# **Web Scraping Workshop**

Using [Requests](http://python-requests.org/) and [Beautiful Soup](http://www.crummy.com/software/BeautifulSoup/), with the most recent [Beautiful Soup 4 docs](http://www.crummy.com/software/BeautifulSoup/bs4/doc/).

## **Getting Started**

Install our tools (preferably in a new virtualenv):

pip install beautifulsoup4  
pip install requests

## **Start Scraping!**

Lets grab the Free Book Samplers from O'Reilly: <http://oreilly.com/store/samplers.html>.

>>> import requests  
>>>  
>>> result = requests.get("http://oreilly.com/store/samplers.html")

Make sure we got a result.

>>> result.status\_code  
200  
>>> result.headers  
...

Store your content in an easy-to-type variable!

>>> c = result.content

Start parsing with Beautiful Soup. NOTE: If you installed with pip, you'll need to import from bs4. If you download the source, you'll need to import from BeautifulSoup (which is what they do in the [online docs](http://www.crummy.com/software/BeautifulSoup/bs3/documentation.html#Quick%20Start)).

>>> from bs4 import BeautifulSoup  
>>> soup = BeautifulSoup(c)  
>>> samples = soup.find\_all("a", "item-title")  
>>> samples[0]  
<a class="item-title" href="http://cdn.oreilly.com/oreilly/booksamplers/9780596004927\_sampler.pdf">  
Programming Perl  
</a>

Now, pick apart individual links.

>>> data = {}  
>>> for a in samples:  
... title = a.string.strip()  
... data[title] = a.attrs['href']

Check out the keys/values in the data dict. Rejoice!

Now go scrape some stuff!

### **Introduction**

The web provides us with more data than any of us can read and understand, so we often want to work with that information programmatically in order to make sense of it. Sometimes, that data is provided to us by website creators via .csv or comma-separated values files, or through an API (Application Programming Interface). Other times, we need to collect text from the web ourselves.

This tutorial will go over how to work with the [Requests](http://docs.python-requests.org/en/master/) and [Beautiful Soup](https://www.crummy.com/software/BeautifulSoup/) Python packages in order to make use of data from web pages. The Requests module lets you integrate your Python programs with web services, while the Beautiful Soup module is designed to make screen-scraping get done quickly. Using the Python interactive console and these two libraries, we’ll go through how to collect a web page and work with the textual information available there.

## Prerequisites

To complete this tutorial, you'll need a development environment for Python 3. You can follow the appropriate guide for your operating system available from the series [How To Install and Set Up a Local Programming Environment for Python 3](https://www.digitalocean.com/community/tutorial_series/how-to-install-and-set-up-a-local-programming-environment-for-python-3) or [How To Install Python 3 and Set Up a Programming Environment on an Ubuntu 16.04 Server](https://www.digitalocean.com/community/tutorials/how-to-install-python-3-and-set-up-a-programming-environment-on-an-ubuntu-16-04-server) to configure everything you need.

Additionally, you should be familiar with:

* The [Python Interactive Console](https://www.digitalocean.com/community/tutorials/how-to-work-with-the-python-interactive-console)
* [Importing Modules in Python 3](https://www.digitalocean.com/community/tutorials/how-to-import-modules-in-python-3)
* HTML structure and tagging

With your development environment set up and these Python programming concepts in mind, let’s start working with Requests and Beautiful Soup.

## Installing Requests

Let’s begin by activating our Python 3 programming environment. Make sure you’re in the directory where your environment is located, and run the following command:

* . my\_env/bin/activate

In order to work with web pages, we’re going to need to request the page. The Requests library allows you to make use of HTTP within your Python programs in a human readable way.

