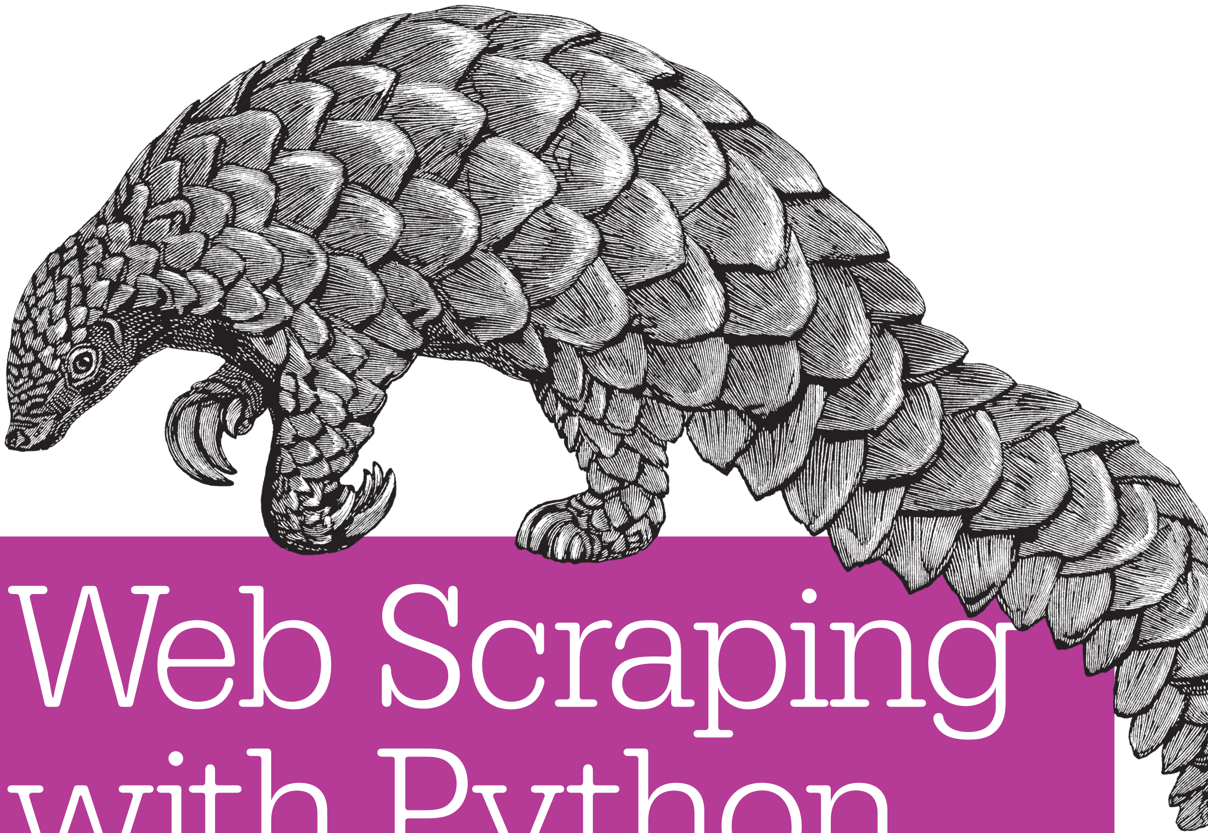
O'REILLY®



Web Scraping with Python

COLLECTING DATA FROM THE MODERN WEB

Free Sampler

Ryan Mitchell

Web Scraping with Python
by Ryan Mitchell

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Editors: Simon St. Laurent and Allyson MacDonald
Production Editor: Shiny Kalapurakkel
Copyeditor: Jasmine Kwityn
Proofreader: Carla Thornton
Indexer: Lucie Haskins
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

June 2015: First Edition

Revision History for the First Edition
2015-06-10: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491910276* for release details.

