Reference

This is a DataCamp course.

Regular expressions & word tokenization

Practicing regular expressions: re.split() and re.findall()

In [2]:

```
import re
my_string = "Let's write RegEx!  Won't that be fun?  I sure think so.  Can you find 4 sentences?  Or perhaps, al

sentence_endings = r"[.?!]"
print(re.split(sentence_endings, my_string))

# Find all capitalized words
capitalized_words = r"[A-Z]\w+"
print(re.findall(capitalized_words, my_string))

# Split my_string on spaces
spaces = r"\s+"
print(re.split(spaces, my_string))

# Find all digits
digits = r"\d+"
print(re.findall(digits, my_string))
```

```
["Let's write RegEx", "  Won't that be fun", '  I sure think so', '  Can you find 4 sentences', '  Or perhaps,
all 19 words', '']
['Let', 'RegEx', 'Won', 'Can', 'Or']
["Let's", 'write', 'RegEx!', "Won't", 'that', 'be', 'fun?', 'I', 'sure', 'think', 'so.', 'Can', 'you', 'find',
'4', 'sentences?', 'Or', 'perhaps,', 'all', '19', 'words?']
['4', '19']
```

Word tokenization with NLTK

Utilize word_tokenize and sent_tokenize from nltk.tokenize to tokenize both words and sentences from Python strings.

In [3]:
```
scene_one = "SCENE 1: [wind] [clop clop clop] \nKING ARTHUR: Whoa there!  [clop clop clop] \nSOLDIER #1: Halt!  W
```

In [4]:
```
from nltk.tokenize import sent_tokenize
from nltk.tokenize import word_tokenize

# Split scene_one into sentences: sentences
sentences = sent_tokenize(scene_one)

tokenized_sent = word_tokenize(sentences[3])

unique_tokens = set(word_tokenize(scene_one))
#set automatically makes a unique set of elements.

print(unique_tokens)
```

```
{'yet', 'this', 'ratios', 'its', 'could', 'minute', 'Well', 'here', 'swallows', 'guiding', 'beat', ']', 'non-mi
gratory', 'pound', 'under', 'every', 'SCENE', 'swallow', 'will', 'carried', 'snows', 'if', 'wind', 'but',
'...', 'length', 'interested', 'tell', 'why', 'son', 'kingdom', 'Not', 'lord', 'with', 'point', 'I', 'feather
s', 'forty-three', 'It', 'sovereign', 'simple', 'creeper', 'King', 'our', 'he', 'KING', 'these', 'strand', 'Cam
elot', 'all', 'five', 'have', 'But', 'you', 'court', 'master', 'temperate', 'Patsy', 'husk', 'Britons', 'secon
d', '!', 'seek', 'order', 'must', 'by', 'Supposing', ':', 'climes', 'covered', 'Are', 'on', 'halves', 'coconut
s', 'Found', 'bird', 'course', 'does', 'clop', 'them', 'question', 'needs', 'winter', 'Pendragon', 'yeah', 'thr
ough', 'trusty', 'speak', 'not', 'The', 'agree', 'where', 'then', 'house', 'strangers', "'em", 'horse', 'do',
'Am', 'carrying', 'that', 'wings', 'ridden', 'using', 'an', 'Saxons', "'ve", 'They', "n't", 'held', 'knights',
'SOLDIER', 'maintain', 'breadth', 'Halt', 'since', 'get', "'m", 'European', 'from', 'use', 'who', "'s", 'at',
'bangin', 'mean', 'is', '.', 'carry', 'warmer', 'two', 'in', 'Mercea', 'grips', 'a', 'defeator', 'other', 'lan
d', 'anyway', 'Arthur', 'Uther', 'We', 'your', 'migrate', 'of', 'matter', '?', 'Please', '2', 'one', 'Will', 'a
sk', 'me', 'Yes', 'African', 'join', 'goes', 'tropical', 'That', 'martin', 'You', 'are', "'re", 'In', 'just',
'Oh', 'may', ',', 'go', 'the', 'plover', 'What', 'England', 'search', 'line', 'velocity', 'my', 'found', 'ounc
e', 'A', 'Court', 'No', 'Where', 'there', 'castle', 'right', 'it', 'times', 'wants', '--', 'fly', "'d", 'sugges
ting', 'south', 'servant', "'", 'coconut', '1', 'Ridden', 'or', 'zone', 'to', 'So', 'weight', 'Wait', 'Who', 'd
orsal', 'grip', 'Listen', 'Pull', '[', 'am', 'back', 'empty', 'sun', 'be', 'Whoa', 'maybe', 'together', 'brin
g', 'got', '#', 'ARTHUR', 'and', 'air-speed', 'they'}
```

Tokenization is fundamental to NLP, and you'll end up using it a lot in text mining and information retrieval projects.

See school budget and other projects where tokenization with other approach and other options.

More regex with re.search()

In [5]:
```
match = re.search("coconuts", scene_one)

print(match.start(), match.end())

# Write a regular expression to search for anything in square brackets: pattern1
pattern1 = r"\[.*\]"

# Use re.search to find the first text in square brackets
print(re.search(pattern1, scene_one))

# Find the script notation at the beginning of the fourth sentence and print it
print(sentences[3])
pattern2 = r"[\w\s]+:" #alphanumeric and whitespace sequence, plus a :
print(re.match(pattern2, sentences[3]))
```

```
580 588
<_sre.SRE_Match object; span=(9, 32), match='[wind] [clop clop clop]'>
ARTHUR: It is I, Arthur, son of Uther Pendragon, from the castle of Camelot.
<_sre.SRE_Match object; span=(0, 7), match='ARTHUR:'>
```

Choosing a tokenizer

Given the following string, which of the below patterns is the best tokenizer? If possible, you want to retain sentence punctuation as separate tokens, but have '#1' remain a single token.

my_string = "SOLDIER #1: Found them? In Mercea? The coconut's tropical!"

