Chapter 13: Scraping with Python 3 (portion from Python 3 Book Ami Gates)

Gathering data can range in process from having a corporation or institution hand you clean and organized data, to scraping and then mining dirty and disorganized data directly from the web. This chapter will focus on collecting or gathering data from the web – **web scraping**.

While scraping generally refers to gathering data, such as from a database, a local machine, or the web, these days it tends to refer to web scraping. Web scraping is the activity of gathering data or information directly from the web, via page sources. Web scraping can be accomplished using one or more of the following methods: an **API** (Application Program[ing] Interface) that is provided by the Website (such as Google's Map API), Python (or other coding language), and the use of special packages (such as **Scrapy** or **BeautifulSoup**). We will talk more about methods below.

Given the profound rise in the creation and use of data, there are many words that share some conceptual overlap and relationship, such as data bases, data mining, web crawling, data munging, data wrangling, data visualization, prediction, inference, description, reduction, differentiation, and so on. There are also a large number of further techniques and methods that range from statistical analysis to machine learning that are employed to better understand and represent data.

Data is almost anything; it can be visual, graphical, audio, text, hyperspectral, remote, video, etc. **Sampling** is the idea of collecting subsets of data, and there are many general sampling methods such as random or stratified. **Web scraping** is the activity of gathering or collecting data from the web. A **database** is a collection of tables that contain organized and usually clean data. Many databases are relational and reflect at least some of the known relationships between the data. **Data mining** is the act of searching through and analyzing usually clean data for patterns or relationships. **Web crawling** is a method that uses a program (also called a bot for robot) to "crawl" through every level of the hierarchy of one or more webpages to either search for a request or index certain content. **Data munging**, **data wrangling**, and **data cleaning** are terms that describe formatting, organizing, correcting, removing errors or anomalies, dealing with missing values, and preparing data for analysis. These terms fall under the category of **data preparation**. In Python 3, the **pandas package** is often used for such data preparation. pandas and related Python 3 Modules will be covered in a different chapter.

Scrapers and Regular Expressions

A **scraper** is any bit of code that can be used (often in conjunction with regular expressions or a tool such as BeautifullSoup) to scrape data. Python 3 (the topic of this book) can be used to create very effective scrapers. There are a few Python 3 modules that are required to create an effective scraper.

Example 1: Using the **`requests`** import

```
# Basic WebScraping
#Author: Ami Gates
import requests
response=requests.get("http://www.mathandstatistics.com/introduct
ion-to-statistics-video-moot")
```

```
txt = response.text
print(txt)
```

This example imports **`requests`** and uses the **`get`** method to store all the HTML code and content for the site, `("http://www.mathandstatistics.com/introduction-to-statistics-video-moot"` into the variable called `response`. Next, the **`text`** method is used to convert the contents of the variable `response` into text so that the following statement can print the results.

The site, `("http://www.mathandstatistics.com/introduction-to-statistics-video-moot"` is my personal eBook for Statistics and Excel. The eBook site is filled with YouTube videos and links to other pages. As such, when you run the code above and view the output, you will see many links to YouTube videos and other pages. The output is HTML code – specifically – the HTML code that encodes that webpage.

The **`urllib`** module will also allow you to make requests to websites, to get the data (as HTML), and to manipulate the data.

Example 2: Using **`urllib`**

```
# Using urllib
import urllib
data=urllib.request.urlopen("http://www.mathandstatistics.com/int
roduction-to-statistics-video-moot")
thedata=data.read()
print(thedata)
```

The output for the above code will be identical to Example 1. In both cases, the HTML is requested using a GET from the same website. In the case of the **`urllib`** module, the methods **`request.urlopen`** are used to make the request and "get" the data. Then the read method is used to store the retrieved data into the variable called **`getdata`**, which is printed. Confirm that both examples return the same results.

Notice that both examples above retrieve the HTML source code "behind" the website in the request. This is called a GET. The GET option is the most basic and simple way to access the HTML (or source) of a webpage. However, in many cases, you will want to send (or post) information to the webpage so that it returns a more specific result. This might include sending login information, requesting a specific stock price, or investigating hikes in Vail, CO. While the GET simply "gets" the entire web site that you included in the request via the URL, the POST method can be used to "send" information to the web server to affect the request.

Example 3: Using a POST

Many (if not most) websites allow the user to search for a specific item. That search is a type of POST. To see this directly, go to the website called, https://pythonprogramming.net and search for "Data Analysis". Figure 1 illustrates this step.

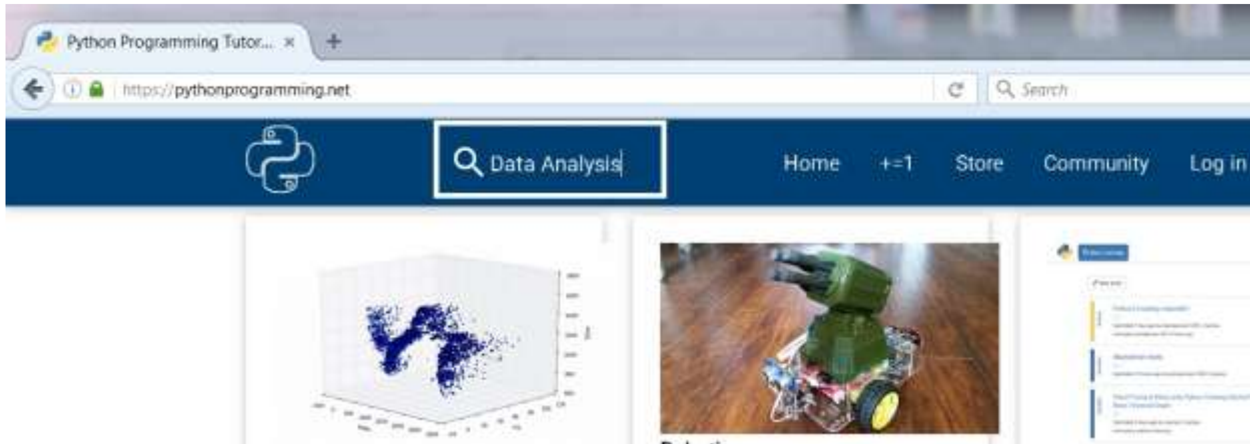


Figure 1: Typing "Data Analysis" into the search option on site, https://pythonprogramming.net

Next, press enter and look at how this updates the URL. The URL was https://pythonprogramming.net, but now it is https://pythonprogramming.net/search/?q=Data+Analysis. Figure 2 illustrates this update to the URL.

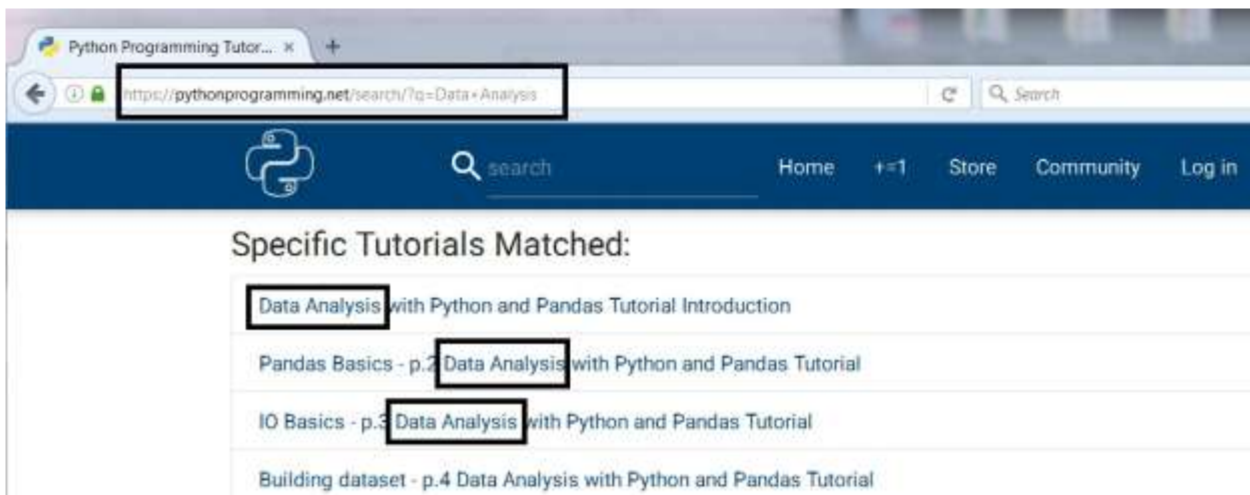


Figure 2: The POST becoming visible in the URL.

Take a close look at the updated URL: https://pythonprogramming.net/search/?q=Data+Analysis.

Notice the "`/search/?q=Data+Analysis`" portion. This is "posting" the following information: this is a search, the query (called q in this case) is Data and Analysis (the + represents logical AND). In other words, this tells the website to return search results for Data Analysis.

The "?" is used to begin the portion of the posted elements of the URL. So, the "?q=" tells the site to query all that follows.

