

Module - 02

Essentials of Natural Language Processing (NLP)

Essentials of NLP

Table of Contents

1. Basic Text Analysis	4
7. If you want to print a word at even position:	9
2. Tokenization.....	13
Split a Sentence vs Tokenize it.....	13
3. POS Tagging	16
4. Stop Word Removal	19
5. Text Normalization.....	22
6. Spelling Correction	24

7. Stemming.....	27
8. Lemmatization.....	31
09. Named Entity Recognition (NER)	34
10. Word Sense Disambiguation	38
11. Sentence Boundary Detection	41



Module Description

In this module, we will learn the essentials of NLP.

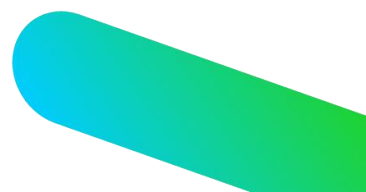
This module includes the following topics:

- Basic text analytics
- Tokenization
- POS tagging
- Stop word removal
- Text normalization
- Spelling correction
- Stemming and lemmatization
- Named entity recognition (NER)
- Word sense disambiguation
- Sentence boundary detection

Learning Objectives

By the end of this module, you will be able to:

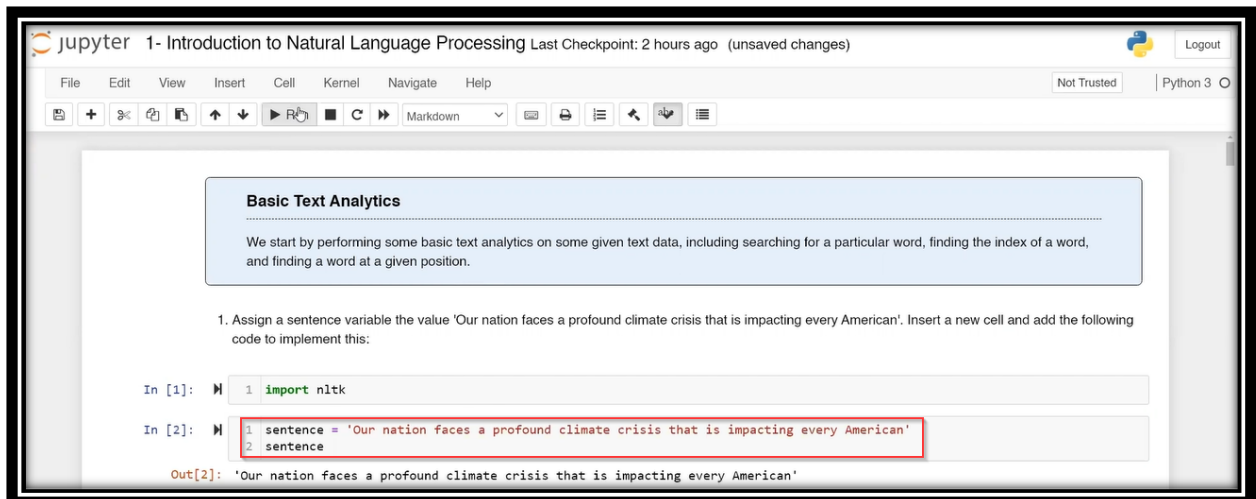
- Perform basic analytics on the given text data.
- Explain the tokenization of a simple sentence.
- Describe PoS tagging.
- Discuss stop word removal function, text normalization function, spelling correction function, stemming function, stemming, lemmatization, named entity recognition, word sense disambiguation, and sentence boundary detection.



1. Basic Text Analysis

This demo covers basic text analytics. First, import the Natural Language Toolkit (NLTK) and run it.

1. Create a variable called **sentence**. Now, assign the value ("Our nation faces a profound climate crisis that is impacting every American") to the sentence variable.

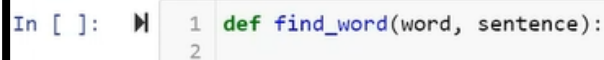


```
jupyter 1- Introduction to Natural Language Processing Last Checkpoint: 2 hours ago (unsaved changes)
File Edit View Insert Cell Kernel Navigate Help
In [1]: 1 import nltk
In [2]: 1 sentence = 'Our nation faces a profound climate crisis that is impacting every American'
        2 sentence
Out[2]: 'Our nation faces a profound climate crisis that is impacting every American'
```

The object is passed through the string, i.e., a group of words which simply is a sentence. The sentence is then displayed or printed.

2. Check whether the word 'nation' belongs to that text.

Create a function with 'def'. **Def** is an abbreviation, a common Python keyword that stands for define, which is a user-defined function. Create a function called find a word. This function will take two parameters, such as a word and a sentence. The "word" is the word that will try to look for or try to find, and the sentence is actually the variable that is already created.



```
In [ ]: 1 def find_word(word, sentence):
        2
```

Next, return the word in sentence. It returns a word/parameter and a sentence.

Now, let's find the word 'nation' using find-word function. Type the word you want to find within the parenthesis and put a comma after it. Enter the second parameter, that is, sentence and run it. The output displayed is True, which means we have the word 'nation' in the sentence.

2. Check whether the word 'nation' belongs to that text using the following code:

```
In [3]: 1 def find_word(word, sentence):  
        2     return word in sentence  
  
In [4]: 1 find_word('nation', sentence)  
  
Out[4]: True
```

The `find_word()` function takes two parameters or two inputs. This function takes the first input and it navigates through the second input, looking for a specific word.

Next, let's look for the word 'Americans', which is not there in the sentence. The sentence has the word American (singular), but not 'Americans' (plural). If you run the code, you will get 'False' as output since the `find_word()` function matches word by word and character by character.

3. Find out the index value of the word. For example, 'crisis'.

The **'def'** keyword is used to define the function. Use the function `get_index` with two parameters (word and text) to get the position of an item that is found in the string. Next, return the index of the first parameter (word). Let's put this function into text using `get_index('crisis', sentence)`. Run the code.

Here is the screenshot showing the code and its output. As you can see, the index of the word 'crisis' in the sentence is 36.

3. Find out the index value of the word 'crisis' using the following code:

```
In [6]: 1 def get_index(word, text):  
        2     return text.index(word)  
  
In [7]: 1 get_index('crisis', sentence)  
  
Out[7]: 36
```

4. To find out the rank of the word 'American', use the following code.

```
get_index('American', sentence.split())
```

The index is simply a location and memory in Python. If you want to get the index for the word 'American', you need to split the sentence into words (not into characters). If the sentence is not split into words, Python is just going to read the characters.

4. To find out the rank of the word 'American', use the following code:

```
In [8]: 1 get_index('American', sentence.split())  
Out[8]: 11
```

Below is the screenshot displaying the rank/index of the word 'American' as 11.

5. Let's print the third word of the given text using **get_word()** function. This function takes two parameters (text and rank or X and Y).

Use **return** statement to **return** a value from the get_word() function. Split the text first and return the rank. Rank is the number that you want to return. Here, you will return the text and its number.

Let's test the function. Enter the parameters for get_word() function as sentence and number 3 and run. Now run the code. Here is the screenshot displaying rank #3.

Change the number in get_word() function and run the code. You see a different index.

