

# Simplifying Outdoor Gear Reviews with a GUI in Python

Charles Vanleuvan - Final Project

---



## Introduction

*Analysis Paralysis* is a term that I am quite familiar with. It is a crippling disability that I must overcome each time I venture to the internet in search of new outdoor gear. It is a condition that was born out of fear. When you are in the woods at night, at the mountaintop in a blizzard, or deep in a canyon during a scorching hot summer's day, you can't afford to have your gear break down on you. At the very least, it is a waste of money,

---

---

at the worst it could be life threatening. This fear has led me to view outdoor gear, such as tents, backpacks, and winter clothing through the lenses of functionality and durability when shopping. In finding the right product that is durable, effective, and hiker-approved, I can spend hours reading product reviews and Q&A responses from REI.com (REI is one of the largest retailers of outdoor gear, and has a large base of users that actively review products). The seemingly unending search through a product's reviews is the motivation for this project.

There is a quicker way to pull up word frequency and text sentiment, and that is the goal of this project. A Graphical User Interface (GUI) that can take a search term (i.e., the product I am looking to buy) and aggregate the sentiment across all the reviews for that product and generate a word cloud is invaluable to me. Ultimately, this GUI serves as a one stop shop for my product searching on REI.com. In a single interface, the word cloud and sentiment score for all reviews can be seen, thus saving countless hours poring over reviews and clicking 'load more reviews'.

The backend of this project contains the scripts to retrieve information from REI's website, via the use of BeautifulSoup and Selenium. This section also has the necessary matplotlib functions to generate the word cloud feature and uses the nltk library to tokenize the reviews and gather a sentiment score for each review.

The frontend of this project is the GUI display. This is the interface that I or any user will use to search products, select products from the search results, and see the word cloud and sentiment scores. This interface was built using the PySimpleGUI library, which is a combination of some other well known GUI designing python libraries.

The output of this program is to display the GUI for the user and show the word clouds and sentiment score plot should the user select to view those. REI.com is the retailer chosen for this study.

## **Web Scraping with BeautifulSoup**

The BeautifulSoup (BS) library provides plenty of functionality to open a web page and retrieve information from that page via parsing the HTML text that generates it. To use BS, a url link must be entered and then from that a *soup* object can be created. This soup

---

object contains the HTML information for a web page, but available for use in python. Both BS and the common urllib packages must be imported to scrape the REI web page search results.

```
#import packages for web scraping
import bs4 as bs
import urllib.request
```

To create a soup object from the REI search page, a URL must first be created containing the search term I supply it.

```
def rei_link(product):
    """Takes a product name and creates a REI search URL from it"""
    product = product.replace(' ', '+')
    rei_url = 'https://www.rei.com/search?q=' + product
    return(rei_url)
```

REI uses the same base URL for all searches, so the rei\_link() function simply takes a string and replaces the spaces with "+" and then returns the full rei\_url to be used in creating the soup object with BS.

To create the soup object, urllib.request is used to create a source, then BS is used to parse the HTML and return it in the soup variable:

```
#create the search results source object
source = urllib.request.urlopen(rei_link(values['_INPUT_'])).read()
#create soup object from the source link
soup = bs.BeautifulSoup(source, 'html.parser')
```

(Note: values['\_INPUT\_'] is the search string returned from the GUI, which is used to generate the url. More on the GUI specific syntax will be discussed further on in this report)

Most searches on a retailers web page will return multiple products, even if you enter a specific product name. To account for this, I used a function to find all the product names in the search results HTML, shown below:

---

```
def products_found(soup_object):
    """A function that returns the products found on search page. Requires a soup object"""
    for item in soup_object.find_all('a', class_ = "_1A-arB0CEJjk5iTZIRpjPs"):
        #This initial find_all() only returns 1 item. Product tags are nested in here

        #return the model names found in search results
        for div in item.find_all('div', class_ = "r9nAQ5Ik_3veCKyyZkP0b"):
            product_models.append(div.text)

        #return the URLs found in search results
        for span in item.find_all('span', class_ = "_2xZVXXKL4Bd0pJyQCumYi9P"):
            product_urls.append(rei_base_url + item.get('href'))
    return("Successfully searched.")
```