Understanding how URLs communicate and post information to websites allows us to do this directly using Python 3. The following code will return the same results as HTML.

```
# Using urllib
# Author Ami Gates
import urllib
data=urllib.request.urlopen("https://pythonprogramming.net/search
/?q=Data+Analysis")
getdata=data.read()
print(getdata)
```

This will return the same page you get if you perform this search by hand on the site itself. If you review the HTML code returned by this small Python program, you will see all the references to the search strings, Data and Analysis.

So, just grabbing all the HTML behind a web URL is a GET and sending information to a website and grabbing the results is a POST.

Next, we will use the **`parse`** method from **`urllib`** to post information via Python, rather than hard coding it into the URL.

Example 4: urllib and parse

```
# Using urllib with parse and urlopen
#Author Ami Gates
import urllib
myURL ="http://www.georgetown.edu/"

#values are the data to post
# The query for Georgetown is “q”
# For multiple values in a search, use a list
values = {'q': ['analysis', 'data'],
          'submit':'search'}
results=urllib.parse.urlencode(values)
#Put data in bytes
results=results.encode("utf-8")
req = urllib.request.Request(myURL, results)
#visit the site with the values
resp = urllib.request.urlopen(req)
resp_results=resp.read()
print(resp_results)
```

The above example creates a **Python Dictionary structure** that contains the expected values that will have to be posted to the website. This example is the same as going directly to http://www.georgetown.edu/ and then performing a search for data analysis on the site. When you do this, look at the URL. While there are a lot of details in the URL, the key elements are that the URL uses “q” for query.

The **`values`** variable in the above code is a dictionary that associates "q" with a list of values that we want to query (search for). In this case, "q" is associated with the list of *data and analysis* (the order does not matter. In addition, the `values` dictionary associates the "submit" value with "search" (the action we want to perform).

The `urllib.parse.urlencode` method will "encode" the items in the `values` dictionary into the correct format to post to the website.

Because post data must be in bytes, the line of code: `results=results.encode("utf-8")` is required and converts the results into bytes (8 bits).

At this point, everything is "set up". The post values have been set in the values dictionary, the post values have been encoded, and the results have been converted to bytes. Next, `urllib.request.urlopen` is used to make this request to the website, and the result is collected, read, and printed with:

```
resp_results=resp.read()
print(resp_results)
```

As these examples suggest, Python scraping code must "match" requirements of the site(s) and post elements of interest. It is a good idea to visit sites and perform searches, etc. to see what the post URL format is.

Scraping Google without its API

Many websites have an API, which we will cover below. The use of APIs is recommended and respect and consideration for all websites is critical.

Although Google and other sites offer excellent and recommended APIs, it is still possible, in some cases, to scrape those sites directly. Google blocks "normal" web requests because they prefer that users utilize their API (which is recommended). However, at this time, Google can be accessed by "convincing" the Google server that we are not a program.

Example 5: Accessing Google - Using headers and files

```
#Author Ami Gates
import urllib
try:
        myURL="https://www.google.com/search?q=ANOVA"
        headers={}
        #This pretends we are not a robot
        #User Agent is one of the attributes of the headers
        headers['User-Agent'] = "Mozilla/5.0 (X11; Linux i686)
AppleWebKit/537.17 (KHTML, like Gecko) Chrome/24.0.1312.27
Safari/537.17"
        request=urllib.request.Request(myURL, headers=headers)
        response=urllib.request.urlopen(request)
        responseData=response.read()
```

```
        NewFile=open("DataFileScrape.txt", "w")
        NewFile.write(str(responseData))
        NewFile.close()
    except Exception as e:
            print(str(e))
```

This example shows a possible method for scraping Google without using its API. This is not recommended and is here only to show the method (and requirement) for using a special headers value so that Google does not think we are a robot. This "headers" options for "User-Agent" is complements of sentdex on YouTube.

This example also uses a file to place the results into. Using a file to collect output allows for further evaluation and parsing of the output with greater ease. Finally, this example is a reminder of the use of Try and except so that an exception will be thrown if an error occurs.

Not all websites allow scraping. In some cases, you can work around this - such as the example above (at least for now). In some cases, you must use the API. It is my view that it is best and most considerate to use APIs if they are available. We will talk about APIs shortly.

Regular Expressions and Mining Data

So far, we have seen a few options for getting and posting data. Web scraping generally results in lots of HTML, CSS, and JavaScript code, along with other content. The data collected is "messy" (though organized), hard to read, and certainly not in clean-data form. As a first step, the data collected can be further explored using **regular expressions** that will return specific items only.

In Python 3, the regular expression module is called **`re`**. A regular expression can be used to describe and locate a string of interest.

The next few figures illustrate basic rules for **identifiers, modifiers**, and **white space** for regular expressions in Python 3. In Figure 5, the \d represents any number. Figure 6 notes that the "+" means one or more. Therefore, \d+ would be one or more numbers (digits) of any kind.

```
Identifiers
\d any number
\D anything but a number
\s space
\S anything but a space
\w any character
\W anything but a character
.  any character, except for a newline
\b the whitespace around words
\. a period
```

Figure 5: Regular Expression Identifiers

```
Modifiers:
{1,3} we're expecting 1-3
+ Match 1 or more
? Match 0 or 1
* Match 0 or more
$ match the end of a string
^ matching the beginning of a string
| either or
[] range or "variance"
{x} expecting "x" amount
```

Figure 6: Regular Expression Modifiers

```
White Space Characters:

\n new line
\s space
\t tab
\e escape
\f form feed
\r return
```

Figure 7: Regular Expression White Space Characters

To use regular expressions, first import **`re`** in Python 3. Working with regular expressions takes a little practice, but is basically "pattern matching". If you are familiar with the format of the data (such as HTML) or specific words of interest, you can create a regular expression to find what you are looking for.

Example 6: Regular Expressions and Scraping with POST

```
# Author Ami Gates
import urllib
import re

def SearchGoogle():
    try:
        myURL="https://www.google.com/search?q=cookies"
        headers={}
        #This pretends we are not a robot
        #User Agent is one of the attributes of the headers
        headers['User-Agent'] = "Mozilla/5.0 (X11; Linux i686)
AppleWebKit/537.17 (KHTML, like Gecko) Chrome/24.0.1312.27
Safari/537.17"
        request=urllib.request.Request(myURL, headers=headers)
        response=urllib.request.urlopen(request)
        responseData=response.read()

        NewFile=open("DataFileScrape.txt", "w")
        NewFile.write(str(responseData))
        NewFile.close()
```

```
    except Exception as e:
            print(str(e))

def RegExp():
    NewFile=open("DataFileScrape.txt", "r")
    AllData=NewFile.readline()
    #print(AllData)
    SomeData=re.findall(r"chocolate\s\w+", AllData)
    print(SomeData)
    NewFile.close()

SearchGoogle()
RegExp()
```

The output for the above code is:

```
runfile('C:/Users/Ami/Documents/Python Scripts/urllib1.py',
wdir='C:/Users/Ami/Documents/Python Scripts')

['chocolate chip', 'chocolate chip', 'chocolate chip', 'chocolate
gourmet', 'chocolate chip', 'chocolate chip']
```

Example 5 above contains two functions. The first, called `SearchGoogle` performs a Google search with hard coded search parameter of “cookies”:

```
myURL="https://www.google.com/search?q=cookies"
```

Next, and as shown in Example 4, the `SearchGoogle` function performs the request and collects the results into a file.

Then, the function `RegExp` is called. This function opens the file that contains the results of the search and uses the following statement:

```
SomeData=re.findall(r"chocolate\s\w+", AllData)
```

This statement uses the **`re`** module and the **`findall`** method. Next, the “r” in the parenthesis signifies that this is a regular expression. The regular expression itself is looking for what word following the word “chocolate”. The following regular expression literally first finds the word chocolate and then a single space (\s) and then one or more characters (\w+) until it reaches any white space.

```
r"chocolate\s\w+"
```

The result which follows (and is also noted above) suggests that the most common word to follow chocolate is “chip”. This is not surprising as the Google search was for the word “cookies”.

```
['chocolate chip', 'chocolate chip', 'chocolate chip', 'chocolate
gourmet', 'chocolate chip', 'chocolate chip']
```

The best way to learn about regular expressions is to practice with them. Use any of the above methods to gather data in a file. Then see what you can extract from the file using different regular expression combinations. Before we move on to other methods, let's look at one more regular expression example.

The methods above can be combined with pre-knowledge of the HTML and/or encoding of a site in order to gather specific data from that site. The following example will illustrate that concept using finanace.yahoo.com.

Example 7: Scraping from a Source

For this example, we will start by visiting to the Site, http://finance.yahoo.com. Next, enter GOOG as the "ticker" or symbol for the Google stock into the search box. When you do this, you will see that the URL updates to include your search, and will read: http://finance.yahoo.com/quote/GOOG?p=GOOG.

Once on the Google Stock Page, you can see things like the current stock price, the opening price, the PE (Price per Earnings Index), the years range and so on. Next, view the **source** of the page by right-clicking and choosing "View Page Source". Note that I use a Windows system and am using Firefox as the Browser. There can be slight differences for MAC and for other Browsers.

