Introduction To Spacy

- 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 analyze text.
- spaCy supports variety of languages that are available in spacy.lang.

In [23]:

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

#create the nlp object
nlp = English()

#process a text for English language
doc = nlp("Hello World!")

#print the document text
print(doc.text)
```

```
Hello World!
```

In [22]:

```
# Import the German Language Class
from spacy.lang.de import German

#create nlp object for German class
nlp_german = German()

# Process a text (this is German for: "Kind regards!")
doc = nlp_german("Liebe Grüße!")

#print the document text
print(doc.text)
```

```
Liebe Grüße!
```

In [24]:

```
    # Import the Spanish language class
    from spacy.lang.es import Spanish

    # Create the nlp object
    nlp = Spanish()

    # Process a text (this is Spanish for: "How are you?")
    doc = nlp("¿Cómo estás?")

    # Print the document text
    print(doc.text)
```

```
¿Cómo estás?
```

This nlp object,

- contains the processing pipeline
- includes language-specific rules for tokenization(tokeinzing text into words and punctuation)etc.

Doc Object

- Doc object are created by processing a string of text with the nlp object.
- The obtain doc object behaves like a normal python sequence pipeline, and lets iterate over the tokens or getting tokens by its index.
- first token is at index 0, second token is at index 1 and so-on.
- When you call nlp on a string, spaCy first tokenizes the text and creates a document object.

In [9]:

```
# doc object, created by processing a string of text with the nlp object
doc = nlp("Hello World!")

#Iterate over tokens in a Doc
token_count = 0
for token in doc:
    token_count += 1
    print(f"Token number {token_count} is: {token}")

#access token with help of index
print(f"\nAccessing first token from doc object with the help of index: {doc[0]}")
```

```
Token number 1 is: Hello
Token number 2 is: World
Token number 3 is: !

Accessing first token from doc object with the help of index: Hello
```

The Token Object

- The token object represents the tokens in the document
  - for example word or a punctuation character
- To get a token at 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.

In [12]:

```
#Index into the Doc to get a single Token
token = doc[1]

#Get the token text via the .text attribute
print(token.text)
```

```
World 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 data itself.
- To create a span, you can use python's slice notation.
- For **example:**
  - 1:3 will create a slice starting from the token at position 1, up to – but not including! – the token at position 3.

In [13]:

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

#A slice from the Doc is a Span object
span = doc[1:3]

#Get the span text via .text attribute
print(span.text)
```

```
World!
```

Lexical Attributes

- Here, we can see some available token attirbutes:
  - **i** is the index of the token within the parent document.
  - **text** return the token text
  - **is_alpha** return boolean values indicating whether the token consists of alphabetic character
    - **example:** the word **"ten"**
  - **is_punct** return whether token is punctuation
    - **example: "one, zero"**
  - **like_num**
    - **example:** a token **"10"**
- These attributes are also called **lexical attributes:** they refer to the entry in the vocabulary and don't depend on the token's context.

In [ ]:

```
#doc object, created by processing the string of text with the nlp object
doc = nlp("It costs $5.")

#lexical attributes: i, text
print("Index: ", [token.i for token in doc])
print("Text: ", [token.text for token in doc])

#lexical attributes: is_alpha, is_punct, like_num
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])
```

Lexical attributes example,

In [26]:

```
# Check whether the next token's text attribute is a percent sign "%".
from spacy.lang.en import English

nlp = English()

# Process the text
doc = nlp(
    "In 1990, more than 60% of people in East Asia were in extreme poverty. "
    "Now less than 4% are."
)

# Iterate over the tokens in the doc
for token in doc:
    # Check if the token resembles a number
    if token.like_num:
        # Get the next token in the document
        next_token = doc[token.i+1]
        # Check if the next token's text equals "%"
        if next_token.text == "%":
            print("Percentage found:", token.text)
```

```
Percentage found: 60
Percentage found: 4
```

Statistical Models

- Some of the most interesting things you can analyze are context-specific:
  - for example, whether a word is a verb or whether a span of text is a person name.
- Statistical models enable spaCy to predict linguistic attributes in context. This usually includes,
  - Part-of-speech tags
  - Syntatic dependencies
  - Named entities.
- Models are trained on large datasets of labeled example texts.
- They can be updated with more examples to fine tune their predicions. for example, to perform better on your specific data.

Model Packages

- spaCy provides a number of pre-trained model packages you can download using the spacy download command.
- For example,
  - the **en_core_web_sm** package is a small English model that supports all core capabilities and is trained on web text.
- **what's not included in the model packages?**
  - *The labelled data that the model was trained on.*
  - Once they're trained, they use binary weights to make predictions. That's why it's not necessary to ship them with their training data.

In [28]:

