Web Scraping

In this section we will learn about Web Scraping and how to use it to collect information from web-pages.

References

- Web Scraping with Python[1] chapters 1 through 3
- Python's requests library[2]
- Python's BeautifulSoup library[3]

Notes

Web Scraping is the process of extracting data from web pages. This consists of a number of activities:

- Programmatically access a web page's content.
- Optionally, submit query data by programmatically filling out a form.
- Programmatically detect and follow links on the web page.
- Parse and process a page's HTML content and extract key information.

This would typically involve two libraries:

- A library to make HTTP requests, like Python's requests library[4].
- A library to parse and process web-pages. We will use Python's BeautifulSoup library[5].

Web Scraping vs APIs

Web Scraping differs from APIs and Web Services. These APIs are designed to be *read by programs*, rather than humans. They require a considerable investment on the part of the server, and therefore not all websites with useful information expose that information via an API.

On the other hand, web-scraping processes the same web-page that humans see on a browser, with the only difference that it can read more of the underlying structure of the page, rather than just the visible text. But in essence, web-scraping requires that we program the computer to read information that was designed to be read by humans. This is a somewhat more challenging endeavor, but it can also be applied

[1] http://acmsel.safaribooksonline.com/9781491910290
[2] http://docs.python-requests.org/en/master/
[3] https://www.crummy.com/software/BeautifulSoup/
[4] http://docs.python-requests.org/en/master/
[5] https://www.crummy.com/software/BeautifulSoup/

to more situations, as there are many more web pages out there than there are web services.

If you can view it in your browser, you can access it via a Python script.

Web APIs Not human-readable. Optimized for programmatic consumption. Not all sites offer them. Some times expose only limited functionality.

Web Pages Human-readable. Optimized for human consumption. Much more prevalent.

Basic Web-Scraping

The basic structure of a web-scraping script would be something like this:

```
import requests
import bs4      # Beautiful Soup

http_response = requests.get("... web page address ...")

# Possibly consider http_response.status_code to see
# if the page could not be found

page_content = BeautifulSoup(http_response.content)

# Navigate the page_content object
# Extract the desired information
# Print or save results
```

As a start example, we will show you how we downloaded the game of thrones books. This would tell us what kind of web address to use.

```
import requests
from bs4 import BeautifulSoup

base_site = "http://www.readbooksvampire.com"
author = "/George_R.R._Martin/"
http_response = requests.get(base_site + author)
bsObj = BeautifulSoup(http_response.content, "html.parser")
print(bsObj.prettify())
```

This `bsObj` represents the entire HTML document, and what you see from the last command is a prinout of that HTML document. You will see a nested structure in terms of "tags", like `<html>`, `<body>` etc, which match with closing tags `</html>`. This creates a nested structure, that we can try to dig in. For instance we can get to the title tag via:

```
bsObj.html.head.title
```

In this case, given there is only one title tag around, we could also do:

```
bsObj.title
```

The main document contents is all within the `<body>` tag within the `<html>` tag.

Tag objects in BeautifulSoup provide many functionalities. They all have a "name" property that speaks to the kind of tag we have:

```
bsObj.title.name
```

We can also get a look at the attributes, if any:

```
bsObj.body.div
bsObj.body.div.attrs
bsObj.body.div['class']
bsObj.body.div.get('class')      ## Safer, returns None if attribute is not there
```

We can also access its chidren, i.e. the contained tags. This is an enumerable structure, and we can iterate over it.

```
for tag in bsObj.body.div.children:
  print(tag)
```

Or we can use a list comprehension and turn it into an actual list or do something else:

```
[
  tag for tag in bsObj.body.div.children
]
```

Note all the extra "newline" tags. They also count as children. We should try to take them out. These represent the text entries within the document, and they can be detected by the fact that they don't have a name:

```
[
  tag
  for tag in bsObj.body.div.children
  if tag.name is not None
]
```

Based on these tools we can now do more complex operations. For instance notice the "a" tag inside the div tags above. It has an "href" field. We can use it to read the "links". That's what <a> tags are, links to other pages.

```
[
  tag.a['href']
  for tag in bsObj.body.div.children
  if tag.name is not None
]
```

We can also try to directly grab all "a" links, via the findAll method:

```
[
  tag['href']
  for tag in bsObj.find_all('a')
]
```

