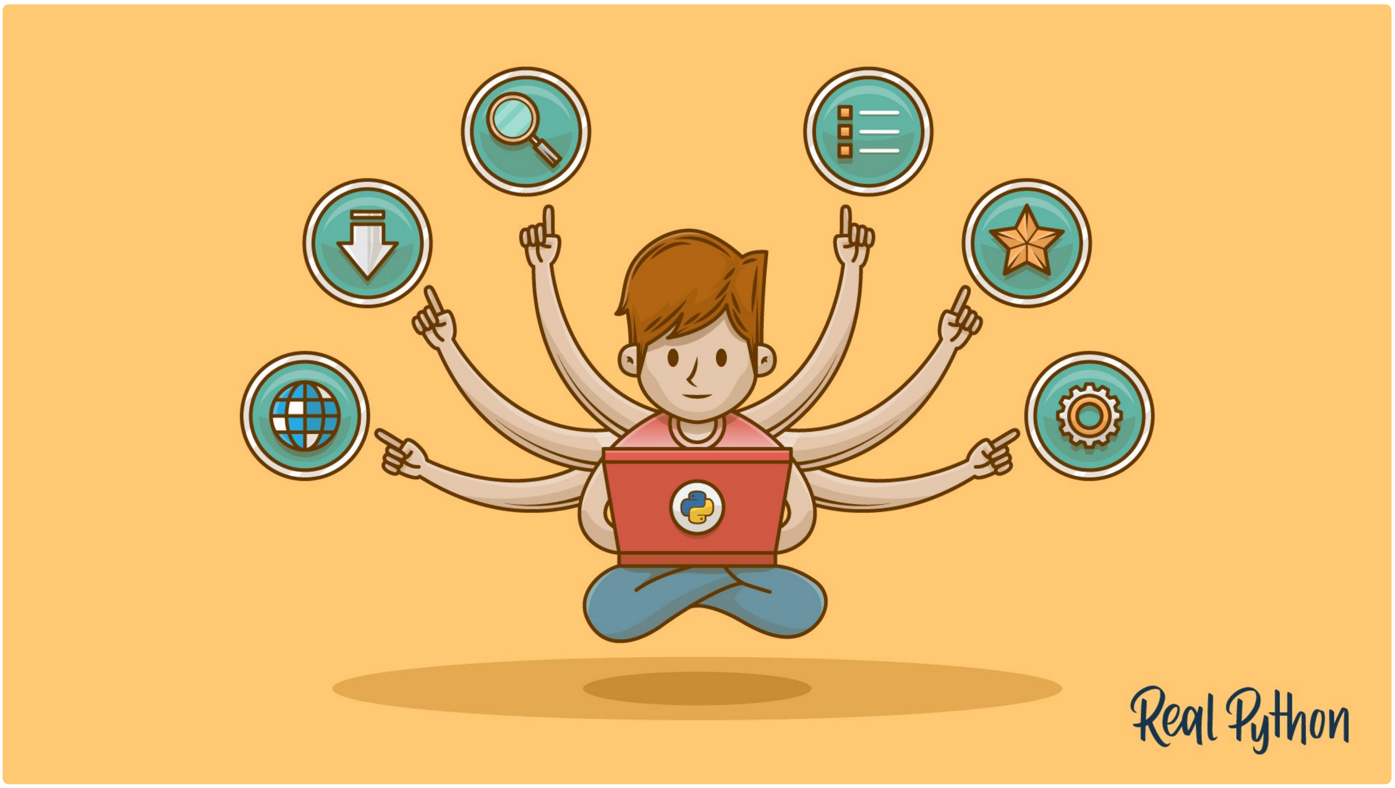


Practical Introduction to Web Scraping in Python

by Colin OKeefe 34 Comments basics web-scraping

Table of Contents

What is web scraping all about?

Imagine that one day, out of the blue, you find yourself thinking “Gee, I wonder who the five most popular mathematicians are?”

