Web scraping with Python

All Versions

Web scraping is an automated, programmatic process through which data can be constantly 'scraped' off webpages. Also known as screen scraping or web harvesting, web scraping can provide instant data from any publicly accessible webpage. On some websites, web scraping may be illegal.

Examples

Scraping using the Scrapy framework

First you have to set up a new Scrapy project. Enter a directory where you'd like to store your code and run:

```
scrapy startproject projectName
```

To scrape we need a spider. Spiders define how a certain site will be scraped. Here's the code for a spider that follows the links to the top voted questions on StackOverflow and scrapes some data from each page (source):

```
import scrapy

class StackOverflowSpider(scrapy.Spider):
    name = 'stackoverflow'  # each spider has a unique name
    start_urls = ['http://stackoverflow.com/questions?sort=votes']  # the parsing starts from a

    def parse(self, response):  # for each request this generator yields, its response is sent t
        for href in response.css('.question-summary h3 a::attr(href)'):  # do some scraping stu
            full_url = response.urljoin(href.extract())
            yield scrapy.Request(full_url, callback=self.parse_question)

    def parse_question(self, response):
        yield {
            'title': response.css('h1 a::text').extract_first(),
            'votes': response.css('.question .vote-count-post::text').extract_first(),
            'body': response.css('.question .post-text').extract_first(),
            'tags': response.css('.question .post-tag::text').extract(),
            'link': response.url,
        }
```

Save your spider classes in the `projectName\spiders` directory. In this case - `projectName\spiders\stackoverflow_spider.py` .

Now you can use your spider. For example, try running (in the project's directory):

```
scrapy crawl stackoverflow
```

Basic example of using requests and lxml to scrape some data

```
# For Python 2 compatibility.
from __future__ import print_function

import lxml.html
import requests


def main():
    r = requests.get("https://httpbin.org")
    html_source = r.text
    root_element = lxml.html.fromstring(html_source)
    # Note root_element.xpath() gives a *list* of results.
    # XPath specifies a path to the element we want.
    page_title = root_element.xpath('/html/head/title/text()')[0]
    print(page_title)

if __name__ == '__main__':
    main()
```

Maintaining web-scraping session with requests

It is a good idea to maintain a web-scraping session to persist the cookies and other parameters. Additionally, it can result into a *performance improvement* because `requests.Session` reuses the underlying

TCP connection to a host:

```
import requests

with requests.Session() as session:
    # all requests through session now have User-Agent header set
    session.headers = {'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_4) AppleWebKi

    # set cookies
    session.get('http://httpbin.org/cookies/set?key=value')

    # get cookies
    response = session.get('http://httpbin.org/cookies')
    print(response.text)
```

Modify Scrapy user agent

Sometimes the default Scrapy user agent (`"Scrapy/VERSION (+http://scrapy.org)"`) is blocked by the host. To change the default user agent open **settings.py** , uncomment and edit the following line to what ever you want.

```
#USER_AGENT = 'projectName (+http://www.yourdomain.com)'
```

For example

```
USER_AGENT = 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_4) AppleWebKit/537.36 (KHTML, like Ge
```

Scraping using BeautifulSoup4

```
from bs4 import BeautifulSoup
import requests

# Use the requests module to obtain a page
res = requests.get('https://www.codechef.com/problems/easy')

# Create a BeautifulSoup object
page = BeautifulSoup(res.text, 'lxml')   # the text field contains the source of the page

# Now use a CSS selector in order to get the table containing the list of problems
datatable_tags = page.select('table.dataTable')  # The problems are in the <table> tag,
                                                 # with class "dataTable"
# We extract the first tag from the list, since that's what we desire
datatable = datatable_tags[0]
# Now since we want problem names, they are contained in <b> tags, which are
# directly nested under <a> tags
prob_tags = datatable.select('a > b')
prob_names = [tag.getText().strip() for tag in prob_tags]

print prob_names
```

Scraping using Selenium WebDriver

Some websites don't like to be scraped. In these cases you may need to simulate a real user working with a browser. Selenium launches and controls a web browser.

```
from selenium import webdriver

browser = webdriver.Firefox()  # launch firefox browser

browser.get('http://stackoverflow.com/questions?sort=votes')  # load url

title = browser.find_element_by_css_selector('h1').text  # page title (first h1 element)

questions = browser.find_elements_by_css_selector('.question-summary')  # question list

for question in questions:  # iterate over questions
    question_title = question.find_element_by_css_selector('.summary h3 a').text
    question_excerpt = question.find_element_by_css_selector('.summary .excerpt').text
    question_vote = question.find_element_by_css_selector('.stats .vote .votes .vote-count-post'

    print "%s\n%s\n%s votes\n-----------\n" % (question_title, question_excerpt, question_vote)
```

Selenium can do much more. It can modify browser's cookies, fill in forms, simulate mouse clicks, take screenshots of web pages, and run custom JavaScript.

Simple web content download with urllib.request

The standard library module `urllib.request` can be used to download web content:

```
from urllib.request import urlopen

response = urlopen('http://stackoverflow.com/questions?sort=votes')
data = response.read()

# The received bytes should usually be decoded according the response's character set
encoding = response.info().get_content_charset()
html = data.decode(encoding)
```

A similar module is also available in Python 2 .

Scraping with curl

imports:

```
from subprocess import Popen, PIPE
from lxml import etree
from io import StringIO
```

Downloading:

```
user_agent = 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/537.36 (KHTML, like Ge
url = 'http://stackoverflow.com'
get = Popen(['curl', '-s', '-A', user_agent, url], stdout=PIPE)
result = get.stdout.read().decode('utf8')
```

`-s` : silent download

`-A` : user agent flag

Parsing:

```
tree = etree.parse(StringIO(result), etree.HTMLParser())
divs = tree.xpath('//div')
```

Syntax

Parameters

Remarks

Useful Python packages for web scraping (alphabetical order)

Making requests and collecting data

requests

A simple, but powerful package for making HTTP requests.

requests-cache

Caching for `requests` ; caching data is very useful. In development, it means you can avoid hitting a site unnecessarily. While running a real collection, it means that if your scraper crashes for some reason (maybe you didn't handle some unusual content on the site...? maybe the site went down...?) you can repeat the collection very quickly from where you left off.

scrapy

Useful for building web crawlers, where you need something more powerful than using `requests` and iterating through pages.

selenium

Python bindings for Selenium WebDriver, for browser automation. Using `requests` to make HTTP requests directly is often simpler for retrieving webpages. However, this remains a useful tool when it is not possible to replicate the desired behaviour of a site using `requests` alone, particularly when JavaScript is required to render elements on a page.

HTML parsing

BeautifulSoup

Query HTML and XML documents, using a number of different parsers (Python's built-in HTML Parser, `html5lib` , `lxml` or `lxml.html`)

lxml

Processes HTML and XML. Can be used to query and select content from HTML documents via CSS selectors and XPath.