Answer: r"(\w+|#\d|\?|!)"

Regex with NLTK tokenization

Build a more complex tokenizer for tweets with hashtags and mentions using nltk and regex. The nltk.tokenize.TweetTokenizer class offers some extra methods and attributes for parsing tweets.

In [6]:
```
tweets = ['This is the best #nlp exercise ive found online! #python',
 '#NLP is super fun! <3 #learning',
 'Thanks @datacamp :) #nlp #python']

from nltk.tokenize import regexp_tokenize
from nltk.tokenize import TweetTokenizer

# Define a regex pattern to find hashtags: pattern1
pattern1 = r"#\w+"

a = regexp_tokenize(tweets[0], pattern1)
print(a)

# Write a pattern that matches both mentions and hashtags
pattern2 = r"([@#]\w+)"

b = regexp_tokenize(tweets[-1], pattern2)
print(b)

# Use the TweetTokenizer to tokenize all tweets into one list
tknzr = TweetTokenizer()
all_tokens = [tknzr.tokenize(t) for t in tweets]
print(all_tokens)
```

```
['#nlp', '#python']
['@datacamp', '#nlp', '#python']
[['This', 'is', 'the', 'best', '#nlp', 'exercise', 'ive', 'found', 'online', '!', '#python'], ['#NLP', 'is', 's
uper', 'fun', '!', '<3', '#learning'], ['Thanks', '@datacamp', ':)', '#nlp', '#python']]
```

Non-ascii tokenization

Tokenization by tokenizing some non-ascii based text, such as German with emoji!

```
In [7]: german_text = 'Wann gehen wir Pizza essen? 🍕 Und fährst du mit Über? 🚕'

        # Tokenize and print all words in german_text.
        all_words = word_tokenize(german_text)
        print(all_words)

        # Tokenize and print only capital words
        capital_words = r"[A-ZÜ]\w+"
        print(regexp_tokenize(german_text, capital_words))

        # Tokenize and print only emoji
        emoji = "['\U0001F300-\U0001F5FF'|'\U0001F600-\U0001F64F'|'\U0001F680-\U0001F6FF'|'\u2600-\u26FF\u2700-\u27BF']"
        print(regexp_tokenize(german_text, emoji))
```

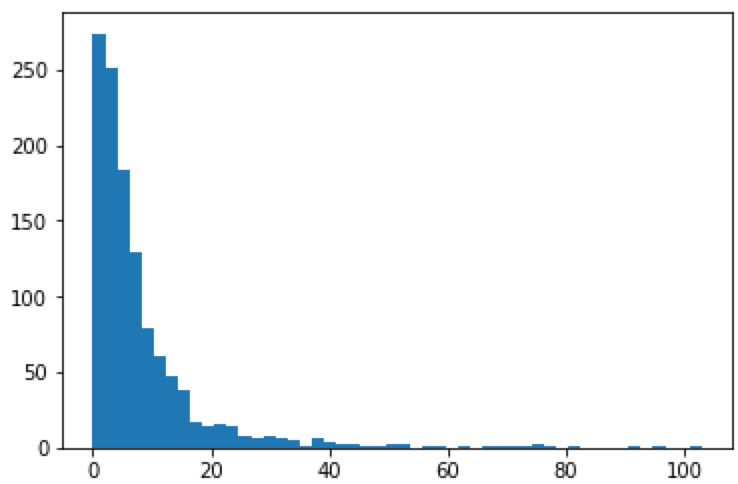
```
['Wann', 'gehen', 'wir', 'Pizza', 'essen', '?', '🍕', 'Und', 'fährst', 'du', 'mit', 'Über', '?', '🚕']
['Wann', 'Pizza', 'Und', 'Über']
['🍕', '🚕']
```

Charting practice

- Find and chart the number of words per line in the script using matplotlib.
- Using list comprehensions to speed up computations.

In [17]:

```
import re
import matplotlib.pyplot as plt
with open('grail.txt', 'r') as myfile:
    holy_grail=myfile.read()

# Split the script into lines: lines
# print(holy_grail)
lines = holy_grail.split('\n') #A natural way to split into lines is to use line breaking '\n'

# Replace all script lines for speaker
pattern = "[A-Z]{2,}(\s)?(#\d)?([A-Z]{2,})?:"
lines = [re.sub(pattern, '', l) for l in lines] #sub, short of substitute
# print(lines)

# Tokenize each line: tokenized_lines
tokenized_lines = [regexp_tokenize(s, "\w+") for s in lines]

# Make a frequency list of lengths: line_num_words
line_num_words = [len(t_line) for t_line in tokenized_lines]
# print(line_num_words)

plt.hist(line_num_words, bins = 50) # Add a bins = 50

plt.show()
```



Simple topic identification

- Using basic NLP models to identify topics from texts based on term frequencies.
- Experiment and compare two simple methods - **bag-of-words and Tf-idf** using NLTK and a new library - Gensim.

Building a Counter with bag-of-words

- Build your bag-of-words counter using a Wikipedia article.

In [20]:
```
with open('wiki_text_debugging.txt', 'r') as myfile:
    article=myfile.read()

from collections import Counter
from nltk.tokenize import word_tokenize

tokens = word_tokenize(article)

lower_tokens = [t.lower() for t in tokens]
# print(lower_tokens)

bow_simple = Counter(lower_tokens)

print(bow_simple.most_common(10))
```

```
[(',', 151), ('the', 150), ('.', 89), ('of', 81), ("''", 68), ('to', 63), ('a', 60), ('in', 44), ('and', 41),
('debugging', 40)]
```

Text preprocessing practice

- Clean up text for better NLP results.
- Remove stop words and non-alphabetic characters, lemmatize, and perform a new bag-of-words on your cleaned text.
- Start with the same tokens you created in the last exercise: lower_tokens.
- **lemmatize and stemming** do similar things.

