# spaCy 101: Everything you need to know

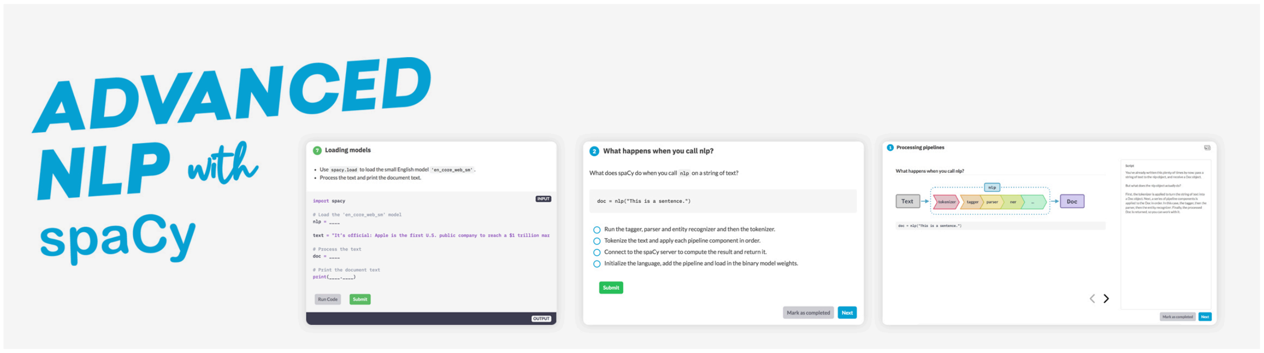
The most important concepts, explained in simple terms

Whether you're new to spaCy, or just want to brush up on some NLP basics and implementation details – this page should have you covered. Each section will explain one of spaCy's features in simple terms and with examples or illustrations. Some sections will also reappear across the usage guides as a quick introduction.

**HELP US IMPROVE THE DOCS**

Did you spot a mistake or come across explanations that are unclear? We always appreciate improvement suggestions or pull requests. You can find a "Suggest edits" link at the bottom of each page that points you to the source.

## Take the free interactive course



In this course you'll learn how to use spaCy to build advanced natural language understanding systems, using both rule-based and machine learning approaches. It includes 55 exercises featuring interactive coding practice, multiple-choice questions and slide decks.

# What’s spaCy?

spaCy is a **free, open-source library** for advanced **Natural Language Processing** (NLP) in Python.

If you’re working with a lot of text, you’ll eventually want to know more about it. For example, what’s it about? What do the words mean in context? Who is doing what to whom? What companies and products are mentioned? Which texts are similar to each other?

spaCy is designed specifically for **production use** and 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**.

**Table of contents**

- ✕ **spaCy is not a platform or "an API"**. Unlike a platform, spaCy does not provide a software as a service, or a web application. It's an open-source library designed to help you build NLP applications, not a consumable service.
- ✕ **spaCy is not an out-of-the-box chat bot engine**. While spaCy can be used to power conversational applications, it's not designed specifically for chat bots, and only provides the underlying text processing capabilities.
- ✕ **spaCy is not research software**. It's built on the latest research, but it's designed to get things done. This leads to fairly different design decisions than NLTK </> or CoreNLP, which were created as platforms for teaching and research. The main difference is that spaCy is integrated and opinionated. spaCy tries to avoid asking the user to choose between multiple algorithms that deliver equivalent functionality. Keeping the menu small lets spaCy deliver generally better performance and developer experience.
- ✕ **spaCy is not a company**. It's an open-source library. Our company publishing spaCy and other software is called Explosion.

## Features

In the documentation, you'll come across mentions of spaCy's features and capabilities. Some of them refer to linguistic concepts, while others are related to more general machine learning functionality.

| | |
|---|---|
| **Tokenization** | Segmenting text into words, punctuations marks etc. |
| **Part-of-speech** (POS) **Tagging** | Assigning word types to tokens, like verb or noun. |
| **Dependency Parsing** | Assigning syntactic dependency labels, describing the relations between individual tokens, like subject or object. |
| **Lemmatization** | Assigning the base forms of words. For example, the lemma of "was" is "be", and the lemma of "rats" is "rat". |
| **Sentence Boundary Detection** (SBD) | Finding and segmenting individual sentences. |
| **Named Entity Recognition** (NER) | Labelling named "real-world" objects, like persons, companies or locations. |
| **Entity Linking** (EL) | Disambiguating textual entities to unique identifiers in a knowledge base. |
| **Similarity** | Comparing words, text spans and documents and how similar they are to each other. |
| **Text Classification** | Assigning categories or labels to a whole document, or parts of a document. |
| **Rule-based Matching** | Finding sequences of tokens based on their texts and linguistic annotations, similar to regular expressions. |
| **Training** | Updating and improving a statistical model's predictions. |
| **Serialization** | Saving objects to files or byte strings. |

## Statistical models

While some of spaCy's features work independently, others require trained pipelines to be loaded, which enable spaCy to **predict** linguistic annotations – for example, whether a word is a verb or a noun. A trained pipeline can consist of multiple components that use a statistical model trained on labeled data. spaCy currently offers trained pipelines for a variety of languages, which can be installed as individual Python modules. Pipeline packages can differ in size, speed, memory usage, accuracy and the data they include. The package you choose always depends on your use case and the texts

- **Binary weights** for the part-of-speech tagger, dependency parser and named entity recognizer to predict those annotations in context.
- **Lexical entries** in the vocabulary, i.e. words and their context-independent attributes like the shape or spelling.
- **Data files** like lemmatization rules and lookup tables.
- **Word vectors**, i.e. multi-dimensional meaning representations of words that let you determine how similar they are to each other.
- **Configuration** options, like the language and processing pipeline settings and model implementations to use, to put spaCy in the correct state when you load the pipeline.

## Linguistic annotations

spaCy provides a variety of linguistic annotations to give you **insights into a text's grammatical structure**. This includes the word types, like the parts of speech, and how the words are related to each other. For example, if you're analyzing text, it makes a huge difference whether a noun is the subject of a sentence, or the object – or whether "google" is used as a verb, or refers to the website or company in a specific context.

LOADING PIPELINES

```
$ python -m spacy download en_core_web_sm
>>> import spacy
>>> nlp = spacy.load("en_core_web_sm")
```

Once you've downloaded and installed a trained pipeline, you can load it via `spacy.load`. This will return a `Language` object containing all components and data needed to process text. We usually call it `nlp`. Calling the `nlp` object on a string of text will return a processed `Doc`:

Editable Code

spaCy v3.7 · Python 3 · via Binder

```
nlp = spacy.load("en_core_web_sm")
doc = nlp("Apple is looking at buying U.K. startup for $1 billion")
for token in doc:
    print(token.text, token.pos_, token.dep_)
```

RUN

Even though a `Doc` is processed – e.g. split into individual words and annotated – it still holds **all information of the original text**, like whitespace characters. You can always get the offset of a token into the original string, or reconstruct the original by joining the tokens and their trailing whitespace. This way, you'll never lose any information when processing text with spaCy.