Products\_found() takes the soup object created from the search and scans the html to find all product names listed in the HTML section where they are stored. In the function's loops, the `class_` field arguments are chosen specifically to narrow down into the section of the HTML document that has the product names. The HTML document for the REI web page is very long and contains many nested tags. Finding the correct tags that contained the product names was a cumbersome process. Selenium below provides an easier approach by using the HTML objects XPath to retrieve information, where XPath is a unique field for every object generated in an HTML document. Products\_found() does not return any output, rather it appends two empty lists in the GUI and fills them with the product names and URL links. The product names will be visible in the GUI to the user so they can specifically choose which product to look at. The order in which product names and product URLs are stored in the two lists is important, as the user will select a product from the GUI dropdown box, and the index number of that product in the product list will be used to retrieve the corresponding URL from the URL list. This URL selection is done with another custom function, shown below:

```
def selected_product_url(index):
    """A function that the user picks what product url to use. Calling the function returns the url"""
    if len(product_models) == 0:
        print("No products have been found.")

    elif len(product_models) == 1:
        return(product_urls)

    else:
        #return the URL for the product
        return(product_urls[product_models.index(index)])
```



---

This function returns the URL of the product selected in the GUI drop down.

At this point, BeautifulSoup has served its purpose, and the task of delving into the specific product reviews will be commanded by Selenium.

## Collecting Product Reviews with Selenium

Now, after searching for a product with the GUI, the dropdown list has been filled with all matches found on the REI web page. Below is what the user would see at this point:



The next action is for the user to select one of the products found to collect all the user reviews. This has to be driven by Selenium, as the product reviews are dynamically generated on the REI web page, thus BeautifulSoup won't be able to retrieve the text information.

A firefox browser was used, which required an import of the GeckoDriverManager package. The *sleep* function from *time* is a critical function, as it helps trick REI into thinking this is not an automated bot. The sleep function is used between Selenium driver functions to pause the script and prevent the host server from blocking the browser's API call.

```
#import packages for web browser driving
from selenium import webdriver
from time import sleep
from webdrivermanager import GeckoDriverManager
from selenium.webdriver.firefox.options import Options
```

After a product is chosen from the drop down list, the Selenium browser object is created.

---

```
#open firefox browser in Selenium
browser = webdriver.Firefox()
print("Browser opened")
```

Once the browser is opened, the script tries to navigate to the URL of the selected product (note: the selected product is named *index* and is the argument used in the `selected_product_url()` function)

```
#navigate to product page
try:
    browser.get(selected_product_url(index))
    print("Opened product page. Waiting 10 seconds.")
    sleep(10)
except:
    print("Could not load url.")
```

I have found that Selenium browser navigation can be inconsistent at times, so a web page may not be able to load for an unknown reason. A try/except block was used for this part to not break the program if the product page could not be loaded.

I mentioned above that REI has a large user base, and they are actively reviewing gear. If the button for 'Load More Reviews' exists, the browser should click it to load as many comments as possible. This again was done with a try/except block.

```
#load more comments, if exists
try:
    browser.find_element_by_xpath('/html[1]/body[1]/div[2]/d
    print("Loading extra reviews. Waiting 10 seconds.")
    sleep(10)
except:
    print("No extra comments to load.")
```

The 'Load More Reviews' button was found via its XPath. An XPath is unique to each object in an HTML page. So if this XPath exists, navigate to it. If not, the script prints that there were no extra comments to load, and continues to the next step. If more reviews are loaded, the `sleep(10)` function is called to tell the browser to wait 10 seconds before making

---

the next navigation. This gives the browser time to load all reviews as well as prevents being caught by the host server as a bot.

Now the browser has a web page with reviews. The script then finds all elements that are bodies of text in a review or Q&A response and stores them in a variable called reviews:

```
#get reviews from web page
reviews = browser.find_elements_by_class_name('bv-content-summary-body-text')
sleep(10)
```

Again, pause for 10 seconds to let the browser pull all the reviews into memory. A custom function was made to retrieve the actual text from the reviews instead of the HTML fluff and store it in a list.

```
def print_reviews(reviews):
    #A function that takes a reviews object and prints them

    review_list = []

    for item in reviews:
        review_list.append(item.text)

    return(review_list)
```