With our programming environment activated, we’ll install Requests with pip:

* pip install requests

While the Requests library is being installed, you’ll receive the following output:

Output

Collecting requests  
 Downloading requests-2.18.1-py2.py3-none-any.whl (88kB)  
 100% |████████████████████████████████| 92kB 3.1MB/s   
...  
Installing collected packages: chardet, urllib3, certifi, idna, requests  
Successfully installed certifi-2017.4.17 chardet-3.0.4 idna-2.5 requests-2.18.1 urllib3-1.21.1

If Requests was previously installed, you would have received feedback similar to the following from your terminal window:

Output

Requirement already satisfied  
...

With Requests installed into our programming environment, we can go on to install the next module.

## Installing Beautiful Soup

Just as we did with Requests, we’ll install Beautiful Soup with pip. The current version of Beautiful Soup 4 can be installed with the following command:

* pip install beautifulsoup4

Once you run this command, you should see output that looks similar to the following:

Output

Collecting beautifulsoup4  
 Downloading beautifulsoup4-4.6.0-py3-none-any.whl (86kB)  
 100% |████████████████████████████████| 92kB 4.4MB/s   
Installing collected packages: beautifulsoup4  
Successfully installed beautifulsoup4-4.6.0

Now that both Beautiful Soup and Requests are installed, we can move on to understanding how to work with the libraries to scrape websites.

## Collecting a Web Page with Requests

With the two Python libraries we’ll be using now installed, we’re can familiarize ourselves with stepping through a basic web page.

Let’s first move into the [Python Interactive Console](https://www.digitalocean.com/community/tutorials/how-to-work-with-the-python-interactive-console):

* python

From here, we’ll import the Requests module so that we can collect a sample web page:

* import requests

We’ll assign the URL (below) of the sample web page, mockturtle.html to the [variable](https://www.digitalocean.com/community/tutorials/how-to-use-variables-in-python-3) url:

* url = 'https://assets.digitalocean.com/articles/eng\_python/beautiful-soup/mockturtle.html'

Next, we can assign the result of a request of that page to the variable page with the [request.get()method](http://docs.python-requests.org/en/master/user/quickstart/#make-a-request). We pass the page’s URL (that was assigned to the url variable) to that method.

* page = requests.get(url)

The variable page is assigned a Response object:

>>> page  
<Response [200]>  
>>>

The Response object above tells us the status\_code property in square brackets (in this case 200). This attribute can be called explicitly:

>>> page.status\_code  
200  
>>>

The returned code of 200 tells us that the page downloaded successfully. Codes that begin with the number 2 generally indicate success, while codes that begin with a 4 or 5 indicate that an error has occurred. You can read more about HTTP status codes from the [W3C’s Status Code Definitions](https://www.w3.org/Protocols/HTTP/1.1/draft-ietf-http-v11-spec-01#Status-Codes).

In order to work with web data, we’re going to want to access the text-based content of web files. We can read the content of the server’s response with page.text (or page.content if we would like to access the response in bytes).

* page.text

Once we press ENTER, we’ll receive the following output:

Output

'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"\n   
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">\n\n<html lang="en-US"   
xmlns="http://www.w3.org/1999/xhtml" xml:lang="en-US">\n<head>\n <meta   
http-equiv="content-type" content="text/html; charset=us-ascii" />\n\n <title>Turtle   
Soup</title>\n</head>\n\n<body>\n <h1>Turtle Soup</h1>\n\n <p class="verse"   
id="first">Beautiful Soup, so rich and green,<br />\n Waiting in a hot tureen!<br />\n Who for   
such dainties would not stoop?<br />\n Soup of the evening, beautiful Soup!<br />\n Soup of   
the evening, beautiful Soup!<br /></p>\n\n <p class="chorus" id="second">Beau--ootiful   
Soo--oop!<br />\n Beau--ootiful Soo--oop!<br />\n Soo--oop of the e--e--evening,<br />\n   
Beautiful, beautiful Soup!<br /></p>\n\n <p class="verse" id="third">Beautiful Soup! Who cares   
for fish,<br />\n Game or any other dish?<br />\n Who would not give all else for two<br />\n   
Pennyworth only of Beautiful Soup?<br />\n Pennyworth only of beautiful Soup?<br /></p>\n\n   
<p class="chorus" id="fourth">Beau--ootiful Soo--oop!<br />\n Beau--ootiful Soo--oop!<br />\n   
Soo--oop of the e--e--evening,<br />\n Beautiful, beauti--FUL SOUP!<br   
/></p>\n</body>\n</html>\n'  
>>>

Here we see that the full text of the page was printed out, with all of its HTML tags. However, it is difficult to read because there is not much spacing.