## Tokenization

During processing, spaCy first **tokenizes** the text, i.e. segments it into words, punctuation and so on. This is done by applying rules specific to each language. For example, punctuation at the end of a sentence should be split off – whereas "U.K." should remain one token. Each `Doc` consists of individual tokens, and we can iterate over them:

Editable Code spaCy v3.7 · Python 3 · via Binder

```
import spacy

nlp = spacy.load("en_core_web_sm")
doc = nlp("Apple is looking at buying U.K. startup for $1 billion")
for token in doc:
    print(token.text)
```

RUN

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Apple | is | looking | at | buying | U.K. | startup | for | $ | 1 | billion |

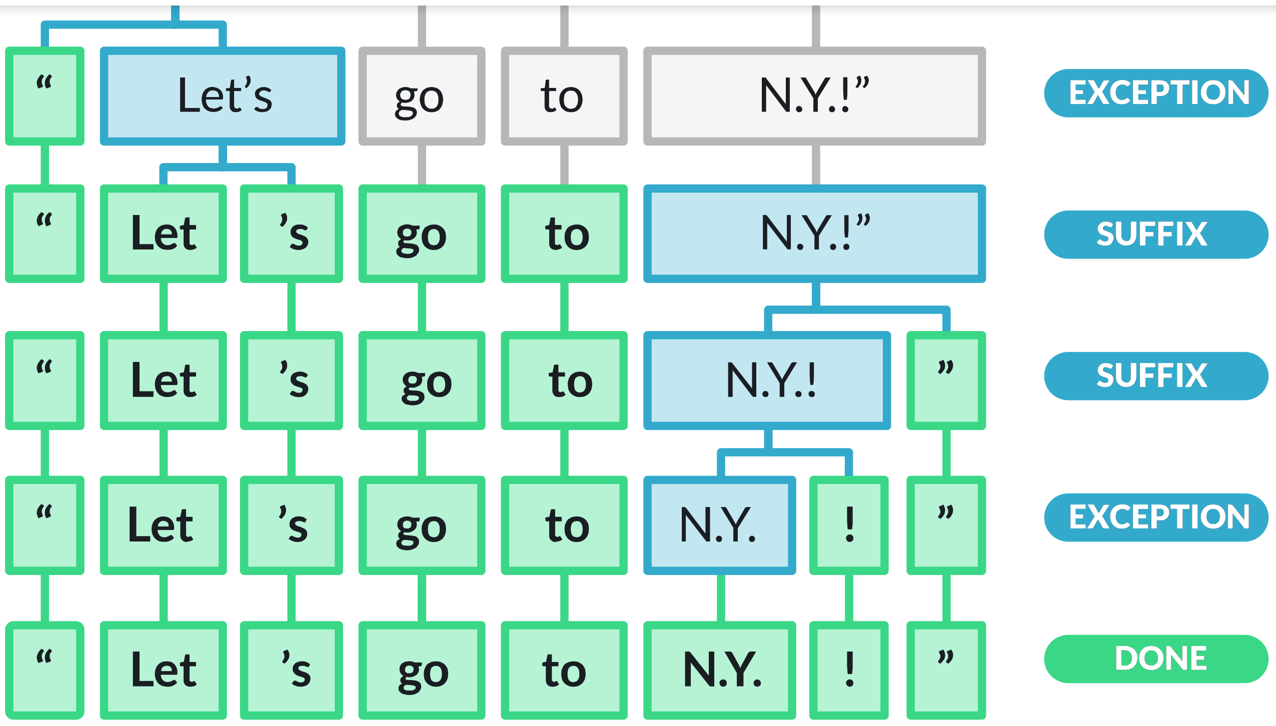
First, the raw text is split on whitespace characters, similar to `text.split(' ')`. Then, the tokenizer processes the text from left to right. On each substring, it performs two checks:

1. **Does the substring match a tokenizer exception rule?** For example, “don’t” does not contain whitespace, but should be split into two tokens, “do” and “n’t”, while “U.K.” should always remain one token.

2. **Can a prefix, suffix or infix be split off?** For example punctuation like commas, periods, hyphens or quotes.

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

**Tokenizer exception:** Special-case rule to split a string into several tokens or prevent a token from being split when punctuation rules are applied.
**Prefix:** Character(s) at the beginning, e.g. `$`, `(`, `“`, `¿`.
**Suffix:** Character(s) at the end, e.g. `km`, `)`, `”`, `!`.
**Infix:** Character(s) in between, e.g. `-`, `--`, `/`, `…`.



While punctuation rules are usually pretty general, tokenizer exceptions strongly depend on the specifics of the individual language. This is why each available language has its own subclass, like `English` or `German`, that loads in lists of hard-coded data and exception rules.

📖 **Tokenization rules**

To learn more about how spaCy’s tokenization rules work in detail, how to **customize and replace** the default tokenizer and how to **add language-specific data**, see the usage guides on language data and customizing the tokenizer.

## Part-of-speech tags and dependencies

NEEDS MODEL

After tokenization, spaCy can **parse** and **tag** a given `Doc`. This is where the trained pipeline and its statistical models come in, which enable spaCy to **make predictions** of which tag or label most likely applies in this context. A trained component includes binary data that is produced by showing a system enough examples for it to make predictions that generalize across the language – for example, a word following “the” in English is most likely a noun.

string representation of an attribute, we need to add an underscore _ to its name.

Editable Code

spaCy v3.7 · Python 3 · via Binder

```
import spacy

nlp = spacy.load("en_core_web_sm")
doc = nlp("Apple is looking at buying U.K. startup for $1 billion")

for token in doc:
    print(token.text, token.lemma_, token.pos_, token.tag_, token.dep_,
            token.shape_, token.is_alpha, token.is_stop)
```

RUN

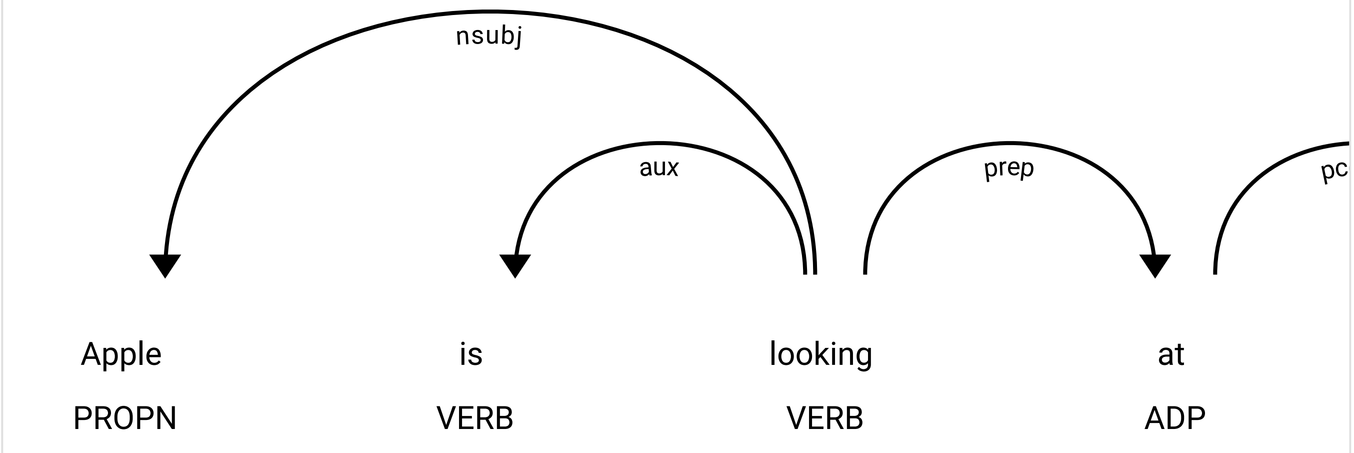
**Text:** The original word text.
**Lemma:** The base form of the word.
**POS:** The simple UPOS part-of-speech tag.
**Tag:** The detailed part-of-speech tag.
**Dep:** Syntactic dependency, i.e. the relation between tokens.
**Shape:** The word shape – capitalization, punctuation, digits.
**is alpha:** Is the token an alpha character?
**is stop:** Is the token part of a stop list, i.e. the most common words of the language?

