Welcome to Tutorial 3!

You're all doing great so far! After this tutorial, you will be able to write your own motif searching algorithm, and use the results for your project.
In this tutorial, we will be learning about file input and output (file I/O) and regular expressions (regex). You're in the home stretch!

Part 0: File Input

Using Python to read and write to files is very helpful when working with larger datasets. The kinds of files we'll be opening and closing are often text files, like you'd write in Notepad or any other text editor.

In the future, you will most likely encounter these two types of text files: CSVs and TSVs. **CSV** stands for "comma-separated variables," and **TSV** stands for "tab-separated variables." As you might guess, these two file types use commas and tabs, respectively, to separate the contents of their files. These are often representations of **data tables** and **spreadsheets.** Putting data inside a CSV or TSV file can be useful for several reasons:

- You can access the data and use it even after you close Python.
- You can "clean up" valuable memory that Python needs in order to work efficiently.
- You can back up data that you worked so hard to clean and organize.
- You can send it to other researchers and collaborators to use.

Let's practice opening and closing some files. We've uploaded a CSV to Canvas called "climatetweets.csv". This is from a project called "Climate Related Tweets Before, During, and After Hurricane Sandy 10-5 through 11-15 2012" conducted by the University of Central Florida. You can view the data source here (https://www.openicpsr.org/openicpsr/project/100202/version/V3/view).

There are many ways to open a data file and read in its contents. Fortunately, there is a convenient library we can use called **csv**. Unsurprisingly, it contains functions and tools we can use to work with CSV files. In the following cell we will import the **csv** library.

In [ ]:
```
import csv
```

Great! We now have the functions to start working with CSVs. But, we need to read in each row of the data file in order to start working with it. The following code will create an empty list called `data`, which we will use to hold our data. Then, it will read in each row of the CSV file one at a time and add (or, `append`) it to `data`.

In [ ]:
```
data = []
with open("climatetweets.csv", "r", encoding="latin-1") as csvDataFile:
    csvReader = csv.reader(csvDataFile)
    for row in csvReader:
        data.append(row)
```

Notice the line with the words `with open( ) as`. This may look strange to you, but it is convention for **opening** and **closing** data files in Python. It's an example of what we call **"syntactic sugar"**: shorthand code that makes it easier for humans to understand, but doesn't change what the computer actually does. In this case, the word `with` will make sure that our file properly opens and closes, no matter what the code chunk that follows does or fails to do. A very good thing for file security!

Notice also that the function `open` takes in an argument that says `"r"`. Can you guess what this `"r"` stands for? Write your answer below. (Hint: it may be useful to use the `help` function to look at the documentation for `open`.)

Your Answer Here

Let's look at the first ~6 entries in our "**data**" list:

In [ ]:
```
print(data[0:6])
```

Hmm, that was a jumble of stuff. It seems like the first list (or "row" in our data table) is a series of dates, while the second list is a bunch of tweets itself. Let's focus in on some of the tweets in our dataset:

In [ ]:
```
print(data[1])
```

This is all a mess!

When you're dealing with raw data, more times than not the data is very messy. We can do some tidying up in order to get more sense out of it. When we start to clean up text, **regular expressions** will be useful.

Part 1: Regex

Regex, or regular expression, is a very powerful concept. It is essentially posing a series of search queries to match the information you are trying to find. A "regex" is a series of characters and symbols that represent a search pattern you are trying to find in a much longer sequence. Here's a regex "cheat sheet":

regularexpressions

Anchors

^	Start of line +
\A	Start of string +
$	End of line +
\Z	End of string +
\b	Word boundary +
\B	Not word boundary +
\<	Start of word
\>	End of word

Character Classes

\c	Control character
\s	White space
\S	Not white space
\d	Digit
\D	Not digit
\w	Word
\W	Not word
\xhh	Hexadecimal character hh
\Oxxx	Octal character xxx

