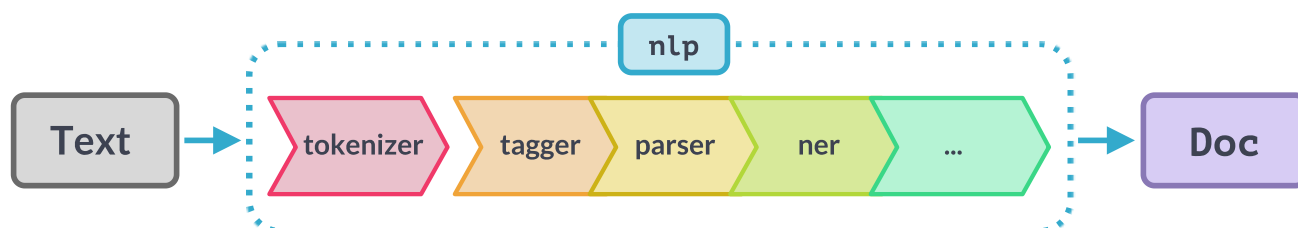


# Language Processing Pipelines

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 used by the [trained pipelines](#) typically include a tagger, a lemmatizer, a parser and an entity recognizer. Each pipeline component returns the processed `Doc`, which is then passed on to the next component.



NAME	COMPONENT	CREATES	DESCRIPTION
tokenizer	<a href="#">Tokenizer</a> ≡	<code>Doc</code>	Segment text into tokens.
PROCESSING PIPELINE			
tagger	<a href="#">Tagger</a> ≡	<code>Token.tag</code>	Assign part-of-speech tags.
parser	<a href="#">DependencyParser</a> ≡	<code>Token.head</code> , <code>Token.dep</code> , <code>Doc.sents</code> , <code>Doc.noun_chunks</code>	Assign dependency labels.
ner	<a href="#">EntityRecognizer</a> ≡	<code>Doc.ents</code> , <code>Token.ent_iob</code> , <code>Token.ent_type</code>	Detect and label named entities.
lemmatizer	<a href="#">Lemmatizer</a> ≡	<code>Token.lemma</code>	Assign base forms.
textcat	<a href="#">TextCategorizer</a> ≡	<code>Doc.cats</code>	Assign document labels.
custom	<a href="#">custom components</a>	<code>Doc._.xxx</code> , <code>Token._.xxx</code> , <code>Span._.xxx</code>	Assign custom attributes, methods or properties.

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](#):

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

Does the order of pipeline components matter?



Why is the tokenizer special?




---


## Processing text

When you call `nlp` on a text, spaCy will **tokenize** it and then **call each component** on the `Doc`, in order. It then returns the processed `Doc` that you can work with.

```
doc = nlp("This is a text")
```

When processing large volumes of text, the statistical models are usually more efficient if you let them work on batches of texts. spaCy's `nlp.pipe`  method takes an iterable of texts and yields processed `Doc` objects. The batching is done internally.

```
texts = ["This is a text", "These are lots of texts", "..."]
- docs = [nlp(text) for text in texts]
+ docs = list(nlp.pipe(texts))
```


In this example, we're using `nlp.pipe`  to process a (potentially very large) iterable of texts as a stream. Because we're only accessing the named entities in `doc.ents` (set by the `ner` component), we'll disable all other components during processing. `nlp.pipe` yields `Doc` objects, so we can iterate over them and access the named entity predictions:

```
import spacy

texts = [
    "Net income was $9.4 million compared to the prior year of $2.7 million.",
    "Revenue exceeded twelve billion dollars, with a loss of $1b.",
]

nlp = spacy.load("en_core_web_sm")
for doc in nlp.pipe(texts, disable=["tok2vec", "tagger", "parser", "attribute_ruler"]):
    # Do something with the doc here
    print([(ent.text, ent.label_) for ent in doc.ents])
```

RUN

You can use the `as_tuples` option to pass additional context along with each doc when using `nlp.pipe` . If `as_tuples` is `True`, then the input should be a sequence of `(text, context)` tuples and the output will be a sequence of `(doc, context)` tuples. For example, you can pass metadata in the context and save it in a [custom attribute](#):

```

import spacy
from spacy.tokens import Doc

if not Doc.has_extension("text_id"):
    Doc.set_extension("text_id", default=None)

text_tuples = [
    ("This is the first text.", {"text_id": "text1"}),
    ("This is the second text.", {"text_id": "text2"})
]

nlp = spacy.load("en_core_web_sm")
doc_tuples = nlp.pipe(text_tuples, as_tuples=True)


docs = []
for doc, context in doc_tuples:
    doc._.text_id = context["text_id"]
    docs.append(doc)

for doc in docs:
    print(f"{doc._.text_id}: {doc.text}")

```

RUN

## Multiprocessing

spaCy includes built-in support for multiprocessing with `nlp.pipe`  using the `n_process` option:

```

# Multiprocessing with 4 processes
docs = nlp.pipe(texts, n_process=4)

# With as many processes as CPUs (use with caution!)
docs = nlp.pipe(texts, n_process=-1)

```

Depending on your platform, starting many processes with multiprocessing can add a lot of overhead. In particular, the default start method `spawn` used in macOS/OS X (as of Python 3.8) and in Windows can be slow for larger models because the model data is copied in memory for each new process. See the [Python docs on multiprocessing](#) for further details.

For shorter tasks and in particular with `spawn`, it can be faster to use a smaller number of processes with a larger batch size. The optimal `batch_size` setting will depend on the pipeline components, the length of your documents, the number of processes and how much memory is available.

```
# Default batch size is `nlp.batch_size` (typically 1000)
docs = nlp.pipe(texts, n_process=2, batch_size=2000)
```

## Pipelines and built-in components

spaCy makes it very easy to create your own pipelines consisting of reusable components – this includes spaCy's default tagger, parser and entity recognizer, but also your own custom processing functions. A pipeline component can be added to an already existing `nlp` object, specified when initializing a `Language` class, or defined within a [pipeline package](#).


When you load a pipeline, spaCy first consults the [meta.json](#) and [config.cfg](#). The config tells spaCy what language class to use, which components are in the pipeline, and how those components should be created. spaCy will then do the following:

1. Load the **language class and data** for the given ID via `get_lang_class` and initialize it. The `Language` class contains the shared vocabulary, tokenization rules and the language-specific settings.
2. Iterate over the **pipeline names** and look up each component name in the `[components]` block. The `factory` tells spaCy which [component factory](#) to use for adding the component with `add_pipe`. The settings are passed into the factory.
3. Make the **model data** available to the `Language` class by calling `from_disk` with the path to the data directory.

So when you call this...

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

... the pipeline's `config.cfg` tells spaCy to use the language `"en"` and the pipeline `["tok2vec", "tagger", "parser", "ner", "attribute_ruler", "lemmatizer"]`. spaCy will then initialize `spacy.lang.en.English`, and create each pipeline component and add it to the processing pipeline. It'll then load in the model data from the data directory and return the modified `Language` class for you to use as the `nlp` object.