Notice that the source starts with `"<!DOCTYPE html>"` which tells us that it is organized (tagged) as HTML. This is great as we can also use these HTML tags to search for information on this. By looking at the source, you can locate important content such as the stock price and the PE. However, because the full source is so huge, finding what you want can be a real challenge. The good news is that there is another option. On the GOOG page, for example, we can highlight the current price and then right-click it. Then choose "Inspect Element". This will highlight the following (on the day I wrote this):

```
<span class="Fw(b) D(ib) Fz(36px) Mb(-4px)" data-
reactid=".dywgtdvjp4.1.$0.0.0.3.1.$main-0-Quote-Proxy.$main-0-
Quote.0.1.0.$price.0">769.54</span>
```

Here you can see that the price (on the day I wrote this) was 769.54. More importantly, you can see some of the HTML tags that surround and identify the price. In other words, this is the specific HTML code that contains the "price" element. This same method can be used to inspect the HTML for the volume of the stock (the number of shares that changed hands thus far from the last time the stock market was open). Again, highlight and right-click the Stock Volume - choose Inspect Element - and you will get the following:

```
<td class="Ta(end) Fw(b)" data-test="TD_VOLUME-value" data-
reactid=".dywgtdvjp4.1.$0.0.0.3.1.$main-0-Quote-Proxy.$main-0-
Quote.2.0.0.0.0.1.1.0.0.$TD_VOLUME.1">1,109,732</td>
```

Now, let's write a small Python 3 Program that allows a user to enter any number of ticker symbols (stock symbols). The program will scrape the site and will return the current price for each stock entered.

Program ScrapeStock.py

```
#ScrapeStock.py
# Scrape and use re to get prices for stocks
# using finance.yahoo.com
# Author Ami Gates

import urllib
import re

def main():
    Num=eval(input("How many stock prices do you want?: "))

    for i in range(Num):
        Ticker=input("Enter Ticker: ")
        StockFile=open("MyStockFile2.txt", "w")
        StockFile.close()
        StockFile=open("MyStockFile2.txt", "a")
        MakeRequest(StockFile, Ticker)

    StockFile.close()



def MakeRequest(StockFile, Ticker="GOOG"):
    #----------using parse
    myURL = "http://finance.yahoo.com/quote/"+Ticker
    #myURL = "http://finance.yahoo.com/quote/GOOG?p=GOOG"
    #myURL = "http://finance.yahoo.com/quote/AAPL?p=AAPL"
    #values are the data to post
    values = {'p':Ticker}
    results=urllib.parse.urlencode(values)
    #Put data in bytes
    results=results.encode("utf-8")
    req = urllib.request.Request(myURL, results)

    #visit the site with the values
    resp = urllib.request.urlopen(req)
    resp_results=resp.read()
    StockFile.write(str(resp_results))
    #print(resp_results)
    GetPriceVol(StockFile, Ticker)
```

```
def GetPriceVol(StockFile, Ticker="GOOG"):
    StockFile.close()
    StockFile=open("MyStockFile2.txt", "r")
    AllText=StockFile.read()
    RegExp='Quote-Proxy\.\$main-0-
Quote\.0\.1\.0\.\$price\.0..(.+?)</span>'

    Pattern=re.compile(RegExp)
    Price=re.findall(Pattern, AllText)
    print("Ticker ", Ticker, "Price ", Price)


main()
```

A possible output for the above program is:

```
How many stock prices do you want?: 3

Enter Ticker: SBUX
Ticker  SBUX Price  ['57.29']

Enter Ticker: GOOG
Ticker  GOOG Price  ['769.54']

Enter Ticker: AAPL
Ticker  AAPL Price  ['106.94']
```

The above program, ScrapeStock.py, is not only an excellent Python3 review, but it also illustrates posting to a website, and using regular expressions to grab the information out of all the data collected in the request. It’s best to type this code in and run it. As long as Yahoo does not change its Site (which can happen) the code will run very well.

Within the program, the following line of code contains the regular expression used:

```
RegExp='Quote-Proxy\.\$main-0-Quote\.0\.1\.0\.\$price\.0..(.+?)</span>'
```

Working with regular expressions takes some practice. The first step is to look at the source code (the HTML in this case) that you want to use to grab information from.

Recall that by inspecting the stock price on the Webpage, we discovered the following:

```
<span class="Fw(b) D(ib) Fz(36px) Mb(-4px)" data-
reactid=".dywgtdvjp4.1.$0.0.0.3.1.$main-0-Quote-Proxy.$main-0-
Quote.0.1.0.$price.0">769.54</span>
```

The first thing to notice is the price itself – which is 769.54. Next, notice what surrounds the price. In the most simplistic view, the <span … surrounds the price. However, by searching the entire site source for <span … will return many results and will not give us the one price result we want.

The next thing to be aware of is that parts of this HTML can change and so would not make an accurate search criteria. With some experimentation, I discovered that the value,

```
reactid=".dywgtdvjp4.1.$0.0.0.3.1
```

changed on each call to the site. Therefore, this should not be included in the search for the price.

One of the "tricks" to seeing what portion of this HTML code will return a unique value (namely the price of the stock) is to perform a few experiments with a small hard-coded program such as the following:

```
##WorkingRE.py
## Gates
import urllib
import re

URL="http://finance.yahoo.com/quote/GOOG?p=GOOG"

Request=urllib.request.Request(URL)
Response=urllib.request.urlopen(Request)
Results=Response.read()

File=open("dataFile.txt", "w")

File.write(str(Results))
File.close()

File=open("dataFile.txt", "r")
AllData=File.read()
File.close()

RegularExp="Quote-Proxy\.\$main-0-Quote\."

NewPattern=re.compile(RegularExp)
Output=re.findall(NewPattern,AllData)

print(Output)
```

This small program hard-codes the URL, makes the post request, and saves all the site source data to a file. Once you have the data in a file, the goal is to create a regular expression that returns the stock price for Google (GOOG) in this case.

Notice the line of code:

```
RegularExp="Quote-Proxy\.\$main-0-Quote\."
```

This is a starting point. You will update this line of code until the program returns the stock price.

If you run this code as-is, it will print 'Quote-Proxy.$main-0-Quote.' many many times which tells you that you have not yet found the unique string to identify the stock price.

Update `RegularExp` to the following and see what happens.

```
RegularExp="Quote-Proxy\.\$main-0-Quote\.0\.1\.0"
```

This is the output:

```
['Quote-Proxy.$main-0-Quote.0.1.0', 'Quote-Proxy.$main-0-
Quote.0.1.0', 'Quote-Proxy.$main-0-Quote.0.1.0', 'Quote-
Proxy.$main-0-Quote.0.1.0', 'Quote-Proxy.$main-0-Quote.0.1.0',
'Quote-Proxy.$main-0-Quote.0.1.0', 'Quote-Proxy.$main-0-
Quote.0.1.0']
```

That is great because this is very short and so we are getting close to making the search unique. As a side note, notice that I use the "\" to place in front of special regular expression symbols, such as "." and "$". When I want "." to just be ".", I need to write it as "\.". Otherwise, "." means "any character except newline". See the re rules above for a reminder.

Finally, change `RegularExp` to the following:

```
RegularExp=RegularExp="Quote-Proxy\.\$main-0-
Quote\.0\.1\.0\.\$price\.0"
```

When you run the code, you get only one output, namely:

```
['Quote-Proxy.$main-0-Quote.0.1.0.$price.0']
```

This means that we have what we need to get the price. The final regular expression that will return the price itself is:

```
RegularExp='Quote-Proxy\.\$main-0-Quote\.0\.1\.0\.\$price\.0..(.+?)</span>'
```

The (.+?) will return anything in that exact location between the string to the left, which we just proved to be unique, and the end of the span to the right.

Run the above code with equal to:

```
RegularExp='Quote-Proxy\.\$main-0-Quote\.0\.1\.0\.\$price\.0..(.+?)</span>'
```

The output will be current price of Google (check it!)

So far, the above examples have covered how to access data from the web, save to a file, and how to create regular expressions to parse and grab given values. Scrapers are often very specific to the site or sites that they scrape. If the site changes, the scraper will have to be updated or it may not work. It is also clear that creating accurate regular expressions and parsing data from sources can be challenging and messy. The next section will look at tools for parsing data.

Parsing Tools

There are several tools available for parsing data.

BeautifulSoup

BeautifulSoup is a Python Library that can be used to grab data from HTML (or XML) files. What makes BeautifulSoup excellent is that it allows (with much greater ease) the extraction of information from websites without the need for complicated regular expressions.

To see this, let’s take the same stock example from above, but let’s now use BeautifulSoup to get the stock prices.

Example 7: Using BeautifulSoup and urllib to scrape Stock Prices

Before we look at and discuss the code, let’s first look at the finance.yahoo.com site and the code for the stock price. Figure 8 illustrates the result of highlighting an element of interest on a website, right-clicking, and choosing *Inspect*. Note that Browsers have uniqueness, such as Firefox calls this “Inspect Element” and Chrome calls it “Inspect”. In any case, the goal is to view the code specific to the element of interest (in our case the stock price).

