A Quick Guide to Tokenization, Lemmatization, Stop Words, and Phrase Matching using spaCy | NLP | Part 2

“**spaCy”** is designed specifically for **production use**. It helps you build applications that process and “understand” large volumes of text. It can be used to build **information extraction** or **natural language understanding** systems, or to pre-process text for **deep learning**. In this article you will learn about Tokenization, Lemmatization, Stop Words and Phrase Matching operations using spaCy.

you can download the Jupyter Notebook for this complete exercise using the below link.
Text Pre Processing Operations using spaCyDownload

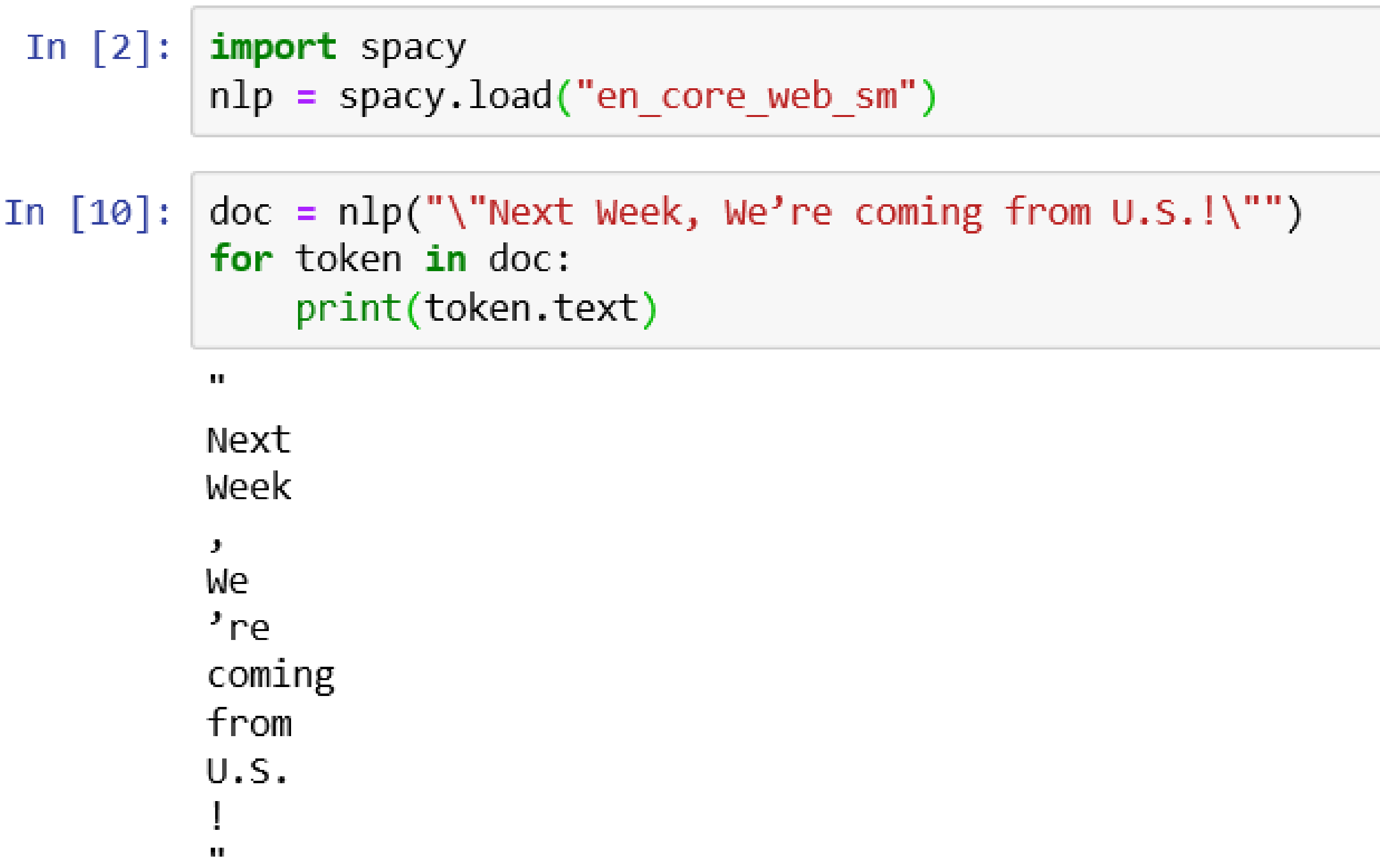
This is the article 2 in the **spaCy Series**. In my last article I have explained about spaCy Installation and basic operations. If you are new to this, I would suggest to start from article 1 for better understanding.

Tokenization

Tokenization is the first step in text processing task. Tokenization is not only breaking the text into components, pieces like words, punctuation etc known as tokens. However it is more than that. spaCy do the intelligent Tokenizer which internally identify whether a “.” is a punctuation and separate it into token or it is part of abbreviation like “U.S.” and do not separate it.

spaCy applies rules specific to the Language type. Let’s understand with an example.

