# Definitions

* Web crawler: an internet bot that systematically browses the World Wide Web [3];
* Data scraping: a technique in which a computer program extracts data from human-readable output coming from another program [5].
* Web scraping: (also called web harvesting or web data extraction) is a computer software technique of extracting information from websites [4].

# Scrapy

Scrapy is a fast high-level screen scraping and web crawling framework written in Python, used to crawl websites and extract structured data from their pages. [1] I have used scrapy to extract food offers from catering sites. The framework will download the source code of a web page. Our interest zone is the HTML code. Using XPath, one can obtain a subset of the tags. Analyzing the structure of the page, a pattern of tags must be found, where the needed information is placed, in our case: food name, ingredients, price etc. Sadly, each web site has its own structure and therefore each has to be analyzed in particular.

## Introduction

This section describes and exemplifies the minimal requirements for understanding how Scrapy works.

[6] Four main steps have to be made for building a functional scraper with Scrapy.

### 1) Creating a scrapy project

This step is done easily with the following command line:

scrapy startproject tutorial

### 2) Defining an item which will be loaded with scraped data

An Item class allows to logically group data. Declare each attribute of the class as a Field. Since we have to have a food name, ingredients, price for each food item, the following declaration would be possible:

**class** **FoodItem(**Item**):**

name **=** Field**()**

ingredients **=** Field**()**

price **=** Field**()**

### 3) Defining a spider which scrapes data

The spider-classes are in charge of processing the web pages’ source code and extracting the required information into an item. Because the spider contains rules for extracting items, it is particular for each web site. For the above defined FoodItem, a template spider will look as follows:

**class** **TemplateSpider(**CrawlSpider**):**

#mandatory class attributes

name **=** "template"

allowed\_domains **=** **[**'example.com'**]**

start\_urls **=** **[**'http://www.example.com/'**]**

**def** parse**(**self**,** response**):**

#hxs will select specified tags from the HTML source code

#using XPath

hxs **=** HtmlXPathSelector**(**response**)**

#create a new FoodItem instace

item **=** FoodItem**()**

#select the text within h1 tags

item**[**'name'**]** **=** hxs**.**select**(**'//h1/text()'**).**extract**()**

#select the text within p tags, within h1 tags

item**[**'ingredients'**]** **=** hxs**.**select**(**'//h1/p/text()'**).**extract**()**

#same, but apply a regex matching a price format, $<number>

item**[**'price'**]** **=** hxs**.**select**(**'//h1/p/text()'**).**re**(**'\$[-+]?[0-9]\*\.?[0-9]+$'**)**

**return** item

### 4) Defining an item pipeline which stores the scraped data

After item has been scraped, it is sent to the Item Pipeline for further processing. There are multiple uses for pipeline, one of which is storing the item in a custom way.

Scrapy can log data in many formats, from which the XML has been chosen. By default, an item will be stored as follows: <field\_name> field\_value </field\_name>. We will see that a modified version of this feature will be used.

## Examples

For this website, [www.allspicecatering.com](http://www.allspicecatering.com), an extract from HTML source code looks like this:

<table border=**"0"** name=**""** width=**"100%"** cellpadding=**"5"** cellspacing=**"0"**>

<tr align=**""** valign=**""**>

<td colspan=**"1"** rowspan=**"1"** width=**"33%"** align=**""** valign=**"top"**><p><font size=**"4"** color=**"#4d4183"**>**Bagel Tray**</font><br>**Assorted Sliced Bagels with Cream Cheeses, Butter** *&amp;* **Jelly**<br>**$3.50**</p>

<p><br><font size=**"4"** color=**"#4d4183"**>**Bagels** *&amp;* **Lox**<br></font>**Assorted Bagels with Cream Cheese, Sliced Tomatoes, Bermuda Onion, and Butter.**<br>**$7.50**</p><br>

</td>

<td colspan=**"1"** rowspan=**"1"** width=**"34%"** align=**""** valign=**"top"**><p><font size=**"4"** color=**"#4d4183"**>**Bakery Basket**</font><br>**Fruit** *&amp;* **Nut Breads, Croissants, Coffee Cake, Muffins, Sweet Rolls**<br>**$5**</p>

<p><br><font size=**"4"** color=**"#4d4183"**>**Breakfast Buffet**</font><br>**Bakery Fresh Breads, Sweet Rolls** *&amp;* **Bagels** *&amp;* **a Tray of Sliced Fruit**<br>**$7.50**</p><br>

</td>

<td colspan=**"1"** rowspan=**"1"** width=**"33%"** align=**""** valign=**"top"**><p><font size=**"4"** color=**"#4d4183"**>**European Breakfast**</font><br>**Cheddar** *&amp;* **Brie Cheeses, Croissants, Rolls, Pound Cake, Sliced Fruit**<br>**$8.50**</p>

<p><br><font size=**"4"** color=**"#4d4183"**>**Health Bar**</font><br>**Low Fat Yogurt, Granola, Cereal Bars** *&amp;* **Fresh Fruit**<br>**$6**</p><br>

</td>

</tr>

</table>

* Food name:

One can notice that the food pieces of information are placed between <p> tags. The food name is also between <font> tags, so a good rule (XPath statement) for extracting this food name may be //p/font[@size=’4’]/text(). Translated into English, this means: the text within <font> tags with attribute size=’4’ within any <p> tag [2]. Of course, there may be <p><font> tags containing other kind of information which will be also extracted and thus hardens the process. The solution for this issue are some optimizations and consulting an ontology which are presented in chapter X.

* Ingredients:

The ingredients’ XPath statement can simply //p/text(). Notice that the rule is not recursive, i.e. the text within <font> tags will be ignored even if it is a child node of <p>.