POSIX Character Classes

[:upper:]	Upper case letters
[:lower:]	Lower case letters
[:alpha:]	All letters
[:alnum:]	Digits and letters
[:digit:]	Digits
[:xdigit:]	Hexadecimal digits
[:punct:]	Punctuation
[:blank:]	Space and tab
[:space:]	Blank characters
[:cntrl:]	Control characters
[:graph:]	Printed characters
[:print:]	Printed characters and spaces
[:word:]	Digits, letters and underscore

Assertions

?=	Lookahead assertion +
?!	Negative lookahead +
?<=	Lookbehind assertion +
?!= or ?<!	Negative lookbehind +
?>	Once-only Subexpression
?()	Condition [if then]
?()|	Condition [if then else]
?#	Comment

Note Items marked + should work in most regular expression implementations.

Sample Patterns

([A-Za-z0-9-]+)	Letters, numbers and hyphens
(\d{1,2}\/\d{1,2}\/\d{4})	Date (e.g. 21/3/2006)
([^\s]+(?=\.(jpg|gif|png))\.\2)	jpg, gif or png image
(^[1-9]{1}$\|^[1-4]{1}[0-9]{1}$|^50$)	Any number from 1 to 50 inclusive
(#?([A-Fa-f0-9]){3}(([A-Fa-f0-9]){3})?)	Valid hexadecimal colour code
((?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,15})	8 to 15 character string with at least one upper case letter, one lower case letter, and one digit (useful for passwords).
(\w+@[a-zA-Z_]+?\.[a-zA-Z]{2,6})	Email addresses
(\<(/?[^\>]+)\>)	HTML Tags

Note *These patterns are intended for reference purposes and have not been extensively tested. Please use with caution and test thoroughly before use.*

Quantifiers

*	0 or more +
*?	0 or more, ungreedy +
+	1 or more +
+?	1 or more, ungreedy +
?	0 or 1 +
??	0 or 1, ungreedy +
{3}	Exactly 3 +
{3,}	3 or more +
{3,5}	3, 4 or 5 +
{3,5}?	3, 4 or 5, ungreedy +

Special Characters

\	Escape Character +
\n	New line +
\r	Carriage return +
\t	Tab +
\v	Vertical tab +
\f	Form feed +
\a	Alarm
[\b]	Backspace
\e	Escape
\N{name}	Named Character

String Replacement (Backreferences)

$n	nth non-passive group
$2	"xyz" in /^(abc(xyz))$/
$1	"xyz" in /^(?:abc)(xyz)$/
$`	Before matched string
$'	After matched string
$+	Last matched string
$&	Entire matched string
$_	Entire input string
$$	Literal "$"

Ranges

.	Any character except new line (\n) +
(a|b)	a or b +
(...)	Group +
(?:...)	Passive Group +
[abc]	Range (a or b or c) +
[^abc]	Not a or b or c +
[a-q]	Letter between a and q +
[A-Q]	Upper case letter + between A and Q +
[0-7]	Digit between 0 and 7 +
\n	nth group/subpattern +

Note *Ranges are inclusive.*

Pattern Modifiers

g	Global match
i	Case-insensitive
m	Multiple lines
s	Treat string as single line
x	Allow comments and white space in pattern
e	Evaluate replacement
U	Ungreedy pattern

Metacharacters (must be escaped)

^	[	.
$	{	*
(	\	+
)	|	?
<	>	

Available free from AddedBytes.com

Unsurprisingly, there is a very useful Python library with functions for working with regular expressions. It is called `re` .

```
In [ ]: # Run this code to import the library!
        import re
```

In the next few cells, we are going to play around with regular expressions and clean up some tweets. If you're interested in learning more about regular expressions, this website (https://www.regular-expressions.info/reference.html) is a good place to start.

Let's first look at some example tweets, just to get an idea of the sorts of cleaning we are going to do.

```
In [ ]: ex_tweet_1 = data[1][2]
        ex_tweet_2 = data[1][6]
        ex_tweet_3 = data[1][8]

        for tweet in [ex_tweet_1, ex_tweet_2, ex_tweet_3]:
            print(tweet, '\n') #Separate each tweet by a newline ('\n') to make it mor
        e readable
```

Very interesting. It seems like each tweet has an ellipsis ("...") at the front and beginning of each tweet. It also seems like there are some (broken) links embedded within the tweets.

