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 extract information about the world's mountains from the wikipedia page linking all mountains.

```
import requests
from bs4 import BeautifulSoup

base_site = "http://en.wikipedia.org"
mountains_list_age = "/wiki/List_of_mountains_by_elevation"
http_response = requests.get(base_site + mountains_list_age)
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 printout of that HTML document. You can also use your browser's developer tools to see a page.

You will see a nested structure in terms of "tags", like `<html>`, `<body>` etc, which match with closing tags `</html>`. This creates a tree-like nested structure, that we can try to dig into. The *BeautifulSoup* library offers access to this structure in two ways:

- You can navigate from a tag to its children.
- You can target all tags with specific properties (e.g. all "link tags", the tag with a specific id, etc).

For instance we can reach the title tagfrom the top object by navigating its parent chain: "'python bsObj.htm

tag within the

' tag.

Notice that what you get back is not just the text, but a **tag object**:

```
type(bsObj.title)
```

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.children:
  print(tag.name)
  if tag.name is not None:
    print(tag.attrs)
```

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

```
elems = [
  tag for tag in bsObj.body.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:

```
elems = [
  tag
  for tag in bsObj.body.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.get("href")
  for tag in bsObj.body.children
  if tag.name is not None
  if tag.a is not None
]
```

We can also try to directly grab all “a” links, via the findAll method. This might actually give us more links, as the above method clearly gave us very few links (it only looked at immediate children, not deeper descendants).

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

We now need to narrow that list down to the elements we want. Typically this requires:

- Looking throught the tree structure on the browser, and identifying the elements we want to access.
- Identifying some unique property that all those elements share.
- Writing code to pick out the elements with that property.

For example, we can see that all the entries we are after are inside tables. We could start by targeting those tables, and more specifically the tbody elements that hold the non-header rows of those tables.

```
rows = [
  row
  for tbody in bsObj.find_all('tbody')
  for row in tbody.find_all('tr')
]
len(rows)
```

Woah that’s a lot of rows! Let us examine one of these rows:

```
row0 = rows[0]
```

Hm that is interesting. Remember that we show a thead element in the browser, and that element contained that heading? It appears that in the processed page, the headings are actually in the tbody: Some times Javascript that runs on the page will change the document structure.

```
bsObj.find_all('thead')      # There aren't any!
```

So what this means is that we must somehow skip the header rows. The only way to really detect them is by checking one of the values. Notice that they have the th tag in them, while normal table cells have the td tag in them. So we can try to detect that:

```
rows = [
  row
  for tbody in bsObj.find_all('tbody')
  for row in tbody.find_all('tr')
  if row.th is None
]
len(rows)
```

This is interesting. We had exactly 9 fewer entries, which matches with the 9 divisions of mountains based on height.

Now let us consider each row:

```
row0 = rows[0]
print(row0)
```

Let's say we want to record the following information for each mountain: Its name, the link to its webpage, its elevation in meters, and which mountain range it is a part of. We'll create a little dictionary from each entry. Our method would be as follows:

- Figure out how to create the dictionary based on one row.
- Turn it into a function and apply it to the entire list via a list comprehension.

Let's get started! Here is the various information, which we discovered one at a time with some trial and error:

```
{
  "name": row0.a.text,
  "link": row0.a.get("href"),
  "height": row0.find_all("td")[1].text,
  "range": row0.find_all("td")[3].text
}
```

Hm notice the weird `\xa0` symbols. These are invisible whitespace characters. We can take them away via a regular expression match:

```
import re
whitespace = re.compile("\xa0+")
whitespace.sub("", row0.find_all("td")[3].text)

{
  "name": row0.a.text,
  "link": row0.a.get("href"),
  "height": row0.find_all("td")[1].text,
  "range": whitespace.sub("", row0.find_all("td")[3].text)
}
```

Now that we have something working, we'll try it out on the full list:

```
mountains = [
  {
    "name": row.a.text,
    "link": row.a.get("href"),
    "height": row.find_all("td")[1].text,
    "range": whitespace.sub("", row.find_all("td")[3].text)
  }
  for row in rows
]
```

Oops, we hit an error! To help diagnose this, we can start by doing a for loop instead, and print information:

```
for row in rows:
  print({
    "name": row.a.text,
    "link": row.a.get("href"),
    "height": row.find_all("td")[1].text,
    "range": whitespace.sub("", row.find_all("td")[3].text)
  })
```

Looks like the error happened right after the "Santa Fe Baldy" mountain entry. If we search for it we find a "Mount Baldwin" entry following it, but without a link to it. Therefore our row.a.text part failed. We'll need to either use a more complicated function, or read through the beautifulSoup documentation for other methods of getting the text that might work for None objects as well. Or we can also ask our row object for its available methods:

```
dir(row0.a)
```

You should see a get_text method there. Let's find out more about it:

```
help(row0.a.get_text)
```

You can also search for the method's documentation on the BeautifulSoup page.

Now, let's use that instead and see if it works.