In the next section, we can leverage the Beautiful Soup module to work with this textual data in a more human-friendly manner.

## Stepping Through a Page with Beautiful Soup

The Beautiful Soup library creates a parse tree from parsed HTML and XML documents (including documents with non-closed tags or [tag soup](https://en.wikipedia.org/wiki/Tag_soup) and other malformed markup). This functionality will make the web page text more readable than what we saw coming from the Requests module.

To start, we’ll import Beautiful Soup into the Python console:

* from bs4 import BeautifulSoup

Next, we’ll run the page.text document through the module to give us a BeautifulSoup object — that is, a parse tree from this parsed page that we’ll get from running Python’s built-in [html.parser](https://docs.python.org/3/library/html.parser.html) over the HTML. The constructed object represents the mockturtle.html document as a nested data structure. This is assigned to the variable soup.

* soup = BeautifulSoup(page.text, 'html.parser')

To show the contents of the page on the terminal, we can print it with the prettify() method in order to turn the Beautiful Soup parse tree into a nicely formatted Unicode string.

* print(soup.prettify())

This will render each HTML tag on its own line:

Output

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"  
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">  
<html lang="en-US" xml:lang="en-US" xmlns="http://www.w3.org/1999/xhtml">  
 <head>  
 <meta content="text/html; charset=utf-8" http-equiv="content-type"/>  
 <title>  
 Turtle Soup  
 </title>  
 </head>  
 <body>  
 <h1>  
 Turtle Soup  
 </h1>  
 <p class="verse" id="first">  
 Beautiful Soup, so rich and green,  
 <br/>  
 Waiting in a hot tureen!  
 <br/>  
 Who for such dainties would not stoop?  
 <br/>  
 Soup of the evening, beautiful Soup!  
 ...  
</html>

In the output above, we can see that there is one tag per line and also that the tags are nested because of the tree schema used by Beautiful Soup.

### **Finding Instances of a Tag**

We can extract a single tag from a page by using Beautiful Soup’s find\_all method. This will return all instances of a given tag within a document.

* soup.find\_all('p')

Running that method on our object returns the full text of the song along with the relevant <p> tags and any tags contained within that requested tag, which here includes the line break tags <br/>:

Output

[<p class="verse" id="first">Beautiful Soup, so rich and green,<br/>  
 Waiting in a hot tureen!<br/>  
 Who for such dainties would not stoop?<br/>  
 Soup of the evening, beautiful Soup!<br/>  
 Soup of the evening, beautiful Soup!<br/></p>, <p class="chorus" id="second">Beau--ootiful Soo--oop!<br/>  
...  
 Beau--ootiful Soo--oop!<br/>  
 Soo--oop of the e--e--evening,<br/>  
 Beautiful, beauti--FUL SOUP!<br/></p>]

You will notice in the output above that the data is contained in square brackets [ ]. This means it is a Python [list data type](https://www.digitalocean.com/community/tutorials/understanding-lists-in-python-3).

Because it is a list, we can call a particular item within it (for example, the third <p> element), and use the get\_text() method to extract all the text from inside that tag:

* soup.find\_all('p')[2].get\_text()

The output that we receive will be what is in the third <p> element in this case:

Output

'Beautiful Soup! Who cares for fish,\n Game or any other dish?\n Who would not give all else for two\n Pennyworth only of Beautiful Soup?\n Pennyworth only of beautiful Soup?'

Note that \n line breaks are also shown in the returned string above.

