Chapter 9

Scraping Using Regular Expressions

Skills you will learn: Using the Python re module to scrape a simple website.

Introduction: In Chapter 9 of *The Data Journalist*, we learned how to scrape a simple website using the Beautiful Soup module to parse the html page returned by a web server. As with many tasks, there's more than one way to do a job, and this tutorial explores scraping using regular expressions.

Regular expressions are a powerful text searching language. Originally developed as part of Perl, regular expressions allow for sophisticated extraction of elements from arbitrary text. Especially in web pages that are poorly structured, regular expressions offer a powerful alternative for pulling out the elements you want. This tutorial walks through a straightforward scrape using regex, as regular expressions are often called, using the Python re module.

Getting Started

Once you've got Python up and running, create a new script file in your text editor and save it as regexScraper.py.

At the top, we'll import the modules we'll use in our code.

```
import urllib2, re, time
```

It's a good idea to store the address of the website we want to scrape as a variable called url. We're going to scrape the website of the Governor General of Canada, and in particular the part that allows users to search for recipients of various national honours. It can be found at http://www.gg.ca.

```
url = "http://www.gg.ca"
```

Next, we'll tell urllib2 to open the address stored in the url variable and read it. This can be done in a multi-step process or with one line of code that stores the read page in a variable that we'll call the_page

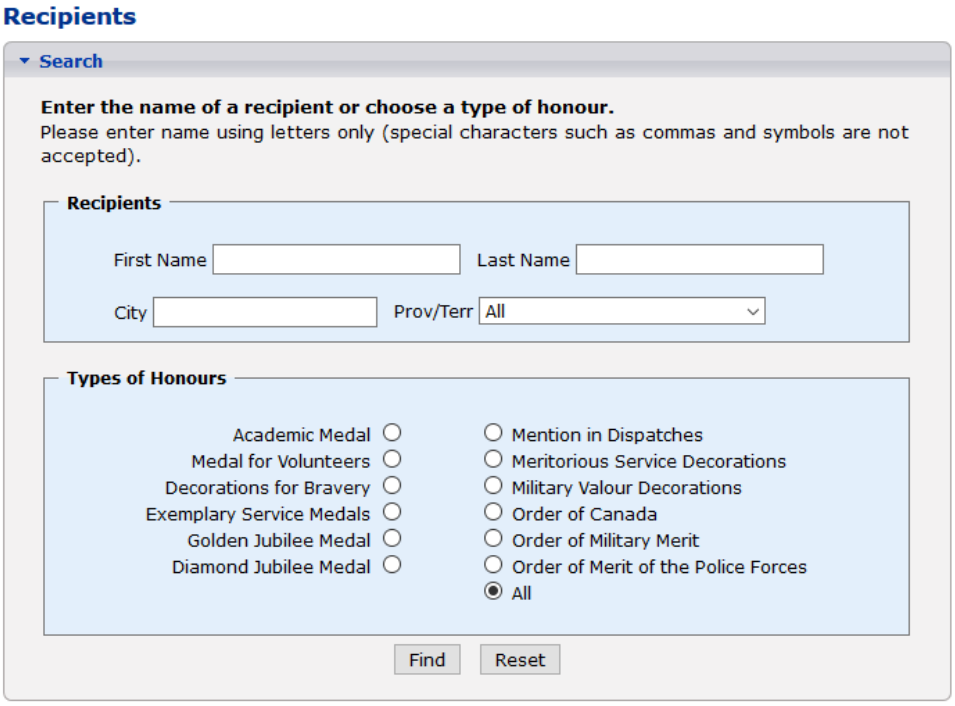
```
the_page = urllib2.urlopen(url).read()
```

Add a line to print out this variable on screen to see what the script downloaded. The full script so far should look like this:

```
import urllib2, re, time
url = "http://www.gg.ca"
the_page = urllib2.urlopen(url).read()
print the_page
```

You should see a stream of HTML coded text. This is the raw data behind the Governor General's webpage. If you bring up the same page in a web browser, we can see a link to the section of the website about Honours, the awards that the Governor General gives out.