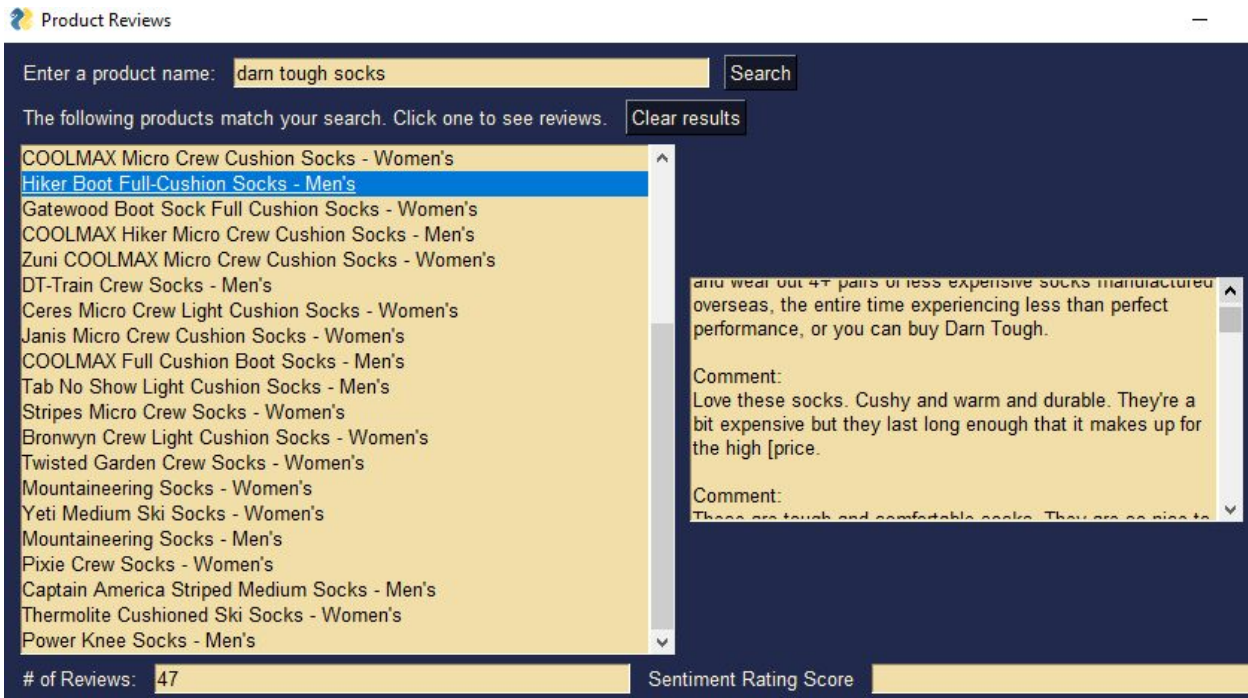
This function is then called with the reviews variable as the argument supplied to it.

```
#display review text in GUI
window.FindElement('_REVIEW_TEXT_').Update('\n\nComment:\n'.join(print_reviews(reviews)))
print("Displayed review text.")
sleep(12)

window.FindElement('_REVIEWS_').Update(len(reviews))
print("Found {} reviews.".format(len(reviews)))
print("End product search")
```

Each review or Q&A response text is then displayed in the GUI, and the metadata (how many reviews there are) is displayed as well. At this point, the text information is stored as strings in the list which is suited for use in the word cloud and sentiment analysis.

Here is what the user sees in the GUI at this point:



The next step is generating a word cloud to see the most frequently used words in all the reviews and responses.

## Generating a Word Cloud

To build the word cloud, a tokenizing function and graphing framework are needed. Word tokenizing is done with the very popular nltk library, and graphing is done with matplotlib. These are both imported along with the necessary functions inside those detailed libraries.

```
#import packages for wordcloud
from wordcloud import WordCloud, STOPWORDS
import pandas as pd

#import tokenizer packages
from nltk.tokenize import word_tokenize
from nltk.sentiment.vader import SentimentIntensityAnalyzer
from nltk.tokenize import sent_tokenize
```

The starting point for the word cloud is a list of reviews. We have already achieved that above and stored it in a variable called reviews. Then, a custom word cloud function is created to create and return the cloud plot. Word\_cloud is shown below:



```
def word_cloud(reviews):
    """A Function that generates a word cloud from a web page containing product reviews"""

    #create a text string to add all words to
    review_words = ''
    #stop words
    stopwords = set(STOPWORDS)
    #iterate through the reviews
    for item in reviews:
        #tokenize the text
        tokens = word_tokenize(item.text)
        #convert the tokens into lower case
        for i in range(len(tokens)):
            tokens[i] = tokens[i].lower()

        #append to review_words
        review_words += " ".join(tokens) + " "

    #generate word cloud
    wordcloud = WordCloud(width = 800,
                           height = 800,
                           background_color = 'white',
                           stopwords = stopwords,
                           min_font_size = 10).generate(review_words)

    #plot the word cloud object
    plt.figure(figsize = (8,8), facecolor = None)
    plt.imshow(wordcloud)
    plt.axis("off")
    plt.tight_layout(pad = 0)
    return(plt.show(block=False))
```

An empty string variable is created which this function will append with words. Then, looping over each review returned in the HTML object, the `word_tokenize()` function is used on the text strings of each review. This breaks down the review into a list of strings, where each string is a word. Then, another for loop takes each word in the tokens variable and appends it to the review\_words variable. Next, the word cloud is created and given width, height, and stopwords. The standard index of stopwords was chosen. The review\_words string is supplied, and then the plot figure is created. This function returns the plot with an added argument `block = False` to allow the plot to popup in the GUI rather than in terminal.

