

This line of code loads the model
that will be used in tokenization

Statistical model used by Spacy <https://spacy.io/models>



```
nlp = spacy.load("en_core_web_sm", max_length=1529140)
```



A parameter that
ensures that we don't
work with files too big.
If there are problems,
the number should be
increased. This means
more resources used.

This line of code creates the tokenizer based on the vocabulary available in the model. It helps in tokenization because you are telling the computer that word constructs like “don’t” exist.

Bonus: It makes the code run faster because we are using only parts of Spacy

```
tokenizer = Tokenizer(nlp.vocab)
```

Creates an empty list and stores it in result. This is where we will keep our tokens before we save them.

```
result = list()
```

A for loop helps us go over a list of items. A list of items is generated by the pipe method in tokenizer. The pipe method receives 2 parameters. A list of texts and how many texts to process at once.

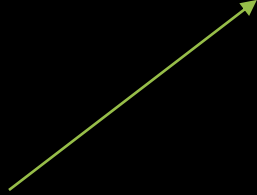
A for loop works by accessing each element of a list one at a time.

```
for i in [1,2,3]:  
    print(i)
```

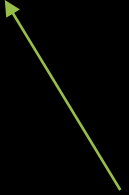
Will result in:

1
2
3

```
for tokens in tokenizer.pipe(findTxts('TextFiles'), batch_size=50):
```



The pipe method waits for the function inside the brackets to be finished before executing



The findTxts function receives a String path where are the text files are stored. Using this path, it will create a list of texts out of the texts in the Annual Reports

We are going to add the tokens we found to the list that will hold all the tokens in the end. We must do this because we are going one at a time through each processed file.

I will talk in detail about what this does at the end

```
for tokens in tokenizer.pipe(findTxts('TextFiles'), batch_size=50):
```

```
    result.extend(tokens.to_array("ORTH"))
```

Due to the nature of spacy, the tokens are stored as positions pointing to the words in vocabulary. Every word in the vocabulary has a number as an identifier. "ORTH" is just telling spacy to give us the positions because it can do a lot more!

Counter is a type of dictionary in python. It receives a list as an argument and then counts how often each element occurs.

A dictionary is a key:value pair.

Example:

```
{'apple':1,'car':2}
```

This means that if I write dictionary['apple'] it will give me the value 2.

```
pos_counts = Counter(result)
```

In Python, when you work with files you must open and then close them. This is a shorter syntax for that. When the code that runs inside this syntax is done, it will close the file. The open function receives a path, a mode, a newline and a type of encoding. At the end it generates an object to be used to write.


If you write a line, what it should input




Object used to write



```
with open("Counts/CSRWordFreqDict.csv", mode="w+", newline="", encoding='utf-8') as csv_file:
```



w+ means opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, it creates a new file for reading and writing.



Makes special characters that are not useful be ignored

The write method is used with a special syntax that helps in writing to files. It can be used in other scenarios also.

```
print("%s" % ('apple'))
```

"%s" is the pattern

% is the link

('apple') is the word that falls into the pattern. If I were to put a number, it would throw an error.

```
csv_file.write("%s,%s\n" % ('word', 'count'))
```

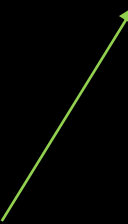

The `most_common` method generates a sorted list. The sorting is done based on the count. The most common element is the first one. In this case it will return a list formed of tuples. The tuples are formed of the position and the count.

Tuples are group of elements. `(1,2)` is a tuple made from 1 and 2. A list of tuples looks like this:
`list = [(1,2),(5,5),(3,2)]`
At position 0 you get `(1,2)`. This can be accessed by asking for the its position. `list[0][0]` gives 1 and `list[0][1]` gives 2.

```
for orth_id, count in pos_counts.most_common():  
    csv_file.write("%s, %d\n" %(tokens.vocab.strings[orth_id], count))
```