### **Finding Tags by Class and ID**

HTML elements that refer to CSS selectors like class and ID can be helpful to look at when working with web data using Beautiful Soup. We can target specific classes and IDs by using the find\_all() method and passing the class and ID strings as arguments.

First, let’s find all of the instances of the class chorus. In Beautiful Soup we will assign the string for the class to the keyword argument class\_:

* soup.find\_all(class\_='chorus')

When we run the above line, we’ll receive the following list as output:

Output

[<p class="chorus" id="second">Beau--ootiful Soo--oop!<br/>  
 Beau--ootiful Soo--oop!<br/>  
 Soo--oop of the e--e--evening,<br/>  
 Beautiful, beautiful Soup!<br/></p>, <p class="chorus" id="fourth">Beau--ootiful Soo--oop!<br/>  
 Beau--ootiful Soo--oop!<br/>  
 Soo--oop of the e--e--evening,<br/>  
 Beautiful, beauti--FUL SOUP!<br/></p>]

The two <p>-tagged sections with the class of chorus were printed out to the terminal.

We can also specify that we want to search for the class chorus only within <p> tags, in case it is used for more than one tag:

* soup.find\_all('p', class\_='chorus')

Running the line above will produce the same output as before.

We can also use Beautiful Soup to target IDs associated with HTML tags. In this case we will assign the string 'third' to the keyword argument id:

* soup.find\_all(id='third')

Once we run the line above, we’ll receive the following output:

Output

[<p class="verse" id="third">Beautiful Soup! Who cares for fish,<br/>  
 Game or any other dish?<br/>  
 Who would not give all else for two<br/>  
 Pennyworth only of Beautiful Soup?<br/>  
 Pennyworth only of beautiful Soup?<br/></p>]

The text associated with the <p> tag with the id of third is printed out to the terminal along with the relevant tags.

## **Making API Requests in Python**

In order to work with APIs in Python, we need tools that will make those requests. In Python, the most common library for making requests and working with APIs is the [**requests** library](https://2.python-requests.org/en/master/). The requests library isn’t part of the standard Python library, so you’ll need to install it to get started.

If you use pip to manage your Python packages, you can install requests using the following command:

pip install requests

If you use conda, the command you’ll need is:

conda install requests

Once you’ve installed the library, you’ll need to import it. Let’s start with that important step:

import requests

Now that we’ve installed and imported the requests library, let’s start using it.

## **Making Our First API Request**

There are many different types of requests. The most commonly used one, a **GET** request, is used to retrieve data. Because we’ll just be working with retrieving data, our focus will be on making ‘get’ requests.

When we make a request, the response from the API comes with a **response code** which tells us whether our request was successful. Response codes are important because they immediately tell us if something went wrong.



To make a ‘GET’ request, we’ll use the [requests.get() function](https://2.python-requests.org/en/master/user/quickstart/#make-a-request), which requires one argument — the URL we want to make the request to. We’ll start by making a request to an API endpoint that doesn’t exist, so we can see what that response code looks like.

response = requests.get("http://api.open-notify.org/this-api-doesnt-exist")

The get() function returns a [response object](https://2.python-requests.org/en/master/user/advanced/#request-and-response-objects). We can use the [response.status\_code](https://2.python-requests.org/en/master/user/quickstart/#response-status-codes) attribute to receive the status code for our request:

print(response.status\_code)

404

The ‘404’ status code might be familiar to you — it’s the status code that a server returns if it can’t find the file we requested. In this case, we asked for this-api-doesnt-exist which (surprise, surprise) didn’t exist!

Let’s learn a little more about common status codes.

## **API Status Codes**

Status codes are returned with every request that is made to a web server. Status codes indicate information about what happened with a request. Here are some codes that are relevant to *GET* requests:

* 200: Everything went okay, and the result has been returned (if any).
* 301: The server is redirecting you to a different endpoint. This can happen when a company switches domain names, or an endpoint name is changed.
* 400: The server thinks you made a bad request. This can happen when you don’t send along the right data, among other things.
* 401: The server thinks you’re not authenticated. Many APIs require login ccredentials, so this happens when you don’t send the right credentials to access an API.
* 403: The resource you’re trying to access is forbidden: you don’t have the right permissions to see it.
* 404: The resource you tried to access wasn’t found on the server.
* 503: The server is not ready to handle the request.