Stemming and lemmatization

- For grammatical reasons, documents are going to use different forms of a word, such as organize, organizes, and organizing. Additionally, there are families of derivationally related words with similar meanings, such as democracy, democratic, and democratization. In many situations, it seems as if it would be useful for a search for one of these words to return documents that contain another word in the set.
- The goal of both stemming and lemmatization is to reduce inflectional forms and sometimes derivationally related forms of a word to a common base form. For instance:

am, are, is ⇒ be car, cars, car's, cars' ⇒ car The result of this mapping of text will be something like: the boy's cars are different colors ⇒ the boy car be differ color

In [24]:
```
from nltk.corpus import stopwords
english_stops = set(stopwords.words('english'))
# print(english_stops)
#Extra code.

from nltk.stem import WordNetLemmatizer

# Retain alphabetic words: alpha_only
alpha_only = [t for t in lower_tokens if t.isalpha()] # many ways to achieve this.

# Remove all stop words: no_stops
no_stops = [t for t in alpha_only if t not in english_stops] # the key words 'not in'
print(len(no_stops))
wordnet_lemmatizer = WordNetLemmatizer()

# Lemmatize all tokens into a new list: lemmatized
lemmatized = [wordnet_lemmatizer.lemmatize(t) for t in no_stops]
print (len(lemmatized)) #Compare the output of length before and after lemmatization, seems no change here.

# Create the bag-of-words: bow
bow = Counter(lemmatized)
# bag-of-words can also have other forms, not like the tuple pair list below.

print(bow.most_common(10))
```

```
1257
1257
[('debugging', 40), ('system', 25), ('software', 16), ('bug', 16), ('problem', 15), ('tool', 15), ('computer', 14), ('process', 13), ('term', 13), ('used', 12)]
```

Creating and querying a corpus with gensim

From Wikipedia:

- Gensim is an open-source library for unsupervised topic modeling and natural language processing, using modern statistical machine learning.
- Gensim includes streamed parallelized implementations of fastText, word2vec and doc2vec algorithms, as well as latent semantic analysis (LSA, LSI, SVD), non-negative matrix factorization (NMF), latent Dirichlet allocation (LDA), tf-idf and random projections.

In this exercise:

- Create your first gensim dictionary and corpus!
- Use these data structures to investigate word trends and potential interesting topics in a document, which were preprocessed by lowercasing all words, tokenizing them, and removing stop words and punctuation, and were then stored in a list of document tokens called articles.
- Do some light preprocessing and then generate the gensim dictionary and corpus.

Definition of corpus: a collection of written texts, especially the entire works of a particular author or a body of writing on a particular subject.

Prepare data for next cell. Not all the articles are read-in yet. If necessary, I should read in all the 12 articles in the folder.

In [1]:
```python
from collections import Counter
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
english_stops = set(stopwords.words('english'))

with open('wiki_text_bug.txt', 'r', encoding="utf8") as myfile:
    article=myfile.read() #If without encoding="utf8", then I will have problem in reading.
tokens = word_tokenize(article)
lower_tokens = [t.lower() for t in tokens]
alpha_only = [t for t in lower_tokens if t.isalpha()]
no_stops = [t for t in alpha_only if t not in english_stops]
wordnet_lemmatizer = WordNetLemmatizer()
lemmatized0= [wordnet_lemmatizer.lemmatize(t) for t in no_stops]

with open('wiki_text_computer.txt', 'r', encoding="utf8") as myfile:
    article=myfile.read() #If without encoding="utf8", then I will have problem in reading.
tokens = word_tokenize(article)
lower_tokens = [t.lower() for t in tokens]
alpha_only = [t for t in lower_tokens if t.isalpha()]
no_stops = [t for t in alpha_only if t not in english_stops]
wordnet_lemmatizer = WordNetLemmatizer()
lemmatized1= [wordnet_lemmatizer.lemmatize(t) for t in no_stops]

articles = [lemmatized0,lemmatized1]
print(len(articles))
#It should be many. I only read in two of them.
```

```
2
```

In [9]:
```
import gensim
from gensim import corpora
from gensim.corpora.dictionary import Dictionary

# Create a Dictionary from the articles: dictionary
dictionary = Dictionary(articles)
print(type(dictionary) )
#This is the real dictionary but not the dictionary type in python. It is just a collection of
#unique tokens.
print(dictionary)

# Select the id for "computer": computer_id
computer_id = dictionary.token2id.get("computer")
print(computer_id)

# Use computer_id with the dictionary to print the word
print(dictionary.get(computer_id))

# Create a MmCorpus: corpus
corpus = [dictionary.doc2bow(article) for article in articles] #doc to bag of words

print(len(corpus))
print('--------------------------')
print(corpus[1][:10])
```

```
<class 'gensim.corpora.dictionary.Dictionary'>
Dictionary(2829 unique tokens: ['abandon', 'able', 'abstract', 'abuse', 'access']...)
228
computer
2
--------------------------
[(1, 9), (2, 3), (4, 6), (8, 5), (9, 1), (11, 1), (12, 1), (13, 1), (15, 2), (17, 1)]
```

In different context, corpus has different structure. Be familiar with the corpus in previous example.

Gensim bag-of-words

- Use the new gensim corpus and dictionary to see the most common terms per document and across all documents.
- Python defaultdict and itertools are used to help with the creation of intermediate data structures for analysis.