```
!python3 -m spacy download en_core_web_sm
```

```
Defaulting to user installation because normal site-packages is not writeable
Collecting en_core_web_sm==2.3.1
  Downloading https://github.com/explosion/spacy-models/releases/download/en_core_web_sm-2.3.1/en_co
re_web_sm-2.3.1.tar.gz (12.0 MB)
     |████████████████████████████████| 12.0 MB 1.2 MB/s eta 0:00:01     |████████████
| 5.6 MB 725 kB/s eta 0:00:09
Requirement already satisfied: spacy<2.4.0,>=2.3.0 in /home/anish/.local/lib/python3.8/site-packages
(from en_core_web_sm==2.3.1) (2.3.4)
Requirement already satisfied: blis<0.8.0,>=0.4.0 in /home/anish/.local/lib/python3.8/site-packages
(from spacy<2.4.0,>=2.3.0->en_core_web_sm==2.3.1) (0.7.3)
Requirement already satisfied: wasabi<1.1.0,>=0.4.0 in /home/anish/.local/lib/python3.8/site-package
s (from spacy<2.4.0,>=2.3.0->en_core_web_sm==2.3.1) (0.8.0)
Requirement already satisfied: requests<3.0.0,>=2.13.0 in /usr/lib/python3/dist-packages (from spacy
<2.4.0,>=2.3.0->en_core_web_sm==2.3.1) (2.22.0)
Requirement already satisfied: tqdm<5.0.0,>=4.38.0 in /home/anish/.local/lib/python3.8/site-packages
(from spacy<2.4.0,>=2.3.0->en_core_web_sm==2.3.1) (4.48.2)
Requirement already satisfied: catalogue<1.1.0,>=0.0.7 in /home/anish/.local/lib/python3.8/site-pack
ages (from spacy<2.4.0,>=2.3.0->en_core_web_sm==2.3.1) (1.0.0)
Requirement already satisfied: srsly<1.1.0,>=1.0.2 in /home/anish/.local/lib/python3.8/site-packages
(from spacy<2.4.0,>=2.3.0->en_core_web_sm==2.3.1) (1.0.4)
Requirement already satisfied: numpy>=1.15.0 in /usr/lib/python3/dist-packages (from spacy<2.4.0,>=2
.3.0->en_core_web_sm==2.3.1) (1.17.4)
Requirement already satisfied: preshed<3.1.0,>=3.0.2 in /home/anish/.local/lib/python3.8/site-packag
es (from spacy<2.4.0,>=2.3.0->en_core_web_sm==2.3.1) (3.0.4)
Requirement already satisfied: plac<1.2.0,>=0.9.6 in /home/anish/.local/lib/python3.8/site-packages
(from spacy<2.4.0,>=2.3.0->en_core_web_sm==2.3.1) (1.1.3)
Requirement already satisfied: murmurhash<1.1.0,>=0.28.0 in /home/anish/.local/lib/python3.8/site-pa
ckages (from spacy<2.4.0,>=2.3.0->en_core_web_sm==2.3.1) (1.0.4)
Requirement already satisfied: thinc<7.5.0,>=7.4.1 in /home/anish/.local/lib/python3.8/site-packages
(from spacy<2.4.0,>=2.3.0->en_core_web_sm==2.3.1) (7.4.3)
Requirement already satisfied: setuptools in /usr/lib/python3/dist-packages (from spacy<2.4.0,>=2.3.
0->en_core_web_sm==2.3.1) (45.2.0)
Requirement already satisfied: cymem<2.1.0,>=2.0.2 in /home/anish/.local/lib/python3.8/site-packages
(from spacy<2.4.0,>=2.3.0->en_core_web_sm==2.3.1) (2.0.4)
Requirement already satisfied: numpy>=1.15.0 in /usr/lib/python3/dist-packages (from spacy<2.4.0,>=2
.3.0->en_core_web_sm==2.3.1) (1.17.4)
Requirement already satisfied: cymem<2.1.0,>=2.0.2 in /home/anish/.local/lib/python3.8/site-packages
(from spacy<2.4.0,>=2.3.0->en_core_web_sm==2.3.1) (2.0.4)
Requirement already satisfied: murmurhash<1.1.0,>=0.28.0 in /home/anish/.local/lib/python3.8/site-pa
ckages (from spacy<2.4.0,>=2.3.0->en_core_web_sm==2.3.1) (1.0.4)
Requirement already satisfied: wasabi<1.1.0,>=0.4.0 in /home/anish/.local/lib/python3.8/site-package
s (from spacy<2.4.0,>=2.3.0->en_core_web_sm==2.3.1) (0.8.0)
Requirement already satisfied: catalogue<1.1.0,>=0.0.7 in /home/anish/.local/lib/python3.8/site-pack
ages (from spacy<2.4.0,>=2.3.0->en_core_web_sm==2.3.1) (1.0.0)
Requirement already satisfied: srsly<1.1.0,>=1.0.2 in /home/anish/.local/lib/python3.8/site-packages
(from spacy<2.4.0,>=2.3.0->en_core_web_sm==2.3.1) (1.0.4)
Requirement already satisfied: numpy>=1.15.0 in /usr/lib/python3/dist-packages (from spacy<2.4.0,>=2
.3.0->en_core_web_sm==2.3.1) (1.17.4)
Requirement already satisfied: preshed<3.1.0,>=3.0.2 in /home/anish/.local/lib/python3.8/site-packag
es (from spacy<2.4.0,>=2.3.0->en_core_web_sm==2.3.1) (3.0.4)
Requirement already satisfied: tqdm<5.0.0,>=4.38.0 in /home/anish/.local/lib/python3.8/site-packages
(from spacy<2.4.0,>=2.3.0->en_core_web_sm==2.3.1) (4.48.2)
Requirement already satisfied: plac<1.2.0,>=0.9.6 in /home/anish/.local/lib/python3.8/site-packages
(from spacy<2.4.0,>=2.3.0->en_core_web_sm==2.3.1) (1.1.3)
Requirement already satisfied: murmurhash<1.1.0,>=0.28.0 in /home/anish/.local/lib/python3.8/site-pa
ckages (from spacy<2.4.0,>=2.3.0->en_core_web_sm==2.3.1) (1.0.4)
Requirement already satisfied: cymem<2.1.0,>=2.0.2 in /home/anish/.local/lib/python3.8/site-packages
(from spacy<2.4.0,>=2.3.0->en_core_web_sm==2.3.1) (2.0.4)
Requirement already satisfied: blis<0.8.0,>=0.4.0 in /home/anish/.local/lib/python3.8/site-packages
(from spacy<2.4.0,>=2.3.0->en_core_web_sm==2.3.1) (0.7.3)
Building wheels for collected packages: en-core-web-sm
  Building wheel for en-core-web-sm (setup.py) ... done
  Created wheel for en-core-web-sm: filename=en_core_web_sm-2.3.1-py3-none-any.whl size=12047105 sha
256=ff7739dc8bf2d22b81285452364f9d3ec437a226763c4efb1d59b90d69cea844
  Stored in directory: /tmp/pip-ephem-wheel-cache-8h6d3xha/wheels/ee/4d/f7/563214122be1540b5f9197b52
cb3ddb9c4a8070808b22d5a84
Successfully built en-core-web-sm
Installing collected packages: en-core-web-sm
Successfully installed en-core-web-sm-2.3.1
WARNING: You are using pip version 20.3; however, version 20.3.1 is available.
You should consider upgrading via the '/usr/bin/python3 -m pip install --upgrade pip' command.
✔ Download and installation successful
You can now load the model via spacy.load('en_core_web_sm')
```

The spacy.load method loads a model package by name and returns an nlp object.

In [29]:

```
import spacy

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

- The package provides the **binary weights** that enable spaCy to make predictions.
- It also includes the **vocabulary**, and **meta information** to tell 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, the word types in context.
- First, we load the small English model and receive an nlp object.
- Next, we're processing the text "She ate the pizza".
- For each token in the doc, we can print the text and the .pos_ attribute, the predicted part-of-speech tag.

In [30]:

```
import spacy

# load the small english model
nlp = spacy.load("en_core_web_sm")

# process 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_)
```

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

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

Predicting Syntatic Dependencies

- In addition to part-of-speech tags, we can also predict how the words are related.
- 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.

In [31]:

```
import spacy

# load the small english model
nlp = spacy.load("en_core_web_sm")

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

#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
```

Note:

- *A **noun** is naming word. It is used to put a name to things to allow for ease of reference and a common frame of understanding.*
- **_Verbs** are actions or "doing words" used to express an action or state of being.
- *A **pronoun** (I, me, he, she, herself, you, it, that, they, each, few, many, who, whoever, whose, someone, everybody, etc.) is a word that takes the place of a noun*
- a **determiner** is a word that introduces a noun. It always comes before a noun, not after, and it also comes before any other adjectives used to describe. Determiners are required before a singular noun but are optional when it comes to introducing plural nouns.
- **adjective** is a word naming an attribute of a noun, such as sweet, red, or technical

Dependency label scheme

- To describe syntactic dependencies, spaCy uses a standardized label scheme. Here's an example of some common 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**.
- **Dependecy label scheme**
  -

Predicting Named Entities

- Named entities are **real world objects** 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.
- **doc.ents** returns an iterator of Span objects, so we can print the **entity text** and the **entity label** using the **.label_** attribute.

In [5]:

```
import spacy

# load the small english model
nlp = spacy.load("en_core_web_sm")

# process a text
doc = nlp("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 labels
    print("entity text:", ent.text, "  entity labels:", ent.label_)
```

```
entity text: Apple   entity labels: ORG
entity text: U.K.   entity labels: GPE
entity text: $1 billion   entity labels: MONEY
```

- In this case model is correctly predicting, **Apple** as an **Organization** (***ORG***), **U.K.** as a **geopolitical entity** (***GPE***), and **$1 billion** as **money**

spaCy Explain method

- A quick tip: To get definitions for the most common tags and labels, you can use the **spacy.explain** helper function.
- For example,
  - "GPE" for geopolitical entity isn't exactly intuitive – but spacy.explain can tell you that it refers to countries, cities and states.
  - The same work for part-of-speech tags and dependency labels.

In [12]:

```
# get quick definitions of the most common tags and labels.

# GPE, explain
print("Named Entity explain, GPE means:", spacy.explain("GPE"))

# part-of-speech tag, explain
print("POS explain, NNP means:", spacy.explain("NNP"))

# Dependency label, explain
print("Dependency label explain, dobj means:", spacy.explain("dobj"))
print("Dependency label explain, nsubj means:", spacy.explain("nsubj"))
```

```
Named Entity explain, GPE means: Countries, cities, states
POS explain, NNP means: noun, proper singular
Dependency label explain, dobj means: direct object
Dependency label explain, nsubj means: nominal subject
```

Rule-based matching

- 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.
- Example:
  - Find the word **duck** only if it's **verb**, not a **noun.**
  - **duck(verb)** vs. **duck(noun)**

match patterns

- 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 [13]:

```
# match exact token text
[{"TEXT": "iPhone"}, {"TEXT": "X"}]
```

Out[13]:

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

In the above example, we're looking for two tokens with the text **iphone** and **X**

In [14]:

```
# match lexical attributes
[{"LOWER": "iphone"}, {"LOWER": "x"}]
```

Out[14]:

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

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

In [15]:

```
# match any token attributes
[{"LEMMA": "buy"}, {"POS": "NOUN"}]
```

Out[15]:

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

- 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 patten would match phrases like **buying milk** or **bought flowers**

