

Getting data

The problem!

Data exists in different sources and different formats
and we have to work with whatever format we get

or

We go to data analysis with data in the format we have,
not the format we want!

Sources of data

local data

csv files
pdf files
xls files

web data

json
xml
html

database servers

mysql
postgres
mongoDB

RESTful Web Services

REST: Representational State Transfer

“A network of web pages connected through links and HTTP commands (GET, POST, etc.)”

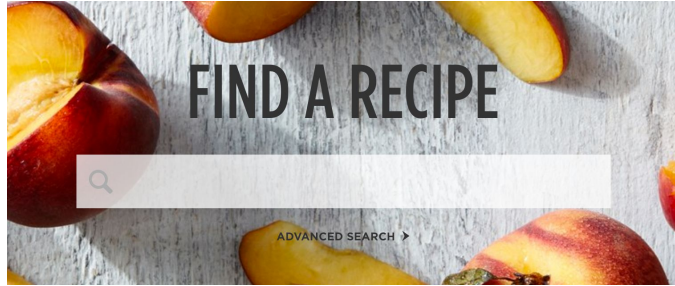
RESTful: A web service that conforms to the REST standards

RESTful Web Services

URLs: RESTful Web Services deliver resources to the client. Each resources (html, json, image, etc.) is associated with a URL and an HTTP method

RESTful: A web service that conforms to the REST standards

Example: Epicurious



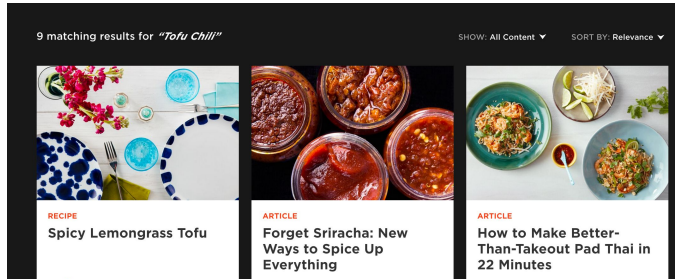
<http://www.epicurious.com>

type "Tofu Chili" in search
box

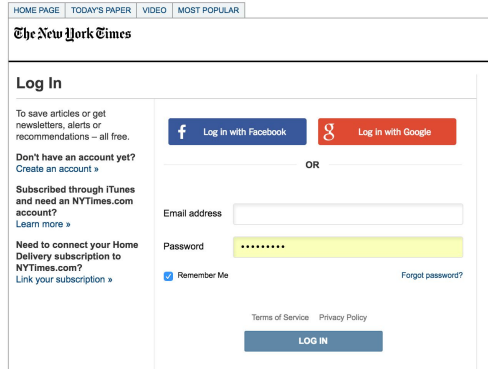
http GET

new url:

[http://www.epicurious.com/
arch/Tofu%20Chili](http://www.epicurious.com/search/Tofu%20Chili)



Example: NYTIMES login



The screenshot shows the New York Times login page. At the top, there are navigation links: HOME PAGE, TODAY'S PAPER, VIDEO, and MOST POPULAR. Below this is the New York Times logo. The main heading is "Log In". On the left, there are three sections of text: "To save articles or get newsletters, alerts or recommendations - all free.", "Don't have an account yet? Create an account >", and "Subscribed through iTunes and need an NYTimes.com account? Learn more >". Below these is a section titled "Need to connect your Home Delivery subscription to NYTimes.com? Link your subscription >". On the right, there are two buttons: "Log in with Facebook" and "Log in with Google". Below these is an "OR" separator. Then, there are input fields for "Email address" and "Password". The password field is highlighted in yellow. Below the password field is a checkbox labeled "Remember Me" and a link "Forgot password?". At the bottom of the form is a "LOG IN" button. Above the "LOG IN" button are links for "Terms of Service" and "Privacy Policy".