978-1-491-91027-6

[LSI]

Table of Contents

PART I
Building Scrapers

This section focuses on the basic mechanics of web scraping: how to use Python to request information from a web server, how to perform basic handling of the server's response, and how to begin interacting with a website in an automated fashion. By the end, you'll be cruising around the Internet with ease, building scrapers that can hop from one domain to another, gather information, and store that information for later use.

To be honest, web scraping is a fantastic field to get into if you want a huge payout for relatively little upfront investment. In all likelihood, 90% of web scraping projects you'll encounter will draw on techniques used in just the next six chapters. This section covers what the general (albeit technically savvy) public tends to think of when they think of "web scrapers":

- Retrieving HTML data from a domain name
- Parsing that data for target information
- Storing the target information
- Optionally, moving to another page to repeat the process

This will give you a solid foundation before moving on to more complex projects in part II. Don't be fooled into thinking that this first section isn't as important as some of the more advanced projects in the second half. You will use nearly all the information in the first half of this book on a daily basis while writing web scrapers.

CHAPTER 1

Your First Web Scraper

Once you start web scraping, you start to appreciate all the little things that browsers do for us. The Web, without a layer of HTML formatting, CSS styling, JavaScript execution, and image rendering, can look a little intimidating at first, but in this chapter, as well as the next one, we'll cover how to format and interpret data without the help of a browser.

This chapter will start with the basics of sending a GET request to a web server for a specific page, reading the HTML output from that page, and doing some simple data extraction in order to isolate the content that we are looking for.

Connecting

If you haven't spent much time in networking, or network security, the mechanics of the Internet might seem a little mysterious. We don't want to think about what, exactly, the network is doing every time we open a browser and go to *http://google.com*, and, these days, we don't have to. In fact, I would argue that it's fantastic that computer interfaces have advanced to the point where most people who use the Internet don't have the faintest idea about how it works.

However, web scraping requires stripping away some of this shroud of interface, not just at the browser level (how it interprets all of this HTML, CSS, and JavaScript), but occasionally at the level of the network connection.

To give you some idea of the infrastructure required to get information to your browser, let's use the following example. Alice owns a web server. Bob uses a desktop computer, which is trying to connect to Alice's server. When one machine wants to talk to another machine, something like the following exchange takes place:

1. Bob's computer sends along a stream of 1 and 0 bits, indicated by high and low voltages on a wire. These bits form some information, containing a header and body. The header contains an immediate destination of his local router's MAC address, with a final destination of Alice's IP address. The body contains his request for Alice's server application.
2. Bob's local router receives all these 1's and 0's and interprets them as a packet, from Bob's own MAC address, and destined for Alice's IP address. His router stamps its own IP address on the packet as the "from" IP address, and sends it off across the Internet.
3. Bob's packet traverses several intermediary servers, which direct his packet toward the correct physical/wired path, on to Alice's server.
4. Alice's server receives the packet, at her IP address.
5. Alice's server reads the packet port destination (almost always port 80 for web applications, this can be thought of as something like an "apartment number" for packet data, where the IP address is the "street address"), in the header, and passes it off to the appropriate application – the web server application.
6. The web server application receives a stream of data from the server processor. This data says something like:
   - This is a GET request
   - The following file is requested: index.html
7. The web server locates the correct HTML file, bundles it up into a new packet to send to Bob, and sends it through to its local router, for transport back to Bob's machine, through the same process.

And *voilà*! We have The Internet.

So, where in this exchange did the web browser come into play? Absolutely nowhere. In fact, browsers are a relatively recent invention in the history of the Internet, when Nexus was released in 1990.

Yes, the web browser is a very useful application for creating these packets of information, sending them off, and interpreting the data you get back as pretty pictures, sounds, videos, and text. However, a web browser is just code, and code can be taken apart, broken into its basic components, re-written, re-used, and made to do anything we want. A web browser can tell the processor to send some data to the application that handles your wireless (or wired) interface, but many languages have libraries that can do that just as well.