Matcher Examples

- **Steps:**

  1. *To use a pattern, we first import the matcher from spacy.matcher.*
  2. *Load a model and create the nlp object*
  3. *Initialize **matcher** with the shared vocabulary, **nlp.vocab***
  4. *Add a pattern using **matcher.add** method*
     - *First argument: **unique ID** to identify which pattern was matched.*
     - *Second argument: **optional callback** normally set to **None***
     - *Third argument: **pattern** desired pattern*
  5. *Call **matcher** on any doc to match the pattern. This will return **the matches***
     - *When you call matcher on a doc, it returns a list of tuples*
     - *Each tuple consists of three values:*
       - ***the match ID:** hash value of the pattern name ,*
       - ***the start index:** start index of matched span ,*
       - ***the end index:** end index of matched span*
  6. *Iterate over the matches and create span object: a slice of the doc at the start and end index.*

In [23]:

```
# import spacy
import spacy

# Import the matcher from spacy
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 matcher
pattern = [{"TEXT": "iphone"}, {"TEXT": "X"}]
matcher.add("IPHONE_PATTERN", None, pattern)

# process some text
doc = nlp("Upcoming iphone X release date leaked")

# call the matcher on the doc
matches = matcher(doc)

# Iterate over the matches
for match_id, start, end in matches:
    # Get the matched span
    matched_span = doc[start:end]
    print(f"matched text is: {matched_span}")
```

```
matched text is: iphone X
```

patterns-lexical attributes

- Now, lets see an example of a more complex pattern using lexical attributes
- *Look for five tokens having following requirements:*
  - *A token consisting of only digits*
  - *Three case-insensitive tokens for **fifa**, **world**, **cup***
  - *Token that consists of punctuation*

In [32]:

```
# load the model and creat nlp object
nlp = spacy.load("en_core_web_sm")

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

#Add pattern to the matcher
pattern = [
    {"IS_DIGIT": True},
    {"LOWER": "fifa"},
    {"LOWER": "world"},
    {"LOWER": "cup"},
    {"IS_PUNCT": True}
]
matcher.add("FIFA_PATTERN", None, pattern)

# process some text
doc = nlp("2018 FIFA World Cup: France won!")

# call the matcher on the doc
matches = matcher(doc)

# Iterate over the matches
for match_id, start, end in matches:
    matched_span = doc[start:end]
    print("matched patterns: ", matched_span)
```

```
matched patterns:  2018 FIFA World Cup:
```

In [33]:

```
# matching other token attributes


# add pattern to matcher
pattern = [
    {"LEMMA": "love", "POS": "VERB"},
    {"POS": "NOUN"}
]
matcher.add("LOVE_PATTERNS", None, pattern)

# process some text
doc = nlp("I loved dogs but now I love cats more.")

# call the matcher on the doc
matches = matcher(doc)

# Iterate over the matches
for match_id, start, end in matches:
    matched_span = doc[start:end]
    print("matched patterns:", matched_span)
```

```
matched patterns: loved dogs
matched patterns: love cats
```

- In above example we're looking for two tokens:
  - A verb with lemma **love**, followed by a noun
- This pattern matched **loved dogs** and **love cats**

patterns-using operators and quantifiers

- opertors and quantifiers let you define how often a token should be matched.
  - They can be added using the **OP** key.
- **lets define General function named _match*patterns()* that matches the pattern as shown:**

In [42]:

```
#Define General function to obtained matched patterns

# import spacy
import spacy

# Import the matcher from spacy
from spacy.matcher import Matcher


def match_patterns(pattern_id, pattern, some_text):
    """
    matches pattern passed as argument to the text given

    Arguments:
    pattern_id: Unique pattern identifier
    pattern: desired pattern
    some_text: text from where matche pattern to be extracted

    Returns:
    matches: list of tuples of matched pattern
    """

    # 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 pattern to matcher
    matcher.add(pattern_id, None, pattern)

    # process some text
    doc = nlp(some_text)

    # call the matcher on the doc
    matches = matcher(doc)

    # Iterate over the matches
    for match_id, start, end in matches:
        matched_span = doc[start:end]
        print("matched patterns:", matched_span)
    return matches
```

In [43]:

```
# patterns-using operators and quantifiers

# give patten id
pattern_id = "QUNATIFIERS"

# define pattern
pattern = [
    {"LEMMA": "buy"},
    {"POS": "DET", "OP": "?"},  # optional: match 0 or 1 times
    {"POS": "NOUN"}
]
# initialize some text
some_text = "I bought a smartphone. Now I'm buying apps."

# call match_patterns
matches = match_patterns(pattern_id, pattern, some_text)
```

```
matched patterns: bought a smartphone
matched patterns: buying apps
```

- In above example, the ? operator makes the determiner token optional,
- so it will match a token with the lemma **buy**, an optional article and a noun.
- **OP** can have four values:
  - An **!** negates the token, so it's matched 0 times.
  - A **+** matches a token 1 or more times.
  - **?** matches 0 or 1 times
  - And finally, an "*" matches 0 or more times.
  -

Q1.Write one pattern that only matches forms of “download” (tokens with the lemma “download”), followed by a token with the part-of-speech tag "PROPN" (proper noun).

In [44]:

```
# write a pattern that matches a form "download" plus proper noun

#set pattern id
pattern_id = "DOWNLOAD_THINGS_PATTERN"

#define pattern
pattern = [
    {"LEMMA": "download"},
    {"POS": "PROPN"}
]

#some text
some_text = "i downloaded Fortnite on my laptop and can't open the game at all. Help? so when I was downloading M
inecraft, I got the Windows version where it is the '.zip' folder and I used the default program to unpack it...
do I also need to download Winzip?"

#call previously defined function
matches = match_patterns(pattern_id, pattern, some_text)
```

```
matched patterns: downloaded Fortnite
matched patterns: downloading Minecraft
```

Q2.Write one pattern that matches adjectives ("ADJ") followed by one or two "NOUN"s (one noun and one optional noun).

In [45]:

```
# pattern that matches a form, adjectives followed by one or two Noun

# set pattern id
pattern_id = "ADJ_NOUN_PATTERN"

# define pattern
pattern = [
    {"POS": "ADJ"},
    {"POS": "NOUN"},
    {"POS": "NOUN", "OP":"?"}
]

# some text
some_text = "Features of the app include a beautiful design, smart search, automatic \
    labels and optional voice responses."

# call previously defined function for pattern matching
matches = match_patterns(pattern_id, pattern, some_text)
```

```
matched patterns: beautiful design
matched patterns: smart search
matched patterns: optional voice
matched patterns: optional voice responses
```

Large-Scale data analysis with spaCy

Data Structures: Vocab, Lexemes and StringStore

Vocab

- spaCy stores all shared data in a vocabulary, the **Vocab**.
  - This includes words, but also the labels schemes for tags and entities.
- To save memory, spaCy encodes all strings to hash values i.e. If a word occurs more than once, we don't need to save it every time.
- spaCy uses a hash function to generate an ID and stores the string only once in the string store.
  - The string store is available as **nlp.vocab.strings.**
- String Store is a lookup table that works in both directions.
  - You can look up a string and get its hash, and look up a hash to get its string value.
    - Internally, spaCy only communicates in hash IDs
- However, Hash IDs can't be reversed, though. If a word is not in the vocabulary, there's no way to get its string. That's why we always need to pass around the shared vocab.

In [55]:

```
import spacy

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

# Hashes can't be reversed - that's why we need to provide the shared vocab
doc = nlp("i love coffee.")

# obtained hash from strings
coffee_hash = nlp.vocab.strings["coffee"]

# obtained string from hash
coffee_string = nlp.vocab.strings[coffee_hash]

# show string
coffee_string
```

Out[55]:

```
'coffee'
```

Q.Look up the string “cat” in nlp.vocab.strings to get the hash. Look up the hash to get back the string.

In [58]:

```
# import model
from spacy.lang.en import English

nlp = English()
doc = nlp("I have a cat")

# look up the hash for the word "cat"
cat_hash = nlp.vocab.strings["cat"]
print(f"cat hash is: {cat_hash}")

# look up the cat_hash to get the string
cat_string = nlp.vocab.strings[cat_hash]
print(f"string from cat_hash is: {cat_string}")
```

```
cat hash is: 5439657043933447811
string from cat_hash is: cat
```

Lexemes

- Lexemes are context-independent entries in the vocabulary.
- You can get a lexeme by looking up a string or a hash ID in the vocab.
- Lexemes expose attributes, just like tokens.
- They hold context-independent information about a word, like the text, or whether the word consists of alphabetic characters.
- Lexemes don't have part-of-speech tags, dependencies or entity labels. Those depend on the context.