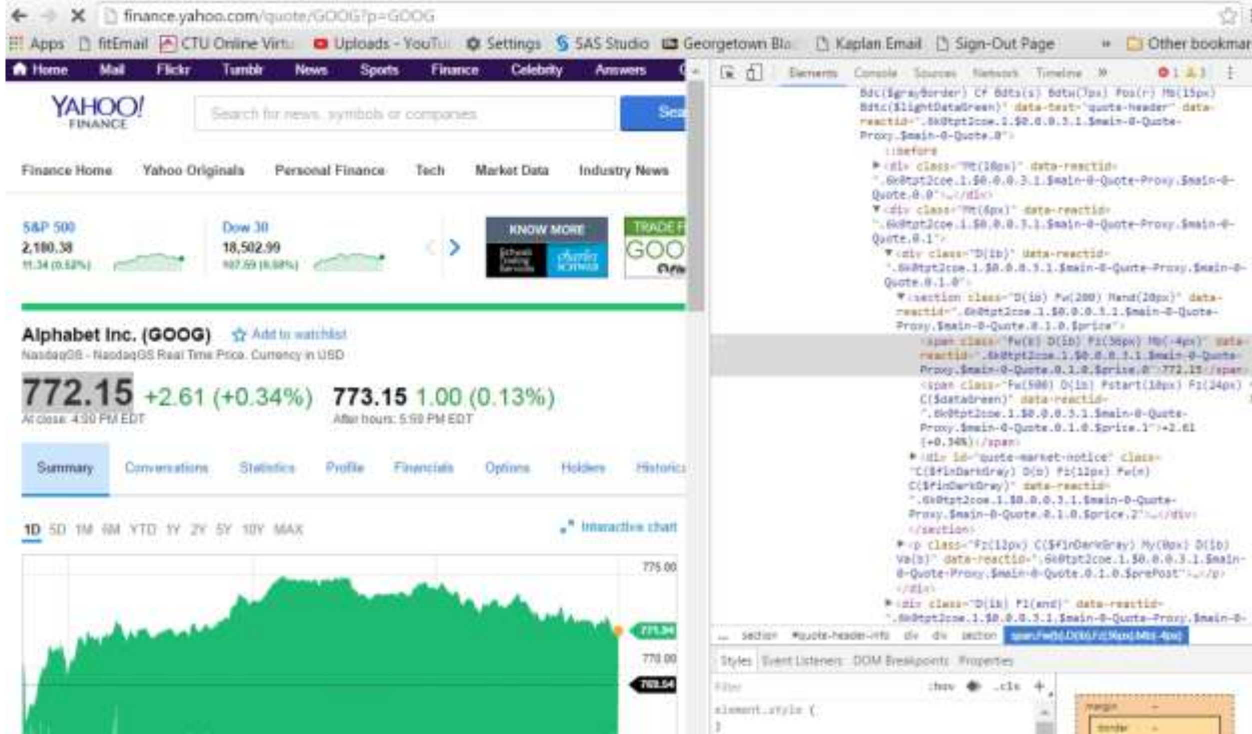


Figure 8: An Inspection of the code for the GOOG stock price on the finance.yahoo.com site

On Figure 8, the gray shaded code represents the specific HTML content that displays the stock price. The code is:

```
<span class="Fw(b) D(ib) Fz(36px) Mb(-4px)" data-
reactid=".6k0tpt2coe.1.$0.0.0.3.1.$main-0-Quote-Proxy.$main-0-
Quote.0.1.0.$price.0">772.15</span>
```

In the examples above, we used regular expressions to find and collect the information we wanted.

Example 8: BeautifulSoup

In this example we will instead use **BeautifulSoup**. The following code performs the same function as the code in the last example. It scrapes *finance.yahoo.com* for stock prices. This program makes a post request to *finance.yahoo.com,* then uses BeautifulSoup to read the source of the site and save the contents to a file. Next, the code opens the file and again using BeautifulSoup is locates the stock price. It is not necessary to write the contents to a file. The code below also contains a line (commented out) that allows for direct scraping of the web, without writing or reading from a file.

Within the code below, note the two lines of code:

```
soup2=BeautifulSoup(StockFile.read(), "lxml")
```

```
price2=soup2.find("span", {"class":"Fw(b) D(ib) Fz(36px) Mb(-
4px)"}).contents
```

The first line above opens the file (which now contains the source code from the website) using BeautifulSoup and format **lxml**. The contents are saved to variable, soup2. Next, the **`find`** method is used to locate the stock price.

Recall, the HTML source itself is:

```
<span class="Fw(b) D(ib) Fz(36px) Mb(-4px)" data-
reactid=".6k0tpt2coe.1.$0.0.0.3.1.$main-0-Quote-Proxy.$main-0-
Quote.0.1.0.$price.0">772.15</span>
```

As such, we are looking for the contents of the **`span`** TAG whose *class* is "**`Fw(b) D(ib) Fz(36px) Mb(-4px)`**". Thus, the `find` method looks for:

```
soup2.find("span", {"class":"Fw(b) D(ib) Fz(36px) Mb(-4px)"}).contents
```

Review the following code to see several steps.

```
# Using urllib and BeautifulSoup
# Author Ami Gates

import urllib
from bs4 import BeautifulSoup

def main():
    Num=eval(input("How many stock prices do you want?: "))

    for i in range(Num):
        Ticker=input("Enter Ticker: ")
        StockFile=open("MyStockFile3.txt", "w", encoding="utf-8")
        MakeRequest(StockFile, Ticker)

    StockFile.close()



def MakeRequest(StockFile, Ticker="GOOG"):
    #----------using parse

    myURL = "http://finance.yahoo.com/quote/"+Ticker
    #myURL = "http://finance.yahoo.com/quote/GOOG?p=GOOG"
    #myURL = "http://finance.yahoo.com/quote/AAPL?p=AAPL"
    #values are the data to post
    values = {'p':Ticker}
    results=urllib.parse.urlencode(values)
    #Put data in bytes
    results=results.encode("utf-8")
```

```
    req = urllib.request.Request(myURL, results)
    #req = urllib.request.Request(myURL)

    #visit the site with the values
    resp = urllib.request.urlopen(req)

    #Soup It
    soup=BeautifulSoup(resp.read(), "lxml")
    StockFile.write(str(soup))
    StockFile.close()
    #print(soup.prettify()[1:100])

    #Note - the element from the source of the site for the stock price is
    # <span class="Fw(b) D(ib) Fz(36px) Mb(-4px)" data-
reactid=".6k0tpt2coe.1.$0.0.0.3.1.
    #$main-0-Quote-Proxy.$main-0-Quote.0.1.0.$price.0">772.15</span>

    ##*********This line of code will get the stock price directly from the
website
    #price=soup.find("span", {"class":"Fw(b) D(ib) Fz(36px) Mb(-
4px)"}).contents

    StockFile=open("MyStockFile3.txt", "r", encoding="utf8")
    soup2=BeautifulSoup(StockFile.read(), "lxml")
    price2=soup2.find("span", {"class":"Fw(b) D(ib) Fz(36px) Mb(-
4px)"}).contents
    StockFile.close()
    ##"data-reactid":".6k0tpt2coe.1.$0.0.0.3.1.$main-0-Quote-Proxy.$main-0-
Quote.0.1.0.$price.0"})
    #print(type(soup))
    #print(resp_results)
    ##--GetPriceVol(StockFile, Ticker)
    print("The Ticker is ",Ticker, " Price: ", price2)


main()
```

This example illustrates the power and ease of using BeautifulSoup rather than creating the regular expressions. However, in both cases, you must be familiar with the source code (in this case HTML) and what you are looking for.

lxml

lxml is a library (http://lxml.de) in Python that allows for parsing of HTML and XML. lxml is written in C and BeautifulSoup is written in Python. Beautifulsoup support Python's standard HTML parser, but it also supports lxml as a parser – which offers greater speed. Like lxml, html5lib is also another parser option.

Because BeautifulSoup has the reputation of being more robust, especially in the face of poorly formed HTML pages, it is widely used and versatile tool. As the code above illustrates, BeautifulSoup can use the lxml parser option.

Example 9: Beautiful Soup and csv

This next example program illustrates how to scrape table HTML from a website and create a csv (comma separated values) file with the data. Keep in mind always that scrapers only work as long as the website remains the same. If the owner of the site makes a change, the scraper must be updated to manage that alteration. In this example, the first program uses `urllib` and the second uses `requests`. However, they both do the same thing.

```
#Sports Data with BS and CV
# USING urllib
#Author Gates and Singh
# Additional Ref: “Web Scraping with Python, Mitchell”

from bs4 import BeautifulSoup
from urllib.request import urlopen
import csv

def Sports():
    csvName = "TestCSV.csv"

url="http://www.thehuddle.com/stats/2006/plays_weekly.php?week=1&
pos=wr&col=FPTS&ccs=6"
    page=urlopen(url)
    soup = BeautifulSoup(page, "lxml")
    table=soup.findAll("table")[0]
    All_TR=table.findAll("tr")
    csvFile=open(csvName, "wt")
    playerwriter = csv.writer(csvFile, delimiter=',')
    playerwriter.writerow(['Player', 'Team', 'Plays', 'Fpts',
'Run', 'Ryd', 'RunTD', 'Pass', 'Cmp',
    'Pyds', 'PTD', 'Fum', 'Int'  ])
    for nextTR in All_TR:
        csvRow=[]
        for nextTD in nextTR.findAll("td"):
            csvRow.append(nextTD.text.strip())
        playerwriter.writerow(csvRow)
    csvFile.close()

Sports()
```

The above program generates a file called, "TestCSV.csv". Inside that file is a table that has 13 columns (from Player to Int) and a row for each Player that populates information for each of the 13 variables. The `csv` methods, `writer` and `writerow` are utilized as it beautiful soup.