Let's take a look at how this is done in Python:

```
from urllib.request import urlopen
html = urlopen("http://pythonscraping.com/pages/page1.html")
print(html.read())
```

You can save this code as *scrapetest.py* and run it in your terminal using the command:

```
$python scrapetest.py
```

Note that if you also have Python 2.x installed on your machine, you may need to explicitly call Python 3.x by running the command this way:

```
$python3 scrapetest.py
```

This will output the complete HTML code for the page at *http://pythonscraping.com/pages/page1.html*. More accurately, this outputs the HTML file *page1.html*, found in the directory *<web root>/pages*, on the server located at the domain name *http://pythonscraping.com*.

What's the difference? Most modern web pages have many resource files associated with them. These could be image files, JavaScript files, CSS files, or any other content that the page you are requesting is linked to. When a web browser hits a tag such as `<img src="cuteKitten.jpg">`, the browser knows that it needs to make another request to the server to get the data at the file *cuteKitten.jpg* in order to fully render the page for the user. Keep in mind that our Python script doesn't have the logic to go back and request multiple files (yet);it can only read the single HTML file that we've requested.

So how does it do this? Thanks to the plain-English nature of Python, the line

```
from urllib.request import urlopen
```

means what it looks like it means: it looks at the Python module request (found within the urllib library) and imports only the function `urlopen`.

urllib or urllib2?

If you've used the urllib2 library in Python 2.x, you might have noticed that things have changed somewhat between urllib2 and urllib. In Python 3.x, urllib2 was renamed urllib and was split into several submodules: `urllib.request`, `urllib.parse`, and `urllib.error`. Although function names mostly remain the same, you might want to note which functions have moved to submodules when using the new urllib.

`urllib` is a standard Python library (meaning you don't have to install anything extra to run this example) and contains functions for requesting data across the web, handling cookies, and even changing metadata such as headers and your user agent. We will be using `urllib` extensively throughout the book, so we recommend you read the Python documentation for the library (*https://docs.python.org/3/library/urllib.html*).

urlopen is used to open a remote object across a network and read it. Because it is a fairly generic library (it can read HTML files, image files, or any other file stream with ease), we will be using it quite frequently throughout the book.

An Introduction to BeautifulSoup

"Beautiful Soup, so rich and green,
Waiting in a hot tureen!
Who for such dainties would not stoop?
Soup of the evening, beautiful Soup!"

The BeautifulSoup library was named after a Lewis Carroll poem of the same name in *Alice's Adventures in Wonderland*. In the story, this poem is sung by a character called the Mock Turtle (itself a pun on the popular Victorian dish Mock Turtle Soup made not of turtle but of cow).

Like its Wonderland namesake, BeautifulSoup tries to make sense of the nonsensical; it helps format and organize the messy web by fixing bad HTML and presenting us with easily-traversible Python objects representing XML structures.

Installing BeautifulSoup

Because the BeautifulSoup library is not a default Python library, it must be installed. We will be using the BeautifulSoup 4 library (also known as BS4) throughout this book. The complete instructions for installing BeautifulSoup 4 can be found at Crummy.com; however, the basic method for Linux is:

```
$sudo apt-get install python-bs4
```

and for Macs:

```
$sudo easy_install pip
```

This installs the Python package manager *pip*. Then run the following:

```
$pip install beautifulsoup4
```

to install the library.

Again, note that if you have both Python 2.x and 3.x installed on your machine, you might need to call `python3` explicitly:

```
$python3 myScript.py
```

Make sure to also use this when installing packages, or the packages might be installed under Python 2.x, but not Python 3.x:

```
$sudo python3 setup.py install
```

If using pip, you can also call `pip3` to install the Python 3.x versions of packages:

```
$pip3 install beautifulsoup4
```

Installing packages in Windows is nearly identical to the process for the Mac and Linux. Download the most recent BeautifulSoup 4 release from the download URL above, navigate to the directory you unzipped it to, and run:

```
>python setup.py install
```

And that's it! BeautifulSoup will now be recognized as a Python library on your machine. You can test this out by opening a Python terminal and importing it:

```
$python
> from bs4 import BeautifulSoup
```

The import should complete without errors.

