The Applications Of Web Scraping Are:

- Classifying Projects
- E-commerce
- Marketing
- Academic Research
- Product Building
- Travel
- Sales
- SERP Scraping
    - **SERP**: or search engine results page scraping

****

To Install The Required Library, We Can Use:

- **Run Command**: *pip install requests bs4*

****

To Read The Data Using BS4:

```
from bs4 import BeautifulSoup as bs;
import requests;

data = requests.get('http://pythonscraping.com/pages/page1.html');
t1 = bs(data.content, 'html.parser');
print(t1.h1);
```

****

In fact, any of the following function calls would produce the same output:

bs.html.body.h1

bs.body.h1

bs.html.h1

****

The first is the HTML string that the object is based on, and the second specifies the parser that you want BeautifulSoup to use to create that object. In the majority of cases, it makes no difference which parser you choose.

****

html.parser is a parser that is included with Python 3 and requires no extra installations to use.

****

Another popular parser is lxml. This can be installed through pip:

$ pip install lxml

lxml can be used with BeautifulSoup by changing the parser string provided:

bs = BeautifulSoup(data.content, 'lxml')

****

lxml has some advantages over html.parser in that it is generally better at parsing "messy" or malformed HTML code. It is forgiving and fixes problems like unclosed tags, tags that are improperly nested, and missing head or body tags.

****

lxml is also somewhat faster than html.parser, although speed is not necessarily an advantage in web scraping, given that the speed of the network itself will almost always be your largest bottleneck.

****

One of the disadvantages of lxml is that it needs to be installed separately and depends on third-party C libraries to function. This can cause problems for portability and ease of use, compared to html.parser.

****

Using this BeautifulSoup object, you can use the find_all function to extract a Python list of proper nouns found by selecting only the text within tags

```
import requests;
from bs4 import BeautifulSoup;

data = requests.get('http://www.pythonscraping.com/pages/warandpeace.html');
bs = BeautifulSoup(data.content, 'lxml');
t1 = bs.find_all('span', attrs={'class': 'green'});

print("The Green Span List Length is: ", len(t1));

for name in t1:
    print(name.get_text());
```

****

find() and find_all() with BeautifulSoup:

BeautifulSoup's find() and find_all() are the two functions you will likely use the most. With them, you can easily filter HTML pages to find lists of desired tags, or a single tag, based on their various attributes. The two functions are extremely similar, as evidenced by their definitions in the BeautifulSoup documentation:

find_all(tag, attrs, recursive, text, limit, **kwargs)

find(tag, attrs, recursive, text, **kwargs)

****

The tag parameter is one that you've seen before; you can pass a string name of a tag or even a Python list of string tag names. For example, the following returns a list of all the header tags in a document:1
.find_all(['h1','h2','h3','h4','h5','h6'])

****

Unlike the tag parameter, which can be either a string or an iterable, the attrs parameter must be a Python dictionary of attributes and values. It matches tags that contain any one of those attributes. For example, the following function would return *both* the green and red span tags in the HTML document:

```
.find_all('span', {'class': ['green', 'red']})
```

****

The recursive parameter is a boolean. How deeply into the document do you want to go? If recursive is set to True, the find_all function looks into children, and children's children, etc., for tags that match the parameters. If it is False, it will look only at the top-level tags in your document. By default, find_all works recursively (recursive is set to True). In general, it's a good idea to leave this as is, unless you really know what you need to do and performance is an issue.

****

The text parameter is unusual in that it matches based on the text content of the tags, rather than properties of the tags themselves. For instance, if you want to find the number of times "the prince" is surrounded by tags on the example page, you could replace your .find_all() function in the previous example with the following lines:

```
nameList = bs.find_all(text='the prince')
print(len(nameList))
```

****

The limit parameter, of course, is used only in the find_all method; find is equivalent to the same find_all call, with a limit of 1. You might set this if you're interested in retrieving only the first *x* items from the page. Be aware that this gives you the first items on the page in the order they occur in the document, not necessarily the first ones you want.