5. To print the third word of the given text, use the following code:

```
In [9]: 1 def get_word(text,rank):  
        2     return text.split()[rank]  
  
In [10]: 1 get_word(sentence, 3)  
Out[10]: 'a'
```

To flip/reverse the word you can use minus or negative numbers as shown in the screenshot below. For example, the reverse of the number index 2, i.e., 'faces' is 'secaf'.

```
In [12]: 1 get_word(sentence,2)[::-1]
Out[12]: 'secaf'
```

6. Let's now create a function to concatenate the first and the last words of a given sentence.

First, define the function using the keyword *'def'*. Next, name the function. You could give whatever name you want, and this function will take one parameter. Now declare some parameters.

```
words = text.split()
```

Let's print the word just for troubleshooting purposes.

The first word is always going to be a word at the index zero. Print this again for troubleshooting purposes.

The last word is going to be the words using the length function (counting the length of the entire sentence by using `len(words) - 1`) starting from the right-hand side. If you were to count the index from the left to right, it would start from 0, 1, 2, 3. If you were to count the indexes from right to left, you would start from -1, -2, -3 all the way to the end.

Let's print the last word. Return `first_word`, plus, concatenate with `last_word`. That's it.

Here you:

- Defined a few parameters.
- Split the entire sentence into tokens or words to navigate through them.
- Defined the `first_word` as the zero index, or the word with zero index.
- Defined the `last_word` as the last word, starting from the right to left, -1.

Now, let's concatenate the first word plus the second word. Add the first to the last.

Now, execute the code.



```
In [17]: 1 sentence
Out[17]: 'Our nation faces a profound climate crisis that is impacting every American'

In [15]: 1 def concat_words(text):
2         """
3         This method will concat first and last words of given text
4         """
5         words = text.split()
6         print(words)
7
8         first_word = words[0]
9         print(first_word)
10
11        last_word = words[len(words)-1]
12        print(last_word)
13
14        return first_word + last_word
```

To concatenate, use the following code:

`concat_words()`

Let's pass text. Copy and paste the 'sentence' in the parenthesis. Note: It's always good to look for the error. Sometimes you might need to Google it, sometimes you might need to look into the codes, , mismatch, or typos.

Let's execute the 'concat_words' code. It takes the text parameter as input and performs all the logic.

```
In [16]: 1 concat_words(sentence)
['Our', 'nation', 'faces', 'a', 'profound', 'climate', 'crisis', 'that', 'is', 'impacting', 'every', 'American']
Our
American
Out[16]: 'OurAmerican'
```

Let's look at the sentence again. It's one sentence and when you split it tries to create some tokenization. So, why you need to do that is to facilitate the navigating through the first_word which is zero. The last_word from the right is -1, which is American. Let us now concatenate. First word is 'Our' and the last word is 'American'.

S

```
In [16]: 1 concat_words(sentence)
['Our', 'nation', 'faces', 'a', 'profound', 'climate', 'crisis', 'that', 'is', 'impacting', 'every', 'American']
Our
American
Out[16]: 'OurAmerican'
```

This is basic data analytics or text analytics using basic functions.

7. If you want to print a word at even position:

1. Create a function. Give it a name. For example, 'get_even_position_words'. It takes one parameter.

Next, tokenize it since you are looking for a position.

Now, split the entire paragraph, sentence, or dataset that you stored in your object. It takes one parameter. Breaking the sentence into words helps you navigate and find the position you are looking for.

Now, look for all indexes using the function below:

```
In [ ]: 1 def get_even_position_words(text):  
2       words = text.split()  
3       return [words[i] for i in range(len(words)) if i%2 == 0]  
4
```

Note: The above function takes a sentence or document or paragraph and splits into words. It will then navigate through all the indexes. Every word has a specific index starting from zero. It is going to display the even numbers such as 0, 2, 4, 6, 8, 10, etc.

Now, let's run the code.

Let's implement the function into the entire sentence. Copy and paste the function and pass it to the sentence and run the code.

We don't know whether it's working. So, we simply create a function with specific logic. And now we are going to implement this function into the entire sentence. Now pass the sentence to the function and run it.

In the screenshot below, you can notice the even position of the words.

Our - 00, faces -02, profound - 04, crisis - 06, is - 08, and every - 10.

```
In [21]: 1 sentence
Out[21]: 'Our nation faces a profound climate crisis that is impacting every American'

In [18]: 1 def get_even_position_words(text):
2         words = text.split()
3         return [words[i] for i in range(len(words)) if i%2 == 0]
4

In [20]: 1 get_even_position_words(sentence)
Out[20]: ['Our', 'faces', 'profound', 'crisis', 'is', 'every']
```

It's going to display every even position of the words. So you can see Our, faces, profound, crisis, is, every.

8. If you want to print the last three letters of any text:

If you want to print the last three letters of any text, create another function, called 'get_last_n_letters'. This function takes two parameters - a text or data and number of letters you want to get. Let's return the text and look for -n (n = start from the last index), starting from the last index).

Let's run the code and implement it. To implement copy and paste the function, It takes two parameters. It takes the data as first sentence, for example, you want letter number 3 and run the code. Here is the screenshot displaying the 3 letters from -n - 'can':

```
In [22]: 1 def get_last_n_letters(text, n):
2         return text[-n:]
3

In [23]: 1 get_last_n_letters(sentence, 3)
Out[23]: 'can'
```

Note: When you try to put index with negative value or sign, it will count from right to left.

For example: -12 = Index 0, -11 = Index 1.

How about if you want to print each word of the given text or a sentence in the reverse order?

Create a function called `get_reverse` and return the text in reverse order. Next, run the code.

Now implement by giving sentence as input and run the code. The screenshot below displays the reversed order of the words within the sentence and the order of the characters within a word.

```
In [24]: 1 def get_reverse(text):
          2     return text[::-1]
          3

In [26]: 1 sentence

Out[26]: 'Our nation faces a profound climate crisis that is impacting every American'

In [25]: 1 get_reverse(sentence)

Out[25]: 'naciReMA yreve gniTcapmi si taht sisirc etamilc dnuoforp a secaf noitan ruO'
```

- To print each word of the given text or a sentence in reverse order, maintaining their sequence.

Create another function called `get_word_reverse`. It takes only one parameter.

Specify or declare a variable `'words = text.split()'` to split the entire document or paragraph or sentence.

Declare what to return using `return ' '.join(word[::-1] for word in words)`. It means you will reverse all the words in the sentence, and you will join them.

Apply or implement this function in the sentence.

You have now created the function that takes this sentence, splits it into words so you can navigate because you are going to use the 'for' loop. Now, you need the for loop to go to every word at a time. Right now, it's considered as one word though it is a sentence.

You cannot use 'for' loop unless the sentence is tokenized or split into words using the `split()` function. Now, let's test the function. The function changes the order of words, and it concatenates or joins.

```
In [29]: ► 1 def get_word_reverse(text):  
2     words = text.split()  
3     return ' '.join(word[::-1] for word in words)  
4
```

```
In [30]: ► 1 get_word_reverse(sentence)
```

```
Out[30]: 'ruO noitan secaf a dnuoforp etamilc sisirc taht si gnitcapmi yreve naciremA'
```

This is basic text analytics using 'for' loops and using some functions.

2. Tokenization

Tokenization refers to the procedure of splitting a sentence into its constituent parts – called unigrams. The reason we split a sentence or paragraph into words or sentences is to be able to navigate through them to look for something that interests us.

If you have a sentence and want to compare it with another word or sentence, we have to split them into words and compare them word by word.

First, import from the NLTK, the Natural Language Toolkit, which is installed.

Let's import a **word_tokenize**. The **word_tokenize** is the method used to tokenize or split a sentence into words. But before that, just make sure to compare **word_tokenize** with the **split function**.

```
In [ ]: 1 from nltk import word_tokenize, download
        2 download(['punkt', 'averaged_perceptron_tagger', 'stopwords'])
```

Split a Sentence vs Tokenize it

The **word_tokenize** is totally different to the `split()` function. For example, you have created this function earlier to split a sentence into words.