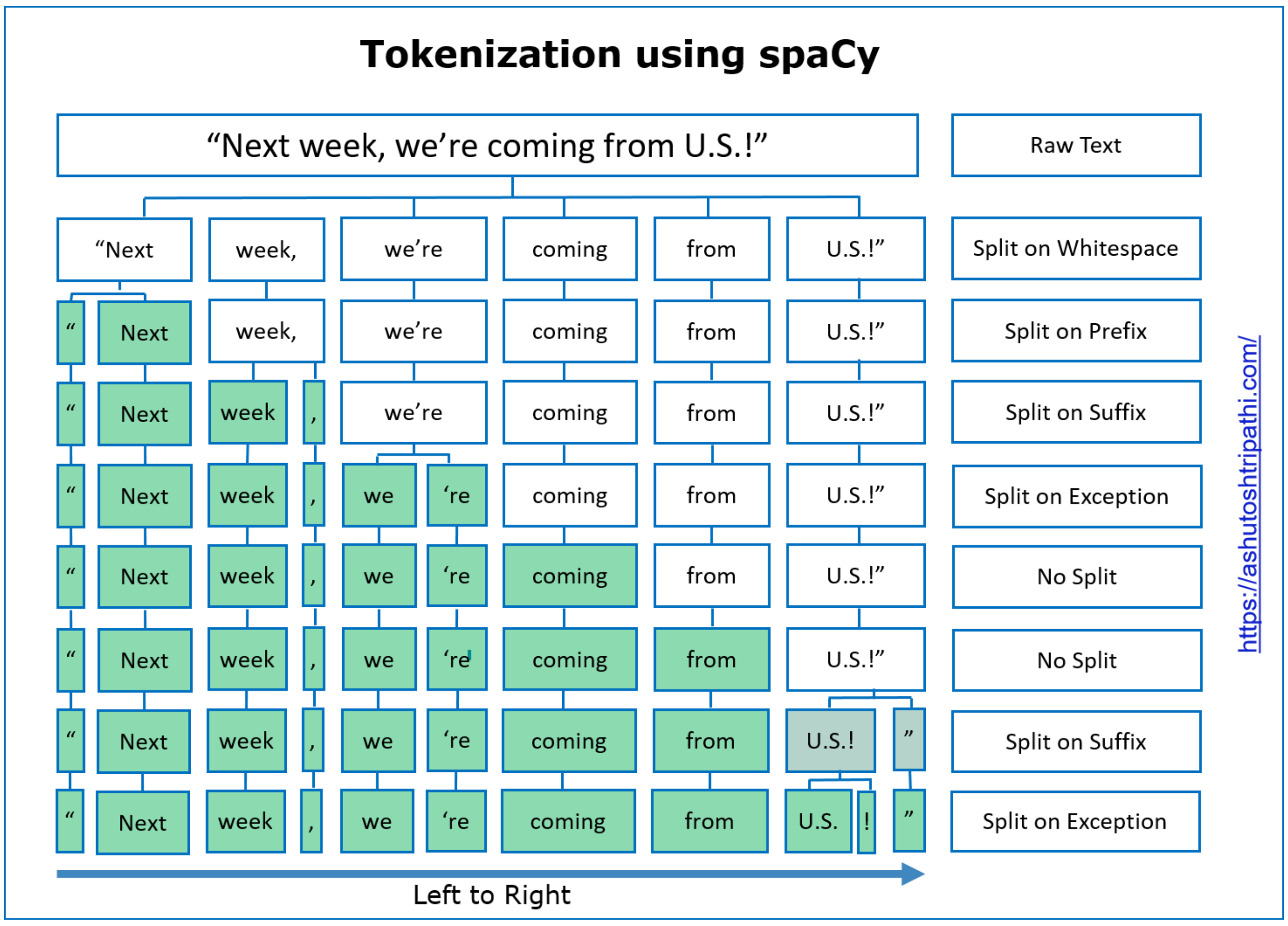
```
import spacy
nlp = spacy.load("en_core_web_sm")
doc = nlp("\"Next Week, We’re coming from U.S.!\"")
    for token in doc:
    print(token.text)
```



spaCy Tokens

- **spaCy** start spliting first based on the white space available in the raw text.
- Then it processes the text from left to right and on each item (splitted based on white space) it performs the following two checks:
    - **Exception Rule Check:** Punctuation available in “U.S.” should not be treated as further tokens. It should remain one. However we’re should be splitted into “we” and ” ‘re “
    - **Prefix, Suffix and Infix check:** Punctuation like commas, periods, hyphens or quotes to be treated as tokens and separated out.

If there’s a match, the rule is applied and the Tokenizer continues its loop, starting with the newly split sub strings. This way, spaCy can split complex, nested tokens like combinations of abbreviations and multiple punctuation marks.



