The Art of Webscrapping

Siddhant Sanyam

2011-09-03 Sat

What is Webscrapping

Web scrapping simply means automating the process of

- visiting a website
- copying the data on some temporary storage (think of a clipboard)
- pasting it on some spreadsheet program
- then performing various calculation on it, or simply saving it.

Some Examples

- Our college website provides result to individual student by querying their roll number.
- What if you need the list of all those students who failed in more than 3 subjects?
- The result of an exam is about to be out on a website, how would you track the website for any changes?
- Scrap of all the xkcd's from the website or any other website, which doesn't provide its API
- Fill up tons of form on a website whose data is stored in an spreadsheet
- Making offline app interface, or non-interactive terminal program to interact with a website (whose API is not open)
- There are few real python script related to web scrapping on my Github page

We'll look into all these examples during the talk.

Why web scrapping

- Because it saves time.
- Time is money ;-)
- Since it is a program, it makes less errors than a typical human
- Most applications are moving to web, but we still need desktop software
- It's fun to program a webscraper

Why Python for webscrapping

- Less pain full, by nature
- Language of the internet ~ easy to program internet apps
- Batteries Included ~ preloaded modules to start scrappying on fly
- Get's you started instantly

How will this talk proceed

- We'll start looking up on each area of Webscrapping, from basic to advanced
- Not only you would learn about webscrapping but many fundamental concepts of HTTP and how webserver intracts with web browsers
- Along side, we'll try to do as much examples as possible
- We'll try to solve real world problems, hence any suggestions are invited
- Slides sucks, don't rely on them. I'll be demonstrating most of the things on a screen session, you can connect with ssh on my computer and see everything in close up.
- There is a chat server on my PC, you can connect to it, I'll be posting any links or similar stuff on it.
- Remember, doing few things well is better rather than doing many thing sloppily.

When to Web scrap

- It depends on various factors, which decides when it is favorable to webscrap
- If the scrapper can be developed and can run in the time less than what a human(s) would spend to do the same task
- When information needed is fairly large and you can't risk for a humane mistake

And when not to

- Writing scrappers for a one time task which a human might be able to do in 5 minutes would be a waste of time.
- When the website have captchas. This means that the website doesn't want you to web scrap
- When it is written in the TOS not to.

Elements of webscrapping

- Fetching or crawling
- Parsing
  - Regex
  - Markup Processor
  - Custom Parsers
- Analyzing
- Storing, stateness
- Posting back

Fetching source code of a web location

- UrlOpen
- Fetching with GET Parameters
- Fetching with POST Data
- Adding additional headers

urllib a simple module to fetch remote resources

- Provides a method `urlopen` which can fetch resources from various protocols like http, https(no verification), ftp.
- Opens up a file-like descriptor which allows reading the remote resource
- When web scrapping, it is used to fetch the HTML source of the website.
- Respects proxies.

Simple use of urllib.urlopen

```
import urllib
response_object=urllib.urlopen("http://facebook.com")
s=response_object.read()
print s
```

Passing GET parameters to urlopen

- By default the mode is GET, so you just need to append the parameters to the URL string

```
import urllib
response_object=urllib.urlopen("""
http://en.wikipedia.org/w/index.php?title=Haar-like_feat
s=response_object.read()
print s
```

Using urllib.urlencode

- It allows you to generate your query string based on a dictionary
- Always use urlencode for encoding URL parameters
- It automatically takes care of converting spaces into '+', and other symbols to their hex equivalent

```
params={"title":"Haar-like_features", "oldid":"449167022
encoded_string=urllib.urlencode(params)
baseurl="http://en.wikipedia.org/w/index.php"
fullurl=baseurl + "?" + encoded_string
```

What about POST?

