Web Scraping for Data Collection

Inspired by code from tsvm and Digital Ocean

Data Collection

Sometimes, our data comes in a nice and clean format:

- ipums.org - Census Bureau Data
- collegescorecard.ed.gov - Education Data
- gss.norc.org - General Social Survey

Data Collection

Sometimes, not so much...

```
<p class="text-gray text-truncate mb-xs-0 text-body">
Dungeons and Dragons Your Stick Figure Family rolled a 1 / D20 vinyl  Dragon decal / RPG / vinyl sticker for cars windows / gifts for geeks
</p>
<div class="v2-listing-card__shop">
<p class="text-gray-lighter text-body-smaller display-inline-block mr-xs-1">FlairdeGeek</p>
<div class="v2-listing-card__rating icon-t-2">
<div class="stars-svg stars-smaller ">
<input type="hidden" name="initial-rating" value="4.985">
<input type="hidden" name="rating" value="4.985">
<span class="screen-reader-only">5 out of 5 stars</span>
```

Search results for "dungeons and dragons" on Etsy

What can we do?

We need data, but it is embedded in a web site, and wrapped in lots of HTML code.

- Download all of the code from the webpage and manually cut out the information?
- Copy and paste from the tables on a rendered website?

This might be feasible on a small scale, but what if I want to know about information stored on 1000's of pages?

Web Scraping

Web Scraping is the process of extracting information from web pages.

- Irregular
- Interactive
- Iterative

No webpage is designed the same as any other (typically), so we have to build an understanding of the pages we want to scrape before we can write our code.

What to Know

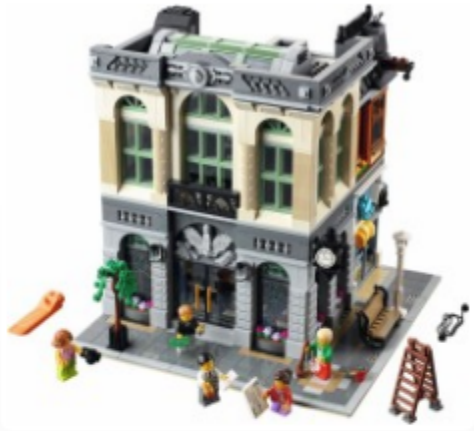
In order to scrape a web page, we need to have a basic understanding of:

- HTML tags
- CSS formatting, (here also)
- JSON data structures

From there, we will spend LOTS of time on Google making sure we get it right for the page(s) that we care about.

Brickset

Let's say that I want to collect information about lots of different Lego sets.



10251: Brick Bank

10251-1 | ADVANCED MODELS | MODULAR BUILDINGS | 2016

BANK | BASEPLATE | BRICK BUILT TREE | CREATOR EXPERT | D2C

LAMPPOST | LAUNDROMAT | MODULAR BUILDING | SAFE

★★★★★ 5 REVIEWS

PIECES 2380

MINIFIGS 5

RRP $169.99, 149.99€ | More

PPP 7.1c, 6.3c

PACKAGING Box

AVAILABILITY Retail - limited

FIRST SOLD

USA: Jan 16, UK/EU: Jan 16

INSTRUCTIONS Yes

ADDITIONAL IMAGES 26

SET TYPE Normal

NOTES Labelled Creator - Expert.

RELATED SETS

Connects with 10182-1 | 10185-1 | 10190-1 | 10197-1 | 10232-1

Brickset

This is a good page to learn scraping, because we also have the option of getting our search results back as a CSV, so we can check whether or not we collect the correct information.

Say we want to collect

- Set name
- # of Pieces
- # of Minifigs (Lego people)
- The URL of a picture of the set

Brickset

We need to

1. Identify the pieces of HTML that surround the information we need,
2. Create code that will allow us to extract that information
3. Create code that can do this for all pages

Let's go to the Brickset website 2016 collection to find what we are looking for

Our Objects of Interest

In order to find the way in which information is stored in the Brickset page, we will use our browser's developer tools to explore the page source.

Right click on the name ("10251: Brick Bank") of the first lego set after you follow the link, and choose the **Inspect** option

Our Objects of Interest

```
<h1>
  <a href="/sets/10251-1/Brick-Bank">
    <span>10251: </span> Brick Bank
  </a>
</h1>
```

We can see that the set's name is inside of an `<h1>` tag, as well as an `<a>` tag (header and link tags, respectively)

Our Objects of Interest

If we continue further, we will see that most other information is contained in a list:

```
<dl>
<dt>Pieces</dt>
  <dd><a class="plain" href="/inventories/10251-1">2380</a></dd>
<dt>Minifigs</dt>
  <dd><a class="plain" href="/minifigs/inset-10251-1">5</a></dd>
<dt>RRP</dt>
  <dd>$169.99, 149.99€ | <a class="popuplink plain" href="prices?set=10251-1">More</a></dd>
<dt>PPP</dt>
  <dd>7.1c, 6.3c</dd>
<dt>Packaging</dt>
  <dd>Box</dd>
<dt>Availability</dt>
  <dd>Retail - limited</dd>
<dt>First sold</dt>
  <dd>USA: Jan 16, UK/EU: Jan 16</dd>
<dt>Instructions</dt>
  <dd><a class="popuplink plain" href="instructions2?set=10251-1&amp;s=1">Yes</a></dd>
<dt>Additional images</dt>
  <dd><a class="plain" href="/sets/10251-1?more-images">26</a></dd>
<dt>Set type</dt>
  <dd>Normal</dd>
</dl>
```

Extracting the Data

In order to write code that will retrieve this data from each search result, we will use the `scrapy` library. Unfortunately, this cannot be easily done in a notebook.