Fundamentally, a [spaCy pipeline package](#) consists of three components: **the weights**, i.e. binary data loaded in from a directory, a **pipeline** of functions called in order, and **language data** like the tokenization rules and language-specific settings. For example, a Spanish NER pipeline requires different weights, language data and components than an English parsing and tagging pipeline. This is also why the pipeline state is always held by the `Language` class. `spacy.load`  puts this all together and returns an instance of `Language` with a pipeline set and access to the binary data:

```
lang = "en"
pipeline = ["tok2vec", "tagger", "parser", "ner", "attribute_ruler", "lemmatizer"]
data_path = "path/to/en_core_web_sm/en_core_web_sm-3.0.0"

cls = spacy.util.get_lang_class(lang) # 1. Get Language class, e.g. English
nlp = cls()                          # 2. Initialize it
for name in pipeline:
    nlp.add_pipe(name, config={...}) # 3. Add the component to the pipeline
nlp.from_disk(data_path)            # 4. Load in the binary data
```

When you call `nlp` on a text, spaCy will **tokenize** it and then **call each component** on the `Doc`, in order. Since the model data is loaded, the components can access it to assign annotations to the `Doc` object, and subsequently to the `Token` and `Span` which are only views of the `Doc`, and don't own any data themselves. All components return the modified document, which is then processed by the next component in the pipeline.

```
doc = nlp.make_doc("This is a sentence") # Create a Doc from raw text
for name, proc in nlp.pipeline:          # Iterate over components in order
    doc = proc(doc)                      # Apply each component
```
















The current processing pipeline is available as `nlp.pipeline`, which returns a list of `(name, component)` tuples, or `nlp.pipe_names`, which only returns a list of human-readable component names.

```
print(nlp.pipeline)
# [('tok2vec', <spacy.pipeline.Tok2Vec>), ('tagger', <spacy.pipeline.Tagger>), ('parse
```

```
print(nlp.pipe_names)
# ['tok2vec', 'tagger', 'parser', 'ner', 'attribute_ruler', 'lemmatizer']
```

## Built-in pipeline components


spaCy has a number of built-in pipeline components that you can initialize them by calling `nlp.add_pipe` with their names and spaCy will know how to create them. See the [API documentation](#) for a full list of available pipeline components and component functions.


STRING NAME	COMPONENT	DESCRIPTION
<code>tagger</code>	<code>Tagger</code> 	Assign part-of-speech-tags.
<code>parser</code>	<code>DependencyParser</code> 	Assign dependency labels.
<code>ner</code>	<code>EntityRecognizer</code> 	Assign named entities.
<code>entity_linker</code>	<code>EntityLinker</code> 	Assign knowledge base IDs to named entities. Should be added after the entity recognizer.
<code>entity_ruler</code>	<code>EntityRuler</code> 	Assign named entities based on pattern rules and dictionaries.
<code>textcat</code>	<code>TextCategorizer</code> 	Assign text categories: exactly one category is predicted per document.
<code>textcat_multilabel</code>	<code>MultiLabel_TextCategorizer</code> 	Assign text categories in a multi-label setting: zero, one or more labels per document.
<code>lemmatizer</code>	<code>Lemmatizer</code> 	Assign base forms to words using rules and lookups.
<code>trainable_lemmatizer</code>	<code>EditTreeLemmatizer</code> 	Assign base forms to words.
<code>morphologizer</code>	<code>Morphologizer</code> 	Assign morphological features and coarse-grained POS tags.
<code>attribute_ruler</code>	<code>AttributeRuler</code> 	Assign token attribute mappings and rule-based exceptions.
<code>senter</code>	<code>SentenceRecognizer</code> 	Assign sentence boundaries.
<code>sentencizer</code>	<code>Sentencizer</code> 	Add rule-based sentence segmentation without the dependency parse.
<code>tok2vec</code>	<code>Tok2Vec</code> 	Assign token-to-vector embeddings.
<code>transformer</code>	<code>Transformer</code> 	Assign the tokens and outputs of a transformer model.

## Disabling, excluding and modifying components



If you don't need a particular component of the pipeline – for example, the tagger or the parser, you can **disable or exclude** it. This can sometimes make a big difference and improve loading and inference speed. There are two different mechanisms you can use:

1. **Disable:** The component and its data will be loaded with the pipeline, but it will be disabled by default and not run as part of the processing pipeline. To run it, you can explicitly enable it by calling `nlp.enable_pipe` . When you save out the `nlp` object, the disabled component will be included but disabled by default.
2. **Exclude:** Don't load the component and its data with the pipeline. Once the pipeline is loaded, there will be no reference to the excluded component.


Disabled and excluded component names can be provided to `spacy.load`  as a list.

```
# Load the pipeline without the entity recognizer
nlp = spacy.load("en_core_web_sm", exclude=["ner"])

# Load the tagger and parser but don't enable them
nlp = spacy.load("en_core_web_sm", disable=["tagger", "parser"])
# Explicitly enable the tagger later on
nlp.enable_pipe("tagger")
```

In addition to `disable`, `spacy.load()` also accepts `enable`. If `enable` is set, all components except for those in `enable` are disabled. If `enable` and `disable` conflict (i.e. the same component is included in both), an error is raised.

```
# Load the complete pipeline, but disable all components except for tok2vec and tagger
nlp = spacy.load("en_core_web_sm", enable=["tok2vec", "tagger"])
# Has the same effect, as NER is already not part of enabled set of components
nlp = spacy.load("en_core_web_sm", enable=["tok2vec", "tagger"], disable=["ner"])
# Will raise an error, as the sets of enabled and disabled components are conflicting
nlp = spacy.load("en_core_web_sm", enable=["ner"], disable=["ner"])
```


As a shortcut, you can use the `nlp.select_pipes`  context manager to temporarily disable certain components for a given block. At the end of the `with` block, the disabled pipeline components will be restored automatically. Alternatively, `select_pipes` returns an object that lets you call its `restore()` method to restore the disabled components when needed. This can be useful if you want to prevent unnecessary code indentation of large blocks.

```
# 1. Use as a context manager
with nlp.select_pipes(disable=["tagger", "parser", "lemmatizer"]):
    doc = nlp("I won't be tagged and parsed")
doc = nlp("I will be tagged and parsed")




# 2. Restore manually
disabled = nlp.select_pipes(disable="ner")
doc = nlp("I won't have named entities")
disabled.restore()
```

If you want to disable all pipes except for one or a few, you can use the `enable` keyword. Just like the `disable` keyword, it takes a list of pipe names, or a string defining just one pipe.


```
# Enable only the parser
with nlp.select_pipes(enable="parser"):
    doc = nlp("I will only be parsed")
```

The `nlp.pipe`  method also supports a `disable` keyword argument if you only want to disable components during processing:

```
for doc in nlp.pipe(texts, disable=["tagger", "parser", "lemmatizer"]):
    # Do something with the doc here
```


Finally, you can also use the `remove_pipe`  method to remove pipeline components from an existing pipeline, the `rename_pipe`  method to rename them, or the `replace_pipe`  method to replace them with a custom component entirely (more details on this in the section on [custom components](#)).

```
nlp.remove_pipe("parser")
nlp.rename_pipe("ner", "entityrecognizer")
nlp.replace_pipe("tagger", "my_custom_tagger")
```

The `Language` object exposes different `attributes`  that let you inspect all available components and the components that currently run as part of the pipeline.