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Apple | apple | `PROPN` | `NNP` | `nsubj` | `Xxxxx` | `True` | `False` |
| is | be | `AUX` | `VBZ` | `aux` | `xx` | `True` | `True` |
| looking | look | `VERB` | `VBG` | `ROOT` | `xxxx` | `True` | `False` |
| at | at | `ADP` | `IN` | `prep` | `xx` | `True` | `True` |
| buying | buy | `VERB` | `VBG` | `pcomp` | `xxxx` | `True` | `False` |
| U.K. | u.k. | `PROPN` | `NNP` | `compound` | `X.X.` | `False` | `False` |
| startup | startup | `NOUN` | `NN` | `dobj` | `xxxx` | `True` | `False` |
| for | for | `ADP` | `IN` | `prep` | `xxx` | `True` | `True` |
| $ | $ | `SYM` | `$` | `quantmod` | `$` | `False` | `False` |
| 1 | 1 | `NUM` | `CD` | `compound` | `d` | `False` | `False` |
| billion | billion | `NUM` | `CD` | `pobj` | `xxxx` | `True` | `False` |

TIP: UNDERSTANDING TAGS AND LABELS

Most of the tags and labels look pretty abstract, and they vary between languages. `spacy.explain` will show you a short description – for example, `spacy.explain("VBZ")` returns "verb, 3rd person singular present".

Using spaCy's built-in displaCy visualizer, here's what our example sentence and its dependencies look like:



📖 **Part-of-speech tagging and morphology**

To learn more about **part-of-speech tagging** and rule-based morphology, and how to **navigate and use the parse tree** effectively, see the usage guides on part-of-speech tagging and using the dependency parse.

## Named Entities NEEDS MODEL

A named entity is a "real-world object" that's assigned a name – for example, a person, a country, a product or a book title. spaCy can **recognize various types of named entities in a document, by asking the model for a prediction**. Because models are statistical and strongly depend on the examples they were trained on, this doesn't always work *perfectly* and might need some tuning later, depending on your use case.

Named entities are available as the `ents` property of a `Doc` :

Editable Code spaCy v3.7 · Python 3 · via Binder

```
import spacy
```

```
for ent in doc.ents:
    print(ent.text, ent.start_char, ent.end_char, ent.label_)
```

RUN

**Text:** The original entity text.
**Start:** Index of start of entity in the `Doc`.
**End:** Index of end of entity in the `Doc`.
**Label:** Entity label, i.e. type.

| TEXT | START | END | LABEL | DESCRIPTION |
|---|---|---|---|---|
| Apple | 0 | 5 | `ORG` | Companies, agencies, institutions. |
| U.K. | 27 | 31 | `GPE` | Geopolitical entity, i.e. countries, cities, states. |
| $1 billion | 44 | 54 | `MONEY` | Monetary values, including unit. |

Using spaCy's built-in displaCy visualizer, here's what our example sentence and its named entities look like:

Apple **ORG** is looking at buying U.K. **GPE** startup for $1 billion **MONEY**

### 📖 Named Entity Recognition

To learn more about entity recognition in spaCy, how to **add your own entities** to a document and how to **train and update** the entity predictions of a model, see the usage guides on named entity recognition and training pipelines.

Similarity is determined by comparing **word vectors** or “word embeddings”, multi-dimensional meaning representations of a word. Word vectors can be generated using an algorithm like word2vec and usually look like this:

BANANA.VECTOR

```
array([2.02280000e-01,  -7.66180009e-02,   3.70319992e-01,
       3.28450017e-02,  -4.19569999e-01,   7.20689967e-02,
      -3.74760002e-01,   5.74599989e-02,  -1.24009997e-02,
       5.29489994e-01,  -5.23800015e-01,  -1.97710007e-01,
      -3.41470003e-01,   5.33169985e-01,  -2.53309999e-02,
       1.73800007e-01,   1.67720005e-01,   8.39839995e-01,
       5.51070012e-02,   1.05470002e-01,   3.78719985e-01,
       2.42750004e-01,   1.47449998e-02,   5.59509993e-01,
       1.25210002e-01,  -6.75960004e-01,   3.58420014e-01,
       # ... and so on ...
       3.66849989e-01,   2.52470002e-03,  -6.40089989e-01,
      -2.97650009e-01,   7.89430022e-01,   3.31680000e-01,
      -1.19659996e+00,  -4.71559986e-02,   5.31750023e-01], dtype=float32)
```

**Important note**

To make them compact and fast, spaCy’s small pipeline packages (all packages that end in `sm`) **don’t ship with word vectors**, and only include context-sensitive **tensors**. This means you can still use the `similarity()` methods to compare documents, spans and tokens – but the result won’t be as good, and individual tokens won’t have any vectors assigned. So in order to use *real* word vectors, you need to download a larger pipeline package:

```
- python -m spacy download en_core_web_sm
+ python -m spacy download en_core_web_lg
```

Pipeline packages that come with built-in word vectors make them available as the `Token.vector` attribute. `Doc.vector` and `Span.vector` will default to an average of

Editable Code

spaCy v3.7 · Python 3 · via Binder

```
import spacy

nlp = spacy.load("en_core_web_md")
tokens = nlp("dog cat banana afskfsd")

for token in tokens:
    print(token.text, token.has_vector, token.vector_norm, token.is_oov)
```

RUN

**Text**: The original token text.
**has vector**: Does the token have a vector representation?
**Vector norm**: The L2 norm of the token's vector (the square root of the sum of the values squared)
**OOV**: Out-of-vocabulary

The words "dog", "cat" and "banana" are all pretty common in English, so they're part of the pipeline's vocabulary, and come with a vector. The word "afskfsd" on the other hand is a lot less common and out-of-vocabulary – so its vector representation consists of 300 dimensions of `0`, which means it's practically nonexistent. If your application will benefit from a **large vocabulary** with more vectors, you should consider using one of the larger pipeline packages or loading in a full vector package, for example, `en_core_web_lg`, which includes **685k unique vectors**.

spaCy is able to compare two objects, and make a prediction of **how similar they are**. Predicting similarity is useful for building recommendation systems or flagging duplicates. For example, you can suggest a user content that's similar to what they're currently looking at, or label a support ticket as a duplicate if it's very similar to an already existing one.

Each `Doc`, `Span`, `Token` and `Lexeme` comes with a `.similarity` method that lets you compare it with another object, and determine the similarity. Of course similarity is always subjective – whether two words, spans or documents are similar really depends on how you're looking at it. spaCy's similarity implementation usually assumes a pretty general-purpose definition of similarity.

1. Compare two different tokens and try to find the two most *dissimilar* tokens in the texts with the lowest similarity score (according to the vectors).
2. Compare the similarity of two `Lexeme` objects, entries in the vocabulary. You can get a lexeme via the `.lex` attribute of a token. You should see that the similarity results are identical to the token similarity.