← <https://myaccount.nytimes.com/auth/login>

← type Email address and Password in the form

↓ http POST

← new url:

<http://www.nytimes.com/>



Example: Google GEOCODING API

HTTP GET request with
a JSON response

https://maps.googleapis.com/maps/api/geocode/json?address=Columbia_University,_New_York,_NY

All google API requests take the form:

`<api_url>/<response_type>?<parameters>`

json (or xml)

address=Columbia_University,_New_York,_NY

`https://maps.googleapis.com/maps/api/geocode/`

What we need

The ability to

- * create and send HTTP requests
- * receive and process HTTP responses
- * convert data residing in JSON/XML/HTML format into python objects

Python libraries for getting web data

- * Send an http request and get an http response
 - * requests
 - * urllib.requests (urllib2 on python2)
- * parse the response and extract data
 - * json
 - * lxml
 - * BeautifulSoup, Selenium (for html data)

http requests

requests: Python library for handling http requests and responses

<http://docs.python-requests.org/en/master/>

using requests

- * Import the library

```
import requests
```

- * Construct the url

```
url = "http://www.epicurious.com/search/Tofu+Chili"
```

- * Send the request and get a response

```
response = requests.get(url)
```

- * Check if the request was successful

```
if response.status_code == 200:
```

```
    "SUCCESS"!!!!
```

```
else:
```

```
    "FAILURE"!!!
```

response status codes

- * 200 or 201

the request response cycle worked as planned

- * Other 200s

the request response cycle worked but there is additional information associated with the response

- * 400s

there was an error (page not found/malformed request/etc.)

- * General rule of thumb

check if the status code was 200 for accessing data through a GET or POST HTTP request

- * `response.content` **response content**
returns the content of the HTTP response
- * `response.content.decode('utf-8')`
if the content is byte encoded (which it usually is!),
converts it into unicode - a python str
- * **What is unicode?**
<http://unicode.org/standard/WhatIsUnicode.html>
- * **General rule of thumb**
web pages are usually returned as byte strings and need
to be decoded. utf-8 is the usual decoding (but not always!)
- * **What is utf-8?**
https://www.w3schools.com/charsets/ref_html_utf8.asp

Try this

1. Open https://en.wikipedia.org/wiki/main_page using the requests library
2. Check the status code. Did your request work?
3. Get the content. Decode it. Then search the page for the string “Did you know” using the str find function
4. If your find function returned a positive number - Great!
5. If it returned -1 (that means it was not found), you’ve done something wrong. Try figuring out what went wrong

web data formats

- * HTML

the common format when scraping web pages for data

- * JSON or XML

usually when accessing data through an API or when the server is explicitly sharing data with you

JSON

JavaScript Object Notation

- Standard for "serializing" data objects for storage or transmission
- Human-readable, useful for data interchange
- Also useful for representing and storing semistructured data
- Stored as plain (byte strings or utf-8 strings) text

JSON constructs and Python equivalents

JSON	Python
number	int,float
string	str
Null	None
true/false	True/False
Object	dict
Array	list

python json library

`json.loads(<str>)`: converts a JSON string to python objects

`json.dumps(<python_object>)`: converts a python object into a JSON formatted string

python json library

Converts a json string to an equivalent
python type

```
import json
str → json_data = '[{"b": [2, 4], "c": 3.0, "a": "A"}]'
python_data = json.loads(json_data)
list ←
```

python json library

Converts a json string to an equivalent python type

```
str → import json  
data_string = json.dumps(python_data)  
                                ↑  
                                list
```

requests and json

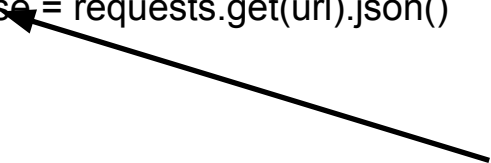
The response object handles json

```
address="Columbia University, New York, NY"  
url="https://maps.googleapis.com/maps/api/geocode/json?address=%s" % (address)  
response = requests.get(url).json()
```

the requests library handles
spaces in a url for you



response now points to a
python data object



requests and json

requests.json returns an exception if the
JSON object is ill-formed

note that some http errors are returned as
JSON

always check for exceptions!

requests and json

```
address="Columbia University, New York, NY"
url="https://maps.googleapis.com/maps/api/geocode/json?address=%s" % (address)
try:
    response = requests.get(url)
    if not response.status_code == 200:
        print("HTTP error",response.status_code)
    else:
        try:
            response_data = response.json()
        except:
            print("Response not in valid JSON format")
except:
    print("Something went wrong with requests.get")
print(type(response_data))
```


requests and json: example

Let's take a look at the JSON object returned by Google Geocoding API

Working with json

Problem 1: Write a function that takes an address as an argument and returns a (latitude, longitude) tuple

```
def get_lat_lng(address_string):  
    #python code goes here
```