This next program is the same in goal and result as the program above. However, it uses `requests` rather than `urllib`.

```
#Sports Data with BS and CV
#Author Gates and Singh
# Additional Ref: "Web Scraping with Python, Mitchell"
# USES requests

from bs4 import BeautifulSoup
import requests
import csv

def Sports():
    csvName = "TestCSV2.csv"

url="http://www.thehuddle.com/stats/2006/plays_weekly.php?week=1&
pos=wr&col=FPTS&ccs=6"
    page=requests.get(url)
    soup = BeautifulSoup(page.text, "lxml")
    table=soup.findAll("table")[0]
    All_TR=table.findAll("tr")
    csvFile=open(csvName, "wt")
    playerwriter = csv.writer(csvFile, delimiter=',')
    playerwriter.writerow(['Player', 'Team', 'Plays', 'Fpts',
'Run', 'Ryd', 'RunTD', 'Pass', 'Cmp',
    'Pyds', 'PTD', 'Fum', 'Int'  ])
    for nextTR in All_TR:
        csvRow=[]
        for nextTD in nextTR.findAll("td"):
            csvRow.append(nextTD.text.strip())
        playerwriter.writerow(csvRow)
    csvFile.close()

Sports()
```

A Note about XPath and XSLT

When I was a lot younger and data science had not quite picked up its current popularity, XML and XSLT were common tools for document organization and parsing. XPath is an element in the XSLT standard and can be used to crawl through XML documents, elements, and attributes. XPath contains a library of functions and methods that allow it to navigate XML and to select specific items. While this book does not delve into XML or XPath, they are similar in concept to HTML and BeautifulSoup.

Using APIs

Some websites offer an API (Application Program[ing] Interface) for gathering data from their site. Using an API can be easier in some cases and it also protects the website from misuse. Before we jump into a few API examples, let's first define and look at **JSON, REST[ful]**, and **SOAP**.

JSON

JSON stands for JavaScript Object Notation. The information in this paragraph is taken from the official JSON site (http://www.json.org/), where further details can be obtained as needed. JSON is a data-exchange. While it is easy for us to read, it is also easy for computers to parse and render. It is a text format that is language independent.

JSON is built on two structure types. First, a collection of name and value pairs. In Python, this type of name and value pair structure can be realized as a dictionary object. Second, an ordered list of values. In Python this might be a list or numpy array.

JSON, in concept, is very much like XML. It offers structure to data and information so that it can be parsed, searched, and rendered. This next example is taken from http://json.org/example.html and compares basic JSON mark-up with XML for the same information. Figure 9 illustrates.

In JSON

```
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}}
```

In XML

```
<menu id="file" value="File">
  <popup>
    <menuitem value="New" onclick="CreateNewDoc()" />
    <menuitem value="Open" onclick="OpenDoc()" />
    <menuitem value="Close" onclick="CloseDoc()" />
  </popup>
</menu>
```

Figure 9: JSON Versus XML for Mark-up

SOAP, REST, and Web APIs

Both SOAP (Simple Object Access Protocol) and REST (Representative State Transfer) are web services. They are also thought of as architectural styles for network applications and communications. They offer an interface or communications protocol between clients (seeking data for example) and the server (where the website with the data is stored).

SOAP is an older model that can be more robust, but also more complicated to use. SOAP focusses on "operations" or actions that can be taken between the client and server. SOAP is standardized, has built-in error handling, is extensible, and is language, platform, and transport independent. SOAP uses only XML for messages.

Unlike SOAP (or others like CORBA or RPC), REST offers a simplified method for accessing web servers, and uses HTTP to make calls (communications) between clients and servers. Systems that use REST are often referred to as **RESTful**. REST uses HTTP for all four CRUD (Create (POST)/Read (GET)/Update (PUT)/Delete) GET and POST operations. REST is also platform independent and offers many other features that fall outside of the scope of this book.

The key to knowing about SOAP and REST is that they are used by APIs. TWITTER, Google, Flickr, FaceBook all use REST. A **Web API** is an interface defined on a request-response (client-server) system.

Simply put, a **web API** is a program that uses HTTP to make a request and get a response. The response is usually in JSON or XML (see example 9 above). HTTP offers methods such as GET and POST as shown in several examples above. Web APIs, Scrapers, and regular expressions can all be combined to add robustness to data gathering.

API Examples and Methods

Several website offer APIs for accessing data from their servers (sites). A few include Wikipedia, Google (including YouTube and Geocoding) (https://developers.google.com/apis-explorer/#p/), Twitter, Facebook, Amazon, and Ebay. Many site use **OAuth** as an authentication and access method.

Google API Example From Google Maps: Timezones

Using Google's APIs takes a bit of getting used to. To use Google APIs, you will need a "key" that Google supplies. You will also need to ENABLE the API you wish to use. Before you start, be sure that you have a Google Account. You can do this by creating a new gmail account or using one that you already have.

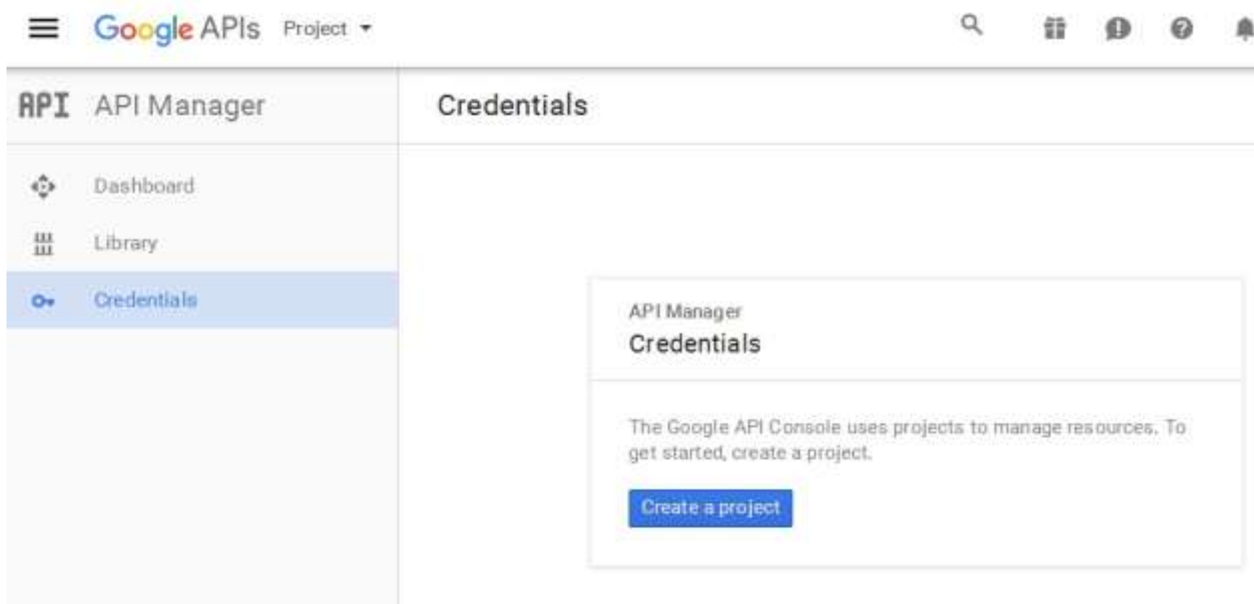
Next, go to, https://console.developers.google.com/apis/credentials?project=phrasal-ground-142018&pli=1 and sign in with your gmail login information.