NAME	DESCRIPTION
<code>nlp.pipeline</code>	<code>(name, component)</code> tuples of the processing pipeline, in order.
<code>nlp.pipe_names</code>	Pipeline component names, in order.
<code>nlp.components</code>	All <code>(name, component)</code> tuples, including disabled components.
<code>nlp.component_names</code>	All component names, including disabled components.
<code>nlp.disabled</code>	Names of components that are currently disabled.

## Sourcing components from existing pipelines V3.0 ?

Pipeline components that are independent can also be reused across pipelines. Instead of adding a new blank component, you can also copy an existing component from a trained pipeline by setting the `source` argument on `nlp.add_pipe` . The first argument will then be interpreted as the name of the component in the source pipeline – for instance, `"ner"`. This is especially useful for [training a pipeline](#) because it lets you mix and match components and create fully custom pipeline packages with updated trained components and new components trained on your data.

### Editable Code

spaCy v3.7 · Python 3 · via Binder




```
import spacy

# The source pipeline with different components
source_nlp = spacy.load("en_core_web_sm")
print(source_nlp.pipe_names)

# Add only the entity recognizer to the new blank pipeline
nlp = spacy.blank("en")
nlp.add_pipe("ner", source=source_nlp)
print(nlp.pipe_names)
```

RUN

## Analyzing pipeline components V3.0 ?

The `nlp.analyze_pipes`  method analyzes the components in the current pipeline and outputs information about them like the attributes they set on the `Doc`  and `Token` , whether they retokenize the `Doc` and which scores they produce during training. It will also show warnings if components require values that aren't set by previous component – for instance, if the entity linker is used but no component that runs before it sets named entities. Setting `pretty=True` will pretty-print a table instead of only returning the structured data.

#### Editable Code

spaCy v3.7 · Python 3 · via Binder

```
import spacy

nlp = spacy.blank("en")
nlp.add_pipe("tagger")
# This is a problem because it needs entities and sentence boundaries
nlp.add_pipe("entity_linker")
analysis = nlp.analyze_pipes(pretty=True)
```

RUN



#### Example output





## Creating custom pipeline components

A pipeline component is a function that receives a `Doc` object, modifies it and returns it – for example, by using the current weights to make a prediction and set some annotation on the document. By adding a component to the pipeline, you'll get access to the `Doc` at any point **during processing** – instead of only being able to modify it afterwards.

ARGUMENT	TYPE	DESCRIPTION
----------	------	-------------

<code>doc</code>	<code>Doc</code> 	The <code>Doc</code> object processed by the previous component.
RETURNS	<code>Doc</code> 	The <code>Doc</code> object processed by this pipeline component.


The `@Language.component`  decorator lets you turn a simple function into a pipeline component. It takes at least one argument, the **name** of the component factory. You can use this name to add an instance of your component to the pipeline. It can also be listed in your pipeline config, so you can save, load and train pipelines using your component.

Custom components can be added to the pipeline using the `add_pipe`  method. Optionally, you can either specify a component to add it **before or after**, tell spaCy to add it **first or last** in the pipeline, or define a **custom name**. If no name is set and no `name` attribute is present on your component, the function name is used.

## ARGUMENT DESCRIPTION

<code>last</code>	If set to <code>True</code> , component is added <b>last</b> in the pipeline (default).  TYPE: <code>bool</code>
<code>first</code>	If set to <code>True</code> , component is added <b>first</b> in the pipeline.  TYPE: <code>bool</code>
<code>before</code>	String name or index to add the new component <b>before</b> .  TYPE: <code>Union[str, int]</code>
<code>after</code>	String name or index to add the new component <b>after</b> .  TYPE: <code>Union[str, int]</code>

## Examples: Simple stateless pipeline components

The following component receives the `Doc` in the pipeline and prints some information about it: the number of tokens, the part-of-speech tags of the tokens and a conditional message based on the document length. The `@Language.component`  decorator lets you register the component under the name `"info_component"`.

### Editable Code

spaCy v3.7 · Python 3 · via Binder

```
import spacy
from spacy.language import Language

@Language.component("info_component")
def my_component(doc):
    print(f"After tokenization, this doc has {len(doc)} tokens.")
    print("The part-of-speech tags are:", [token.pos_ for token in doc])
    if len(doc) < 10:
        print("This is a pretty short document.")
```

```

    return doc

nlp = spacy.load("en_core_web_sm")
nlp.add_pipe("info_component", name="print_info", last=True)
print(nlp.pipe_names) # ['tagger', 'parser', 'ner', 'print_info']
doc = nlp("This is a sentence.")

```

RUN

Here's another example of a pipeline component that implements custom logic to improve the sentence boundaries set by the dependency parser. The custom logic should therefore be applied **after** tokenization, but *before* the dependency parsing – this way, the parser can also take advantage of the sentence boundaries.

Editable Code

spaCy v3.7 · Python 3 · via Binder

```

import spacy
from spacy.language import Language


@Language.component("custom_sentencizer")
def custom_sentencizer(doc):
    for i, token in enumerate(doc[:-2]):
        # Define sentence start if pipe + titlecase token
        if token.text == "|" and doc[i + 1].is_title:
            doc[i + 1].is_sent_start = True
        else:
            # Explicitly set sentence start to False otherwise, to tell
            # the parser to leave those tokens alone
            doc[i + 1].is_sent_start = False
    return doc

nlp = spacy.load("en_core_web_sm")
nlp.add_pipe("custom_sentencizer", before="parser") # Insert before the parser
doc = nlp("This is. A sentence. | This is. Another sentence.")
for sent in doc.sents:
    print(sent.text)

```



RUN

## Component factories and stateful components

Component factories are callables that take settings and return a **pipeline component function**. This is useful if your component is stateful and if you need to customize their creation, or if you need access to the current `nlp` object or the shared vocab. Component factories can be registered using the `@Language.factory`  decorator and they need at least **two named arguments** that are filled in automatically when the component is added to the pipeline:

#### ARGUMENT DESCRIPTION

<code>nlp</code>	The current <code>nlp</code> object. Can be used to access the shared vocab.  TYPE: <code>Language</code>
<code>name</code>	The <b>instance name</b> of the component in the pipeline. This lets you identify different instances of the same component.  TYPE: <code>str</code>

All other settings can be passed in by the user via the `config` argument on `nlp.add_pipe` . The `@Language.factory`  decorator also lets you define a `default_config` that's used as a fallback.

```
import spacy
from spacy.language import Language

@Language.factory("my_component", default_config={"some_setting": True})
def my_component(nlp, name, some_setting: bool):
    return MyComponent(some_setting=some_setting)

nlp = spacy.blank("en")
nlp.add_pipe("my_component", config={"some_setting": False})
```

How is `@Language.factory` different from `@Language.component`? 

Can I add the `@Language.factory` decorator to a class? 

## Language-specific factories V3.0

There are many use cases where you might want your pipeline components to be language-specific. Sometimes this requires entirely different implementation per language, sometimes the only difference is in the settings or data. spaCy allows you to register factories of the **same name** on both the `Language` base class, as well as its **subclasses** like `English` or `German`. Factories are resolved starting with the

specific subclass. If the subclass doesn't define a component of that name, spaCy will check the `Language` base class.