This word cloud function is called in the GUI when the user selects it as a button. Then, a window pops up with the graph inside. Below is an example word cloud for the Darn Tough Hiking socks:



---

done because the nltk sentiment analysis package requires sentences rather than words to create a positivity or negativity score. When the user selects the “Sentiment Rating Plot” button in the GUI, that triggers the script to loop through all reviews and append the tokenized sentences to a sentences list. A custom function is then needed to generate the average sentiment compound score. Sentiment scores from nltk’s Sentiment Intensity Analyzer are broken down into Positive, Neutral, Negative, and Compound, where compound is the compounded sentiment. 1 is purely positive and -1 is purely negative.

```
if event == 'Sentiment Rating Plot':
    #sentences list to be appended to
    sentences = []

    #loop through reviews and pull out sentences
    for item in reviews:
        sentence_tokens = sent_tokenize(item.text)

        #append to sentences list
        for sentence in sentence_tokens:
            sentences.append(sentence)

    #calculate average sentiment rating and display it
    window.FindElement('_RATING_').Update(sentiment_compound_score(sentences))
    #return sentiment plot
    sentiment_plot(sentences)

    print("Displayed sentiment plot.")
```

The custom function `sentiment_compound_score()` calculates the average compound score, shown below:

```
def sentiment_compound_score(sentence_list):
    #A function that calculates the average compound score for all sentences in a product review listing
    compound_scores = []

    for sentence in sentence_list:
        compound_scores.append(SentimentIntensityAnalyzer().polarity_scores(sentence)['compound'])

    #return the average value
    return(sum(compound_scores) / len(compound_scores))
```

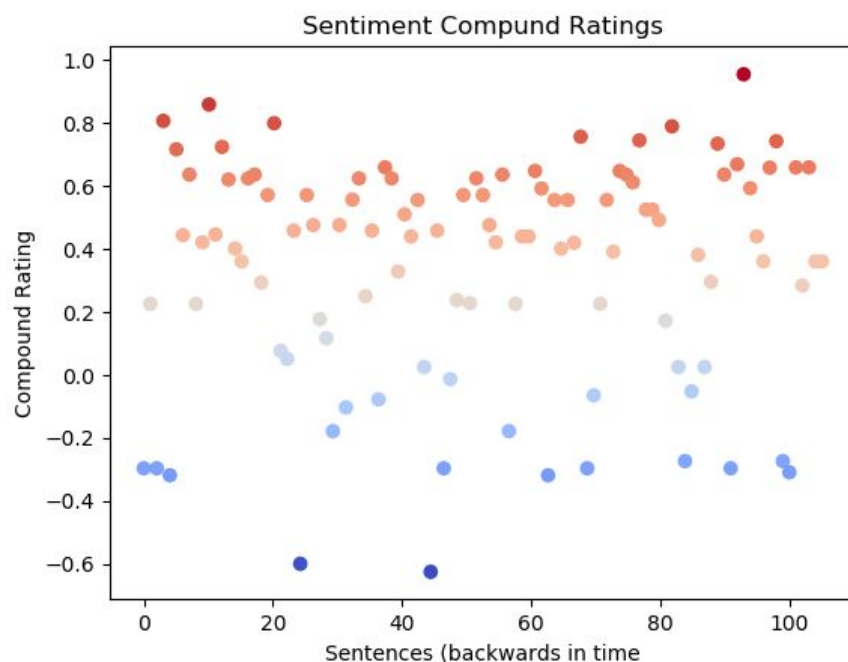
---

It takes a tokenized sentence list and applies nltk's `SentimentIntensityAnalyzer` function, retrieving the compound score (a float data type) from the result. This is summed and then divided by the count to calculate the average and return it as a float.