From here, you can complete the OAuth and you can create credentials. The following several images will illustrate the steps that you will take on the Google Developer Site. Within the following steps, I create a "new project", a "Key", and I "ENABLE" the API that I want to use. I also locate the base format

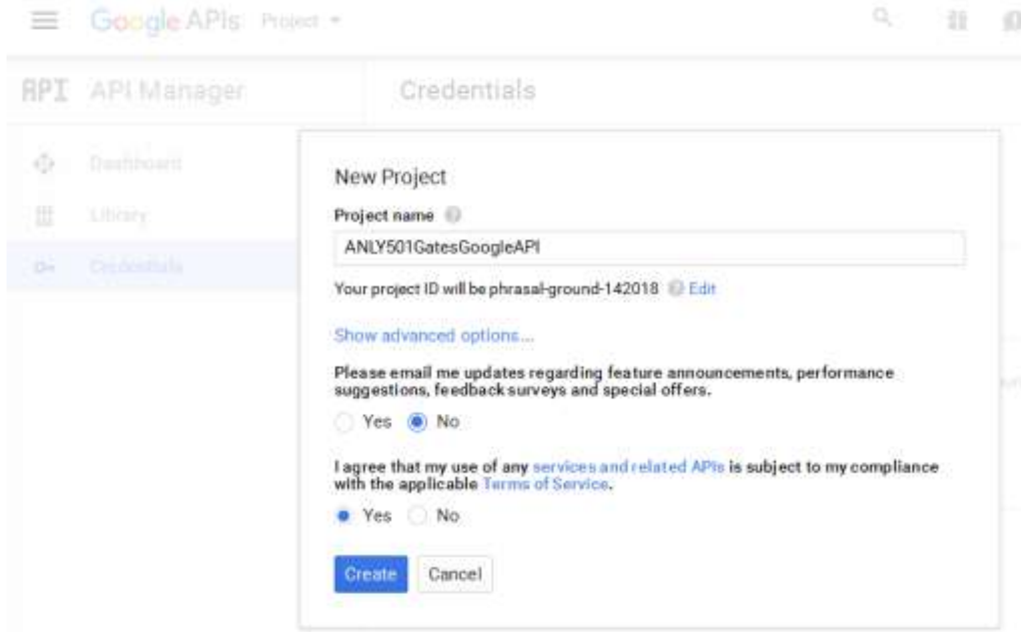
for the API and then review the rest of the site for details. One you complete the steps, find the Google Maps Timezones API and use the code included to test the methods and your credentials. Note that I will have to hide my Google KEY (because its mine) and you will replace it with yours.

The following Image-Set will illustrate many of the steps you will see in the Google Developer Console area as you create your credentials.

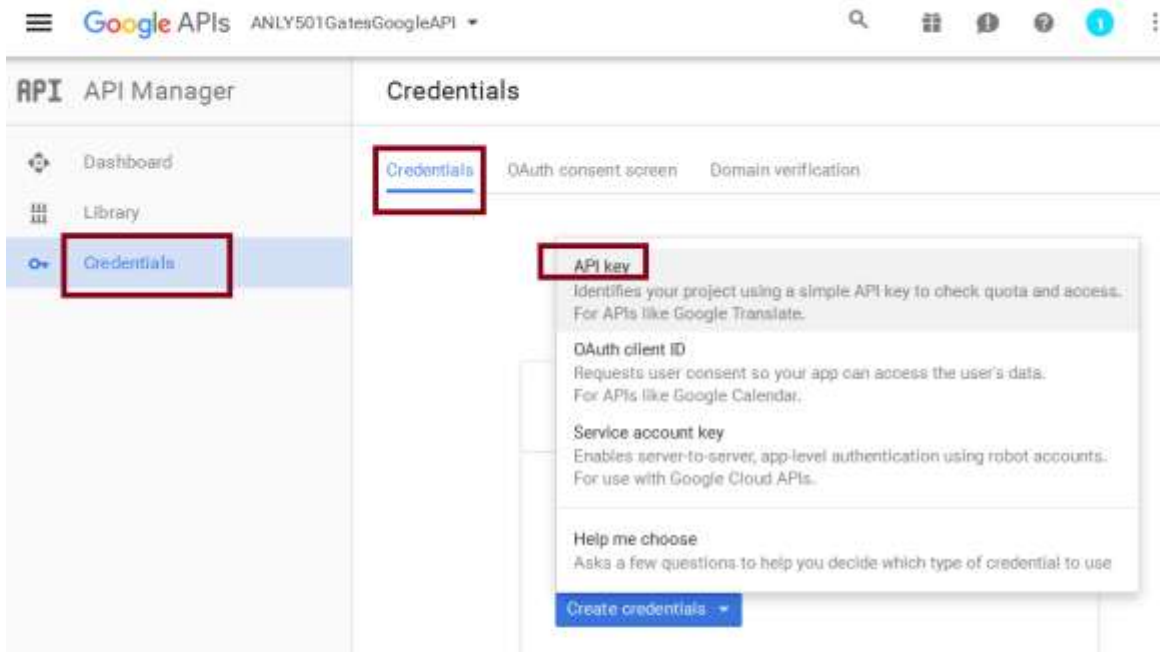
Step 1: Create a "New Project"



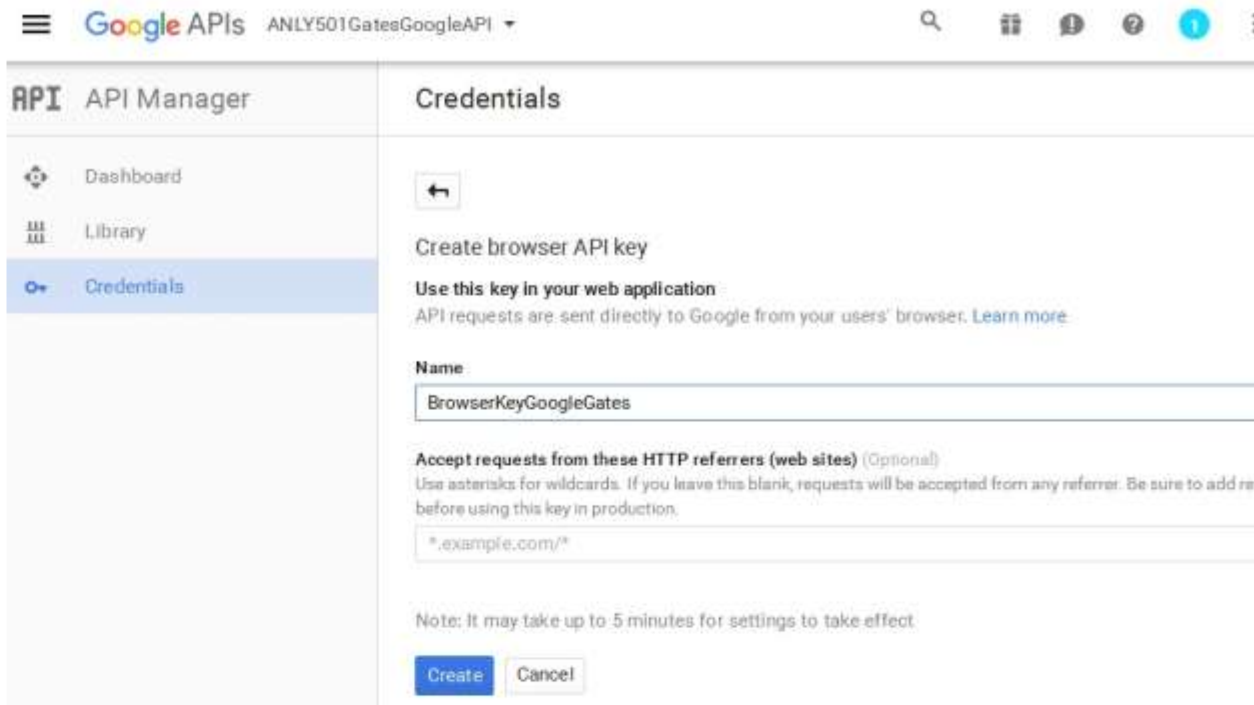
Step 2:

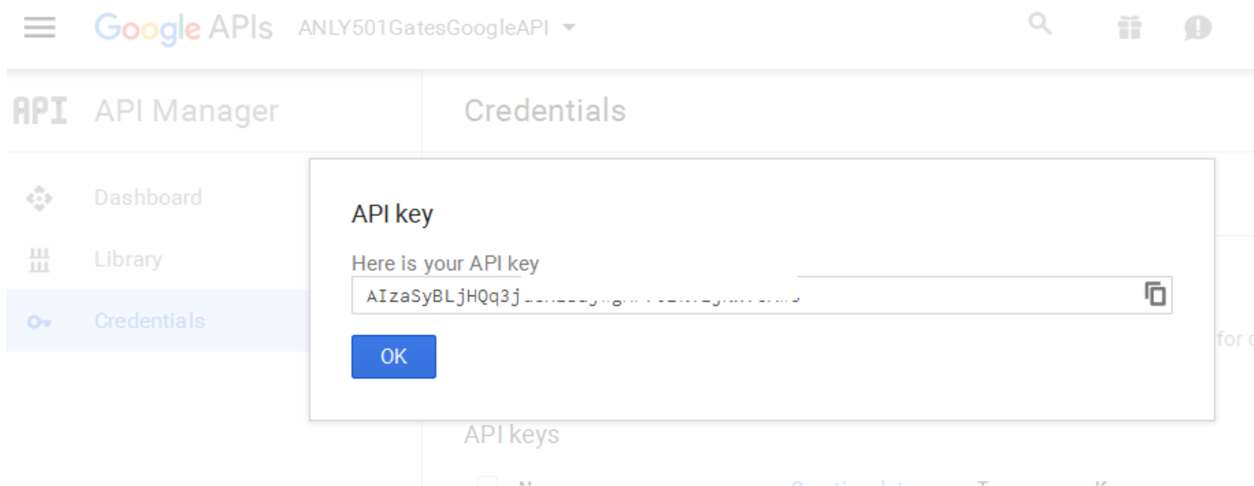


Note that my New Project name is: "ANLY501GatesGoogleAPI". This images shows the next steps in creating the new project and its ID (which is default unless you edit it).