You do a bit of thinking, and you get the idea to use Wikipedia’s XTools to measure the popularity of a mathematician by equating popularity with pageviews. For example, look at the page on Henri Poincaré. There, you can see that Poincaré’s pageviews for the last 60 days are, as of December 2017, around 32,000.

Next, you Google “famous mathematicians” and find this resource that lists 100 names. Now you have a page listing mathematicians' names as well as a website that provides information about how "popular" that mathematician is. Now what?

This is where Python and web scraping come in. Web scraping is about downloading structured data from the web, selecting some of that data, and passing along what you selected to another process.

In this tutorial, you will be writing a Python program that downloads the list of 100 mathematicians and their XTools pages, selects data about their popularity, and finishes by telling us the top 5 most popular mathematicians of all time! Let's get started.

Important: We've received an email from an XTools maintainer informing us that scraping XTools is harmful and that automation APIs should be used instead:

> This article on your site is essentially a guide to scraping XTools […] This is not necessary, and it's causing problems for us. We have APIs that should be used for automation, and furthermore, for pageviews specifically folks should be using the official pageviews API.

The example code in the article was modified to no longer make requests to the XTools website. The web scraping techniques demonstrated here are still valid, but please do not use them on web pages of the XTools project. Use the provided automation API instead.

Setting Up Your Python Web Scraper

You will be using Python 3 and Python virtual environments throughout the tutorial. Feel free to set things up however you like. Here is how I tend to do it:

Shell

```
$ python3 -m venv venv
$ . ./venv/bin/activate
```

You will need to install only these two packages:

- requests for performing your HTTP requests
- BeautifulSoup4 for handling all of your HTML processing

Let's install these dependencies with `pip`:

Shell

```
$ pip install requests BeautifulSoup4
```

Finally, if you want to follow along, fire up your favorite text editor and create a file called `mathematicians.py`. Get started by including these `import` statements at the top:

Python

```python
from requests import get
from requests.exceptions import RequestException
from contextlib import closing
from bs4 import BeautifulSoup
```

Making Web Requests

Your first task will be to download web pages. The `requests` package comes to the rescue. It aims to be an easy-to-use tool for doing all things HTTP in Python, and it doesn't dissappoint. In this tutorial, you will need only the `requests.get()` function, but you should definitely checkout the full documentation when you want to go further.

First, here's your function:

Python

```
def simple_get(url):
    """
    Attempts to get the content at `url` by making an HTTP GET request.
    If the content-type of response is some kind of HTML/XML, return the
    text content, otherwise return None.
    """
    try:
        with closing(get(url, stream=True)) as resp:
            if is_good_response(resp):
                return resp.content
            else:
                return None

    except RequestException as e:
        log_error('Error during requests to {0} : {1}'.format(url, str(e)))
        return None


def is_good_response(resp):
    """
    Returns True if the response seems to be HTML, False otherwise.
    """
    content_type = resp.headers['Content-Type'].lower()
    return (resp.status_code == 200
            and content_type is not None
            and content_type.find('html') > -1)


def log_error(e):
    """
    It is always a good idea to log errors.
    This function just prints them, but you can
    make it do anything.
    """
    print(e)
```

The `simple_get()` function accepts a single `url` argument. It then makes a `GET` request to that URL. If nothing goes wrong, you end up with the raw HTML content for the page you requested. If there were any problems with your request (like the URL is bad, or the remote server is down), then your function returns `None`.

You may have noticed the use of the `closing()` function in your definition of `simple_get()`. The `closing()` function ensures that any network resources are freed when they go out of scope in that `with` block. Using `closing()` like that is good practice and helps to prevent fatal errors and network timeouts.

