysis directly from an R script, *.r*. We used the RStudio IDE to provide additional support for our programming session, such as dedicated panes for help documentation and information about the objects in our environment.

By contrast, Python in some ways behaves more like "lower-level" programming languages, where code needs first to be compiled into a machine-readable file, and *then* run. This can make it relatively trickier to conduct piecemeal data analysis from a Python script, *.py*. This pain point of working with Python for statistical and, more broadly, scientific computing caught the attention of physicist and software developer Fernando Pérez, who with colleagues in 2001 launched the IPython project to make a more interactive interpreter for Python (IPython as a playful shorthand for "interactive Python"). One result of this initiative was a new type of file, the *IPython Notebook*, denoted with the *.ipynb* file extension.

This project gained traction and in 2014, IPython was spun into the broader Project Jupyter, a language-agnostic initiative to develop interactive, open source computing software. Thus, the IPython Notebook became the Jupyter Notebook while retaining the *.ipynb* extension. Jupyter notebooks run as interactive web applications that allow users to combine code with text, equations, and more to create media-rich interactive documents. In fact, Jupyter was named in part as an homage to the notebooks Galileo used to record his discovery of the planet Jupiter's moons. A *kernel* is used behind the scenes to execute the notebook's code. By downloading Anaconda, you've set up all these necessary parts to execute Python from a Jupyter notebook: now you just need to launch a session.

RStudio, Jupyter Notebooks, and Other Ways to Code

You may be unhappy to leave RStudio to learn yet another interface. But remember that code and application are often decoupled in open source frameworks; it's easy to "remix" these languages and platforms. For example, R is one of the many dozens of languages with a kernel for Jupyter. Along with the Galileo reference, Jupyter is *also* a portmanteau of its three core supported languages: Julia, Python, and R.

Conversely, it's possible to execute Python scripts from inside RStudio with the help of R's `reticulate` package, which can more broadly be used to run Python code from R. This means it's possible to, for example, import and manipulate data in Python and then use R to visualize the results. Other popular programs for working with Python code include PyCharm and Visual Studio Code. RStudio also has its own notebook application with R Notebooks. The same concept as Jupyter of interspersing code and text applies, and it supports several languages including R and Python.

As you're starting to see, there's a whole galaxy of tools available for coding in R and Python, more than can be covered in this book. Our focus has been on R scripts from RStudio and Python from Jupyter Notebooks because they are both relatively more beginner-friendly and common than other configurations. Once you get comfortable with these workflows, search online for the development environments mentioned here. As you continue to learn, you'll pick up even more ways to interact with these languages.

The steps for launching a Jupyter notebook vary for Windows and Mac computers. On Windows, open the Start menu, then search for and launch `Anaconda Prompt`. This is a command-line tool for working with your Anaconda distribution and yet another way to interact with Python code. For a further introduction to running Python from the command line with the experience of an Excel user in mind, check out Felix Zumstein's *Python for Excel* (O'Reilly). From inside the Anaconda prompt, enter `jupyter notebook` at the cursor and hit `Enter`. Your command will resemble the following, but with a different home directory path:

```
(base) C:\Users\User> jupyter notebook
```

On a Mac, open Launchpad, then search for and launch Terminal. This is the command-line interface that ships with Macs and can be used to communicate with Python. From inside the Terminal prompt, enter `jupyter notebook` at the cursor and hit `Enter`. Your command line will resemble the following, but with a different home directory path:

```
user@MacBook-Pro ~ % jupyter notebook
```

After doing this on either system, a couple of things will happen: first, an additional terminal-like window will launch on your computer. *Do not close this window.* This is what establishes the connection to the kernel. Additionally, the Jupyter notebook interface should automatically open in your default web browser. If it does not, the terminal-like window will include a link that you can paste into your browser. Figure 10-1 shows what you should see in your browser. Jupyter launches with a File-Explorer-like interface. You can now navigate to the folder in which you'd like to save your notebooks.

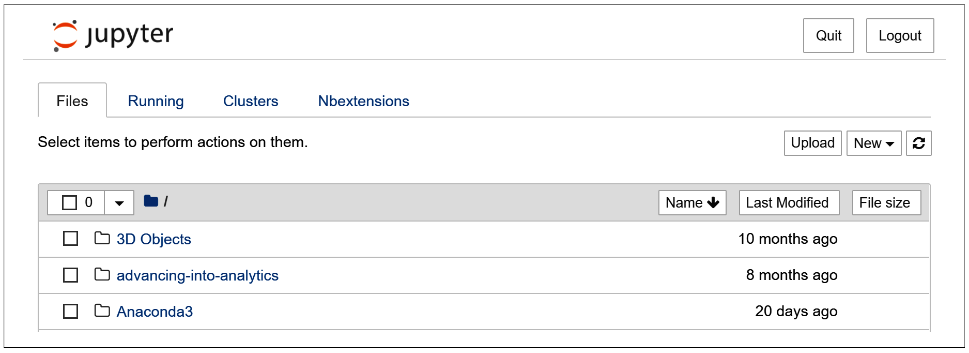


Figure 10-1. Jupyter landing page

To open a new notebook, head to the upper-right side of your browser window and select New → Notebook → Python 3. A new tab will open with a blank Jupyter notebook. Like RStudio, Jupyter provides far more features than we can cover in an introduction; we'll focus on the key pieces to get you started. The four main components of a Jupyter notebook are labeled in Figure 10-2; let's walk through each.

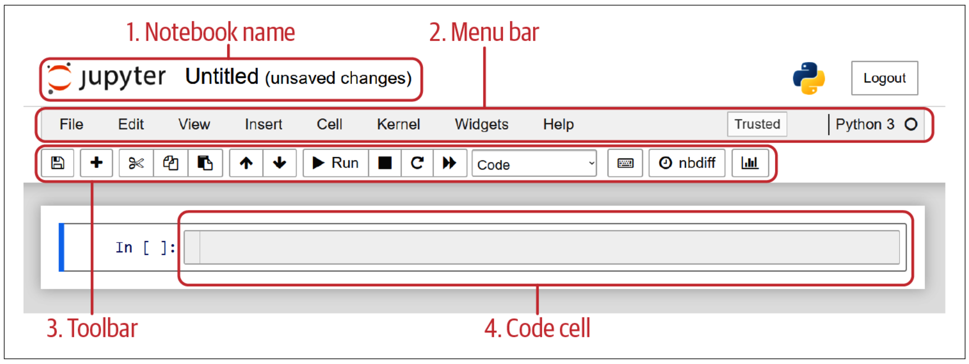


Figure 10-2. Elements of the Jupyter interface

First, the notebook name: this is the name of our *.ipynb* file. You can rename the notebook by clicking and writing over the current name.

Next, the menu bar. This contains different operations for working with your notebook. For example, under File you can open and close notebooks. Saving them isn't much of an issue, because Jupyter notebooks are autosaved every two minutes. If you ever need to convert your notebook to a *.py* Python script or other common file type, you can do so by going to File → Download as. There's also a Help section containing several guides and links to reference documentation. You can learn about Jupyter's many keyboard shortcuts from this menu.

Earlier, I mentioned that the *kernel* is how Jupyter interacts with Python under the hood. The *Kernel* option in the menu bar contains helpful operations for working with it. Computers being what they are, sometimes all that's needed to get your Python code working is to restart the kernel. You can do this by going to Kernel → Restart.

Immediately underneath the menu bar is the toolbar. This contains helpful icons for working with your notebook, which can be more convenient than navigating through the menu: for example, several icons here relate to interacting with the kernel.

You can also insert and relocate *cells* in your notebook, where you'll be spending most of your time in Jupyter. To get started, let's do one last thing with the toolbar: you'll find a drop-down menu there currently set to Code; change it to Markdown.

Now, navigate to your first code cell and type in the phrase, `Hello, Jupyter!` Head back to the toolbar and select the Run icon. A couple of things will happen. First, you'll see that your `Hello, Jupyter!` cell will render to look as it might in a word processing document. Next, you'll see that a new code cell is placed underneath your previous one, and that it's set for you to enter more information. Your notebook should resemble Figure 10-3.

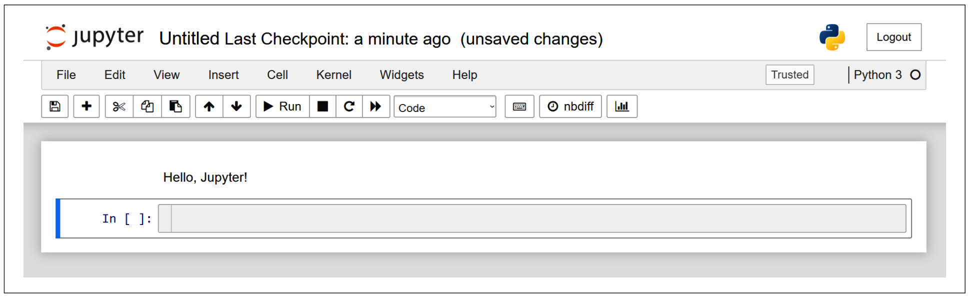


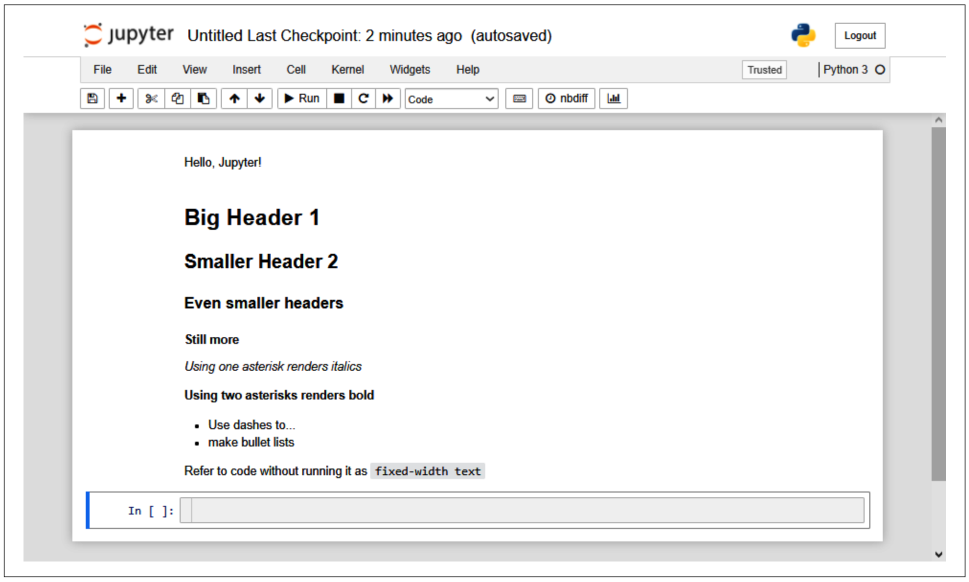
Figure 10-3. "Hello, Jupyter!"

Now, go back to the toolbar and again choose "Markdown" from the drop-down menu. As you're beginning to find out, Jupyter notebooks consist of modular cells that can be of different types. We'll focus on the two most common: Markdown and Code. Markdown is a plain-text markup language that uses regular keyboard characters to format text.

Insert the following text into your blank cell:

```
# Big Header 1
## Smaller Header 2
### Even smaller headers
#### Still more

*Using one asterisk renders italics*

**Using two asterisks renders bold**

- Use dashes to...
- Make bullet lists

Refer to code without running it as `fixed-width text`
```

Now run the cell: you can do this either from the toolbar or with the shortcut Alt + Enter for Windows, Option + Return for Mac. Your selection will render as in Figure 10-4.



Hello, Jupyter!

Big Header 1

Smaller Header 2

Even smaller headers

Still more

Using one asterisk renders italics

Using two asterisks renders bold

- Use dashes to...
- make bullet lists

Refer to code without running it as `fixed-width text`

Figure 10-4. Examples of Markdown formatting in Jupyter

To learn more about Markdown, return to the Help section of the menu bar. It's worth studying up to build elegant notebooks, which can include images, equations, and more. But in this book, we'll focus on the *code* block, as that's where executable Python goes. You should now be on your third code cell; you can leave this one as a Code format. Finally, we'll get coding in Python.