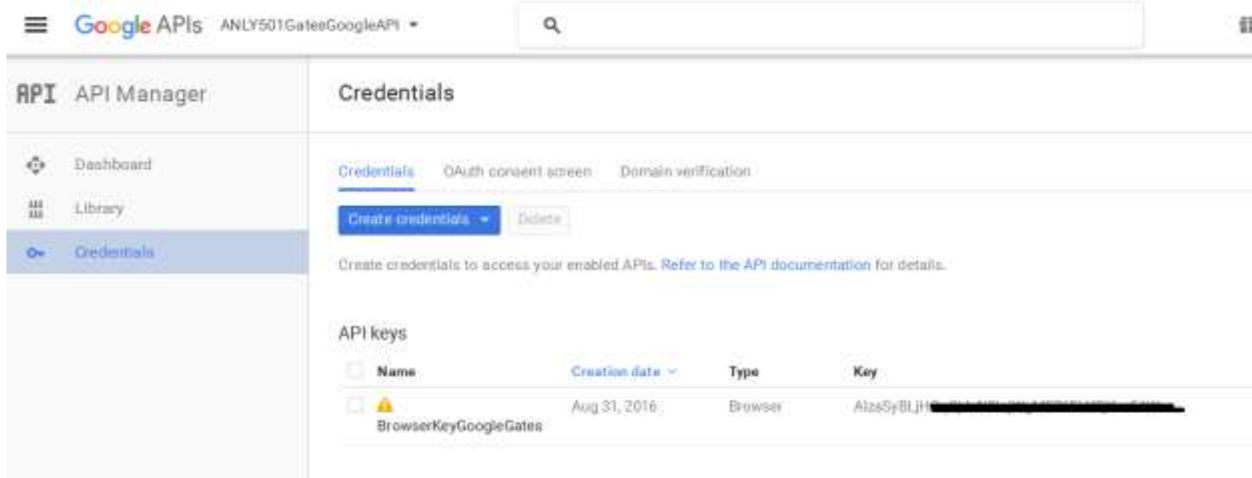


Next, click the blue "Create credentials" tab and selected API Key, which will be required for accessing the API.





Google then created a Key.



Once done, you can see your key name and number on the your "Credentials" area as long as you are logged in.

Example 10: Using the Google Maps Timezone API

NOTE: The following is the exact URL that will request the desired information.

```
https://maps.googleapis.com/maps/api/timezone/json?location=39.6034810,-119.6822510&timestamp=1331161200&key=YOURKEY
```

```
# Author Ami Gates
# GoogleMapsAPI TImezones

import requests
import urllib

maps_key = 'YOUR KEY'
timezone_base_url =
'https://maps.googleapis.com/maps/api/timezone/json'
lat=64
lng=-21
timestamp='1331161200'


def F1(lat,lng, timezone_base_url,maps_key, timestamp):
    # This joins the parts of the URL together into one string.
    url = timezone_base_url + '?' + urllib.parse.urlencode({
        'location': "%s,%s" % (lat, lng),
        'timestamp': timestamp,
        'key': maps_key
    })
    #print(url)
```

```
    response = requests.get(url)
    results = response.json()['timeZoneId']
    #results = response.json()
    print(results)

F1(lat,lng,timezone_base_url,maps_key, timestamp)
```

The output to the above code is:

```
Atlantic/Reykjavik
```

Pit Falls for Using APIs and notes about Google APIs

Many of the challenges in using APIs result from not having "exactly" what the API is expecting. In other words, the URL that is applied to the GET (via `response = requests.get(url)` for example) must be formatted exactly as the API expects it. If it is not, an error may occur. Each API has its own "format" to adhere to. In some cases, the URL must also contain expected parameters. For example, when using the Google Timezone API, the UNIX timestamp must be included or the API will not function:

```
url = timezone_base_url + '?' + urllib.parse.urlencode({
        'location': "%s,%s" % (lat, lng),
        'timestamp': timestamp,
        'key': maps_key
})
```

Google APIs must also be ENABLED before each use. There are also several options to choose from for gathering and storing the results.

Wikipedia API and Endpoints

Like Google, Wiki also offers an API. Wiki's API site is, https://www.mediawiki.org/wiki/API:Main_page. There, you will find the **endpoint** (a homepage for API service) link. The Wiki endpoint (or base url) is, https://en.wikipedia.org/w/api.php. Next, the site offers the format for an API request:

```
https://en.wikipedia.org/w/api.php?action=query&titles=Main%20Page&prop=revis
ions&rvprop=content&format=json
```

Note that "format" is set to "json". The parameters for the URL include: "action", "titles", "prop", "rvprop", and "format".

Details about options and parameters can be found here:
https://en.wikipedia.org/w/api.php?action=help&modules=main.

Example 11: Using Wiki API and JSON

```
# WIKI API
# Author Ami Gates

import json
import urllib
from urllib.request import urlopen

def WIKI():

    WikiFile = open("WikiDataFile.txt", "w", encoding="utf-8")

#URL="https://en.wikipedia.org/w/api.php?action=query&titles=Main%20Pag
e&prop=revisions&rvprop=content&format=json"
    Base_URL="https://en.wikipedia.org/w/api.php"
    URL = Base_URL + '?' + urllib.parse.urlencode({
        'action': 'query',
        'prop': 'revisions',
        'rvprop': 'content',
        'titles': 'Frank Sinatra',
        'id': 'Legacy_and_honors',
        'imlimit':'1',
        'class':'infobox biography vcard',
        'format': 'json'
    })

    response=urlopen(URL).read().decode('utf-8')
    JSONresp=json.loads(response)
    #print(JSONresp.keys())
    #print(JSONresp)

    ##Note - this can also be read as BS
    #html = urlopen(URL)
    #soup = BeautifulSoup(html, "lxml")
    #print(soup.decode())

    WikiFile.write(str(JSONresp))
    WikiFile.close()
    WikiFile = open("WikiDataFile.txt", "r", encoding="utf-8")
    Data=WikiFile.read()
    #print(type(Data))
    print(Data[1:100])
    WikiFile.close()



WIKI()
```

The output for this example is:

```
{'query': {'pages': {'11181': {'ns': 0, 'title': 'Frank Sinatra',
'pageid': 11181, 'revisions': [{'*
```

Note that the output includes only 101 characters per the line of code above:

```
print(Data[1:100])
```

In Example 10, the Wiki API is used. A dictionary structure is set up to populate the parameters for the URL, such as 'action': 'query'. The search in this case, titles': 'Frank Sinatra', focuses on Frank. The results are returned as JSON. They are converted to string type and saved to a text file for further analysis.

The AirNow (U.S. Environmental Protection Agency (EPA)) API

In this next example, rather than using a more common API such as Google or Twitter, I decided to start with a question: "Can I collect good air data"? Next, I did a quick Google search for "air pollution API" and I quickly found, "AirNow Developer Tools" (https://docs.airnowapi.org/).

AirNow requires API users to create an account (https://docs.airnowapi.org/login). I followed the steps, created and activated the account, and then logged in. Next, I reviewed the Site thoroughly to learn about how air quality is measured and how to use the API.

By reading the Site, I determined how to get an API "key" which is required for their Site (https://docs.airnowapi.org/faq). As with most Sites, there are data use guidelines and rate limitations. In addition, the AirNow site offers a different alternative if the user wishes to populate a database rather than perform a few end-user queries (ftp.airnowapi.org).

From the Web Services link (https://docs.airnowapi.org/webservices), I selected to investigate current or historical forecasted AQI values by latitude and longitude (https://docs.airnowapi.org/forecastsbylatlon/query).

The Forecast By Latitude and Longitude Site allows for practice with the URL format and requirements.

The example on the Site is Lat 39 and Lng -121. The URL to make this request is shown as:

```
http://www.airnowapi.org/aq/forecast/latLong/?format=text/csv&latitude
=39.0509&longitude=-121.4453&date=2016-09-
01&distance=25&API_KEY=D9AA91E7-070D-4221-867C-EFF5E0D8C2C7
```

This shows my KEY and the overall parameters of the request. If I use their tool on the site to run this example request, the response is:

"DateIssue","DateForecast","ReportingArea","StateCode","Latitude","Longitude","ParameterName","AQI","CategoryNumber","CategoryName","ActionDay","Discussion"

"2016-09-01 ","2016-09-01 ","Yuba City/Marysville","CA","39.1389","-121.6175","O3","-1","2","Moderate","false",

"2016-09-01 ","2016-09-02 ","Yuba City/Marysville","CA","39.1389","-121.6175","O3","-1","2","Moderate","false",

I also investigated several other query tools and URL formats on the AirNow Web Service Site. My next step is to create Python 3 code that will make certain requests and collect the responses. As a side note, not all locations have an AQI detection method set up and so will not return a complete response.