If we scroll through the initial list of items, we will see that the links we want, to the various books we want to include, are all inside table tags. We can therefore do the following:

```
[
  tag['href']
  for table in bsObj.find_all('table')
  for tag in table.find_all('a')
]
```

Let's also grab a bit more information from each link:

```
books = [
  {
    'link': tag['href'],
    'title': tag['title']
  }
  for table in bsObj.find_all('table')
  for tag in table.find_all('a')
]
```

Before we move on, we note that the first item in that list is not actually an individual book, but the name of the whole collection. So we'll leave it out.

```
books.pop(0)
```

Before we move on, let's discuss the overall plan:

- We collect the link of all books, as above
- For each book, we visit its page and collect links to all chapters
- For each chapter, we visit its page and collect the text

Following links

Following links simply requires making a new request. For instance let's grab the first book, and use its link to download a new resource.

```
book1 = books[0]
http_response1 = requests.get(base_site + book1['link'])
bookObj1 = BeautifulSoup(http_response1.content, "html.parser")
print(bookObj1.prettify())
```

We now need to grab the chapter links. It is not easy to see a way to get hold of them, but we can use regular expressions.

```
import re
bookObj1.find_all('a', { 'href': re.compile("\d+\.html") })
```

The regular expression `\d+\.html` basically tells BeautifulSoup to look for links whose href contains "at least one (+) digit (\d) followed by a dot (.) followed by the word html".

Let's write it slightly differently:

```
chapters1 = [
  {
    "title": book1["title"],
    "link": tag["href"],
    "chapter": ....
  }
  for tag in bookObj1.find_all('a', { 'href': re.compile("\d+\.html") })
]
```

We just need to figure out what to put in the "chapter". We need to grab the number part of the url. Here's how we can do that:

```
chapters1 = [
  {
    "title": book1["title"],
    "link": tag["href"],
    "chapter": re.search("\d+", tag["href"]).group()
  }
  for tag in bookObj1.find_all('a', { 'href': re.compile("\d+\.html") })
]
```

Regular expressions are quite powerful at extracting parts of strings!

We should automate this into a function "getChapters":

```
def getChapters(book):
  http_response = requests.get(base_site + book['link'])
  bookObj = BeautifulSoup(http_response.content, "html.parser")
  return [
    {
      "title": book["title"],
      "link": tag["href"],
      "chapter": re.search("\d+", tag["href"]).group()
    }
    for tag in bookObj.find_all('a', { 'href': re.compile("\d+\.html") })
  ]

allChapters = [
  chapter
  for book in books
  for chapter in getChapters(book)
]
```

Getting the text out

Now we will try to visit a chapter, and look at its format.

```
chapter1 = allChapters[0]
chapter1
```

We will now grab the link:

```
http_response1 = requests.get(base_site + chapter1['link'])
bookObj = BeautifulSoup(http_response1.content, "html.parser")
print(bookObj.prettify())
```

Scrolling through this, we find that the paragraphs are inside elements, all inside a main <td> element of class "concss". We'll try to get hold of that:

```
bookObj.find('td', { "class": "concss" }).find_all('div')
```

If you look through the list, you will see some empty divs. We probably don't want those. But we also don't really want the "div" parts, only their contents:

```
[
  tag.get_text()
  for tag in bookObj.find('td', { "class": "concss" }).find_all('div')
]
```

You will see entries that contain: '\xa0'. These are the whitespaces that we want to remove. We'll therefore filter them out:

```
[
  tag.get_text()
  for tag in bookObj.find('td', { "class": "concss" }).find_all('div')
  if tag.get_text() != '\xa0'
]
```

We now have our list of paragraphs! Now let's turn this into a function:

```
def getParagraphs(chapter):
  http_response = requests.get(base_site + chapter['link'])
  chapterObj = BeautifulSoup(http_response.content, "html.parser")
  return [
    {
      "paragraph": ind,
      "text": tag.get_text(),
      "title": chapter["title"],
      "chapter": chapter["chapter"]
    }
    for ind, tag in enumerate(bookObj.find('td', { "class": "concss" }).find_all('div'))
    if tag.get_text() != '\xa0'
  ]
```

Finally, we'll put it all together:

```
allParagraphs = [
  paragraph
  for book in books
  for chapter in getChapters(book)
  for paragraph in getParagraphs(chapter)
]
```

We could then proceed to, for instance, write the allParagraphs list to a json file.