Tokenization using spaCy

- **Prefix**: Look for Character(s) at the beginning ‣ `$ ( “ ¿`
- **Suffix**: Look for Character(s) at the end ‣ `mm ) , . ! ”` mm is an example of unit
- **Infix**: Look for Character(s) in between ‣ `- -- / ...`
- **Exception**: Special-case rule to split a string into several tokens or prevent a token from being split when punctuation rules are applied ‣ `St. N.Y.`

Notice that tokens are pieces of the original text. Tokens are the basic building blocks of a Doc object – everything that helps us understand the meaning of the text is derived from tokens and their relationship to one another.

Prefixes, Suffixes and Infixes as Tokens

- spaCy will separate punctuation that does *not* form an integral part of a word.
- Quotation marks, commas, and punctuation at the end of a sentence will be assigned their own token.
- However, punctuation that exists as part of an email address, website or numerical value will be kept as part of the token.

```
doc2 = nlp(u"We're here to guide you! Send your query, email contact@enetwork.ai or visit us at http://www.enetwork.ai!")
for t in doc2:
    print(t)
```

```
In [12]: doc2 = nlp(u"We're here to guide you! Send your query, \
email contact@enetwork.ai or visit us at http://www.enetwork.ai!")

for t in doc2:
    print(t)
```

```
We
're
here
to
guide
you
!
Send
your
query
,
email
contact@enetwork.ai
or
visit
us
at
http://www.enetwork.ai
!
```

Tokens

Note that the exclamation points, comma are assigned their own tokens. However point, colon present in email address and website URL are not isolated. Hence both the email address and website are preserved.

```
doc3 = nlp(u'A 40km U.S. cab ride costs $100.60')
for t in doc3:
    print(t)
```

```
In [13]: doc3 = nlp(u'A 40km U.S. cab ride costs $100.60')
for t in doc3:
    print(t)
```

```
A
40
km
U.S.
cab
ride
costs
$
100.60
```

Tokens

Here the distance unit and dollar sign are assigned their own tokens, however the dollar amount is preserved, point in amount is not isolated.

Exceptions in Token generation

Punctuation that exists as part of a known abbreviation will be kept as part of the token.

```
doc4 = nlp(u"Let's visit the St. Louis in the U.S. next year.")
for t in doc4:
    print(t)
```

```
In [14]: doc4 = nlp(u"Let's visit the St. Louis in the U.S. next year.")
for t in doc4:
    print(t)
```

```
Let
's
visit
the
St.
Louis
in
the
U.S.
next
year
.
```

Tokens

Here the abbreviations for "Saint" and "United States" are both preserved. Mean point next to St. is not separated as token. Same in U.S.

Counting Tokens

Using len() function, you can count the number of tokens in a document.

len(doc4)

```
In [16]: len(doc4)
Out[16]: 12
```

Counting Vocab Entries

`Vocab` objects contain a full library of items!

```
In [17]: len(doc4.vocab)
Out[17]: 57852
```

```
In [18]: len(doc3.vocab)
Out[18]: 57852
```

```
In [19]: len(doc2.vocab)
Out[19]: 57852
```

```
In [20]: len(doc.vocab)
Out[20]: 57852
```

vocab counting

see all doc obj are created from english language model, which we have loaded in the begining using

nlp = spacy.load("en_core_web_sm")

Hence vocab len will be same.

Indexing and Slicing in Token

- `Doc` objects can be thought of as lists of `token` objects.
- As such, individual tokens can be retrieved by index position.
- spans of tokens can be retrieved through slicing:

```
In [21]: doc5 = nlp(u'Mock Interviews are of great help in cracking real interviews. However, they are always ingonred')

         # Retrieve the third token:
         doc5[2]
Out[21]: are
```

```
In [22]: # Retrieve three tokens from the middle:
         doc5[2:5]
Out[22]: are of great
```

```
In [23]: # Retrieve the last four tokens:
         doc5[-4:]
Out[23]: they are always ingonred
```

Indexing and Slicing in spaCy

Assignment of token is not allowed

```
In [24]: doc6 = nlp(u'I am doing good at Mock Interviews.')
         doc7 = nlp(u'There are high chances of cracking the Data Science Interviews.')
```

```
In [25]: doc6[3]
Out[25]: good
```

```
In [26]: doc7[3]
Out[26]: chances
```

```
In [27]: doc6[3] = doc7[3]

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-27-c84f13888331> in <module>
----> 1 doc6[3] = doc7[3]

TypeError: 'spacy.tokens.doc.Doc' object does not support item assignment
```

Lemmatization

- In contrast to stemming, Lemmatization looks beyond word reduction, and considers a language's full vocabulary to apply a **morphological analysis** to words.
- The lemma of 'was' is 'be', lemma of "rats" is "rat" and the lemma of 'mice' is 'mouse'. Further, the lemma of 'meeting' might be 'meet' or 'meeting' depending on its use in a sentence.
- Lemmatization looks at surrounding text to determine a given word's part of speech. It does not categorize phrases.