- It is as simple as GET
- Just pass the query string as the second arguement to urlopen

```
query_string=urllib.urlencode({"content":"Paste Content'
f=urllib.urlopen("http://dpaste.com/",query_string)
print f.geturl()
```

Checking the end url incase of redirects

- Webserver might give `301`, `302` or other kind of redirects.
- urllib take cares of that
- In case of redirect, use `geturl()` to get the final url

Example

- Tracking changes in content of a website.
- Why would you do that?
  - Well, some important result is coming online. You don't want to mannually refresh the page and check if something has changed

Let's code it

Example

- Making `dpasteme` a simple script to post data on http://dpaste.com
- Why would you do that?
  - When you have to paste a log while asking help online, copying from terminal and pasting on browser is not cool
- Something like

```
cat lspci|grep USB|dpasteme
http://dpaste.com/614691/
```

That's it for now

- We'll be covering Fetching resources in much detailed way in the later sections
- There is a module called urllib2 which allows you handle URL fetching in much advanced ways
- All that later :)

Parsing

- Once you have fetched a remote location, you need to extract useful information out from it
- Parsing refers to this activity
- You'll have to parse HTML, XML, JS, JSON etc.
- Let's see how we can parse the data

The naive way

- The naive way is to use `str.find` and slice out the strings.
- This method is very basic and works on a small domain of strings
- find, slice, index can be used
- quick, dirty, not robust

Using Regex

- Python have a `re` module
- It is powerful and can be used for screen scrapping
- It can be inherently slow since it is regex
- Although the speed can be improved by compiling the regex
- I'll show you how it is used in one of my script later.

Dedicated Marker Parsers

- The good news - most of the web is HTML
- By nature, HTML is easier to parse, because it is structured.
- Hence you can use some of the dedicated parsers available
- Python comes with HTML, JSON, XML parsers etc.
- Most of the time in Web scrapping, you'll be using HTML parsers to parse the source code

Custom Parsers

- There might be instances when you'll have to write your custom parsers
- These parsers, along with the prebuilt parsers can offer great deal of flexibility
- If you write these parsers, please publish it open source online so that we can use them too

Enter BeautifulSoup

- It is an HTML markup
- It handles bad markup easily
- Handles encoding by converting them to UTF-8
- It is Beautiful, seriously, you'll feel it when you use it

Using Beautiful Soup

- Filling the Soup
- Basic HTML traversal
- find, findAll
- Pretty Print
- Unicode trouble

Filling the soup

```
from BeautifulSoup import BeautifulSoup
html="""
<html><head><title>SimplePage</title></head>
<body>
<pre> Simple Body </pre>
</body>
</html>"""
soup = BeautifulSoup(html)
print soup.prettify()
```

Filling Soup with urllib.urlopen

```
from BeautifulSoup import BeautifulSoup as BS
import urllib
resp = urllib.urlopen("http://facebook.com")
soup = BS(resp)
print soup.prettify()
```

Using BeautifulSoup

We'll check usage by demonstration

Example

Fetching results of list of student from a University website

- Why would you that?
  - Because you want to know the list of all those students who got an ‘F’ in the same subject you got ‘F’.

Crawling

- Crawling basically means using links scrapped from one page as the target for the next scrap
- Examples
  - Generating a dependency tree of ArchLinux using the online database
  - Downloading Manga from mangareader.net
  - Whenever links have some hash value which is not determinable.

When should you crawl

- Whenever you have to
- Crawling means more HTTP requests
- More time spend on the network hence slow script
- Golden Rule: Avoid crawling

Before the next example, urllretrieve

- urlopen is a fine method to fetch resources like HTML, JS, JSON
- Many times you need to download files like Images, Videos, or Binaries
- You have no interest in parsing these files
- urlretrieve is a better option

Why urlretrieve

- It is a function made to ease out downloading of files
- Oversimplified wget
- Allows you to track your download
- Let's take an example to clear things up

Example of simple usage of urlretrieve

```
import urllib
urllib.urlretrieve("http://festember.in/11/images/bartending
```