```
for row in rows:
  print({
    "name": row.a.get_text(),
    "link": row.a.get("href"),
    "height": row.find_all("td")[1].text,
    "range": whitespace.sub("", row.find_all("td")[3].text)
  })
```

Nope, it didn't work, as row.a is in fact the None value and doesn't implement the get_text method. We could use row.td.get_text() instead, but this would not fix the link entry for us.

At this point, we would consider making a function that takes a row as input and returns this dictionary:

```
def row_to_dict(row):
  anchor = row.a
  tds = row.find_all("td")
  return {
    "name": row.td.get_text(),
    "link": None if row.a is None else row.a.get("href"),
    "height": tds[1].text,
    "range": whitespace.sub("", tds[3].text)
  }

mountains = [
  row_to_dict(row)
  for row in rows
]
```

Well done! You've completed your first web scraping. We could now continue to do more with each mountain. For example we could follow the links to each mountain's page, and search for an image of the mountain there, and include that link.

Following links

Following links simply requires making a new request. For instance let's take a look at the first mountain, use its link to download its full page, then look at all the image tags on that page:

```
m1 = mountains[0]
http_response1 = requests.get(base_site + m1['link'])
mObj1 = BeautifulSoup(http_response1.content, "html.parser")
images = mObj1.find_all('img')
pprint(images)
```

Woah there are 104 images on that page. We need to somehow only pick one. We would like this to be the one on the section on the top right.

It looks like the right side always contains a table element with classes infobox and vcard. There are many of these, but the first one is probably the one we want. We could try to target that table element, then grab the first image tag within it:

```
sidebar = mObj1.find('table', class_="infobox")
sidebar.find('img')
```

Now we'll write a little function that does this: for a dictionary entry for a mountain, it follows the link to the mountain's main page, if there is one, grabs this image tag, then extracts the src link from it.

```
def update_image_link(mountain):
  if mountain["link"] is None:
    mountain["image_link"] = None
  else:
    http_response = requests.get(base_site + mountain["link"])
    mObj = BeautifulSoup(http_response.content, "html.parser")
    sidebar = mObj.find('table', class_="infobox")
    img = sidebar.find('img')
    mountain["image_link"] = img.get("src")

for mountain in mountains:
  print(mountain["name"])
  update_image_link(mountain)
```

We added the printout to see the process as it goes. This script will take a while to run as it has to access over 1400 different webpages.

And we see it got stuck on Batura IV. Looking at this name in the initial webpage shows a "red link". This points to a non-existing page, so of course we won't find any images there. We have two ways to try to fix this:

- Detect which links point to those pages, and exclude them.
- Be more robust in our search for the image link on that page, namely check that the sidebar exists before trying to find its image, and checking that the image exists before trying to get its source link.

In this case we can follow either approach. We will try them both. Notice how the link for Batura IV has index.php as a part of it. Surely the valid links won't contain that. So we can use a regular expression to look for that:

```
indexRegEx = re.compile("index.php")
def update_image_link(mountain):
  if mountain["link"] is None or \
        indexRegEx.search(mountain["link"]) is not None:
```

```
    mountain["image_link"] = None
    return
  http_response = requests.get(base_site + mountain["link"])
  mObj = BeautifulSoup(http_response.content, "html.parser")
  sidebar = mObj.find('table', class_="infobox")
  img = sidebar.find('img')
  mountain["image_link"] = img.get("src")

for mountain in mountains:
  print(mountain["name"])
  update_image_link(mountain)
```

Good, that helped us make progress! But now we are stuck on Lungser Kangri. Following its link we see that there is no sidebar in the resulting page. So let's add some safeguards there. As we seem to be setting the mountain["link"] to None in many places, let's do it only once instead, at the beginning:

```
indexRegEx = re.compile("index.php")
def update_image_link(mountain):
  mountain["image_link"] = None
  if mountain["link"] is None:
    return
  if indexRegEx.search(mountain["link"]) is not None:
    return
  http_response = requests.get(base_site + mountain["link"])
  mObj = BeautifulSoup(http_response.content, "html.parser")
  sidebar = mObj.find('table', class_="infobox")
  if sidebar is None:
    return
  img = sidebar.find('img')
  if img is None:
    return
  mountain["image_link"] = img.get("src")

for mountain in mountains:
  print(mountain["name"])
  update_image_link(mountain)
```