Run the code and apply it into the sentence **"I'm learning NLP fundamentals"**. When you try to split this sentence into words using the `split` function, it simply splits into words; but it does not consider factorizations. Sometimes the numbers and special characters are given as a different token or different words. That is the difference between `split` and `tokenize`.

```
In [33]: 1 def split_words(text):
        2     words = text.split()
        3     print(words)
```

The `tokenize` method or `word_tokenize` method considers the punctuation, numbers, strings of words, or a special or unigrams and it splits accordingly. Displayed below is the output of a `split` function.

```
In [34]: 1 split_words("I am learning NLP Fundamentals.")  
        ['I', 'am', 'learning', 'NLP', 'Fundamentals.']
```

Let's create another function called "**get_tokens**" and this **get_tokens** function takes one parameter, that is, the "**sentence**" and now we are going to return. Let us first tokenize using "word_tokenize" method. Next, pass it to the sentence, which is a parameter that you are going to use or the input you are going to use to pre-process. Let's return the words. Run the code.

```
In [35]: 1 def get_tokens(sentence):  
        2     words = word_tokenize(sentence)  
        3     return words  
        4
```

Let us now apply the `get_tokens()` and pass to it the sentence "**I am learning NLP Fundamental**". Now let's compare the split method or the split function to the word tokenize_method which is the NLTK. `Split()` is a Python function. The "**word_tokenize**" is a NLTK method used for text pre-processing. Now let's run the code. A name error is displayed – "name 'word_tokenize' is not defined as displayed in the screenshot below.

```
NameError                                Traceback (most recent call last)  
<ipython-input-36-22b8f34a8911> in <module>  
----> 1 get_tokens("I am learning NLP Fundamentals.")  
  
<ipython-input-35-d35609dd83b4> in get_tokens(sentence)  
      1 def get_tokens(sentence):  
----> 2     words = word_tokenize(sentence)  
      3     return words  
      4  
  
NameError: name 'word_tokenize' is not defined
```

This is because you did not run the function above. Sometimes, you have to run it twice just to double check. Now, run the function one more time and then apply the function. Given below is the screenshot that depicts it.

```
In [34]: 1 split_words("I am learning NLP Fundamentals.")
['I', 'am', 'learning', 'NLP', 'Fundamentals.']
```

```
In [39]: 1 def get_tokens(sentence):
2         words = word_tokenize(sentence)
3         return words
4
```

```
In [40]: 1 get_tokens("I am learning NLP Fundamentals.")
Out[40]: ['I', 'am', 'learning', 'NLP', 'Fundamentals', '.']
```

Now, you can see that in the `split()` function, the word 'Fundamental' ended with a full stop, whereas in the `get_tokens()` function, the full stop is considered an actual token. It remains the same for the `print()` function.

Sentence Tokenize

At a few instances, you may need to apply sentence tokenize. If you have a paragraph or if you have a chapter and you want to split this chapter or the paragraphs into sentences, you could use sentence tokenize. In this unit, you learned the **word_tokenize** because this is what applies to a very small sentence or a sentence with few words.

```
In [34]: 1 split_words("I am learning NLP Fundamentals.")
['I', 'am', 'learning', 'NLP', 'Fundamentals.']
```

```
In [39]: 1 def get_tokens(sentence):
2         words = word_tokenize(sentence)
3         return words
4
```

```
In [40]: 1 get_tokens("I am learning NLP Fundamentals.")
Out[40]: ['I', 'am', 'learning', 'NLP', 'Fundamentals', '.']
```

4. In order to view the list of tokens generated, we need to view it using the `print()` function. Insert a new cell and add the following code to implement this:

```
In [ ]: 1 print(get_tokens("I am reading NLP Fundamentals."))
```

```
In [ ]: 1
```

```
In [ ]: 1
```

```
In [ ]: 1
```



3. POS Tagging

This unit covers the PoS tagging concept.

In NLP, the term PoS refers to parts of speech. PoS tagging refers to the process of tagging words within sentences with their respective PoS.

The reason you need to learn about PoS of words is that it helps us understand the true meaning of the word.

Let's look into the list of possible PoS tagging. The NLTK comes with a collection of tools including PoS tagging. These tools are mainly used for text preprocessing and preparation.

Let's now print the list of all possible tagging. As given here, the dollar sign means dollar. If you scroll down, there are a number of taggings - CD means Cardinal, DT- determiner, EX- extential, FW -foreign word, IN preposition or subordinating conjunction, JJ is adjective or numeral ordinal. So there are a lot of abbreviations that represent a specific tagging within the English language.

Let's import the Natural Language Toolkit (NLTK) and word tokenize. Again, the need of word tokenize was explained in the previous section. It is used to split a part of the sentence, given dataset, or the given text into words so that you can navigate word by word.

Import the `word_tokenize` method, which will help you split the given text into words. The PoS tag - this is the method that helps annotate any word with its original words.

Create **`get_tokens`** function. This function takes a text, or an input and it splits this input into words and returns the list of those words. Let's do **`word_tokenize`**. Pass the sentence to it and run it. You need to return words.

So again, in this predefined or defined function, with the name `get_tokens` it takes a sentence as a parameter input, splits it into words, and reiterates those words the list of those words.

And now applying it on "I am reading NLP Fundamentals" and post to it add a full stop. Run it and it is done. You can see all the elements of the sentence gets split into parts, including the punctuation.

```
In [47]: 1 words = get_tokens("I am reading NLP Fundamentals.")
        2 print(words)
        ['I', 'am', 'reading', 'NLP', 'Fundamentals', '.']
```


Now you have the tokens, parts of speech, and specific elements of the sentence through which navigation is possible. Let's use the PoS tag method try to figure out it's part of speech and simply what it does.

Let's now use the `pos_tag()` method.

Let's create a function and give it a name `get_pos`, then `(word)`, and then it returns the part of speech. Now apply that to the word.

The words here refer to the sentence that we had here, which has already been tokenized or split into a unigram. Let's run the function. Now apply it on `pos_tag`. You will see an error message saying `pos_tag` is not defined. The importing of PoS tag from NLTK was not done. It is recommended to go through the error and see exactly what it means.

In this case, the error reads as name '`pos_tag`', which is the method used to generate a part of speech for the given words of a sentence and then gets split into parts. It simply means we forgot to run a declaration.


In serial number 50, the `pos_tag` is imported to create a function that splits our given text into PoS/unigrams, so that you can navigate through them. Pass the identified sentence that you want to try. Let's test the same sentence that was used earlier, *"I am reading NLP fundamentals."* The sentence has been split into parts. Now, the function created reads the sentence word by word and tries to determine it's part of speech.

```
In [52]: 1 words = get_tokens("I am reading NLP Fundamentals.")
          2 print(words)

['I', 'am', 'reading', 'NLP', 'Fundamentals', '.']
```

Now let's use the function on the sentence that we already tokenized.

```
In [54]: 1 get_pos(words)

Out[54]: [('I', 'PRP'),
          ('am', 'VBP'),
          ('reading', 'VBG'),
          ('NLP', 'NNP'),
          ('Fundamentals', 'NNS'),
          ('.', '.')]

```

To know what these abbreviations mean, go to the list and look for them.

PRP is pronoun possessive and RB is adverb. The part of speech helps us understand the true meaning of a word.



