https://campus.datacamp.com/courses/advanced-nlp-with-spacy/finding-words-phrases-names-and-concepts?ex=1

01.01

Introduction to spaCy

ADVANCED NLP WITH SPACY

Ines Montani
spaCy core developer

The nlp object

```
# Import the English language class
from spacy.lang.en import English

# Create the nlp object
nlp = English()
```

At the center of spaCy is the object containing the processing pipeline.
We usually call this variable "nlp".

For example, to create an English nlp object, you can import the English language class from spacy.lang.en and instantiate it.

You can use the nlp object like a function to analyse text.
It contains all the different components in the pipeline.
It also includes language-specific rules used for tokenizing the text into words and punctuation.

spaCy supports a variety of languages that are available in the spacy.lang

- contains the processing pipeline
- includes language-specific rules for tokenization etc.

The Doc object

```
# Created by processing a string of text with the nlp object
doc = nlp("Hello world!")

# Iterate over tokens in a Doc
for token in doc:
    print(token.text)
```

```
Hello
world
!
```

When you process a text with the nlp object, spaCy creates a DOC object -short for "document".

The DOC lets you access information about the text in a structured way, and no information is lost.

The DOC behaves like a normal Python sequence by the way and lets you iterate over its tokens, or get a token by its index.

But more on that later!

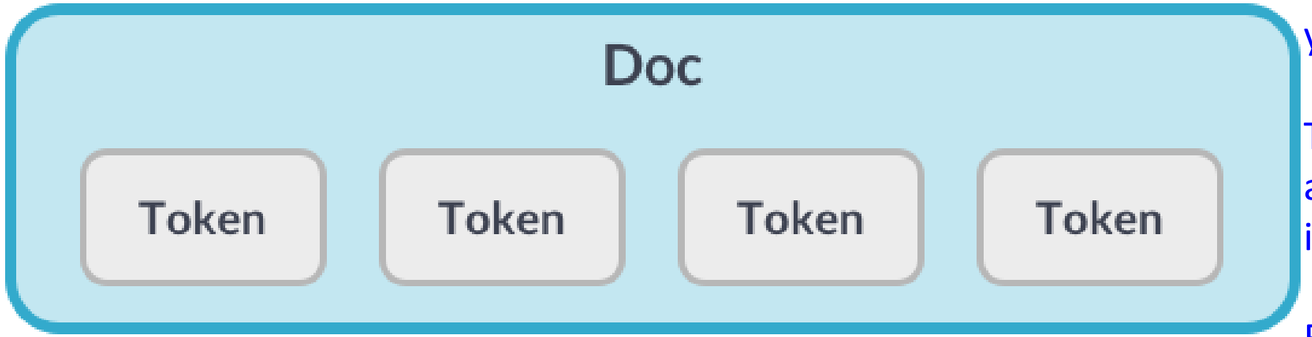
The Token object

Token objects represent the tokens in a document - for example, a word or a punctuation character.

To get a token at a specific position, you can index into the Doc.

Token objects also provide various attributes that let you access more information about the tokens.

For example, the .text attribute returns the verbatim token text.




```
doc = nlp("Hello world!")

# Index into the Doc to get a single Token
token = doc[1]
# Get the token text via the .text attribute
print(token.text)
```

```
world
```

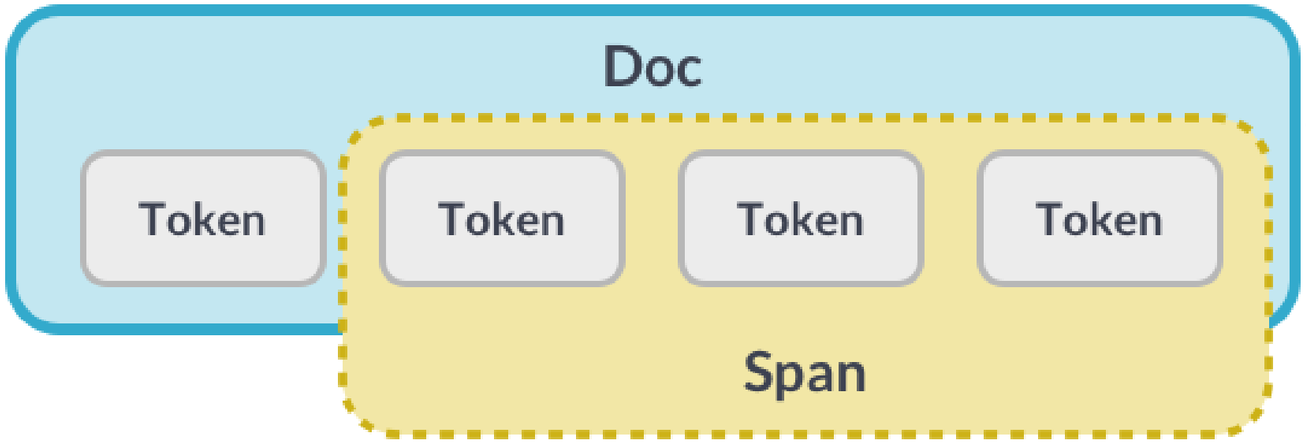
The Span object

A Span object is a slice of the document consisting of one or more tokens.