In [63]:

```
# create doc using nlp object
doc = nlp("I love coffee")

# get a lexeme by looking up a string or a hash ID in the vocab
lexeme = nlp.vocab["coffee"]

# print the lexical attributes
print("text: ", lexeme.text, "\nhash: ", lexeme.orth, "\nis_alpha: ", lexeme.is_alpha)
```

```
text:  coffee
hash:  3197928453018144401
is_alpha:  True
```

- As seen above, lexemes contains the **context-independent** information about a word
  - **lexeme.text->** word text
  - **lexeme.orth->** the hash
  - **is_alpha->** return true if text is alphabetical
- As seen above, lexeme doesnot tell context-dependent part-of-speech tags, dependencies or entity labels.

vocab, hashes and lexemes

- Explaination:
  - **Doc** contains word in context– in this case, the tokens "I", "love" and "coffee" with their part-of-speech tags and dependencies.
  - Each **token** refers to a **lexeme**, which knows the word's **hash ID**.
  - To get the **string representation of the word**, spaCy looks up the hash in the **string store.**
- Example:

In [75]:

```
import spacy

# load the small english model
nlp = spacy.load("en_core_web_sm")

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

# access token 0 of doc to predict pos tags, doc contains word in context
print(f"pos_tags of She: {doc[0].pos_}")

# get lexeme by looking a string for token 0, lexeme doesnot contain word in context
token_lexeme = nlp.vocab[doc[0].text]
print("hash ID: ", token_lexeme.orth)

# use hash ID using lexeme to get string representation
print("String using Hash ID:", nlp.vocab.strings[token_lexeme.orth])
```

```
pos_tags of She: PRON
hash ID:  5252949303365547547
String using Hash ID: She
```

Data Structures: Doc, Span and Token

- We take a look at most important data structures: the **Doc**, and its views **Token** and **Span**

The Doc Object

- The **Doc** is one of the central data structures in spaCy. It's created automatically when you process a text with the **nlp** object. But you can also instantiate the class manually.
- After creating the **nlp** object, we can import the **Doc** class from **spacy.tokens**.
- The **Doc** class takes three arguments:
  - **the shared vocab**
  - **the words**
  - **the spaces**
- **Example,**

In [76]:

```
# create an nlp object
from spacy.lang.en import English
nlp = English()

# Import the Doc Class
from spacy.tokens import Doc

# The words and spaces to create the Doc from, desired text: "Hello word!"
words = ["Hello", "world", "!"]
spaces = [True, False, False] #spaces after each token, True if required False if doesnot require

# create a doc manually
doc = Doc(nlp.vocab, words=words, spaces=spaces,)
print(doc.text)
```

```
Hello world!
```

- Here we've created a doc from three words.
- The **spaces** are a list of boolean values indicating whether the word is followed by a space.

The Span Object

- A **span** is a slice of **doc** consisting of one or more tokens.
- The **span** takes at least three arguments:
  - **doc** it refer to
  - **start index**
  - **end index**
  - optional **label**
- **Remember:** end index is exclusive

In [87]:

```
# Import the Doc and Span classes
from spacy.tokens import Doc, Span

# The words and spaces to create the doc from
words = ["Hello", "world", "!"]
spaces = [True, False, False]

# create a Doc manually
doc = Doc(nlp.vocab, words=words, spaces=spaces)

# create a span manually
span = Span(doc, 0, 2)
print("span without label: ", span.text)

# create a span with a label
span_with_label = Span(doc, 0, 2, label="Greetings")
print("Span with label: ", span_with_label)

# Add span to the doc.ents
#The doc.ents are writable, so we can add entities manually by overwriting it with a list of spans.
doc.ents = [span_with_label]
```

```
span without label:  Hello world
Span with label:  Hello world
```

Q.Create the Doc and Span objects manually, and update the named entities-just like spaCy does behind the scenes.

- *Import the Doc and Span classes from spacy.tokens*
- *Use the Doc class directly to create a doc from the words and spaces.*
- *Create a Span for "David Bowie" from the doc and assign it the label "PERSON"*
- *Overwrite the doc.ents with a list of one entity, the "David Bowie" span.*

In [91]:

```
#load model
from spacy.lang.en import English
nlp = English()

# Import the Doc and Span Classes
from spacy.tokens import Doc, Span

words = ["I", "like", "David", "Bowie"]
spaces = [True, True, True, False]

# create a doc from words and spaces
doc = Doc(nlp.vocab, words=words, spaces=spaces)
print("Doc: ", doc.text)

# create a span for "David Browie" from the doc and assign it the label "PERSON"
span = Span(doc, 2, 4, label="PERSON")
print("Span text: ", span.text, "\nSpan label:", span.label_)

# Add the span to the doc's entities
#The doc.ents are writable, so we can add entities manually by overwriting it with a list of spans.
doc.ents = [span]

# Print entities' text and labels
print([(ent.text, ent.label_) for ent in doc.ents])
```

```
Doc:  I like David Bowie
Span text:  David Bowie
Span label: PERSON
[('David Bowie', 'PERSON')]
```

Tips(Doc and Span)

- The Doc and Span are very powerful and optimized for performance. They give you access to all references and relationships of the words and sentences.
- If your application needs to output strings, make sure to convert the doc as late as possible. If you do it too early, you'll lose all relationships between the tokens.
- To keep things consistent, try to use built-in token attributes wherever possible.
  - For example, token.i for the token index.
- Also, don't forget to always pass in the shared vocab!

Q.Analyze a text and collect all proper nouns that are followed by a verb

In [94]:

```
import spacy

nlp = spacy.load("en_core_web_sm")
doc = nlp("Berlin looks like a nice city")

for token in doc:
    if token.pos_ == "PROPN":
        if doc[token.i+1].pos_ == "VERB":
            print("Proper noun found:", token.text)
```

```
Proper noun found: Berlin
```

Word vectors and semantic similarity

- spaCy can compare two objects and predict how similar they are.
  - for example, documents, spans or single tokens.
- The **Doc**, **Token** and **Span** objects have a **.similarity** method that takes another object and returns a floating point number between 0 and 1, indicating how similar they are.
- One thing that's very important: In order to use similarity, you need a larger spaCy model that has word vectors included.
- For example, the medium or large English model – but not the small one. So if you want to use vectors, always go with a model that ends in "md" or "lg".
- **Note:**
  - Similarity is always subjective- whether "dog" and "cat" are similar really depends on how you're looking at it.
  - spaCy's similarity model usually assumes a pretty general-purpose definition of similarity.