Splits the tuple in 2 for us



Searches in the vocabulary for the word at the position and returns it

Putting it all together

```
# 'w' open for writing '+' open a disk file for updating (reading and writing)
with open("Counts/CSRWordFreqDict.csv", mode="w+", newline="", encoding='utf-8') as csv_file:
    # headers of csv
    csv_file.write("%s,%s\n" % ('word', 'count'))
    # most_common() returns a list of ordered tuples with the key and count
    for orth_id, count in pos_counts.most_common():
        csv_file.write("%s, %d\n" %
                        (tokens.vocab.strings[orth_id], count))
# get the duration of the run printed
```

The following slides are functions defined outside the main code that are used throughout all the code.

```
def findTxts(path):
```

```
    ...
```

Function that receives a path for text files to read them and put them all in an array.

It does this by going to the path, checking each file there and taking those that end in .txt.

It then reads from the files and appends them to a list that is returned at the end of the execution.

The files should be in the TextFiles/ path relative to where the script is placed.

Parameters:

path - String

Output:

A list containing the texts in TextFiles/

```
    ...
```

```
result = list()
```

```
for root, dirs, files in os.walk(path):
```

```
    for file in files:
```

```
        if file.endswith('.txt'):
```

```
            filePath = open('TextFiles/'+file,  
                             'r', encoding='utf-8')
```

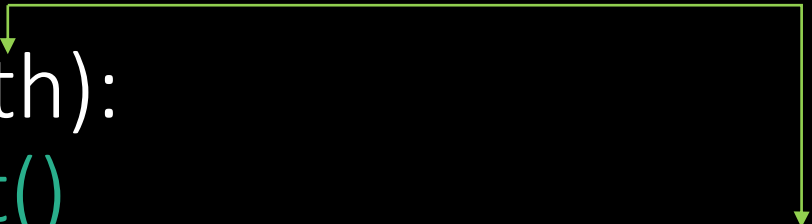
```
            text = filePath.read()
```

```
            result.append(cleanText(text))
```

```
return result
```

os.walk returns the directory path, directory names, and file names. It scans all the directories and subdirectories bottom-to-up. Because we use only one directory, we go through the files in it and check if the files end in .txt

More details about os.walk() at https://www.tutorialspoint.com/python/os_walk.htm



```
def findTxts(path):  
    result = list()  
    for root, dirs, files in os.walk(path):  
        for file in files:  
            if file.endswith('.txt'):
```

os.walk returns the directory path,
directory names, and file names. It
scans all the directories and
subdirectories bottom-to-up.
Because we use only one directory,
we go through the files in it and
check if the files end in .txt

This is the mode. 'r' is for mode set to reading.

filePath = *open*('TextFiles/'+file,'r', encoding='utf-8')

text = filePath.*read*()

Reads the whole
document

result.*append*(cleanText(text))

return result

Returns result of
function and exists
findTxts

Function that cleans the text

```
def findTxts(path):
```

```
    ...
```

Function that receives a path for text files to read them and put them all in an array.

It does this by going to the path, checking each file there and taking those that end in .txt.

It then reads from the files and appends them to a list that is returned at the end of the execution.

The files should be in the TextFiles/ path relative to where the script is placed.

Parameters:

path - String

Output:

A list containing the texts in TextFiles/

```
    ...
```

```
result = list()
```

```
for root, dirs, files in os.walk(path):
```

```
    for file in files:
```

```
        if file.endswith('.txt'):
```

```
            filePath = open('TextFiles/'+file,  
                             'r', encoding='utf-8')
```

```
            text = filePath.read()
```

```
            result.append(cleanText(text))
```

```
return result
```

```
def cleanText(text):  
    ...  
  
    Function that receives a string. The transformation it applies on the string are the following:  
    1) Sets it to lowercase  
    2) Removes any characters except letters,-,-,', ' and whitespaces  
    3) Replaces multiple whitespaces in just one  
  
    Parameters:  
    text - String  
  
    Output:  
    Cleaned String  
    ...  
  
    # text to lowercase  
    text = text.lower()  
    # keep only letters, -, ' and space  
    text = re.sub(r"[^A-Za-z-\\-\\'\\'\\ ]", ' ', text)  
    # replace multiple whitespace with just one  
    return re.sub(r"\s+", ' ', text)
```


We are using the default
RegularExpression library in python.
RegularExpression are a way of
finding pattern in text and doing
different operations with it.