In addition, there is an .exe installer for pip on Windows, so you can easily install and manage packages:

```
>pip install beautifulsoup4
```

Keeping Libraries Straight with Virtual Environments

If you intend to work on multiple Python projects or you need a way to easily bundle projects with all associated libraries, or you're worried about potential conflicts between installed libraries, you can install a Python virtual environment to keep everything separated and easy to manage.

When you install a Python library without a virtual environment, you are installing it *globally*. This usually requires that you be an administrator, or run as root, and that Python library exists for every user and every project on the machine. Fortunately, creating a virtual environment is easy:

```
$ virtualenv scrapingEnv
```

This creates a new environment called *scrapingEnv*, which you must activate in order to use:

```
$ cd scrapingEnv/
$ source bin/activate
```

After you have activated the environment, you will see that environment's name in your command line prompt, reminding you that you're currently working with it. Any libraries you install or scripts you run will be under that virtual environment only.

Working in the newly-created scrapingEnv environment, I can install and use BeautifulSoup, for instance:

```
(scrapingEnv)ryan$ pip install beautifulsoup4
(scrapingEnv)ryan$ python
> from bs4 import BeautifulSoup
>
```

I can leave the environment with the deactivate command, after which I can no longer access any libraries that were installed inside the virtual environment:

```
(scrapingEnv)ryan$ deactivate
ryan$ python
> from bs4 import BeautifulSoup
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'bs4'
```

Keeping all your libraries separated by project also makes it easy to zip up the entire environment folder and send it to someone else. As long as they have the same version of Python installed on their machine, your code will work from the virtual environment without requiring them to install any libraries themselves.

Although we won't explicitly instruct you to use a virtual environment in all of this book's examples, keep in mind that you can apply virtual environment any time simply by activating it beforehand.

Running BeautifulSoup

The most commonly used object in the BeautifulSoup library is, appropriately, the BeautifulSoup object. Let's take a look at it in action, modifying the example found in the beginning of this chapter:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
html = urlopen("http://www.pythonscraping.com/pages/page1.html")
bsObj = BeautifulSoup(html.read())
print(bsObj.h1)
```

The output is:

```
<h1>An Interesting Title</h1>
```

As in the example before, we are importing the urlopen library and calling `html.read()` in order to get the HTML content of the page. This HTML content is then transformed into a BeautifulSoup object, with the following structure:

- **html** → *<html><head>...</head><body>...</body></html>*
  — **head** → *<head><title>A Useful Page<title></head>*
    — **title** → *<title>A Useful Page</title>*
  — **body** → *<body><h1>An Int...</h1><div>Lorem ip...</div></body>*
    — **h1** → *<h1>An Interesting Title</h1>*
    — **div** → *<div>Lorem Ipsum dolor...</div>*

Note that the `<h1>` tag that we extracted from the page was nested two layers deep into our BeautifulSoup object structure (html → body → h1). However, when we actually fetched it from the object, we called the h1 tag directly:

```
bsObj.h1
```

In fact, any of the following function calls would produce the same output:

```
bsObj.html.body.h1
bsObj.body.h1
bsObj.html.h1
```

We hope this small taste of `BeautifulSoup` has given you an idea of the power and simplicity of this library. Virtually any information can be extracted from any HTML (or XML) file, as long as it has some identifying tag surrounding it, or near it. In chapter 3, we'll delve more deeply into some more-complex BeautifulSoup function calls, as well as take a look at regular expressions and how they can be used with `BeautifulSoup` in order to extract information from websites.

Connecting Reliably

The web is messy. Data is poorly formatted, websites go down, and closing tags go missing. One of the most frustrating experiences in web scraping is to go to sleep with a scraper running, dreaming of all the data you'll have in your database the next day—only to find out that the scraper hit an error on some unexpected data format and stopped execution shortly after you stopped looking at the screen. In situations like these, you might be tempted to curse the name of the developer who created the website (and the oddly formatted data), but the person you should really be kicking is yourself, for not anticipating the exception in the first place!

Let's take a look at the first line of our scraper, after the import statements, and figure out how to handle any exceptions this might throw:

```
html = urlopen("http://www.pythonscraping.com/pages/page1.html")
```

There are two main things that can go wrong in this line:

- The page is not found on the server (or there was some error in retrieving it)
- The server is not found

In the first situation, an HTTP error will be returned. This HTTP error may be "404 Page Not Found," "500 Internal Server Error," etc. In all of these cases, the `urlopen` function will throw the generic exception "HTTPError" We can handle this exception in the following way:

```
try:
    html = urlopen("http://www.pythonscraping.com/pages/page1.html")
except HTTPError as e:
    print(e)
    #return null, break, or do some other "Plan B"
else:
    #program continues. Note: If you return or break in the
    #exception catch, you do not need to use the "else" statement
```

If an HTTP error code is returned, the program now prints the error, and does not execute the rest of the program under the `else` statement.

If the server is not found at all (if, say, *http://www.pythonscraping.com* was down, or the URL was mistyped), `urlopen` returns a `None` object. This object is analogous to `null` in other programming languages. We can add a check to see if the returned html is `None`:

```
    if html is None:
        print("URL is not found")
    else:
        #program continues
```

Of course, if the page is retrieved successfully from the server, there is still the issue of the content on the page not quite being what we expected. Every time you access a tag in a BeautifulSoup object, it's smart to add a check to make sure the tag actually exists. If you attempt to access a tag that does not exist, BeautifulSoup will return a `None` object. The problem is, attempting to access a tag on a `None` object itself will result in an `AttributeError` being thrown.

The following line (where `nonExistentTag` is a made-up tag, not the name of a real BeautifulSoup function):

```
print(bsObj.nonExistentTag)
```

returns a `None` object. This object is perfectly reasonable to handle and check for. The trouble comes if you don't check for it, but instead go on and try to call some other function on the `None` object, as illustrated in the following:

```
print(bsObj.nonExistentTag.someTag)
```

which returns the exception:

```
AttributeError: 'NoneType' object has no attribute 'someTag'
```

So how can we guard against these two situations? The easiest way is to explicitly check for both situations:

```
        try:
            badContent = bsObj.nonExistingTag.anotherTag
        except AttributeError as e:
            print("Tag was not found")
        else:
            if badContent == None:
               print ("Tag was not found")
            else:
                print(badContent)
```

This checking and handling of every error does seem laborious at first, but it's easy to add a little reorganization to this code to make it less difficult to write (and, more importantly, much less difficult to read). This code, for example, is our same scraper written in a slightly different way:

```
from urllib.request import urlopen
from urllib.error import HTTPError
from bs4 import BeautifulSoup
 def getTitle(url):
    try:
        html = urlopen(url)
    except HTTPError as e:
        return None
    try:
        bsObj = BeautifulSoup(html.read())
        title = bsObj.body.h1
    except AttributeError as e:
        return None
    return title
title = getTitle("http://www.pythonscraping.com/pages/page1.html")
if title == None:
    print("Title could not be found")
else:
    print(title)
```

In this example, we're creating a function `getTitle`, which returns either the title of the page, or a `None` object if there was some problem with retrieving it. Inside `getTitle`, we check for an `HTTPError`, as in the previous example, and also encapsulate two of the BeautifulSoup lines inside one `try` statement. An `AttributeError` might be thrown from either of these lines (if the server did not exist, `html` would be a `None` object, and `html.read()` would throw an `AttributeError`). We could, in fact, encompass as many lines as we wanted inside one `try` statement, or call another function entirely, which can throw an `AttributeError` at any point.

When writing scrapers, it's important to think about the overall pattern of your code in order to handle exceptions and make it readable at the same time. You'll also likely want to heavily reuse code. Having generic functions such as `getSiteHTML` and `getTitle` (complete with thorough exception handling) makes it easy to quickly—and reliably—scrape the web.