In [115]:

```
#Download medium english model with vectors
!python3 -m spacy download en_core_web_md
```

```
Defaulting to user installation because normal site-packages is not writeable
Collecting en_core_web_md==2.3.1
  Downloading https://github.com/explosion/spacy-models/releases/download/en_core_web_md-2.3.1/en_co
re_web_md-2.3.1.tar.gz (50.8 MB)
     |████████████████████████████████| 50.8 MB 137 kB/s eta 0:00:01     |████████████████████
| 33.6 MB 1.7 MB/s eta 0:00:11
Requirement already satisfied: spacy<2.4.0,>=2.3.0 in /home/anish/.local/lib/python3.8/site-packages
(from en_core_web_md==2.3.1) (2.3.4)
Requirement already satisfied: blis<0.8.0,>=0.4.0 in /home/anish/.local/lib/python3.8/site-packages
(from spacy<2.4.0,>=2.3.0->en_core_web_md==2.3.1) (0.7.3)
Requirement already satisfied: setuptools in /usr/lib/python3/dist-packages (from spacy<2.4.0,>=2.3.
0->en_core_web_md==2.3.1) (45.2.0)
Requirement already satisfied: murmurhash<1.1.0,>=0.28.0 in /home/anish/.local/lib/python3.8/site-pa
ckages (from spacy<2.4.0,>=2.3.0->en_core_web_md==2.3.1) (1.0.4)
Requirement already satisfied: thinc<7.5.0,>=7.4.1 in /home/anish/.local/lib/python3.8/site-packages
(from spacy<2.4.0,>=2.3.0->en_core_web_md==2.3.1) (7.4.3)
Requirement already satisfied: numpy>=1.15.0 in /usr/lib/python3/dist-packages (from spacy<2.4.0,>=2
.3.0->en_core_web_md==2.3.1) (1.17.4)
Requirement already satisfied: catalogue<1.1.0,>=0.0.7 in /home/anish/.local/lib/python3.8/site-pack
ages (from spacy<2.4.0,>=2.3.0->en_core_web_md==2.3.1) (1.0.0)
Requirement already satisfied: srsly<1.1.0,>=1.0.2 in /home/anish/.local/lib/python3.8/site-packages
(from spacy<2.4.0,>=2.3.0->en_core_web_md==2.3.1) (1.0.4)
Requirement already satisfied: tqdm<5.0.0,>=4.38.0 in /home/anish/.local/lib/python3.8/site-packages
(from spacy<2.4.0,>=2.3.0->en_core_web_md==2.3.1) (4.48.2)
Requirement already satisfied: plac<1.2.0,>=0.9.6 in /home/anish/.local/lib/python3.8/site-packages
(from spacy<2.4.0,>=2.3.0->en_core_web_md==2.3.1) (1.1.3)
Requirement already satisfied: preshed<3.1.0,>=3.0.2 in /home/anish/.local/lib/python3.8/site-packag
es (from spacy<2.4.0,>=2.3.0->en_core_web_md==2.3.1) (3.0.4)
Requirement already satisfied: requests<3.0.0,>=2.13.0 in /usr/lib/python3/dist-packages (from spacy
<2.4.0,>=2.3.0->en_core_web_md==2.3.1) (2.22.0)
Requirement already satisfied: cymem<2.1.0,>=2.0.2 in /home/anish/.local/lib/python3.8/site-packages
(from spacy<2.4.0,>=2.3.0->en_core_web_md==2.3.1) (2.0.4)
Requirement already satisfied: wasabi<1.1.0,>=0.4.0 in /home/anish/.local/lib/python3.8/site-package
s (from spacy<2.4.0,>=2.3.0->en_core_web_md==2.3.1) (0.8.0)
Requirement already satisfied: numpy>=1.15.0 in /usr/lib/python3/dist-packages (from spacy<2.4.0,>=2
.3.0->en_core_web_md==2.3.1) (1.17.4)
Requirement already satisfied: cymem<2.1.0,>=2.0.2 in /home/anish/.local/lib/python3.8/site-packages
(from spacy<2.4.0,>=2.3.0->en_core_web_md==2.3.1) (2.0.4)
Requirement already satisfied: murmurhash<1.1.0,>=0.28.0 in /home/anish/.local/lib/python3.8/site-pa
ckages (from spacy<2.4.0,>=2.3.0->en_core_web_md==2.3.1) (1.0.4)
Requirement already satisfied: murmurhash<1.1.0,>=0.28.0 in /home/anish/.local/lib/python3.8/site-pa
ckages (from spacy<2.4.0,>=2.3.0->en_core_web_md==2.3.1) (1.0.4)
Requirement already satisfied: srsly<1.1.0,>=1.0.2 in /home/anish/.local/lib/python3.8/site-packages
(from spacy<2.4.0,>=2.3.0->en_core_web_md==2.3.1) (1.0.4)
Requirement already satisfied: numpy>=1.15.0 in /usr/lib/python3/dist-packages (from spacy<2.4.0,>=2
.3.0->en_core_web_md==2.3.1) (1.17.4)
Requirement already satisfied: catalogue<1.1.0,>=0.0.7 in /home/anish/.local/lib/python3.8/site-pack
ages (from spacy<2.4.0,>=2.3.0->en_core_web_md==2.3.1) (1.0.0)
Requirement already satisfied: tqdm<5.0.0,>=4.38.0 in /home/anish/.local/lib/python3.8/site-packages
(from spacy<2.4.0,>=2.3.0->en_core_web_md==2.3.1) (4.48.2)
Requirement already satisfied: blis<0.8.0,>=0.4.0 in /home/anish/.local/lib/python3.8/site-packages
(from spacy<2.4.0,>=2.3.0->en_core_web_md==2.3.1) (0.7.3)
Requirement already satisfied: plac<1.2.0,>=0.9.6 in /home/anish/.local/lib/python3.8/site-packages
(from spacy<2.4.0,>=2.3.0->en_core_web_md==2.3.1) (1.1.3)
Requirement already satisfied: wasabi<1.1.0,>=0.4.0 in /home/anish/.local/lib/python3.8/site-package
s (from spacy<2.4.0,>=2.3.0->en_core_web_md==2.3.1) (0.8.0)
Requirement already satisfied: cymem<2.1.0,>=2.0.2 in /home/anish/.local/lib/python3.8/site-packages
(from spacy<2.4.0,>=2.3.0->en_core_web_md==2.3.1) (2.0.4)
Requirement already satisfied: preshed<3.1.0,>=3.0.2 in /home/anish/.local/lib/python3.8/site-packag
es (from spacy<2.4.0,>=2.3.0->en_core_web_md==2.3.1) (3.0.4)
WARNING: You are using pip version 20.3; however, version 20.3.1 is available.
You should consider upgrading via the '/usr/bin/python3 -m pip install --upgrade pip' command.
✔ Download and installation successful
You can now load the model via spacy.load('en_core_web_md')
```

- Example:
  - Let's say we want to find out whether two documents are similar.
  - First, we load the medium English model, "en_core_web_md".
  - We can then create two doc objects and use the first doc's **similarity** method to compare it to the second.

In [120]:

```
#Similarity Examples no.1

# Load a medium model with vectors
import en_core_web_md
nlp = en_core_web_md.load()

# Compare two documents
doc1 = nlp("I like fast food")
doc2 = nlp("I like pizza")
print(doc1.similarity(doc2))
```

```
0.8627204117787385
```

In [121]:

```
# Compare two tokens
doc = nlp("I like pizza and pasta")
token1 = doc[2]
token2 = doc[4]
print(token1.similarity(token2))
```

```
0.7369546
```

According to the word vectors, the tokens "pizza" and "pasta" are kind of similar, and receive a score of 0.7.

You can also use the similarity methods to compare different types of objects.

For example, a document and a token.

```
Here, the similarity score is pretty low and the two objects are considered fairly dissimilar.
```

Here's another example comparing a span – "pizza and pasta" – to a document about McDonalds.

```
The score returned here is 0.61, so it's determined to be kind of similar.
```

In [128]:

```
# Similarity examples no.2
# Compare a document with a token
doc = nlp("I like pizza")
token = nlp("soap")[0]

print(doc.similarity(token))
```

```
0.32531983166759537
```

In [123]:

```
# Compare a span with a document
span = nlp("I like pizza and pasta")[2:5]
doc = nlp("McDonalds sells burgers")

print(span.similarity(doc))
```

```
0.6199092090831612
```

In [127]:

```
# Example
import en_core_web_md

nlp = en_core_web_md.load()  # make sure to use larger model!
tokens = nlp("dog cat banana")

for token1 in tokens:
    for token2 in tokens:
        print(token1.text, token2.text, token1.similarity(token2))
```

```
dog dog 1.0
dog cat 0.80168545
dog banana 0.24327643
cat dog 0.80168545
cat cat 1.0
cat banana 0.28154364
banana dog 0.24327643
banana cat 0.28154364
banana banana 1.0
```

- In this case, the model's predictions are pretty on point.
- A dog is very similar to a cat, whereas a banana is not very similar to either of them.
- Identical tokens are obviously 100% similar to each other (just not always exactly 1.0, because of vector math and floating point imprecisions).

Q.How does spacy predict similarity?

- Similarity is determined using word vectors, multidimensional representations of meanings of words.
- **word vectors** are generated using an algorithm like **Word2Vec** and a lots of text.
- Vectors can be added to spaCy's statistical models.
- Default: Similarity return by spaCy is the cosine similarity between two vectors- but this can be adjusted if necessary.
- Vectors for objects consisting of several tokens, like the **Doc** and **Span**, default to the average of their token vectors.
  - That's also why you usually get more value out of shorter phrases with fewer irrelevant words.

word vectors in spaCy

- To get an idea how word vectors look like, example:
  - First, load the medium model again, which ships with word vectors.
  - Next, process a text and look up a token's vector using the **.vector** attribute.
  - The result is a 300-dimensional vector of the word "banana".