You can test `simple_get()` like this:

Python >>>

```
>>> from mathematicians import simple_get
>>> raw_html = simple_get('https://realpython.com/blog/')
>>> len(raw_html)
33878

>>> no_html = simple_get('https://realpython.com/blog/nope-not-gonna-find-it')
>>> no_html is None
True
```

Wrangling HTML With BeautifulSoup

Once you have raw HTML in front of you, you can start to select and extract. For this purpose, you will be using `BeautifulSoup`. The `BeautifulSoup` constructor parses raw HTML strings and produces an object that mirrors the HTML document's structure. The object includes a slew of methods to select, view, and manipulate DOM nodes and text content.

Consider the following quick and contrived example of an HTML document:

HTML

```html
<!DOCTYPE html>
<html>
<head>
  <title>Contrived Example</title>
</head>
<body>
<p id="eggman"> I am the egg man </p>
<p id="walrus"> I am the walrus </p>
</body>
</html>
```

If the above HTML is saved in the file `contrived.html`, then you can use `BeautifulSoup` like this:

Python >>>

```python
>>> from bs4 import BeautifulSoup
>>> raw_html = open('contrived.html').read()
>>> html = BeautifulSoup(raw_html, 'html.parser')
>>> for p in html.select('p'):
...     if p['id'] == 'walrus':
...         print(p.text)

'I am the walrus'
```

Breaking down the example, you first parse the raw HTML by passing it to the `BeautifulSoup` constructor. `BeautifulSoup` accepts multiple back-end parsers, but the standard back-end is `'html.parser'`, which you supply here as the second argument. (If you neglect to supply that `'html.parser'`, then the code will still work, but you will see a warning print to your screen.)

The `select()` method on your `html` object lets you use CSS selectors to locate elements in the document. In the above case, `html.select('p')` returns a list of paragraph elements. Each `p` has HTML attributes that you can access like a `dict`. In the line `if p['id'] == 'walrus'`, for example, you check if the `id` attribute is equal to the string `'walrus'`, which corresponds to `<p id="walrus">` in the HTML.

Using BeautifulSoup to Get Mathematician Names

Now that you have given the `select()` method in `BeautifulSoup` a short test drive, how do you find out what to supply to `select()`? The fastest way is to step out of Python and into your web browser's developer tools. You can use your browser to examine the document in some detail. I usually look for `id` or `class` element attributes or any other information that uniquely identifies the information I want to extract.

To make matters concrete, turn to the list of mathematicians you saw earlier. If you spend a minute or two looking at this page's source, you can see that each mathematician's name appears inside the text content of an `<li>` tag. To make matters even simpler, `<li>` tags on this page seem to contain nothing but names of mathematicians.

Here's a quick look with Python:

Python >>>

```
>>> raw_html = simple_get('http://www.fabpedigree.com/james/mathmen.htm')
>>> html = BeautifulSoup(raw_html, 'html.parser')
>>> for i, li in enumerate(html.select('li')):
        print(i, li.text)

0  Isaac Newton
 Archimedes
 Carl F. Gauss
 Leonhard Euler
 Bernhard Riemann

1  Archimedes
 Carl F. Gauss
 Leonhard Euler
 Bernhard Riemann

2  Carl F. Gauss
 Leonhard Euler
 Bernhard Riemann

 3  Leonhard Euler
 Bernhard Riemann

4  Bernhard Riemann

# 5 ... and many more...
```

The above experiment shows that some of the `<li>` elements contain multiple names separated by newline characters, while others contain just a single name. With this information in mind, you can write your function to extract a single list of names:

Python

```python
def get_names():
    """
    Downloads the page where the list of mathematicians is found
    and returns a list of strings, one per mathematician
    """
    url = 'http://www.fabpedigree.com/james/mathmen.htm'
    response = simple_get(url)

    if response is not None:
        html = BeautifulSoup(response, 'html.parser')
        names = set()
        for li in html.select('li'):
            for name in li.text.split('\n'):
                if len(name) > 0:
                    names.add(name.strip())
        return list(names)

    # Raise an exception if we failed to get any data from the url
    raise Exception('Error retrieving contents at {}'.format(url))
```

The `get_names()` function downloads the page and iterates over the `<li>` elements, picking out each name that occurs. Next, you add each name to a Python `set`, which ensures that you don't end up with duplicate names. Finally, you convert the set to a list and return it.