Note spaCy do not have stemming. Due to the reason that Lemmatization is seen as more informative than stemming.

```
doc1 = nlp(u"I am a runner running in a race because I love to run since I ran today")
for token in doc1:
    print(token.text, '\t', token.pos_, '\t', token.lemma, '\t', token.lemma_)
```

In [3]:
```
doc1 = nlp(u"I am a runner running in a race because I love to run since I ran today")
for token in doc1:
    print(token.text, '\t', token.pos_, '\t', token.lemma, '\t', token.lemma_)
```

```
I        PRON    561228191312463089      -PRON-
am       VERB    10382539506755952630    be
a        DET     11901859001352538922    a
runner   NOUN    12640964157389618806    runner
running          VERB    12767647472892411841    run
in       ADP     3002984154512732771     in
a        DET     11901859001352538922    a
race     NOUN    8048469955494714898     race
because          ADP     16950148841647037698    because
I        PRON    561228191312463089      -PRON-
love     VERB    3702023516439754181     love
to       PART    3791531372978436496     to
run      VERB    12767647472892411841    run
since    ADP     10066841407251338481    since
I        PRON    561228191312463089      -PRON-
ran      VERB    12767647472892411841    run
today    NOUN    11042482332948150395    today
```

Lemmatization

Creating a Function to find and print Lemma in more structured way.

```
def find_lemmas(text):
    for token in text:
        print(f'{token.text:{12}} {token.pos_:{6}} {token.lemma:<{22}}{token.lemma_}')
```

Here we're using an **f-string** to format the printed text by setting minimum field widths and adding a left-align to the lemma hash value.

Now, let's call that function

```
doc2 = nlp(u"I saw eighteen mice today!")
find_lemmas(doc2)
```

In [4]:
```
def find_lemmas(text):
    for token in text:
        print(f'{token.text:{12}} {token.pos_:{6}} {token.lemma:<{22}} {token.lemma_}')
```

Here we're using an **f-string** to format the printed text by setting minimum field widths and adding a left-align to the lemma hash value.

Now, let's call that function

In [6]:
```
doc2 = nlp(u"I saw eighteen mice today!")
find_lemmas(doc2)
```

```
I            PRON   561228191312463089     -PRON-
saw          VERB   11925638236994514241   see
eighteen     NUM    9609336664675087640    eighteen
mice         NOUN   1384165645700560590    mouse
today        NOUN   11042482332948150395   today
!            PUNCT  17494803046312582752   !
```

Note that the lemma of `saw` is `see`, lemma of `mice` is `mouse`, `mice` is the plural form of `mouse`, and see `eighteen` is a number, *not* an expanded form of `eight` and this is detected while computing lemmas hence it has kept `eighteen` as untouched.

```
doc3 = nlp(u"I am meeting him tomorrow at the meeting.")
find_lemmas(doc3)
```

In [7]:
```
doc3 = nlp(u"I am meeting him tomorrow at the meeting.")
find_lemmas(doc3)
```

```
I            PRON   561228191312463089     -PRON-
am           VERB   10382539506755952630   be
meeting      VERB   6880656908171229526    meet
him          PRON   561228191312463089     -PRON-
tomorrow     NOUN   3573583789758258062    tomorrow
at           ADP    11667289587015813222   at
the          DET    7425985699627899538    the
meeting      NOUN   14798207169164081740   meeting
.            PUNCT  12646065887601541794   .
```

Here the lemma of `meeting` is determined by its Part of Speech tag.

for first `meeting` which is verb it has calculated lemma as `meet`. and for second `meeting` which is Noun, and it has calculated lemma as `meeting` itself.

That is where we can see that spaCy take care of the part of speech while calculating the Lemmas.

```
doc4 = nlp(u"That's an enormous automobile")
find_lemmas(doc4)
```

In [8]:
```
doc4 = nlp(u"That's an enormous automobile")
find_lemmas(doc4)
```

```
That         DET    4380130941430378203    that
's           VERB   10382539506755952630   be
an           DET    15099054000809333061   an
enormous     ADJ    17917224542039855524   enormous
automobile   NOUN   7211811266693931283    automobile
```

Note that Lemmatization does *not* reduce words to their most basic synonym – that is, `enormous` doesn't become `big` and `automobile` doesn't become `car`.

