<a href="https://explosion.ai"><img src="https://explosion.ai/assets/img/logo.svg" width="125" height="125" align="right" /></a>

# Contribute to spaCy

Thanks for your interest in contributing to spaCy ? The project is maintained

by [@honnibal](https://github.com/honnibal) and [@ines](https://github.com/ines),

and we'll do our best to help you get started. This page will give you a quick

overview of how things are organised and most importantly, how to get involved.

## Table of contents

1. [Issues and bug reports](#issues-and-bug-reports)

2. [Contributing to the code base](#contributing-to-the-code-base)

3. [Code conventions](#code-conventions)

4. [Adding tests](#adding-tests)

5. [Updating the website](#updating-the-website)

6. [Publishing extensions and plugins](#publishing-spacy-extensions-and-plugins)

7. [Code of conduct](#code-of-conduct)

## Issues and bug reports

First, [do a quick search](https://github.com/issues?q=+is%3Aissue+user%3Aexplosion)

to see if the issue has already been reported. If so, it's often better to just

leave a comment on an existing issue, rather than creating a new one. Old issues

also often include helpful tips and solutions to common problems. You should

also check the [troubleshooting guide](https://spacy.io/usage/#troubleshooting)

to see if your problem is already listed there.

If you're looking for help with your code, consider posting a question on

[Stack Overflow](http://stackoverflow.com/questions/tagged/spacy) instead. If you

tag it `spacy` and `python`, more people will see it and hopefully be able to

help. 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.

### Submitting issues

When opening an issue, use a \*\*descriptive title\*\* and include your

\*\*environment\*\* (operating system, Python version, spaCy version). Our

[issue template](https://github.com/explosion/spaCy/issues/new) helps you

remember the most important details to include. If you've discovered a bug, you

can also submit a [regression test](#fixing-bugs) straight away. When you're

opening an issue to report the bug, simply refer to your pull request in the

issue body. A few more tips:

- \*\*Describing your issue:\*\* Try to provide as many details as possible. What

exactly goes wrong? \_How\_ is it failing? Is there an error?

"XY doesn't work" usually isn't that helpful for tracking down problems. Always

remember to include the code you ran and if possible, extract only the relevant

parts and don't just dump your entire script. This will make it easier for us to

reproduce the error.

- \*\*Getting info about your spaCy installation and environment:\*\* If you're

using spaCy v1.7+, you can use the command line interface to print details and

even format them as Markdown to copy-paste into GitHub issues:

`python -m spacy info --markdown`.

- \*\*Checking the model compatibility:\*\* If you're having problems with a

[statistical model](https://spacy.io/models), it may be because the

model is incompatible with your spaCy installation. In spaCy v2.0+, you can check

this on the command line by running `python -m spacy validate`.

- \*\*Sharing a model's output, like dependencies and entities:\*\* spaCy v2.0+

comes with [built-in visualizers](https://spacy.io/usage/visualizers) that

you can run from within your script or a Jupyter notebook. For some issues, it's

helpful to \*\*include a screenshot\*\* of the visualization. You can simply drag and

drop the image into GitHub's editor and it will be uploaded and included.

- \*\*Sharing long blocks of code or logs:\*\* If you need to include long code,

logs or tracebacks, you can wrap them in `<details>` and `</details>`. This

[collapses the content](https://developer.mozilla.org/en/docs/Web/HTML/Element/details)

so it only becomes visible on click, making the issue easier to read and follow.

### Issue labels

[See this page](https://github.com/explosion/spaCy/labels) for an overview of

the system we use to tag our issues and pull requests.

## Contributing to the code base

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

[spaCy 101 guide](https://spacy.io/usage/spacy-101) and the

[`help wanted (easy)`](https://github.com/explosion/spaCy/issues?q=is%3Aissue+is%3Aopen+label%3A%22help+wanted+%28easy%29%22)

label, which we use to tag bugs and feature requests that are easy and

self-contained. If you've decided to take on one of these problems and you're

making good progress, don't forget to add a quick comment to the issue. You can

also use the issue to ask questions, or share your work in progress.