Here's an example of a pipeline component that overwrites the normalized form of a token, the `Token.norm_` with an entry from a language-specific lookup table. It's registered twice under the name `"token_normalizer"` – once using `@English.factory` and once using `@German.factory`:

#### Editable Code

spaCy v3.7 · Python 3 · via Binder

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

class TokenNormalizer:
    def __init__(self, norm_table):
        self.norm_table = norm_table

    def __call__(self, doc):
        for token in doc:
            # Overwrite the token.norm_ if there's an entry in the data
            token.norm_ = self.norm_table.get(token.text, token.norm_)
        return doc

@English.factory("token_normalizer")
def create_en_normalizer(nlp, name):
    return TokenNormalizer({"realise": "realize", "colour": "color"})

@German.factory("token_normalizer")
def create_de_normalizer(nlp, name):
    return TokenNormalizer({"daß": "dass", "wußte": "wusste"})

nlp_en = English()
nlp_en.add_pipe("token_normalizer") # uses the English factory
print([token.norm_ for token in nlp_en("realise colour daß wußte")])


nlp_de = German()
nlp_de.add_pipe("token_normalizer") # uses the German factory
print([token.norm_ for token in nlp_de("realise colour daß wußte")])
```

RUN

## Example: Stateful component with settings

This example shows a **stateful** pipeline component for handling acronyms: based on a dictionary, it will detect acronyms and their expanded forms in both directions and add them to a list as the custom



`doc._.acronyms` [extension attribute](#). Under the hood, it uses the `PhraseMatcher`  to find instances of the phrases.

The factory function takes three arguments: the shared `nlp` object and component instance `name`, which are passed in automatically by spaCy, and a `case_sensitive` config setting that makes the matching and acronym detection case-sensitive.

#### Editable Code

spaCy v3.7 · Python 3 · via Binder

```
from spacy.language import Language
from spacy.tokens import Doc
from spacy.matcher import PhraseMatcher
import spacy

DICTIONARY = {"lol": "laughing out loud", "brb": "be right back"}
DICTIONARY.update({value: key for key, value in DICTIONARY.items()})

@Language.factory("acronyms", default_config={"case_sensitive": False})
def create_acronym_component(nlp: Language, name: str, case_sensitive: bool):
    return AcronymComponent(nlp, case_sensitive)

class AcronymComponent:
    def __init__(self, nlp: Language, case_sensitive: bool):
        # Create the matcher and match on Token.lower if case-insensitive
        matcher_attr = "TEXT" if case_sensitive else "LOWER"
        self.matcher = PhraseMatcher(nlp.vocab, attr=matcher_attr)
        self.matcher.add("ACRONYMS", [nlp.make_doc(term) for term in DICTIONARY])
        self.case_sensitive = case_sensitive
        # Register custom extension on the Doc
        if not Doc.has_extension("acronyms"):
            Doc.set_extension("acronyms", default=[])

    def __call__(self, doc: Doc) -> Doc:
        # Add the matched spans when doc is processed
        for _, start, end in self.matcher(doc):
            span = doc[start:end]
            acronym = DICTIONARY.get(span.text if self.case_sensitive else span.text.lower())
            doc._.acronyms.append((span, acronym))
        return doc

# Add the component to the pipeline and configure it
nlp = spacy.blank("en")
nlp.add_pipe("acronyms", config={"case_sensitive": False})

# Process a doc and see the results
doc = nlp("LOL, be right back")
print(doc._.acronyms)
```


# Initializing and serializing component data

Many stateful components depend on **data resources** like dictionaries and lookup tables that should ideally be **configurable**. For example, it makes sense to make the `DICTIONARY` in the above example an argument of the registered function, so the `AcronymComponent` can be re-used with different data. One logical solution would be to make it an argument of the component factory, and allow it to be initialized with different dictionaries.

```
@Language.factory("acronyms", default_config={"data": {}, "case_sensitive": False})
def create_acronym_component(nlp: Language, name: str, data: Dict[str, str], case_sens
# 🚨 Problem: data ends up in the config file
return AcronymComponent(nlp, data, case_sensitive)
```

However, passing in the dictionary directly is problematic, because it means that if a component saves out its config and settings, the `config.cfg` will include a dump of the entire data, since that's the config the component was created with. It will also fail if the data is not JSON-serializable.

## Option 1: Using a registered function

If what you're passing in isn't JSON-serializable – e.g. a custom object like a `model` – saving out the component config becomes impossible because there's no way for spaCy to know *how* that object was created, and what to do to create it again. This makes it much harder to save, load and train custom pipelines with custom components. A simple solution is to **register a function** that returns your resources. The `registry`  lets you **map string names to functions** that create objects, so given a name and optional arguments, spaCy will know how to recreate the object. To register a function that returns your custom dictionary, you can use the `@spacy.registry.misc` decorator with a single argument, the name:

```
@spacy.registry.misc("acronyms.slang_dict.v1")
def create_acronyms_slang_dict():
    dictionary = {"lol": "laughing out loud", "brb": "be right back"}
```


```
dictionary.update({value: key for key, value in dictionary.items()})
return dictionary
```

In your `default_config` (and later in your [training config](#)), you can now refer to the function registered under the name `"acronyms.slang_dict.v1"` using the `@misc` key. This tells spaCy how to create the value, and when your component is created, the result of the registered function is passed in as the key `"dictionary"`.

```
- default_config = {"dictionary:" DICTIONARY}
+ default_config = {"dictionary": {"@misc": "acronyms.slang_dict.v1"}}
```

Using a registered function also means that you can easily include your custom components in pipelines that you [train](#). To make sure spaCy knows where to find your custom `@misc` function, you can pass in a Python file via the argument `--code`. If someone else is using your component, all they have to do to customize the data is to register their own function and swap out the name. Registered functions can also take **arguments**, by the way, that can be defined in the config as well – you can read more about this in the docs on [training with custom code](#).

## Option 2: Save data with the pipeline and load it in once on initialization

Just like models save out their binary weights when you call `nlp.to_disk` , components can also **serialize** any other data assets – for instance, an acronym dictionary. If a pipeline component implements its own `to_disk` and `from_disk` methods, those will be called automatically by `nlp.to_disk` and will receive the path to the directory to save to or load from. The component can then perform any custom saving or loading. If a user makes changes to the component data, they will be reflected when the `nlp` object is saved. For more examples of this, see the usage guide on [serialization methods](#).