Solution to problem 1

```
def get_lat_lng(address):
    url="https://maps.googleapis.com/maps/api/geocode/json?address=%s"%(address)
    import requests
    response = requests.get(url)
    if response.status_code == 200:
        lat = response.json()['results'][0]['geometry']['location']['lat']
        lng = response.json()['results'][0]['geometry']['location']['lng']
        return lat,lng
    else:
        return None
```

xml

Extensible Markup Language

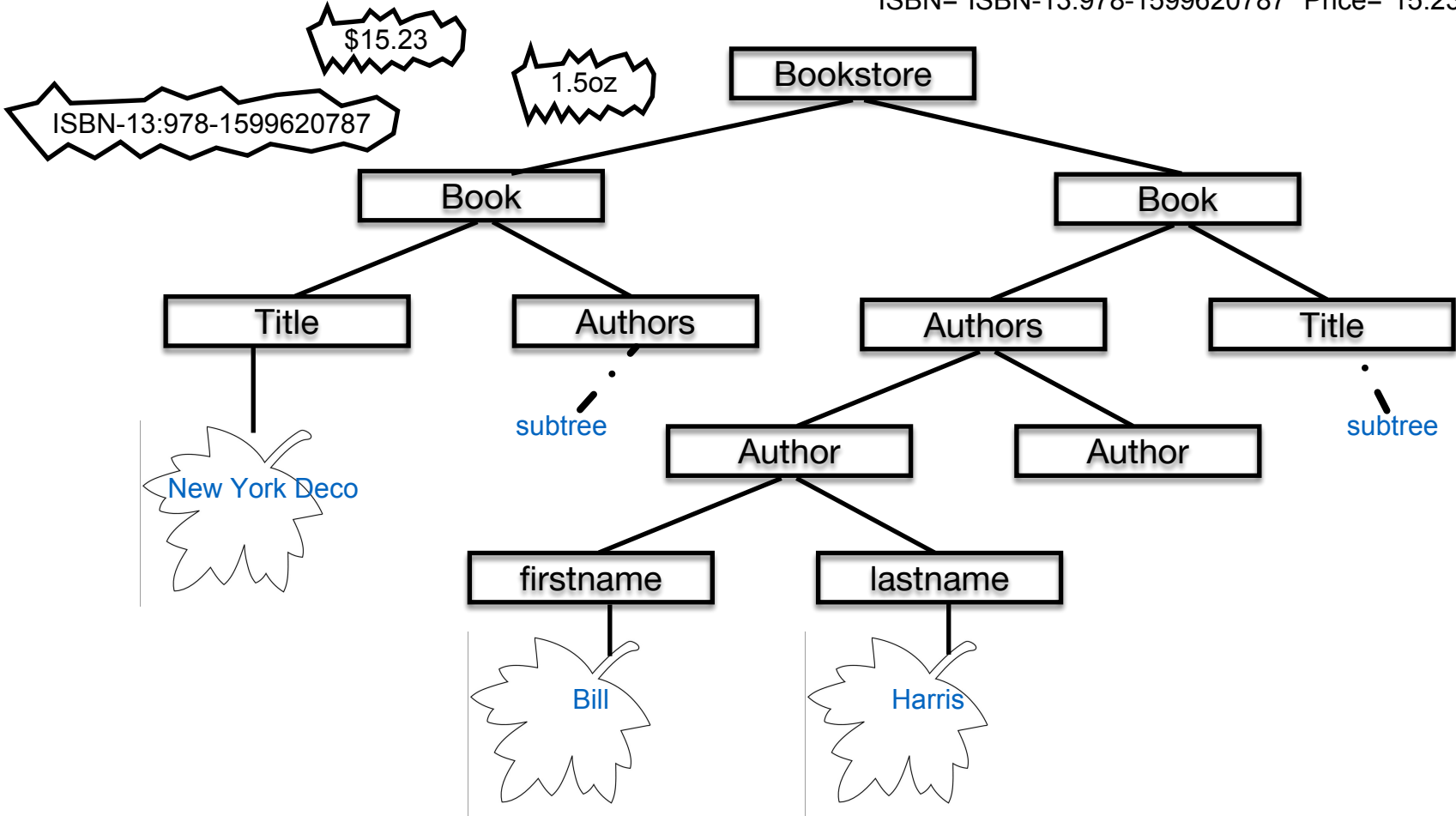
- Tree structure
- Tagged elements (nested)
- Attributes
- Text (leaves of the tree)