Python can be used as a fancy calculator, just like Excel and R. Table 10-1 lists some common arithmetic operators in Python.

Table 10-1. Common arithmetic operators in Python

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponent
%%	Modulo
//	Floor division

Enter in the following arithmetic, then run the cells:

```
In [1]: 1 + 1
Out[1]: 2

In [2]: 2 * 4
Out[2]: 8
```

As Jupyter code blocks are executed, they are given numbered labels of their inputs and outputs with `In []` and `Out []`, respectively.

Python also follows the order of operations; let's try running a few examples from within the same cell:

```
In [3]: # Multiplication before addition
        3 * 5 + 6
        2 / 2 - 7 # Division before subtraction
Out[3]: -6.0
```

By default, Jupyter notebooks will only return the output of the last-run code within a cell, so we'll break this into two. You can split a cell at the cursor on either Windows or Mac with the keyboard shortcut Ctrl + Shift + - (Minus):

```
In [4]:  # Multiplication before addition
         3 * 5 + 6

Out[4]: 21

In [5]:  2 / 2 - 7 # Division before subtraction

Out[5]: -6.0
```

And yes, Python also uses code comments. Similar to R, they start with a hash, and it's also preferable to keep them to separate lines.

Like Excel and R, Python includes many functions for both numbers and characters:

```
In [6]: abs(-100)

Out[6]: 100
```

```
In [7]: len('Hello, world!')

Out[7]: 13
```

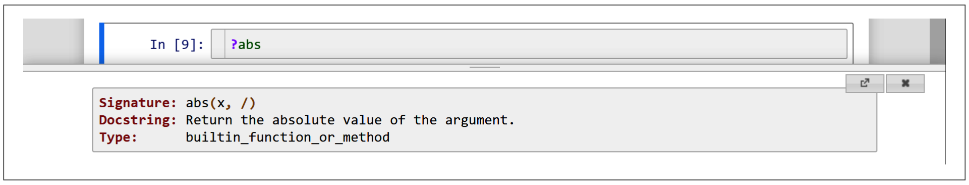
Unlike Excel, but like R, Python is case-sensitive. That means *only* `abs()` works, not `ABS()` or `Abs()`.

```
In [8]:  ABS(-100)

        ---------------------------------------------------------------------------
        NameError                                 Traceback (most recent call last)
        <ipython-input-20-a0f3f8a69d46> in <module>
        ----> 1 print(ABS(-100))
            2 print(Abs(-100))

        NameError: name 'ABS' is not defined
```

Python and Indentation

In Python, whitespace is more than a suggestion: it can be a *necessity* for code to run. That's because the language relies on proper indentation to compile and execute code blocks, or pieces of Python that are meant to be executed as a unit. You won't run into the problem in this book, but as you continue to experiment with other features of Python, such as how to write functions or loops, you'll see how prevalent and critical indentation is to the language.

Similar to R, you can use the ? operator to get information about functions, packages, and more. A window will open as in Figure 10-5, which you can then expand or open in a new window.



```
In [9]: ?abs

Signature: abs(x, /)
Docstring: Return the absolute value of the argument.
Type:      builtin_function_or_method
```

Figure 10-5. Launching documentation in Jupyter notebooks

Comparison operators mostly work the same in Python as in R and Excel; in Python, results are either returned as `True` or `False`.

```
In [10]: # Is 3 greater than 4?
         3 > 4

Out[10]: False
```

As with R, you check for whether two values are equal with ==; a single equals sign = is used to assign objects. We'll stick with = throughout to assign objects in Python.

```
In [11]:  # Assigning an object in Python
          my_first_object = abs(-100)
```

You may have noticed there was no `Out []` component of cell `11`. That's because we only *assigned* the object; we didn't print anything. Let's do that now:

```
In [12]: my_first_object

Out[12]: 100
```

Object names in Python must start with a letter or underscore, and the rest of the name can contain only letters, numbers, or underscores. There are also a few off-limit keywords. Again, you're left with broad license for naming objects in Python, but just because you *can* name an object `scooby_doo` doesn't mean you should.

Python and PEP 8

The Python Foundation uses Python Enhancement Proposals (PEPs) to announce changes or new features to the language. PEP 8 offers a style guide that is the universal standard for writing Python code. Among its many rules and guidelines are conventions for naming objects, adding comments, and more. You can read the full PEP 8 guide on the Python Foundation's website (*https://oreil.ly/KdmIf*).

Just like in R, our objects in Python can consist of different data types. Table 10-2 shows some basic Python types. Do you see the similarities and differences to R?

Table 10-2. Basic object types in Python

Data type	Example
String	`'Python'`, `'G. Mount'`, `'Hello, world!'`
Float	`6.2`, `4.13`, `3.1`
Integer	`3`, `-1`, `12`
Boolean	`True`, `False`

Let's assign some objects. We can find what type they are with the `type()` function:

```
In [13]:  my_string = 'Hello, world'
          type(my_string)

Out[13]: str

In [14]: # Double quotes work for strings, too
         my_other_string = "We're able to code Python!"
         type(my_other_string)

Out[14]: str
```

```
In [15]: my_float = 6.2
         type(my_float)

Out[15]: float


In [16]: my_integer = 3
         type(my_integer)

Out[16]: int

In [17]: my_bool = True
         type(my_bool)

Out[17]: bool
```

You've worked with objects in R, so you're probably not surprised that it's possible to use them as part of Python operations.

```
In [18]:  # Is my_float equal to 6.1?
          my_float == 6.1

Out[18]: False

In [19]:  # How many characters are in my_string?
          # (Same function as Excel)
          len(my_string)

Out[19]: 12
```

Closely related to functions in Python are *methods*. A method is affixed to an object with a period and does something to that object. For example, to capitalize all letters in a string object, we can use the `upper()` method:

```
In [20]:  my_string.upper()

Out[20]: 'HELLO, WORLD'
```

Functions and methods are both used to perform operations on objects, and we'll use both in this book. As you are probably hoping, Python, like R, can store multiple values in a single object. But before getting into that, let's consider how *modules* work in Python.

Modules in Python

Python was designed as a general-purpose programming language, so even some of the simplest functions for working with data aren't available out of the box. For example, we won't have luck finding a function to take the square root of a number:

```
In [21]:  sqrt(25)

          ---------------------------------------------------------
          NameError              Traceback (most recent call last)
          <ipython-input-18-1bf613b64533> in <module>
          ----> 1 sqrt(25)

          NameError: name 'sqrt' is not defined
```

This function *does* exist in Python. But to access it, we'll need to bring in a *module*, which is like a package in R. Several modules come installed with Python as part of the Python Standard Library; for example, the `math` module contains many mathematical functions, including `sqrt()`. We can call this module into our session with the `import` statement:

```
In [22]:  import math
```

Statements are instructions telling the interpreter what to do. We just told Python to, well, *import* the `math` module. The `sqrt()` function should now be available to us; give it a try:

```
In [23]:  sqrt(25)

          ---------------------------------------------------------
          NameError              Traceback (most recent call last)
          <ipython-input-18-1bf613b64533> in <module>
          ----> 1 sqrt(25)

          NameError: name 'sqrt' is not defined
```

Honestly, I'm not fibbing about a `sqrt()` function. The reason we're still getting errors is we need to explicitly tell Python *where* that function comes from. We can do that by prefixing the module name before the function, like so:

```
In [24]:  math.sqrt(25)

Out[24]: 5.0
```

The Standard Library is full of helpful modules. Then there are the thousands of third-party modules, bundled into *packages* and submitted to the Python Package Index. `pip` is the standard package-management system; it can be used to install from the Python Package Index as well as outside sources.

The Anaconda distribution has done much of the lifting for working with packages. First off, some of the most popular Python packages come preinstalled. Additionally, Anaconda includes features to ensure all packages on your machine are compatible. For this reason, it's preferred to install packages directly from Anaconda rather than from `pip`. Python package installation is generally done from the command line; you worked there earlier when you were in the Anaconda Prompt (Windows) or Terminal (Mac). However, we can execute command-line code from Jupyter by including an exclamation mark in front of it. Let's install `plotly`, a popular package for data visualization, from Anaconda. The statement to use is `conda install`:

```
In [25]:  !conda install plotly
```

Not all packages are available to download from Anaconda; in that case, we can install via `pip`. Let's do it for the `pyxlsb` package, which can be used to read binary `.xlsb` Excel files into Python:

```
In [26]:  !pip install pyxlsb
```

Although downloading packages right from Jupyter is convenient, it can be an unpleasant surprise for others to try running your notebook only to get hit with lengthy or unnecessary downloads. That's why it's common to comment out install commands, a convention I follow in the book repository.

If you're using Anaconda to run Python, it's best to install things first via `conda` and only then install via `pip` if the package is not available.

Upgrading Python, Anaconda, and Python packages

Table 10-3 lists several other helpful commands for maintaining your Python environment. You can also install and maintain Anaconda packages from a point-and-click interface using the Anaconda Navigator, which is installed with Anaconda Individual Edition. To get started, launch the application on your computer, then navigate to the Help menu to read the documentation for more.

Table 10-3. Helpful commands for maintaining Python packages

Command	Description
`conda update anaconda`	Updates Anaconda distribution
`conda update python`	Updates Python version
`conda update -- all`	Updates all possible packages downloaded via `conda`
`pip list -- outdated`	Lists all packages downloaded via `pip` that can be updated

Conclusion

In this chapter, you learned how to work with objects and packages in Python and got the hang of working with Jupyter notebooks.

Exercises

The following exercises provide additional practice and insight on these topics:

1. From a new Jupyter notebook, do the following:
   - Assign the sum of 1 and –4 as `a`.
   - Assign the absolute value of `a` as `b`.
   - Assign `b` minus 1 as `d`.
   - Is `d` greater than 2?
2. The Python Standard Library includes a module `random` containing a function `randint()`. This function works like `RANDBETWEEN()` in Excel; for example, `randint(1, 6)` will return an integer between 1 and 6. Use this function to find a random number between 0 and 36.
3. The Python Standard Library also includes a module called `this`. What happens when you import that module?
4. Download the `xlutils` package from Anaconda, then use the `?` operator to retrieve the available documentation.

I will again encourage you to begin using the language as soon as possible in your everyday work, even if at first it's just as a calculator. You can also try performing the same tasks in R *and* Python, then comparing and contrasting. If you learned by relating R to Excel, the same will work for relating Python to R.

CHAPTER 11

Data Structures in Python

In Chapter 10, you learned about simple Python object types like strings, integers, and Booleans. Now let's look at grouping multiple values together in what's called a *collection*. Python by default comes with several collection object types. We'll start this chapter with the *list*. We can put values into a list by separating each entry with commas and placing the results inside square brackets:

```
In [1]: my_list = [4, 1, 5, 2]
        my_list

Out[1]: [4, 1, 5, 2]
```

This object contains all integers, but itself is *not* an integer data type: it is a *list*.

```
In [2]: type(my_list)

Out[2]: list
```

In fact, we can include all different sorts of data inside a list…even other lists.

```
In [3]: my_nested_list = [1, 2, 3, ['Boo!', True]]
        type(my_nested_list)

Out[3]: list
```

Other Collection Types in Base Python

Python includes several other built-in collection object types besides the list, most notably the *dictionary*, along with still more in the Standard Library's `collections` module. Collection types vary by how they store values and can be indexed or modified.

As you're seeing, lists are quite versatile for storing data. But right now, we're really interested in working with something that could function like an Excel range or R vector, and then move into tabular data. Does a simple list fit the bill? Let's give it a whirl by trying to multiply `my_list` by two.

```
In [4]:  my_list * 2

Out[4]: [4, 1, 5, 2, 4, 1, 5, 2]
```

This is probably *not* what you are looking for: Python took you literally and, well, doubled your *list*, rather than the *numbers inside* your list. There are ways to get what we want here on our own: if you've worked with loops before, you could set one up here to multiply each element by two. If you've not worked with loops, that's fine too: the better option is to import a module that makes it easier to perform computations in Python. For that, we'll use `numpy`, which is included with Anaconda.