It's only a view of the DOC and doesn't contain any dat itself.

To create a Span, you can use Python's slice notation.

For example, 1 : (colon) 3 will create a slice starting from the token at position 1, up to but not including! - the token at position 3.




```
doc = nlp("Hello world!")

# A slice from the Doc is a Span object
span = doc[1:4]
# Get the span text via the .text attribute
print(span.text)
```

```
world!
```

Lexical attributes

```
doc = nlp("It costs $5.")
print('Index:    ', [token.i for token in doc])
print('Text:     ', [token.text for token in doc])
print('is_alpha:', [token.is_alpha for token in doc])
print('is_punct:', [token.is_punct for token in doc])
print('like_num:', [token.like_num for token in doc])
```

```
Index:    [0, 1, 2, 3, 4]
Text:     ['It', 'costs', '$', '5', '.']
is_alpha: [True, True, False, False, False]
is_punct: [False, False, False, False, True]
like_num: [False, False, False, True, False]
```

Here you can see some of the available token attributes:

(1) "i" is the index of the token within the parent document.

(2) "text" returns the token text.

(3) "is_alpha", "is_punct" and "like_num" return boolean values indicating whether the token consists of alphanumeric characters, whether it's punctuation or whether it resembles a number. For example, a token "10" (1,0) or the word ten (T,E,N).

These attributes are also called lexical attributes: they refer to the entry in the vocaulary and don't depend on the token's context.

Let's see this in action and process your first text with spaCy.

Let's practice!

ADVANCED NLP WITH SPACY

https://campus.datacamp.com/courses/advanced-nlp-with-spacy/finding-words-phrases-names-and-concepts?ex=5

01.05

Statistical Models

ADVANCED NLP WITH SPACY

Ines Montani
spaCy core developer

What are statistical models?

- Enable spaCy to predict linguistic attributes *in context*
  - Part-of-speech tags
  - Syntactic dependencies
  - Named entities
- Trained on labeled example texts
- Can be updated with more examples to fine-tune predictions

Some of the most interesting things you can analyse are context specific: for example, whether a word is a verb or whether a span of text is a person name.

Statistica models enable spaCy to make predictions in context. This usually includes part-of speech tags, syntatic dependencies and named entities.

Models are trained on large datasets of labeled examples texts. They can be updated with more examples to fine-tune their predictions - for example, to perform better on your specific data.

Model Packages

spaCy provides a number of pre-trained model packages you can download.
For example, the "en_core_web_sm" package is a small English model that supports all core capabilities and is trained on web text.


```
import spacy

nlp = spacy.load('en_core_web_sm')
```

- Binary weights
- Vocabulary
- Meta information (language, pipeline)

The package provide the binary weights that enable spaCy to make predictions.
It also includes the vocabulary and meta information to tel spaCy which language class to use and how to configure the processing pipeline.

Predicting Part-of-speech Tags

Let's take a look at the model's predictions.
In this example, we're using spaCy to predict part-of-speech tags (.pos_ attribute of the token), the word types in context.

```
import spacy

# Load the small English model
nlp = spacy.load('en_core_web_sm')

# Process a text
doc = nlp("She ate the pizza")

# Iterate over the tokens
for token in doc:
    # Print the text and the predicted part-of-speech tag
    print(token.text, token.pos_)
```

In spaCy, attributes that return strings usually end with _ (attribute without the underscore return an ID).

Here, the model correctly predicted "ate" as a verb and "pizza" as a noun.

```
She PRON
ate VERB
the DET
pizza NOUN
```

Predicting Syntactic Dependencies

```
for token in doc:
    print(token.text, token.pos_, token.dep_, token.head.text)
```