Getting the Popularity Score

Nice, you're nearly done! Now that you have a list of names, you need to pick out the pageviews for each one. The function you write is similar to the function you made to get the list of names, only now you supply a name and pick out an integer value from the page.

Again, you should first check out an example page in your browser's developer tools. It looks as if the text appears inside an `<a>` element, and the `href` attribute of that element always contains the string `'latest-60'` as a substring. That's all the information you need to write your function:

Python

```python
def get_hits_on_name(name):
    """
    Accepts a `name` of a mathematician and returns the number
    of hits that mathematician's Wikipedia page received in the
    last 60 days, as an `int`
    """
    # url_root is a template string that is used to build a URL.
    url_root = 'URL_REMOVED_SEE_NOTICE_AT_START_OF_ARTICLE'
    response = simple_get(url_root.format(name))

    if response is not None:
        html = BeautifulSoup(response, 'html.parser')

        hit_link = [a for a in html.select('a')
                    if a['href'].find('latest-60') > -1]

        if len(hit_link) > 0:
            # Strip commas
            link_text = hit_link[0].text.replace(',', '')
            try:
                # Convert to integer
                return int(link_text)
            except:
                log_error("couldn't parse {} as an `int`".format(link_text))

    log_error('No pageviews found for {}'.format(name))
    return None
```

Putting It All Together

You have reached a point where you can finally find out which mathematician is most beloved by the public! The plan is simple:

- Get a list of names
- Iterate over the list to get a "popularity score" for each name
- Finish by sorting the names by popularity

Simple, right? Well, there's one thing that hasn't been mentioned yet: errors.

Working with real-world data is messy, and trying to force messy data into a uniform shape will invariably result in the occasional error jumping in to mess with your nice clean vision of how things ought to be. Ideally, you would like to keep track of errors when they occur in order to get a better sense of the of quality your data.

For your present purposes, you will track instances in which you could not find a popularity score for a given mathematician's name. At the end of the script, you will print a message showing the number of mathematicians who were left out of the rankings.

Here's the code:

Python

```python
if __name__ == '__main__':
    print('Getting the list of names....')
    names = get_names()
    print('... done.\n')

    results = []

    print('Getting stats for each name....')

    for name in names:
        try:
            hits = get_hits_on_name(name)
            if hits is None:
                hits = -1
            results.append((hits, name))
        except:
            results.append((-1, name))
            log_error('error encountered while processing '
                      '{}, skipping'.format(name))

    print('... done.\n')

    results.sort()
    results.reverse()

    if len(results) > 5:
        top_marks = results[:5]
    else:
        top_marks = results

    print('\nThe most popular mathematicians are:\n')
    for (mark, mathematician) in top_marks:
        print('{} with {} pageviews'.format(mathematician, mark))

    no_results = len([res for res in results if res[0] == -1])
    print('\nBut we did not find results for '
          '{} mathematicians on the list'.format(no_results))
```

That’s it!

When you run the script, you should see at the following report:

Shell

```
The most popular mathematicians are:

Albert Einstein with 1089615 pageviews
Isaac Newton with 581612 pageviews
Srinivasa Ramanujan with 407141 pageviews
Aristotle with 399480 pageviews
Galileo Galilei with 375321 pageviews

But we did not find results for 19 mathematicians on our list
```

Conclusion & Next Steps

Web scraping is a big field, and you have just finished a brief tour of that field, using Python as you guide. You can get pretty far using just `requests` and `BeautifulSoup`, but as you followed along, you may have come up with few questions:

- What happens if page content loads as a result of asynchronous JavaScript requests? (Check out Selenium's Python API.)
- How do I write a web spider or search engine bot that traverses large portions of the web?
- What is this Scrapy thing I keep hearing about?