In [48]:

```python
from collections import defaultdict
import itertools
#Extra code

# Save the 2nd document: doc
doc = corpus[1]

# Sort the doc for frequency: bow_doc
bow_doc = sorted(doc, key=lambda w: w[1], reverse=True)
# sort according to specific element in a tuple list.

# Print the top 5 words of the document alongside the count
for word_id, word_count in bow_doc[:5]:
    print(dictionary.get(word_id), word_count)

print('_________________________')

# Create the defaultdict: total_word_count
total_word_count = defaultdict(int)
# This is like total_word_count = {} except we also specify the type of the dictionary.

#see collections about the itertools.chain.from_iterable()
for word_id, word_count in itertools.chain.from_iterable(corpus):
    total_word_count[word_id] += word_count


# print(corpus[0]). Here there are two elements in corpus. In the original document, there are many.
# So there are many same word_id across many elements of the corpus.

# Create a sorted list from the defaultdict: sorted_word_count
sorted_word_count = sorted(total_word_count.items(), key=lambda w: w[1], reverse=True)

# Print the top 5 words across all documents alongside the count
for word_id, word_count in sorted_word_count[:5]:
    print(dictionary.get(word_id), word_count)
```

```
<class 'list'>
computer 349
program 79
machine 76
first 60
```

```
cite 59
```

```
computer 390
bug 134
program 109
software 90
machine 81
```

Here corpus is a nested list, so we itertools.chain.from_iterable() to handle.

What is tf-idf?

See details in other notes about tf-idf.

In [49]:

```python
from gensim.models.tfidfmodel import TfidfModel

tfidf = TfidfModel(corpus)

tfidf_weights = tfidf[doc] #doc = corpus[1]

print(tfidf_weights[:5])

# Sort the weights from highest to lowest: sorted_tfidf_weights
sorted_tfidf_weights = sorted(tfidf_weights, key=lambda w: w[1], reverse=True)

# Print the top 5 weighted words
for term_id, weight in sorted_tfidf_weights[:5]:
    print(dictionary.get(term_id), weight)
```

```
[(1313, 0.05833941067182285), (1314, 0.008334201524546121), (1315, 0.008334201524546121), (1316, 0.008334201524
546121), (1317, 0.033336806098184485)]
circuit 0.21668923963819914
modern 0.20835503811365302
cpu 0.18335243354001465
architecture 0.17501823201546854
stored 0.1583498289663763
```

Named-entity recognition (NER)

- How to identify the who, what and where of your texts using pre-trained models on English and non-English text.
- Learn how to use some new libraries - polyglot and spaCy - to add to your NLP toolbox.

NER with NLTK

- Use nltk to find the named entities in this article.
- What might the article be about, given the names found?

In [2]:

```
#Without encoding="utf8", the following sentence does not work.
with open('uber_apple.txt', 'r', encoding="utf8") as myfile:
    article=myfile.read()
#extra code

import nltk
sentences = nltk.sent_tokenize(article)

# Tokenize each sentence into words: token_sentences
token_sentences = [nltk.word_tokenize(sent) for sent in sentences]
print(token_sentences[0:2])
print('--------------------------')

# Tag each tokenized sentence into parts of speech: pos_sentences
pos_sentences = [nltk.pos_tag(sent) for sent in token_sentences]
print(pos_sentences[0:2])
print('--------------------------')

# Create the named entity chunks: chunked_sentences
chunked_sentences = nltk.ne_chunk_sents(pos_sentences, binary=True)
# print(len(list(chunked_sentences)))
print(next(chunked_sentences))
print(next(chunked_sentences))
print('--------------------------')

# Test for stems of the tree with 'NE' tags
for sent in chunked_sentences:
    for chunk in sent:
        if hasattr(chunk, "label") and chunk.label() == "NE":
            print(chunk)
```

```
[['\ufeffThe', 'taxi-hailing', 'company', 'Uber', 'brings', 'into', 'very', 'sharp', 'focus', 'the', 'questio
n', 'of', 'whether', 'corporations', 'can', 'be', 'said', 'to', 'have', 'a', 'moral', 'character', '.'], ['If',
'any', 'human', 'being', 'were', 'to', 'behave', 'with', 'the', 'single-minded', 'and', 'ruthless', 'greed', 'o
f', 'the', 'company', ',', 'we', 'would', 'consider', 'them', 'sociopathic', '.']]
--------------------------
[[('\ufeffThe', 'JJ'), ('taxi-hailing', 'JJ'), ('company', 'NN'), ('Uber', 'NNP'), ('brings', 'VBZ'), ('into',
'IN'), ('very', 'RB'), ('sharp', 'JJ'), ('focus', 'VB'), ('the', 'DT'), ('question', 'NN'), ('of', 'IN'), ('whe
ther', 'IN'), ('corporations', 'NNS'), ('can', 'MD'), ('be', 'VB'), ('said', 'VBD'), ('to', 'TO'), ('have', 'V
B'), ('a', 'DT'), ('moral', 'JJ'), ('character', 'NN'), ('.', '.')], [('If', 'IN'), ('any', 'DT'), ('human', 'J
J'), ('being', 'VBG'), ('were', 'VBD'), ('to', 'TO'), ('behave', 'VB'), ('with', 'IN'), ('the', 'DT'), ('single
```

```
-minded', 'JJ'), ('and', 'CC'), ('ruthless', 'JJ'), ('greed', 'NN'), ('of', 'IN'), ('the', 'DT'), ('company',
'NN'), (',', ','), ('we', 'PRP'), ('would', 'MD'), ('consider', 'VB'), ('them', 'PRP'), ('sociopathic', 'JJ'),
('.', '.')]]
--------------------------
(S
  The/JJ
  taxi-hailing/JJ
  company/NN
  Uber/NNP
  brings/VBZ
  into/IN
  very/RB
  sharp/JJ
  focus/VB
  the/DT
  question/NN
  of/IN
  whether/IN
  corporations/NNS
  can/MD
  be/VB
  said/VBD
  to/TO
  have/VB
  a/DT
  moral/JJ
  character/NN
  ./.)
(S
  If/IN
  any/DT
  human/JJ
  being/VBG
  were/VBD
  to/TO
  behave/VB
  with/IN
  the/DT
  single-minded/JJ
  and/CC
  ruthless/JJ
  greed/NN
  of/IN
```

```
  the/DT
  company/NN
  ,/,
  we/PRP
  would/MD
  consider/VB
  them/PRP
  sociopathic/JJ
  ./.)
--------------------------
(NE Uber/NNP)
(NE Beyond/NN)
(NE Apple/NNP)
(NE Uber/NNP)
(NE Uber/NNP)
(NE Travis/NNP Kalanick/NNP)
(NE Tim/NNP Cook/NNP)
(NE Apple/NNP)
(NE Silicon/NNP Valley/NNP)
(NE CEO/NNP)
(NE Yahoo/NNP)
(NE Marissa/NNP Mayer/NNP)
```