****

```
import requests;
from bs4 import BeautifulSoup;

data = requests.get('http://www.pythonscraping.com/pages/warandpeace.html');
bs = BeautifulSoup(data.content, 'lxml');

t1 = bs.find_all('span', attrs={'class': 'green'});
t2 = bs.find_all(text='the prince')

print("The Green Span List Length is: ", len(t1));
print("The Text Content 'The Prince' List Length is: ", len(t2));

for name in t1:
    print(name.get_text());
```

****

The additional kwargs parameter allows you to pass any additional named arguments you want into the method. Any extra arguments that find or find_all doesn't recognize will be used as tag attribute matchers. For example:

```
title = bs.find_all(id='title', class_='text')
```

****

You might have noticed that BeautifulSoup already has a way to find tags based on their attributes and values: the attr parameter. Indeed, the following two lines are identical:

```
bs.find(id='text')
bs.find(attrs={'id':'text'})
```

****

However, the syntax of the first line is shorter and arguably easier to work with for quick filters where you need tags by a particular attribute. When filters get more complex, or when you need to pass attribute value options as a list in the arguments, you may want to use the attrs parameter:

```
bs.find(attrs={'class':['red', 'blue', 'green']})
```

****

bs.div.find_all('img') will find the first div tag in the document, and then retrieve a list of all img tags that are descendants of that div tag.

****

As a complement to **next_siblings**, the **previous_siblings** function often can be helpful if there is an easily selectable tag at the end of a list of sibling tags that you would like to get.

****

Dealing with parents

When scraping pages, you will likely discover that you need to find parents of tags less frequently than you need to find their children or siblings. Typically, when you look at HTML pages with the goal of crawling them, you start by looking at the top layer of tags, and then figure out how to drill your way down into the exact piece of data that you want. Occasionally, however, you can find yourself in odd situations that require BeautifulSoup's parent-finding functions, **.parent** and **.parents**.

****

```
import requests;
from bs4 import BeautifulSoup;

data = requests.get('http://www.pythonscraping.com/pages/page3.html');
bs = BeautifulSoup(data.content, 'lxml');

print(bs.find('img',{'src':'../img/gifts/img1.jpg'}).parent.previous_sibling.get_
text())
```

****

****

To Use Regular Expressions With Python:

```
import requests;
from bs4 import BeautifulSoup;
import re;

data = requests.get("http://www.pythonscraping.com/pages/page3.html");
bs = BeautifulSoup(data.content, 'lxml');
images = bs.find_all('img', { 'src': re.compile('..\/img\/gifts/img.*.jpg') } );
```

```
(.venv) G:\QA-Selenium-bs4-JMeter-PyTest\Scraping-Data-With-Python-3rd-Edition>python test_regex_01.py
The Image src is:  ../img/gifts/img1.jpg
The Image src is:  ../img/gifts/img2.jpg
The Image src is:  ../img/gifts/img3.jpg
The Image src is:  ../img/gifts/img4.jpg
The Image src is:  ../img/gifts/img6.jpg
```

****

Accessing Attributes

So far, you've looked at how to access and filter tags and access content within them. However, often in web scraping you're not looking for the content of a tag; you're looking for its attributes. This becomes especially useful for tags such as a, where the URL it is pointing to is contained within the href attribute; or the img tag, where the target image is contained within the src attribute. With tag objects, a Python list of attributes can be automatically accessed by calling this:

myTag.attrs

Keep in mind that this literally returns a Python dictionary object, which makes retrieval and manipulation of these attributes trivial. The source location for an image, for example, can be found using the following line:
myImgTag.attrs['src']

****

```
import requests;
from bs4 import BeautifulSoup;

data = requests.get('http://www.pythonscraping.com/pages/page3.html');
bs = BeautifulSoup(data.content, 'lxml');
tags = bs.find_all(lambda tag: len(tag.attrs) >= 2 );
```

****

Lambda functions are so useful you can even use them to replace existing BeautifulSoup functions:

bs.find_all(lambda tag: tag.get_text() =='Or maybe he\'s only resting?')

This also can be accomplished without a lambda function:

bs.find_all('', text='Or maybe he\'s only resting?')

****

If there's no easy way to isolate the tag you want or any of its parents, can you find a sibling? Use the .parent method and then drill back down to the target tag.

****

To Separate The Command Into Multiple Lines:

```
links = bs\
.find('div', { 'id': 'bodyContent' }) \
.find_all('a', { 'href': re.compile('^(/wiki/)((?!:).)*$') });
```

****

Redirects allow a web server to point one domain name or URL to a piece of content at a different location. There are two types of redirects:

• **Server-side redirects**, where the URL is changed before the page is loaded

• **Client-side redirects**, sometimes seen with a "You will be redirected in 10 seconds" type of message, where the page loads before redirecting to the new one

****

Note (To Remember): When We Want To Select Depending On Multiple Classes, We CanUse List:

```
body  = bs.find('div', { 'class': ['page', 'page-article'] }).text;
```

