Lecture 14- Three ways to scrape the web

Jonathan Chang

2017-02-23

Web Scraping Three Ways, a recipe

Problem: people post data on their websites and don't provide a download. You are too lazy busy to manually copy down the data into Excel or something.

Solution: Use your new Practical Computing skills!

Check out all the cool data on Fishbase. Wouldn't it be great if we could download all of that?

Search for your favorite fish. I like the starry flounder, *Platichthys stellatus*.

Let's try to get some of this lovely data programatically.

Part 0. A brief introduction to the web

There are three different languages that power all the websites that you use. Facebook, Google, etc. all use HTML, CSS, and Javascript together to let you download pictures of animals when you're having a bad day.

- HTML gives *structure* to content (text, images, etc.)
- CSS gives *appearance* to content (font colors, image borders, etc.)
- Javascript makes content *interactive* (a clock that ticks in real time, sparkly things that follow your mouse cursor)

We'll only focus on HTML today.

HTML structure

Think of HTML as imposing a kind of structure to your documents. For example, suppose you're reading a scientific paper. You can imagine a pretty straightforward structure: the entire document encloses a number of sections, like the introduction, methods, and results, which themselves enclose several paragraphs of text each. It might look like:

In HTML, it might look like:

```
<main>
  <h1>What is web scraping?</h1>

  <section>
    <h2>Defining "web"</h2>
    <p>Some body paragraph...</p>
    <p>Some more body text...</p>
  </section>

  <section>
    <h2>Figuring out "scraping"</h2>
    <p>Some body paragraph...</p>
    <p>Some more body text...</p>
  </section>
</main>
```

HTML is composed of "tags", the funky things in angle brackets. The actual text `<p>` is an "opening tag", and the actual text `</p>` is called a "closing tag". Everything inside of these tags is a part of the `<p>` element, which is part of that abstract structure that your browser generates when it sees these tags.

One way to check out this structure is to right click on this page and click "Inspect Element". (If you're using Microsoft Edge, you'll need to press F12 to open Developer Tools to get it to show up; if you're using Safari, you'll need to check the "Show Develop menu in menu bar" option in Safari's preferences.) You'll be able to see the structure of the document in the form of HTML tags!

Part 1. Web scraping, in the shell

Species are all named with Latin, but sometimes it's nice to know where the Latin roots come from. Let's try to build something that will fetch the etymology for something on Fishbase.

We can download the source of this page directly. In terminal, download the page and let the HTML flow through you:

```
curl -s 'http://fishbase.us/summary/SpeciesSummary.php?ID=4249' > flatfish.html
gedit flatfish.html
```

This is a lot! Let's use `grep` to find the Etymology:

```
grep Etymology flatfish.html
```

There's some HTML tags left over, but that's nothing that a find-and-replace regular expression couldn't fix up.

You can also use this directly in a pipeline, like so:

```
curl -s 'http://fishbase.us/summary/SpeciesSummary.php?ID=4249' | grep Etymology
```

Exercise 1

In a pipeline, use `curl`, `grep`, and `cut` to extract the "Fisheries" status of the starry flounder.

Part 2. Web scraping, in Python

Suppose we want to get all the species in a family from Fishbase. Go back to the Fishbase home page and scroll down to "Information by Family". Pick a family from the drop down, then click "Nominal species".

We want a list of all the current names for this family. Use the Inspector to observe that the current name is always in an `<i>` tag nested within an `<a>` tag. Let's exploit this with Python!

First install the `beautifulsoup` Python library.

```
conda install -c anaconda beautifulsoup4
```

Now start up Python notebook.

These lines load up the modules we'll need to download things from the web (`urllib.request`) and to parse the HTML that comes out of it (`bs4`).

```
import urllib.request
from bs4 import BeautifulSoup
```

Set the link that we want to visit. (This will be different depending on what family you picked).

```
family_url = "http://www.fishbase.org/Nomenclature/NominalSpeciesList.php?family=Abyssocottidae"
```

This works just like a `open()` does with a file on your local computer. Python will download the contents of your link when you call `html.read()`.

```
html = urllib.request.urlopen(family_url)
html_doc = html.read()
html.close()
```

This tells Beautiful Soup to parse your document into a structure that Python understands, instead of an undifferentiated mess of tag soup.

```
soup = BeautifulSoup(html_doc)
```

Time to sample this soup. We want to find all the `<a>` tags, and for every `<a>` tag, we want to extract the `<i>` tags that are nested within them to get the species names inside.