NumPy arrays

```
In [5]:  import numpy
```

As its name suggests, `numpy` is a module for numerical computing in Python and has been foundational to Python's popularity as an analytics tool. To learn more about `numpy`, visit the Help section of Jupyter's menu bar and select "NumPy reference." We'll focus for right now on the `numpy` *array*. This is a collection of data with all items of the same type and that can store data in up to any number, or *n* dimensions. We'll focus on a *one-dimensional* array and convert our first one from a list using the `array()` function:

```
In [6]:  my_array = numpy.array([4, 1, 5, 2])
         my_array

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

At first glance a `numpy` array looks a *lot* like a list; after all, we even created this one *from* a list. But we can see that it really is a different data type:

```
In [7]: type(my_list)

Out[7]: list

In [8]: type(my_array)

Out[8]: numpy.ndarray
```

Specifically, it's an `ndarray`, or *n*-dimensional array. Because it's a different data structure, it may behave differently with operations. For example, what happens when we multiply a `numpy` array?

```
In [9]: my_list * 2

Out[9]: [4, 1, 5, 2, 4, 1, 5, 2]


In [10]: my_array * 2

Out[10]: array([ 8,  2, 10,  4])
```

In many ways this behavior should remind you of an Excel range or an R vector. And indeed, like R vectors, numpy arrays will *coerce* data to be of the same type:

```
In [11]: my_coerced_array = numpy.array([1, 2, 3, 'Boo!'])
         my_coerced_array

Out[11]: array(['1', '2', '3', 'Boo!'], dtype='<U11')
```

Data Types in NumPy and Pandas

You'll notice that data types in numpy and later pandas work a bit differently than standard Python. These so-called dtypes are built to read and write data quickly and work with low-level programming languages like C or Fortran. Don't worry too much about the specific dtypes being used; focus on the general kind of data you're working with, such as floating point, string, or Boolean.

As you're seeing, numpy is a lifesaver for working with data in Python. Plan to import it *a lot*…which means typing it a lot. Fortunately, you can lighten the load with *aliasing*. We'll use the as keyword to give numpy its conventional alias, np:

```
In [12]:  import numpy as np
```

This gives the module a temporary, more manageable name. Now, each time we want to call in code from numpy during our Python session, we can refer to its alias.

```
In [13]: import numpy as np
         # numpy also has a sqrt() function:
         np.sqrt(my_array)

Out[13]: array([2.        , 1.        , 2.23606798, 1.41421356])
```

Remember that aliases are *temporary* to your Python session. If you restart your kernel or start a new notebook, the alias won't work anymore.

Indexing and Subsetting NumPy Arrays

Let's take a moment to explore how to pull individual items from a `numpy` array, which we can do by affixing its index number in square brackets directly next to the object name:

```
In [14]: # Get second element... right?
         my_array[2]

Out[14]: 5
```

For example, we just pulled the second element from our array… *or did we?* Let's revisit `my_array`; what is *really* showing in the second position?

```
In [15]: my_array

Out[15]: array([4, 1, 5, 2])
```

It appears that `1` is in the second position, and `5` is actually in the *third*. What explains this discrepancy? As it turns out, it's because Python counts things differently than you and I usually do.

As a warm-up to this strange concept, imagine being so excited to get your hands on a new dataset that you download it several times. That hastiness leaves you with a series of files named like this:

- *dataset.csv*
- *dataset (1).csv*
- *dataset (2).csv*
- *dataset (3).csv*

As humans, we tend to start counting things at one. But computers often start counting at *zero*. Multiple file downloads is one example: our second file is actually named `dataset (1)`, not `dataset (2)`. This is called *zero-based indexing*, and it happens *all over* in Python.

Zero- and One-Based Indexing

Computers often count from zero, but not all the time. In fact, Excel and R both implement *one-based* indexing, where the first element is considered to be in position one. Programmers can have strong opinions about which is a better design, but you should be comfortable working in both frameworks.

This is all to say that, to Python, indexing something with the number `1` returns the value in the *second* position, indexing with `2` returns the third, and so on.

```
In [16]: # *Now* let's get the second element
         my_array[1]

Out[16]: 1
```

It's also possible to subset a selection of consecutive values, called *slicing* in Python. Let's try finding the second through fourth elements. We already got the zero-based kicker out of the way; how hard could this be?

```
In [17]: # Get second through fourth elements... right?
         my_array[1:3]

Out[17]: array([1, 5])
```

But wait, there's more. In addition to being zero-indexed, slicing is *exclusive* of the ending element. That means we need to "add 1" to the second number to get our intended range:

```
In [18]: # *Now* get second through fourth elements
         my_array[1:4]

Out[18]: array([1, 5, 2])
```

There's much more you can do with slicing in Python, such as starting at the *end* of an object or selecting all elements from the start to a given position. For now, the important thing to remember is that *Python uses zero-based indexing*.

Two-dimensional `numpy` arrays can serve as a tabular Python data structure, but all elements must be of the same data type. This is rarely the case when we're analyzing data in a business context, so to meet this requirement we'll move to `pandas`.

Introducing Pandas DataFrames

Named after the *panel data* of econometrics, `pandas` is especially helpful for manipulating and analyzing tabular data. Like `numpy`, it comes installed with Anaconda. The typical alias is `pd`:

```
In [19]: import pandas as pd
```

The `pandas` module leverages `numpy` in its code base, and you will see some similarities between the two. `pandas` includes, among others, a one-dimensional data structure called a *Series*. But its most widely used structure is the two-dimensional *DataFrame* (sound familiar?). It's possible to create a DataFrame from other data types, including `numpy` arrays, using the `DataFrame` function:

```
In [20]: record_1 = np.array(['Jack', 72, False])
         record_2 = np.array(['Jill', 65, True])
         record_3 = np.array(['Billy', 68, False])
         record_4 = np.array(['Susie', 69, False])
         record_5 = np.array(['Johnny', 66, False])
         roster = pd.DataFrame(data = [record_1,
             record_2, record_3, record_4, record_5],
               columns = ['name', 'height', 'injury'])

         roster

Out[20]:
              name height injury
         0    Jack     72  False
         1    Jill     65   True
         2   Billy     68  False
         3   Susie     69  False
         4  Johnny     66  False
```

DataFrames generally include named *labels* for each column. There will also be an *index* running down the rows, which by default starts at (you guessed it) 0. This is a pretty small dataset to explore, so let's find something else. Unfortunately, Python does not include any DataFrames out of the gate, but we can find some with the `seaborn` package. `seaborn` also comes installed with Anaconda and is often aliased as `sns`. The `get_dataset_names()` function will return a list of DataFrames available to use:

```
In [21]: import seaborn as sns
         sns.get_dataset_names()

Out[21]:
        ['anagrams', 'anscombe', 'attention', 'brain_networks', 'car_crashes',
        'diamonds', 'dots', 'exercise', 'flights', 'fmri', 'gammas',
        'geyser', 'iris', 'mpg', 'penguins', 'planets', 'tips', 'titanic']
```

Does *iris* sound familiar? We can load it into our Python session with the `load_dataset()` function, and print the first five rows with the `head()` method.

```
In [22]: iris = sns.load_dataset('iris')
         iris.head()

Out[22]:
               sepal_length  sepal_width  petal_length  petal_width species
        0           5.1          3.5           1.4          0.2  setosa
        1           4.9          3.0           1.4          0.2  setosa
        2           4.7          3.2           1.3          0.2  setosa
        3           4.6          3.1           1.5          0.2  setosa
        4           5.0          3.6           1.4          0.2  setosa
```

Importing Data in Python

As with R, it's most common to read in data from external files, and we'll need to deal with directories to do so. The Python Standard Library includes the `os` module for working with file paths and directories:

```
In [23]: import os
```

For this next part, have your notebook saved in the main folder of the book repository. By default, Python sets the current working directory to wherever your active file is located, so we don't have to worry about changing the directory as we did in R. You can still check and change it with the `getcwd()` and `chdir()` functions from `os`, respectively.

Python follows the same general rules about relative and absolute file paths as R. Let's see if we can locate *test-file.csv* in the repository using the `isfile()` function, which is in the `path` submodule of `os`:

```
In [24]: os.path.isfile('test-file.csv')

Out[24]: True
```

Now we'd like to locate that file as contained in the *test-folder* subfolder.

```
In [25]: os.path.isfile('test-folder/test-file.csv')

Out[25]: True
```

Next, try putting a copy of this file in the folder one up from your current location. You should be able to locate it with this code:

```
In [26]:  os.path.isfile('../test-file.csv')

Out[26]: True
```

Like with R, you'll most commonly read data in from an external source to operate on it in Python, and this source can be nearly anything imaginable. `pandas` includes functions to read data from, among others, both *.xlsx* and *.csv* files into DataFrames. To demonstrate, we'll read in our reliable *star.xlsx* and *districts.csv* datasets from the book repository. The `read_excel()` function is used to read Excel workbooks:

```
In [27]: star = pd.read_excel('datasets/star/star.xlsx')
         star.head()

Out[27]:
   tmathssk  treadssk             classk  totexpk   sex freelunk   race
0       473       447        small.class        7  girl       no  white
1       536       450        small.class       21  girl       no  black
2       463       439  regular.with.aide        0   boy      yes  black
3       559       448            regular       16   boy       no  white
4       489       447        small.class        5   boy      yes  white

   schidkn
0       63
1       20
2       19
3       69
4       79
```

Similarly, we can use `pandas` to read in *.csv* files with the `read_csv()` function:

```
In [28]: districts = pd.read_csv('datasets/star/districts.csv')
         districts.head()

Out[28]:
             schidkn      school_name       county
          0        1          Rosalia  New Liberty
          1        2  Montgomeryville       Topton
          2        3             Davy     Wahpeton
          3        4         Steelton    Palestine
          4        6       Tolchester      Sattley
```

If you'd like to read in other Excel file types or specific ranges and worksheets, for example, check the `pandas` documentation.

Exploring a DataFrame

Let's continue to size up the *star* DataFrame. The `info()` method will tell us some important things, such as its dimensions and types of columns:

```
In [29]: star.info()

    <class 'pandas.core.frame.DataFrame'>
    RangeIndex: 5748 entries, 0 to 5747
    Data columns (total 8 columns):
    #   Column    Non-Null Count  Dtype
    ---  ------    --------------  -----
    0   tmathssk  5748 non-null   int64
    1   treadssk  5748 non-null   int64
    2   classk    5748 non-null   object
    3   totexpk   5748 non-null   int64
    4   sex       5748 non-null   object
    5   freelunk  5748 non-null   object
    6   race      5748 non-null   object
    7   schidkn   5748 non-null   int64
    dtypes: int64(4), object(4)
    memory usage: 359.4+ KB
```

We can retrieve descriptive statistics with the `describe()` method:

```
In [30]: star.describe()

Out[30]:
                tmathssk     treadssk      totexpk      schidkn
        count  5748.000000  5748.000000  5748.000000  5748.000000
        mean    485.648051   436.742345     9.307411    39.836639
        std      47.771531    31.772857     5.767700    22.957552
        min     320.000000   315.000000     0.000000     1.000000
        25%     454.000000   414.000000     5.000000    20.000000
        50%     484.000000   433.000000     9.000000    39.000000
```

```
75%     513.000000   453.000000    13.000000    60.000000
max     626.000000   627.000000    27.000000    80.000000
```

By default, pandas only includes descriptive statistics of numeric variables. We can override this with `include = 'all'`.