xml: Example

```
<Bookstore>
  <Book ISBN="ISBN-13:978-1599620787" Price="15.23" Weight="1.5">
    <Title>New York Deco</Title>
    <Authors>
      <Author Residence="New York City">
        <First_Name>Richard</First_Name>
        <Last_Name>Berenholtz</Last_Name>
      </Author>
    </Authors>
  </Book>
  <Book ISBN="ISBN-13:978-1579128562" Price="15.80">
    <Remark>
      Five Hundred Buildings of New York and over one million other books are available for Amazon Kindle.
    </Remark>
    <Title>Five Hundred Buildings of New York</Title>
    <Authors>
      <Author Residence="Beijing">
        <First_Name>Bill</First_Name>
        <Last_Name>Harris</Last_Name>
      </Author>
      <Author Residence="New York City">
        <First_Name>Jorg</First_Name>
        <Last_Name>Brockmann</Last_Name>
      </Author>
    </Authors>
  </Book>
</Bookstore>
```

XML Tree

ISBN="ISBN-13:978-1599620787" Price="15.23" Weight="1.5"

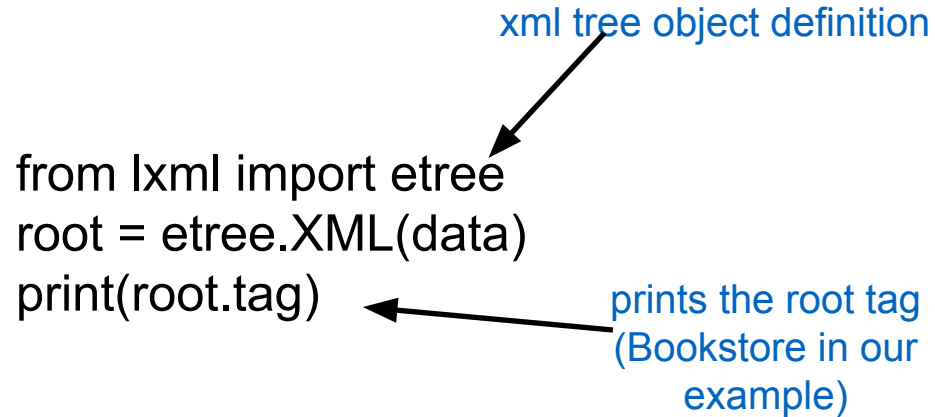


lxml: Python xml library

```
from lxml import etree
root = etree.XML(data)
print(root.tag)
```

xml tree object definition

prints the root tag
(Bookstore in our
example)



<http://lxml.de/1.3/tutorial.html>


lxml: Python xml library

Examining the tree

```
print(etree.tostring(root, pretty_print=True).decode("utf-8"))
```


lxml: Iterating over children of a tag

root is a collection of
children so we can
iterate over it

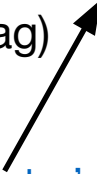


```
for child in root:  
    print(child.tag)
```

lxml: Iterating over elements

```
for element in root.iter():  
    print(element.tag)
```


iter is an 'iterator'. it
generates a sequence of
elements in the order
they appear in the xml
code



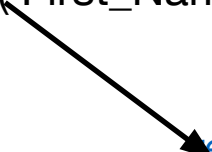
lxml: Iterating over elements

```
for element in root.iter("Author"):
    print(element.find('First_Name').text, element.find('Last_Name').text)
```

iterates over the tree
matching only those
elements that have
“Author” as the tag



returns the first element
that has “First_Name”
as the tag



lxml: using XPath

XPath: expression for
navigating through an
xml tree

```
for element in root.findall('Book/Title'):  
    print(element.text)
```



Try this

Find the last names of all authors in the tree “root” using path

Solution

```
for element in root.findall('Book/Authors/Author/Last_Name'):  
    print(element.text)
```

lxml: Finding by attribute value

```
root.find('Book[@Weight="1.5"]/Authors/Author/First_Name').text
```

Problem 3

Print first and last names of all authors who live in
New York City

HTML

HyperText Markup Language

- Formats text
- Tagged elements (nested)
- Attributes
- Derived from SGML (but who cares!)
- Closely related to XML
- Can contain runnable scripts

HTML/CSS

Study this on your own!

Make sure you've reviewed the first two topics ("Intro to html" and "Intro to CSS") on Khan Academy

<https://www.khanacademy.org/computing/computer-programming/html-css>

getting data from the
web

creeping, crawling, pouncing!