### What belongs in spaCy?

Every library has a different inclusion philosophy — a policy of what should be

shipped in the core library, and what could be provided in other packages. Our

philosophy is to prefer a smaller core library. We generally ask the following

questions:

- \*\*What would this feature look like if implemented in a separate package?\*\*

Some features would be very difficult to implement externally – for example,

changes to spaCy's built-in methods. In contrast, a library of word

alignment functions could easily live as a separate package that depended on

spaCy — there's little difference between writing `import word\_aligner` and

`import spacy.word\_aligner`. spaCy v2.0+ makes it easy to implement

[custom pipeline components](https://spacy.io/usage/processing-pipelines#custom-components),

and add your own attributes, properties and methods to the `Doc`, `Token` and

`Span`. If you're looking to implement a new spaCy feature, starting with a

custom component package is usually the best strategy. You won't have to worry

about spaCy's internals and you can test your module in an isolated

environment. And if it works well, we can always integrate it into the core

library later.

- \*\*Would the feature be easier to implement if it relied on "heavy" dependencies spaCy doesn't currently require?\*\*

Python has a very rich ecosystem. Libraries like scikit-learn, SciPy, Gensim or

TensorFlow/Keras do lots of useful things — but we don't want to have them as

dependencies. If the feature requires functionality in one of these libraries,

it's probably better to break it out into a different package.

- \*\*Is the feature orthogonal to the current spaCy functionality, or overlapping?\*\*

spaCy strongly prefers to avoid having 6 different ways of doing the same thing.

As better techniques are developed, we prefer to drop support for "the old way".

However, it's rare that one approach \_entirely\_ dominates another. It's very

common that there's still a use-case for the "obsolete" approach. For instance,

[WordNet](https://wordnet.princeton.edu/) is still very useful — but word

vectors are better for most use-cases, and the two approaches to lexical

semantics do a lot of the same things. spaCy therefore only supports word

vectors, and support for WordNet is currently left for other packages.

- \*\*Do you need the feature to get basic things done?\*\* We do want spaCy to be

at least somewhat self-contained. If we keep needing some feature in our

recipes, that does provide some argument for bringing it "in house".

### Getting started

To make changes to spaCy's code base, you need to fork then clone the GitHub repository

and build spaCy from source. You'll need to make sure that you have a

development environment consisting of a Python distribution including header

files, a compiler, [pip](https://pip.pypa.io/en/latest/installing/),

[virtualenv](https://virtualenv.pypa.io/en/stable/) and

[git](https://git-scm.com) installed. The compiler is usually the trickiest part.

```

python -m pip install -U pip

git clone https://github.com/explosion/spaCy

cd spaCy

python -m venv .env

source .env/bin/activate

export PYTHONPATH=`pwd`

pip install -r requirements.txt

python setup.py build\_ext --inplace

```

If you've made changes to `.pyx` files, you need to recompile spaCy before you

can test your changes by re-running `python setup.py build\_ext --inplace`.

Changes to `.py` files will be effective immediately.

? \*\*For more details and instructions, see the documentation on [compiling spaCy from source](https://spacy.io/usage/#source) and the [quickstart widget](https://spacy.io/usage/#section-quickstart) to get the right commands for your platform and Python version.\*\*

### Contributor agreement

If you've made a contribution to spaCy, you should fill in the

[spaCy contributor agreement](.github/CONTRIBUTOR\_AGREEMENT.md) to ensure that

your contribution can be used across the project. If you agree to be bound by

the terms of the agreement, fill in the [template](.github/CONTRIBUTOR\_AGREEMENT.md)

and include it with your pull request, or submit it separately to

[`.github/contributors/`](/.github/contributors). The name of the file should be

your GitHub username, with the extension `.md`. For example, the user

example\_user would create the file `.github/contributors/example\_user.md`.

### Fixing bugs

When fixing a bug, first create an

[issue](https://github.com/explosion/spaCy/issues) if one does not already exist.

The description text can be very short – we don't want to make this too

bureaucratic.

Next, create a test file named `test\_issue[ISSUE NUMBER].py` in the

[`spacy/tests/regression`](spacy/tests/regression) folder. Test for the bug

you're fixing, and make sure the test fails. Next, add and commit your test file

referencing the issue number in the commit message. Finally, fix the bug, make

sure your test passes and reference the issue in your commit message.

? \*\*For more information on how to add tests, check out the [tests README](spacy/tests/README.md).\*\*

## Code conventions

Code should loosely follow [pep8](https://www.python.org/dev/peps/pep-0008/).

As of `v2.1.0`, spaCy uses [`black`](https://github.com/ambv/black) for code

formatting and [`flake8`](http://flake8.pycqa.org/en/latest/) for linting its

Python modules. If you've built spaCy from source, you'll already have both

tools installed.