In [129]:

```
# Load a larger model with vectors
nlp = en_core_web_md.load()

doc = nlp("I have a banana")
# Access the vector via the token.vector attribute
print(doc[3].vector)
```

```
[ 2.0228e-01 -7.6618e-02  3.7032e-01  3.2845e-02 -4.1957e-01  7.2069e-02
 -3.7476e-01  5.7460e-02 -1.2401e-02  5.2949e-01 -5.2380e-01 -1.9771e-01
 -3.4147e-01  5.3317e-01 -2.5331e-02  1.7380e-01  1.6772e-01  8.3984e-01
  5.5107e-02  1.0547e-01  3.7872e-01  2.4275e-01  1.4745e-02  5.5951e-01
  1.2521e-01 -6.7596e-01  3.5842e-01 -4.0028e-02  9.5949e-02 -5.0690e-01
 -8.5318e-02  1.7980e-01  3.3867e-01  1.3230e-01  3.1021e-01  2.1878e-01
  1.6853e-01  1.9874e-01 -5.7385e-01 -1.0649e-01  2.6669e-01  1.2838e-01
 -1.2803e-01 -1.3284e-01  1.2657e-01  8.6723e-01  9.6721e-02  4.8306e-01
  2.1271e-01 -5.4990e-02 -8.2425e-02  2.2408e-01  2.3975e-01 -6.2260e-02
  6.2194e-01 -5.9900e-01  4.3201e-01  2.8143e-01  3.3842e-02 -4.8815e-01
 -2.1359e-01  2.7401e-01  2.4095e-01  4.5950e-01 -1.8605e-01 -1.0497e+00
 -9.7305e-02 -1.8908e-01 -7.0929e-01  4.0195e-01 -1.8768e-01  5.1687e-01
  1.2520e-01  8.4150e-01  1.2097e-01  8.8239e-02 -2.9196e-02  1.2151e-03
  5.6825e-02 -2.7421e-01  2.5564e-01  6.9793e-02 -2.2258e-01 -3.6006e-01
 -2.2402e-01 -5.3699e-02  1.2022e+00  5.4535e-01 -5.7998e-01  1.0905e-01
  4.2167e-01  2.0662e-01  1.2936e-01 -4.1457e-02 -6.6777e-01  4.0467e-01
 -1.5218e-02 -2.7640e-01 -1.5611e-01 -7.9198e-02  4.0037e-02 -1.2944e-01
 -2.4090e-04 -2.6785e-01 -3.8115e-01 -9.7245e-01  3.1726e-01 -4.3951e-01
  4.1934e-01  1.8353e-01 -1.5260e-01 -1.0808e-01 -1.0358e+00  7.6217e-02
  1.6519e-01  2.6526e-04  1.6616e-01 -1.5281e-01  1.8123e-01  7.0274e-01
  5.7956e-03  5.1664e-02 -5.9745e-02 -2.7551e-01 -3.9049e-01  6.1132e-02
  5.5430e-01 -8.7997e-02 -4.1681e-01  3.2826e-01 -5.2549e-01 -4.4288e-01
  8.2183e-03  2.4486e-01 -2.2982e-01 -3.4981e-01  2.6894e-01  3.9166e-01
 -4.1904e-01  1.6191e-01 -2.6263e+00  6.4134e-01  3.9743e-01 -1.2868e-01
 -3.1946e-01 -2.5633e-01 -1.2220e-01  3.2275e-01 -7.9933e-02 -1.5348e-01
  3.1505e-01  3.0591e-01  2.6012e-01  1.8553e-01 -2.4043e-01  4.2886e-02
  4.0622e-01 -2.4256e-01  6.3870e-01  6.9983e-01 -1.4043e-01  2.5209e-01
  4.8984e-01 -6.1067e-02 -3.6766e-01 -5.5089e-01 -3.8265e-01 -2.0843e-01
  2.2832e-01  5.1218e-01  2.7868e-01  4.7652e-01  4.7951e-02 -3.4008e-01
 -3.2873e-01 -4.1967e-01 -7.5499e-02 -3.8954e-01 -2.9622e-02 -3.4070e-01
  2.2170e-01 -6.2856e-02 -5.1903e-01 -3.7774e-01 -4.3477e-03 -5.8301e-01
 -8.7546e-02 -2.3929e-01 -2.4711e-01 -2.5887e-01 -2.9894e-01  1.3715e-01
  2.9892e-02  3.6544e-02 -4.9665e-01 -1.8160e-01  5.2939e-01  2.1992e-01
 -4.4514e-01  3.7798e-01 -5.7062e-01 -4.6946e-02  8.1806e-02  1.9279e-02
  3.3246e-01 -1.4620e-01  1.7156e-01  3.9981e-01  3.6217e-01  1.2816e-01
  3.1644e-01  3.7569e-01 -7.4690e-02 -4.8480e-02 -3.1401e-01 -1.9286e-01
 -3.1294e-01 -1.7553e-02 -1.7514e-01 -2.7587e-02 -1.0000e+00  1.8387e-01
  8.1434e-01 -1.8913e-01  5.0999e-01 -9.1960e-03 -1.9295e-03  2.8189e-01
  2.7247e-02  4.3409e-01 -5.4967e-01 -9.7426e-02 -2.4540e-01 -1.7203e-01
 -8.8650e-02 -3.0298e-01 -1.3591e-01 -2.7765e-01  3.1286e-03  2.0556e-01
 -1.5772e-01 -5.2308e-01 -6.4701e-01 -3.7014e-01  6.9393e-02  1.1401e-01
  2.7594e-01 -1.3875e-01 -2.7268e-01  6.6891e-01 -5.6454e-02  2.4017e-01
 -2.6730e-01  2.9860e-01  1.0083e-01  5.5592e-01  3.2849e-01  7.6858e-02
  1.5528e-01  2.5636e-01 -1.0772e-01 -1.2359e-01  1.1827e-01 -9.9029e-02
 -3.4328e-01  1.1502e-01 -3.7808e-01 -3.9012e-02 -3.4593e-01 -1.9404e-01
 -3.3580e-01 -6.2334e-02  2.8919e-01  2.8032e-01 -5.3741e-01  6.2794e-01
  5.6955e-02  6.2147e-01 -2.5282e-01  4.1670e-01 -1.0108e-02 -2.5434e-01
  4.0003e-01  4.2432e-01  2.2672e-01  1.7553e-01  2.3049e-01  2.8323e-01
  1.3882e-01  3.1218e-03  1.7057e-01  3.6685e-01  2.5247e-03 -6.4009e-01
 -2.9765e-01  7.8943e-01  3.1168e-01 -1.1966e+00 -4.7156e-02  5.3175e-01]
```

Similarity depends on the application context

- Predicting similarity can be useful for many types of applications:
  - To recommend user similar texts based on the ones they have read.
  - It can also be helpful to **flag duplicate content**, like posts on an online platform
- There is no objective definition of word **Similarity**, it always depends on the context and what your applications needs to do.
- **Example:**
  - spaCy's default word vectors assign a very high similarity score to **I like cats** and **I hate cats**.
  - This makes sense, because both texts express sentiment about cats.
  - But in a different application context, you might want to consider the phrases as very **dissimilar**, because they talk about opposite sentiments.

