

Forum:[Hack Forums](#)

Thread: [How to write a web crawler from scratch with Proxy support](#)

Gazo - July 25, 2019, 11:34 pm

Overview

Most Python web crawling/scraping tutorials use some kind of crawling library. This is great if you want to get things done quickly, but if you do not understand how scraping works under the hood then when problems arise it will be difficult to know how to fix them.

In this tutorial I will be going over how to write a web crawler completely from scratch in Python using only the Python Standard Library and the requests module (<https://pypi.org/project/requests/2.7.0/>). I will also be going over how you can use a proxy API (<https://proxyorbit.com>) to prevent your crawler from getting blacklisted.

This is mainly for educational purposes, but with a little attention and care this crawler can become as robust and useful as any scraper written using a library. Not only that, but it will most likely be lighter and more portable as well.

I am going to assume that you have a basic understanding of Python and programming in general. Understanding of how HTTP requests work and how Regular Expressions work will be needed to fully understand the code. I won't be going into deep detail on the implementation of each individual function. Instead, I will give high level overviews of how the code samples work and why certain things work the way they do.

The crawler that we'll be making in this tutorial will have the goal of "indexing the internet" similar to the way Google's crawlers work. Obviously we won't be able to index the internet, but the idea is that this crawler will follow links all over the internet and save those links somewhere as well as some information on the page.

Start Small

The first task is to set the groundwork of our scraper. We're going to use a class to house all our functions. We'll also need the re and requests modules so we'll import them

You can see that this is very simple to start. It's important to build these kinds of things incrementally. Code a little, test a little, etc.

We have two instance variables that will help us in our crawling endeavors later.

Code starting_url

Is the initial starting URL that our crawler will start out

Code visited

This will allow us to keep track of the URLs that we have currently visited to prevent visiting the same URL twice. Using a set() keeps visited URL lookup in O(1) time making it very fast.

Crawl Sites

Now we will get started actually writing the crawler. The code below will make a request to the `starting_url` and extract all links on the page. Then it will iterate over all the new links and gather new links from the new pages. It will continue this recursive process until all links have been scraped that are possible from the starting point. Some websites don't link outside of themselves so these sites will stop sooner than sites that do link to other sites.

As we can see a fair bit of new code has been added.

To start, `get_html`, `get_links`, `crawl`, and `extract_info` methods were added.

Code `get_html()`

Is used to get the HTML at the current link

Code `get_links()`

Extracts links from the current page

Code `extract_info()`

Will be used to extract specific info on the page.

the

Code `crawl()` function has also been added and it is probably the most important and complicated piece of this code. "crawl" works recursively. It starts at the `start_url`, extracts links from that page, iterates over those links, and then feeds the links back into itself recursively.

If you think of the web like a series of doors and rooms, then essentially what this code is doing is looking for those doors and walking through them until it gets to a room with no doors. When this happens it works its way back to a room that has unexplored doors and enters that one. It does this forever until all doors accessible from the starting location have been accessed. This kind of process lends itself very nicely to recursive code.

If you run this script now as is it will explore and print all the new URLs it finds starting from `google.com`

Extract Content

Now we will extract data from the pages. This method (`extract_info`) is largely based on what you are trying to do with your scraper. For the sake of this tutorial, all we are going to do is extract meta tag information if we can find it on the page.

Not much has changed here besides the new print formatting and the `extract_info` method.

The magic here is in the regular expression in the `extract_info` method. It searches in the HTML for all meta tags that follow the format `<meta name=X content=Y>` and returns a Python dictionary of the format `{X:Y}`

This information is then printed to the screen for every URL for every request.

Integrate Rotating Proxy API

One of the main problems with web crawling and web scraping is that sites will ban you either if you make too many requests, don't use an acceptable user agent, etc. One of the ways to limit this is by using proxies and setting a different user agent for the crawler. Normally the proxy approach requires you to go out and purchase or source manually a list of proxies from somewhere else. A lot of the time these proxies don't even work or are incredibly slow making web crawling much more difficult.