****

```
def getElements(self, bs, selector):
        # select Returns Empty String If No Object Found
        items = bs.select(selector);

        if items is not None and len(items)> 0:
            return '\n'.join([item.get_text().trim() for item in items])
**********************************************************************************************
```

If We Have Multiple Websites That Has Different Types Of Data Structure, Then We Can Use Inheritance:

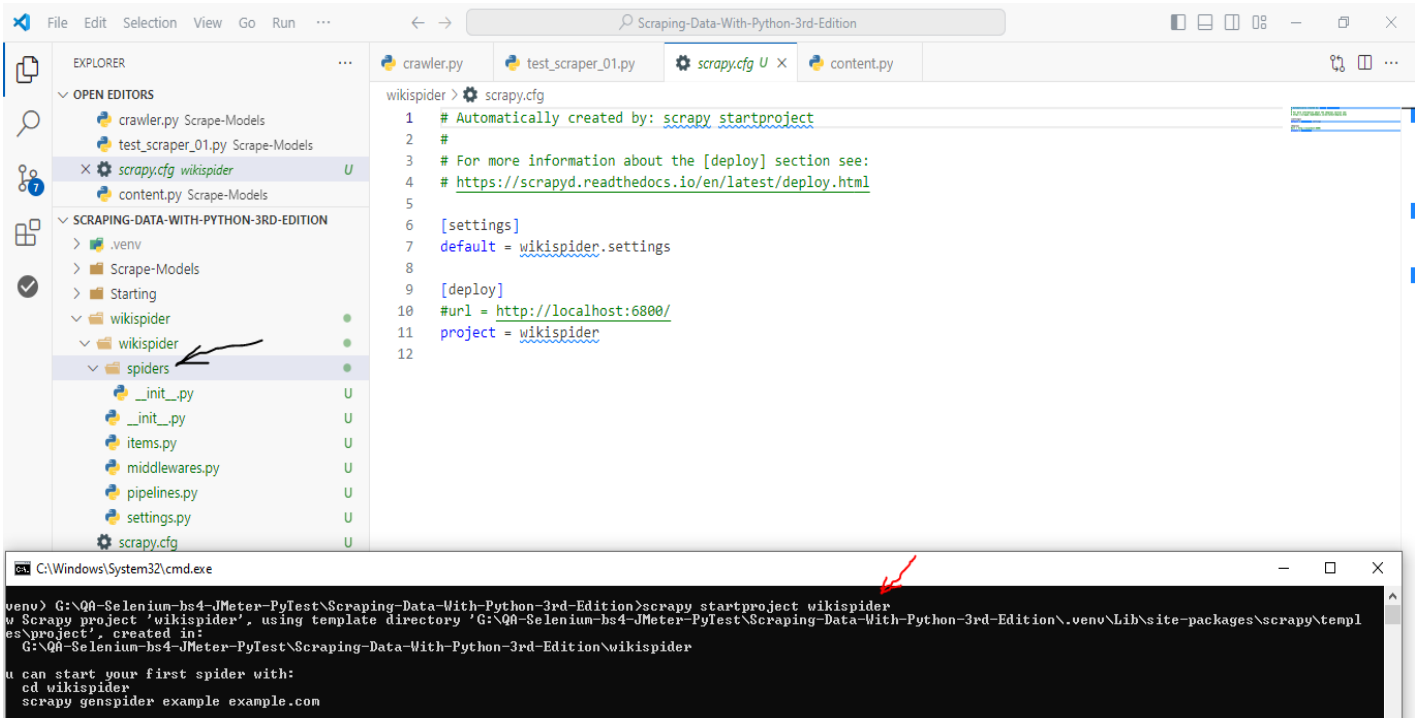
```
class Website:
    def __init__(self, name, url, title, body):
        self.name       = name;
        self.url        = url;
        self.titleTag   = title;
        self.bodyTag    = body;


class Product(Website):
        """Contains information for scraping a product page"""
        def __init__(self, name, url, titleTag, productNumberTag, priceTag):
                Website.__init__(self, name, url, titleTag)
                self.productNumberTag = productNumberTag
                self.priceTag = priceTag

class Article(Website):
        """Contains information for scraping an article page"""
        def __init__(self, name, url, titleTag, bodyTag, dateTag):
                Website.__init__(self, name, url, titleTag)
                self.bodyTag = bodyTag
                self.dateTag = dateTag

**********************************************************************************************
```

To Start New Project Using Scrapy:



****