Web scraping: Automating the process of extracting information from web pages

- * for data collection and analysis
- * for incorporating in a web app

APIs (Application Programming Interface): Functions and libraries for communicating with specific web servers

- * for data collection and analysis
- * for incorporating in a web app

Web crawling: Automating the process of traversing links on web pages

- * for indexing the web
- * for collecting data from multiple websites

Legal and ethical issues

- ➡ Often against the 'Terms of Use' of a web site
- ➡ but, regardless, murky and not fully settled
- ➡ See: https://en.wikipedia.org/wiki/Web_scraping#Legal_issues
- ➡ probably depends upon three things:
 - ⊙ factual, non-proprietary data is generally ok
 - ⊙ proprietary data scraping depends on what you do with it
 - ⊙ potential or actual damage to the scrapee
- ➡ Public vs. private information. Public information is rarely unethical
- ➡ Purpose. Ethical or unethical depends on why you're scraping the web site
- ➡ Always better to try and get the information openly using APIs or contacting the owner of the server
- ➡ Is there a public interest involved? If yes, it's probably ethical to scrape

Libraries for web scraping

requests: Python library for connecting to a web page, managing the connection and retrieving contents of the page

Beautiful Soup: A library that utilizes the 'tag structure' of an html page to quickly parse the contents of a page and retrieve data

Selenium: A library that utilizes the 'tag structure' of an html page to execute javascript scripts on the page and retrieve data.
Slower than Beautiful Soup but gets around the 'javascript' problem

BeautifulSoup4

- ➡ HTML (and XML) parser
- ➡ Uses 'tags'
- ➡ Creates a parse tree (using lxml/html5lib or other python parser)
- ➡ Can handle incomplete tagging
- ➡ tags are organized in hierarchical dictionaries

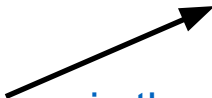
<https://www.crummy.com/software/BeautifulSoup/bs4/doc/>

bs4

initialize bs4 object: BeautifulSoup(document,parser)
parser: lxml (fast) or html5lib (slower but more robust)

```
import requests
from bs4 import BeautifulSoup
url = "http://www.epicurious.com/search/Tofu%20Chili"
response = requests.get(url)
page_soup = BeautifulSoup(response.content,'lxml')
print(page_soup.prettify())
```

page_soup is the object from
which we will extract the
data we need



Unique data identifiers

- ➡ We want to create a list of recipes and links to the recipes
- ➡ We need to figure out how to 'programmatically' extract each recipe name and recipe link
- ➡ Search for the tag with a unique attribute value that identifies recipes and recipe links
- ➡ The easiest way is to examine the page source on a browser

Finding unique data identifiers

```
for tag in page_soup.find_all('article'):  
    print(tag.get('class'))
```

This gets the innermost tags with the recipe name.

looks like class='recipe-content-card' gives us the recipes

['article-content-card']

['gallery-content-card']

['recipe-content-card']

['article-content-card']

['recipe-content-card']

['recipe-content-card']

['recipe-content-card']

['article-content-card']

['recipe-content-card']

['recipe-content-card']

['recipe-content-card']

['recipe-content-card']

['article-content-card']

['article-content-card']

['recipe-content-card']

prints

bs4 functions

`<tag>.find(<tag_name>,attribute=value)`

finds the first matching child tag (recursively)

`<tag>.find_all(<tag_name>,attribute=value)`

finds all matching child tags (recursively)

`<tag>.get_text()`

returns the marked up text

`<tag>.parent`

returns the (immediate) parent

`<tag>.parents`

returns all parents (recursively)

`<tag>.children`

returns the (direct) children

`<tag>.descendants`

returns all children (recursively)

`<tag>.get(attribute)`

returns the value of the specified attribute

`<tag>.name`

returns the name of a tag

`<tag>.attrs`

returns all the attributes of a tag

Problem 4: Extract recipes and recipe links

Write a function `epicurious_recipes(search_string)` that returns the list of recipes and links associated with `search_string`

Call the function with a `search_string`, open the link associated with the first recipe, then return the ingredients and preparation instructions associated with that link