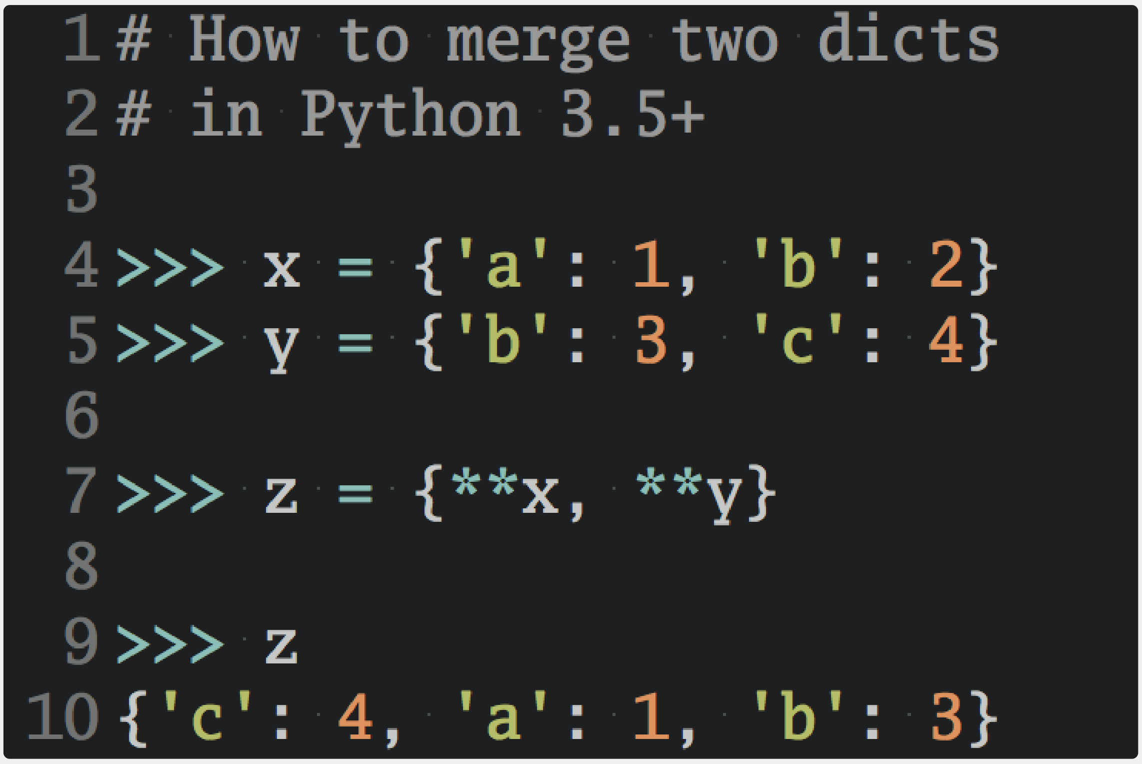
These are topics for another post… Keep your eyes peeled! There will be a followup that uses Selenium and a headless browser to deal with dynamic content:

Don't miss the follow up tutorial: Click here to join the Real Python Newsletter and you'll know when the next installment comes out.

Until then, happy scraping!

☒ Python Tricks ☒

Get a short & sweet **Python Trick** delivered to your inbox every couple of days. No spam ever. Unsubscribe any time. Curated by the Real Python team.



```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

Email Address

Send Me Python Tricks »

About **Colin OKeefe**

Colin is a freelance Software Creative who travels the unixverse in the good ship Python.

» More about Colin

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

What Do You Think?

Keep Learning

Related Tutorial Categories: basics web-scraping

All Tutorial Topics

advanced api basics best-practices community databases data-science devops django docker flask front-end intermediate machine-learning python testing tools web-dev web-scraping

Table of Contents