Clicking that link brings us to a page about honours. Further clicking the Find a Recipient link brings us to a page with a searchable database that we can scrape.



Let's change the address in the url variable so it points to this database search screen. (The URL has an additional variable that is telling the web server we want the data in English.

```
url = "http://www.gg.ca/honours.aspx?lan=eng"
```

Before we proceed, let's go back to a web browser and explore the way the search interface to the honours database works. By submitting sample searches, we find that it will allow us to search by name and type of recipient, and also that it will allow us to enter a blank search, with no criteria for the search.

A blank search generates more than 300,000 results -- far more than we want. Instead, let's filter it by clicking on the radio button for Order of Canada. That narrows the list to more than 6,000 records, a manageable size for scraping.

The URL displayed by the browser that generates these results looks like this:

http://www.gg.ca/honours.aspx?ln=&fn=&t=12&p=&c=&pg=1&types=12

That means the web developer has set up the search engine so that the "types" variable indicates which of the awards we want to search and chose the number 12 to represent Order of Canada recipients. If we change the two variables with the value of 12 in the URL to, for example, 10, we'll instead see a list of those who received Military Valour Decorations.

If you look at the search results for Order of Canada recipients in the browser, you'll notice that it displays only 50 of the more than 6,000 records. The results have been paginated. We can click on the the "Next" link in the lower right corner.

That brings up the next page of search results, with the next 50 names. The URL is:

http://www.gg.ca/honours.aspx?t=12&types=12&pg=2

So we now know that we can add a variable called "pg" to the search URL and work our way through each page of results by simply increasing the value of this variable. Let's go back to our script.

We'll change the base URL in the url variable to the address that brings up the list of Order of Canada recipients, but we'll leave off value representing the page number for now:

```
url = "http://www.gg.ca/honours.aspx?t=12&types=12&pg="
```

Next, we'll create a variable called "counter" that will represent the number of each page of results, and set up a loop to do this over and over. For now, let's run this loop only five times.

We'll also need a line of code that will add the increasing page number to the full URL. One complication here: Python can't add a number to a string of text. The counter variable will have to be converted to a text string before it's put on the end of the URL, using the built-in Python str() function.

The full script should look like this:

```
import urllib2, re, time
counter =1
while counter < 6:
    url =
"http://www.gg.ca/honours.aspx?t=12&types=12&pg=" +
str(counter)
    the_page = urllib2.urlopen(url).read()
    print the_page
```

The script should print out the raw HTML of the first five pages -- 250 names -- of Order of Canada recipients.
This data isn't, however, in a useable format. We need to extract the name and city of each of these recipients. If we use View Source in a browser to look at the raw data, we see that the name of each recipient is buried in the HTML and separated out with HTML tags:

```
<td>
 <a href='/honour.aspx?id=9934&t=12&ln=Adams'>Miriam
Adams</a>
 </td>

<td>
Toronto, Ontario
</td>
```

We are going to extract this data using regular expressions via the Python re module. We'll create a function to extract the data we need from each page. We need to give it a name when we define it -- "name_extractor"-- and tell it the name of the variable the rest of the program is going to send it.

```
def name_extractor(the_page):
```

Having the data we want to extract on separate lines is a headache, so we want to use Regular Expressions to remove all the line breaks in the HTML file. Depending on the operating system used when the file was created, line breaks are encoded as \r" or "\n". We'll add these two lines to the name_extractor function. Make sure to indent them one tab or four spaces to the right, in comparison to the def line.

```
the_page = re.sub("\r", "", the_page)
the_page = re.sub("\n", "", the_page)
```

These lines update the value of the the_page varible and use the Regular Expression "sub" command to substitute an empty text string "" wherever there is a "\r" or "\n" line break in the the_page variable.

Next, we'd like to take out all those messy spaces in the HTML code, because they also make it hard to find exact patterns in the text. Again, we'll use Regular Expressions:

```
the_page = re.sub(" +", " ", the_page)
```

This command searches for a blank space and because there is a plus sign after it, will grab any number of consecutive blank spaces. It then substitutes a single blank space.