Example of urlretrieve with progress bar

```
def progress_printer(transferred,block_size,total_size):
    """Prints a progress bar. transferred is the number of b
    block_size is the size of each block. total_size is the
    the file."""
    speed=0
    try:
        speed=float(block_size)/speedtimer.next()/1000
    except ZeroDivisionError:
        pass
    completed=transferred*block_size
    percent=(completed*100)/total_size
    sys.stdout.write("\r%.2f%% complete[%d/%d bytes] @ %.2f
                     (percent,completed,total_size,speed))
    sys.stdout.flush()
```

Example xkcd

Fetching the latest xkcd comic automatically. Given a range

- Why would you do that?
  - Because everyone enjoy reading xkcd. For slow internet connection it is best to keep a local copy

Let's hack this

A slightly advanced example: downloading mangas

- Fetching some manga for offline reading: onemangadl
- http://github.com/siddhant3s/onemangadl
- It downloads mangas (to oversimplify, it's a Japanese comic book)
- Why would you do that?
  - Because I like reading manga's
  - It's an awesome example for web scrapping
- Let's make this

Before the next section, urllib2

- Allows fetching of remote resources with advance mechanisms
- Helps in simulating a real web browser
- Allows, cookies, authentication, persistence, custom headers etc.

Some concepts related to urllib2

- Works on request, response model
- Make a `request` model
- Pass it to urllib2
- Do stuff on the `response` object you received

urlopen

- Also has a urlopen
- Works same way as urllib's urlopen
- Can handle the `request` object

```
import urllib2
response = urllib2.urlopen('http://facebook.com/')
html = response.read()
```

A simple request object

- This object mimics an actual HTTP request

```
import urllib2

req = urllib2.Request('http://facebook.com')
response = urllib2.urlopen(req)
the_page = response.read()
```

- At a simplest level, request object can be supplied with just the URL of the location

Again, the data?

- Sending GET data is same as appending the query string (remember `urllib.urlencode` ?)
- Sending POST data is also simple

```
import urllib, urllib2
query_string=urllib.urlencode({"content":"Paste Content'

req = urllib2.Request(url, query_string)
response = urllib2.urlopen(req)
the_page = response.read()
```

The Fun part, adding Headers

- Many times you need to add additional HTTP headers to your requests
- Like `User Agent`, for convincing web server that you are a browser
- This is the single most powerful feature of urllib2

Adding HTTP headers

```
import urllib
import urllib2

url = 'http://localhost/~siddhant3s/phpinfo.php'
user_agent = 'Mozilla/4.0 (compatible; MSIE 5.5; Windows NT)
values = {'name' : 'Siddhant Sanyam',
          'age' : '20'}

headers = { 'User-Agent' : user_agent }

data = urllib.urlencode(values)
req = urllib2.Request(url, data, headers)
response = urllib2.urlopen(req)
the_page = response.read()
s=the_page[the_page.find("User-Agent ")):]
```

Getting the response headers

- We can use the .info() on the response object to fetch all the header received

```
response.info().keys()
```

Persistence problem

- HTTP was stateless in initial days
- Bad news, now it isn't
- Websites have login and session systems
- HTTP states are maintained by use of headers

Headers

- We know how to send Headers
- We know how to recieved headers
- Problem solved!

Wait, there are better ways in Python

- urllib2 have concepts of `openers` and `handlers`
- They allow to create, use pre-built handlers to handler case of persistence
- We have BasicAuth managers, Cookie managers, Proxy processors
- urllib2 does all this with help of few handlers

Openers and Handlers

- Till now we were using urlopen to open all URLs
- We can make custom openers and choose specific Handlers
- The default handler has ProxyHandler, UnknownHandler, HTTPHandler, HTTPDefaultErrorHandler, HTTPRedirectHandler, FTPHandler, FileHandler, HTTPErrorProcessor preinstalled
- Of course you can add/remove these handlers