Editable Code

spaCy v3.7 · Python 3 · via Binder

```
import spacy

nlp = spacy.load("en_core_web_md")  # make sure to use larger package!
doc1 = nlp("I like salty fries and hamburgers.")
doc2 = nlp("Fast food tastes very good.")

# Similarity of two documents
print(doc1, "<->", doc2, doc1.similarity(doc2))
# Similarity of tokens and spans
french_fries = doc1[2:4]
burgers = doc1[5]
print(french_fries, "<->", burgers, french_fries.similarity(burgers))
```

RUN

## What to expect from similarity results

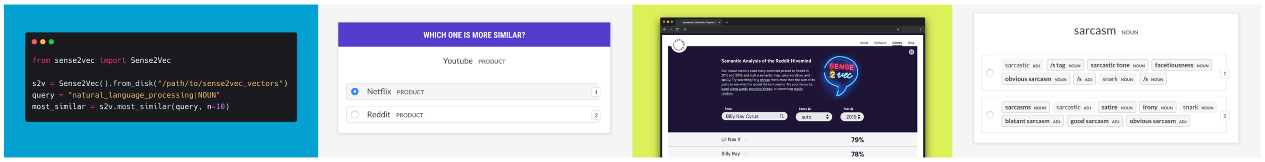
Computing similarity scores can be helpful in many situations, but it's also important to maintain **realistic expectations** about what information it can provide. Words can be related to each other in many ways, so a single "similarity" score will always be a **mix of different signals**, and vectors trained on different data can produce very different results that may not be useful for your purpose. Here are some important considerations to keep in mind:

- There's no objective definition of similarity. Whether "I like burgers" and "I like pasta" is similar **depends on your application**. Both talk about food preferences, which makes them very similar – but if you're analyzing mentions of food, those sentences are pretty dissimilar, because they talk about very different foods.

which isn't necessarily representative of the phrase "fast food".

- Vector averaging means that the vector of multiple tokens is **insensitive to the order** of the words. Two documents expressing the same meaning with dissimilar wording will return a lower similarity score than two documents that happen to contain the same words while expressing different meanings.

### 💡 Tip: Check out sense2vec



```
from sense2vec import Sense2Vec

s2v = Sense2Vec().from_disk("/path/to/sense2vec_vectors")
query = "natural_language_processing|NOUN"
most_similar = s2v.most_similar(query, n=10)
```

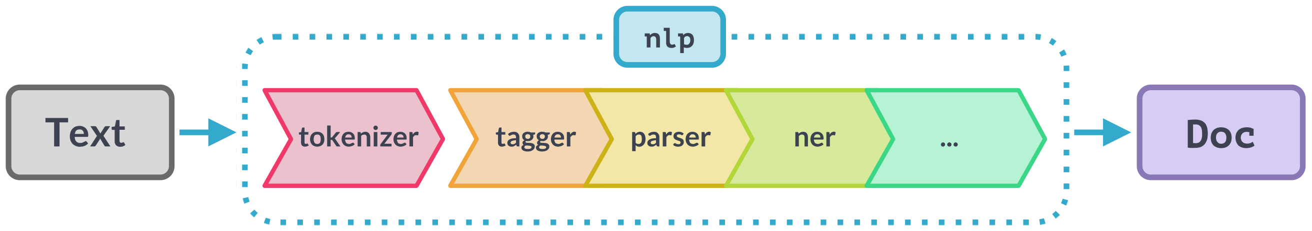
`sense2vec` is a library developed by us that builds on top of spaCy and lets you train and query more interesting and detailed word vectors. It combines noun phrases like "fast food" or "fair game" and includes the part-of-speech tags and entity labels. The library also includes annotation recipes for our annotation tool Prodigy that let you evaluate vectors and create terminology lists. For more details, check out our blog post. To explore the semantic similarities across all Reddit comments of 2015 and 2019, see the interactive demo.

### 📖 Word vectors

To learn more about word vectors, how to **customize them** and how to load **your own vectors** into spaCy, see the usage guide on using word vectors and semantic similarities.

## Pipelines

When you call `nlp` on a text, spaCy first tokenizes the text to produce a `Doc` object. The `Doc` is then processed in several different steps – this is also referred to as the **processing pipeline**. The pipeline



**Name**: ID of the pipeline component.
**Component:** spaCy's implementation of the component.
**Creates:** Objects, attributes and properties modified and set by the component.

| NAME | COMPONENT | CREATES | DESCRIPTION |
|---|---|---|---|
| **tokenizer** | `Tokenizer` | `Doc` | Segment text into tokens. |
| **tagger** | `Tagger` | `Token.tag` | Assign part-of-speech tags. |
| **parser** | `DependencyParser` | `Token.head`, `Token.dep`, `Doc.sents`, `Doc.noun_chunks` | Assign dependency labels. |
| **ner** | `EntityRecognizer` | `Doc.ents`, `Token.ent_iob`, `Token.ent_type` | Detect and label named entities. |
| **lemmatizer** | `Lemmatizer` | `Token.lemma` | Assign base forms. |
| **textcat** | `TextCategorizer` | `Doc.cats` | Assign document labels. |
| **custom** | custom components | `Doc._.xxx`, `Token._.xxx`, `Span._.xxx` | Assign custom attributes, methods or properties. |

PROCESSING PIPELINE

The capabilities of a processing pipeline always depend on the components, their models and how they were trained. For example, a pipeline for named entity recognition needs to include a trained named entity recognizer component with a statistical model and weights that enable it to **make predictions** of entity labels. This is why each pipeline specifies its components and their settings in the config:

```
pipeline = ["tok2vec", "tagger", "parser", "ner"]
```