Those acronym are used in Penn Treebank Project. Here is the complete list of tags with their meanings:

CC - Coordinating conjunction
CD - Cardinal number
DT - Determiner
EX - Existential there
FW -Foreign word
IN - Preposition or subordinating conjunction
JJ - Adjective
JJR - Adjective, comparative
JJS - Adjective, superlative
LS - List item marker
MD - Modal
NN - Noun, singular or mass
NNS - Noun, plural
NNP - Proper noun, singular
NNPS - Proper noun, plural

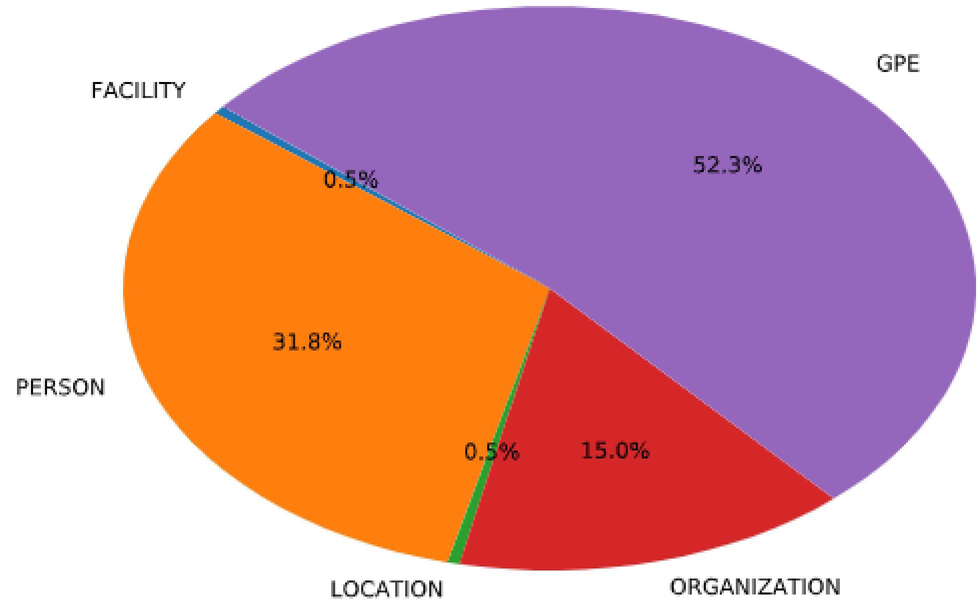
PDT - Predeterminer
POS - Possessive ending
PRP - Personal pronoun
PRP $ - Possessive pronoun
RB - Adverb
RBR - Adverb, comparative
RBS - Adverb, superlative
RP - Particle
SYM - Symbol
TO - to
UH - Interjection
VB - Verb, base form
VBD - Verb, past tense
VBG - Verb, gerund or present participle
VBN - Verb, past participle
VBP - Verb, non-3rd person singular present
VBZ - Verb, 3rd person singular present
WDT - Wh-determiner
WP - Wh-pronoun
WP $ - Possessive wh-pronoun
WRB - Wh-adverb

Charting practice

- Use some extracted named entities and their groupings from a series of newspaper articles to chart the diversity of named entity types in the articles.
- Use a defaultdict called ner_categories, with keys representing every named entity group type, and values to count the number of each different named entity type. There is a chunked sentence list called chunked_sentences similar to the last exercise, but this time with non-binary category names.
- Use hasattr() to determine if each chunk has a 'label' and then simply use the chunk's .label() method as the dictionary key.

In [5]:

```python
from collections import defaultdict
import itertools
import matplotlib.pyplot as plt
# Create the defaultdict: ner_categories
ner_categories = defaultdict(int)

# Create the nested for loop
for sent in chunked_sentences:
    for chunk in sent:
        if hasattr(chunk, 'label'):
            ner_categories[chunk.label()] += 1

# Create a list from the dictionary keys for the chart labels: labels
labels = list(ner_categories.keys())

# Create a list of the values: values
values = [ner_categories.get(l) for l in labels]

# Create the pie chart
plt.pie(values, labels=labels, autopct='%1.1f%%', startangle=140)

# Display the chart. Data is not correct.
plt.show()
```



Stanford library with NLTK

When using the Stanford library with NLTK, what is needed to get started?

1. normal installation of NLTK.
2. An installation of the Stanford Java Library.
3. Both NLTK and an installation of the Stanford Java Library.
4. NLTK, the Stanford Java Libraries and some environment variables to help with integration.

Answer: 4.

Comparing NLTK with spaCy NER

- Using the same text you used in the first exercise of this chapter, see the results using spaCy's NER annotator. How will they compare?