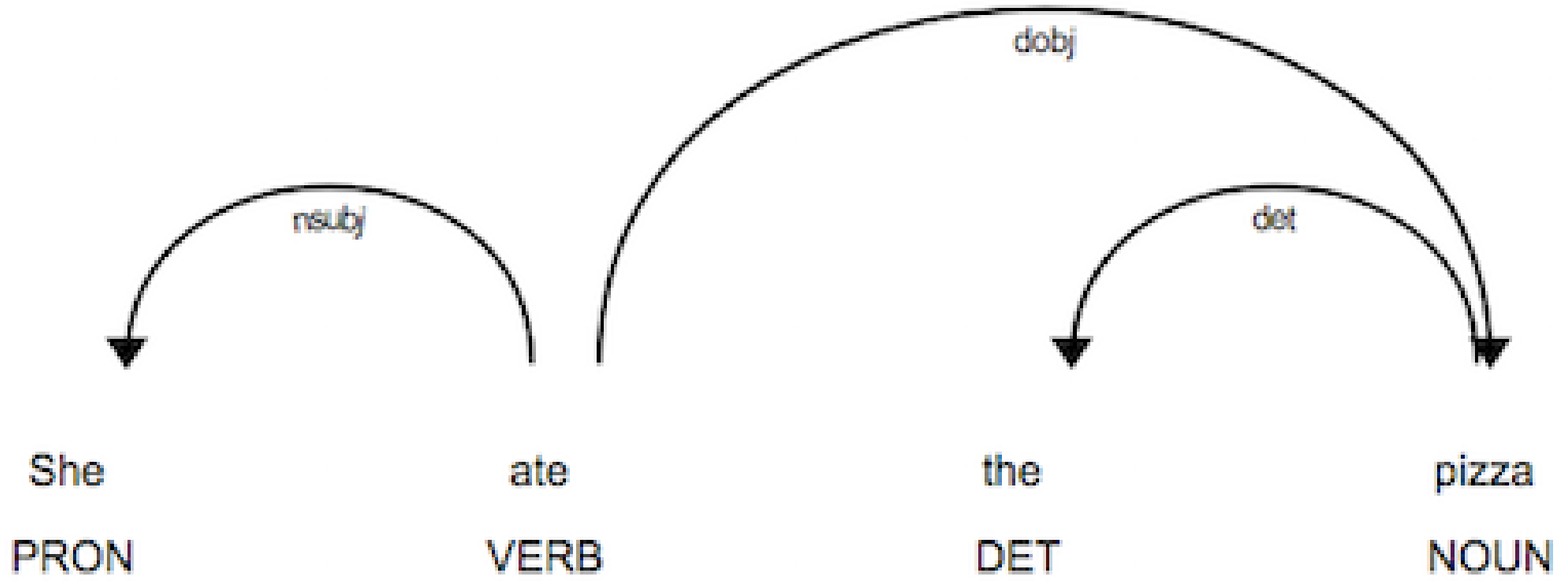
```
She PRON nsubj ate
ate VERB ROOT ate
the DET det pizza
pizza NOUN dobj ate
```

In addition to the part-of-speech tags (.pos_), we can also predict how the words are related.
For example, whether a word is the subject of the sentence or an object.

The .dep_ attribute returns the predicted dependency label.

The .head attribute returns the syntatic head token.

You can also think of it as the parent token this word is attached to.



To describe syntatic dependencies, spaCy uses a standardized label scheme.

Here's an example of some commen labels:

The pronoun "She" is a nominal subject attached to the verb, in this case, to "ate".

The noun "pizza" is a direct object attached to the verb "ate". It is eaten by the subject, "she".

The determiner "the", also known as an article, is attached to the noun "pizza".

Label	Description	Example
nsubj	nominal subject	She
dobj	direct object	pizza
det	determiner (article)	the

Predicting Named Entities

Named entities are real world object that are assigned a name, for example, a person, an organization or a country.

The doc.ents property lets you access the named entities predicted by the model.
It returns an iterator of Span objects, so we can print the entity text and the entity label using the "label underscore" attribute.

In this case, the model is correctly predicting "Apple" as an organization, "U.K." as a geopolitical entity and "$1 billion! as money.

```
# Process a text
doc = nlp(u"Apple is looking at buying U.K. startup for $1 billion")

# Iterate over the predicted entities
for ent in doc.ents:

    # Print the entity text and its label
    print(ent.text, ent.label_)
```

```
Apple ORG
U.K. GPE
$1 billion MONEY
```

Tip: the explain method

A quick tip:
To get definitions for the most common tags and labels, you can use the spacy.explain() helper function.

Get quick definitions of the most common tags and labels.

```
spacy.explain('GPE')
```

```
Countries, cities, states'
```

```
spacy.explain('NNP')
```

```
'noun, proper singular'
```

```
spacy.explain('dobj')
```

```
'direct object'
```

Let's practice!

ADVANCED NLP WITH SPACY

https://campus.datacamp.com/courses/advanced-nlp-with-spacy/finding-words-phrases-names-and-concepts?ex=10

01.10

Rule-based Matching

ADVANCED NLP WITH SPACY

Ines Montani
spaCy core developer

Why not just regular expressions?