\*\*?? Note that formatting and linting is currently only possible for Python

modules in `.py` files, not Cython modules in `.pyx` and `.pxd` files.\*\*

### Code formatting

[`black`](https://github.com/ambv/black) is an opinionated Python code

formatter, optimised to produce readable code and small diffs. You can run

`black` from the command-line, or via your code editor. For example, if you're

using [Visual Studio Code](https://code.visualstudio.com/), you can add the

following to your `settings.json` to use `black` for formatting and auto-format

your files on save:

```json

{

"python.formatting.provider": "black",

"[python]": {

"editor.formatOnSave": true

}

}

```

[See here](https://github.com/ambv/black#editor-integration) for the full

list of available editor integrations.

#### Disabling formatting

There are a few cases where auto-formatting doesn't improve readability – for

example, in some of the the language data files like the `tag\_map.py`, or in

the tests that construct `Doc` objects from lists of words and other labels.

Wrapping a block in `# fmt: off` and `# fmt: on` lets you disable formatting

for that particular code. Here's an example:

```python

# fmt: off

text = "I look forward to using Thingamajig. I've been told it will make my life easier..."

heads = [1, 0, -1, -2, -1, -1, -5, -1, 3, 2, 1, 0, 2, 1, -3, 1, 1, -3, -7]

deps = ["nsubj", "ROOT", "advmod", "prep", "pcomp", "dobj", "punct", "",

"nsubjpass", "aux", "auxpass", "ROOT", "nsubj", "aux", "ccomp",

"poss", "nsubj", "ccomp", "punct"]

# fmt: on

```

### Code linting

[`flake8`](http://flake8.pycqa.org/en/latest/) is a tool for enforcing code

style. It scans one or more files and outputs errors and warnings. This feedback

can help you stick to general standards and conventions, and can be very useful

for spotting potential mistakes and inconsistencies in your code. The most

important things to watch out for are syntax errors and undefined names, but you

also want to keep an eye on unused declared variables or repeated

(i.e. overwritten) dictionary keys. If your code was formatted with `black`

(see above), you shouldn't see any formatting-related warnings.

The [`.flake8`](.flake8) config defines the configuration we use for this

codebase. For example, we're not super strict about the line length, and we're

excluding very large files like lemmatization and tokenizer exception tables.

Ideally, running the following command from within the repo directory should

not return any errors or warnings:

```bash

flake8 spacy

```

#### Disabling linting

Sometimes, you explicitly want to write code that's not compatible with our

rules. For example, a module's `\_\_init\_\_.py` might import a function so other

modules can import it from there, but `flake8` will complain about an unused

import. And although it's generally discouraged, there might be cases where it

makes sense to use a bare `except`.

To ignore a given line, you can add a comment like `# noqa: F401`, specifying

the code of the error or warning we want to ignore. It's also possible to

ignore several comma-separated codes at once, e.g. `# noqa: E731,E123`. Here

are some examples:

```python

# The imported class isn't used in this file, but imported here, so it can be

# imported \*from\* here by another module.

from .submodule import SomeClass # noqa: F401

try:

do\_something()

except: # noqa: E722

# This bare except is justified, for some specific reason

do\_something\_else()

```

### Python conventions

All Python code must be written in an \*\*intersection of Python 2 and Python 3\*\*.

This is easy in Cython, but somewhat ugly in Python. Logic that deals with

Python or platform compatibility should only live in

[`spacy.compat`](spacy/compat.py). To distinguish them from the builtin

functions, replacement functions are suffixed with an underscore, for example

`unicode\_`. If you need to access the user's version or platform information,

for example to show more specific error messages, you can use the `is\_config()`

helper function.

```python

from .compat import unicode\_, is\_config

compatible\_unicode = unicode\_('hello world')

if is\_config(windows=True, python2=True):

print("You are using Python 2 on Windows.")

```

Code that interacts with the file-system should accept objects that follow the

`pathlib.Path` API, without assuming that the object inherits from `pathlib.Path`.

If the function is user-facing and takes a path as an argument, it should check

whether the path is provided as a string. Strings should be converted to

`pathlib.Path` objects. Serialization and deserialization functions should always

accept \*\*file-like objects\*\*, as it makes the library io-agnostic. Working on

buffers makes the code more general, easier to test, and compatible with Python

3's asynchronous IO.

