Models & Languages

spaCy's trained pipelines can be installed as **Python packages**. This means that they're a component of your application, just like any other module. They're versioned and can be defined as a dependency in your `requirements.txt`. Trained pipelines can be installed from a download URL or a local directory, manually or via pip. Their data can be located anywhere on your file system.

IMPORTANT NOTE

If you're upgrading to spaCy v3.x, you need to **download the new pipeline packages**. If you've trained your own pipelines, you need to **retrain** them after updating spaCy.

Quickstart

Install a default trained pipeline package, get the code to load it from within spaCy and an example to test it. For more options, see the section on available packages below.

Language

English

Loading style

Use spacy.load() | Import as module

Select for

efficiency | accuracy

Show text example

```
$ python -m spacy download en_core_web_sm

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

Language support

spaCy currently provides support for the following languages. You can help by improving the existing language data and extending the tokenization patterns. See here for details on how to contribute to development. Also see the training documentation for how to train your own pipelines on your data.

USAGE NOTE

If a trained pipeline is available for a language, you can download it using the `spacy download` command. In order to use languages that don't yet come with a trained pipeline, you have to import them directly, or use `spacy.blank` :

```
from spacy.lang.fi import Finnish
nlp = Finnish()  # use directly
nlp = spacy.blank("fi")  # blank instance
```

If lemmatization rules are available for your language, make sure to install spaCy with the `lookups` option, or install `spacy-lookups-data` separately in the same environment:

```
pip install -U spacy[lookups]
```

LANGUAGE	CODE	LANGUAGE DATA	PIPELINES

LANGUAGE	CODE	LANGUAGE DATA	PIPELINES
Chinese	zh	lang/zh	4 packages
Danish	da	lang/da	4 packages
Dutch	nl	lang/nl	3 packages
English	en	lang/en	4 packages
French	fr	lang/fr	4 packages
German	de	lang/de	4 packages
Greek	el	lang/el	3 packages
Italian	it	lang/it	3 packages
Japanese	ja	lang/ja	3 packages
Lithuanian	lt	lang/lt	3 packages
Macedonian	mk	lang/mk	3 packages
Multi-language	xx	lang/xx	2 packages
Norwegian Bokmål	nb	lang/nb	3 packages
Polish	pl	lang/pl	3 packages
Portuguese	pt	lang/pt	3 packages
Romanian	ro	lang/ro	3 packages
Russian	ru	lang/ru	3 packages
Spanish	es	lang/es	4 packages
Afrikaans	af	lang/af	*none yet*
Albanian	sq	lang/sq	*none yet*
Arabic	ar	lang/ar	*none yet*
Armenian	hy	lang/hy	*none yet*
Basque	eu	lang/eu	*none yet*
Bengali	bn	lang/bn	*none yet*
Bulgarian	bg	lang/bg	*none yet*
Croatian	hr	lang/hr	*none yet*

LANGUAGE	CODE	LANGUAGE DATA	PIPELINES
Estonian	et	lang/et	none yet
Finnish	fi	lang/fi	none yet
Gujarati	gu	lang/gu	none yet
Hebrew	he	lang/he	none yet
Hindi	hi	lang/hi	none yet
Hungarian	hu	lang/hu	none yet
Icelandic	is	lang/is	none yet
Indonesian	id	lang/id	none yet
Irish	ga	lang/ga	none yet
Kannada	kn	lang/kn	none yet
Korean	ko	lang/ko	none yet
Kyrgyz	ky	lang/ky	none yet
Latvian	lv	lang/lv	none yet
Ligurian	lij	lang/lij	none yet
Luxembourgish	lb	lang/lb	none yet
Malayalam	ml	lang/ml	none yet
Marathi	mr	lang/mr	none yet
Nepali	ne	lang/ne	none yet
Persian	fa	lang/fa	none yet
Sanskrit	sa	lang/sa	none yet
Serbian	sr	lang/sr	none yet
Setswana	tn	lang/tn	none yet
Sinhala	si	lang/si	none yet
Slovak	sk	lang/sk	none yet
Slovenian	sl	lang/sl	none yet
Swedish	sv	lang/sv	none yet

Tamil	`ta`	`lang/ta` </>	*none yet*
Tatar	`tt`	`lang/tt` </>	*none yet*
Telugu	`te`	`lang/te` </>	*none yet*
Thai	`th`	`lang/th` </>	*none yet*
Turkish	`tr`	`lang/tr` </>	*none yet*
Ukrainian	`uk`	`lang/uk` </>	*none yet*
Urdu	`ur`	`lang/ur` </>	*none yet*
Vietnamese	`vi`	`lang/vi` </>	*none yet*
Yoruba	`yo`	`lang/yo` </>	*none yet*

Dependencies

Some language tokenizers require external dependencies.

- **Japanese:** Unidic, Mecab </> , SudachiPy </>
- **Korean:** mecab-ko, mecab-ko-dic, natto-py </>
- **Russian:** pymorphy2 </>
- **Thai:** pythainlp </>
- **Ukrainian:** pymorphy2 </>
- **Vietnamese:** Pyvi </>
- **Chinese:** Jieba </> , spacy-pkuseg </>

Multi-language support