So, the full function in the script should now look like this:

```
def name_extractor(the_page):
      the_page = re.sub("\r", "", the_page)
      the_page = re.sub("\n", "", the_page)
      the_page = re.sub(" +", " ", the_page)
```

Next, we'll add a command to the function that will find the name and city and province of each recipient in this cleaned-up HTML data. The pattern we're looking for in the HTML appears like this:

```
<td> <a href='/honour.aspx?id=9934&t=12&ln=Adams'>Miriam
Adams</a> </td> <td> Toronto, Ontario </td>
```

Each of the 50 names listed on each page will follow this pattern. This is the tricky part. We have to use a Regular Expression wildcard that is represented by .+? to match all the parts of that HTML pattern that change for every name. The period represents any character, the plus sign means that character can be repeated one or more times and the ? character ensures that the expression matches the pattern only once (and so doesn't keep on matching to the end of the line).

And because we'll searching for all 50 names in each page that follow this pattern, we'll have to find every instance of it. To do this, we'll use the Regular Expression "find iteration" function and add these lines to the function we are defining:

```
for each in re.finditer('<a
href="/honour\.aspx\?id=.+?>(.+?)</a>\s?</td>\s?<td>\s?(.+?
)\s?</td>', the_page): recipient = each.group(1)
 city_prov = each.group(2)
```

These lines of code look for each instance of the pattern in the HTML, then, pull out the part that contains the recipient's name and store it, then find the part that contains the city and province, and store it too.

It then assigns those to two pieces of information to variables called recipient and city_prov by recalling the referenced values using the group command.

The code now uses a Python "for" loop to iterate through each of the names on the page. For loops are explained in Chapter 9 of *The Data Journalist*.

We can add a line to print the two variables out to the screen. The entire function should now look like this:

```
def name_extractor(the_page):
      the_page = re.sub("\r", "", the_page)
      the_page = re.sub("\n", "", the_page)
      the_page = re.sub(" +", " ", the_page)
      for each in re.finditer('<a
href="/honour\.aspx\?id=.+?>(.+?)</a>\s?</td>\s?<td>\s?(.+?
)\s?</td>', the_page):
     recipient = each.group(1)
     city_prov = each.group(2)
     print recipient, city_prov
```

Now we can pull it all together by having the main part of our script in the loop send each page it downloads from the Governor General's website to the function. We'll take out the line in the loop that prints the page variable and replace it with a line that calls the function. We'll also import the time module at the top and use to it put it to sleep for one second between each cycle through the loop, so we don't overload the server with page requests.

The entire script now looks like this:

```
import urllib2, re, time

def name_extractor(the_page):
      the_page = re.sub("\r", "", the_page)
      the_page = re.sub("\n", "", the_page)
      the_page = re.sub(" +", " ", the_page)
      for each in re.finditer('<a
href="/honour\.aspx\?id=.+?>(.+?)</a>\s?</td>\s?<td>\s?(.+?
)\s?</td>', the_page):
      recipient = each.group(1)
      city_prov = each.group(2)
      print recipient, city_prov

counter =1
while counter < 6:
      url =
"http://www.gg.ca/honours.aspx?t=12&types=12&pg=" +
str(counter)
      the_page = urllib2.urlopen(url).read()
```

```
        name_extractor(the_page)
        counter = counter + 1
        time.sleep(1)
```

If you've coded this script properly, it should print out the first 250 names in the database of Order of Canada recipients, 50 at a time. If you want the entire dataset, you can change the value that sets the limit on the loop to a higher figure, such as 150, to be sure you'll get all the pages. The script will crash and give an error message when it runs out of names.

The process may seem dense and complex, but consider that, with only 18 lines of code, we wrote a program that will extract a government database and display it all on screen. All you have to do is highlight it and copy it into Excel or a text editor to save it for later use, or you can have Python write the results to a CSV file using the CSV module as shown in Chapter 9.