At the same time, a sentiment plot is created to show the compound sentiment for all sentences over time in a product's page. The `sentiment_plot` function takes a tokenized list of sentences as the input, creates a variable based on the compound scores for each sentence in the list, and creates an independent variable based on the number of sentences in the list. The Numpy function `linspace` is used to generate data for each sentence to be used as an x variable. Lastly, the function creates the scatterplot figure and returns it. When the user selects the "Sentiment" button in the GUI, this plot will be returned.

```
def sentiment_plot(sentences):  
    """A function that takes list of sentences and gets sentiment scores"""  
  
    y = [SentimentIntensityAnalyzer().polarity_scores(sentence)['compound'] for sentence in sentences if SentimentIntensityAnalyzer().polarity_scores(sentence)['compound'] != 0]  
    x = np.linspace(0, len(y), len(y))  
  
    plt.figure()  
    plt.scatter(x,y, c=y, cmap='coolwarm')  
    plt.xlabel('Sentences (backwards in time)')  
    plt.ylabel('Compound Rating')  
    plt.title('Sentiment Compound Ratings')  
    return plt.show(block=False)
```

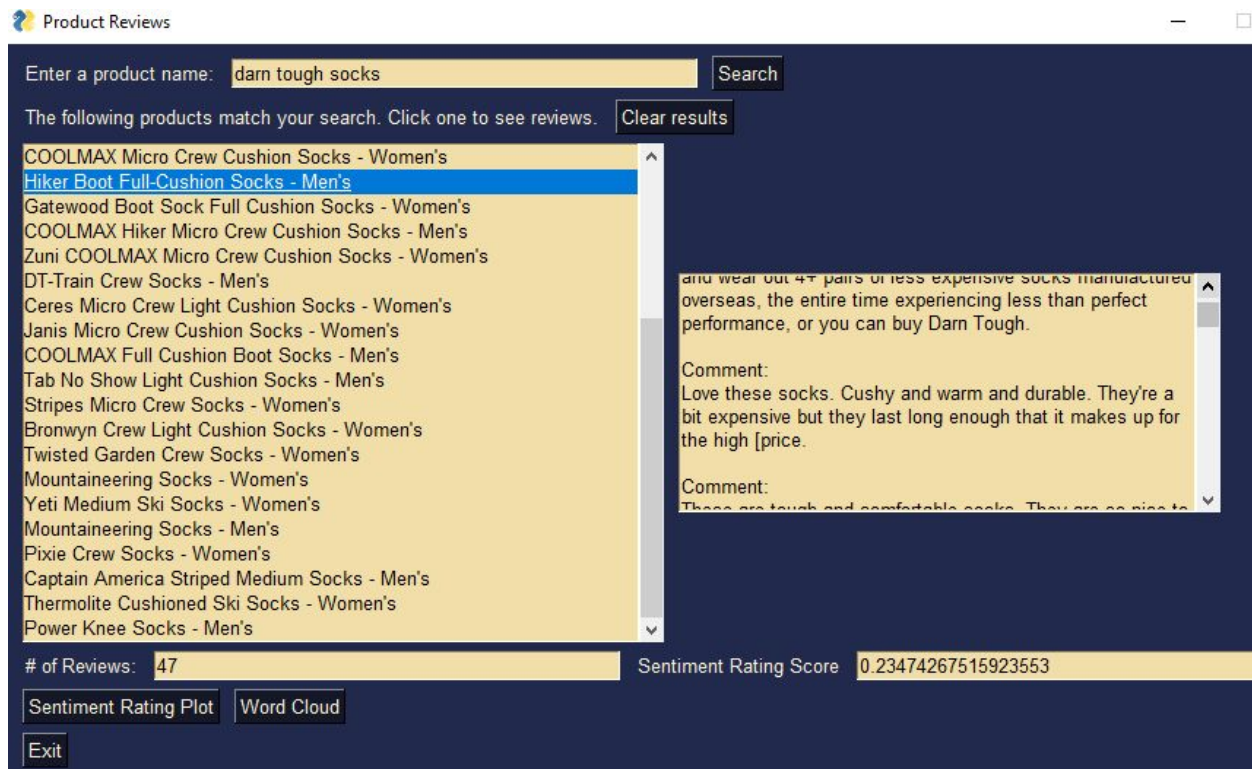
Here is an example scatterplot for the same Darn Tough hiking socks:





The y axis contains the compound score for each sentence and the x axis is the sentences going from most recent to oldest, left to right. The sentiment scores are colored by their compound score, more red indicates more positive, more blue indicates more negative. All 0, or perfectly neutral, sentiment scores were omitted from this plot, as they do not provide any use.

In the GUI, the average score is displayed. At this point, all items in the GUI are filled out, and looks like the below screenshot:



## Wrapping It All Up In a GUI

All the above program details describe the backend of this program. The user will only see the GUI and interact with buttons or text boxes in that interface. PySimpleGUI was used as the library from which to build this interface. It was chosen because of it's relatively simpler documentation and command syntax. It actually is a combination of other well known Python GUI libraries, so it combines aspects of each into its own library. Building the GUI starts with selecting a theme (blue color) and then designing the layout of the window.