4. Stop Word Removal

In this unit, we will cover another critical preprocessing step, which is “stop word removal”.

Stop word removal is critical in terms of pre-processing the text in a way that it removes anything that wouldn't add any value to the analytics. It removes punctuations, numbers, and most common words like “a,” “am,” “and,” “the”, etc.

Import the necessary libraries such as the Natural Language Tool Kit (nltk). The nltk is already downloaded. Next, import the word tokenizer again. word_tokenizer method is used to split a text into words, parts, unigrams, and tokens. It helps navigate through the words and compare them to the list of the words that you want to remove. You will remove if there is a match, otherwise keep it.

Upon running the code, it simply imports all the methods that are needed. You are going to need to word tokenize and you are going to need a list of words and stop words. These stop words need to be in English. Since we are dealing with English statements, the text we are going to use is “stopwords.words(‘english’”).

```
In [56]: 1 stop_words = stopwords.words('english')
```

Now, let's print the stop words. Let's take a look these words just to be familiar with what you are supposed to remove from the text.

Use the print() function and pass into it the variable in which you have stored the English words and stop words that you imported from natural language toolkit. Let's print the words. Here is the screenshot displaying the most common words.

```
In [57]: 1 print(stop_words)

['I', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're", "you've", "you'll", "you'd", 'your', 'yours', 'yourself', 'yourselves', 'he', 'him', 'his', 'himself', 'she', "she's", 'her', 'hers', 'herself', 'it', "it's", 'its', 'its', 'elf', 'they', 'them', 'their', 'theirs', 'themselves', 'what', 'which', 'who', 'whom', 'this', 'that', "that'll", 'these', 'those', 'am', 'is', 'are', 'was', 'were', 'be', 'been', 'being', 'have', 'has', 'had', 'having', 'do', 'does', 'did', 'doin', 'g', 'a', 'an', 'the', 'and', 'but', 'if', 'or', 'because', 'as', 'until', 'while', 'of', 'at', 'by', 'for', 'with', 'about', 'against', 'between', 'into', 'through', 'during', 'before', 'after', 'above', 'below', 'to', 'from', 'up', 'down', 'in', 'o', 'ut', 'on', 'off', 'over', 'under', 'again', 'further', 'then', 'once', 'here', 'there', 'when', 'where', 'why', 'how', 'al', 'l', 'any', 'both', 'each', 'few', 'more', 'most', 'other', 'some', 'such', 'no', 'nor', 'not', 'only', 'own', 'same', 'so', 'than', 'too', 'very', 's', 't', 'can', 'will', 'just', 'don', "don't", 'should', "should've", 'now', 'd', 'll', 'm', 'o', 're', 've', 'y', 'ain', 'aren', "aren't", 'couldn', "couldn't", 'didn', "didn't", 'doesn', "doesn't", 'hadn', "hadn't", 'has', 'n', "hasn't", 'haven', "haven't", 'isn', "isn't", 'ma', 'mightn', "mightn't", 'mustn', "mustn't", 'needn', "needn't", 'sha', 'n', "shan't", 'shouldn', "shouldn't", 'wasn', "wasn't", 'weren', "weren't", 'won', "won't", 'wouldn', "wouldn't"]
```

As you see, these are the most common words that wouldn't add any value to the analytics. Sometimes they have a meaning but, in most cases, you don't have to care about these words. So, just take them out from your text and leave only words that would have a meaning and

would bring some insight to extract better information and make better decisions. Now, you know the list of words that you are going to remove.

To remove the stop words from a sentence, you first assign a string to the sentence variable and tokenize it into words using the `word_tokenize()` method.

First, create a variable called 'sentence' and assign the sentence to it. Create another variable called 'sentence_words', which is simply a tokenized version of the original sentence. Now, run the function to print the sentence_words. The screenshot below displays the tokenized words.

```
In [58]: 1 sentence = "I am learning Python. It is one of the most popular programming languages"
        2 sentence_words = word_tokenize(sentence)
```

5. To print the list of tokens, insert a new cell and add the following code:

```
In [59]: 1 print(sentence_words)

['I', 'am', 'learning', 'Python', '.', 'It', 'is', 'one', 'of', 'the', 'most', 'popular', 'programming', 'languages']
```

Now, it's easy to navigate through this variable and compare it to the stop words, since you have it already transformed or parsed into words.

To remove stop words, you need to loop through each word in the sentence, check whether there are any stop words, and then combine them to form a complete sentence.

Create **remove_stop_words()** function. This function takes two parameters, tokenize sentence and list of stop words. It is going to compare them using the 'for' loop. And it's going to keep only the words in the sentence that don't have a match in the word list in the stop words.

Let's do return as usual and use the `join()` function. Let's do 'for' loop – **"word for word in sentence if word not in stop_words"**. What it does is it looks for words in the sentence and keeps them if they have a match in the list of stop words. It and not keeps them otherwise. Let's run this.

```
In [60]: 1 def remove_stop_words(sentence, stop_words):
        2     return ' '.join([word for word in sentence if word not in stop_words])
        3
```

So far, everything looks fine. Let's check if everything is going to work. Let's print. Apply the function "remove_stop_words". Apply the 'sentence_words', which is the tokenized version of the original sentence. It also takes the list of stop_words as input. This is what you use to make a comparison and take an action if there is a match. Let's run it.

The screenshot below shows the removed stop words.

```
1 print(remove_stop_words(sentence_words, stop_words))
```

I learning Python . It one popular programming languages

Add your own stop words to the stop word list.

So, sometimes you might end up having a lot of common words, a lot of words that wouldn't add any value to your analytics. What you could do is update the list of stop words with your own words. If there is a word that you find with high occurrence, you could update the stop words with that word to be removed.

There are two or three more words in the sentence, such as, I, it, and a full stop. Let's add or extend three streams into the stop word list.

Now, you need to apply the function **remove_stop_words** again on this tokenized sentence. Now run the function.

For example, you want to add learning, and a comma.

Add specific words that you know (based on your experience or observations) that wouldn't add any value to the analytics. You could extend the list of stop words. You could update the list of stop words by adding the words you want to remove and run the function again.

Let's add the words 'learning' and a full stop to the list of stop words. It'd probably be 136 words. Now print. Let's first get the number of stop words; it's 182.

There are 182 words that you can remove if there is a match. But out of these 182, sometimes you are going to end up having some words that you don't really need, or you feel should be removed. For example, let's remove the words 'one', 'learning' and '.' Now run the function. The screenshot below displays the sentence with the removed stop words:

```
1 stop_words.extend(['I', 'It', 'one', 'learning', '.'])  
2 print(remove_stop_words(sentence_words, stop_words))
```

Python popular programming languages

The same process could be applied to a large text. You remove unnecessary words or strings and keep only what is needed for the analysis. The same method could be used to remove special characters to remove anything that is not a string. For example, numbers, white spaces, and special characters.

5. Text Normalization

This unit covers text normalization, which is a process of transforming a variation of a text or words into its standard format. For example, the words such as "US" and "United States", and "Bombay" and "Mumbai" are two different words. Although they are different, they mean the same thing.

Why keep both words? We should have only one format or one form to facilitate the analytics. For example, words such as "does", "doing", and "done" mean the same thing, but they are three different words.

Text normalization transforms these different words with the same meaning into a standard form. Unfortunately, there is no method that we can borrow from the NLTK or natural language toolkit. You need to do this manually using the `replace()` function.

Let's create a sentence – "I visited the US from the UK 22-10-18" and run it."