**Does the order of pipeline components matter?**

**Why is the tokenizer special?**

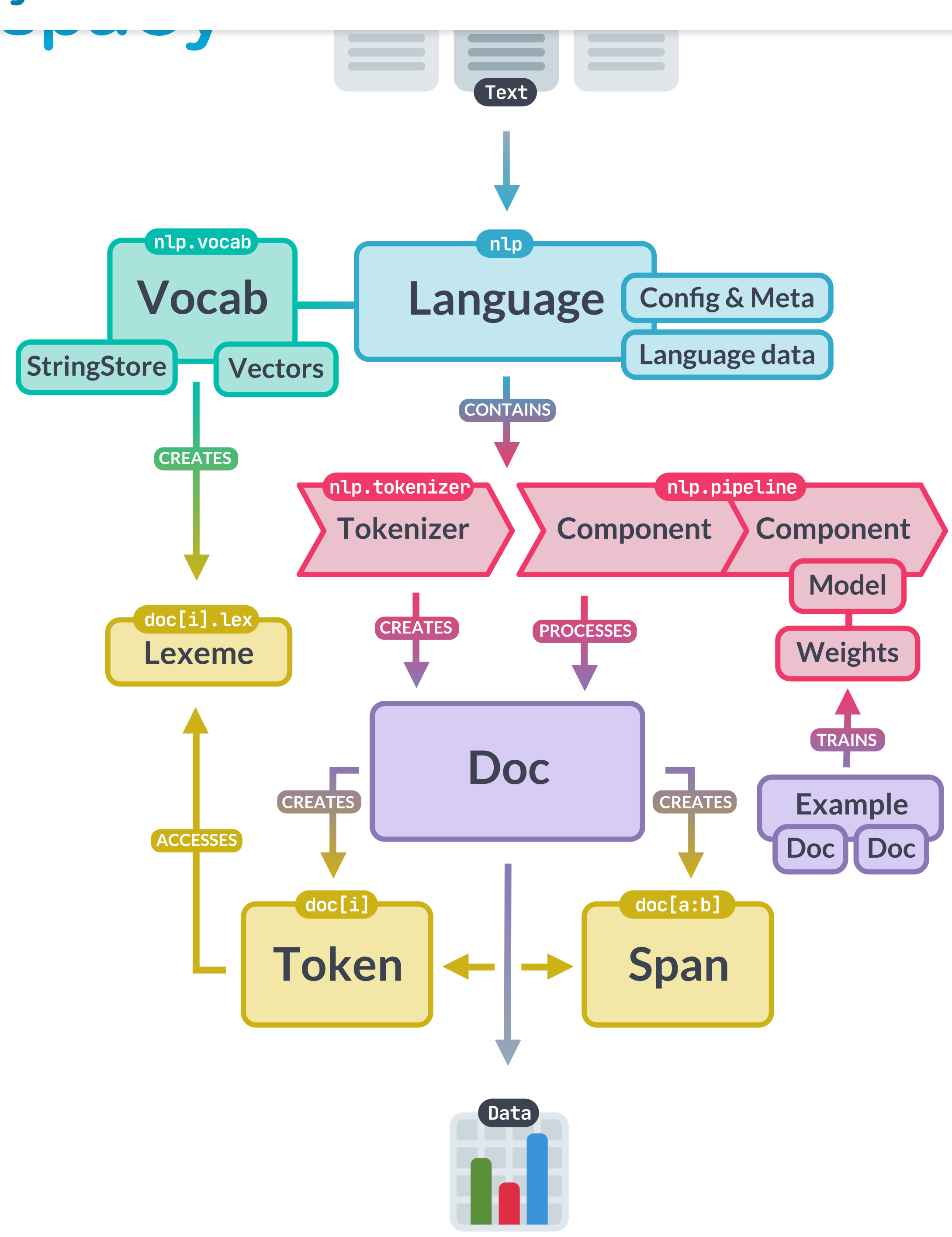
📖 **Processing pipelines**

To learn more about **how processing pipelines work** in detail, how to enable and disable their components, and how to **create your own**, see the usage guide on language processing pipelines.

## Architecture

The central data structures in spaCy are the `Language` class, the `Vocab` and the `Doc` object. The `Language` class is used to process a text and turn it into a `Doc` object. It's typically stored as a variable called `nlp`. The `Doc` object owns the **sequence of tokens** and all their annotations. By centralizing strings, word vectors and lexical attributes in the `Vocab`, we avoid storing multiple copies of this data. This saves memory, and ensures there's a **single source of truth**.

Text annotations are also designed to allow a single source of truth: the `Doc` object owns the data, and `Span` and `Token` are **views that point into it**. The `Doc` object is constructed by the `Tokenizer`, and then **modified in place** by the components of the pipeline. The `Language` object coordinates these components. It takes raw text and sends it through the pipeline, returning an **annotated document**. It also orchestrates training and serialization.

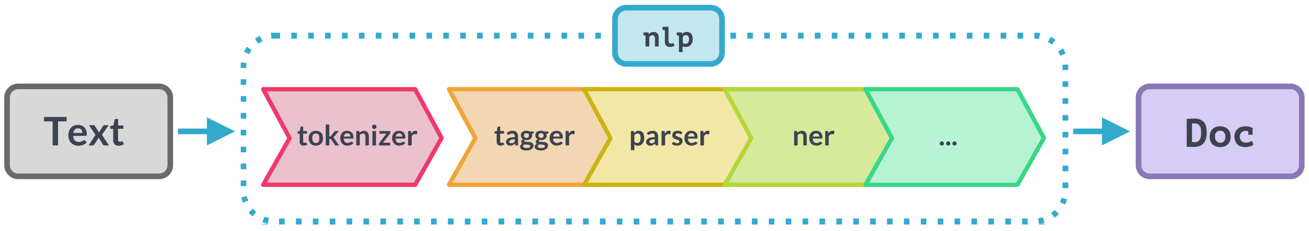


## Container objects

| | |
|---|---|
| `Doc` | A container for accessing linguistic annotations. |
| `DocBin` | A collection of `Doc` objects for efficient binary serialization. Also used for training data . |
| `Example` | A collection of training annotations, containing two `Doc` objects: the reference data and the predictions. |
| `Language` | Processing class that turns text into `Doc` objects. Different languages implement their own subclasses of it. The variable is typically called `nlp` . |
| `Lexeme` | An entry in the vocabulary. It's a word type with no context, as opposed to a word token. It therefore has no part-of-speech tag, dependency parse etc. |
| `Span` | A slice from a `Doc` object. |
| `SpanGroup` | A named collection of spans belonging to a `Doc` . |
| `Token` | An individual token — i.e. a word, punctuation symbol, whitespace, etc. |

## Processing pipeline

The processing pipeline consists of one or more **pipeline components** that are called on the `Doc` in order. The tokenizer runs before the components. Pipeline components can be added using `Language.add_pipe` . They can contain a statistical model and trained weights, or only make rule-based modifications to the `Doc` . spaCy provides a range of built-in components for different language processing tasks and also allows adding custom components.



| | |
|---|---|
| `AttributeRuler` | Set token attributes using matcher rules. |
| `DependencyParser` | Predict syntactic dependencies. |
| `EditTreeLemmatizer` | Predict base forms of words. |
| `EntityLinker` | Disambiguate named entities to nodes in a knowledge base. |
| `EntityRecognizer` | Predict named entities, e.g. persons or products. |
| `EntityRuler` | Add entity spans to the `Doc` using token-based rules or exact phrase matches. |
| `Lemmatizer` | Determine the base forms of words using rules and lookups. |
| `Morphologizer` | Predict morphological features and coarse-grained part-of-speech tags. |
| `SentenceRecognizer` | Predict sentence boundaries. |
| `Sentencizer` | Implement rule-based sentence boundary detection that doesn't require the dependency parse. |
| `Tagger` | Predict part-of-speech tags. |
| `TextCategorizer` | Predict categories or labels over the whole document. |
| `Tok2Vec` | Apply a "token-to-vector" model and set its outputs. |
| `Tokenizer` | Segment raw text and create `Doc` objects from the words. |
| `TrainablePipe` | Class that all trainable pipeline components inherit from. |
| `Transformer` | Use a transformer model and set its outputs. |
| Other functions | Automatically apply something to the `Doc`, e.g. to merge spans of tokens. |

## Matchers

Matchers help you find and extract information from `Doc` objects based on match patterns describing the sequences you're looking for. A matcher operates on a `Doc` and gives you access to

| NAME | DESCRIPTION |
|---|---|
| `DependencyMatcher` | Match sequences of tokens based on dependency trees using Semgrex operators. |
| `Matcher` | Match sequences of tokens, based on pattern rules, similar to regular expressions. |
| `PhraseMatcher` | Match sequences of tokens based on phrases. |