```
In [31]: star.describe(include = 'all')

Out[31]:
           tmathssk     treadssk            classk      totexpk   sex  \
count   5748.000000  5748.000000              5748  5748.000000  5748
unique          NaN          NaN                 3          NaN     2
top             NaN          NaN  regular.with.aide          NaN   boy
freq            NaN          NaN              2015          NaN  2954
mean     485.648051   436.742345               NaN     9.307411   NaN
std       47.771531    31.772857               NaN     5.767700   NaN
min      320.000000   315.000000               NaN     0.000000   NaN
25%      454.000000   414.000000               NaN     5.000000   NaN
50%      484.000000   433.000000               NaN     9.000000   NaN
75%      513.000000   453.000000               NaN    13.000000   NaN
max      626.000000   627.000000               NaN    27.000000   NaN

       freelunk   race      schidkn
count      5748   5748  5748.000000
unique        2      3          NaN
top          no  white          NaN
freq       2973   3869          NaN
mean        NaN    NaN    39.836639
std         NaN    NaN    22.957552
min         NaN    NaN     1.000000
25%         NaN    NaN    20.000000
50%         NaN    NaN    39.000000
75%         NaN    NaN    60.000000
max         NaN    NaN    80.000000
```

`NaN` is a special `pandas` value to indicate missing or unavailable data, such as the standard deviation of a categorical variable.

Indexing and Subsetting DataFrames

Let's return to the small *roster* DataFrame, accessing various elements by their row and column position. To index a DataFrame we can use the `iloc`, or *integer location*, method. The square bracket notation will look familiar to you, but this time we need to index by both row *and* column (again, both starting at zero). Let's demonstrate on the *roster* DataFrame we created earlier.

```
In [32]:  # First row, first column of DataFrame
          roster.iloc[0, 0]

Out[32]: 'Jack'
```

It's possible to employ slicing here as well to capture multiple rows and columns:

```
In [33]: # Second through fourth rows, first through third columns
         roster.iloc[1:4, 0:3]

Out[33]:
     name height injury
 1   Jill     65   True
 2  Billy     68  False
 3  Susie     69  False
```

To index an entire column by name, we can use the related `loc` method. We'll leave a blank slice in the first index position to capture all rows, then name the column of interest:

```
In [34]:  # Select all rows in the name column
          roster.loc[:, 'name']

Out[34]:
        0      Jack
        1      Jill
        2     Billy
        3     Susie
        4    Johnny
        Name: name, dtype: object
```

Writing DataFrames

`pandas` also includes functions to write DataFrames to both *.csv* files and *.xlsx* workbooks with the `write_csv()` and `write_xlsx()` methods, respectively:

```
In [35]: roster.to_csv('output/roster-output-python.csv')
         roster.to_excel('output/roster-output-python.xlsx')
```

Conclusion

In a short time, you were able to progress all the way from single-element objects, to lists, to `numpy` arrays, then finally to `pandas` DataFrames. I hope you were able to see the evolution and linkage between these data structures while appreciating the added benefits of the packages introduced. The following chapters on Python will rely heavily on `pandas`, but you've seen here that `pandas` itself relies on `numpy` and the basic rules of Python, such as zero-based indexing.

Exercises

In this chapter, you learned how to work with a few different data structures and collection types in Python. The following exercises provide additional practice and insight on these topics:

1. Slice the following array so that you are left with the third through fifth elements.

   ```
   practice_array = ['I', 'am', 'having', 'fun', 'with', 'Python']
   ```

2. Load the `tips` DataFrame from `seaborn`.
   - Print some information about this DataFrame, such as the number of observations and each column's type.
   - Print the descriptive statistics for this DataFrame.
3. The book repository (*https://oreil.ly/RKmg0*) includes an *ais.xlsx* file in the *ais* subfolder of the *datasets* folder. Read it into Python as a DataFrame.
   - Print the first few rows of this DataFrame.
   - Write just the *sport* column of this DataFrame back to Excel as *sport.xlsx*.

CHAPTER 12

Data Manipulation and Visualization in Python

In Chapter 8 you learned how to manipulate and visualize data, with heavy help from the `tidyverse` suite of packages. Here we'll demonstrate similar techniques on the same *star* dataset, this time in Python. In particular, we'll use `pandas` and `seaborn` to manipulate and visualize data, respectively. This isn't a comprehensive guide to what these modules, or Python, can do with data analysis. Instead, it's enough to get you exploring on your own.

As much as possible, I'll mirror the steps and perform the same operations that we did in Chapter 8. Because of this familiarity, I'll focus less on the whys of manipulating and visualizing data than I will on hows of doing it in Python. Let's load the necessary modules and get started with *star*. The third module, `matplotlib`, is new for you and will be used to complement our work in `seaborn`. It comes installed with Anaconda. Specifically, we'll be using the `pyplot` submodule, aliasing it as `plt`.

```
In [1]:  import pandas as pd
         import seaborn as sns
         import matplotlib.pyplot as plt

         star = pd.read_excel('datasets/star/star.xlsx')
         star.head()
Out[1]:
   tmathssk  treadssk             classk  totexpk   sex freelunk   race  \
0       473       447        small.class        7  girl       no  white
1       536       450        small.class       21  girl       no  black
2       463       439  regular.with.aide        0   boy      yes  black
3       559       448            regular       16   boy       no  white
4       489       447        small.class        5   boy      yes  white

   schidkn
```

```
0       63
1       20
2       19
3       69
4       79
```

Column-Wise Operations

In Chapter 11 you learned that `pandas` will attempt to convert one-dimensional data structures into Series. This seemingly trivial point will be quite important when selecting columns. Let's take a look at an example: say we *just* wanted to keep the *tmathssk* column from our DataFrame. We could do so using the familiar single-bracket notation, but this technically results in a Series, not a DataFrame:

```
In [2]:  math_scores = star['tmathssk']
         type(math_scores)

Out[2]: pandas.core.series.Series
```

It's probably better to keep this as a DataFrame if we aren't positive that we want *math_scores* to stay as a one-dimensional structure. To do so, we can use two sets of brackets instead of one:

```
In [3]: math_scores = star[['tmathssk']]
        type(math_scores)

Out[3]: pandas.core.frame.DataFrame
```

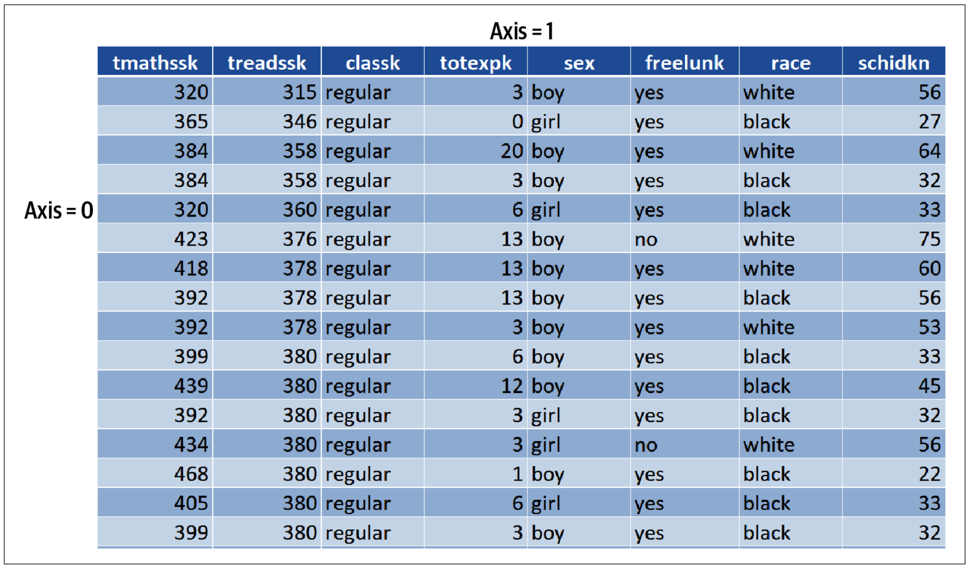
Following this pattern, we can keep only the desired columns in *star*. I'll use the `col umns` attribute to confirm.

```
In [4]:  star = star[['tmathssk','treadssk','classk','totexpk','schidkn']]
         star.columns

Out[4]: Index(['tmathssk', 'treadssk', 'classk',
             'totexpk', 'schidkn'], dtype='object')
```

Object-Oriented Programming in Python

So far you've seen methods and functions in Python. These are things that objects can *do*. Attributes, on the other hand, represent some *state* of an object itself. These are affixed to an object's name with a period; unlike methods, no parentheses are used. Attributes, functions, and methods are all elements of *object-oriented programming* (OOP), a paradigm meant to structure work into simple and reusable pieces of code. To learn more about how OOP works in Python, check out *Python in a Nutshell*, 3rd edition by Alex Martelli et al. (O'Reilly).

To drop specific columns, use the `drop()` method. `drop()` can be used to drop columns *or* rows, so we'll need to specify which by using the `axis` argument. In `pandas`, rows are axis `0` and columns axis `1`, as Figure 12-1 demonstrates.


Axis = 1
Axis = 0

tmathssk	treadssk	classk	totexpk	sex	freelunk	race	schidkn
320	315	regular	3	boy	yes	white	56
365	346	regular	0	girl	yes	black	27
384	358	regular	20	boy	yes	white	64
384	358	regular	3	boy	yes	black	32
320	360	regular	6	girl	yes	black	33
423	376	regular	13	boy	no	white	75
418	378	regular	13	boy	yes	white	60
392	378	regular	13	boy	yes	black	56
392	378	regular	3	boy	yes	white	53
399	380	regular	6	boy	yes	black	33
439	380	regular	12	boy	yes	black	45
392	380	regular	3	girl	yes	black	32
434	380	regular	3	girl	no	white	56
468	380	regular	1	boy	yes	black	22
405	380	regular	6	girl	yes	black	33
399	380	regular	3	boy	yes	black	32

Figure 12-1. Axes of a pandas DataFrame

Here's how to drop the *schidkn* column:

```
In [5]: star = star.drop('schidkn', axis=1)
        star.columns

Out[5]: Index(['tmathssk', 'treadssk',
              'classk', 'totexpk'], dtype='object')
```

Let's now look at deriving new columns of a DataFrame. We can do that using bracket notation—this time, I *do* want the result to be a Series, as each column of a DataFrame is actually a Series (just as each column of an R data frame is actually a vector). Here I'll calculate combined math and reading scores:

```
In [6]: star['new_column'] = star['tmathssk'] + star['treadssk']
        star.head()

Out[6]:
   tmathssk  treadssk             classk  totexpk  new_column
0       473       447        small.class        7         920
1       536       450        small.class       21         986
2       463       439  regular.with.aide        0         902
3       559       448            regular       16        1007
4       489       447        small.class        5         936
```

Again, *new_column* isn't a terribly descriptive variable name. Let's fix that with the `rename()` function. We'll use the `columns` argument and pass data to it in a format you're likely unfamiliar with:

```
In [7]: star = star.rename(columns = {'new_column':'ttl_score'})
        star.columns

Out[7]: Index(['tmathssk', 'treadssk', 'classk', 'totexpk', 'ttl_score'],
           dtype='object')
```

The curly bracket notation used in the last example is a Python *dictionary*. Dictionaries are collections of *key-value* pairs, with the key and value of each element separated by a colon. This is a core Python data structure and one to check out as you continue learning the language.

Row-Wise Operations

Now let's move to common operations by row. We'll start with sorting, which can be done in `pandas` with the `sort_values()` method. We'll pass a list of columns we want to sort by in their respective order to the `by` argument:

```
In [8]: star.sort_values(by=['classk', 'tmathssk']).head()

Out[8]:
     tmathssk  treadssk   classk  totexpk  ttl_score
309       320       360  regular        6        680
1470      320       315  regular        3        635
2326      339       388  regular        6        727
2820      354       398  regular        6        752
4925      354       391  regular        8        745
```

By default, all columns are sorted ascendingly. To modify that, we can include another argument, `ascending`, which will contain a list of `True`/`False` flags. Let's sort *star* by class size (*classk*) ascending and math score (*treadssk*) descending. Because we're not assigning this output back to *star*, this sorting is not permanent to the dataset.