Now, let's replace "US" with "United States", "UK with United Kingdom", and "18" with "2018" in the sentence. To do so, use the `replace()` function and store the updated output in the "normalized_sentence" variable.

Create a function. It's always good to create functions to automate your snippets of code. Create a function called **normalize()**, which takes one input, i.e., text, which could be a sentence, word, or a paragraph.

In the given sentence, replace "US" with "United States", "UK with United Kingdom", and "18" with "2018". It's simple, and you could do the same with most common words or abbreviations, or words that are similar, and they have variation.

Now, the function is created with methods that are to be used or the words that are to be replaced. Run it. The function is created with logic and methods that you want to use or the words you want to replace. Run the code.

Now, let's apply that function into the sentence that is already created- "I visited US from the UK on 22-10-2018". Now, apply the function to the sentence and then print it out.

```
1 normalized_sentence = normalize(sentence)
2 print(normalized_sentence)
```

I visited the United States from the United Kingdom on 22-10-2018

Here, the `replace()` statement is used. There are other methods that you could use but are more complicated as it would require barcoding. The 'for' loop could be used that loop through

words in a sentence and transform them to whatever you need and if there is a match of a certain word, transform those words into a specific form that was agreed upon at the beginning.

Sometimes it doesn't matter if the sentence is used to create a normalized function. As far as you have the words you are looking for in any sentence, the normalized function would work. So in this example, the US and UK are two separate powers. Now, when they are normalized, the United States and United Kingdom are two superpowers. It simply looks into US and UK in this new sentence and replaces these two words with the given words that we used to build the function called normalized. Thus, text normalization is very simple and is addressed as the normalization from the helicopter view.



6. Spelling Correction

This unit covers spelling correction, which comes handy to find typos in documents, sentences, text, especially when:

- Text typing is done manually.
- There is no auto correction systems installed in your system.

Spelling correction uses a method called **speller**. And **speller** is not an NLTK package or method.

Firstly, you need to install **autocorrect** method and run it. It will take a moment. The library is called **autocorrect**. But out of this autocorrect, let's import a method called **speller**.

```
1 from nltk import word_tokenize
2 from autocorrect import Speller
```

The word tokenization is mainly used/is an essential critical part or process used for all NLTK pre-processing steps.

- If you require to perform lemmatization, you will need to tokenize a sentence.
- If you require to perform spell correction, you will need to tokenize text.
- If you require to perform a PoS tagging, you will need to tokenize the text.

Tokenization is important as it helps us navigate through words word by word, part by part, or unigram by unigram.

Let's import the word_tokenizer again from the NLTK (natural language toolkit). From the **autocorrect** method library, import the speller method. So on doing autocorrects, it supports many languages, including English.

So, while importing, you need to specify the language to be used. So in this case, the speller is going to use the language English. And now on applying this speller here on this word 'Natureal', of course, it's misspelled. So, now you can see if the autocorrect library will take care of this typo. Run the function. The output is 'Natural.'

```
1 spell = Speller(lang='en')
2 spell('Natureal')
```

```
'Natural'
```


Behind the scene, the speller compares the given word to an online dictionary that this autocorrect uses. It looks for the nearest word from the dictionary to the word given/misspelled word. Sometimes, there is more than a word that is close. But it uses an Octavian distance or probability to determine the optimal closest/nearest word from the dictionary to the given misspelled word or the word that needs to be corrected.

Below is the screenshot that displays a sentence with a lot of typos:

```
1 sentence = word_tokenize("Ntural Luang@age Processin deals with the art of extracting insightes from Natural Languaes")
```

This sentence has to be created and tokenized. What this tokenizer again does is it splits the entire sentence into parts/words. It enables you to navigate through those words and compare them to the words in the online dictionary, using the **speller**. This is a method that was imported with the autocorrect library and it's taken action to correct accordingly.

Now, create a sentence and tokenize it. You can do this in another way by creating a sentence and then overwriting it with the tokenized sentence. The first sentence is stored with a couple of misspelled words. Next, take the words or the sentence/database stored into a variable called **sentence** and tokenize it.

Now, let's print out the entire sentence after tokenizing it. The sentence is now tokenized. However, the spelling of 'Natural Language' is not corrected yet. It simply takes the entire sentence and splits it, tokenizes it into words.

```
In [79]: 1 sentence = "Ntural Luanguage Processin deals with the art of extracting insightes from Natural Languaes"
          2 sentence = word_tokenize(sentence)
          3 sentence

Out[79]: ['Ntural',
          'Luanguage',
          'Processin',
          'deals',
          'with',
          'the',
          'art',
          'of',
          'extracting',
          'insightes',
          'from',
          'Natural',
          'Languaes']
```

Now that we have got the tokens, loop through each token/word in the sentence, compare them to an online dictionary, correct the tokens (speller uses autocorrect to find the right spelling), and assign them to a new variable. It's simply comparison. Sometimes a word can have many words in a dictionary that they are closer in meaning to.

For example, in words such as 'their' or 'there', either one of them could be correct. But the method speller uses some Octavian distance mathematical equation behind the scenes that helps determine the closest word in a dictionary to a given misspelled word or a word that is to be corrected.

Create a function called **correct_spelling()**. It takes one parameter. It looks into the tokenized words it is going to spell. Here is the method imported using the English language. The spell method takes a look at every word in the list.

The `correct_spelling()` function takes a look at the tokenized words. And if they are misspelled, it is going to spell them right through comparing them with the given words in the dictionaries.

```
1 def correct_spelling(tokens):  
2     sentence_corrected = ' '.join([spell(word) for word in tokens])  
3     return sentence_corrected  
  
1 print(correct_spelling(sentence))  
2 #print(['Natural', 'Language', 'Procession', 'deals', 'with', 'the', 'art', 'of', 'extracting', 'insights', 'from', 'Natu
```

Run the code. Print it. In the tokenized words, the first three words were misspelled or had a typo in them. Now it's corrected.

```
1 print(correct_spelling(sentence))  
2 #print(['Natural', 'Language', 'Procession', 'deals', 'with', 'the', 'art', 'of', 'extracting', 'insights', 'from', 'Natu
```

Natural Language Processing deals with the art of extracting insights from Natural Languages

This is one of the methods for correcting misspelled words or typos. And this comes handy when you didn't have an autocorrection system, which most of the systems have.

7. Stemming

This unit covers the stemming method.

Stemming is a process or method that is used to remove all variations and forms of a certain text or words and keep only the root word of it, called stem.

There are four types of stemming. They are:

1. PorterStemmer()
2. SnowballStemmer()
3. LancasterStemmer()
4. RegexpStemmer() (Regular Expression Stemmer)

Among these, Porter **Stemmer** is the most common one.

Stemming is a tool that comes with Natural Language Toolkit (NLTK).

Import the stem from NLTK. Next, run the function. Run the cell to import.

Now create a function with two parameters or to impose the word to stem and the stemmer method. It returns the stemmer.stem(word). Take over at the method and the stemming. Specify which type of stemmer to use (porter stemmer, snowball stemmer or lancaster, regular express).

Let's use PorterStemmer() which is the most common one as shown below.

```
1 def get_stems(word, stemmer):  
2     return stemmer.stem(word)  
3 porterStem = stem.PorterStemmer()  
4
```

The stemming, for example, looks for common words with affixes and suffixes and try to remove them, adds on to a word and keep only the root of that word.

The reason why you need stemming is it does helps to reduce the dimensionality and it does helps narrow down the variations and the forms that help us to keep only the root of certain words.

For example, the word "product" might get transformed into production, which is a process for making a product and products with plural with s. So all these three words mean different

things. In fact, it has the root or the same root or the same meaning i.e., product. The stemming removes 'ion' and 's' keeping only words 'product'.

Sometimes the stemming makes a lot of typos because it removes affixes and prefixes. It does not correct the spelling of the root of a word.

Going forward, you will learn the next step of stemming, which is lemmatization that does a better job than stemming, but both of them they have pros and cons.

Stemming is faster but does not fix the spelling of a word; lemmatization is slower than stemming but it fixes the wrong spelling of any given root of words.

You've learned to create a function that does stemming. Now, let's apply this function to get stems. Let's apply it on 'production' and see what it does.

There's a misspelling here, "production", that's why it was not recognized. It has to be single or double quotations and you need to specify the stem or the method. For example, porterStem.

You have the parameter of the word and the stemmer, which is the method. Now apply the function that is created. The function takes the word and applies the stemmer method, which is porterStem. Once you execute the function, the output is displayed as 'product' as shown in the below screenshot.

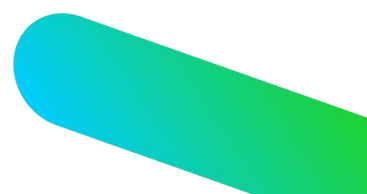
```
In [102]: 1 get_stems('production',porterStem)
Out[102]: 'product'
```

Let's have a look at some more examples of stemming for root word "product". Notice how porterStem is giving the root (stem) of the word "product."

The word 'products' with s is stemmed to 'product' by simply removing the 's' after product.

```
In [102]: 1 get_stems('products',porterStem)
Out[102]: 'product'
```

Similarly, the word producted (may not be an English word) is stemmed to product by removing the 'ed'.



```
1 get_stems('producted',porterStem)
'product'
```

Similarly, let's stem the word 'coming',

Use the porterStem method that was created earlier to stem the word. This function takes two parameters. Use the code below for the input 'coming'.

```
In [106]: 1 get_stems('came',porterStem)
Out[106]: 'came'
```

The word 'coming' is stemmed to 'come'. The word 'came' is stemmed to 'came'.

The word 'firing' is stemmed to 'fire'.

```
In [108]: 1 get_stems("firing",porterStem)
Out[108]: 'fire'
```

Similarly, the word 'battling' is stemmed to 'battl' which is a misspelled word.

```
In [109]: 1 get_stems("battling",porterStem)
Out[109]: 'battl'
```

The stemming simply looks for the narrowest root for a given word. It might make some mistakes such as misspellings, and unfortunately it does not correct them. To fix any misspelled outcome of the stemmer function, there is another method that can be applied. It is called **lemmatization**. Or you can also use the spelling correction function that we created in the previous section.

There are many other stemmers: Snowball Steamers, Lancaster, and Regular Expression.

So, let's take the paragraph and assign it to "Para". In this paragraph, we are going to do triple quotation. Single quotations are for one sentence, one line, and there is no apostrophe. Double quotations are for one line, one sentence, and if there is an apostrophe somewhere in the middle, it is always good to have double quotations. Otherwise, it might confuse the single quotations with apostrophes. If you are having more than a sentence or a line, it is always better to use triple quotations.

Here, the paragraph is created. You could try this with different sentences or paragraphs from any online website for the training purpose.

```
In [ ]: 1 para = '''In most languages, words get transformed into various forms when being used in a sentence. For example, the word  
2 sen_token = word_tokenize(sen)  
3 sen_sw = remove_stop_words(sen_token, stop_words)  
4 get_stems(sen_sw, porterStem)
```

The paragraph created is stored in the parameter called 'para'. Let's use most of the functions that were created earlier.

The word_tokenizer is the method that we are going to use to tokenize this entire paragraph.

Next, pass the para to word_tokenize. Then para stop_words (para_sw). If you remember, the **remove_stop_words** method or function that was created takes two parameters. The first parameter is the list of words that you want to treat, or you want to compare with the list of stop words. The stop words or tokens are the paragraph stored in the parameters, which you would tokenize using the **word_tokenize** method.

After that, let's stem it. Here para SW means para stop words, Next, you need to specify the stemmer method you are going to use. The beauty of the function is that you can automate the snippet of codes. Once you create a function to perform specific operations, you could call it anytime. Now, let's run the code.

Here is the output. Now from the sentence, you would notice that the word 'most' is gone because it is tokenized first. Upon removing stop words, the word 'most' was gone. Update stop words using the extend method that you've learned from the previous section.

```
In [111]: 1 para = '''In most languages, words get transformed into various forms when being used in a sentence. For example, the word  
2 para_token = word_tokenize(para)  
3 para_sw = remove_stop_words(para_token, stop_words)  
4 get_stems(para_sw, porterStem)
```

Out[111]: "in languages , words get transformed various forms used sentence for example , word `` product `` might get transformed ``
production `` referring process making something transformed `` products `` plural form necessary convert words base forms ,
carry meaning case stemming process helps us if look following figure , get perfect idea words get transformed base form"

Now, use the extend method and add any of these words that are not needed to the list and run the code. In the output, you can now see only the words that are needed for any analytics. To recap, a stemmer is a method of transforming a given word into its root. Unfortunately, the stemming removes the affixes and prefixes of word. The remaining word might not have any English meaning. That is one of the pros of this statement. It's very fast in transforming a word into its root but it creates misspelled words.

8. Lemmatization

This unit covers the lemmatization method, which is a process that fixes the incorrect result of the stemming method.

Stemming, as you have seen earlier, simply extracts the word, the root of the word through random axing in words. Whereas, lemmatization is the process of converting words to their base based on the grammatical form by applying the PoS tagging of word and also comparing the word to an online dictionary called **“WordNet”**.

Lemmatization is more accurate, but it's a very slow process compared to stemming. Stemming is faster, but it comes with incorrect results and the lemmatization is very slow.

Let's import **“WordNet”**, a dictionary you are going to use. Also import the method **WordNetLemmatizer** from the “Natural Language toolkit” that is already installed.

Upon running, the package WordNet is already up to date, which is good as shown below.

```
In [113]: 1 from nltk import download
          2 download('wordnet')
          3 from nltk.stem.wordnet import WordNetLemmatizer

[nltk_data] Downloading package wordnet to
[nltk_data] C:\Users\medwa\AppData\Roaming\nltk_data...
[nltk_data] Package wordnet is already up-to-date!
```

Let us now initialize the Lemmatizer by calling the method that was just imported. Save that into a parameter or into a variable called **“Lemmatizer”**.

Now that the word Lemmatizer is initialized, let's create a function that lemmatizes for us.

Create a function named **get_lemma(word)**. Lemmatizer is another type of stemmer, but it's very slow in terms of correcting the typo by comparing the stemmed word in a given online dictionary called WordNet.


```
In [ ]: 1 def get_lemma(word):  
2
```

get_lemma function takes a word because it works word by word. Return **lemmatizer.lemmatize** and pass a parameter or the input to it. Run the code.

Apply the word '**products**' to the get_lemma function. It takes almost two or three seconds. Instead of finding random axing word or root word of product, it looks for the grammatical base of the word 'products' and compares it with online dictionary for accuracy.

2. Create an object of the WordNetLemmatizer class. Insert a new cell and add the following code to implement this:

```
1 lemmatizer = WordNetLemmatizer()
```

3. Bring the word to its proper form by using the lemmatize() method of the WordNetLemmatizer class. Insert a new cell and add the following code to implement this:

```
1 def get_lemma(word):  
2     return lemmatizer.lemmatize(word)  
3
```

```
1 get_lemma('products')
```

The same applies for production. It compares with the dictionary and lets dictionary find the meaning or find the match.