Simple Example of HTTPBasicAuth

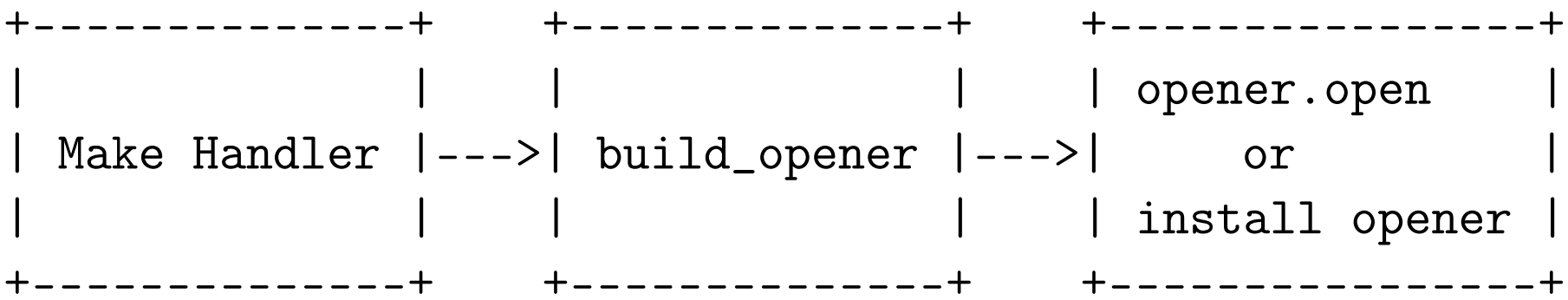
```
# The Password Manager
# It takes care of converting password to the default encodi
password_mgr = urllib2.HTTPPasswordMgrWithDefaultRealm()

# All urls nested after /foo/ will be requested with BasicAu
top_level_url = "http://example.com/foo/"
# Add the username and password.
# If we knew the realm, we could use it instead of None.
password_mgr.add_password(None, top_level_url, username, pas

handler = urllib2.HTTPBasicAuthHandler(password_mgr)

# create "opener" (OpenerDirector instance)
opener = urllib2.build_opener(handler)
```

Using Handlers and Openers



Some useful opener methods

- `opener.open`
- `opener.add_handler`
- `urllib2.install_opener`
- `opener.addheaders`

Some useful Request object methods

- `add_data`
- `get_method`
- `has_data`
- `add_headers`
- `add_unredirected_header`
- `has_header`

Now let's do some login

- To do login and maintain state, you just need to handle cookies
- Cookies are sent in the headers
- Manually processing header works fine
- But **you** have to do it
- Python can do that for you

How Online Sessions work

- Login in a website works simply due to persistent nature of cookies
- When you login, server sends a Session ID in cookies.
- Your browser stores this session ID
- For all further requests (until you logout), the browser sends this session ID back to server
- Web Server recognizes it to be a correct session ID and servers the page.

A sample session with phpinfo

```
import urllib
import urllib2

url = 'http://localhost/~siddhant3s/phpinfo.php'
user_agent = 'Mozilla/4.0 (compatible; MSIE 5.5; Windows NT)
values = {'name' : 'Siddhant Sanyam',
          'age' : '20'}

headers = { 'User-Agent' : user_agent ,'Cookies':'SESSION_ID

data = urllib.urlencode(values)
req = urllib2.Request(url, data, headers)
response = urllib2.urlopen(req)
the_page = response.read()
s=the_page[the_page.find("HTTP_COOKIE"):]
```

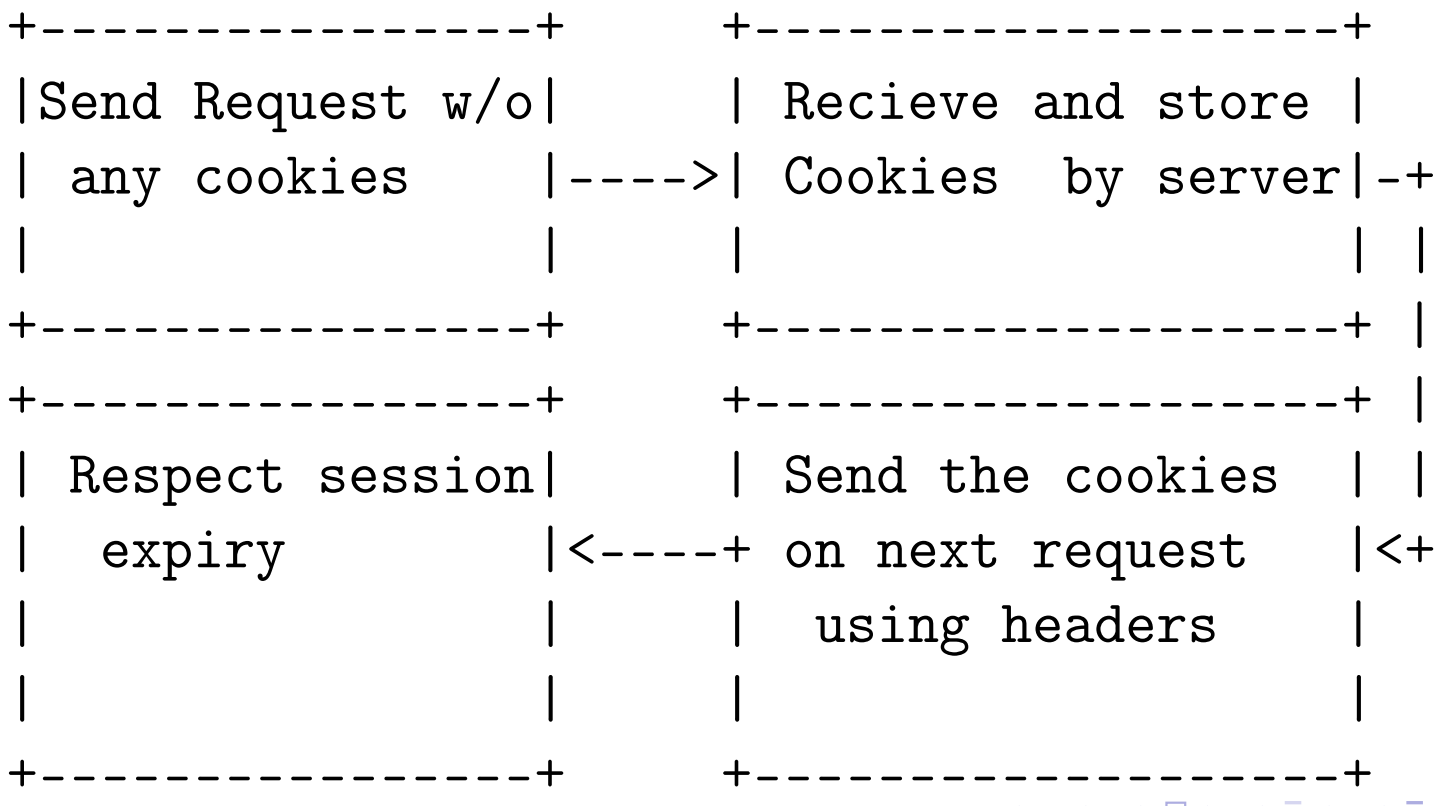
Example of catching cookies

```
import urllib2
request = urllib2.Request("http://localhost/~siddhant3s/
response=urllib2.urlopen(request)
print response.info().keys()
#['x-powered-by', 'transfer-encoding', 'set-cookie', 'ex
'connection', 'pragma'#, 'cache-control', 'date', 'content-t

response.info()['set-cookie']
#'PHPSESSID=k9l406qvsblk6ju2mkkedm1qi5; path=/'
```

The hard way, manage cookies yourself

- I just demonstrated catching and sending cookies
- So the hard way was this
- The basic flow would be like this



The easy way, enter HTTPCookieProcessor

- urllib2 has a handler called HTTPCookieProcessor
- It can maintain and manage cookies between subsequent requests
- Hence you can simulate a logged in user

Few concepts

- CookieJar : an object which has information about cookies
- HTTPCookieProcessor : urllib2 Handler which handles cookies and save it to a CookieJar
- While making request, HTTPCookieProcessor sends any cookies in CookieJar
- While recieving a response, HTTPCookieProcessor stores cookie in the CookieJar