1. Create our spider (code to **crawl** a **web**site) as a **subclass** of the `scrapy.Spider` class
2. Customize that class to detect the data we care about
3. From a command prompt, set our spider loose on the sites we are scraping

Extracting the Data

```
import scrapy

class BrickSetSpider(scrapy.Spider):
  name = "brickset_spider"
  start_urls = [
        'http://brickset.com/sets/year-2016'
        ]
```

Let's start by creating our class, and giving it a helpful name.

We then need to tell our spider where it will start crawling.

Extracting the Data

Next, we need to create a header for our CSV, so that it is clear what each column of extracted data contains:

```
with open('myresults.csv', 'w') as f:
        f.write("name, pieces, minifigs, image\n")
```

This also goes inside of our `BrickSetSpider` class.

Extracting the Data

Now, define a method named `parse`. This method will be called for each web page that is crawled by the spider, and will perform some operations (that we need to specify) for each `.set` object.

```
def parse(self, response):
  with open('myresults.csv', 'a') as f:
    SET_SELECTOR = ".set"
    for brickset in response.css(SET_SELECTOR):
      ...
```

Extracting the Data

We need to identify where the name, pieces, minifigs, and image information are stored. We will use this to direct our spider to collect that information.

```
def parse(self, response):
  with open('myresults.csv', 'a') as f:
    SET_SELECTOR = ".set"
    for brickset in response.css(SET_SELECTOR):
      NAME_SELECTOR = 'h1 ::text'
      PIECES_SELECTOR = './/dl[dt/text() = "Pieces"]/dd/a/text()'
      MINIFIGS_SELECTOR = './/dl[dt/text() = "Minifigs"]/dd[2]/a/text()'
      IMAGE_SELECTOR = 'img ::attr(src)'
      ...
```

CSS vs XPath Selectors

- CSS selectors allow us to use CSS tags to denote the elements from which we would like to retrive information.
    - Helpful when we have unique CSS tags associated with the information that we want to collect
    - Use `_____.css()` to work with these selectors
- XPath selectors allow us to direct our scrape through specific HTML tags or labels
    - Typically more robust
    - Use `_____.xpath()` to work with these selectors

Extracting the Data

Now, we need to collect the information we care about, extract the values, and then store them as a row in our new csv.

```
def parse(self, response):
  with open('myresults.csv', 'a') as f:
    SET_SELECTOR = ".set"
    for brickset in response.css(SET_SELECTOR):
      ...
      result = str(brickset.css(NAME_SELECTOR).extract_first()) + ","
      result += str(brickset.xpath(PIECES_SELECTOR).extract_first()) + ","
      result += str(brickset.xpath(MINIFIGS_SELECTOR).extract_first()) + ","
      result += str(brickset.css(IMAGE_SELECTOR).extract_first()) + "\n"

      f.write(result)
      ...
```

Extracting the Data

Finally, if our results spill beyond a single page, we need to specify how to move to the next page of results. This will initiate a new crawl of the next search result page:

```
def parse(self, response):
  with open('myresults.csv', 'a') as f:
    SET_SELECTOR = ".set"
    for brickset in response.css(SET_SELECTOR):
      ...
      NEXT_PAGE_SELECTOR = '.next a ::attr(href)'
      next_page = response.css(NEXT_PAGE_SELECTOR).extract_first()
      if next_page:
        yield scrapy.Request(
          response.urljoin(next_page),
          callback=self.parse
        )
```

Running Our Spider

In order to run our Spider, we will need to open a Terminal/PowerShell/Command Prompt:

Lab Computers/Windows:

```
python C:\ProgramFiles\Anaconda3\Scripts\scrapy runspider bricks.py
```

Mac/Unix/Linux:

```
scrapy runspider bricks.py
```

In both cases, we MUST be in the directory containing our `bricks.py` script

Learning What We Need

How can we try different selectors out until we get the right ones?

We can run the `scrapy` shell.

Unix/Linux:

```
scrapy shell
```

Windows (in the lab):

```
python C:\ProgramFiles\Anaconda3\Scripts\scrapy shell
```

Another (harder) Crawl

In the case of Brickset, the data that we want to collect is relatively easy to access. This is certainly not going to be the case for every website.

- Let's take a look at scraping some data from a search on Etsy.com
- Let's say that we are interested in collecting the URL, name, an image of the item, item description, lowest price listed, and average rating

Etsy Crawl

1. Find where the information we want is stored
2. Determine the structure of our crawl
3. Write code to move through all the necessary pages
4. Write code to break down the information on each page

You can take a look at an example scrape of Etsy in this script

Lab/Homework For This Week

This week, your homework is to scrape Brickset for some collection of Lego sets (of your choice)

Turn in: Your spider script (see my examples) named `mySpider.py`, as well as a CSV containing the results from your scrape, named `results.csv`. You will only get credit if your script runs without errors, and returns the data that you have provided in the CSV.

I will write a script that tests whether or not your script returns a CSV that is equivalent to the CSV you turn in.