```
import srsly
from spacy.util import ensure_path

class AcronymComponent:
    # other methods here...


    def to_disk(self, path, exclude=tuple()):
        path = ensure_path(path)
        if not path.exists():
            path.mkdir()
        srsly.write_json(path / "data.json", self.data)
```

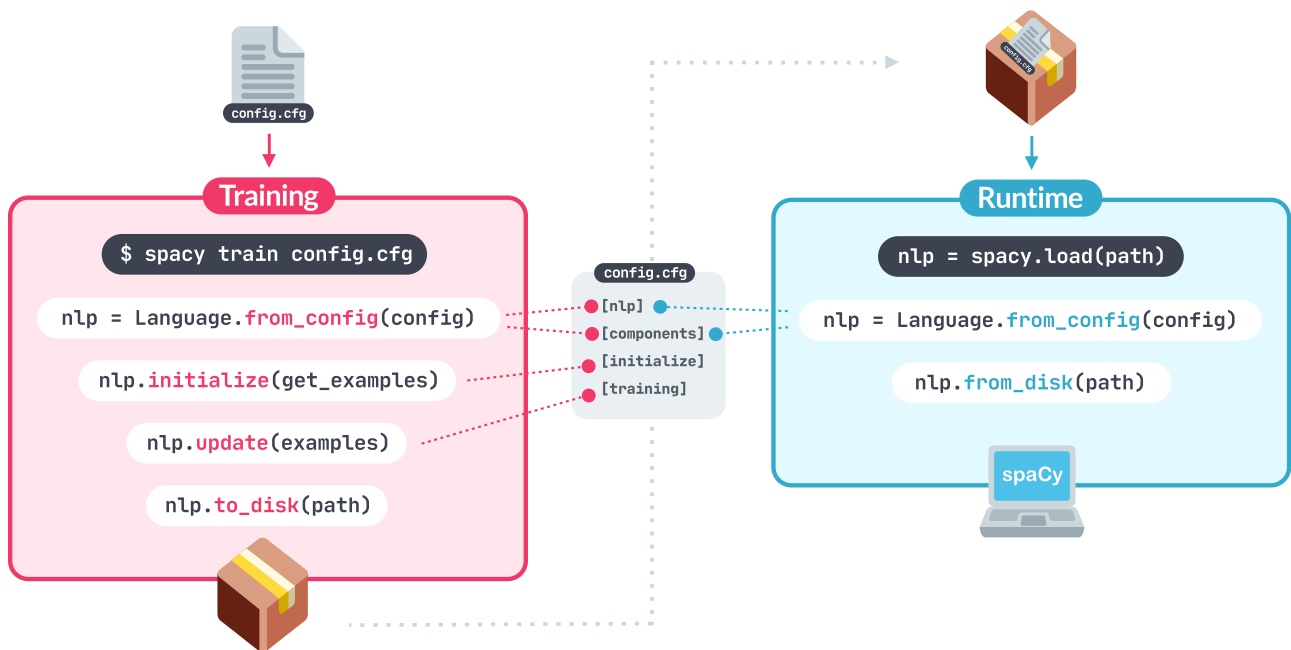
```
def from_disk(self, path, exclude=tuple()):
    self.data = srsly.read_json(path / "data.json")
    return self
```

Now the component can save to and load from a directory. The only remaining question: How do you **load in the initial data**? In Python, you could just call the pipe's `from_disk` method yourself. But if you're adding the component to your [training config](#), spaCy will need to know how to set it up, from start to finish, including the data to initialize it with.

While you could use a registered function or a file loader like `srsly.read_json.v1`  as an argument of the component factory, this approach is problematic: the component factory runs **every time the component is created**. This means it will run when creating the `nlp` object before training, but also every time a user loads your pipeline. So your runtime pipeline would either depend on a local path on your file system, or it's loaded twice: once when the component is created, and then again when the data is by `from_disk`.

```
@Language.factory("acronyms", default_config={"data": {}, "case_sensitive": False})
def create_acronym_component(nlp: Language, name: str, data: Dict[str, str], case_sens
# 🚨 Problem: data will be loaded every time component is created
return AcronymComponent(nlp, data, case_sensitive)
```

To solve this, your component can implement a separate method, `initialize`, which will be called by `nlp.initialize`  if available. This typically happens before training, but not at runtime when the pipeline is loaded. For more background on this, see the usage guides on the [config lifecycle](#) and [custom initialization](#).



A component's `initialize` method needs to take at least **two named arguments**: a `get_examples` callback that gives it access to the training examples, and the current `nlp` object. This is mostly used by trainable components so they can initialize their models and label schemes from the data, so we can ignore those arguments here. All **other arguments** on the method can be defined via the config – in this case a dictionary `data`.

```
class AcronymComponent:
    def __init__(self):
        self.data = {}

    def initialize(self, get_examples=None, nlp=None, data={}):
        self.data = data
```

When `nlp.initialize` runs before training (or when you call it in your own code), the `[initialize]` block of the config is loaded and used to construct the `nlp` object. The custom acronym component will then be passed the data loaded from the JSON file. After training, the `nlp` object is saved to disk, which will run the component's `to_disk` method. When the pipeline is loaded back into spaCy later to use it, the `from_disk` method will load the data back in.

## Python type hints and validation

V3.0 ?

spaCy's configs are powered by our machine learning library Thinc's [configuration system](#), which supports [type hints](#) and even [advanced type annotations](#) using `pydantic` `</>`. If your component factory provides type hints, the values that are passed in will be **checked against the expected types**. If the value can't be cast to an integer, spaCy will raise an error. `pydantic` also provides strict types like `StrictFloat`, which will force the value to be an integer and raise an error if it's not – for instance, if your config defines a float.

The following example shows a custom pipeline component for debugging. It can be added anywhere in the pipeline and logs information about the `nlp` object and the `Doc` that passes through. The `log_level` config setting lets the user customize what log statements are shown – for instance, `"INFO"` will show info logs and more critical logging statements, whereas `"DEBUG"` will show everything. The value is annotated as a `StrictStr`, so it will only accept a string value.

#### Editable Code

spaCy v3.7 · Python 3 · via Binder

```
import spacy
from spacy.language import Language
from spacy.tokens import Doc
from pydantic import StrictStr
import logging

@Language.factory("debug", default_config={"log_level": "DEBUG"})
class DebugComponent:
    def __init__(self, nlp: Language, name: str, log_level: StrictStr):
        self.logger = logging.getLogger(f"spacy.{name}")
        self.logger.setLevel(log_level)
        self.logger.info(f"Pipeline: {nlp.pipe_names}")

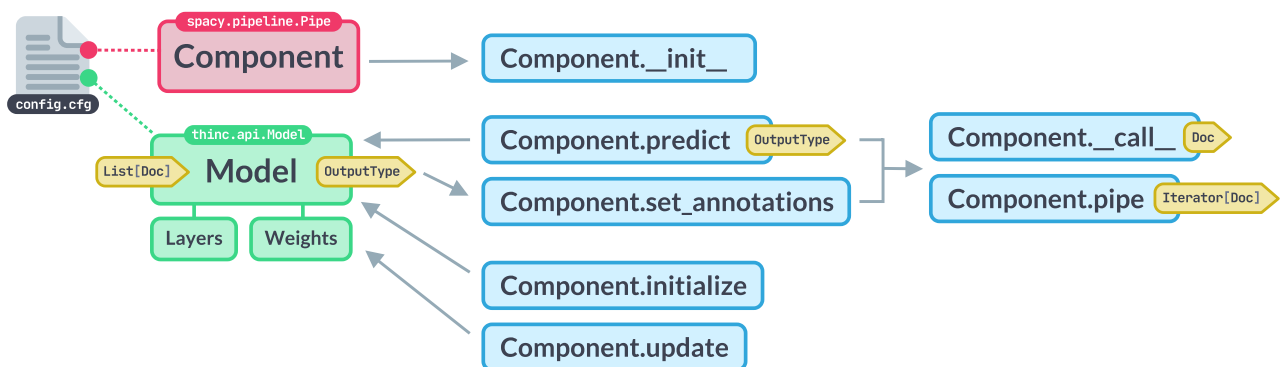
    def __call__(self, doc: Doc) -> Doc:
        is_tagged = doc.has_annotation("TAG")
        self.logger.debug(f"Doc: {len(doc)} tokens, is tagged: {is_tagged}")
        return doc