```
# Standard import
from spacy.lang.xx import MultiLanguage
nlp = MultiLanguage()

# With lazy-loading
nlp = spacy.blank("xx")
```

is `xx`. The language class, a generic subclass containing only the base language data, can be found in `lang/xx` .

To train a pipeline using the neutral multi-language class, you can set `lang = "xx"` in your training config. You can also import the `MultiLanguage` class directly, or call `spacy.blank("xx")` for lazy-loading.

Chinese language support

The Chinese language class supports three word segmentation options, `char`, `jieba` and `pkuseg`.

MANUAL SETUP

```
from spacy.lang.zh import Chinese

# Character segmentation (default)
nlp = Chinese()
# Jieba
cfg = {"segmenter": "jieba"}
nlp = Chinese.from_config({"nlp": {"tokenizer": cfg}})
# PKUSeg with "mixed" model provided by pkuseg
cfg = {"segmenter": "pkuseg"}
nlp = Chinese.from_config({"nlp": {"tokenizer": cfg}})
nlp.tokenizer.initialize(pkuseg_model="mixed")
```

CONFIG.CFG

```
[nlp.tokenizer]
@tokenizers = "spacy.zh.ChineseTokenizer"
segmenter = "char"
```

`char`	**Character segmentation:** Character segmentation is the default segmentation option. It's enabled when you create a new `Chinese` language class or call `spacy.blank("zh")`.
`jieba`	**Jieba:** to use Jieba for word segmentation, you can set the option `segmenter` to `"jieba"`.
`pkuseg`	**PKUSeg**: As of spaCy v2.3.0, support for PKUSeg has been added to support better segmentation for Chinese OntoNotes and the provided Chinese pipelines. Enable PKUSeg by setting tokenizer option `segmenter` to `"pkuseg"`.

Changed in v3.0

In v3.0, the default word segmenter has switched from Jieba to character segmentation. Because the `pkuseg` segmenter depends on a model that can be loaded from a file, the model is loaded on initialization (typically before training). This ensures that your packaged Chinese model doesn't depend on a local path at runtime.

Details on spaCy's Chinese API

Details on trained and custom Chinese pipelines

Japanese language support

MANUAL SETUP

```
from spacy.lang.ja import Japanese

# Load SudachiPy with split mode A (default)
nlp = Japanese()
# Load SudachiPy with split mode B
cfg = {"split_mode": "B"}
nlp = Japanese.from_config({"nlp": {"tokenizer": cfg}})
```

The Japanese language class uses SudachiPy for word segmentation and part-of-speech tagging. The default Japanese language class and the provided Japanese pipelines use SudachiPy

CONFIG.CFG

```
[nlp.tokenizer]
@tokenizers = "spacy.ja.JapaneseTokenizer"
split_mode = "A"
```

If you run into errors related to `sudachipy`, which is currently under active development, we suggest downgrading to `sudachipy==0.4.9`, which is the version used for training the current Japanese pipelines .

Installing and using trained pipelines

The easiest way to download a trained pipeline is via spaCy's `download` command. It takes care of finding the best-matching package compatible with your spaCy installation.

IMPORTANT NOTE FOR V3.0

Note that as of spaCy v3.0, shortcut links like `en` that create (potentially brittle) symlinks in your spaCy installation are **deprecated**. To download and load an installed pipeline package, use its full name:

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

- nlp = spacy.load("en")
+ nlp = spacy.load("en_core_web_sm")
```

```
# Download best-matching version of a package for your spaCy
installationpython -m spacy download en_core_web_sm# Download exact package
versionpython -m spacy download en_core_web_sm-3.0.0 --direct
```

```
pip install -U spacypython -m spacy download en_core_web_sm
```

```
import spacy
nlp = spacy.load("en_core_web_sm")
doc = nlp("This is a sentence.")
```

If you're in a **Jupyter notebook** or similar environment, you can use the `!` prefix to execute commands. Make sure to **restart your kernel** or runtime after installation (just like you would when installing other Python packages) to make sure that the installed pipeline package can be found.

```
!python -m spacy download en_core_web_sm
```

Installation via pip

To download a trained pipeline directly using pip, point `pip install` to the URL or local path of the wheel file or archive. Installing the wheel is usually more efficient. To find the direct link to a package, head over to the releases , right click on the archive link and copy it to your clipboard.

```
# With external URL
$ pip install https://github.com/explosion/spacy-models/releases/download/en_cor
$ pip install https://github.com/explosion/spacy-models/releases/download/en_cor

# With local file

$ pip install /Users/you/en_core_web_sm-3.0.0-py3-none-any.whl
$ pip install /Users/you/en_core_web_sm-3.0.0.tar.gz
```

By default, this will install the pipeline package into your `site-packages` directory. You can then use `spacy.load` to load it via its package name or import it explicitly as a module. If you need to

You can also add the direct download link to your application's `requirements.txt`. For more details, see the section on working with pipeline packages in production.