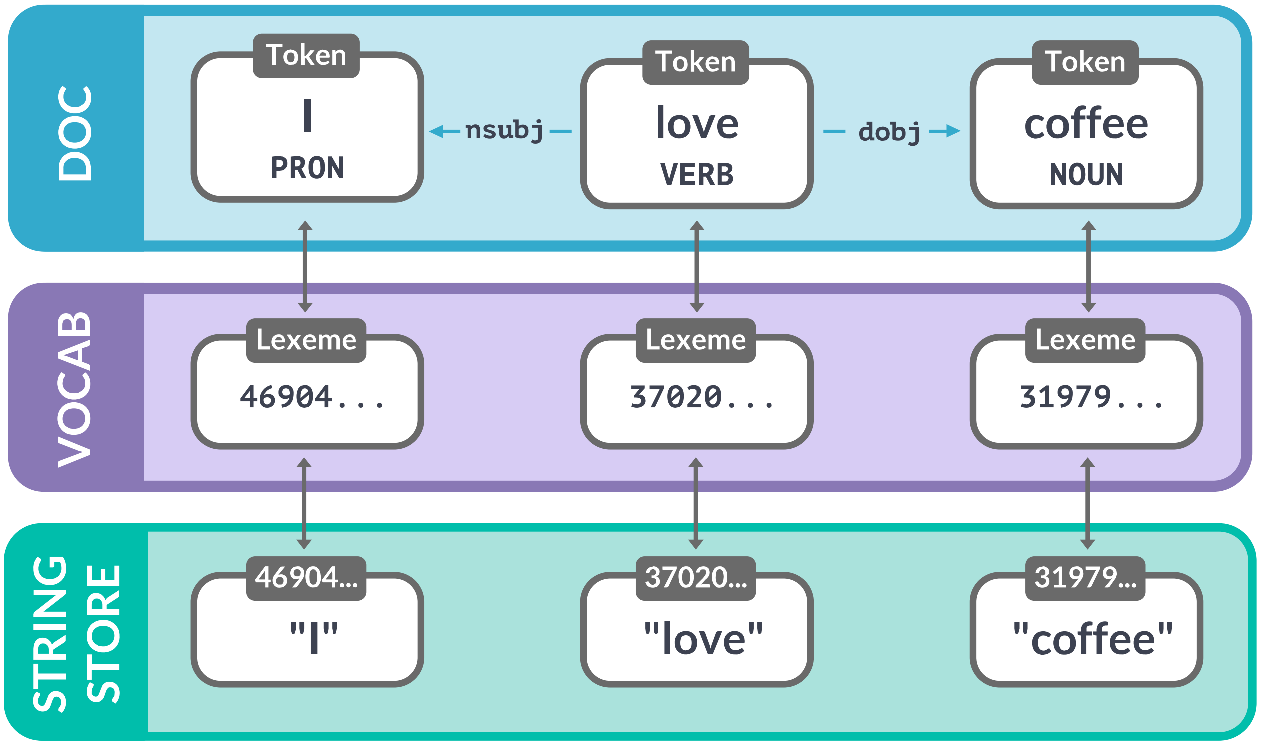
## Other classes

| | |
|---|---|
| `Corpus` | Class for managing annotated corpora for training and evaluation data. |
| `KnowledgeBase` | Abstract base class for storage and retrieval of data for entity linking. |
| `InMemoryLookupKB` | Implementation of `KnowledgeBase` storing all data in memory. |
| `Candidate` | Object associating a textual mention with a specific entity contained in a `KnowledgeBase`. |
| `Lookups` | Container for convenient access to large lookup tables and dictionaries. |
| `MorphAnalysis` | A morphological analysis. |
| `Morphology` | Store morphological analyses and map them to and from hash values. |
| `Scorer` | Compute evaluation scores. |
| `StringStore` | Map strings to and from hash values. |
| `Vectors` | Container class for vector data keyed by string. |
| `Vocab` | The shared vocabulary that stores strings and gives you access to `Lexeme` objects. |

## Vocab, hashes and lexemes

Whenever possible, spaCy tries to store data in a vocabulary, the `Vocab`, that will be **shared by multiple documents**. To save memory, spaCy also encodes all strings to **hash values** – in this case for example, "coffee" has the hash `3197928453018144401`. Entity labels like "ORG" and part-of-speech tags like "VERB" are also encoded. Internally, spaCy only "speaks" in hash values.

**Token**: A word, punctuation mark etc. *in context*, including its attributes, tags and dependencies.
**Lexeme**: A "word type" with no context. Includes the word shape and flags, e.g. if it's lowercase, a digit or punctuation.
**Doc**: A processed container of tokens in context.
**Vocab**: The collection of lexemes.



If you process lots of documents containing the word "coffee" in all kinds of different contexts, storing the exact string "coffee" every time would take up way too much space. So instead, spaCy hashes the string and stores it in the `StringStore` ≣ . You can think of the `StringStore` as a **lookup table that works in both directions** – you can look up a string to get its hash, or a hash to get its string:

Editable Code

spaCy v3.7 · Python 3 · via Binder

```python
import spacy

nlp = spacy.load("en_core_web_sm")
doc = nlp("I love coffee")
print(doc.vocab.strings["coffee"])  # 3197928453018144401
print(doc.vocab.strings[3197928453018144401])  # 'coffee'
```

RUN

Now that all strings are encoded, the entries in the vocabulary **don't need to include the word text** themselves. Instead, they can look it up in the `StringStore` via its hash value. Each entry in the consists of alphabetic characters won't ever change. Its hash value will also always be the same.

Editable Code spaCy v3.7 · Python 3 · via Binder

```
import spacy

nlp = spacy.load("en_core_web_sm")
doc = nlp("I love coffee")
for word in doc:
    lexeme = doc.vocab[word.text]
    print(lexeme.text, lexeme.orth, lexeme.shape_, lexeme.prefix_, lexeme.suffix_
            lexeme.is_alpha, lexeme.is_digit, lexeme.is_title, lexeme.lang_)
```

RUN

**Text**: The original text of the lexeme.
**Orth**: The hash value of the lexeme.
**Shape**: The abstract word shape of the lexeme.
**Prefix**: By default, the first letter of the word string.
**Suffix**: By default, the last three letters of the word string.
**is alpha**: Does the lexeme consist of alphabetic characters?
**is digit**: Does the lexeme consist of digits?

| TEXT | ORTH | SHAPE | PREFIX | SUFFIX | IS_ALPHA | IS_DIGIT |
|---|---|---|---|---|---|---|
| I | 4690420944186131903 | X | I | I | True | False |
| love | 3702023516439754181 | xxxx | l | ove | True | False |
| coffee | 3197928453018144401 | xxxx | c | fee | True | False |

The mapping of words to hashes doesn't depend on any state. To make sure each value is unique, spaCy uses a hash function to calculate the hash **based on the word string**. This also means that the hash for "coffee" will always be the same, no matter which pipeline you're using or how you've configured spaCy.

objects you create have access to the same vocabulary. If they don't, spaCy might not be able to find the strings it needs.

Editable Code spaCy v3.7 · Python 3 · via Binder

```
import spacy
from spacy.tokens import Doc
from spacy.vocab import Vocab

nlp = spacy.load("en_core_web_sm")
doc = nlp("I love coffee")  # Original Doc
print(doc.vocab.strings["coffee"])  # 3197928453018144401
print(doc.vocab.strings[3197928453018144401])  # 'coffee' 👍

empty_doc = Doc(Vocab())  # New Doc with empty Vocab
# empty_doc.vocab.strings[3197928453018144401] will raise an error :(

empty_doc.vocab.strings.add("coffee")  # Add "coffee" and generate hash
print(empty_doc.vocab.strings[3197928453018144401])  # 'coffee' 👍

new_doc = Doc(doc.vocab)  # Create new doc with first doc's vocab
print(new_doc.vocab.strings[3197928453018144401])  # 'coffee' 👍
```

RUN

If the vocabulary doesn't contain a string for `3197928453018144401`, spaCy will raise an error. You can re-add "coffee" manually, but this only works if you actually *know* that the document contains that word. To prevent this problem, spaCy will also export the `Vocab` when you save a `Doc` or `nlp` object. This will give you the object and its encoded annotations, plus the "key" to decode it.

## Serialization

If you've been modifying the pipeline, vocabulary, vectors and entities, or made updates to the component models, you'll eventually want to **save your progress** – for example, everything that's in your `nlp` object. This means you'll have to translate its contents and structure into a format that can

WHAT'S PICKLE?