```
In [9]: # Sort by class size ascending and math score descending
        star.sort_values(by=['classk', 'tmathssk'],
         ascending=[True, False]).head()

Out[9]:
      tmathssk  treadssk   classk  totexpk  ttl_score
724        626       474  regular       15       1100
1466       626       554  regular       11       1180
1634       626       580  regular       15       1206
2476       626       538  regular       20       1164
2495       626       522  regular        7       1148
```

To filter a DataFrame, we'll first use conditional logic to create a Series of `True`/`False` flags indicating whether each row meets some criteria. We'll then keep only the rows in the DataFrame where records in the Series are flagged as `True`. For example, let's keep only the records where `classk` is equal to `small.class`.

```
In [10]: small_class = star['classk'] == 'small.class'
         small_class.head()

Out[10]:
        0     True
        1     True
        2    False
        3    False
        4     True
        Name: classk, dtype: bool
```

We can now filter by this resulting Series by using brackets. We can confirm the number of rows and columns in our new DataFrame with the `shape` attribute:

```
In [11]: star_filtered = star[small_class]
         star_filtered.shape

Out[11]: (1733, 5)
```

`star_filtered` will contain fewer rows than *star*, but the same number of columns:

```
In [12]: star.shape

Out[12]: (5748, 5)
```

Let's try one more: we'll find the records where `treadssk` is at least `500`:

```
In [13]: star_filtered = star[star['treadssk'] >= 500]
         star_filtered.shape

Out[13]: (233, 5)
```

It's also possible to filter by multiple conditions using and/or statements. Just like in R, `&` and `|` indicate "and" and "or" in Python, respectively. Let's pass both of the previous criteria into one statement by placing each in parentheses, connecting them with `&`:

```
In [14]: # Find all records with reading score at least 500 and in small class
         star_filtered = star[(star['treadssk'] >= 500) &
                    (star['classk'] == 'small.class')]
         star_filtered.shape

Out[14]: (84, 5)
```

Aggregating and Joining Data

To group observations in a DataFrame, we'll use the `groupby()` method. If we print `star_grouped`, you'll see it's a `DataFrameGroupBy` object:

```
In [15]: star_grouped = star.groupby('classk')
         star_grouped

Out[15]: <pandas.core.groupby.generic.DataFrameGroupBy
            object at 0x000001EFD8DFF388>
```

We can now choose other fields to aggregate this grouped DataFrame by. Table 12-1 lists some common aggregation methods.

Table 12-1. Helpful aggregation functions in `pandas`

Method	Aggregation type
`sum()`	Sum
`count()`	Count values
`mean()`	Average
`max()`	Highest value
`min()`	Lowest value
`std()`	Standard deviation

The following gives us the average math score for each class size:

```
In [16]: star_grouped[['tmathssk']].mean()

Out[16]:
                       tmathssk
    classk
    regular            483.261000
    regular.with.aide  483.009926
    small.class        491.470283
```

Now we'll find the highest total score for each year of teacher experience. Because this would return quite a few rows, I will include the `head()` method to get just a few. This practice of adding multiple methods to the same command is called method *chaining*:

```
In [17]: star.groupby('totexpk')[['ttl_score']].max().head()

Out[17]:
                ttl_score
        totexpk
        0             1171
        1             1133
        2             1091
        3             1203
        4             1229
```

Chapter 8 reviewed the similarities and differences between Excel's `VLOOKUP()` and a left outer join. I'll read in a fresh copy of *star* as well as *districts*; let's use `pandas` to join these datasets. We'll use the `merge()` method to "look up" data from *school-districts* into *star*. By setting the `how` argument to `left`, we'll specify a left outer join, which again is the join type most similar to `VLOOKUP()`:

```
In [18]: star = pd.read_excel('datasets/star/star.xlsx')
         districts = pd.read_csv('datasets/star/districts.csv')
         star.merge(districts, how='left').head()

Out[18]:
   tmathssk  treadssk             classk  totexpk   sex freelunk   race  \
0       473       447        small.class        7  girl       no  white
1       536       450        small.class       21  girl       no  black
2       463       439  regular.with.aide        0   boy      yes  black
3       559       448            regular       16   boy       no  white
4       489       447        small.class        5   boy      yes  white

   schidkn    school_name          county
0       63     Ridgeville     New Liberty
1       20  South Heights         Selmont
2       19      Bunnlevel         Sattley
3       69          Hokah      Gallipolis
4       79   Lake Mathews  Sugar Mountain
```

Python, like R, is quite intuitive about joining data: it knew by default to merge on *schidkn* and brought in both *school_name* and *county*.

Reshaping Data

Let's take a look at widening and lengthening a dataset in Python, again using `pandas`. To start, we can use the `melt()` function to combine *tmathssk* and *treadssk* into one column. To do this, I'll specify the DataFrame to manipulate with the `frame` argument, which variable to use as a unique identifier with `id_vars`, and which variables to melt into a single column with `value_vars`. I'll also specify what to name the resulting value and label variables with `value_name` and `var_name`, respectively:

```
In [19]: star_pivot = pd.melt(frame=star, id_vars = 'schidkn',
            value_vars=['tmathssk', 'treadssk'], value_name='score',
            var_name='test_type')
         star_pivot.head()

Out[19]:
            schidkn test_type  score
         0       63  tmathssk    473
         1       20  tmathssk    536
         2       19  tmathssk    463
         3       69  tmathssk    559
         4       79  tmathssk    489
```

How about renaming *tmathssk* and *treadssk* as *math* and *reading*, respectively? To do this, I'll use a Python dictionary to set up an object called `mapping`, which serves as something like a "lookup table" to recode the values. I'll pass this into the `map()` method which will recode *test_type*. I'll also use the `unique()` method to confirm that only *math* and *reading* are now found in *test_type*:

```
In [20]: # Rename records in `test_type`
         mapping = {'tmathssk':'math','treadssk':'reading'}
         star_pivot['test_type'] = star_pivot['test_type'].map(mapping)

         # Find unique values in test_type
         star_pivot['test_type'].unique()

Out[20]: array(['math', 'reading'], dtype=object)
```

To widen *star_pivot* back into separate *math* and *reading* columns, I'll use the `pivot_table()` method. First I'll specify what variable to index by with the `index` argument, then which variables contain the labels and values to pivot from with the `columns` and `values` arguments, respectively.

It's possible in `pandas` to set unique index columns; by default, `pivot_table()` will set whatever variables you've included in the `index` argument there. To override this, I'll use the `reset_index()` method. To learn more about custom indexing in `pandas`, along with countless other data manipulation and analysis techniques we couldn't cover here, check out *Python for Data Analysis*, 2nd edition by Wes McKinney (O'Reilly).

```
In [21]: star_pivot.pivot_table(index='schidkn',
          columns='test_type', values='score').reset_index()

Out[21]:
         test_type  schidkn        math     reading
         0                1  492.272727  443.848485
         1                2  450.576923  407.153846
         2                3  491.452632  441.000000
         3                4  467.689655  421.620690
         4                5  460.084746  427.593220
         ..             ...         ...         ...
         74              75  504.329268  440.036585
         75              76  490.260417  431.666667
         76              78  468.457627  417.983051
         77              79  490.500000  434.451613
         78              80  490.037037  442.537037

         [79 rows x 3 columns]
```

Data Visualization

Let's now briefly touch on data visualization in Python, specifically using the `seaborn` package. `seaborn` works especially well for statistical analysis and with `pandas` DataFrames, so it's a great choice here. Similarly to how `pandas` is built on top of `numpy`, `seaborn` leverages features of another popular Python plotting package, `matplotlib`.

`seaborn` includes many functions to build different plot types. We'll modify the arguments within these functions to specify which dataset to plot, which variables go along the x- and y-axes, which colors to use, and so on. Let's get started by visualizing the count of observations for each level of *classk*, which we can do with the `countplot()` function.

Our dataset is *star*, which we'll specify with the `data` argument. To place our levels of *classk* along the x-axis we'll use the `x` argument. This results in the countplot shown in Figure 12-2:

```
In [22]: sns.countplot(x='classk', data=star)
```

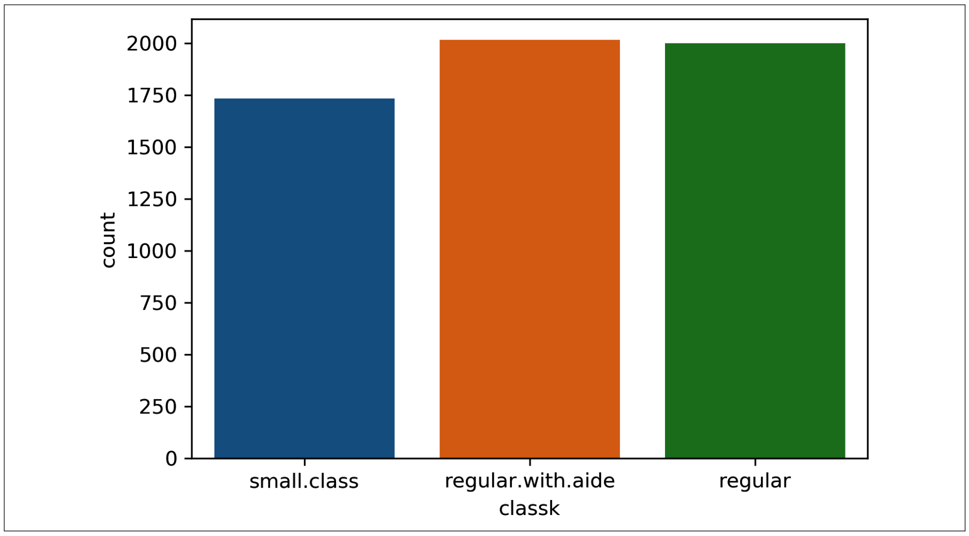


Figure 12-2. Countplot

Now for a histogram of *treadssk*, we'll use the `displot()` function. Again, we'll specify `x` and `data`. Figure 12-3 shows the result:

```
In [23]: sns.displot(x='treadssk', data=star)
```

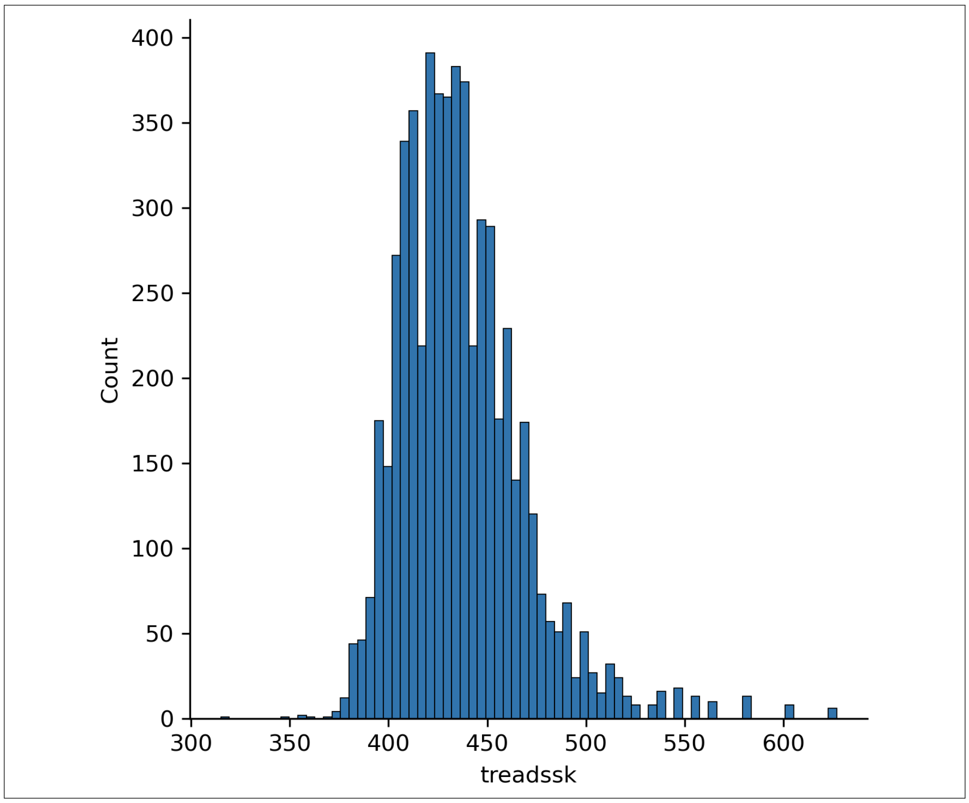


Figure 12-3. Histogram

`seaborn` functions include many optional arguments to customize a plot's appearance. For example, let's change the number of bins to 25 and the plot color to pink. This results in Figure 12-4:

```
In [24]: sns.displot(x='treadssk', data=star, bins=25, color='pink')
```

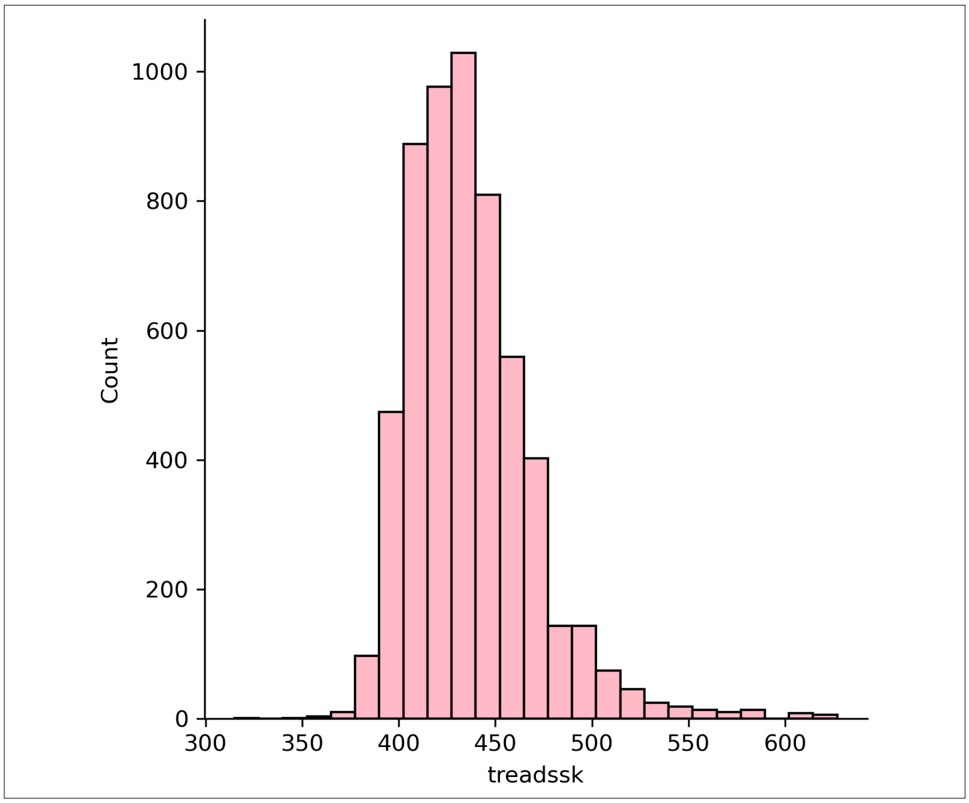


Figure 12-4. Custom histogram

To make a boxplot, use the `boxplot()` function as in Figure 12-5:

```
In [25]: sns.boxplot(x='treadssk', data=star)
```

In any of these cases so far, we could've "flipped" the plot by instead including the variable of interest in the `y` argument. Let's try it with our boxplot, and we'll get what's shown in Figure 12-6 as output:

```
In [26]: sns.boxplot(y='treadssk', data=star)
```

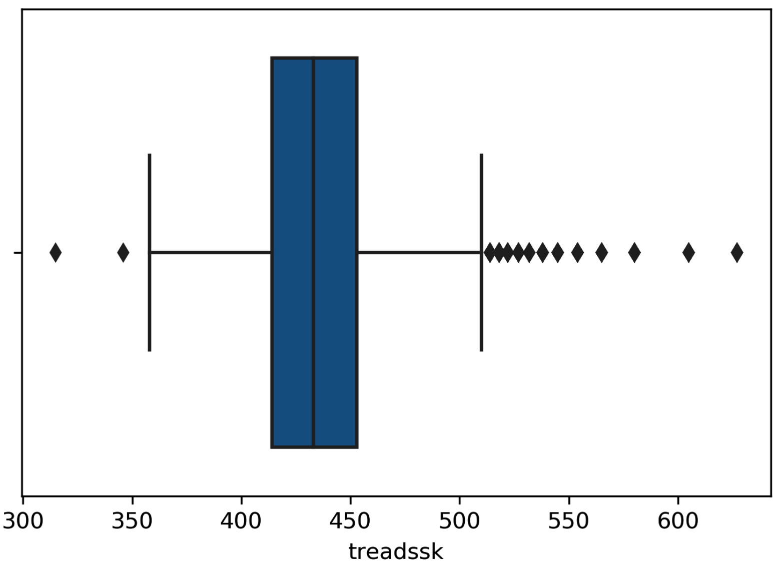


Figure 12-5. Boxplot

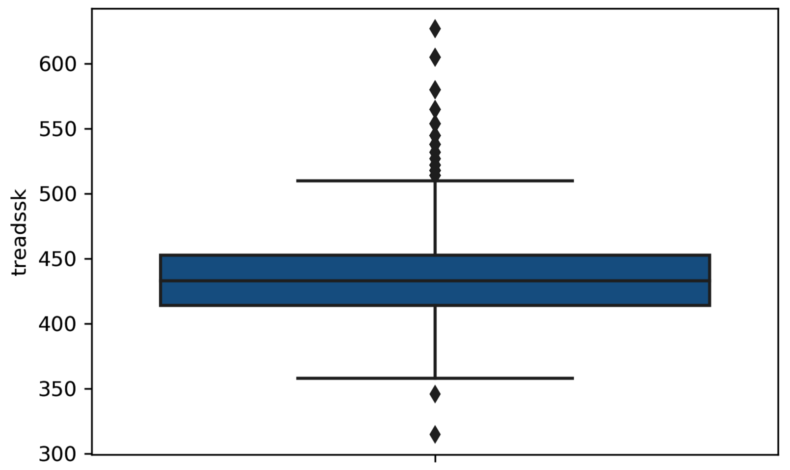


Figure 12-6. "Flipped" boxplot

To make a boxplot for each level of class size, we'll include an additional argument to plot *classk* along the x-axis, giving us the boxplot by group depicted in Figure 12-7:

```
In [27]: sns.boxplot(x='classk', y='treadssk', data=star)
```

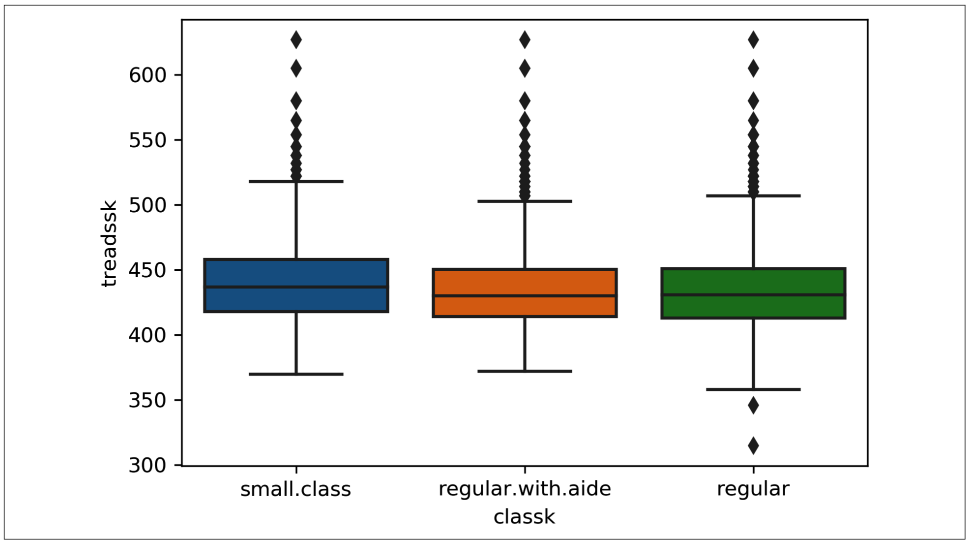


Figure 12-7. Boxplot by group

Now let's use the `scatterplot()` function to plot the relationship of *tmathssk* on the x-axis and *treadssk* on the y. Figure 12-8 is the result:

```
In [28]: sns.scatterplot(x='tmathssk', y='treadssk', data=star)
```

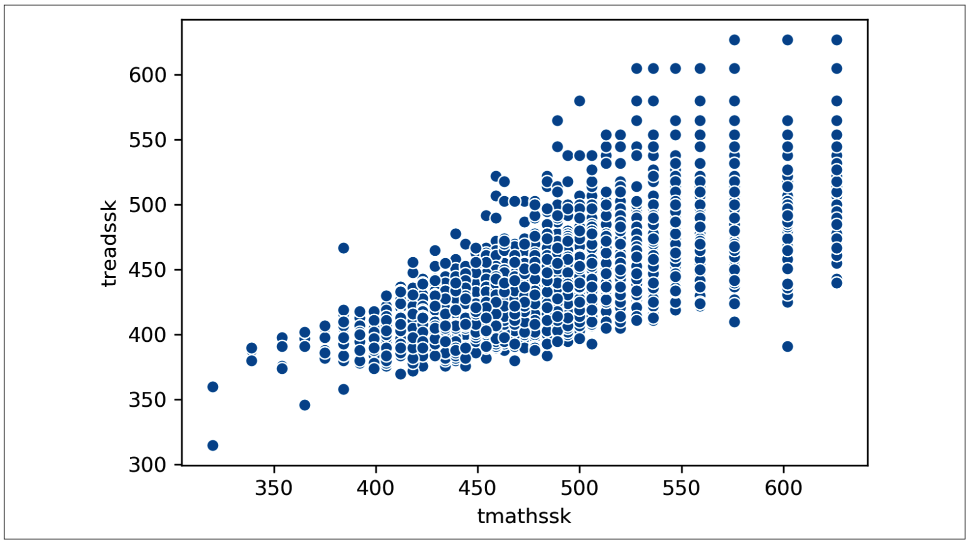


Figure 12-8. Scatterplot

Let's say we wanted to share this plot with an outside audience, who may not be familiar with what *treadssk* and *tmathssk* are. We can add more helpful labels to this chart by borrowing features from `matplotlib.pyplot`. We'll run the same `scatterplot()` function as before, but this time we'll also call in functions from `pyplot` to add custom x- and y-axis labels, as well as a chart title. This results in Figure 12-9:

```
In [29]: sns.scatterplot(x='tmathssk', y='treadssk', data=star)
         plt.xlabel('Math score')
         plt.ylabel('Reading score')
         plt.title('Math score versus reading score')
```

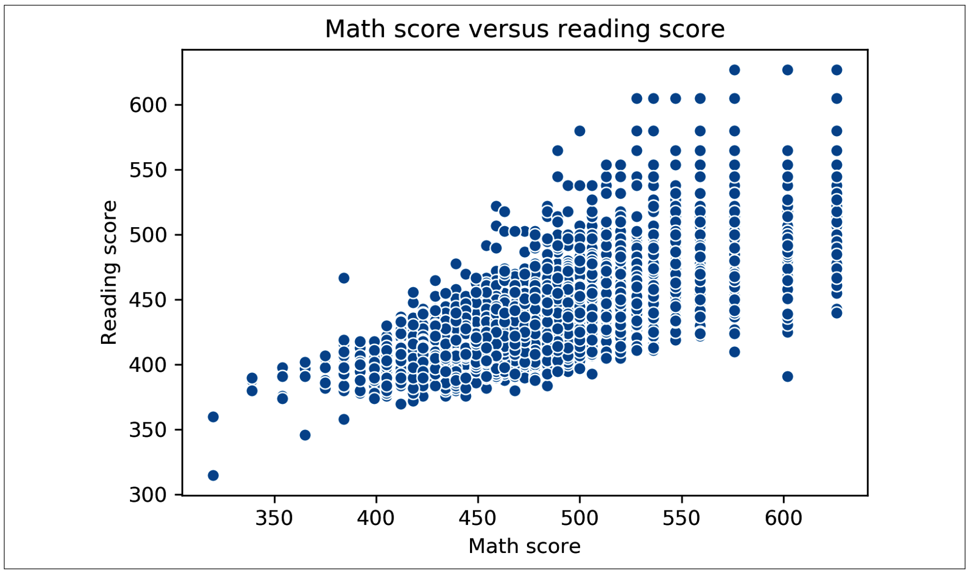


Figure 12-9. Scatterplot with custom axis labels and title

`seaborn` includes many more features for building visually appealing data visualizations. To learn more, check out the official documentation (*https://oreil.ly/2joMU*).

Conclusion

There's so much more that `pandas` and `seaborn` can do, but this is enough to get you started with the true task at hand: to explore and test relationships in data. That will be the focus of Chapter 13.

Exercises

The book repository (*https://oreil.ly/hFEOG*) has two files in the *census* subfolder of *datasets*, *census.csv* and *census-divisions.csv*. Read these into Python and do the following:

1. Sort the data by region ascending, division ascending and population descending. (You will need to combine datasets to do this.) Write the results to an Excel worksheet.
2. Drop the postal code field from your merged dataset.
3. Create a new column, *density*, that is a calculation of population divided by land area.
4. Visualize the relationship between land area and population for all observations in 2015.
5. Find the total population for each region in 2015.
6. Create a table containing state names and populations, with the population for each year 2010–2015 kept in an individual column.

CHAPTER 13
Capstone: Python for Data Analytics

At the end of Chapter 8 you extended what you learned about R to explore and test relationships in the *mpg* dataset. We'll do the same in this chapter, using Python. We've conducted the same work in Excel and R, so I'll focus less on the whys of our analysis in favor of the hows of doing it in Python.

To get started, let's call in all the necessary modules. Some of these are new: from `scipy`, we'll import the `stats` submodule. To do this, we'll use the `from` keyword to tell Python what module to look for, then the usual `import` keyword to choose a submodule. As the name suggests, we'll use the `stats` submodule of `scipy` to conduct our statistical analysis. We'll also be using a new package called `sklearn`, or *scikit-learn*, to validate our model on a train/test split. This package has become a dominant resource for machine learning and also comes installed with Anaconda.

```
In [1]: import pandas as pd
        import seaborn as sns
        import matplotlib.pyplot as plt
        from scipy import stats
        from sklearn import linear_model
        from sklearn import model_selection
        from sklearn import metrics
```

With the `usecols` argument of `read_csv()` we can specify which columns to read into the DataFrame:

```
In [2]: mpg = pd.read_csv('datasets/mpg/mpg.csv',usecols=
           ['mpg','weight','horsepower','origin','cylinders'])
        mpg.head()

Out[2]:
     mpg  cylinders  horsepower  weight origin
 0  18.0          8         130    3504    USA
 1  15.0          8         165    3693    USA
```

```
  2  18.0          8         150    3436    USA
  3  16.0          8         150    3433    USA
  4  17.0          8         140    3449    USA
```

Exploratory Data Analysis

Let's start with the descriptive statistics:

```
In[3]: mpg.describe()

Out[3]:
              mpg   cylinders  horsepower       weight
count  392.000000  392.000000  392.000000   392.000000
mean    23.445918    5.471939  104.469388  2977.584184
std      7.805007    1.705783   38.491160   849.402560
min      9.000000    3.000000   46.000000  1613.000000
25%     17.000000    4.000000   75.000000  2225.250000
50%     22.750000    4.000000   93.500000  2803.500000
75%     29.000000    8.000000  126.000000  3614.750000
max     46.600000    8.000000  230.000000  5140.000000
```

Because *origin* is a categorical variable, by default it doesn't show up as part of `describe()`. Let's explore this variable instead with a frequency table. This can be done in `pandas` with the `crosstab()` function. First, we'll specify what data to place on the index: *origin*. We'll get a count for each level by setting the `columns` argument to `count`:

```
In [4]: pd.crosstab(index=mpg['origin'], columns='count')

Out[4]:
col_0   count
origin
Asia       79
Europe     68
USA       245
```

To make a two-way frequency table, we can instead set `columns` to another categorical variable, such as `cylinders`:

```
In [5]: pd.crosstab(index=mpg['origin'], columns=mpg['cylinders'])

Out[5]:
cylinders  3   4  5   6    8
origin
Asia       4  69  0   6    0
Europe     0  61  3   4    0
USA        0  69  0  73  103
```

Next, let's retrieve descriptive statistics for *mpg* by each level of *origin*. I'll do this by chaining together two methods, then subsetting the results:

```
In[6]: mpg.groupby('origin').describe()['mpg']

Out[6]:
        count       mean       std   min    25%   50%     75%   max
origin
Asia     79.0  30.450633  6.090048  18.0  25.70  31.6  34.050  46.6
Europe   68.0  27.602941  6.580182  16.2  23.75  26.0  30.125  44.3
USA     245.0  20.033469  6.440384   9.0  15.00  18.5  24.000  39.0
```

We can also visualize the overall distribution of *mpg*, as in Figure 13-1:

```
In[7]: sns.displot(data=mpg, x='mpg')
```

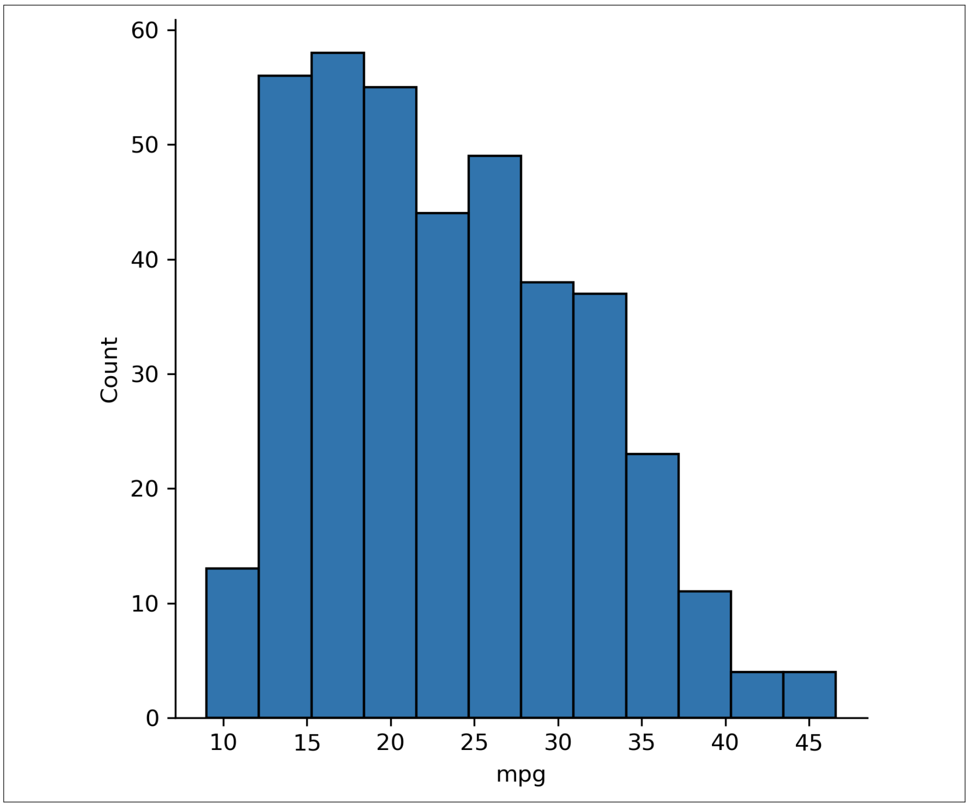


Figure 13-1. Histogram of mpg

Now let's make a boxplot as in Figure 13-2 comparing the distribution of *mpg* across each level of *origin*:

```
In[8]: sns.boxplot(x='origin', y='mpg', data=mpg, color='pink')
```

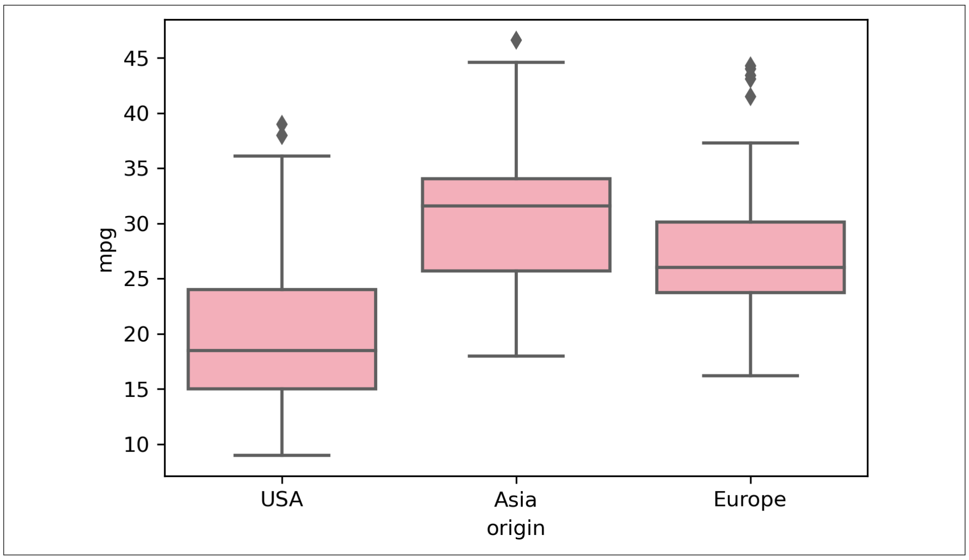


Figure 13-2. Boxplot of mpg by origin

Alternatively, we can set the `col` argument of `displot()` to `origin` to create faceted histograms, such as in Figure 13-3:

```
In[9]: sns.displot(data=mpg, x="mpg", col="origin")
```

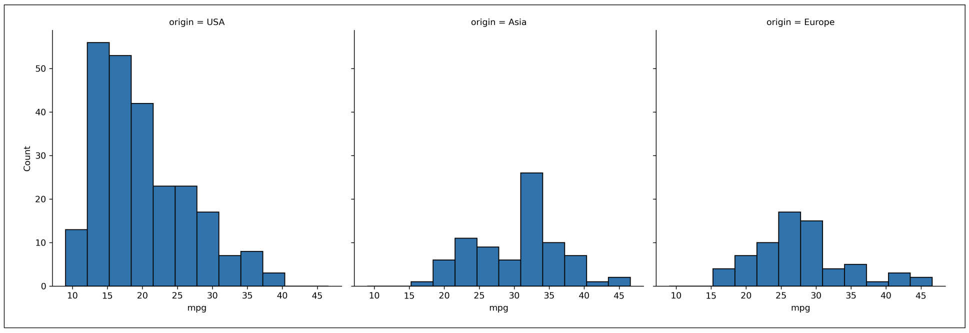


Figure 13-3. Faceted histogram of mpg by origin

Hypothesis Testing

Let's again test for a difference in mileage between American and European cars. For ease of analysis, we'll split the observations in each group into their own DataFrames.