nlp = spacy.load("en_core_web_sm")
nlp.add_pipe("debug", config={"log_level": "DEBUG"})
doc = nlp("This is a text...")
```

RUN

## Trainable components V3.0 ?

spaCy's `TrainablePipe` 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.



You'll need the following:

1. **Model:** A Thinc `Model` instance. This can be a model implemented in `Thinc`, or a `wrapped model` implemented in PyTorch, TensorFlow, MXNet or a fully custom solution. The model must take a list of `Doc` objects as input and can have any type of output.
2. **TrainablePipe subclass:** A subclass of `TrainablePipe` that implements at least two methods: `TrainablePipe.predict` and `TrainablePipe.set_annotations`.
3. **Component factory:** A component factory registered with `@Language.factory` that takes the `nlp` object and component `name` and optional settings provided by the config and returns an instance of your trainable component.











NAME	DESCRIPTION
<code>predict</code>	Apply the component's model to a batch of <code>Doc</code> objects (without modifying them) and return the scores.
<code>set_annotations</code>	Modify a batch of <code>Doc</code> objects, using pre-computed scores generated by <code>predict</code> .

By default, `TrainablePipe.__init__` takes the shared vocab, the `Model` and the name of the component instance in the pipeline, which you can use as a key in the losses. All other keyword arguments will become available as `TrainablePipe.cfg` and will also be serialized with the component.

**Why components should be passed a `Model` instance, not create it**



For some use cases, it makes sense to also overwrite additional methods to customize how the model is updated from examples, how it's initialized, how the loss is calculated and to add evaluation scores to the training output.

NAME	DESCRIPTION
<code>update</code> 	Learn from a batch of <code>Example</code>  objects containing the predictions and gold-standard annotations, and update the component's model.
<code>initialize</code> 	Initialize the model. Typically calls into <code>Model.initialize</code> and can be passed custom arguments via the <code>[initialize]</code>  config block that are only loaded during training or when you call <code>nlp.initialize</code>  , not at runtime.
<code>get_loss</code> 	Return a tuple of the loss and the gradient for a batch of <code>Example</code>  objects.
<code>score</code> 	Score a batch of <code>Example</code>  objects and return a dictionary of scores. The <code>@Language.factory</code>  decorator can define the <code>default_score_weights</code> of the component to decide which keys of the scores to display during training and how they count towards the final score.

## Extension attributes

spaCy allows you to set any custom attributes and methods on the `Doc`, `Span` and `Token`, which become available as `Doc._`, `Span._` and `Token._` – for example, `Token._.my_attr`. This lets you store additional information relevant to your application, add new features and functionality to spaCy, and implement your own models trained with other machine learning libraries. It also lets you take advantage of spaCy's data structures and the `Doc` object as the “single source of truth”.




Why `._` and not just a top-level attribute?



How is the `._` implemented?





There are three main types of extensions, which can be defined using the `Doc.set_extension`  , `Span.set_extension`  and `Token.set_extension`  methods.

---

# Description

1.

**Attribute extensions.** Set a default value for an attribute, which can be overwritten manually at any time. Attribute extensions work like “normal” variables and are the quickest way to store arbitrary information on a `Doc` , `Span` or `Token` .

```
Doc.set_extension("hello", default=True)
assert doc._.hello
doc._.hello = False
```

2.

**Property extensions.** Define a getter and an optional setter function. If no setter is provided, the extension is immutable. Since the getter and setter functions are only called when you *retrieve* the attribute, you can also access values of previously added attribute extensions. For example, a `Doc` getter can average over `Token` attributes. For `Span` extensions, you’ll almost always want to use a property – otherwise, you’d have to write to *every possible* `Span` in the `Doc` to set up the values correctly.

```
Doc.set_extension("hello", getter=get_hello_value, setter=set_hello_value)
assert doc._.hello
doc._.hello = "Hi!"
```

3.

**Method extensions.** Assign a function that becomes available as an object method. Method extensions are always immutable. For more details and implementation ideas, see [these examples](#).

```
Doc.set_extension("hello", method=lambda doc, name: f"Hi {name}!")
```

```
assert doc._.hello("Bob") == "Hi Bob!"
```

Before you can access a custom extension, you need to register it using the `set_extension` method on the object you want to add it to, e.g. the `Doc`. Keep in mind that extensions are always **added globally** and not just on a particular instance. If an attribute of the same name already exists, or if you're trying to access an attribute that hasn't been registered, spaCy will raise an `AttributeError`.

```
from spacy.tokens import Doc, Span, Token

fruits = ["apple", "pear", "banana", "orange", "strawberry"]
is_fruit_getter = lambda token: token.text in fruits
has_fruit_getter = lambda obj: any([t.text in fruits for t in obj])