Problem 4: Step 1

given a recipe url, get the description and the ingredients

```
def get_recipe_detail(url):  
    import requests  
    from bs4 import BeautifulSoup  
    html_data = requests.get(url)  
    if not html_data.status_code == 200:  
        return "", []  
    recipe_data = BeautifulSoup(html_data.content, 'lxml')  
    description = get_description(recipe_data)  
    ing_list = get_ingredients(recipe_data)  
    return description, ing_list
```

need to write these two functions



Problem 4: Step 2

write the `get_description` function

```
def get_description(recipe_page_data):  
    description_tag = recipe_page_data.find('div',itemprop = 'description')  
    if description_tag:  
        return description_tag.get_text()  
    return ""
```

Problem 4: Step 3

write the `get_ingredients` function

```
def get_ingredients(ing_page_data):  
    ing_list = list()  
    for item in ing_page_data.find_all('li', class_='ingredient'):  
        ing_list.append(item.get_text())  
    return ing_list
```


Problem 4: Step 4

write the function that:

- takes key words as an argument

- and returns a list of tuples

 - (name,description,ingredients)

Problem 4: Step 4 code

```
def get_recipes(keywords):
    recipe_list = list()
    import requests
    from bs4 import BeautifulSoup
    url = "http://www.epicurious.com/search/" + keywords
    response = requests.get(url)
    if not response.status_code == 200:
        return None
    try:
        results_page = BeautifulSoup(response.content, 'lxml')
        recipes = results_page.find_all('article', class_="recipe-content-card")
        for recipe in recipes:
            recipe_link = "http://www.epicurious.com" + recipe.find('a').get('href')
            recipe_name = recipe.find('a').get_text()
            try:
                recipe_description = recipe.find('p', class_='dek').get_text()
            except:
                recipe_description = ""
            recipe_list.append((recipe_name, recipe_link, recipe_description))
        return recipe_list
    except:
        return None
```

logging in with requests and beautifulsoup

➡ Figure out the login url

➡ <https://en.wikipedia.org/w/index.php?title=Special:UserLogin&returnto=Main+Page>

➡ Look for the login form in the html source

➡ `form_tag = page_soup.find('form')`

➡ Look for ALL the inputs in the login form (some may be tricky!)

➡ `input_tags = form_tag.find_all('input')`

➡ Create a Python dict object with key,value pairs for each input

➡ Use `requests.session` to create an open session object

➡ Send the login request (POST)

➡ Send followup requests keeping the sessions object open

Setting up the inputs

```
payload = {  
    'wpName': username,  
    'wpPassword': password,  
    'wploginattempt': 'Log in',  
    'wpEditToken': "+\\",  
    'title': "Special:UserLogin",  
    'authAction': "login",  
    'force': "",  
    'wpForceHttps': "1",  
    'wpFromhttp': "1",  
    '#wpLoginToken': "", #We need to read this from the page  
}
```

Extracting token information

wpLoginToken: the value of this attribute is provided by the page. we need to extract it.

```
login_page_response =  
s.get('https://en.wikipedia.org/w/index.php?title=Special:UserLogin&returnto=  
=Main+Page')  
soup = BeautifulSoup(login_page_response.content, 'lxml')  
token = soup.find('input', {'name': "wpLoginToken"}).get('value')
```

Finalizing session parameters

username=<your username>

password=<your password>

```
def get_login_token(response):
```

```
    soup = BeautifulSoup(response.text, 'lxml')
```

```
    token = soup.find('input',{'name':"wpLoginToken"}).get('value')
```

```
    return token
```

```
payload = {
```

```
    'wpName': username,
```

```
    'wpPassword': password,
```

```
    'wploginattempt': 'Log in',
```

```
    'wpEditToken': "+\\",
```

```
    'title': "Special:UserLogin",
```

```
    'authAction': "login",
```

```
    'force': "",
```

```
    'wpForceHttps': "1",
```

```
    'wpFromhttp': "1",
```

```
    #'wpLoginToken': '',
```

```
}
```

Activating session

with requests.session() as s:

```
response = s.get('https://en.wikipedia.org/w/index.php?title=Special:UserLogin&returnto=Main+Page')
```

```
payload['wpLoginToken'] = get_login_token(response)
```

```
#Send the login request
```

```
response_post = s.post('https://en.wikipedia.org/w/index.php?title=Special:UserLogin&action=submitlogin&type=login',  
                        data=payload)
```

```
#Get another page and check if we're still logged in
```

```
response = s.get('https://en.wikipedia.org/wiki/Special:Watchlist')
```