Example 12: AirNow API and Python 3

```
# AirNow API Python Code
# Author Ami Gates
import urllib
from urllib.request import urlopen

def AirNow():
##http://www.airnowapi.org/aq/forecast/zipCode/?format=text/csv&zipCode
=20007&date=2016-09-01&distance=25&API_KEY=D9AA91E7-070D-4221-867C-
EFF5E0D8C2C7
    File=open("AirNow.txt", "w", encoding="utf-8")
    File.close()
    baseURL="http://www.airnowapi.org/aq/"
    (latitude,longitude) = (39,-77)
    date='2016-09-01'
    miles=25
    zipcode="20007"

    LatLngURL=baseURL + "forecast/latLong/?" + urllib.parse.urlencode({
        'format': "text/csv",
        'latitude': latitude,
        'longitude': longitude,
        'date':date,
        'distance':miles,
        'API_KEY':'D9AA91E7-070D-4221-867C-EFF5E0D8C2C7'
    })
    #print(LatLngURL)
    zipURL=baseURL + "forecast/zipCode/?" + urllib.parse.urlencode({
        'format': "text/csv",
        'zipCode':zipcode,
        'date':date,
        'distance':miles,
        'API_KEY':'D9AA91E7-070D-4221-867C-EFF5E0D8C2C7'
    })
    #print(zipURL)
    File=open("AirNow.txt", "w")
    for URL in [LatLngURL,zipURL]:
```

```
        response=urlopen(URL).read().decode('utf-8')
        File.write(response)

    File.close()

AirNow()
```

The output of this program is a text file that is comma delimited. As such, the file can be moved to your desktop (for example) and can be opened with Excel for other viewing (as desired).

Example 13: JSON and APIs

This next example will use the Air Now API (https://docs.airnowapi.org/) and JSON to grab AQI data from the Air Now Site. This data will be placed into a file. First, review the following code. Notice that the response from the AirNow site will be in JSON. As such, we can collect the values from the JSON output by referencing the keyword. For example, within the JSON response returned, the keyword, “StateCode” will reference the state.

```
# AirNow Example Ch 13
# API AirNow JSON request
# Author Ami Gates

import requests

def main():
    AirNow()


def AirNow():

    # The Latitude  - Lat - and Longitude - Long will
    # be used to read in data from Air Now for AQI PM2.5 and Ozone
    Lat = []
    Long = []

    # Read Lat and Long data from file into list
    LatLong = open("LatLong.txt", "r")
    #The LatLong.txt must have this format:
    # 34, -56.7
    # 21, -121
    # etc. where the lat is first then comma then long


    for line in LatLong:
        if "\n" in line:
            #Remove the newline from each line
            line=(line.split(sep="\n"))[0]
        #Grab the Latitude from the line
        Lat.append(float((line.split(sep=","))[0]))
```

```
        #print(Lat)
        Long.append( float((line.split(sep=","))[1] ))
        #print(Long.pop())

    LatLong.close()
    #print(Long)
    ScrapeAirNow(Lat, Long)

def ScrapeAirNow(Lat,Long):

    OutFile=open("AirNowResultsLatLong.csv", "w")

    date= '2016-09-10T00-0000'
    # Setup parameters for API call
    URLPost = {'API_KEY': 'D9AA91E7-070D-4221-867C-EFF5E0D8C2C7',
                    'latitude': 'placeholder',
                    'longitude':'placeholder',
                    #date format 2016-09-10T00-0000
                    'date': date,
                    'distance': '5',
                    'format': 'application/json'}

    endpoint = "http://www.airnowapi.org/aq/observation/latLong/historical/"

    #Example AirNow URL format
    # http://www.airnowapi.org/aq/observation/latLong/historical/?
    #format=application/json&latitude=38.3651&longitude=-114.4141&
    #date=2016-09-10T00-0000&distance=25&API_KEY=D9AA91E7-070D-4221-867C-XXXX
    # For each Lat and Long, request data from the URL and write it to a file.

    #Reverse Long so that elements can be "popped" off
    Long.reverse()

    for nextLat in Lat:
        URLPost['latitude'] = nextLat
        URLPost['longitude']=Long.pop()
        #print(URLPost)
        response=requests.get(endpoint, URLPost)

        #txt = response.text
        #print(txt)

        # Get the json response and pull out fields to print
        jsontxt = response.json()
        for list in jsontxt:
            #print("List is ", list)
            AQIType = list['ParameterName']
            AQIValue = str(list['AQI'])
            State = list['StateCode']
            City = list['ReportingArea']
            DateObs=list['DateObserved']
            print(City + "," + State + "," + DateObs + "," + AQIType + "," + AQIValue)
            OutFile.write(City + "," + State + "," + DateObs + "," + AQIType + "," +
AQIValue + "\n")

    OutFile.close()
```

```
main()
```

The above program reads latitudes and longitudes from the file called, "LatLong.txt", and uses each set of Lat and Long to make a request using the AirNow API for the AQI (PM 2.5 and Ozone) for those locations.

The "LatLong.txt" file has the following look:

```
39, -76
38.8, -77.3
38.9, -77.3
42, -87.7
37, -122
40.8, -74
64, -21.8
```

Given the above file contents for LatLong.txt , the output will create a csv file, "AirNowResultsLatLong.csv", with the following results.

```
Northern Virginia,VA,2016-09-10 ,OZONE,51
Northern Virginia,VA,2016-09-10 ,PM2.5,55
Northern Virginia,VA,2016-09-10 ,PM10,20
Northern Virginia,VA,2016-09-10 ,OZONE,51
Northern Virginia,VA,2016-09-10 ,PM2.5,55
Northern Virginia,VA,2016-09-10 ,PM10,20
Chicago,IL,2016-09-10 ,OZONE,31
Chicago,IL,2016-09-10 ,PM2.5,33
Santa Cruz,CA,2016-09-10 ,OZONE,28
Santa Cruz,CA,2016-09-10 ,PM2.5,27
Fort Lee,NJ,2016-09-10 ,OZONE,54
Fort Lee,NJ,2016-09-10 ,PM2.5,62
```

Libraries urllib and requests: A Quick Comparisons Example with JSON

Throughout this chapter, several examples for scraping and using APIs were presented. Some examples used the library, **urllib**, and others used the library, **requests**. Both have advantages. Notice that the “requests” library has a built-in json decoder (which is nice). That begin said, the json.loads method with the decoding line can be used to reach the same goal if using the urllib library. The “requests” library is also a bit easier to use as it does not require you to encode the URL.

There are many options for accessing web information. This last example will illustrate the use of “requests” and “urllib” for the same goal and resulting in the same output. As such, you can see how to use both, as well as their differences.

```
#ReqVersusUrllib.py
#Author Ami Gates

#This program shows a comparison between using the Requests Library
versus urllib

def main():

BaseURL="http://www.airnowapi.org/aq/observation/latLong/historical/"
    # Example complete URL for AirNow
    #http://www.airnowapi.org/aq/observation/latLong/historical/?
    #format=application/json&latitude=38.3651&longitude=-114.4141&
    #date=2016-09-10T00-0000&distance=25&API_KEY=D9AA91E7-XXXXXXXX

    URLPost = {'API_KEY': 'D9AA91E7-070D-4221-YOUR KEY HERE XXX',
                    'latitude': 38.9,
                    'longitude':-77.3,
                    'date': '2016-09-10T00-0000',
                    'distance': '5',
                    'format': 'application/json'}
    #Uses “requests”
    UseRequest(BaseURL, URLPost)
    #Uses “urllib”
    UseUrllib(BaseURL, URLPost)

def UseRequest(BaseURL, URLPost):
    #Uses request library
    import requests

    response=requests.get(BaseURL, URLPost)
    jsontxt = response.json()
    print(jsontxt, "\n")

    for list in jsontxt:
            AQIType = list['ParameterName']
            City=list['ReportingArea']
            AQIValue=list['AQI']
            print("For Location ", City, " the AQI for ", AQIType, "is
", AQIValue, "\n")


def UseUrllib(BaseURL, URLPost):
    #Uses urllib library
    import urllib
    from urllib.request import urlopen
```

```
    import json

    URL=BaseURL + "?"+ urllib.parse.urlencode(URLPost)
    WebURL=urlopen(URL)
    data=WebURL.read()
    encoding = WebURL.info().get_content_charset('utf-8')
    jsontxt = json.loads(data.decode(encoding))

    print(jsontxt, "\n")

    for list in jsontxt:
            AQIType = list['ParameterName']
            City=list['ReportingArea']
            AQIValue=list['AQI']
            print("For Location ", City, " the AQI for ", AQIType, "is
", AQIValue, "\n")


main()
```

The output for running either function from main, `UseRequest(BaseURL, URLPost)` or `UseUrllib(BaseURL, URLPost)`, will result in the following:

```
For Location  Northern Virginia  the AQI for  OZONE is  51

For Location  Northern Virginia  the AQI for  PM2.5 is  55

For Location  Northern Virginia  the AQI for  PM10 is  20
```

Last Word

This chapter covers some basics of making requests to websites, gathering information, and saving it in varying formats. To further organize gathered data, JSON or BeautifulSoup can be used with knowledge of how these formats are parsed and how contained elements are accessed. Further references include: “Web Scraping with Python”, by Mitchell, “Python for Data Analysis”, by McKinney, and “RESTful Web APIs”, by Rudy.

Once you have collected the data, following steps include parsing, munging, reformatting, and cleaning. This is the topic of the next chapter.