- To minimize execution times, you'll be asked to specify the keyword arguments tagger=False, parser=False, matcher=False when loading the spaCy model, because only the entity is cared about.

In [ ]:
```python
#Without encoding="utf8", the following sentence does not work.
with open('uber_apple.txt', 'r', encoding="utf8") as myfile:
    article=myfile.read()
#extra code

import spacy

# Instantiate the English model: nlp
nlp = spacy.load('en', tagger=False, parser=False, matcher=False)

# Create a new document: doc
doc = nlp(article)

# Print all of the found entities and their labels
for ent in doc.ents:
    print(ent.label_, ent.text)
```

Having problems when running !pip install spacy. So spacy is not installed. The following is the results from Datacamp.

```
ORG Uber
ORG Uber
ORG Apple
ORG Uber
ORG Uber
PERSON Travis Kalanick
ORG Uber
PERSON Tim Cook
ORG Apple
CARDINAL Millions
ORG Uber
GPE drivers’
LOC Silicon Valley’s
ORG Yahoo
PERSON Marissa Mayer
MONEY $186m
```

spaCy NER Categories Which are the extra categories that spacy uses compared to nltk in its named-entity recognition?
Answer:
NORP, CARDINAL, MONEY, WORKOFART, LANGUAGE, EVENT

French NER with polyglot I

- Use the polyglot library to identify French entities. The library functions slightly differently than spacy.
- Having problems when running !pip install polyglot. The following is the results from Datacamp.

In [ ]:
```
with open('french.txt', 'r', encoding="utf8") as myfile:
    article=myfile.read()

from polyglot.text import Text

# Create a new text object using Polyglot's Text class: txt
txt = Text(article)

# Print each of the entities found
for ent in txt.entities:
    print(ent)

print(type(ent))
```

['Charles', 'Cuvelliez'] ['Charles', 'Cuvelliez'] ['Bruxelles'] ['l'IA'] ['Julien', 'Maldonato'] ['Deloitte'] ['Ethiquement'] ['l'IA'] ['.']

French NER with polyglot II

- Complete the work you began in the previous exercise.
- Use a list comprehension to create a list of tuples, in which the first element is the entity tag, and the second element is the full string of the entity text.

In [ ]:
```
# Create the list of tuples: entities
entities = [(ent.tag, ' '.join(ent)) for ent in txt.entities]

print(entities)
```

[('I-PER', 'Charles Cuvelliez'), ('I-PER', 'Charles Cuvelliez'), ('I-ORG', 'Bruxelles'), ('I-PER', 'l'IA'), ('I-PER', 'Julien Maldonato'), ('I-ORG', 'Deloitte'), ('I-PER', 'Ethiquement'), ('I-LOC', 'l'IA'), ('I-PER', '.')]

Spanish NER with polyglot

- Use polyglot with some Spanish annotation. This article is not written by a newspaper, so it is an example of a more blog-like text. How do you think that might compare when finding entities?

- Determine how many of the entities contain the words "Márquez" or "Gabo" - these refer to the same person in different ways!

In [ ]:
```
count = 0

# Iterate over all the entities
for ent in txt.entities:
    # Check whether the entity contains 'Márquez' or 'Gabo'
    if "Márquez" in ent or "Gabo" in ent:
        # Increment count
        count += 1

print(count)

percentage = count / len(txt.entities)
print(percentage)
```

Building a "fake news" classifier

See count vector and td-idf vector in the notes "A Comprehensive Guide to Understand and Implement Text Classification in Python.pdf" in the same folder.

CountVectorizer for text classification

- Use pandas alongside scikit-learn to create a SPARSE text vectorizer to train and test a simple supervised model.
- Set up a CountVectorizer and investigate some of its features.

In [15]:

```python
import pandas as pd
df = pd.read_csv('fake_or_real_news.csv')
#If I use above, then there will be problems of many comma appearance (see the warning in output).
#So I use the following instead. Figure out the reason in the future.
df = df.loc[:,['Unnamed: 0','title','text','label']].astype('U')
# print(df.head())

df.info()
print('-----------------------')
a = df.loc[0,'title']
print(a)
print('-----------------------')
print(type(a))
print('-----------------------')
#extra code above

from sklearn.feature_extraction.text import CountVectorizer
from sklearn.model_selection import train_test_split

y = df.label

X_train, X_test, y_train, y_test = train_test_split(df['text'], y, test_size=0.33, random_state=53)

count_vectorizer = CountVectorizer(stop_words='english')

count_train = count_vectorizer.fit_transform(X_train.values)

count_test = count_vectorizer.transform(X_test.values)

# Print the first 10 features of the count_vectorizer
print(count_vectorizer.get_feature_names()[:10])
```

```
C:\Users\ljyan\Anaconda3\lib\site-packages\IPython\core\interactiveshell.py:2728: DtypeWarning: Columns (24,25,
26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,
63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,
100,101,102,103,104,105,106,107,108,109,110,111,112,113,114,115,116,117,118,119,120,121,122,123,124,125,126,12
7,128,129,130,131,132,133,134,135,136,137,138,139,140) have mixed types. Specify dtype option on import or set
low_memory=False.
  interactivity=interactivity, compiler=compiler, result=result)

<class 'pandas.core.frame.DataFrame'>
```

```
RangeIndex: 7795 entries, 0 to 7794
Data columns (total 4 columns):
Unnamed: 0    7795 non-null object
title         7795 non-null object
text          7795 non-null object
label         7795 non-null object
dtypes: object(4)
memory usage: 243.7+ KB
-----------------------
You Can Smell Hillary’s Fear
-----------------------
<class 'str'>
-----------------------
['00', '000', '0000', '000000031', '00000031', '0001', '0002', '000ft', '000x', '001']
```

I added astype('U') in two places in the cell above. Otherwise it will not work. Figure out why

TfidfVectorizer for text classification

- Similar to the sparse CountVectorizer created in the previous exercise, here we work on creating tf-idf vectors of the documents. Set up a TfidfVectorizer and investigate some of its features.