Stop Words

- Words like “a” and “the” appear so frequently that they don’t require tagging as thoroughly as nouns, verbs and modifiers.
- We call them *stop words*, and they can be filtered from the text to be processed.
- **spaCy holds a built-in list of some 305 English stop words**.

```
In [9]: # Print the set of spaCy's default stop words (remember that sets are unordered):
        print(nlp.Defaults.stop_words)
```

```
{'whither', 'somehow', 'toward', 'very', 'top', 'between', 'down', 'enough', 'much', 'onto', 'thus', 'from', 'two', 'unless',
'sixty', 'she', 'former', 'they', 'upon', 'because', 'just', 'into', 'so', 'next', 'under', 'a', 'an', 'along', 'seemed', 'ha
s', 'twelve', 'were', 'first', 'while', 'will', 'amount', 'my', 'neither', 'per', 'you', 'anyone', 'least', 'our', 'its', 'nam
e', 'regarding', 'being', 'no', 'still', 'thru', 'three', 'alone', 'already', 'beforehand', 'amongst', 'either', 'hereafter',
'this', 'who', 'am', 'wherein', 'go', 'using', 'quite', 'even', 'keep', 'give', 'which', 'eight', 'nevertheless', 'against', 'u
s', 'really', 'anywhere', 'another', 'across', 'had', 'he', 'besides', 'somewhere', 'as', 'above', 'be', 'everyone', 'made', 's
ay', 'the', 'cannot', 'five', 'herself', 'put', 'was', 'whence', 'together', 'any', 'however', 'or', 'please', 'fifty', 'himsel
f', 'hundred', 'moreover', 'most', 'everything', 'them', 'whoever', 'their', 'part', 'whereby', 'do', 'nine', 'noone', 'everywh
ere', 'used', 'here', 'of', 'also', 'own', 'whether', 'when', 'indeed', 'and', 'something', 'never', 'whereafter', 'among', 'la
tter', 'nowhere', 'only', 'off', 'anyhow', 'anyway', 'call', 'see', 'can', 'hence', 'his', 'are', 'meanwhile', 'namely', 'hav
e', 'mine', 'same', 'your', 'yourselves', 'ourselves', 'those', 'fifteen', 'rather', 'whenever', 'eleven', 'for', 'various', 'e
lse', 'now', 'beyond', 'might', 'yourself', 'ca', 'others', 'ours', 'seeming', 'every', 'where', 'too', 'each', 'that', 'on',
'side', 'thereby', 'wherever', 're', 'hers', 'whereas', 'perhaps', 'became', 'once', 'by', 'both', 'one', 'formerly', 'could',
'him', 'again', 'to', 'it', 'must', 'about', 'nobody', 'often', 'herein', 'there', 'below', 'four', 'thereupon', 'her', 'your
s', 'up', 'always', 'during', 'empty', 'becomes', 'become', 'then', 'is', 'several', 'beside', 'full', 'since', 'sometimes', 'm
ostly', 'someone', 'around', 'whatever', 'may', 'due', 'should', 'at', 'take', 'none', 'serious', 'ten', 'over', 'therefore',
'whose', 'why', 'back', 'move', 'all', 'does', 'sometime', 'well', 'what', 'less', 'becoming', 'towards', 'few', 'hereby', 'si
x', 'elsewhere', 'myself', 'ever', 'nor', 'except', 'out', 'twenty', 'whom', 'otherwise', 'whereupon', 'make', 'forty', 'but',
'latterly', 'whole', 'afterwards', 'yet', 'without', 'how', 'doing', 'such', 'seems', 'other', 'we', 'thence', 'almost', 'there
after', 'did', 'been', 'not', 'throughout', 'third', 'after', 'some', 'though', 'with', 'get', 'behind', 'me', 'although', 'fro
nt', 'than', 'bottom', 'hereupon', 'before', 'last', 'therein', 'anything', 'nothing', 'if', 'more', 'many', 'these', 'would',
'seem', 'through', 'i', 'in', 'done', 'themselves', 'until', 'itself', 'via', 'within', 'further', 'show'}
```

stop words in spaCy

You can print the total number of stop words using the `len()` function.

```
In [10]: len(nlp.Defaults.stop_words)

Out[10]: 305
```

Check if a word is a stop word

```
In [11]: nlp.vocab['fifteen'].is_stop

Out[11]: True
```

```
In [12]: nlp.vocab['Ashutosh'].is_stop

Out[12]: False
```

Adding a user defined stop word

There may be times when you wish to add a stop word to the default set. Perhaps you decide that `'btw'` (common shorthand for “by the way”) should be considered a stop word.

```
#Add the word to the set of stop words. Use lowercase!
nlp.Defaults.stop_words.add('btw') #alwasy use lowercase while adding the stop words
#Set the stop_word tag on the lexeme
nlp.vocab['btw'].is_stop = True
```

```
In [13]: # Add the word to the set of stop words. Use lowercase!
         nlp.Defaults.stop_words.add('btw') #alwasy use lowercase while adding the stop words

         # Set the stop_word tag on the lexeme
         nlp.vocab['btw'].is_stop = True
```

```
In [14]: len(nlp.Defaults.stop_words)

Out[14]: 306
```

```
In [15]: nlp.vocab['btw'].is_stop

Out[15]: True
```

stop words

Removing a stop word

Alternatively, you may decide that `'without'` should not be considered a stop word.

```
#Remove the word from the set of stop words
nlp.Defaults.stop_words.remove('without')
#Remove the stop_word tag from the lexeme
nlp.vocab['without'].is_stop = False
len(nlp.Defaults.stop_words)
nlp.vocab['beyond'].is_stop
```

```
In [16]: # Remove the word from the set of stop words
         nlp.Defaults.stop_words.remove('without')

         # Remove the stop_word tag from the lexeme
         nlp.vocab['without'].is_stop = False
```

```
In [17]: len(nlp.Defaults.stop_words)
```

```
Out[17]: 305
```

```
In [18]: nlp.vocab['beyond'].is_stop
```

```
Out[18]: True
```

Vocabulary and Matching

In this section we will identify and label specific phrases that match patterns we can define ourselves.