Pickle is Python's built-in object persistence system. It lets you transfer arbitrary Python objects between processes. This is usually used to load an object to and from disk, but it's also used for distributed computing, e.g. with PySpark or Dask. When you unpickle an object, you're agreeing to execute whatever code it contains. It's like calling `eval()` on a string – so don't unpickle objects from untrusted sources.

All container classes, i.e. `Language` (`nlp`), `Doc`, `Vocab` and `StringStore` have the following methods available:

| METHOD | RETURNS | EXAMPLE |
|---|---|---|
| `to_bytes` | bytes | `data = nlp.to_bytes()` |
| `from_bytes` | object | `nlp.from_bytes(data)` |
| `to_disk` | - | `nlp.to_disk("/path")` |
| `from_disk` | object | `nlp.from_disk("/path")` |

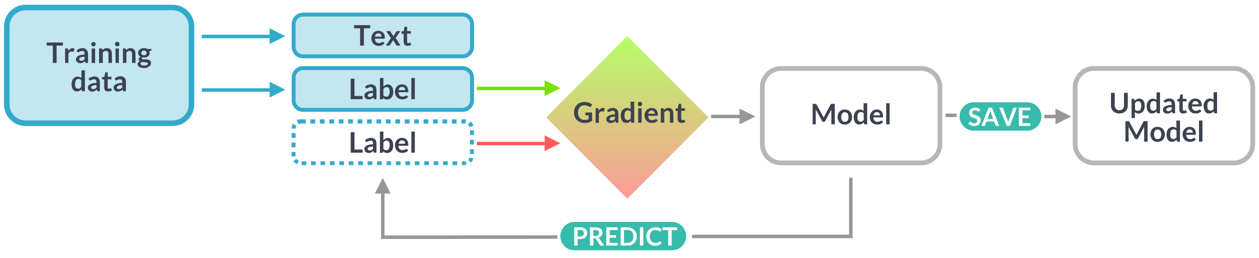
📖 **Saving and loading**

To learn more about how to **save and load your own pipelines**, see the usage guide on saving and loading.

# Training

spaCy's tagger, parser, text categorizer and many other components are powered by **statistical models**. Every "decision" these components make – for example, which part-of-speech tag to assign, or whether a word is a named entity – is a **prediction** based on the model's current **weight values**. The weight values are estimated based on examples the model has seen during **training**. To train a model, Training is an iterative process in which the model's predictions are compared against the reference annotations in order to estimate the **gradient of the loss**. The gradient of the loss is then used to calculate the gradient of the weights through backpropagation. The gradients indicate how the weight values should be changed so that the model's predictions become more similar to the reference labels over time.

**Training data:** Examples and their annotations.
**Text:** The input text the model should predict a label for.
**Label:** The label the model should predict.
**Gradient:** The direction and rate of change for a numeric value. Minimising the gradient of the weights should result in predictions that are closer to the reference labels on the training data.



When training a model, we don't just want it to memorize our examples – we want it to come up with a theory that can be **generalized across unseen data**. After all, we don't just want the model to learn that this one instance of "Amazon" right here is a company – we want it to learn that "Amazon", in contexts *like this*, is most likely a company. That's why the training data should always be representative of the data we want to process. A model trained on Wikipedia, where sentences in the first person are extremely rare, will likely perform badly on Twitter. Similarly, a model trained on romantic novels will likely perform badly on legal text.

This also means that in order to know how the model is performing, and whether it's learning the right things, you don't only need **training data** – you'll also need **evaluation data**. If you only test the model with the data it was trained on, you'll have no idea how well it's generalizing. If you want to train a model from scratch, you usually need at least a few hundred examples for both training and evaluation.

📖 **Training pipelines and models**

To learn more about **training and updating** pipelines, how to create training data and how to improve spaCy's named models, see the usage guides on training.

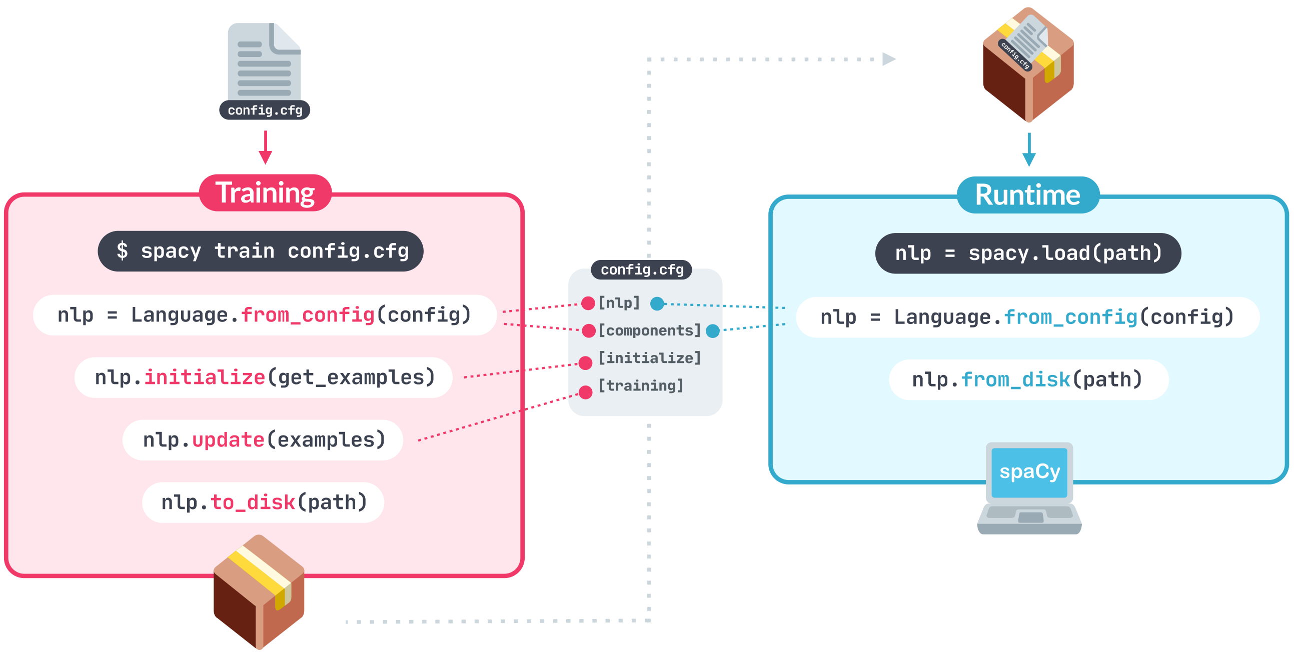
Training config files include all **settings and hyperparameters** for training your pipeline. Instead of providing lots of arguments on the command line, you only need to pass your `config.cfg` file to `spacy train` ≣ . This also makes it easy to integrate custom models and architectures, written in your framework of choice. A pipeline's `config.cfg` is considered the "single source of truth", both at **training** and **runtime**.