Although spaCy uses a lot of classes, \*\*inheritance is viewed with some suspicion\*\*

— it's seen as a mechanism of last resort. You should discuss plans to extend

the class hierarchy before implementing.

We have a number of conventions around variable naming that are still being

documented, and aren't 100% strict. A general policy is that instances of the

class `Doc` should by default be called `doc`, `Token` `token`, `Lexeme` `lex`,

`Vocab` `vocab` and `Language` `nlp`. You should avoid naming variables that are

of other types these names. For instance, don't name a text string `doc` — you

should usually call this `text`. Two general code style preferences further help

with naming. First, \*\*lean away from introducing temporary variables\*\*, as these

clutter your namespace. This is one reason why comprehension expressions are

often preferred. Second, \*\*keep your functions shortish\*\*, so they can work in a

smaller scope. Of course, this is a question of trade-offs.

### Cython conventions

spaCy's core data structures are implemented as [Cython](http://cython.org/) `cdef`

classes. Memory is managed through the `cymem.cymem.Pool` class, which allows

you to allocate memory which will be freed when the `Pool` object is garbage

collected. This means you usually don't have to worry about freeing memory. You

just have to decide which Python object owns the memory, and make it own the

`Pool`. When that object goes out of scope, the memory will be freed. You do

have to take care that no pointers outlive the object that owns them — but this

is generally quite easy.

All Cython modules should have the `# cython: infer\_types=True` compiler

directive at the top of the file. This makes the code much cleaner, as it avoids

the need for many type declarations. If possible, you should prefer to declare

your functions `nogil`, even if you don't especially care about multi-threading.

The reason is that `nogil` functions help the Cython compiler reason about your

code quite a lot — you're telling the compiler that no Python dynamics are

possible. This lets many errors be raised, and ensures your function will run

at C speed.

Cython gives you many choices of sequences: you could have a Python list, a

numpy array, a memory view, a C++ vector, or a pointer. Pointers are preferred,

because they are fastest, have the most explicit semantics, and let the compiler

check your code more strictly. C++ vectors are also great — but you should only

use them internally in functions. It's less friendly to accept a vector as an

argument, because that asks the user to do much more work.

Here's how to get a pointer from a numpy array, memory view or vector:

```cython

cdef void get\_pointers(np.ndarray[int, mode='c'] numpy\_array, vector[int] cpp\_vector, int[::1] memory\_view) nogil:

pointer1 = <int\*>numpy\_array.data

pointer2 = cpp\_vector.data()

pointer3 = &memory\_view[0]

```

Both C arrays and C++ vectors reassure the compiler that no Python operations

are possible on your variable. This is a big advantage: it lets the Cython

compiler raise many more errors for you.

When getting a pointer from a numpy array or memoryview, take care that the data

is actually stored in C-contiguous order — otherwise you'll get a pointer to

nonsense. The type-declarations in the code above should generate runtime errors

if buffers with incorrect memory layouts are passed in.

To iterate over the array, the following style is preferred:

```cython

cdef int c\_total(const int\* int\_array, int length) nogil:

total = 0

for item in int\_array[:length]:

total += item

return total

```

If this is confusing, consider that the compiler couldn't deal with

`for item in int\_array:` — there's no length attached to a raw pointer, so how

could we figure out where to stop? The length is provided in the slice notation

as a solution to this. Note that we don't have to declare the type of `item` in

the code above — the compiler can easily infer it. This gives us tidy code that

looks quite like Python, but is exactly as fast as C — because we've made sure

the compilation to C is trivial.

Your functions cannot be declared `nogil` if they need to create Python objects

or call Python functions. This is perfectly okay — you shouldn't torture your

code just to get `nogil` functions. However, if your function isn't `nogil`, you

should compile your module with `cython -a --cplus my\_module.pyx` and open the

resulting `my\_module.html` file in a browser. This will let you see how Cython

is compiling your code. Calls into the Python run-time will be in bright yellow.

This lets you easily see whether Cython is able to correctly type your code, or

whether there are unexpected problems.

Finally, if you're new to Cython, you should expect to find the first steps a

bit frustrating. It's a very large language, since it's essentially a superset

of Python and C++, with additional complexity and syntax from numpy. The

[documentation](http://docs.cython.org/en/latest/) isn't great, and there are

many "traps for new players". Working in Cython is very rewarding once you're

over the initial learning curve. As with C and C++, the first way you write

something in Cython will often be the performance-optimal approach. In contrast,

Python optimisation generally requires a lot of experimentation. Is it faster to

have an `if item in my\_dict` check, or to use `.get()`? What about `try`/`except`?