- Match on `Doc` objects, not just strings
- Match on tokens and token attributes
- Use the model's predictions
- Example: "duck" (verb) vs. "duck" (noun)

In this video, we'll take a look at spaCy's matcher, which lets you write rules to find words and phrases in text.

Compared to regular expressions, the matcher works with Doc and Token objects instead of only strings.
It's also more flexible: you can search for texts but also other lexical attributes.
You can even write rules that use the model's predictions.

For example, find the word "duck" only it it's a verb, not a noun.

Match patterns

- Lists of dictionaries, one per token
- Match exact token texts

```
[{'ORTH': 'iPhone'}, {'ORTH': 'X'}]
```

- Match lexical attributes

```
[{'LOWER': 'iphone'}, {'LOWER': 'x'}]
```

- Match any token attributes

```
[{'LEMMA': 'buy'}, {'POS': 'NOUN'}]
```

Match patterns are lists of dictionaries.
Each dictionary describes one token.

The keys are the names of token attributes, mapped to their expected values.

In this example, we're looking for two tokens with the text "iPhone" and "X".

We can also match on other token attributes. Here, we're looking for two tokens whose lowercase forms equal "iphone" and "x".

We can even write patterns using attributes predicted by the model. Here, we're matching a token with the lemma "buy", plus a noun.
The lemma is the base form, so this pattern would match phrases like "buying milk" or "bought flowers".

Using the Matcher (1)

```
import spacy

# Import the Matcher
from spacy.matcher import Matcher

# Load a model and create the nlp object
nlp = spacy.load('en_core_web_sm')

# Initialize the matcher with the shared vocab
matcher = Matcher(nlp.vocab)

# Add the pattern to the matcher
pattern = [{'ORTH': 'iPhone'}, {'ORTH': 'X'}]
matcher.add('IPHONE_PATTERN', None, pattern)

# Process some text
doc = nlp("New iPhone X release date leaked")

# Call the matcher on the doc
matches = matcher(doc)
```

---> The first argument is a unique ID to identify which pattern was matched.
The second argument is an optional callback. We don't need one here, so we set it to None.
The third argument is the pattern.

Using the Matcher (2)

```
# Call the matcher on the doc
doc = nlp("New iPhone X release date leaked")
matches = matcher(doc)
# Iterate over the matches
for match_id, start, end in matches:
    # Get the matched span
    matched_span = doc[start:end]
    print(matched_span.text)
```

When you call the matcher on a doc, it returns a list of tuples.
Each tuple consists of three values: the match ID, thhe start index and the end index of matched span.

This means we can iterate over the matches and create a Span object: a slice of the doc at the start and end index.

```
iPhone X
```

- `match_id` : hash value of the pattern name
- `start` : start index of matched span
- `end` : end index of matched span

Matching lexical attributes

Here is an example of a more complex pattern using lexical attributes.
We are looking for five tokens:
- A token consisting of only digits.
- Three case insensitive tokens for "fifa", "world" and "cup"
- A token that consists of punctuation.

The pattern matches the tokens "2018 FIFA World Cup:".

```
pattern = [
    {'IS_DIGIT': True},
    {'LOWER': 'fifa'},
    {'LOWER': 'world'},
    {'LOWER': 'cup'},
    {'IS_PUNCT': True}
]
```

```
doc = nlp("2018 FIFA World Cup: France won!")
```

```
2018 FIFA World Cup:
```

Matching other token attributes

```
pattern = [
    {'LEMMA': 'love', 'POS': 'VERB'},
    {'POS': 'NOUN'}
]
```

In this example, we are looking for two tokens: a verb with the lemma "love", followed by a noun.

This pattern will match "loved dogs" and "love cats".

```
doc = nlp("I loved dogs but now I love cats more.")
```

```
loved dogs
love cats
```

Using operators and quantifiers (1)

Operator and quantifiers let you define how often a token should be matched.

They can be added using the "OP" key.

Here, the "?" operator makes the determiner token optional, so it will match a token with the lemma "buy", an optional article and a noun.

```
pattern = [
    {'LEMMA': 'buy'},
    {'POS': 'DET', 'OP': '?'},  # optional: match 0 or 1 times
    {'POS': 'NOUN'}
]
```

```
doc = nlp("I bought a smartphone. Now I'm buying apps.")
```

```
bought a smartphone
buying apps
```

Using operators and quantifiers (2)

	Description
`{'OP': '!'}`	Negation: match 0 times
`{'OP': '?'}`	Optional: match 0 or 1 times
`{'OP': '+'}`	Match 1 or more times
`{'OP': '*'}`	Match 0 or more times

Operators can make your patterns a lot more poweful, but they also add more complexity -so use them wisely.

Token based matching opens up a lot of new possibilities for information extraction.
So let's try it out and write some patterns!.

Let's practice!

ADVANCED NLP WITH SPACY