---

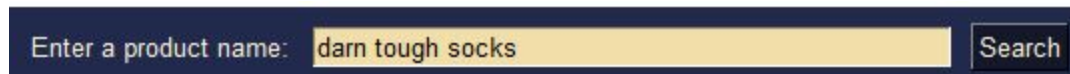
Every text box, input box, dropdown box, or button has to be explicitly called in the layout for the interface. With each feature call, arguments are passed to give text to the feature, assign a key name (so the values supplied to it can be retrieved dynamically), or change the size to allow all the text to be seen. The window is broken down into rows, and the layout is created as a list of rows. The first row is the details for what appears at the top of the interface.

```
#create GUI
sg.theme('DarkBlue17')

layout = [ [sg.Text('Enter a product name:'),sg.InputText(key = '_INPUT_'),sg.Button('Search')],
            [sg.Text('The following products match your search. Click one to see reviews.'), sg.Button('Clear results')],
            [sg.Listbox([], enable_events=True, key='_PRODUCTS_',size=(60,20)),sg.Multiline(key = '_REVIEW_TEXT_',size=(50,10))],
            [sg.Text('# of Reviews:'), sg.InputText(key = '_REVIEWS_'), sg.Text('Sentiment Rating Score'), sg.InputText(key = '_RATING_')],
            [sg.Button('Sentiment Rating Plot'), sg.Button('Word Cloud')],
            [sg.Button('Exit')]]

window = sg.Window('Product Reviews', layout)
```

So, the feature `[sg.Text('Enter a product name:'), sg.InputText(key = '_INPUT_'), sg.Button('Search')]` in the top row creates the top row seen in the GUI, as below:



A key is assigned to the InputText feature so that the product name the user enters there can be used for backend processing after hitting the 'Search' button. Buttons, InputText, and Text features are easy to understand, but Listboxes and Multiline features are a bit more complex to integrate into an interface, especially since they are generated dynamically as the user is using the interface. The list box and multi line feature, since they are only to be populated after the user searches a product and selects one, begin as empty areas. To display the window on the user screen, a while loop is used. While True, the event and values of the event are recorded. An event is a button click, and the values are the item selected or typed. The window remains open, actively monitoring all events until an event condition is met. If the event is closing the window (`sg.WIN_CLOSED`) or clicking the exit button (`event == 'Exit'`), then the program is ended and the GUI closed. All information stored in memory, such as sentence lists or word clouds, are lost.

---

```
while True:
    event, values = window.read()

    #check if window is closed
    if event == sg.WIN_CLOSED or event == 'Exit': ...
    if event == 'Clear results':|...

    if event == 'Search': ...

    if event == '_PRODUCTS_': ...

    if event == 'Word Cloud': ...

    if event == 'Sentiment Rating Plot': ...

    try:
        browser.quit()
    except:
        pass
    window.Close()
```

```
#check if window is closed
if event == sg.WIN_CLOSED or event == 'Exit':
    break
```

If the user selects the 'Clear Results' button, then the list box containing all the product names will be cleared with the below *Update* function, where '\_PRODUCTS\_' is the key referencing the list box.

```
if event == 'Clear results':
    window.FindElement('_PRODUCTS_').Update('')
```

All events in the event loop follow the same structure of checking if the event matches a condition, then performing some action with the information supplied to that feature in the interface. When a user types in a product name and clicks the 'Search' button, that begins the list of actions of scraping the REI website, pulling all matching product names, and retrieving the product URLs.

---

```
if event == 'Search':
    #create the search results source object
    source = urllib.request.urlopen(rei_link(values['_INPUT_'])).read()
    #create soup object from the source link
    soup = bs.BeautifulSoup(source, 'html.parser')

    #clear product_models and product_urls
    product_models = []
    product_urls = []

    #scrape the website search results
    products_found(soup)
    print("Found products")
    for product in product_models:
        print(product)

    #for the products returned into product_models, add each to the Listbox
    for product in product_models:
        window.FindElement('_PRODUCTS_').Update(values = product_models)
```

## Questions

### Does a well known brand get more reviews?

Deciding between a well known brand or a relatively unknown brand is a common decision when searching for a product. Knowing how many reviews are available for the products you search can help make your decision on which product to look at. There is a very well known sock company called Darn Tough that makes hiking socks. My expectation is that those products have many more reviews per product than a relatively unknown sock company that I have never heard of. I wanted to see what the average review count is for a Darn Tough sock compared to a smaller company called Farm To Feet socks.

To do so, Selenium web driver was used to search both brands' socks, and then the review count was pulled via the HTML class name. The counts of reviews for each product returned in the search was stored in a list as an integer (the leading and trailing parentheses symbols need to be trimmed out) and then an average was calculated.