First, use the `.find_all()` method to find all of the links in the document.

```
links = soup.find_all("a")
```

Next, we use a `for` loop to iterate over all the links that we found.

```
for link in links:
```

We now want to find all of the `<i>` tags. We'll need another call to `find_all`, but this time we're calling it on `link` and not `soup`. (`soup` refers to the whole document, while `link` refers to the `<a>` tag we're *currently* working on).

```
    italics = link.find_all("i")
```

We need to loop over all of the `<i>` tags, so we use a second level `for` loop to iterate over them.

```
    for italic in italics:
```

Now we can just write out the species name:

```
        print italic.string
```

All together, the code looks like this:

```
import urllib.request
from bs4 import BeautifulSoup

family_url = "http://www.fishbase.org/Nomenclature/NominalSpeciesList.php?family=Abyssocottidae"
# some students will have used this page instead. the code works for both
# family_url = "http://www.fishbase.us/identification/specieslist.php?famcode=584"

html = urllib.request.urlopen(family_url)
html_doc = html.read()

soup = BeautifulSoup(html_doc)

# find all links
links = soup.find_all("a")
for link in links:
    # find all italic tags within each link
    italics = link.find_all("i")
    for italic in italics:
        # this should be the species name
        print italic.string
```

Exercise 2:

Go back to the page for the starry flounder and click on the "Synonyms" link, next to Classification / Names. Starting from the above code, modify it to list all the synonyms of *Platichthys stellatus*.

Hint: use the Inspector to examine the names. Is the HTML the same as the nominal species list as before?

Bonus: Also indicate which one is the preferred synonym (rendered in bold).

Part 3. Meet my friend, "JSON API"

If we're lucky, the website we're trying to wrestle data from has left the dark ages and implemented an easy way for scripts to get at their juicy datas. These are called "application programming interfaces", or APIs for short. They provide data in a **standardized, machine-readable** format, so you don't have to use regular expressions and a 40-level deep `for` loop to find what you want.

The Fishbase API is documented here. Look at all this stuff they let you access! Let's get some data for our friend *P. stellatus*...

```
curl 'https://fishbase.ropensci.org/species/?genus=Platichthys&species=stellatus'
```

What's all this?! This is JSON format, short for Javascript object notation. If you squint at it, it almost looks *exactly* like the syntax for dictionaries and lists that you've been writing for weeks!

Unfortunately it's not quite the same. Luckily there's a Python module that will parse it for us. (Note that if you're consuming an API over the web, they'll usually be delivered as either JSON or XML. XML looks a lot like HTML, so you can just use Beautiful Soup to handle it).

```
import urllib.request
import json

api_url = "https://fishbase.ropensci.org/species/?genus=Platichthys&species=stellatus"

raw_json = urllib.request.urlopen(api_url)
decoded_json = raw_json.read().decode('utf-8')
parsed_json = json.loads(decoded_json)
raw_json.close()
```

Try just typing `parsed_json` into Python. Now that mess of JSON text is just a big nested Python data structure! Try `parsed_json["data"][0]["Comments"]`, or `parsed_json["data"][0].keys()` for a list of all the available fields.

Exercise 3

1. Print out a bit of information for the starry flounder, like its max weight ("Weight") or its max length ("Length").
2. If you go to https://fishbase.ropensci.org/species/?genus=Platichthys it'll give you two entries, for both species in genus *Platichthys*. Modify your code from Step 1 to print out the scientific name and a bit of data for each species.

   Hint: `parsed_json["data"][0]` holds data for the *first* search result. You'll need to use a `for` loop to get data for both species. What container type is `parsed_json["data"]`, a dictionary or a list?

Part 4. Conclusion

There's a lot we didn't go over today. For further reading, check out these resources:

- This free ebook for learning HTML and CSS, brought to you by Interneting is Hard(tm)
- The reference manual for Beautiful Soup
- API reference for Fishbase
- Some examples of JSON and XML

Homework

Using any or all of the above methods, generate a Markdown or LaTeX report of either (a) some of your favorite fish species, or (b) a genus or family of related fish. The report should contain at least 2 pieces of data about the fish, not including its scientific name.

Example 1. I like the mummichog, the Dover sole, the grunt sculpin, and the slingjaw wrasse. I write a script to download the species pages for each of these fish, parse the HTML, and write out the following markdown document:

```
# My favorite fish

## Fundulus heteroclitus

Family: Fundulidae
Max length: 15 cm

## Microstomus pacificus

Family: Pleuronectidae
Max length: 76 cm

## Rhamphocottus richardsonii

Family: Rhamphocottidae
Max length: 8.9 cm

## Epibulus insidiator

Family: Labridae
Max length: 54 cm
```

Example 2. I like the family Istiophoridae (marlins). I write a script to use the Fishbase API to search for members of that family, then generate a report like above.

Hint 1: If you use the API, first search for a species in that family, retrieve the `FamCode`, then repeat the search but using only the `FamCode`, e.g., https://fishbase.ropensci.org/species/?FamCode=419

Hint 2: If you scrape by parsing HTML, you'll need to retrieve the `href=""` attribute from the `<a>` tag to get the location of the species page. The link to the species page will be in `link["href"]`.