Manual download and installation

In some cases, you might prefer downloading the data manually, for example to place it into a custom directory. You can download the package via your browser from the latest releases , or configure your own download script using the URL of the archive file. The archive consists of a package directory that contains another directory with the pipeline data.

DIRECTORY STRUCTURE

```
└── en_core_web_md-3.0.0.tar.gz       # downloaded archive
    ├── setup.py                      # setup file for pip installation
    ├── meta.json                     # copy of pipeline meta
    └── en_core_web_md                # pipeline package
        ├── __init__.py               # init for pip installation
        └── en_core_web_md-3.0.0      # pipeline data            ├──
config.cfg            # pipeline config
            ├── meta.json             # pipeline meta
            └── ...                   # directories with component data
```

You can place the **pipeline package directory** anywhere on your local file system.

Installation from Python

Since the `spacy download` command installs the pipeline as a **Python package**, we always recommend running it from the command line, just like you install other Python packages with `pip install`. However, if you need to, or if you want to integrate the download process into another CLI command, you can also import and call the `download` function used by the CLI via Python.

> Keep in mind that the `download` command installs a Python package into your environment. In order for it to be found after installation, you will need to **restart or reload** your Python process so that new packages are recognized.

```
spacy.cli.download("en_core_web_sm")
```

Using trained pipelines with spaCy

To load a pipeline package, use `spacy.load` with the package name or a path to the data directory:

IMPORTANT NOTE FOR V3.0

Note that as of spaCy v3.0, shortcut links like `en` that create (potentially brittle) symlinks in your spaCy installation are **deprecated**. To download and load an installed pipeline package, use its full name:

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

```
import spacy
nlp = spacy.load("en_core_web_sm")          # load package "en_core_web_sm"
nlp = spacy.load("/path/to/en_core_web_sm")  # load package from a directory

doc = nlp("This is a sentence.")
```

Tip: Preview model info

You can use the `info` command or `spacy.info()` method to print a pipeline package's meta data before loading it. Each `Language` object with a loaded pipeline also exposes the pipeline's meta data as the attribute `meta`. For example, `nlp.meta['version']` will return the package version.

Importing pipeline packages as modules

If you've installed a trained pipeline via `spacy download` or directly via pip, you can also `import` it and then call its `load()` method with no arguments:

Editable Code spaCy v3.0 · Python 3 · via Binder

```
nlp = en_core_web_sm.load()
doc = nlp("This is a sentence.")
```

RUN

How you choose to load your trained pipelines ultimately depends on personal preference. However, **for larger code bases**, we usually recommend native imports, as this will make it easier to integrate pipeline packages with your existing build process, continuous integration workflow and testing framework. It'll also prevent you from ever trying to load a package that is not installed, as your code will raise an `ImportError` immediately, instead of failing somewhere down the line when calling `spacy.load()`. For more details, see the section on working with pipeline packages in production.

Using trained pipelines in production

If your application depends on one or more trained pipeline packages, you'll usually want to integrate them into your continuous integration workflow and build process. While spaCy provides a range of useful helpers for downloading and loading pipeline packages, the underlying functionality is entirely based on native Python packaging. This allows your application to handle a spaCy pipeline like any other package dependency.

Downloading and requiring package dependencies

spaCy's built-in `download` command is mostly intended as a convenient, interactive wrapper. It performs compatibility checks and prints detailed error messages and warnings. However, if you're downloading pipeline packages as part of an automated build process, this only adds an unnecessary layer of complexity. If you know which packages your application needs, you should be specifying them directly.

Because pipeline packages are valid Python packages, you can add them to your application's `requirements.txt`. If you're running your own internal PyPi installation, you can upload the pipeline packages there. pip's requirements file format supports both package names to download via a PyPi server, as well as direct URLs.

```
spacy>=3.0.0,<4.0.0
https://github.com/explosion/spacy-models/releases/download/en_core_web_sm-3.0.0
```

Specifying `#egg=` with the package name tells pip which package to expect from the download URL. This way, the package won't be re-downloaded and overwritten if it's already installed - just like when you're downloading a package from PyPi.

All pipeline packages are versioned and specify their spaCy dependency. This ensures cross-compatibility and lets you specify exact version requirements for each pipeline. If you've trained your own pipeline, you can use the `spacy package` command to generate the required meta data and turn it into a loadable package.

Loading and testing pipeline packages

Pipeline packages are regular Python packages, so you can also import them as a package using Python's native `import` syntax, and then call the `load` method to load the data and return an `nlp` object:

```
import en_core_web_sm
nlp = en_core_web_sm.load()
```

In general, this approach is recommended for larger code bases, as it's more "native", and doesn't rely on spaCy's loader to resolve string names to packages. If a package can't be imported, Python will raise an `ImportError` immediately. And if a package is imported but not used, any linter will catch that.

Similarly, it'll give you more flexibility when writing tests that require loading pipelines. For example, instead of writing your own `try` and `except` logic around spaCy's loader, you can use pytest's `importorskip()` method to only run a test if a specific pipeline package or version is installed. Each pipeline package exposes a `__version__` attribute which you can also use to perform your own version compatibility checks before loading it.