Combining models and rules

- **statisticals model**
  - Statistical models are useful if your application needs to be able to generalize based on a few examples.
    - For instance, detecting product or person names usually benefits from a statistical model.
  - To do this, you would use spaCy's entity recognizer, dependency parser, or part-of-speech tagger.
- **Rule-based approaches**
  - Rule-based approaches on the other hand come in handy if there's a more or less finite number of instances you want to find.
    - For example, all countries or cities of the world, drug names or even dog breeds.
  - In spaCy, you can achieve this with custom tokenization rules, as well as the matcher and phrase matcher.
  -

- **Debugging Patterns**

Efficient phrase matching

- Sometimes it's more efficient to match exact strings instead of writing patterns describing the individual tokens.
- The phrase matcher is another helpful tool to find sequences of words in your data.
- It performs keyword search on the document, but instead of only finding strings, it gives you direct access to the tokens in context.
- It takes **Doc** objects as patterns.
- It's also really fast.
  - This makes it very useful for matching large dictionaries and word lists on large volumes of text.
- **Examples**

In [132]:

```
# import PhraseMatcher
from spacy.matcher import PhraseMatcher

#Instead of a list of dictionaries, we pass in a Doc object as the pattern.
matcher = PhraseMatcher(nlp.vocab)

pattern = nlp("Golden Retriever")
matcher.add("DOG", None, pattern)
doc = nlp("I have a Golden Retriever")

# Iterate over the matches
matches = matcher(doc)

for match_id, start, end in matches:
    # Get the matched span
    span = doc[start:end]
    print("Matched span: ", span.text)
```

```
Matched span:  Golden Retriever
```

- **Efficient Phrase matching** is especially true for finite categories of things- like all countries of the world.
- **Example:**

In [2]:

```
import spacy
from spacy.matcher import PhraseMatcher
from spacy.tokens import Span

nlp = spacy.load("en_core_web_sm")
matcher = PhraseMatcher(nlp.vocab)

patterns = [nlp("Czech Republic"), nlp("Slovakia")]
matcher.add("COUNTRY", None, *patterns) #*patterns-> Czech Republic Slovakia

# create a doc and reset entities
doc = nlp("Czech Republic may help Slovakia protect its airspace")
doc.ents = []

# Iterate over the matches
for match_id, start, end in matcher(doc):
    # Create a Span with the label for "GPE"
    span = Span(doc, start, end, label="GPE")

    # overwrite the doc.ents and add the span
    doc.ents = list(doc.ents)+[span]

    # Get the span's root head token
    span_root_head = span.root.head

    # print the text of the span root's head token and the span text
    print("**Span's root head token, and span text**")
    print(span_root_head.text, "-->", span.text)
    print("\n")
# Print the entities and labels in the documents
print("**Printing and labels:**")
print([(ent.text, ent.label_)for ent in doc.ents if ent.label_=="GPE"])
```

```
**Span's root head token, and span text**
help --> Czech Republic


**Span's root head token, and span text**
protect --> Slovakia


**Printing and labels:**
[('Czech Republic', 'GPE'), ('Slovakia', 'GPE')]
```

Processing Pipelines

- About spaCy's processing pipeline.
  - **processing pipelines:** a series of function applied to a doc attributes like **part-of-speech tags**, **dependency labels**, or **named entities**
- Learn what goes under the hood when you process a text.
- How to write your own components and add them to the pipeline, and
- How to use custom attributes to add your own metadata to the documents, spans and tokens

What happens when you call nlp?

- First the tokenizer is applied to turn the string of text into a **Doc** object (i.e Tokenize text).
- Next, a series of pipeline components is applied to the **doc** in order. They include:
  - **tagger**
  - **The parser**
  - **The entity recognizer** and so on....
- Finally, the processed doc is returned, so you can work with it.
- **Image showing pipelines**

Built-in pipeline components

- spaCy ships with the following built-in pipeline components.
  - The **part-of-speech tagger** sets the **token.tag** and **token.pos** attributes.
  - The **dependency parser** adds the **token.dep** and **token.head** attributes and is also responsible for **detecting sentences** and **base noun phrases**, also known as **noun chunk.**
  - The **named entity recognizer** adds the detected entities to the **doc.ents** property.
    - It also sets the **entity type** attributes on the token that indicate if a token is part of an entity or not.
  - Finally, **text classifier** sets category labels that apply to the whole text, and adds them to the **doc.cats** property
  - Because text categories are always very specific, the text classifier is not included in any of the pretrained models by default. But you can use it to train your own system.
  - **Image View: Built-in Pipeline Components**

Pipeline attributes

- To see the names of the pipeline components present in the current nlp object, you can use the **nlp.pipe_names** attribute.
- For a list of **component name** and **component function** tuples, you can use the **nlp.pipeline** attribute.
  - The **component funtions** are the functions applied to the doc to process it and set attributes- example: part-of-speech tags or named entities.

In [7]:

```
import spacy

nlp = spacy.load("en_core_web_sm")

# list of pipeline component names
print(f"**The list of pipeline component names is:**\n {nlp.pipe_names}")

# list of (name, component) tuples
print(f"\n**The list of name and component tuples is:**\n {nlp.pipeline}")
```

```
**The list of pipeline component names is:**
 ['tagger', 'parser', 'ner']

**The list of name and component tuples is:**
 [('tagger', <spacy.pipeline.pipes.Tagger object at 0x7f0086788970>), ('parser', <spacy.pipeline.pip
es.DependencyParser object at 0x7f00837d10a0>), ('ner', <spacy.pipeline.pipes.EntityRecognizer objec
t at 0x7f00837d11c0>)]
```

Amazing ,
Whenever you're unsure about the current pipeline, you can inspect it by printing **nlp.pipe_names** or **nlp.pipeline**

Custom Pipeline Components

- Custome pipeline components is the powerful feature of spaCy's
- Custome pipeline components let you add your own function to the spaCy pipeline that is executed when you call the **nlp** object on a text.

Q. Why Custom Components?

- Custom components are executed automatically when you call the **nlp** object on a text.
- They're useful for adding your own custom metadata to documents and tokens.
- You can also use **custom components** to update built-in attributes, like the named entity spans i.e **doc.ents**.

Anatomy (Structure) of a custom component

- a **custom pipeline component** is a function or callable that takes a doc, modifies it and return it, so it can be processed by the next component in the pipeline.
- Components can be added to the pipeline using **nlp.add_pipe** method.
  - This **method** takes at least one argument: **the component function**.
- Additionally, to specify **where** to add the component in the pipeline, you can use the following **keyword** shown:

  - Setting last to True will add the component last in the pipeline. This is the default behavior.
  - Setting first to True will add the component first in the pipeline, right after the tokenizer.
  - The before and after arguments let you define the name of an existing component to add the new component before or after. For example, before="ner" will add it before the named entity recognizer.
- **Example**

In [21]:

```
import spacy

# create the nlp object
nlp = spacy.load("en_core_web_sm")

# Define Custom Components
def custom_component(doc):
    """
    print the doc's length

    Arguments:
    doc: document object

    Returns:
    doc: Document object after adding cutom components
    """
    # print the doc's length
    print("**Doc length from custom pipeline**:", len(doc))

    # return the doc object
    return doc

# Add the custom component in first of the pipeline
nlp.add_pipe(custom_component, first=True)

# print the pipeline component names
print("Pipeline:", nlp.pipe_names)

# process text
doc = nlp("My name is anish thapaliya.")
```

```
Pipeline: ['custom_component', 'tagger', 'parser', 'ner']
**Doc length from custom pipeline**: 6
```

As seen above, when we process the text using **nlp** object, the **custom component** is applied to the doc and the length of the document is printed

Q. Write a custom component that uses the PhraseMatcher to find animal names in the document and adds the matched spans to doc.ents.

In [30]:

```
import spacy
from spacy.matcher import PhraseMatcher
from spacy.tokens import Span

nlp = spacy.load("en_core_web_sm")

animals = ["Golden Retriever", "cat", "turtle", "Rattus norvegicus"]

animal_patterns = list(nlp.pipe(animals))

print("animal patterns:", animal_patterns)

matcher = PhraseMatcher(nlp.vocab)

matcher.add("ANIMAL", None, *animal_patterns)

# Define the custom components
def animal_component(doc):
    matches = matcher(doc)

    # create a span for each match and assign a label "ANIMAL"
    span = [Span(doc, start, end, label="ANIMAL") for match_id, start, end in matches]

    # overwrite the doc.ents with the matched span
    doc.ents = span

    return doc

# Add the component to the pipeline after ner
nlp.add_pipe(animal_component, after='ner')
print("added custom components:", nlp.pipe_names)

# process the text and print the text and label for the doc.ents
doc = nlp("I have a cat and a Golden Retriever")
print([(ent.text, ent.label_) for ent in doc.ents])
print(doc.ents)
```

```
animal patterns: [Golden Retriever, cat, turtle, Rattus norvegicus]
added custom components: ['tagger', 'parser', 'ner', 'animal_component']
[('cat', 'ANIMAL'), ('Golden Retriever', 'ANIMAL')]
(cat, Golden Retriever)
```

Setting Custom Attributes

- Adding custom attributes to the **DOC**, **Token**, and **Span** objects to store custom data.

Training a neural network model

- Focused specially on named entity recognizer.
- Learn how to update spaCy's statistical models to customize them for your use case.
  - example: to predict new entity type in online comments.
- Write your own training loop from scratch, and understand the basics of how training works.