Rule-based Matching

- spaCy offers a rule-matching tool called `Matcher`.
- It allows you to build a library of token patterns.
- It then matches those patterns against a Doc object to return a list of found matches.

You can match on any part of the token including text and annotations, and you can add multiple patterns to the same matcher.

```
#Import the Matcher library
from spacy.matcher import Matcher
matcher = Matcher(nlp.vocab)
```

```
In [107]: Import the Matcher library
          from spacy.matcher import Matcher
          matcher = Matcher(nlp.vocab)
```

Creating patterns

In literature, the phrase 'united states' might appear as one word or two, with or without a hyphen. In this section we'll develop a matcher named 'unitedstates' that finds all three:

```
pattern1 = [{'LOWER': 'unitedstates'}]
pattern2 = [{'LOWER': 'united'}, {'LOWER': 'states'}]
pattern3 = [{'LOWER': 'united'}, {'IS_PUNCT': True}, {'LOWER': 'states'}]
matcher.add('UnitedStates', None, pattern1, pattern2, pattern3)
```

Breaking it further:

- `pattern1` looks for a single token whose lowercase text reads 'unitedstates'
- `pattern2` looks for two adjacent tokens that read 'united' and 'states' in that order
- `pattern3` looks for three adjacent tokens, with a middle token that can be any punctuation.*

* Remember that single spaces are not tokenized, so they don't count as punctuation.
Once we define our patterns, we pass them into `matcher` with the name 'unitedstates', and set *callbacks* to `None`

Applying the matcher to a Doc object

To make you understand I have written United States differently like "United States", "UnitedStates", "United-States" and "United–States"

```
doc = nlp(u'The United States of America is a country consisting of 50 independent states. The first constitution of the UnitedStates was adopted in 1788. The current United-States flag was designed by a high school student – Robert G. Heft.')
found_matches = matcher(doc)
print(found_matches)
for match_id, start, end in found_matches:
    string_id = nlp.vocab.strings[match_id] # get string representation
    span = doc[start:end] # get the matched span
    print(match_id, string_id, start, end, span.text)
```

```
In [110]: doc = nlp(u'The United States of America is a country consisting of 50 independent states. \
          The first constitution of the UnitedStates was adopted in 1788.\
          The current United-States flag was designed by a high school student – Robert G. Heft.')
```

```
In [111]: found_matches = matcher(doc)
          print(found_matches)
```

```
[(15845173719804281779, 1, 3), (15845173719804281779, 19, 20), (15845173719804281779, 25, 28)]
```

`matcher` returns a list of tuples. Each tuple contains an ID for the match, with start & end tokens that map to the span `doc[start:end]`

```
In [112]: for match_id, start, end in found_matches:
              string_id = nlp.vocab.strings[match_id]  # get string representation
              span = doc[start:end]                    # get the matched span
              print(match_id, string_id, start, end, span.text)
```

```
15845173719804281779 UnitedStates 1 3 United States
15845173719804281779 UnitedStates 19 20 UnitedStates
15845173719804281779 UnitedStates 25 28 United-States
```

Setting pattern options and quantifiers

You can make token rules optional by passing an `'OP':'*'` argument. This lets us streamline our patterns list:

```
#Redefine the patterns:
pattern1 = [{'LOWER': 'unitedstates'}]
pattern2 = [{'LOWER': 'united'}, {'IS_PUNCT': True, 'OP':'*'}, {'LOWER': 'states'}]
#Remove the old patterns to avoid duplication:
matcher.remove('UnitedStates')
#Add the new set of patterns to the 'SolarPower' matcher:
matcher.add('someNameToMatcher', None, pattern1, pattern2)
doc = nlp(u'United--States has the world's largest coal reserves.')
found_matches = matcher(doc)
print(found_matches)
```

```
In [113]: # Redefine the patterns:
pattern1 = [{'LOWER': 'unitedstates'}]
pattern2 = [{'LOWER': 'united'}, {'IS_PUNCT': True, 'OP':'*'}, {'LOWER': 'states'}]

# Remove the old patterns to avoid duplication:
matcher.remove('UnitedStates')

# Add the new set of patterns to the 'SolarPower' matcher:
matcher.add('someNameToMatcher', None, pattern1, pattern2)
```

```
In [114]: doc = nlp(u'United--States has the world’s largest coal reserves.')
```

```
In [115]: found_matches = matcher(doc)
print(found_matches)

[(14270899081666383025, 0, 3)]
```

This found both two-word patterns, with and without the hyphen!