CONFIG.CFG (EXCERPT)

```
[training]
accumulate_gradient = 3

[training.optimizer]
@optimizers = "Adam.v1"

[training.optimizer.learn_rate]
@schedules = "warmup_linear.v1"
warmup_steps = 250
total_steps = 20000
initial_rate = 0.01
```



📖 **Training configuration system**

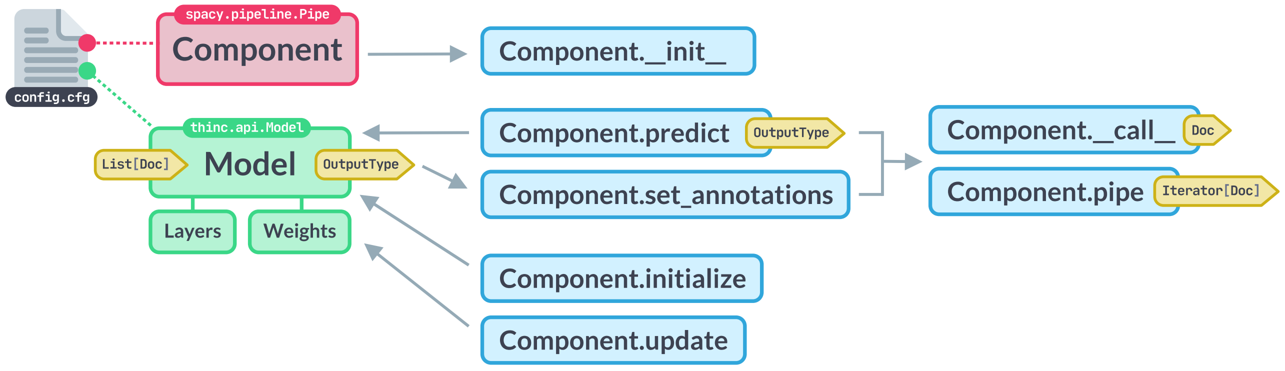
## Trainable components

spaCy's `Pipe` class helps you implement your own trainable components that have their own model instance, make predictions over `Doc` objects and can be updated using `spacy train`. This lets you plug fully custom machine learning components into your pipeline that can be configured via a single training config.

CONFIG.CFG (EXCERPT)

```
[components.my_component]
factory = "my_component"

[components.my_component.model]
@architectures = "my_model.v1"
width = 128
```



📖 **Custom trainable components**

## Language data

Every language is different – and usually full of **exceptions and special cases**, especially amongst the most common words. Some of these exceptions are shared across languages, while others are **entirely specific** – usually so specific that they need to be hard-coded. The `lang` module contains all language-specific data, organized in simple Python files. This makes the data easy to update and extend.

The **shared language data** in the directory root includes rules that can be generalized across languages – for example, rules for basic punctuation, emoji, emoticons and single-letter abbreviations. The **individual language data** in a submodule contains rules that are only relevant to a particular language. It also takes care of putting together all components and creating the `Language` subclass – for example, `English` or `German`. The values are defined in the `Language.Defaults`.

```python
from spacy.lang.en import English
from spacy.lang.de import German

nlp_en = English()  # Includes English data
nlp_de = German()  # Includes German data
```

| | |
|---|---|
| **Stop words** `stop_words.py` | List of most common words of a language that are often useful to filter out, for example "and" or "I". Matching tokens will return `True` for `is_stop`. |
| **Tokenizer exceptions** `tokenizer_exceptions.py` | Special-case rules for the tokenizer, for example, contractions like "can't" and abbreviations with punctuation, like "U.K.". |
| **Punctuation rules** `punctuation.py` | Regular expressions for splitting tokens, e.g. on punctuation or special characters like emoji. Includes rules for prefixes, suffixes and infixes. |
| **Character classes** `char_classes.py` | Character classes to be used in regular expressions, for example, Latin characters, quotes, hyphens or icons. |
| **Lexical attributes** `lex_attrs.py` | Custom functions for setting lexical attributes on tokens, e.g. `like_num`, which includes language-specific words like "ten" or "hundred". |
| **Syntax iterators** `syntax_iterators.py` | Functions that compute views of a `Doc` object based on its syntax. At the moment, only used for noun chunks. |
| **Lemmatizer** `lemmatizer.py` `spacy-lookups-data` | Custom lemmatizer implementation and lemmatization tables. |

# Community & FAQ

We're very happy to see the spaCy community grow and include a mix of people from all kinds of different backgrounds – computational linguistics, data science, deep learning, research and more. If you'd like to get involved, below are some answers to the most important questions and resources for further reading.

## Help, my code isn't working!

reported. If you're having installation or loading problems, make sure to also check out the troubleshooting guide. Help with spaCy is available via the following platforms:

HOW DO I KNOW IF SOMETHING IS A BUG?

Of course, it's always hard to know for sure, so don't worry – we're not going to be mad if a bug report turns out to be a typo in your code. As a simple rule, any C-level error without a Python traceback, like a **segmentation fault** or **memory error**, is **always** a spaCy bug.

Because models are statistical, their performance will never be *perfect*. However, if you come across **patterns that might indicate an underlying issue**, please do file a report. Similarly, we also care about behaviors that **contradict our docs**.

- Stack Overflow: **Usage questions** and everything related to problems with your specific code. The Stack Overflow community is much larger than ours, so if your problem can be solved by others, you'll receive help much quicker.
- GitHub discussions </> : **General discussion**, **project ideas** and **usage questions**. Meet other community members to get help with a specific code implementation, discuss ideas for new projects/plugins, support more languages, and share best practices.
- GitHub issue tracker </> : **Bug reports** and **improvement suggestions**, i.e. everything that's likely spaCy's fault. This also includes problems with the trained pipelines beyond statistical imprecisions, like patterns that point to a bug.

**Important note**

Please understand that we won't be able to provide individual support via email. We also believe that help is much more valuable if it's shared publicly, so that **more people can benefit from it**. If you come across an issue and you think you might be able to help, consider posting a quick update with your solution. No matter how simple, it can easily save someone a lot of time and headache – and the next time you need help, they might repay the favor.

## How can I contribute to spaCy?

You don't have to be an NLP expert or Python pro to contribute, and we're happy to help you get started. If you're new to spaCy, a good place to start is the `help wanted (easy)` label </> on GitHub, which we use to tag bugs and feature requests that are easy and self-contained. We also the source.

Another way of getting involved is to help us improve the language data – especially if you happen to speak one of the languages currently in alpha support. Even adding simple tokenizer exceptions, stop words or lemmatizer data can make a big difference. It will also make it easier for us to provide a trained pipeline for the language in the future. Submitting a test that documents a bug or performance issue, or covers functionality that's especially important for your application is also very helpful. This way, you'll also make sure we never accidentally introduce regressions to the parts of the library that you care about the most.

**For more details on the types of contributions we're looking for, the code conventions and other useful tips, make sure to check out the contributing guidelines </> .**

**Code of Conduct**

spaCy adheres to the Contributor Covenant Code of Conduct. By participating, you are expected to uphold this code.

## I've built something cool with spaCy – how can I get the word out?

First, congrats – we'd love to check it out! When you share your project on Twitter, don't forget to tag @spacy_io so we don't miss it. If you think your project would be a good fit for the spaCy Universe, **feel free to submit it!** Tutorials are also incredibly valuable to other users and a great way to get exposure. So we strongly encourage **writing up your experiences**, or sharing your code and some tips and tricks on your blog. Since our website is open-source, you can add your project or tutorial by making a pull request on GitHub.

If you would like to use the spaCy logo on your site, please get in touch and ask us first. However, if you want to show support and tell others that your project is using spaCy, you can grab one of our **spaCy badges** here:

built with spaCy

```
[![Built with spaCy](https://img.shields.io/badge/built%20with-spaCy-09a3d5.svg)](h
[![Made with love and spaCy](https://img.shields.io/badge/made%20with%20❤%20and-sp
```

</> SUGGEST EDITS

READ NEXT
New in v3.7 →