Why update the statistical model?

- Statistical models make predictions based on the examples they were trained on.
- You can usually make the model more accurate by showing it examples from your domain.
- You often also want to predict categories specific to your problem, so the model needs to learn about them.
- This is essential for text classification, very useful for entity recognition and a little less critical for tagging and parsing.

How training works

- **Steps**

  1. **Initialize** the model weights randomly with nlp.begin_training
  2. **Predict** a few examples with the current weights by calling **nlp.update**
  3. **Compare** prediction with true labels
  4. **Calculate** how to change weights to improve predictions
  5. **Update** weights slightly
  6. Go back to 2.
- **Explanation**

  - The training data are the examples we want to update the model with.
  - The text should be a sentence, paragraph or longer document. For the best results, it should be similar to what the model will see at runtime.
  - The label is what we want the model to predict. This can be a text category, or an entity span and its type.
  - The gradient is how we should change the model to reduce the current error. It's computed when we compare the predicted label to the true label.
  - After training, we can then save out an updated model and use it in our application.

Creating Training Data(1)

In [38]:

```
import json
from spacy.matcher import Matcher
from spacy.lang.en import English

TEXTS = ['How to preorder the iPhone X', 'iPhone X is coming', 'Should I pay $1,000 for the iPhone X?',
         'The iPhone 8 reviews are here', "iPhone 11 vs iPhone 8: What's the difference?",
         'I need a new phone! Any tips?']

nlp = English()
matcher = Matcher(nlp.vocab)

# Two tokens whose lowercase forms match "iphone" and "x"
pattern1 = [{"LOWER": "iphone"}, {"LOWER": "x"}]

# Token whose lowercase form matches "iphone" and a digit
pattern2 = [{"LOWER": "iphone"}, {"IS_DIGIT": True}]

# Add patterns to the matcher and check the result
matcher.add("GADGET", None, pattern1, pattern2)

# create a doc object for each text using nlp.pipe(gives doc object for each text in list)
for doc in nlp.pipe(TEXTS):
    print([doc[start:end] for match_id, start, end in matcher(doc)])
```

```
[iPhone X]
[iPhone X]
[iPhone X]
[iPhone 8]
[iPhone 11, iPhone 8]
[]
```

Creating Training Data(2)

- Create a doc object for each text using nlp.pipe.
- Match on the doc and create a list of matched spans.
- Get (start character, end character, label) tuples of matched spans.
- Format each example as a tuple of the text and a dict, mapping "entities" to the entity tuples.
- Append the example to TRAINING_DATA and inspect the printed data.

In [48]:

```
import spacy
from spacy.matcher import Matcher
from spacy.lang.en import English

# list of texts
TEXTS = ['How to preorder the iPhone X', 'iPhone X is coming', 'Should I pay $1,000 for the iPhone X?',
         'The iPhone 8 reviews are here', "iPhone 11 vs iPhone 8: What's the difference?",
         'I need a new phone! Any tips?']

nlp = English()
matcher = Matcher(nlp.vocab)
pattern1 = [{"LOWER": "iphone"}, {"LOWER": "x"}]
pattern2 = [{"LOWER": "iphone"}, {"IS_DIGIT": True}]
matcher.add("GADGET", None, pattern1, pattern2)

TRAINING_DATA = []

# create doc object for each text in TEXTS
for doc in nlp.pipe(TEXTS):
    # match on the doc and create a list of matched spans
    spans = [doc[start:end] for match_id, start, end in matcher(doc)]

    # Get (start character, end character, label) tuples of matches
    entities = [(span.start_char, span.end_char, "GADGET") for span in spans]

    # Format the matches as a (doc.text, entities) tuple
    training_example = (doc.text, {"entities": entities})

    # Append the example to the TRAINING DATA
    TRAINING_DATA.append(training_example)

# print the training data
print(*TRAINING_DATA, sep='\n')
```

```
('How to preorder the iPhone X', {'entities': [(20, 28, 'GADGET')]})
('iPhone X is coming', {'entities': [(0, 8, 'GADGET')]})
('Should I pay $1,000 for the iPhone X?', {'entities': [(28, 36, 'GADGET')]})
('The iPhone 8 reviews are here', {'entities': [(4, 12, 'GADGET')]})
("iPhone 11 vs iPhone 8: What's the difference?", {'entities': [(0, 9, 'GADGET'), (13, 21, 'GADGET')
]})
('I need a new phone! Any tips?', {'entities': []})
```

- **Whoo-hoo!** training data is created as seen. Training data is usually created by humans who assign labels to texts
- Before training a model with the data, always double-check that the matcher didn't identify any false positives.

The Training Loop

- spaCy gives full control over the training loop
- **Training Loop**
  - **Loop** for a number of times (epoch).
  - **shuffle** the training data.
  - **Divide** the data into batches (mini-batch).
  - **Update** the model for each batch.
  - **save** the updated model.
- **Here's an example**
  - Let's imagine we have a list of training examples consisting of texts and entity annotations.
  - We want to loop for 10 iterations, so we're iterating over a **range** of 10.
  - Next, we use the **random** module to randomly shuffle the training data.
  - We then use spaCy's **minibatch** utility function to divide the examples into batches.
  - For each batch, we get the texts and annotations and call the **nlp.update** method to update the model.
  - Finally, we call the **nlp.to_disk** method to save the trained model to a directory.

In [51]:

```
import spacy
import random
from spacy.lang.en import English

nlp = English()

TRAINING_DATA = [
    ("How to preorder the iPhone X", {"entities": [(20, 28, "GADGET")]})
    # And many more examples...
]

# Loop for 10 iterations
for i in range(10):
    # shuffle the training data
    random.shuffle(TRAINING_DATA)
    for batch in spacy.util.minibatch(TRAINING_DATA):

        # Split the batch in texts and annotations
        texts = [text for text, annotation in batch]
        annotations = [annotation for text, annotation in batch]

        # Update the model
        nlp.update(texts, annotations)

# Save the model
# nlp.to_disk(path_to_model)
```

setting up a new pipeline from scratch

- In this example, we start off with a blank English model using the **spacy.blank** method. The blank model doesn't have any pipeline components, only the language data and tokenization rules.
- We then create a blank entity recognizer and add it to the pipeline.
- Using the **add_label** method, we can add new string labels to the model.
- We can now call **nlp.begin_training** to initialize the model with random weights.
- To get better accuracy, we want to loop over the examples more than once and randomly shuffle the data on each iteration.
- On each iteration, we divide the examples into batches using spaCy's **minibatch** utility function. Each example consists of a text and its annotations.
- Finally, we update the model with the texts and annotations and continue the loop.

In [55]:

```
import spacy

# examples
examples = [['How to preorder the iPhone X', {'entities': [[20, 28, 'GADGET']]}],
            ['iPhone X is coming', {'entities': [[0, 8, 'GADGET']]}],
            ['Should I pay $1,000 for the iPhone X?', {'entities': [[28, 36, 'GADGET']]}],
            ['The iPhone 8 reviews are here', {'entities': [[4, 12, 'GADGET']]}],
            ['Your iPhone goes up to 11 today', {'entities': [[5, 11, 'GADGET']]}],
            ['I need a new phone! Any tips?', {'entities': []}]]

# start with the blank english model
nlp = spacy.blank("en")

# create blank entity recognizer and add it to the pipeline
ner = nlp.create_pipe("ner")
nlp.add_pipe(ner)

# Add a new label
ner.add_label("GADGET")

# start the training, initialzes the model with random weights
nlp.begin_training()

# train for 10 iterations
for itn in range(10):
    random.shuffle(examples)
    losses = {}
    # Divide the examples into batches, with batch_size=2
    for batch in spacy.util.minibatch(examples, size=2):
        texts = [text for text, annotation in batch]
        annotations = [annotation for text, annotation in batch]

        # update the model
        nlp.update(texts, annotations, losses=losses)
    print(losses)
```

```
{'ner': 33.13037347793579}
{'ner': 20.98947110772133}
{'ner': 7.492994154803455}
{'ner': 5.156381610184326}
{'ner': 16.641603469499387}
{'ner': 7.548577504443529}
{'ner': 3.697585553745739}
{'ner': 2.508725864211314}
{'ner': 1.599127044490345}
{'ner': 0.8498655754095612}
```

In [ ]:

```

```