<https://regexr.com/> is a good website
for trying out regex

```
def cleanText(text):  
    text = text.lower()  
    text = re.sub(r"[^A-Za-z—\-\'\"]", '', text)  
    return re.sub(r"\s+", '', text)
```

Sets the text to lowercase


Replaces all characters matching this pattern. In this case the pattern describes all the characters that are not letters, —, -, ', ' or whitespace

Replace multiple instances of whitespace with just one

NPs

noun_chunks will access the base NPs found in the text. The extracted NPs are of type base noun phrase, or “NP chunk”, which is a noun phrase that does not permit other NPs to be nested within it – so no NP-level coordination, no prepositional phrases, and no relative clauses

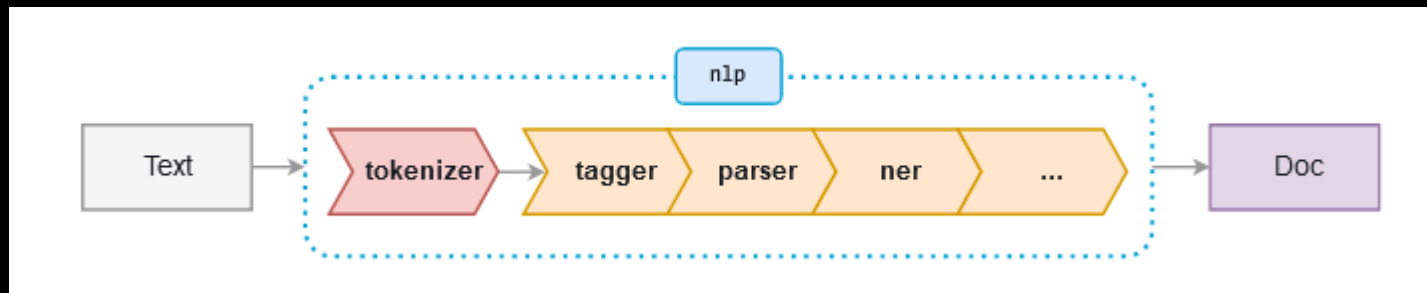
2 processes will run at the same time to process text



```
for doc in nlp.pipe(findTxts('TextFiles'),batch_size=25,n_process=2:  
    count.extend(doc.noun_chunks)
```

Spacy (<https://spacy.io/usage/spacy-101>)

In `texToDict.py` we are using only the tagger. In a complete pipe, there are multiple phases. The tokenizer is always the first step. Then it is followed by the tagger, parser, etc. For example, the tagger detects which part of speech it is and then the parser will make the link between the different parts of speech in the sentence.



After the whole pipe is done, then the result is the doc object that can be used to extract the information from.

A dataframe is a data structure that holds tabular data. Due to a problem in comparing the different strings that came as a result from `noun_chunks`, we will save the results in a dataframe and then count up each unique NP.

```
>>> d = {'col1': [1, 2], 'col2': [3, 4]}
>>> df = pd.DataFrame(data=d)
>>> df
   col1  col2
0     1     3
1     2     4
```

A dataframe receives a dictionary as the definition for its structure.

```
df = pd.DataFrame({'NP': count})
```

The result of this is a table that contains each NP

1	NP
2	united states securities and exchange commission washington d c form
3	one annual report
4	section
5	d
6	the securities exchange act
7	the fiscal year
8	december or transition report
9	section
10	d
11	the securities exchange act
12	the transition period

We want two column in the dataframe. One with the NP and one with the count. To achieve this, we set the next column to be 1 for row in the first column

When a dataframe receives between square brackets a string that doesn't exist in the column list, it will create a new column that receives the value assigned.

```
df['count'] = 1
```

The result of this is a table that contains each NP

1	NP	count
2	united states securities and exchange commission washington d c form	1
3	one annual report	1
4	section	1
5	d	1
6	the securities exchange act	1
7	the fiscal year	1
8	december or transition report	1
9	section	1
10	d	1
11	the securities exchange act	1
12	the transition period	1
13	commission file number - mastec inc exact name	1
14	registrant	1
15	its charter	1

We then save the dataframe to a csv so we can avoid the issue where similar values are not recognized as such. The to_csv method receives a path and then other optional parameters. In this case we tell it to save it without the index.

```
df.to_csv('Counts/TestSpacyNP.csv', index=False)
```

This would be the index column



1	NP	count
2	united states securities and exchange commission washington d c form	1
3	one annual report	1
4	section	1
5	d	1
6	the securities exchange act	1
7	the fiscal year	1
8	december or transition report	1
9	section	1
10	d	1
11	the securities exchange act	1
12	the transition period	1
13	commission file number - mastec inc exact name	1
14	registrant	1
15	its charter	1

The dataframe is read from the file we just saved it in while mentioning that the header is the first column. This means that the name of the columns are put in the first row in the csv.

```
df = pd.read_csv('Counts/TestSpacyNP.csv', header=0)
```


It groups each unique NP and then sums up the value in the other columns, meaning that each 1 will be summed depending on how often the NP occurs. After the group by, the NP is the index.

```
dfAgg = df.groupby(['NP']).sum()
```

NP	count
we	10580
it	1662
the company	1157
employees	1137
they	1126
...	...
human and workplace rights	1
human activity	1
human activities	1
huge savings	1
's well-being	1

Resetting the index will have NP as a column again. The `sort_values` method will receive a column and which type of sorting it should be. We then save it to a new csv file.

```
dfAgg.reset_index().sort_values(by='count', ascending=False).to_csv('Counts/CSRNPWordFreqDict.csv', index=False)
```

	NP	count
84494	we	10580
34742	it	1662
67242	the company	1157
23780	employees	1137
79592	they	1126
...
32012	human and workplace rights	1
32010	human activity	1
32009	human activities	1
32006	huge savings	1
85921	's well-being	1

VPs

To identify verb phrases, we must define our own pattern. It consists of an optional VERB, 0 or multiple ADVERBs and at least a VERB.

POS stands for part of speech

OP stands for operator.

```
pattern = [{'POS': 'VERB', 'OP': '?'},  
           {'POS': 'ADV', 'OP': '*'},  
           {'POS': 'VERB', 'OP': '+'}]
```

To identify verb phrases, we must define our own pattern. It consists of an optional VERB, 0 or multiple ADVERBs and at least a VERB.

POS stands for part of speech

OP stands for operator

```
pattern = [{'POS': 'VERB', 'OP': '?'},  
           {'POS': 'ADV', 'OP': '*'},  
           {'POS': 'VERB', 'OP': '+'}]
```

To identify phrases with Spacy, we will make use of its Matcher class. At creation it receives the vocabulary we are using. To add the pattern we just have defined, we use the add method that receives 3 parameters. A name is given to the pattern, a None is given because there is no callback needed and then the pattern is provided.

```
matcher = Matcher(nlp.vocab)
matcher.add("Verb phrase", None, pattern)
```

Matcher returns a list of tuples that contains the doc and the start and end of the phrase matched. To get the matched phrases, a list comprehension is used to shorten the code.

A list comprehension is a shorter way of writing a for loop to generate a list

<https://www.pythonforbeginners.com/basics/list-comprehensions-in-python>

If you used to do it like this:

```
new_list = []
for i in old_list:
    if filter(i):
        new_list.append(expressions(i))
```

You can obtain the same thing using list comprehension. Notice the append method has vanished!

```
new_list = [expression(i) for i in old_list if filter(i)]
```

```
for doc in nlp.pipe(findTxts('TextFiles'), batch_size=25, n_process=2)
    spans = [doc[start:end] for _, start, end in matcher(doc)]
```

```
spans = list()
for _, start, end in matcher(doc):
    spans.extend(doc[start:end])
```