* Price:

The price will be extracted in the same manner and applied the '\$[-+]?[0-9]\*\.?[0-9]+$' regex matching. This means basically the $ dollar sign and a real number.

At first sight, inspired by the template, one will use the following statements for scrapping:

item**[**'name'**]** **=** hxs**.**select**(**'//p/font[@size=’4’]/text()'**).**extract**()**

item**[**'ingredients'**]** **=** hxs**.**select**(**'//p/text()'**).**extract**()**

item**[**'price'**]** **=** hxs**.**select**(**'//p/text()'**).** re**(**'\$[-+]?[0-9]\*\.?[0-9]+$'**)**

Because a web page contains multiple food items, we will end with a big, ungrouped set of data.

item**[**'name'**]** will have value [<name1>, <name2>, <name3> …]

item**[**'ingredients'**]** will have value [<ingredients3>, <ingredients1>, <ingredients2> …]

item**[**'price'**]** will have value [<price2>, <price1>, <price3> …]

As a consequence, the XML output will look something like this:

<item>

<name>**name1 name2 name3**</name>

<ingredients>**ingredients3 ingredients1 ingredients2**</ingredients>

<price>**price2 price1 price3**</price>

</item>

And this is certainly not what we want. This issue has been resolved by storing an array of instances of a class called *footItem* which contains the required fields. Thus, our Scrapy item will have one field which is called itemArray.

**class** **FoodcrawlersItem(**Item**):**

itemArray **=** Field**()**

This modifies our spider’s parse method to have the following structure:

**def** parse\_item**(**self**,** response**):**

hxs **=** HtmlXPathSelector**(**response**)**

item **=** FoodcrawlersItem**()**

item**[**'itemArray'**]** **=** **{}**

#extract the whole paragraph which contains all the information grouped: title, ingredients, and price

paragraphs **=** hxs**.**select**(**'//div[@class="blockbody"]//p'**).**extract**()**

#for each paragraph, extract the foodItem

foodItemArray **=** **[]**

**for** paragraph **in** paragraphs**:**

paragraphSelector **=** HtmlXPathSelector**(**text**=**paragraph**)**

foodName **=** paragraphSelector**.**select**(**'//p/font[@size="4"]/text()'**).**extract**()**

foodIngredientsAndPrice **=** paragraphSelector**.**select**(** '//p/text()'**)**

#select food ingredients

foodIngredients **=** foodIngredientsAndPrice**.**extract**()**

#select food price

foodPrice **=** foodIngredientsAndPrice**.**re**(**'\$[-+]?[0-9]\*\.?[0-9]+$'**)**

fI **=** foodItem**(**foodNameString**,** category**,** foodIngredientsString**,** foodPriceString**)**

#append the newly created FoodItem to the array

foodItemArray**.**append**(**fI**)**

#hack to make easier to iterate item[‘itemArray]’

item**[**'itemArray'**][**foodNameString**]** **=** fI

**return** item

In addition, the XML exporter has to be modified to fit this new data structure. Investigating how the default exporter works, we have found out that three methods from the class *XmlItemExporter* must be overridden, resulting in a new class *FoodXmlItemExporter*.

* Serialize\_field method must return same value, unmodified
* Export\_item method must not create a new tag for every item
* \_export\_xml\_field method must iterate through an item `itemArray` attribute and create for each the corresponding tag

**class** **FoodXmlItemExporter(**XmlItemExporter**):**

**def** serialize\_field**(**self**,** field**,** name**,** value**):**

**return** value

**def** export\_item**(**self**,** item**):**

**for** name**,** value **in** self**.**\_get\_serialized\_fields**(**item**,** default\_value**=**''**):**

self**.**\_export\_xml\_field**(**name**,** value**)**

**def** \_export\_xml\_field**(**self**,** name**,** serialized\_value**):**

**for** key**,** value **in** serialized\_value**.**iteritems**():**

self**.**xg**.**startElement**(**self**.**item\_element**,** **{})**

self**.**xg**.**startElement**(**"foodName"**,** **{})**

self**.**xg**.**characters**(**value**.**foodName**)**

self**.**xg**.**endElement**(**"foodName"**)**

self**.**xg**.**startElement**(**"ingredients"**,** **{})**

self**.**xg**.**characters**(**value**.**ingredients**)**

self**.**xg**.**endElement**(**"ingredients"**)**

self**.**xg**.**startElement**(**"price"**,** **{})**

self**.**xg**.**characters**(**value**.**price**)**

self**.**xg**.**endElement**(**"price"**)**

self**.**xg**.**startElement**(**"category"**,** **{})**

self**.**xg**.**characters**(**value**.**category**)**

self**.**xg**.**endElement**(**"category"**)**

self**.**xg**.**endElement**(**self**.**item\_element**)**

If only all web sites were the same …

## Data logging

One major problem was logging the pieces of information collected. The built-in mechanism Scrapy has does not fit entirely our requirements. The idea is to export the food items in XML format as follows:

<item>

<foodName>Chicken with Artichokes & Pistachios</foodName>

<ingredients>

Featuring Artichokes, Pistachios & Medallions of Boneless Breast Meat Sauteed with Shallots, Garlic & Basil with Lemon Parsley Rice

</ingredients>

<price>$18</price>

<category>hotentrees</category>

</item>

By default, Scrapy will export

# References

[1] <http://scrapy.org/>

[2] <http://www.w3schools.com/XPath/xpath_syntax.asp>

[3] <http://en.wikipedia.org/wiki/Web_crawler>

[4] <http://en.wikipedia.org/wiki/Web_scraping>

[5] <http://en.wikipedia.org/wiki/Data_scraping>

[6] <http://doc.scrapy.org/en/0.18/intro/tutorial.html>