In [14]:
```python
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf_vectorizer = TfidfVectorizer(stop_words='english', max_df=0.7)

tfidf_train = tfidf_vectorizer.fit_transform(X_train)

tfidf_test = tfidf_vectorizer.transform(X_test)

# Print the first 10 features
print(tfidf_vectorizer.get_feature_names()[:10])

print('-------------------')
# Print the first 5 vectors of the tfidf training data
print(tfidf_train.A[:5])
```

```
['00', '000', '0000', '000000031', '00000031', '0001', '0002', '000ft', '000x', '001']
-------------------
[[0.         0.06265488 0.         ... 0.         0.         0.        ]
 [0.         0.         0.         ... 0.         0.         0.        ]
 [0.         0.         0.         ... 0.         0.         0.        ]
 [0.         0.         0.         ... 0.         0.         0.        ]
 [0.         0.         0.         ... 0.         0.         0.        ]]
```

Inspecting the vectors

- To get a better idea of how the vectors work, investigate them by converting them into pandas DataFrames.

The output results are different from that of Datacamp. Here we have weird stuff below.

In [16]:
```python
count_df = pd.DataFrame(count_train.A, columns=count_vectorizer.get_feature_names())

tfidf_df = pd.DataFrame(tfidf_train.A, columns=tfidf_vectorizer.get_feature_names())

print(count_df.head())

print(tfidf_df.head())

difference = set(count_df.columns) - set(tfidf_df.columns)
print(difference)

print(count_df.equals(tfidf_df))
```

```
   00  000  0000  000000031  00000031  0001  0002  000ft  000x  001  ...  שתי  \
0   0    7     0          0         0     0     0      0     0    0  ...    0
1   0    0     0          0         0     0     0      0     0    0  ...    0
2   0    0     0          0         0     0     0      0     0    0  ...    0
3   0    0     0          0         0     0     0      0     0    0  ...    0
4   0    0     0          0         0     0     0      0     0    0  ...    0

   عربي  תתרכז  תעודת  תנותק  תיירות  תחל  תוצאה  תאמצנה  ยงade
0       0      0    0       0      0      0      0      0      0
1       0      0    0       0      0      0      0      0      0
2       0      0    0       0      0      0      0      0      0
3       0      0    0       0      0      0      0      0      0
4       0      0    0       0      0      0      0      0      0

[5 rows x 56476 columns]
    00       000  0000  000000031  00000031  0001  0002  000ft  000x  001  \
0  0.0  0.062655   0.0        0.0       0.0   0.0   0.0    0.0   0.0  0.0
1  0.0  0.000000   0.0        0.0       0.0   0.0   0.0    0.0   0.0  0.0
2  0.0  0.000000   0.0        0.0       0.0   0.0   0.0    0.0   0.0  0.0
3  0.0  0.000000   0.0        0.0       0.0   0.0   0.0    0.0   0.0  0.0
4  0.0  0.000000   0.0        0.0       0.0   0.0   0.0    0.0   0.0  0.0

   ...  عربي  תתרכז  תעודת  תנותק  תיירות  תחל  תוצאה  תאמצנה  שתי  ยงade
0  ...  0.0     0.0    0.0  0.0     0.0    0.0    0.0    0.0  0.0    0.0
1  ...  0.0     0.0    0.0  0.0     0.0    0.0    0.0    0.0  0.0    0.0
2  ...  0.0     0.0    0.0  0.0     0.0    0.0    0.0    0.0  0.0    0.0
3  ...  0.0     0.0    0.0  0.0     0.0    0.0    0.0    0.0  0.0    0.0
4  ...  0.0     0.0    0.0  0.0     0.0    0.0    0.0    0.0  0.0    0.0
```

```
[5 rows x 56476 columns]
set()
False
```

Training and testing the "fake news" model with CountVectorizer

- Train the "fake news" model using the features identified and extracted. In this first exercise, train and test a Naive Bayes model using the CountVectorizer data.

In [17]:

```
from sklearn.naive_bayes import MultinomialNB
from sklearn import metrics

# Instantiate a Multinomial Naive Bayes classifier: nb_classifier
nb_classifier = MultinomialNB()

nb_classifier.fit(count_train, y_train)

pred = nb_classifier.predict(count_test)

score = metrics.accuracy_score(y_test, pred)
print(score)

# Calculate the confusion matrix: cm
cm = metrics.confusion_matrix(y_test, pred, labels=['FAKE', 'REAL'])
print(cm)
```

```
0.845705402254178
[[906 145]
 [ 79 956]]
```

The output in Datacamp is 0.89...

Training and testing the "fake news" model with TfidfVectorizer

- After we have evaluated the model using the CountVectorizer, do the same using the TfidfVectorizer with a Naive Bayes model.
- At least for this example, the result is not as good as earlier one using CountVectorizer. However, we can improve the score by tuning hyperparameters.

In [18]:
```
nb_classifier = MultinomialNB()
nb_classifier.fit(tfidf_train, y_train)
pred = nb_classifier.predict(tfidf_test)
score = metrics.accuracy_score(y_test, pred)
print(score)
cm = metrics.confusion_matrix(y_test, pred, labels=['FAKE', 'REAL'])
print(cm)
```

```
0.7924601632335795
[[ 714  338]
 [  21 1017]]
```

Improving your model

What are possible next steps you could take to improve the model?

1. Tweaking alpha levels.
2. Trying a new classification model.
3. Training on a larger dataset.
4. Improving text preprocessing.
5. All of the above.