You might notice that all of the status codes that begin with a ‘4’ indicate some sort of error. The first number of status codes indicate their categorization. This is useful — you can know that if your status code starts with a ‘2’ it was successful and if it starts with a ‘4’ or ‘5’ there was an error. If you’re interested you can [read more about status codes](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status)

## **API Documentation**

In order to ensure we make a successful request, when we work with APIs it’s important to consult the documentation. Documentation can seem scary at first, but as you use documentation more and more you’ll find it gets easier.

We’ll be working with the [Open Notify](http://open-notify.org/) API, which gives access to data about the international space station. It’s a great API for learning because it has a very simple design, and doesn’t require authentication. We’ll teach you how to use an API that requires authentication in a later post.

Often there will be multiple APIs available on a particular server. Each of these APIs are commonly called **endpoints**. The first endpoint we’ll use is [http://api.open-notify.org/astros.json](http://open-notify.org/Open-Notify-API/People-In-Space/), which returns data about astronauts currently in space.

If you click the link above to look at the documentation for this endpoint, you’ll see that it says *This API takes no inputs.* This makes it a simple API for us to get started with. We’ll start by making a GET request to the endpoint using the requests library:

response = requests.get("http://api.open-notify.org/astros.json")

print(response.status\_code)

200

We received a ‘200’ code which tells us our request was successful. The documentation tells us that the API response we’ll get is in JSON format. In the next section we’ll learn about JSON, but first let’s use the [response.json() method](https://2.python-requests.org/en/master/user/quickstart/#json-response-content) to see the data we received back from the API:

print(response.json())

{'message': 'success', 'people': [{'name': 'Alexey Ovchinin', 'craft': 'ISS'}, {'name': 'Nick Hague', 'craft': 'ISS'}, {'name': 'Christina Koch', 'craft': 'ISS'}, {'name': 'Alexander Skvortsov', 'craft': 'ISS'}, {'name': 'Luca Parmitano', 'craft': 'ISS'}, {'name': 'Andrew Morgan', 'craft': 'ISS'}], 'number': 6}

## **Working with JSON Data in Python**

[JSON](http://json.org/) (JavaScript Object Notation) is the language of APIs. JSON is a way to encode data structures that ensures that they are easily readable by machines. JSON is the primary format in which data is passed back and forth to APIs, and most API servers will send their responses in JSON format.

You might have noticed that the JSON output we received from the API looked like it contained Python dictionaries, lists, strings and integers. You can think of JSON as being a combination of these objects represented as strings. Let’s look at a simple example:



Python has great JSON support with the [json package](https://docs.python.org/3/library/json.html). The json package is part of the standard library, so we don’t have to install anything to use it. We can both convert *lists* and *dictionaries* to JSON, and convert strings to *lists* and *dictionaries*. In the case of our ISS Pass data, it is a dictionary encoded to a string in JSON format.

The json library has two main functions:

* [json.dumps()](https://docs.python.org/3/library/json.html#json.dumps) — Takes in a Python object, and converts (dumps) it to a string.
* [json.loads()](https://docs.python.org/3/library/json.html#json.loads) — Takes a JSON string, and converts (loads) it to a Python object.

The dumps() function is particularly useful as we can use it to print a formatted string which makes it easier to understand the JSON output, like in the diagram we saw above:

import json

def jprint(obj):

# create a formatted string of the Python JSON object

text = json.dumps(obj, sort\_keys=True, indent=4)

print(text)

jprint(response.json())

{

"message": "success",

"number": 6,

"people": [

{

"craft": "ISS",

"name": "Alexey Ovchinin"

},

{

"craft": "ISS",

"name": "Nick Hague"

},

{

"craft": "ISS",

"name": "Christina Koch"

},

{

"craft": "ISS",

"name": "Alexander Skvortsov"

},

{

"craft": "ISS",

"name": "Luca Parmitano"

},

{

"craft": "ISS",

"name": "Andrew Morgan"

}

]

}

Immediately we can understand the structure of the data more easily – we can see that their are six people currently in space, with their names existing as dictionaries inside a list.