Token.set_extension("is_fruit", getter=is_fruit_getter)
Doc.set_extension("has_fruit", getter=has_fruit_getter)
Span.set_extension("has_fruit", getter=has_fruit_getter)
```

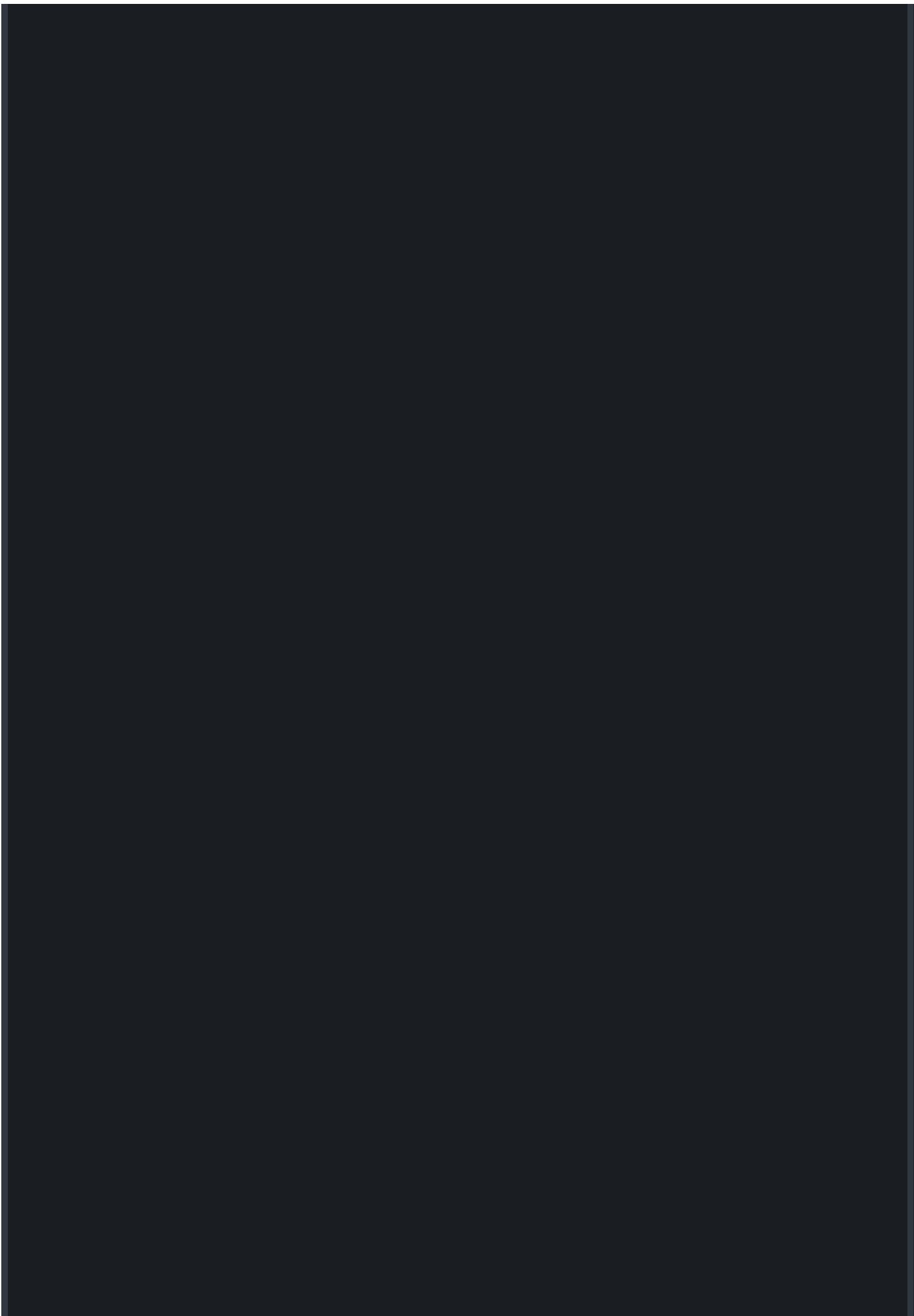
Once you've registered your custom attribute, you can also use the built-in `set`, `get` and `has` methods to modify and retrieve the attributes. This is especially useful if you want to pass in a string instead of calling `doc._.my_attr`.

## Example: Pipeline component for GPE entities and country meta data via a REST API

This example shows the implementation of a pipeline component that fetches country meta data via the [REST Countries API](#), sets entity annotations for countries and sets custom attributes on the `Doc` and `Span` – for example, the capital, latitude/longitude coordinates and even the country flag.

Editable Code

spaCy v3.7 · Python 3 · via Binder



```

import requests
from spacy.lang.en import English
from spacy.language import Language
from spacy.matcher import PhraseMatcher
from spacy.tokens import Doc, Span, Token

@Language.factory("rest_countries")
class RESTCountriesComponent:
    def __init__(self, nlp, name, label="GPE"):
        r = requests.get("https://restcountries.com/v2/all")
        r.raise_for_status() # make sure requests raises an error if it fails
        countries = r.json()
        # Convert API response to dict keyed by country name for easy lookup
        self.countries = {c["name"]: c for c in countries}
        self.label = label
        # Set up the PhraseMatcher with Doc patterns for each country name
        self.matcher = PhraseMatcher(nlp.vocab)
        self.matcher.add("COUNTRIES", [nlp.make_doc(c) for c in self.countries.keys()])
        # Register attributes on the Span. We'll be overwriting this based on
        # the matches, so we're only setting a default value, not a getter.
        Span.set_extension("is_country", default=None)
        Span.set_extension("country_capital", default=None)
        Span.set_extension("country_latlng", default=None)
        Span.set_extension("country_flag", default=None)
        # Register attribute on Doc via a getter that checks if the Doc
        # contains a country entity
        Doc.set_extension("has_country", getter=self.has_country)

    def __call__(self, doc):
        spans = [] # keep the spans for later so we can merge them afterwards
        for _, start, end in self.matcher(doc):
            # Generate Span representing the entity & set label
            entity = Span(doc, start, end, label=self.label)
            # Set custom attributes on entity. Can be extended with other data
            # returned by the API, like currencies, country code, calling code etc.
            entity._.set("is_country", True)
            entity._.set("country_capital", self.countries[entity.text]["capital"])
            entity._.set("country_latlng", self.countries[entity.text]["latlng"])
            entity._.set("country_flag", self.countries[entity.text]["flag"])
            spans.append(entity)
        # Overwrite doc.ents and add entity - be careful not to replace!
        doc.ents = list(doc.ents) + spans
        return doc # don't forget to return the Doc!

    def has_country(self, doc):
        """Getter for Doc attributes. Since the getter is only called
        when we access the attribute, we can refer to the Span's 'is_country'
        attribute here, which is already set in the processing step."""
        return any([entity._.get("is_country") for entity in doc.ents])

nlp = English()
nlp.add_pipe("rest_countries", config={"label": "GPE"})
doc = nlp("Some text about Colombia and the Czech Republic")
print("Pipeline", nlp.pipe_names) # pipeline contains component name

```

```
print("Doc has countries", doc._.has_country) # Doc contains countries
for ent in doc.ents:
    if ent._.is_country:
        print(ent.text, ent.label_, ent._.country_capital, ent._.country_latlng, ent
```

RUN

In this case, all data can be fetched on initialization in one request. However, if you're working with text that contains incomplete country names, spelling mistakes or foreign-language versions, you could also implement a `like_country`-style getter function that makes a request to the search API endpoint and returns the best-matching result.

## User hooks

While it's generally recommended to use the `Doc._`, `Span._` and `Token._` proxies to add your own custom attributes, spaCy offers a few exceptions to allow **customizing the built-in methods** like `Doc.similarity` or `Doc.vector` with your own hooks, which can rely on components you train yourself. For instance, you can provide your own on-the-fly sentence segmentation algorithm or document similarity method.

Hooks let you customize some of the behaviors of the `Doc`, `Span` or `Token` objects by adding a component to the pipeline. For instance, to customize the `Doc.similarity` method, you can add a component that sets a custom function to `doc.user_hooks["similarity"]`. The built-in `Doc.similarity` method will check the `user_hooks` dict, and delegate to your function if you've set one. Similar results can be achieved by setting functions to `Doc.user_span_hooks` and `Doc.user_token_hooks`.

NAME	CUSTOMIZES
<code>user_hooks</code>	<code>Doc.similarity</code> , <code>Doc.vector</code> , <code>Doc.has_vector</code> , <code>Doc.vector_norm</code> , <code>Doc.sents</code>
<code>user_token_hooks</code>	<code>Token.similarity</code> , <code>Token.vector</code> , <code>Token.has_vector</code> , <code>Token.vector_norm</code> , <code>Token.conjuncts</code>
<code>user_span_hooks</code>	<code>Span.similarity</code> , <code>Span.vector</code> , <code>Span.has_vector</code> , <code>Span.vector_norm</code> , <code>Span.root</code>

```

from spacy.language import Language

class SimilarityModel:
    def __init__(self, name: str, index: int):
        self.name = name
        self.index = index

    def __call__(self, doc):
        doc.user_hooks["similarity"] = self.similarity
        doc.user_span_hooks["similarity"] = self.similarity
        doc.user_token_hooks["similarity"] = self.similarity
        return doc

    def similarity(self, obj1, obj2):
        return obj1.vector[self.index] + obj2.vector[self.index]

