Web Scrapping

Jan Carbonell

July 15th, 2019

Academy AI

Week1: Review

How did the week go?

```
1    def week1(effort, exercises):
2        if effort >=100 and exercises =="complete":
3            print("First week complete!! Wohooo")
4        else:
5            print("Great job & keep pushing with the exercises")
```

First Step

Figure 1: The first step is always the hardest

Setup Monday

- We reviewed the methodology and structure of the bootcamp
- You know how to operate the terminal
- You have all the required tools installed
  - Github
  - A text editor or an IDE
  - Anaconda (with DL environment, Pandas, Numpy...)
  - Jupyter Notebook
- You did the first tutorial into Google Collab

Intro Tuesday

- Why Python
- Types
  1. List
  2. Dictionaries
  3. Tuples
  4. Sets
  5. Strings
  6. Comprehensions
- Operations
- Functions
- Recursion
- Loops
- Exceptions & Debugging

Files Wednesday

- Why Files make sense
- Delving into Github
- Creating a text file
- Reading a text file
- Appending a text file

Objects Thursday

- Catching up on Wednesday Exercises
- Object Oriented Programming
  1. Defining a Class
  2. Inheritance
  3. Instantiating a class
  4. Class atributes
  5. Class methods

Numpy Friday

- Numpy
  1. Why Numpy is so efficient
  2. NDArray
  3. Numpy Data Types
  4. Operations with Numpy
  5. Reading a text file
  6. Appending a text file
- Pandas
  1. From CSV to Pandas
  2. Creating Dataframes
  3. Preprocessing Dataframes
  4. Accessing and indexing df
  5. Updating and modifying df
  6. Statistical Analysis

This week

The Plan

1. Scrapping
2. Matplotlib and Sklearn
3. Calculus review
4. Stats and Probability
5. Bayesian Probability
6. (Optional: Intro to Python I)
7. (Optional: Intro to Python II)

Refreshing Git

Let's practice with the remote:

```
cd ak_w1_d3
git remote remove origin
git remote -v
git remote add origin
git remote add git@github.com:jcllobet/akademy_lectures.git
git remote -v

git pull

# you can now update the name of your local repository if you want
```

Today: Parsing & Storing Data

Data Formats

There are 3 common data formats in the web: CSV, XML & JSON

Lecture templates

Available **here**:

CSV

CSV

Filename: beers.csv

```
1 "Name","Appearance","Origin"
2 "Edelweiss","White","Austria"
3 "Cuv\e des Trolls","Blond","Belgium"
4 "Choulette Ambr\e","Amber","France"
5 "Gulden Draak","Dark","Belgium"
```

CSV (2)

Comma-Separated Values

- Human readable plain text containing any number of records
- One row per record
- Fields of a record are separated by a comma
- Header row (optional)

CSV (3)

Pro's

- Very flat data
- Can be imported to excel and other working formats (and converted from there)

Con's

- There is a lack of standard:
  1. Delimiters aren't always commas
  2. Fields aren't always quoted
  3. Easy to make mistakes, hard to spot.

```
1 Name;Appearance;"Origin"
2 Edelweiss;"White";Austria
```

```
1 Name Appearance Origin
2 Edelweiss White Austria
```

XML

XML

Filename: beers.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beers>
3    <title>Great beers</title>
4    <beer>
5      <name>Edelweiss</name>
6      <appearance>White</appearance>
7      <origin>Austria</origin>
8    </beer>
9    <beer>
10     <name>Cuve des Trolls</name>
11     <appearance>Blond</appearance>
12     <origin>Belgium</origin>
13   </beer>
14   ...
15 </beers>
```

XML (2)

Extensible Markup Language

- Human/machine readable text
- XML declaration line
- Focus on a document containing elements and nested sub-documents
- Elements composed of a tag and a content
- Several schemas systems on top of XML

JSON

JSON

Filename: beers.json

```
{
  "title": "Great beers",
  "beers": [
    {
      "name": "Edelweiss",
      "appearance": "White",
      "origin": "Austria"
    },
    {
      "name": "Cuve des Trolls",
      "appearance": "Blond",
      "origin": "Belgium"
    },
    # ...
  ]
}
```

JSON

JavaScript Object Notation

- Human readable text
- Format derived from JavaScript but completely independent
- Key-value pair datas
- Similar to Python dictionaries

This are web most used formats!

For scrapping, dowloading a file, exporting data, sending api requests, it will not be your first time nor your latest.

Parsing

What is parsing

From Data Files into Python Objects

- JSON ¿¿ into ¿¿ Lists, Dictionaries, Beautiful Soup Object
- CSV ¿¿ into ¿¿ Lists, Dictionaries, Beautiful Soup Object
- XML ¿¿ into ¿¿ Lists, Dictionaries, Beautiful Soup Object

Storing

From Python Objects into Data Files

- JSON, CSV, XML ¡¡ into ¡¡ Lists
- JSON, CSV, XML ¡¡ into ¡¡ Dictionaries
- JSON, CSV, XML ¡¡ into ¡¡ Beautiful Soup Object

Parsing CSV Python

Filename: beers.csv

```
1  import csv
2
3  with open('beers.csv') as csv_file:
4      csv_reader =csv.reader(csv_file, delimiter=',')
5      line_count =0
6      for row in csv_reader:
7          if line_count ==0:
8              print(f'Column names are {", ".join(row)}')
9              line_count +=1
10         else:
11             print(f'\t{row[0]} works in the {row[1]} department,
                                                    and was born
                                                   in {row[2]}.')
12             line_count +=1
13     print(f'Processed {line_count} lines.')
```

Parsing CSV Python

Filename: beers.csv

Scrapping

Scrapping with Python

Sometimes there is no API. We have to scrape the HTML directly

HTML

HyperText Markup Language (HTML) is a language that web pages are created in. HTML isn't a programming language, like Python — instead, it's a markup language that tells a browser how to layout content.

We can make a simple HTML document just using this tag:

```
1 <html>
2 </html>
```

Since we have no content, we won't see anything in the browser

HTML

Each HTML document is composed of the head tag, and the body tag. The main content of the web page goes into the body tag. The head tag contains metadata about the title of the page, and other information that generally isn't useful in web scraping:

```
1 <html>
2 <head>
3 </head>
4 <body>
5 </body>
6 </html>
```

This would still have no content

HelloWorld.HTML

We'll now add our first content to the page, in the form of the p tag. The p tag defines a paragraph, and any text inside the tag is shown as a separate paragraph:

```
1  <html>
2  <head>
3  </head>
4  <body>
5  <p>
6  Here's a paragraph of text!
7  </p>
8  <p>
9  Here's a second paragraph of text!
10 </p>
11 </body>
12 </html>
```

HTML Tags

Tags have commonly used names that depend on their position in relation to other tags:

1. **child** — a child is a tag inside another tag. So the two p tags above are both children of the body tag.
2. **parent** — a parent is the tag another tag is inside. Above, the html tag is the parent of the body tag.
3. **sibiling** — a sibiling is a tag that is nested inside the same parent as another tag. For example, head and body are siblings, since they're both inside html.

HTML Tags

```
1  <html>
2  <head>
3  </head>
4  <body>
5  <p>
6  Here's a paragraph of text!
7  <a href="https://www.dataquest.io">Learn Data Science Online</a>
8  </p>
9  <p>
10 Here's a second paragraph of text!
11 <a href="https://www.python.org">Python</a> </p>
12 </body></html>
```

In the above example, we added two a tags. a tags are links, and tell the browser to render a link to another web page. The href property of the tag determines where the link goes.

Common Tags

a and **p** are extremely common html tags. Here are a few others:

1. **div** — indicates a division, or area, of the page.
2. **b** — bolds any text inside.
3. **i** — italicizes any text inside.
4. **table** — creates a table.
5. **form** — creates an input form.

For a full list of tags, look here.

Scrapping with Python

Sometimes there is no API. We have to scrape the HTML directly

The Requests Library

The first thing we'll need to do to scrape a web page is to download the page. We can download pages using the Python **requests** library. –¿ How would we do this with anaconda?

he requests library will make a GET request to a web server, which will download the HTML contents of a given web page for us. There are several different types of requests we can make using requests, of which GET is just one.

```
1 import requests
2 page =requests.get("http://dataquestio.github.io/web-scraping-
                                        pages/simple.html")
3 page
```

Status code of 200 means that the page downloaded successfully. 4 or 5 indicate some sort of error. 404!!

Printing HTML

We can print out the HTML content of the page using the content property:

```
1 import requests
2 page =requests.get("http://dataquestio.github.io/web-scraping-
                                        pages/simple.html")
3 page
4
5 page.status_code
6 page.content
```

Inspect Element

CSS Selector

```
.the_class
```

```
<div class="the_class">
  Some text
</div>
```

Additional documentation for **selectors**.

Parsing Using Beautiful Soup

We can use the BeautifulSoup library to parse this document, and extract the text from the p tag. We first have to import the library, and create an instance of the BeautifulSoup class to parse our document:

```
1 #Code Above
2 ...
3 from bs4 import BeautifulSoup
4 soup =BeautifulSoup(page.content, 'html.parser')
5 print(soup.prettify())
```

We can now print out the HTML content of the page, formatted nicely, using the prettify method on the BeautifulSoup object:

Playing with soup

As all the tags are nested, we can move through the structure one level at a time. We can first select all the elements at the top level of the page using the children property of soup

```
#Code Above
...
list(soup.children)
```

We can now select the html tag and its children by taking the third item in the list:

```
#Code Above
...
html =list(soup.children)[2]
```

Hide yo kids (children)

Now, we can find the children inside the html tag:

```
1 #Code Above
2 ...
3 list(html.children)
```

Also, remember we still have the head and body tags, from which the head is a bit useless:

```
1 ...
2 body =list(html.children)[3]
3 list(body.children)
```

We can now isolate the p tag:

```
1 p =list(body.children)[1]
2 p.get_text()
```

Additional methods

We can use the find_all method to get all the instances of a tag on a page, also, the find gets the first instance.

```
1 soup =BeautifulSoup(page.content, 'html.parser')
2 soup.find_all('p')
3 soup.find('p')
```

Searching by class and ID

We download the page and create a beautiful soup object and use find_all to search for items by class or ID.

```
1 #Code in gdocs
```

Using CSS Selectors

They allow us to specify HTML Tags to style. Some examples:

1. **p a**: finds all a tags inside of a p tag.
2. **body p a**: finds all a tags inside of a p tag inside of a body tag.

Data Extraction Exercise

Extract the 7 days weather from san francisco here (link in gdocs).

1. inspect with the console
2. target the text within the extended forecast
3. Store:
    - The name of the forecast item
    - The description of the conditions
    - A short description of the conditions
    - The temperature low
    - Save all of this into a Pandas dataframe
    - Bonus: Do a plot of the fluctuation of the data

More info on css selectors (**gsearch**)

More Exercises when done with this one :)

What do you want to explore from the internet? The WWWeb is yours now!!