If we compare this to [the documentation for the endpoint](http://open-notify.org/Open-Notify-API/People-In-Space/#json) we’ll see that this matches the specified output for the endpoint.

## **Using an API with Query Parameters**

The http://api.open-notify.org/astros.json endpoint we used earlier does not take any parameters. We just send a GET request and the API sends back data about the number of people currently in space.

It’s very common, however, to have an API endpoint that requires us to specify parameters. An example of this the [http://api.open-notify.org/iss-pass.json endpoint](http://open-notify.org/Open-Notify-API/ISS-Pass-Times/). This endpoint tells us the next times that the international space station will pass over a given location on the earth.

If we look at the documentation, it specifies required lat (latitude) and long (longitude) parameters.

We can do this by adding an optional keyword argument, params, to our request. We can make a dictionary with these parameters, and then pass them into the requests.get function. Here’s what our dictionary would look like, using coordinates for New York City:

parameters = {

"lat": 40.71,

"lon": -74

}

We can also do the same thing directly by adding the parameters directly to the URL. like this: http://api.open-notify.org/iss-pass.json?lat=40.71&lon=-74.

It’s almost always preferable to setup the parameters as a dictionary, because requests takes care of some things that come up, like properly formatting the query parameters, and we don’t need to worry about inserting the values into the URL string.

Let’s make a request using these coordinates and see what response we get.

response = requests.get("http://api.open-notify.org/iss-pass.json", params=parameters)

jprint(response.json())

{

"message": "success",

"request": {

"altitude": 100,

"datetime": 1568062811,

"latitude": 40.71,

"longitude": -74.0,

"passes": 5

},

"response": [

{

"duration": 395,

"risetime": 1568082479

},

{

"duration": 640,

"risetime": 1568088118

},

{

"duration": 614,

"risetime": 1568093944

},

{

"duration": 555,

"risetime": 1568099831

},

{

"duration": 595,

"risetime": 1568105674

}

]

}

## **Understanding the Pass Times**

The JSON response matches what the documentation specified:

* A dictionary with three keys
* The third key, response, contains a list of pass times
* Each pass time is a dictionary with risetime (pass start time) and duration keys.

Let’s extract the pass times from our JSON object:

pass\_times = response.json()['response']

jprint(pass\_times)

[

{

"duration": 395,

"risetime": 1568082479

},

{

"duration": 640,

"risetime": 1568088118

},

{

"duration": 614,

"risetime": 1568093944

},

{

"duration": 555,

"risetime": 1568099831

},

{

"duration": 595,

"risetime": 1568105674

}

]

Next we’ll usea loop to extract just the five risetime values:

risetimes = []

for d in pass\_times:

time = d['risetime']

risetimes.append(time)

print(risetimes)

[1568082479, 1568088118, 1568093944, 1568099831, 1568105674]

These times are difficult to understand – they are in a format known as timestamp or [epoch](https://en.wikipedia.org/wiki/Unix_time). Essentially the time is measured in the number of seconds since January 1st 1970. We can use the Python [datetime.fromtimestamp() method](https://docs.python.org/3/library/datetime.html#datetime.date.fromtimestamp) to convert these into easier to understand times:

from datetime import datetime

times = []

for rt in risetimes:

time = datetime.fromtimestamp(rt)

times.append(time)

print(time)

2019-09-09 21:27:59

2019-09-09 23:01:58

2019-09-10 00:39:04

2019-09-10 02:17:11

2019-09-10 03:54:34

It looks like the ISS passes over New York City often – the next five times happen within a seven hour period!