If we want to clean these tweets to make them more *readable* (not necessarily more *informative*), here are some possible to-do items for us:

- Strip off the leading and trailing ellipses
- Remove any retweet flags
- Remove any links

Let's dive right into using regex. We'll use our example tweets to test everything out.

```
In [ ]: print(ex_tweet_1, '\n')

        # Use the re.sub() function to substitute characters
        ex_tweet_1_clean = re.sub('\...', '', ex_tweet_1)
        print(ex_tweet_1_clean)
```

Notice that we used the `re.sub` function to find characters and replace them. According to the documentation for this function (https://docs.python.org/2/library/re.html#module-contents), `re.sub` needs at least 3 arguments.

- The first argument is the **regex pattern** we are searching for. In this example, we needed to search for three dots "..." together. However, a *single dot* is a special character! In order to get the *literal* string of three dots, we needed to use a backslash to **escape** the special-character meaning of a single dot. (Try removing the backslash (" \ ") and see what happens!)
- The second argument is the text that will be **substituted** at every location the **regex pattern** is found. In this case, we wanted to replace the three dots with "nothing", so I used empty quotes. (Try putting something inside of the quotes and see what happens!)
- The third argument is the **text** that we are searching through. Since I just wanted to try out this regex on our first example tweet, I let this argument be `ex_tweet_1`. (Try it on `ex_tweet_2`, `ex_tweet_3`, or another string that you come up with and see what happens!)

Let's try the exact same code on the second example tweet.

In [ ]:
```
print(ex_tweet_2, '\n')

# Remove the ellipses as we did with the first tweet
ex_tweet_2_clean = re.sub('\...', '', ex_tweet_2)
print(ex_tweet_2_clean)
```

Oops! What happened?
It looks like we accidentally removed too many dots from the end of this tweet. The word "enough" was cut off, and the tweet used dots to signify that it was a shortened word. We'll need a more specific regex that only takes off *three* dots at a time. Luckily, there is a regex expression that signifies the quantity "exactly three of." You just put the number `3` inside curly brackets (`{ }`).

In [ ]:
```
print(ex_tweet_2, '\n')

# It's good practice to work with a copy of the original text, in case somethi
ng goes wrong
ex_tweet_2_clean = ex_tweet_2
# Remove exactly three ellipses from beginning of string
ex_tweet_2_clean = re.sub('^\.{3}', '', ex_tweet_2_clean)
# Remove exactly three ellipses from end of string
ex_tweet_2_clean = re.sub('\.{3}$', '', ex_tweet_2_clean)
print(ex_tweet_2_clean)
```

Looks like our fix worked! We also used the regex characters `^` (indicating the pattern must be at the beginning of the string) and `$` (indicating the pattern must be at the end of the string).
Now that we've fixed this issue, let's go on to the second thing to do: removing retweet flags. It seems that retweet flags have a pattern to them. They always begin with the capital letters "RT", followed by an `@` symbol indicating the user the tweet is retweeted from. We can design a regex to match this pattern:

In [ ]:
```
ex_tweet_2_clean = re.sub('RT\s@[a-zA-Z:]+', '', ex_tweet_2_clean)
print(ex_tweet_2_clean)
```

Great! Let's unpack the regex we used. The regular expression will search for a pattern that has the following things, in *this order*:

- `RT` tells the pattern that we want the *exact* characters 'RT' at the *beginning* of the search pattern.
- `\s` means a single space.
- `@` will tell the pattern to look for '@'. '@' is not a special character in regex, so we do not need to escape it with a backslash.
- `[a-zA-Z:]` will search for any lowercase letter, uppercase letters, or semicolon (":"). The *square brackets* will tell Python that the character in this position can be any of these characters. As you might expect, `a-z` is shorthand for any lowercase letter, and `A-Z` is shorthand for any uppercase letter.
- `+` tells the pattern to search for the previous character "multiple times." So, it will search for multiple instances of lowercase letters, uppercase letters, and semicolons. (The subtlety here is that we did not specify a space " " in the previous search character. So, the regex will stop trying to match the pattern once it sees a space in the original text.)