Here is the output for some more words (get_lemma function):

```
coming : 'coming'  
came : 'came'  
loved : 'loved'  
love : 'love'
```

Difference between "get_lemma" and "get_stems" methods

Let us take the example of the word 'loved'.

Stemmer removes the word 'd' from the word "loved". Sometimes, it could simply be a random word and does not compare it with anything. Most of the time you will get a misspelled word under root word.

The get_lemma function does stemming. It looks for the root of the grammatical word of the word used in the PoS tagging and also using the online dictionary. That's why it's so slow compared to the stemmer. But it's more accurate. There is no misspelling as it fixes typos. That

is the shortcoming of the steamer.

The below image depicts the same.

```
In [117]: 1 get_lemma('production')  
Out[117]: 'production'
```

5. Similarly, use the input as coming now:

```
In [123]: 1 get_stems("loved",stemmer)  
Out[123]: 'love'
```

```
In [124]: 1 get_lemma('loved')  
Out[124]: 'loved'
```

09. Named Entity Recognition (NER)

This unit covers Named Entity Recognition.

The Named Entity Recognition (NER) is the process of extracting important entities, such as person names, place names, organization names, etc.

Here is the list of some entities:

NE Type
ORGANIZATION
PERSON
LOCATION
DATE
TIME
MONEY
PERCENT
FACILITY
GPE

The Named Entity Recognition comes handy in understanding the true meaning of a given word rather than understanding its grammatical meaning through PoS tagging. The named entity helps understand the true physical meaning of the word.

Let's first import a lot of methods from NLTK. Import the PoS tagging again because it helps extract the names of the true entities represented by word if we know that the right PoS tagging and also since you are going to look into word by word, you have to apply the word_tokenizer.

```
In [ ]: 1 from nltk import download
        2 from nltk import pos_tag
        3 from nltk import ne_chunk
        4 from nltk import word_tokenize
        5 download('maxent_ne_chunker')
        6 download('words')
```

Split the given text into words or into parts and navigate through those parts to apply PoS tagging. Once you apply PoS tagging, apply the **ne_chunk**. The **ne_chunk** is the method used to extract named entities or to apply NER.

Let's create a sentence variable and assign it to a string as shown below.

```
In [ ]: 1 sentence = "We are reading a book published by Novels which is based out of Atlanta."
```

Let's import the PoS tagging and the methods that you need to use.

Now, create a variable sentence and enter the sentence.

The next step is to create a function as usual. This function number one is going to take one parameter and that parameter should be word_tokenize. Let's break the text into parts using word_tokenize.

Once you tokenize, PoS tag it. PoS tagging comes after breaking the sentence into parts. After PoS tagging it, apply the NER method or **ne_chunk**.

Tokenize the given text first. Once you tokenize, add some PoS tagging and grammatical tagging to help determine the right named entity recognition or to extract the right name of the entity it represents.

Now, store the sentence in a variable called 'i'. Since you are using **ne_chunk()**, you need to determine the type of the output. Are you trying to extract output just to point out to that entity? Or are you trying to extract the Named entity instead. That is controlled by adding one parameter called 'binary'. Put everything into parenthesis as shown below:

```
In [ ]: 1 def get_ner(text):  
2       i = ne_chunk(pos_tag(word_tokenize(text))), binary = True)
```

To return, look through the given tokenized text.

```
In [ ]: 1 def get_ner(text):  
2       i = ne_chunk(pos_tag(word_tokenize(text))), binary = True  
3       return[a for a in i if len(a)==1]  
4
```

If we have a token, that has at least one character, you have got to process it and PoS tag it. You have to apply the **ne_chunk** to see if you can extract any important named entities or name of an entity. Run the code.

Now apply the `get_ner` and apply the sentence. Here is the output. It is tree-based. You can see three trees here.

```
In [150]: 1 def get_ner(text):
          2     i = ne_chunk(pos_tag(word_tokenize(text)), binary = True)
          3     return[a for a in i if len(a)==1]
          4

In [151]: 1 get_ner(sentence)

Out[151]: [Tree('NE', [('Novels', 'NNP')]), Tree('NE', [('Atlanta', 'NNP']))]
```

If you were to set the `binary` parameter to `false`, then instead of having an `NE`, you would have an actual name of the entity. Let's do that.

Set the `binary` to `false`. Run the code. Now let's apply it to the sentence and compare the two results.

```
In [153]: 1 get_ner(sentence)

Out[153]: [Tree('PERSON', [('Novels', 'NNP')]), Tree('GPE', [('Atlanta', 'NNP']))]
```

If you notice, it thinks that 'Novels' is a person, but it's a library. Atlanta is a GPE, it's a location or geographical area, whereas the other one is a person.

Setting the `binary` to `false` is more meaningful as it extracts more meaningful information compared to setting the `binary` to `true`.

So now, let's go to Google. Open Yahoo website and randomly pick a paragraph.

Below is the screenshot displaying the imported the third library called `spacy`. It is a library used for data visualization.

```
In [154]: 1 # Import spacy and displacy
          2 import spacy
          3 nlp = spacy.load("en_core_web_sm")
          4 from spacy import displacy
```

Let's import this library. Now paste the paragraph copied from the Yahoo website. Since it's a paragraph with many lines, it's better to put it inside triple quotation marks. Otherwise, it may

not work because of the lining. Also it might have some apostrophes or other quotations inside.

```
In [155]: 1 # Create a text as NLP object
          2 para = nlp(u"""A woman in Pennsylvania has experts flummoxed by the animal she discovered outside of her home.Christina I
```

Now transform the entire sentence to a NLP object and store it in a variable called 'para', which stands for paragraph. Let's take a look at this paragraph.

```
In [155]: 1 # Create a text as NLP object
          2 para = nlp(u"""A woman in Pennsylvania has experts flummoxed by the animal she discovered outside of her home.Christina I
```

```
In [156]: 1 para
```

```
Out[156]: A woman in Pennsylvania has experts flummoxed by the animal she discovered outside of her home.Christina Eyth rescued the an
imal earlier this week near her home in Fairfield Township after finding paw prints outside of her door. Eyth followed the t
racks, assuming they had belonged to her neighbor's dog after they had gotten loose. The tracks ended up leading her to dire
ctly an unidentified animal, which Eyth said was exhibiting scared behavior.
```

Applying the named entity recognition would help understand the true meaning of some of the uncommon words that you might not be familiar with. To do that, use '**displacy.render**', which is a method that comes with this library.

Apply **displacy.render** to the paragraph. Let's apply some methods, themes, and styles. Run the code.

```
In [157]: 1 displacy.render(para, style='ent', jupyter=True)
```

A woman in **Pennsylvania GPE** has experts flummoxed by the animal she discovered outside of her home. **Christina Eyth PERSON** rescued the animal **earlier this week DATE** near her home in **Fairfield Township GPE** after finding paw prints outside of her door. **Eyth ORG** followed the tracks, assuming they had belonged to her neighbor's dog after they had gotten loose. The tracks ended up leading her to directly an unidentified animal, which **Eyth ORG** said was exhibiting scared behavior.

You can notice a beautiful outcome.

Pennsylvania is recognized as a geographical location. Christina Eyth is a person. Eyth is an organization. Of course, this is not accurate. Fairfield Township is a city. Eyth is organization. Here, the accuracy might not be that high. You could definitely improve the quality of this named entity recognition by applying regular expression.

10. Word Sense Disambiguation

Sometimes you are going to have similar words with different meanings. A word's meaning depends on its association with other words in a sentence. This means two or more words with the same spelling may have different meanings in different contexts.

Word sense disambiguation comes handy in understanding the true meaning of a word depending on the associations with other words and the context.

The algorithm used for this purpose is called **Lesk algorithm**, which has a huge corpus in the background. It uses an online dictionary called "**WordNet**", similar to the dictionary that lemmatizer uses.

WordNet contains definitions and synonyms of all possible words and languages, and then it takes a word and the context as it posts it and finds a match between the context and all the definitions of the word. The meaning with the highest number of matches with the context of the word will be returned. This is what word sense disambiguation does, using the **Lesk algorithm**.

Let's import the methods that you are going to use. Import **Lesk** from NLTK. **Lesk** is an NLTK method. Since you will be navigated through words of the given text, you will need to import `word_tokenizer`. This will help you split the given text into words, unigrams, and small parts.

Let us create two sentences- **sentence one**, **sentence two**. These two sentences should have at least one same word.

Sentence 1: Keep your savings in the **bank**.

Sentence 2: It's so risky to drive over the **banks** of the road.

The word 'bank' in these two sentences have the same spelling, but do they have different meanings?



Let's figure it out by applying the Lesk method that is imported from the NLTK.

Create a function called **get_synset**. It takes two parameters- the sentence and the word. The screenshot below displays the sentence that you created, you are going to compare, and this is the word you are going to search the meaning for in associations with other words in this sentence.

```
1 def get_synset(sentence, word):  
2
```

Let's 'return' the lesk algorithm, which means applying the Lesk algorithm on this word. Before we apply to Lesk, we need to **word_tokenize** the sentence and search for the given word. Use Lesk algorithm and run it.

2. Declare two variables, sentence1 and sentence2, and assign them with appropriate strings. Insert a new cell and the following code to implement this:

```
In [163]: 1 sentence1 = "Keep your savings in the bank"  
2 sentence2 = "It's so risky to drive over the banks of the road"
```

3. To find the sense of the word "bank" in the preceding two sentences, use the Lesk algorithm provided by the nltk.wsd library. Insert a new cell and add the following code to implement this:

```
In [ ]: 1 def get_synset(sentence, word):  
2     return lesk(word_tokenize(sentence), word)  
3
```

Now look for sentence number one using get_synset, and add parameters such as 'sentence1' and 'bank'. For the word 'bank' in sentence 1, the context of the words used is considered. So it is going to consider 'saving' as a word to determine the true meaning of the word 'bank'.

The second sentence, it is going to look at the words such as 'risky, drive, and road to determine the true meaning of the word bank as well.

The two sentences are created with one word in common. Next, a function is created that looks through this. This function tokenizes all these sentences and applies the Lesk method or algorithm. And now, apply it on the first sentence looking into the word 'bank.'

The word 'bank' is not defined, so, let's run this first, get_synset.


```
In [166]: 1 def get_synset(sentence, word):
          2     return lesk(word_tokenize(sentence), word)
          3
```

```
In [167]: 1 get_synset(sentence1, "bank")
```

```
Out[167]: Synset('savings_bānk.n.02')
```

Now run the code. The output reads as saving_banks. It refers to the bank as a physical entity or place where you can go and save or open an account and save your money. So, here the saving bank refers to the container to keep money, it could be at home, it could be in any different location.

Let's apply the same function to the second word in the second sentence. And let's take a look. So it says here bank version 0.7.

The other one savings_bank. Its version number is 02. This is a reference into WordNet dictionary.

```
In [163]: 1 sentence1 = "Keep your savings in the bank"
          2 sentence2 = "It's so risky to drive over the banks of the road"
```

3. To find the sense of the word "bank" in the preceding two sentences, use the Lesk algorithm provided by the nltk.wsd library. Insert a new cell and add the following code to implement this:

```
In [166]: 1 def get_synset(sentence, word):
          2     return lesk(word_tokenize(sentence), word)
          3
```

```
In [167]: 1 get_synset(sentence1, "bank")
```

```
Out[167]: Synset('savings_bank.n.02')
```

Here, savings_bank.n.02 refers to a container for keeping money safely at home. To check the other sense of the word "bank," write the following code:

```
In [168]: 1 get_synset(sentence2, 'bank')
```

```
Out[168]: Synset('bank.v.07')
```

They are different based on the output. Here, first one, saving_bank.n.02 refers to the container and the second one refers to a road and slope in the turn of the road.

It is so simple how we can solve disambiguation or solving meanings of same words with different contexts or understand the meaning of similar words when they are associated with different words and a different context.

11. Sentence Boundary Detection

Sentence boundary detection is a method of detecting where one sentence ends and another begins.

You might think that punctuation could be the best method to detect the end of a sentence. Unfortunately, punctuation is used for humans. For machines, if you have an abbreviation like Mr. and you have a dot (.) in a sentence it will try to preprocess a text with an abbreviation and full stop as the end of the sentence.

There is a process or method that comes with natural language toolkit (NLTK) and that is **sent_tokenize**. The `sentence_tokenize` is similar to the word 'tokenize'. The word 'tokenize' simply splits the entire sentence into words. It takes the whitespace, special characters, numbers, and punctuations into consideration that's compared to the `split` method.

The **sent_tokenize** splits a text into sentences using the boundaries determined by either punctuation or the meaning of a sentence.

Import **sent_tokenize** from the toolkit that you have already installed.

```
1 import nltk
2 from nltk.tokenize import sent_tokenize
```

Next, create a function called `a sentence`. With that function, return **sent_tokenize** and pass the text to it.

```
1 def get_sentences(text):
2     return sent_tokenize(text)
3
```

Let's apply **get_sentences()** function to a text. You need more than a sentence. Now copy and paste the text within the parenthesis. This is the sentence you will use.

```
1 def get_sentences(text):  
2     return sent_tokenize(text)  
3  
1 NOW! There's a mind-bending array of sales to take advantage of, and Amazon is, of course, at the center of the action!"
```

It is going to be so hard to determine where a sentence ends, even as human.

The **sent_tokenize** method that comes with NLTK is going to look into the sentence that you feed or store in this object, or you pass to this function. It is going to look into the boundaries of your sentence. That makes sense. It's not going to be based 100% on punctuations. But it is going to be based on the meaning as a complete sentence.

Now that the function is already created, pass the sentence (with a lot of punctuations) to the function.

The define function that is created takes text and the **sent_tokenize** function splits the text into sentences rather than splitting it into words. Let's apply the function to the sentence that is copied from other websites. You can see four complete sentences as output in this one paragraph.

```
In [176]: 1 get_sentences("You've obsessed over it, dreamed about it, saved up for it. Well, guess what? Black Friday weekend i  
Out[176]: ["You've obsessed over it, dreamed about it, saved up for it.",  
          'Well, guess what?',  
          'Black Friday weekend is NOW!',  
          "There's a mind-bending array of sales to take advantage of, and Amazon is, of course, at the center of the action"]
```

Let's apply the same with a different sentence with less punctuations. Mr. You see here we got the dot (.) in the sentence, which represents the end of the sentence and theory. By practice, it could be an apostrophe, abbreviation, or many, many things.

Let's use the **sent_tokenize** method which does not take into consideration the punctuation. Otherwise, it is going to return him and Mr dot(.) as a complete sentence. So the dot(.) after Mr. here is not a full stop, it is an abbreviation.