```
In[10]: usa_cars = mpg[mpg['origin']=='USA']
        europe_cars = mpg[mpg['origin']=='Europe']
```

Independent Samples T-test

We can now use the `ttest_ind()` function from `scipy.stats` to conduct the t-test. This function expects two `numpy` arrays as arguments; `pandas` Series also work:

```
In[11]: stats.ttest_ind(usa_cars['mpg'], europe_cars['mpg'])

Out[11]: Ttest_indResult(statistic=-8.534455914399228,
             pvalue=6.306531719750568e-16)
```

Unfortunately, the output here is rather scarce: while it does include the p-value, it doesn't include the confidence interval. To run a t-test with more output, check out the `researchpy` module.

Let's move on to analyzing our continuous variables. We'll start with a correlation matrix. We can use the `corr()` method from `pandas`, including only the relevant variables:

```
In[12]: mpg[['mpg','horsepower','weight']].corr()

Out[12]:
                 mpg  horsepower    weight
mpg         1.000000   -0.778427 -0.832244
horsepower -0.778427    1.000000  0.864538
weight     -0.832244    0.864538  1.000000
```

Next, let's visualize the relationship between *weight* and *mpg* with a scatterplot as shown in Figure 13-4:

```
In[13]: sns.scatterplot(x='weight', y='mpg', data=mpg)
        plt.title('Relationship between weight and mileage')
```

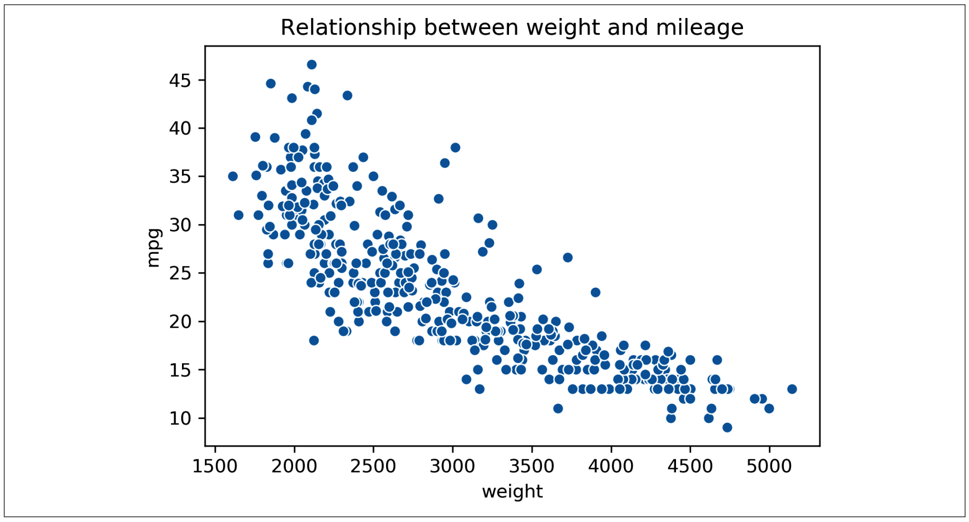


Figure 13-4. Scatterplot of mpg by weight

Alternatively, we could produce scatterplots across all pairs of our dataset with the `pairplot()` function from `seaborn`. Histograms of each variable are included along the diagonal, as seen in Figure 13-5:

```
In[14]: sns.pairplot(mpg[['mpg','horsepower','weight']])
```

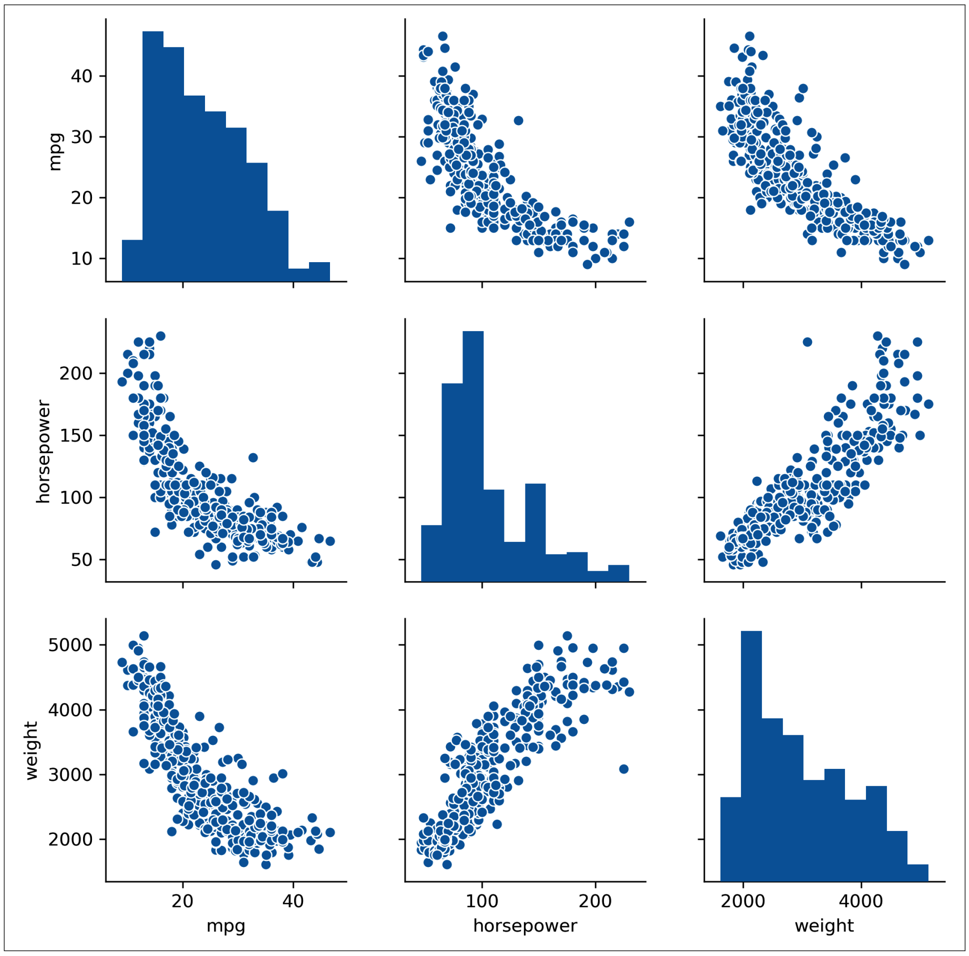


Figure 13-5. Pairplot of mpg, horsepower, and weight

Linear Regression

Now it's time for a linear regression. To do this, we'll use `linregress()` from `scipy`, which also looks for two `numpy` arrays or `pandas` Series. We'll specify which variable is our independent and dependent variable with the `x` and `y` arguments, respectively:

```
In[15]: # Linear regression of weight on mpg
        stats.linregress(x=mpg['weight'], y=mpg['mpg'])
```

```
Out[15]: LinregressResult(slope=-0.007647342535779578,
   intercept=46.21652454901758, rvalue=-0.8322442148315754,
   pvalue=6.015296051435726e-102, stderr=0.0002579632782734318)
```

Again, you'll see that some of the output you may be used to is missing here. *Be careful:* the `rvalue` included is the *correlation coefficient*, not R-square. For a richer linear regression output, check out the `statsmodels` module.

Last but not least, let's overlay our regression line to a scatterplot. `seaborn` has a separate function to do just that: `regplot()`. As usual, we'll specify our independent and dependent variables, and where to get the data. This results in Figure 13-6:

```
In[16]: # Fit regression line to scatterplot
        sns.regplot(x="weight", y="mpg", data=mpg)
        plt.xlabel('Weight (lbs)')
        plt.ylabel('Mileage (mpg)')
        plt.title('Relationship between weight and mileage')
```

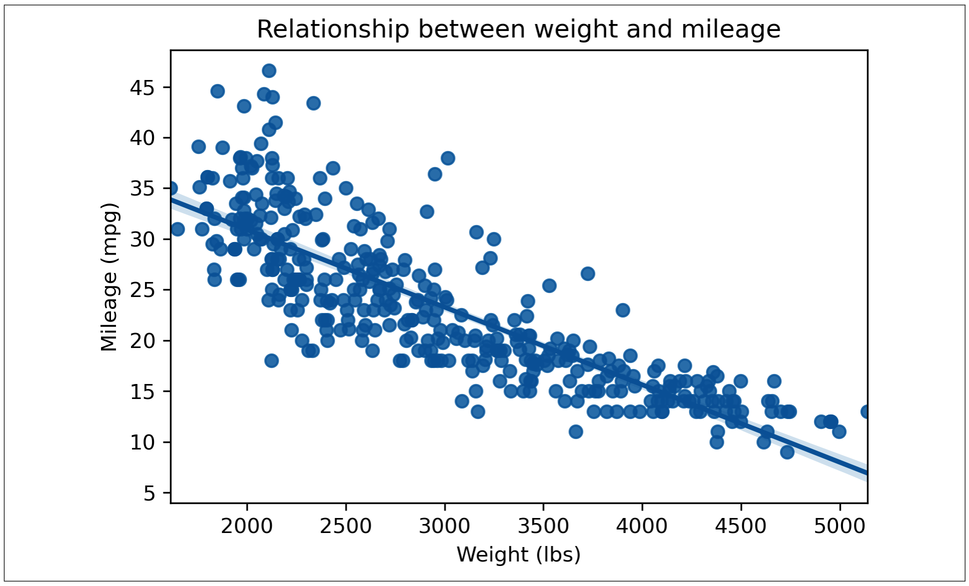


Figure 13-6. Scatterplot with fit regression line of mpg by weight

Train/Test Split and Validation

At the end of Chapter 9 you learned how to apply a train/test split when building a linear regression model in R.

We will use the `train_test_split()` function to split our dataset into *four* DataFrames: not just by training and testing but also independent and dependent variables. We'll pass in a DataFrame containing our independent variable first, then one containing the dependent variable. Using the `random_state` argument, we'll seed the random number generator so the results remain consistent for this example:

```
In[17]: X_train, X_test, y_train, y_test =
        model_selection.train_test_split(mpg[['weight']], mpg[['mpg']],
        random_state=1234)
```

By default, the data is split 75/25 between training and testing subsets:

```
In[18]:  y_train.shape

Out[18]: (294, 1)


In[19]:  y_test.shape

Out[19]: (98, 1)
```

Now, let's fit the model to the training data. First we'll specify the linear model with `LinearRegression()`, then we'll train the model with `regr.fit()`. To get the predicted values for the test dataset, we can use `predict()`. This results in a `numpy` array, not a `pandas` DataFrame, so the `head()` method won't work to print the first few rows. We can, however, slice it:

```
In[20]:  # Create linear regression object
         regr = linear_model.LinearRegression()

         # Train the model using the training sets
         regr.fit(X_train, y_train)

         # Make predictions using the testing set
         y_pred = regr.predict(X_test)

         # Print first five observations
         y_pred[:5]

Out[20]:  array([[14.86634263],
         [23.48793632],
         [26.2781699 ],
         [27.69989655],
         [29.05319785]])
```

The `coef_` attribute returns the coefficient of our test model:

```
In[21]:  regr.coef_

Out[21]: array([[-0.00760282]])
```

To get more information about the model, such as the coefficient p-values or R-squared, try fitting it with the `statsmodels` package.

For now, we'll evaluate the performance of the model on our test data, this time using the `metrics` submodule of `sklearn`. We'll pass in our actual and predicted values to the `r2_score()` and `mean_squared_error()` functions, which will return the R-squared and RMSE, respectively.

```
In[22]:  metrics.r2_score(y_test, y_pred)

Out[22]: 0.6811923996681357


In[23]:  metrics.mean_squared_error(y_test, y_pred)

Out[23]: 21.63348076436662
```

Conclusion

The usual caveat applies to this chapter: we've just scratched the surface of what analysis is possible on this or any other dataset. But I hope you feel you've hit your stride on working with data in Python.

Exercises

Take another look at the *ais* dataset, this time using Python. Read the Excel workbook in from the book repository (*https://oreil.ly/dsZDM*) and complete the following. You should be pretty comfortable with this analysis by now.

1. Visualize the distribution of red blood cell count (*rcc*) by sex (*sex*).
2. Is there a significant difference in red blood cell count between the two groups of sex?
3. Produce a correlation matrix of the relevant variables in this dataset.
4. Visualize the relationship of height (*ht*) and weight (*wt*).
5. Regress *ht* on *wt*. Find the equation of the fit regression line. Is there a significant relationship?
6. Split your regression model into training and testing subsets. What is the R-squared and RMSE on your test model?