@Language.factory("similarity_component", default_config={"index": 0})
def create_similarity_component(nlp, name, index: int):
    return SimilarityModel(name, index)

```

# Developing plugins and wrappers

We're very excited about all the new possibilities for community extensions and plugins in spaCy, and we can't wait to see what you build with it! To get you started, here are a few tips, tricks and best practices. [See here](#) for examples of other spaCy extensions.

## Usage ideas

- **Adding new features and hooking in models.** For example, a sentiment analysis model, or your preferred solution for lemmatization or sentiment analysis. spaCy's built-in tagger, parser and entity recognizer respect annotations that were already set on the `Doc` in a previous step of the pipeline.
- **Integrating other libraries and APIs.** For example, your pipeline component can write additional information and data directly to the `Doc` or `Token` as custom attributes, while making sure no

information is lost in the process. This can be output generated by other libraries and models, or an external service with a REST API.

- **Debugging and logging.** For example, a component which stores and/or exports relevant information about the current state of the processed document, and insert it at any point of your pipeline.

## Best practices

Extensions can claim their own `._` namespace and exist as standalone packages. If you're developing a tool or library and want to make it easy for others to use it with spaCy and add it to their pipeline, all you have to do is expose a function that takes a `Doc`, modifies it and returns it.

- Make sure to choose a **descriptive and specific name** for your pipeline component class, and set it as its `name` attribute. Avoid names that are too common or likely to clash with built-in or a user's other custom components. While it's fine to call your package `"spacy_my_extension"`, avoid component names including `"spacy"`, since this can easily lead to confusion.

```
+ name = "myapp_lemmatizer"
- name = "lemmatizer"
```

- When writing to `Doc`, `Token` or `Span` objects, **use getter functions** wherever possible, and avoid setting values explicitly. Tokens and spans don't own any data themselves, and they're implemented as C extension classes – so you can't usually add new attributes to them like you could with most pure Python objects.

```
+ is_fruit = lambda token: token.text in ("apple", "orange")
+ Token.set_extension("is_fruit", getter=is_fruit)

- token._.set_extension("is_fruit", default=False)
- if token.text in ("apple", "orange"):
-     token._.set("is_fruit", True)
```

- Always add your custom attributes to the **global** `Doc`, `Token` or `Span` objects, not a particular instance of them. Add the attributes **as early as possible**, e.g. in your extension's `__init__` method or in the global scope of your module. This means that in the case of namespace collisions, the user will see an error immediately, not just when they run their pipeline.



```
+ from spacy.tokens import Doc
+ def __init__(attr="my_attr"):
+     Doc.set_extension(attr, getter=self.get_doc_attr)

- def __call__(doc):
-     doc.set_extension("my_attr", getter=self.get_doc_attr)
```

- If your extension is setting properties on the `Doc`, `Token` or `Span`, include an option to **let the user to change those attribute names**. This makes it easier to avoid namespace collisions and accommodate users with different naming preferences. We recommend adding an `attrs` argument to the `__init__` method of your class so you can write the names to class attributes and reuse them across your component.

```
+ Doc.set_extension(self.doc_attr, default="some value")
- Doc.set_extension("my_doc_attr", default="some value")
```

- Ideally, extensions should be **standalone packages** with spaCy and optionally, other packages specified as a dependency. They can freely assign to their own `._` namespace, but should stick to that. If your extension's only job is to provide a better `.similarity` implementation, and your docs state this explicitly, there's no problem with writing to the `user_hooks` and overwriting spaCy's built-in method. However, a third-party extension should **never silently overwrite built-ins**, or attributes set by other extensions.

- If you're looking to publish a pipeline package that depends on a custom pipeline component, you can either **require it** in the package's dependencies, or – if the component is specific and lightweight – choose to **ship it with your pipeline package**. Just make sure the `@Language.component`  or `@Language.factory`  decorator that registers the custom component runs in your package's `__init__.py` or is exposed via an [entry point](#).



Once you're ready to share your extension with others, make sure to **add docs and installation instructions** (you can always link to this page for more info). Make it easy for others to install and use your extension, for example by uploading it to [PyPi](#). If you're sharing your code on GitHub, don't forget to tag it with `spacy` and `spacy-extension` to help people find it. If you post it on Twitter, feel free to tag [@spacy\\_io](#) so we can check it out.

## Wrapping other models and libraries

Let's say you have a custom entity recognizer that takes a list of strings and returns their [BILUO tags](#). Given an input like `["A", "text", "about", "Facebook"]`, it will predict and return `["O", "O", "O", "U-ORG"]`. To integrate it into your spaCy pipeline and make it add those entities to the `doc.ents`, you can wrap it in a custom pipeline component function and pass it the token texts from the `Doc` object received by the component.

The `training.biluo_tags_to_spans` is very helpful here, because it takes a `Doc` object and token-based BILUO tags and returns a sequence of `Span` objects in the `Doc` with added labels. So all your wrapper has to do is compute the entity spans and overwrite the `doc.ents`.

```
import your_custom_entity_recognizer
from spacy.training import biluo_tags_to_spans
from spacy.language import Language

@Language.component("custom_ner_wrapper")
def custom_ner_wrapper(doc):
    words = [token.text for token in doc]
    custom_entities = your_custom_entity_recognizer(words)
    doc.ents = biluo_tags_to_spans(doc, custom_entities)
    return doc
```

The `custom_ner_wrapper` can then be added to a blank pipeline using `nlp.add_pipe`. You can also replace the existing entity recognizer of a trained pipeline with `nlp.replace_pipe`.

Here's another example of a custom model, `your_custom_model`, that takes a list of tokens and returns lists of fine-grained part-of-speech tags, coarse-grained part-of-speech tags, dependency labels and head token indices. Here, we can use the `Doc.from_array` to create a new `Doc` object using those values. To create a numpy array we need integers, so we can look up the string labels in the `StringStore`. The `doc.vocab.strings.add` method comes in handy here, because it returns

the integer ID of the string *and* makes sure it's added to the vocab. This is especially important if the custom model uses a different label scheme than spaCy's default models.

```
import your_custom_model
from spacy.language import Language
from spacy.symbols import POS, TAG, DEP, HEAD
from spacy.tokens import Doc
import numpy

@Language.component("custom_model_wrapper")
def custom_model_wrapper(doc):
    words = [token.text for token in doc]
    spaces = [token.whitespace for token in doc]
    pos, tags, deps, heads = your_custom_model(words)
    # Convert the strings to integers and add them to the string store
    pos = [doc.vocab.strings.add(label) for label in pos]
    tags = [doc.vocab.strings.add(label) for label in tags]
    deps = [doc.vocab.strings.add(label) for label in deps]
    # Create a new Doc from a numpy array
    attrs = [POS, TAG, DEP, HEAD]
    arr = numpy.array(list(zip(pos, tags, deps, heads)), dtype="uint64")
    new_doc = Doc(doc.vocab, words=words, spaces=spaces).from_array(attrs, arr)
    return new_doc
```

[</> SUGGEST EDITS](#)

**READ NEXT**

Embeddings & Transformers