---

```
#Return the average number of reviews for a product. The product was searched with selenium
#empty list to append all review totals to
product_reviews = []

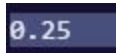
browser = webdriver.Firefox()
browser.get(rei_link('farm to feet hiking socks'))
sleep(3)

#retrieve the review counts for the product and return to the list

review_count = browser.find_elements_by_class_name('bv-rating-label')
sleep(3)
for i in review_count:
    product_reviews.append(int(i.text[1:-1]))

print(sum(product_reviews) / len(product_reviews))
```

Farm to Feet brand socks returns almost no reviews per product:



0.25

On the other hand, Darn Tough has over 38 reviews per product:



38.4333

There is vastly more language to work with when scraping reviews from Darn Tough socks compared to the smaller brand. From this, the takeaway is that the depth of reviews for a product should be considered prior to looking into the reviews in detail.

### **What words are in positive and negative sentiment reviews?**

The words in a sentence must vary greatly between sentences that have a very positive sentiment score and sentences that have a negative sentiment score. The difference in diction can be observed when looking at the reviews for Darn Tough socks again. Darn Tough socks are a good choice for this test because there are many reviews on all of their products, as seen above.

*Take some positive sentiment comments, what are the words?*

---

When pulling the most positively rated sentences out of all the reviews for the socks, it is observed that they indeed are positive words used without any sarcasm in the review.

“Perfect”, “Love”, and “Best” are all positive words that increase the score for this sentence

```
They hug my feet perfectly.  
pos: 0.709,  
  
Love these socks.  
pos: 0.677,  
  
Best socks out there!!!  
pos: 0.629,
```

---

The code to execute this is below:

```
#Open Firefox browser
browser = webdriver.Firefox()

#Navigate to product page
browser.get(selected_product_url())
sleep(5)

#get the text of the reviews and questions
reviews = browser.find_elements_by_class_name('bv-content-summary-body-text')
sleep(3)
counter = 1
for item in reviews:
    print("{} reviews found".format(counter))

#Sentiment Analyzer expects a list of sentences
#create a list of sentence strings from the reviews
sentences = []
for item in reviews:

#print the sentence and the sentiment scores
#change the sentiment score and k == ' ' filter to change between pos and neg
for sentence in sentences:
    #print(sentence)
    ss = SentimentIntensityAnalyzer().polarity_scores(sentence)
    for k in sorted(ss):
        if k == 'pos' and ss[k] > 0.6:
            print(sentence)
            print('{}: {}'.format(k, ss[k]), end='')
            print("\n")
```

Switching between positive and negative sentiment filters can be done in the if statement in the bottom for loop. A filter of 0.6 was set to only retrieve the positive sentiment sentences that scored higher than 0.6

*Take some negative sentiment comments, what are the words?*

However, it is not as clear to determine the sentiment with negative sentences. Pulling the most negatively scored sentences returns text that does not appear to be negatively toned.

---

It is clear though how the NLP package determined these sentences to be negative as they do contain words such as "no","problems","not".

```
They show virtually no wear.  
neg: 0.355,  
  
Longest hike I was ever on and I was the only one with no problems  
neg: 0.308,  
  
Tumble dry on low.  
neg: 0.412,  
  
Do not dry clean.  
neg: 0.429,
```

I do not take any of the above sentences to be a negative sentiment, and actually would suggest that the second sentence is in fact a positive review (especially if product reliability is important). Not having any problems on a long hike suggests that the product is durable to withstand sustained use, and being the only one to *not* have problems suggests that this product is a *better* choice than other products that people have bought.

The takeaway from this is that the base SentimentIntensityAnalyzer does have some flaws in generating negative sentiment scores from the English language. Attributing the above comments a very negative score does bring down the overall score for this product, which can influence a decision on whether to buy it or not.

*Take some neutral sentiment comments, what are the words?*

Following the same inspection, but with neutral sentences, a very inconsistent grading scheme is detected.

```
Think I'll be buying more from this brand if these last.  
neu: 1.0,
```

```
The only socks I wear any more.  
neu: 1.0,
```