Does this numpy operation create a copy? There's no way to guess the answers to

these questions, and you'll usually be dissatisfied with your results — so

there's no way to know when to stop this process. In the worst case, you'll make

a mess that invites the next reader to try their luck too. This is like one of

those [volcanic gas-traps](http://www.wemjournal.org/article/S1080-6032%2809%2970088-2/abstract),

where the rescuers keep passing out from low oxygen, causing another rescuer to

follow — only to succumb themselves. In short, just say no to optimizing your

Python. If it's not fast enough the first time, just switch to Cython.

### Resources to get you started

- [PEP 8 Style Guide for Python Code](https://www.python.org/dev/peps/pep-0008/) (python.org)

- [Official Cython documentation](http://docs.cython.org/en/latest/) (cython.org)

- [Writing C in Cython](https://explosion.ai/blog/writing-c-in-cython) (explosion.ai)

- [Multi-threading spaCy’s parser and named entity recogniser](https://explosion.ai/blog/multithreading-with-cython) (explosion.ai)

## Adding tests

spaCy uses the [pytest](http://doc.pytest.org/) framework for testing. For more

info on this, see the [pytest documentation](http://docs.pytest.org/en/latest/contents.html).

Tests for spaCy modules and classes live in their own directories of the same

name. For example, tests for the `Tokenizer` can be found in

[`/spacy/tests/tokenizer`](spacy/tests/tokenizer). To be interpreted and run,

all test files and test functions need to be prefixed with `test\_`.

When adding tests, make sure to use descriptive names, keep the code short and

concise and only test for one behaviour at a time. Try to `parametrize` test

cases wherever possible, use our pre-defined fixtures for spaCy components and

avoid unnecessary imports.

Extensive tests that take a long time should be marked with `@pytest.mark.slow`.

Tests that require the model to be loaded should be marked with

`@pytest.mark.models`. Loading the models is expensive and not necessary if

you're not actually testing the model performance. If all you need is a `Doc`

object with annotations like heads, POS tags or the dependency parse, you can

use the `get\_doc()` utility function to construct it manually.

? \*\*For more guidelines and information on how to add tests, check out the [tests README](spacy/tests/README.md).\*\*

## Updating the website

For instructions on how to build and run the [website](https://spacy.io) locally see \*\*[Setup and installation](https://github.com/explosion/spaCy/blob/master/website/README.md#setup-and-installation-setup)\*\* in the \_website\_ directory's README.

The docs can always use another example or more detail, and they should always

be up to date and not misleading. To quickly find the correct file to edit,

simply click on the "Suggest edits" button at the bottom of a page.

? \*\*For more info and troubleshooting guides, check out the [website README](website).\*\*

## Publishing spaCy extensions and plugins

We're very excited about all the new possibilities for \*\*community extensions\*\*

and plugins in spaCy v2.0, and we can't wait to see what you build with it!

- An extension or plugin should add substantial functionality, be

\*\*well-documented\*\* and \*\*open-source\*\*. It should be available for users to download

and install as a Python package – for example via [PyPi](http://pypi.python.org).

- Extensions that write to `Doc`, `Token` or `Span` attributes should be wrapped

as [pipeline components](https://spacy.io/usage/processing-pipelines#custom-components)

that users can \*\*add to their processing pipeline\*\* using `nlp.add\_pipe()`.

- When publishing your extension on GitHub, \*\*tag it\*\* with the topics

[`spacy`](https://github.com/topics/spacy?o=desc&s=stars) and

[`spacy-extensions`](https://github.com/topics/spacy-extension?o=desc&s=stars)

to make it easier to find. Those are also the topics we're linking to from the

spaCy website. If you're sharing your project on Twitter, feel free to tag

[@spacy\_io](https://twitter.com/spacy\_io) so we can check it out.

- Once your extension is published, you can open an issue on the

[issue tracker](https://github.com/explosion/spacy/issues) to suggest it for the

[resources directory](https://spacy.io/usage/resources#extensions) on the

website.

? \*\*For more tips and best practices, see the [checklist for developing spaCy extensions](https://spacy.io/usage/processing-pipelines#extensions).\*\*

## Code of conduct

spaCy adheres to the

[Contributor Covenant Code of Conduct](http://contributor-covenant.org/version/1/4/).

By participating, you are expected to uphold this code.