Example, login in a website

```
import urllib, urllib2, cookielib
logindata = urllib.urlencode({'MobileNoLogin':'9876543210',
req = urllib2.urlopen("http://fullonsms.com/home.php")
print req.url
req = urllib2.urlopen("http://fullonsms.com/login.php",logir
req = urllib2.urlopen("http://fullonsms.com/home.php")
print req.url

#Now with cookiemanager
cj = cookielib.CookieJar()
cookiehandler = urllib2.HTTPCookieProcessor(cj)
opener = urllib2.build_opener(cookiehandler)
opener.open("http://fullonsms.com/login.php",logindata)
req = opener.open("http://fullonsms.com/home.php")
print req.url
```

Taking persistance to a level ahead

- CookieJar is stored in Memory
- We need a cookie jar which can be saved to disk
- Advantage: no need of logging in next time

Do it by Hand

- Solution is simple, serialize the CookieJar
- Save it in a file
- Load the file
- And unserialize it

Do it by Python

- `cookielib` has a persistant cookiejar
- It tries to mimic Firefox, Thunderbird etc. in storing the cookies
- It is called MozillaCookieJar
- With this, you can truly be carefree about persistance

The MozillaCokieJar

- Simple to use, this is how you initialize it

```
import cookielib
jar=cookielib.MozillaCookieJar("/path/to/cookie/file")
jar.load() #load the cookies, file must exist
# Use your jar like you would
jar.save()
```

Let's take a complete example including HTTPCookieProcessor

```
import urllib,urllib2,cookielib
filecookiejar = cookielib.MozillaCookieJar(os.path.expanduse
try:
    filecookiejar.load()
except:
    # File not found, make HTTPCookieProcessor w/o any initi
    cookieprocessor=urllib2.HTTPCookieProcessor()
else:
    # File found, cookies loaded from file
    cookieprocessor=urllib2.HTTPCookieProcessor(filecookieja

o = urllib2.build_opener( cookieprocessor )
o.open("http://localhost/~siddhant3s/11")
#some more requests
```

Continued.... saving cookies back to file

```
#Now save the cookies
cookiejar=cookieprocessor.cookiejar
cookienum=enumerate(cookiejar).next()
#Save all cookies
for x in cookieenum:
    cookie = x[1]
    filecookiejar.set_cookie(cookie)

#now save to file
filecookiejar.save()
```

Now let's do some serious shit

- There is fullonsms.com which allows sending 160char full SMSs
- We'll try to make a utility which allows us to send message using terminal
- Why?
  - Because it is faster, I don't need to go the website, close down popups
- WARNING: For educational use only ;)

More parsers

- Python has a beautiful module for parsing JSON
- Useful for the website which have JSON APIs
- XML parser is also there, sometimes you might find it useful
- BeautifulSoup will work most of the time, other heuristic based parsers are there

Use correct parser

- Most of the off-network time on of your program would be spend in parsing
- Choosing good parser is important
- Please use a markup parser if the document is structured in a markup
- Regex should be the last resort

Nasty server checks

- Many webservers would be anti scrappers
- They might impose some heuristic way of checking if you are a bot
- If that is captchas, you really can't do anythign about it
- In most of the other cases, you can fool the webserver
- UserAgent check is very popular webserver check
- You may have to modify different headers in your HTTP Request
- LiveHTTPHeader is your friend

AJAX

- New website have dynamic content loading with AJAX
- Here, you have to closely observe how the AJAX call works
- Dicipher them into simple HTTP requests
- FireBug is your friend in this case

Crawling

- Avoid crawling when you can get the link at one place
- Even if you have to use some slow parsing
- Offline determination of new links to scrap is better than crawling

Thank You

- For any further queries you can write to me at `siddhant3s at gmail dot com`
- I'm usually on IRC ##python at Freenode
- Thank you for your time, thanks for listening