We can go on to apply these `re.sub` patterns to each of the tweets in our dataset. But, remember that we don't necessarily want to copy and paste code over and over again in order to clean each of the tweets! It's best to put all of our data-cleaning work inside of our own data-cleaning *function*. Then, we can use a *loop* to apply our function to every tweet in our dataset.

In [ ]:
```
# Run this code! Each line of this function should look familiar.
def cleanTweets(tweet):
    cleaned_tweet = tweet
    cleaned_tweet = re.sub('^\.{3}', '', cleaned_tweet)
    cleaned_tweet = re.sub('\.{3}$', '', cleaned_tweet)
    cleaned_tweet = re.sub('RT\s@[a-zA-Z:]+', '', cleaned_tweet)
    return(cleaned_tweet)

# Check to see if our function works like we want it to
print(ex_tweet_3, '\n')
print(cleanTweets(ex_tweet_3))
```

Not bad! Let's apply it to all of the tweets in the first "row" of tweets.

In [ ]:
```
for i in range(len(data[1])):
    data[1][i] = cleanTweets(data[1][i])
print(data[1])
```

This looks a little bit more readable than what it was before!
Keep in mind that this is just barely scratching the surface of what is possible with regex.

Your Turn!

The last thing that we might want to do is remove links. From our experience, it seems that each link will start with an "http". Your task is to modify the function `cleanTweets` to remove links. To do this, you will need to:

- Add some code to the `cleanTweets` in order to remove links, by:
- Designing a simple regex to search for the characters "http", followed by any sequence of multiple letters, characters, and numbers. The cheat sheet included above may come in handy.
- Apply your modified function to any row of tweet data to see how it works.

In [ ]:
```
def cleanTweets(tweet):
    cleaned_tweet = tweet
    cleaned_tweet = re.sub('^\.{3}', '', cleaned_tweet)
    cleaned_tweet = re.sub('\.{3}$', '', cleaned_tweet)
    cleaned_tweet = re.sub('RT\s@[a-zA-Z:]+', '', cleaned_tweet)
    # Your Regex After This Line

    return(cleaned_tweet)

# Your Code To Apply Function After This Line
```

Part 2: File Output

Since we've been working with a list of tweets, let's write out our cleaned tweets to another CSV file. We will do this with functions from--you guessed it--the **csv** library. The following code will:

- `open` a new, blank CSV file to write to. We called it "cleaned_climatetweets.csv"
- create a function called `"spamwriter"` that works to `write` data to the CSV file
- clean each row of data using the cleanTweets function
- write each row of cleaned data to the "cleaned_climatetweets.csv" file, using our preivously defined `"spamwriter"` function
- `close` the CSV file that we were wrote our data to

In [ ]:
```
with open("cleaned_climatetweets.csv", "w") as csvfile:
    spamwriter = csv.writer(csvfile, delimiter=',', quotechar='"')
    for row in data:
        for i in range(len(row)):
            row[i] = cleanTweets(row[i])
        spamwriter.writerow([row])
    csvfile.close()
```

Notice that we used the words `with open( ) as` again to write into a new CSV file. This time, we also used a parameter `"w"`. What does the `"w"` mean here?

Your Answer Here

Your Turn!

In the folder on your computer where this Jupyter Notebook is, you will now have a .csv file with all of our stored data. Your task is to:

1. Try opening it.
2. Read in your data, row by row, and add (or, `append`) each row to the variable `cleaned_data` .
3. Show us the third row of your data. (Remember that counting in Python starts at 0!)

In [ ]:
```
cleaned_data = []
# Your Code Here
```