To avoid this problem we are going to be using what is called a "rotating proxy API". A rotating proxy API is an API that takes care of managing the proxies for us. All we have to do is make a request to their API endpoint and boom, we'll get a new working proxy for our crawler. Integrating the service into the platform will require no more than a few extra lines of Python.

The service we will be using is Proxy Orbit (<https://proxyorbit.com>). Full disclosure, I do own and run Proxy Orbit. The service specializes in creating proxy solutions for web crawling applications. The proxies are checked continually to make sure that only the best working proxies are in the pool.

As you can see, not much has really changed here. Three new class variables were created: proxy_orbit_key, user_agent, and proxy_orbit_url

proxy_orbit_key gets the Proxy Orbit API Token from an environment variable named "PROXY_ORBIT_TOKEN"

user_agent sets the User Agent of the crawler to Firefox to make requests look like they are coming from a browser

proxy_orbit_url is the Proxy Orbit API endpoint that we will be hitting. We will be filtering our results only requesting HTTP proxies supporting SSL that have been checked in the last 30 minutes.

in get_html a new HTTP request is being made to the Proxy Orbit API URL to get the random proxy and insert it into the requests module for grabbing the URL we are trying to crawl from behind a proxy.

If all goes well then that's it! We should now have a real working web crawler that pulls data from web pages and supports rotating proxies.

If you have any questions either feel free to comment below or send me a PM

lolwtf - July 26, 2019, 10:17 am

Ok nice, some critique.

```
Code    try:
        html = requests.get(url)
    except Exception as e:
        print(e)
        return ""
```

html here is not the HTML, it is the response object. I would use the following:

```
Code    try:
```

```

    resp = requests.get(url, headers={"User-Agent": "Mozilla lol"}, timeout=2)
    resp.raise_for_status()
    html = resp.content
except Exception as e:
    print(e)

```

Also it is good practice to always set an User-Agent and set a timeout else your crawler can freeze forever. In addition, note my use of `raise_for_status` (look it up).

```

Code for i, link in enumerate(links):
    if not urlparse(link).netloc:
        link_with_base = base + link
        links[i] = link_with_base

```

This can be rewritten to:

```

Code for i, link in enumerate(links):
    if urlparse(link).netloc:
        continue

```

```

    link_with_base = base + link
    links[i] = link_with_base

```

This follows the exit-early strategy and does not unnecessarily introduce indents. This is a powerful habit to follow that is relevant for every programming language.

```

Code def get_html(self, url):

```

This function should raise an exception when no HTML is returned so that callees can properly handle errors. You currently return an empty string on error. In addition, there is too much stuff happening in the try/catch...

You can try something like:

Thanks for the tutorial!

Gazo - July 26, 2019, 11:23 am

(Jul 26, 2019 10:17 AM)l0lwtF Wrote: [\[CITE\]](#) Ok nice, some critique.

```

Code    try:
        html = requests.get(url)
    except Exception as e:
        print(e)
        return ""

```

html here is not the HTML, it is the response object. I would use the following:

```

Code    try:
        resp = requests.get(url, headers={"User-Agent": "Mozilla lol"}, timeout=2)
        resp.raise_for_status()
        html = resp.content
    except Exception as e:
        print(e)

```

Also it is good practice to always set an User-Agent and set a timeout else your crawler can freeze forever. In addition, note my use of `raise_for_status` (look it up).

Code for i, link in enumerate(links):

```
if not urlparse(link).netloc:  
    link_with_base = base + link  
    links[i] = link_with_base
```

This can be rewritten to:

Code for i, link in enumerate(links):

```
if urlparse(link).netloc:  
    continue
```

```
link_with_base = base + link  
links[i] = link_with_base
```

This follows the exit-early strategy and does not unnecessarily introduce indents. This is a powerful habit to follow that is relevant for every programming language.

Code def get_html(self, url):

This function should raise an exception when no HTML is returned so that callees can properly handle errors. You currently return an empty string on error. In addition, there is too much stuff happening in the try/catch...

You can try something like:

Thanks for the tutorial!

Great critiques! Definitely agree with everything here. Thank you for taking the time to go through everything.