The following quantifiers can be passed to the `'OP'` key:

OP	Description
!	Negate the pattern, by requiring it to match exactly 0 times
?	Make the pattern optional, by allowing it to match 0 or 1 times
+	Require the pattern to match 1 or more times
*	Allow the pattern to match zero or more times

spaCy Matcher quantifiers

Careful with lemmas!

Suppose we have another word as “Solar Power” in some sentence. Now, If we want to match on both ‘solar power’ and ‘solar powered’, it might be tempting to look for the *lemma* of ‘powered’ and expect it to be ‘power’. This is not always the case! The lemma of the *adjective* ‘powered’ is still ‘powered’:

```
pattern1 = [{'LOWER': 'solarpower'}]
pattern2 = [{'LOWER': 'solar'}, {'IS_PUNCT': True, 'OP':'*'}, {'LEMMA': 'power'}] # CHANGE THIS PATTERN
#Remove the old patterns to avoid duplication:
matcher.remove('someNameToMatcher') #remove the previously added matcher name
#Add the new set of patterns to the 'SolarPower' matcher:
matcher.add('SolarPower', None, pattern1, pattern2)
doc2 = nlp(u'Solar-powered energy runs solar-powered cars.')
found_matches = matcher(doc2)
print(found_matches)
```

```
In [117]: pattern1 = [{'LOWER': 'solarpower'}]
pattern2 = [{'LOWER': 'solar'}, {'IS_PUNCT': True, 'OP':'*'}, {'LEMMA': 'power'}] # CHANGE THIS PATTERN

# Remove the old patterns to avoid duplication:
matcher.remove('someNameToMatcher') #remove the previously added matcher name

# Add the new set of patterns to the 'SolarPower' matcher:
matcher.add('SolarPower', None, pattern1, pattern2)
```

```
In [118]: doc2 = nlp(u'Solar-powered energy runs solar-powered cars.')
```

```
In [119]: found_matches = matcher(doc2)
print(found_matches)

[(8656102463236116519, 0, 3)]
```

The matcher found the first occurrence because the lemmatizer treated ‘Solar-powered’ as a verb, but not the second as it considered it an adjective.
For this case it may be better to set explicit token patterns.

```
pattern1 = [{'LOWER': 'solarpower'}]
pattern2 = [{'LOWER': 'solar'}, {'IS_PUNCT': True, 'OP':''}, {'LOWER': 'power'}] pattern3 = [{'LOWER': 'solarpowered'}] pattern4 = [{'LOWER': 'solar'}, {'IS_PUNCT': True, 'OP':''}, {'LOWER': 'powered'}]
#Remove the old patterns to avoid duplication:
matcher.remove('SolarPower')
#Add the new set of patterns to the 'SolarPower' matcher:
matcher.add('SolarPower', None, pattern1, pattern2, pattern3, pattern4)
found_matches = matcher(doc2)
print(found_matches)
```

```
In [120]: pattern1 = [{'LOWER': 'solarpower'}]
pattern2 = [{'LOWER': 'solar'}, {'IS_PUNCT': True, 'OP':'*'}, {'LOWER': 'power'}]
pattern3 = [{'LOWER': 'solarpowered'}]
pattern4 = [{'LOWER': 'solar'}, {'IS_PUNCT': True, 'OP':'*'}, {'LOWER': 'powered'}]

# Remove the old patterns to avoid duplication:
matcher.remove('SolarPower')

# Add the new set of patterns to the 'SolarPower' matcher:
matcher.add('SolarPower', None, pattern1, pattern2, pattern3, pattern4)
```

```
In [121]: found_matches = matcher(doc2)
print(found_matches)

[(8656102463236116519, 0, 3), (8656102463236116519, 5, 8)]
```

Other Token Attributes

Besides lemmas, there are a variety of token attributes we can use to determine matching rules:

Attribute	Description
`ORTH`	The exact verbatim text of a token
`LOWER`	The lowercase form of the token text
`LENGTH`	The length of the token text
`IS_ALPHA`, `IS_ASCII`, `IS_DIGIT`	Token text consists of alphanumeric characters, ASCII characters, digits
`IS_LOWER`, `IS_UPPER`, `IS_TITLE`	Token text is in lowercase, uppercase, titlecase

`IS_PUNCT`, `IS_SPACE`, `IS_STOP`	Token is punctuation, whitespace, stop word
`LIKE_NUM`, `LIKE_URL`, `LIKE_EMAIL`	Token text resembles a number, URL, email
`POS`, `TAG`, `DEP`, `LEMMA`, `SHAPE`	The token's simple and extended part-of-speech tag, dependency label, lemma, shape
`ENT_TYPE`	The token's entity label