Answer 5.

Your job in this exercise is to test a few different alpha levels using the Tfidf vectors to determine if there is a better performing combination.

The training and test sets have been created, and tfidf_vectorizer, tfidf_train, and tfidf_test have been computed.

In [19]:
```
import numpy as np
alphas = np.arange(0.00001, 1, .1)

def train_and_predict(alpha):
    nb_classifier = MultinomialNB(alpha=alpha)
    nb_classifier.fit(tfidf_train, y_train)
    pred = nb_classifier.predict(tfidf_test)
    score = metrics.accuracy_score(y_test, pred)
    return score

for alpha in alphas:
    print('Alpha: ', alpha)
    print('Score: ', train_and_predict(alpha))
    print()
```

```
Alpha:  1e-05
Score:  0.8561989895064127

Alpha:  0.10001
Score:  0.8534784298484259

Alpha:  0.20001000000000002
Score:  0.8425961912164788

Alpha:  0.30001000000000005
Score:  0.8363777691410804

Alpha:  0.40001000000000003
Score:  0.8282160901671201

Alpha:  0.50001
Score:  0.8212203653322969

Alpha:  0.60001
Score:  0.8150019432568986

Alpha:  0.70001
Score:  0.8080062184220754

Alpha:  0.80001
Score:  0.8010104935872522
```

```
Alpha:  0.90001
Score:  0.7990672366886903
```

Inspecting your model

- Investigate what it has learned. You can map the important vector weights back to actual words using some simple inspection techniques.

In [30]:
```python
# Get the class labels: class_labels
class_labels = nb_classifier.classes_

feature_names = tfidf_vectorizer.get_feature_names()
print(len(feature_names))
print(len(nb_classifier.coef_[0]))
# print(nb_classifier.coef_[0])

# Zip the feature names together with the coefficient array and sort by weights: feat_with_weights
feat_with_weights = sorted(zip(nb_classifier.coef_[0], feature_names))

# Print the first class label and the top 20 feat_with_weights entries
print(class_labels[0], feat_with_weights[:20])

# Print the second class label and the bottom 20 feat_with_weights entries
print(class_labels[1], feat_with_weights[-20:])
```

```
56476
56476
 [(-10.941588754609102, '00'), (-10.941588754609102, '000'), (-10.941588754609102, '0000'), (-10.9415887546091
02, '000000031'), (-10.941588754609102, '00000031'), (-10.941588754609102, '0001'), (-10.941588754609102, '000
2'), (-10.941588754609102, '000ft'), (-10.941588754609102, '000x'), (-10.941588754609102, '001'), (-10.94158875
4609102, '003'), (-10.941588754609102, '004'), (-10.941588754609102, '005'), (-10.941588754609102, '00684'), (-
10.941588754609102, '006s'), (-10.941588754609102, '007'), (-10.941588754609102, '007s'), (-10.941588754609102,
'008'), (-10.941588754609102, '008s'), (-10.941588754609102, '009')]
 1948. [57]  [(-10.941588754609102, '1-) ,('שכל' ,10.941588754609102-) ,('שיתעקש' ,10.941588754609102-) ,('שינוי
0.941588754609102ש' ,10.941588754609102-) ,('שלו' ,10.941588754609102-) ,('של' ,10.941588754609102-) ,('שכמוני' ,
10.9415887546-) ,('שתי' ,10.941588754609102-) ,('שעת' ,10.941588754609102-) ,('שני' ,10.941588754609102-) ,('נדרש
09102-) ,('תיירות' ,10.941588754609102-) ,('תחל' ,10.941588754609102-) ,('תוצאה' ,10.941588754609102-) ,('תאמצנה' ,
10.94158875460910-) ,('תתרכז' ,10.941588754609102-) ,('תעודת' ,10.941588754609102-) ,('תנותק' ,10.941588754609102
210.941588754609102-) ,('عربي' ,, 'ยงade'), (-10.248441574049156, 'nan')]
```

output from Datacamp:

FAKE [(-12.641778440826338, '0000'), (-12.641778440826338, '000035'), (-12.641778440826338, '0001'), (-12.641778440826338, '0001pt'), (-12.641778440826338, '000km'), (-12.641778440826338, '0011'), (-12.641778440826338, '006s'), (-12.641778440826338, '007'), (-12.641778440826338, '007s'), (-12.641778440826338, '008s'), (-12.641778440826338, '0099'), (-12.641778440826338, '00am'), (-12.641778440826338, '00p'), (-12.641778440826338, '00pm'), (-12.641778440826338, '014'), (-12.641778440826338, '015'), (-12.641778440826338, '018'), (-12.641778440826338, '01am'), (-12.641778440826338, '020'), (-12.641778440826338, '023')] REAL [(-6.790929954967984, 'states'), (-6.765360557845786, 'rubio'), (-6.751044290367751, 'voters'), (-6.701050756752027, 'house'),

(-6.695547793099875, 'republicans'), (-6.6701912490429685, 'bush'), (-6.661945235816139, 'percent'), (-6.589623788689862, 'people'), (-6.559670340096453, 'new'), (-6.489892292073901, 'party'), (-6.452319082422527, 'cruz'), (-6.452076515575875, 'state'), (-6.397696648238072, 'republican'), (-6.376343060363355, 'campaign'), (-6.324397735392007, 'president'), (-6.2546017970213645, 'sanders'), (-6.144621899738043, 'obama'), (-5.756817248152807, 'clinton'), (-5.596085785733112, 'said'), (-5.357523914504495, 'trump')]

https://towardsdatascience.com/multi-class-text-classification-model-comparison-and-selection-5eb066197568 (https://towardsdatascience.com/multi-class-text-classification-model-comparison-and-selection-5eb066197568)