---

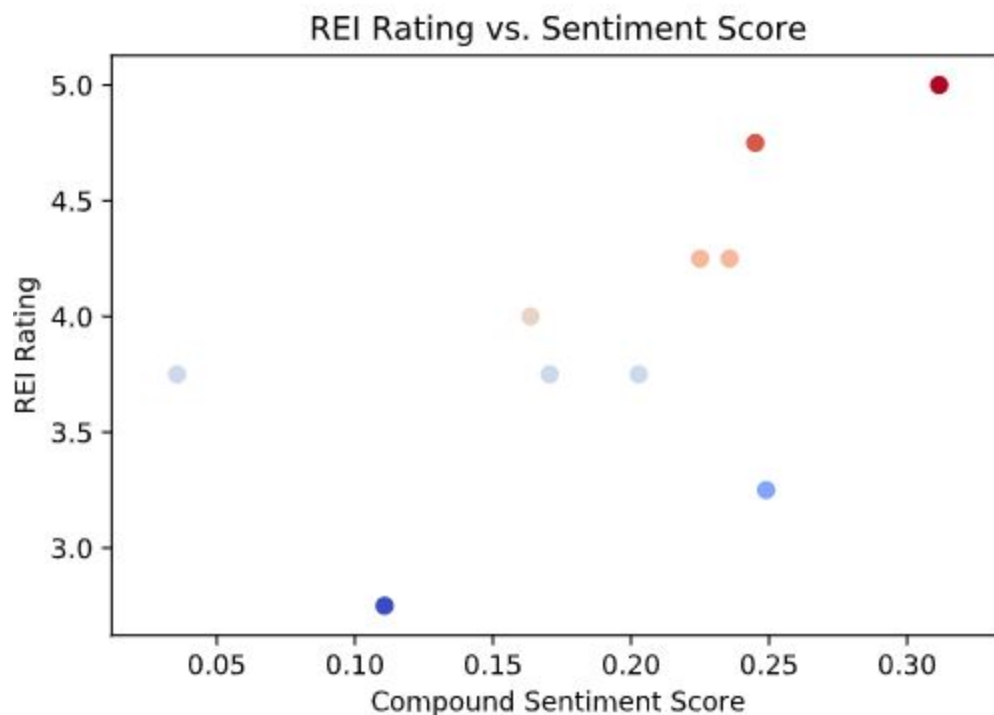
```
You can buy, use and wear out 4+ pairs of less expensive socks manufactured overseas, the entire time experienci  
ng less than perfect performance, or you can buy Darn Tough.  
neu: 0.846,
```

The SentimentAnalyzer is calculating these sentences as very neutral, however to a human they clearly are recommending the product.

Again, the takeaway here is that the SentimentAnalyzer is quite adept at recognizing positive text, but is still inconsistent and lacking when it comes to evaluating negative and neutral text.

### Is there a trend between the REI rating and the Compound Sentiment Score?

For a product that is sold on REI, there can be many reviews. From those reviews, a compound sentiment score can be calculated. Along with that, there is a REI rating associated with every product, which is based out of 5 stars. By running this program for multiple different product types, and picking different specific products, a scatter plot can be generated to show how the REI rating varies as the Sentiment Compound score changes. The below plot is the REI rating for 10 different products, chosen at random, and the associated Sentiment Compound score taken from the reviews on that product.



*The takeaway from this plot*

---

There is an idea from studies of human behavior that voices of disgust and opposition tend to be louder than voices of agreement. In other words, someone who is very unhappy with a product they bought is thought to be more likely to comment their disgust as opposed to someone who is happy and more likely to not say anything. However, the takeaway from this plot between Rating and Sentiment shows that, at least on REI's web page, people that are happy with their products do come out and say so in the review section. If the idea above were absolutely true, then there would not be a direct relation between sentiment score and rating.

## Next Steps

To make a full fledged application or web site, there is a lot of work that can be put into this project. Some improvements include:

1. Add the product rating to the GUI display
2. Add the REI Ratings vs. Compound Sentiment Score scatter plot to the GUI display
3. Dynamic stop words
  - a. Depending on the product, the stop words used in tokenization or word clouds should change. When searching tents, there really is no benefit to seeing 'tent' as the most common used word. It is expected to be there frequently. However, the standard stopwords dictionary from NLTK does not include 'tent' as a stop word. It would have to be added manually with each search, thus dynamically updating the list.
4. Combo-Sentiment
  - a. Some words as a standalone are negative in emotion, such as 'not' and 'bad'. However, the English language is tricky and the sentence "It is really not bad" actually means that the product is quite good. Deciphering the sentiment when negative words are used in a positive sentence is complex and could use some further tuning to improve this project.

The biggest improvement that can be made is to simultaneously pull the product reviews for a product from all possible retailers on the internet and then display all of the

---

sentiment scores and word cloud in the GUI. This requires understanding the HTML makeup of many sites and creating functions to handle the navigation of the HTML on those sites.

## Sources

1. Product information was pulled from REI.com
2. This code is entirely for educational purposes and is in no way to be used for commercial use on REI.com as it is against their terms of service.
3. <https://pysimplegui.readthedocs.io/en/latest/>
4. <https://www.nltk.org/>