Token wildcard

You can pass an empty dictionary `{}` as a wildcard to represent **any token**. For example, you might want to retrieve hashtags without knowing what might follow the `#` character:

```
[{'ORTH': '#'}, {}]
```

Phrase Matcher

In the above section we used token patterns to perform rule-based matching. An alternative – and often more efficient – method is to match on terminology lists. In this case we use PhraseMatcher to create a Doc object from a list of phrases, and pass that into `matcher` instead.

```
#Perform standard imports, reset nlp
import spacy
nlp = spacy.load('en_core_web_sm')
# Import the PhraseMatcher library
from spacy.matcher import PhraseMatcher
matcher = PhraseMatcher(nlp.vocab)
```

For this exercise we're going to import a Wikipedia article on *Reaganomics*
Source: https://en.wikipedia.org/wiki/Reaganomics

```
with open('../TextFiles/reaganomics.txt') as f:
doc3 = nlp(f.read())
#First, create a list of match phrases:
phrase_list = ['voodoo economics', 'supply-side economics', 'trickle-down economics', 'free-market economics']
#Next, convert each phrase to a Doc object:
phrase_patterns = [nlp(text) for text in phrase_list]
#Pass each Doc object into matcher (note the use of the asterisk!):
matcher.add('VoodooEconomics', None, *phrase_patterns)
#Build a list of matches:
matches = matcher(doc3)
#(match_id, start, end)
matches
```

In [125]:
```
# Perform standard imports, reset nlp
import spacy
nlp = spacy.load('en_core_web_sm')
```

In [126]:
```
# Import the PhraseMatcher library
from spacy.matcher import PhraseMatcher
matcher = PhraseMatcher(nlp.vocab)
```

For this exercise we're going to import a Wikipedia article on *Reaganomics*
Source: https://en.wikipedia.org/wiki/Reaganomics

In [131]:
```
with open('../TextFiles/reaganomics.txt') as f:
    doc3 = nlp(f.read())
```

In [132]:
```
# First, create a list of match phrases:
phrase_list = ['voodoo economics', 'supply-side economics', 'trickle-down economics', 'free-market economics']

# Next, convert each phrase to a Doc object:
phrase_patterns = [nlp(text) for text in phrase_list]

# Pass each Doc object into matcher (note the use of the asterisk!):
matcher.add('VoodooEconomics', None, *phrase_patterns)

# Build a list of matches:
matches = matcher(doc3)
```

In [133]:
```
# (match_id, start, end)
matches
```

Out[133]:
```
[(3473369816841043438, 41, 45),
 (3473369816841043438, 49, 53),
 (3473369816841043438, 54, 56),
 (3473369816841043438, 61, 65),
 (3473369816841043438, 673, 677),
 (3473369816841043438, 2985, 2989)]
```

The first four matches are where these terms are used in the definition of Reaganomics:

```
doc3[:70]
```

In [134]:
```
doc3[:70]
```

Out[134]:
```
REAGANOMICS
https://en.wikipedia.org/wiki/Reaganomics

Reaganomics (a portmanteau of [Ronald] Reagan and economics attributed to Paul Harvey)[1] refers to the economic policies promo
ted by U.S. President Ronald Reagan during the 1980s. These policies are commonly associated with supply-side economics, referr
ed to as trickle-down economics or voodoo economics by political opponents, and free-market economics by political advocates.
```

Viewing Matches

There are a few ways to fetch the text surrounding a match. The simplest is to grab a slice of tokens from the doc that is wider than the match:

In [135]:
```
doc3[665:685]  # Note that the fifth match starts at doc3[673]
```
Out[135]:
```
same time he attracted a following from the supply-side economics movement, which formed in opposition to Keynesian
```
In [136]:
```
doc3[2975:2995]  # The sixth match starts at doc3[2985]
```
Out[136]:
```
against institutions.[66] His policies became widely known as "trickle-down economics", due to the significant
```
Another way is to first apply the `sentencizer` to the Doc, then iterate through the sentences to the match point:

In [137]:
```
# Build a list of sentences
sents = [sent for sent in doc3.sents]

# In the next section we'll see that sentences contain start and end token values:
print(sents[0].start, sents[0].end)
```
```
0 35
```
In [138]:
```
# Iterate over the sentence list until the sentence end value exceeds a match start value:
for sent in sents:
    if matches[4][1] < sent.end:  # this is the fifth match, that starts at doc3[673]
        print(sent)
        break
```
```
At the same time he attracted a following from the supply-side economics movement, which formed in opposition to Keynesian dema
nd-stimulus economics.
```
This is all about text pre-processing operations which include Tokenization, Lemmatization, Stop Words and Phrase Matching. Hope you enjoyed the post.

Related Articles:

If you have any feedback to improve the content or any thought please write in the comment section below. Your comments are very valuable.

Thank You!

References: