flake8 Documentation

Release 6.0.0

Ian Stapleton Cordasco

Nov 23, 2022

CONTENTS

1 Quickstart **1**
- 1.1 Installation . . . 1
- 1.2 Using Flake8 . . . 1

2 FAQ and Glossary **3**
- 2.1 Frequently Asked Questions . . . 3
- 2.2 Glossary of Terms Used in Flake8 Documentation . . . 4

3 User Guide **5**
- 3.1 Using Flake8 . . . 5

4 Plugin Developer Guide **35**
- 4.1 Writing Plugins for Flake8 . . . 35

5 Contributor Guide **45**
- 5.1 Exploring Flake8's Internals . . . 45

6 Release Notes and History **65**
- 6.1 Release Notes and History . . . 65

7 General Indices **89**

Index **91**

CHAPTER

ONE

QUICKSTART

1.1 Installation

To install **Flake8**, open an interactive shell and run:

```
python<version> -m pip install flake8
```

If you want **Flake8** to be installed for your default Python installation, you can instead use:

```
python -m pip install flake8
```

Note: It is **very** important to install **Flake8** on the *correct* version of Python for your needs. If you want **Flake8** to properly parse new language features in Python 3.5 (for example), you need it to be installed on 3.5 for **Flake8** to understand those features. In many ways, Flake8 is tied to the version of Python on which it runs.

1.2 Using Flake8

To start using **Flake8**, open an interactive shell and run:

```
flake8 path/to/code/to/check.py
# or
flake8 path/to/code/
```

Note: If you have installed **Flake8** on a particular version of Python (or on several versions), it may be best to instead run `python<version> -m flake8`.

If you only want to see the instances of a specific warning or error, you can *select* that error like so:

```
flake8 --select E123,W503 path/to/code/
```

Alternatively, if you want to add a specific warning or error to *ignore*:

```
flake8 --extend-ignore E203,W234 path/to/code/
```

Please read our user guide for more information about how to use and configure **Flake8**.

CHAPTER

TWO

FAQ AND GLOSSARY

2.1 Frequently Asked Questions

2.1.1 When is Flake8 released?

Flake8 is released *as necessary*. Sometimes there are specific goals and drives to get to a release. Usually, we release as users report and fix bugs.

2.1.2 How can I help Flake8 release faster?

Look at the next milestone. If there's work you can help us complete, that will help us get to the next milestone. If there's a show-stopping bug that needs to be released, let us know but please be kind. **Flake8** is developed and released entirely on volunteer time.

2.1.3 What is the next version of Flake8?

In general we try to use milestones to indicate this. If the last release on PyPI is 3.1.5 and you see a milestone for 3.2.0 in GitHub, there's a good chance that 3.2.0 is the next release.

2.1.4 Why does Flake8 use ranges for its dependencies?

Flake8 uses ranges for mccabe, pyflakes, and pycodestyle because each of those projects tend to add *new* checks in minor releases. It has been an implicit design goal of **Flake8**'s to make the list of error codes stable in its own minor releases. That way if you install something from the 2.5 series today, you will not find new checks in the same series in a month from now when you install it again.

Flake8's dependencies tend to avoid new checks in patch versions which is why **Flake8** expresses its dependencies roughly as:

```
pycodestyle >= 2.0.0, < 2.1.0
pyflakes >= 0.8.0, != 1.2.0, != 1.2.1, != 1.2.2, < 1.3.0
mccabe >= 0.5.0, < 0.6.0
```

This allows those projects to release patch versions that fix bugs and for **Flake8** users to consume those fixes.

2.1.5 Should I file an issue when a new version of a dependency is available?

No. The current Flake8 core team (of one person) is also a core developer of pycodestyle, pyflakes, and mccabe. They are aware of these releases.

2.2 Glossary of Terms Used in Flake8 Documentation

check
A piece of logic that corresponds to an error code. A check may be a style check (e.g., check the length of a given line against the user configured maximum) or a lint check (e.g., checking for unused imports) or some other check as defined by a plugin.

class
error class
A larger grouping of related *error codes*. For example, `W503` and `W504` are two codes related to whitespace. `W50` would be the most specific class of codes relating to whitespace. `W` would be the warning class that subsumes all whitespace errors.

error
error code
violation
The symbol associated with a specific *check*. For example, pycodestyle implements *checks* that look for whitespace around binary operators and will either return an error code of `W503` or `W504`.

formatter
A *plugin* that augments the output of **Flake8** when passed to `flake8 --format`.

mccabe
The project **Flake8** depends on to calculate the McCabe complexity of a unit of code (e.g., a function). This uses the `C` *class* of *error codes*.

plugin
A package that is typically installed from PyPI to augment the behaviour of **Flake8** either through adding one or more additional *checks* or providing additional *formatters*.

pycodestyle
The project **Flake8** depends on to provide style enforcement. pycodestyle implements *checks* for **PEP 8**. This uses the `E` and `W` *classes* of *error codes*.

pyflakes
The project **Flake8** depends on to lint files (check for unused imports, variables, etc.). This uses the `F` *class* of *error codes* reported by **Flake8**.

warning
Typically the `W` class of *error codes* from pycodestyle.

CHAPTER

THREE

USER GUIDE

All users of **Flake8** should read this portion of the documentation. This provides examples and documentation around **Flake8**'s assortment of options and how to specify them on the command-line or in configuration files.

3.1 Using Flake8

Flake8 can be used in many ways. A few:

- invoked on the command-line
- invoked via Python

This guide will cover all of these and the nuances for using **Flake8**.

Note: This portion of **Flake8**'s documentation does not cover installation. See the *Installation* section for how to install **Flake8**.

3.1.1 Invoking Flake8

Once you have *installed* **Flake8**, you can begin using it. Most of the time, you will be able to generically invoke **Flake8** like so:

```
$ flake8 ...
```

Where you simply allow the shell running in your terminal to locate **Flake8**. In some cases, though, you may have installed **Flake8** for multiple versions of Python (e.g., Python 3.8 and Python 3.9) and you need to call a specific version. In that case, you will have much better results using:

```
$ python3.8 -m flake8
```

Or

```
$ python3.9 -m flake8
```

Since that will tell the correct version of Python to run **Flake8**.

Note: Installing **Flake8** once will not install it on both Python 3.8 and Python 3.9. It will only install it for the version of Python that is running pip.

It is also possible to specify command-line options directly to **Flake8**:

```
$ flake8 --select E123
```

Or

```
$ python<version> -m flake8 --select E123
```

Note: This is the last time we will show both versions of an invocation. From now on, we'll simply use `flake8` and assume that the user knows they can instead use `python<version> -m flake8` instead.

It's also possible to narrow what **Flake8** will try to check by specifying exactly the paths and directories you want it to check. Let's assume that we have a directory with python files and sub-directories which have python files (and may have more sub-directories) called `my_project`. Then if we only want errors from files found inside `my_project` we can do:

```
$ flake8 my_project
```

And if we only want certain errors (e.g., `E123`) from files in that directory we can also do:

```
$ flake8 --select E123 my_project
```

If you want to explore more options that can be passed on the command-line, you can use the `--help` option:

```
$ flake8 --help
```

And you should see something like:

```
Usage: flake8 [options] file file ...

Options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit

  ...
```

3.1.2 Configuring Flake8

Once you have learned how to *invoke* **Flake8**, you will soon want to learn how to configure it so you do not have to specify the same options every time you use it.

This section will show you how to make

```
$ flake8
```

Remember that you want to specify certain options without writing

```
$ flake8 --select E123,W456 --enable-extensions H111
```

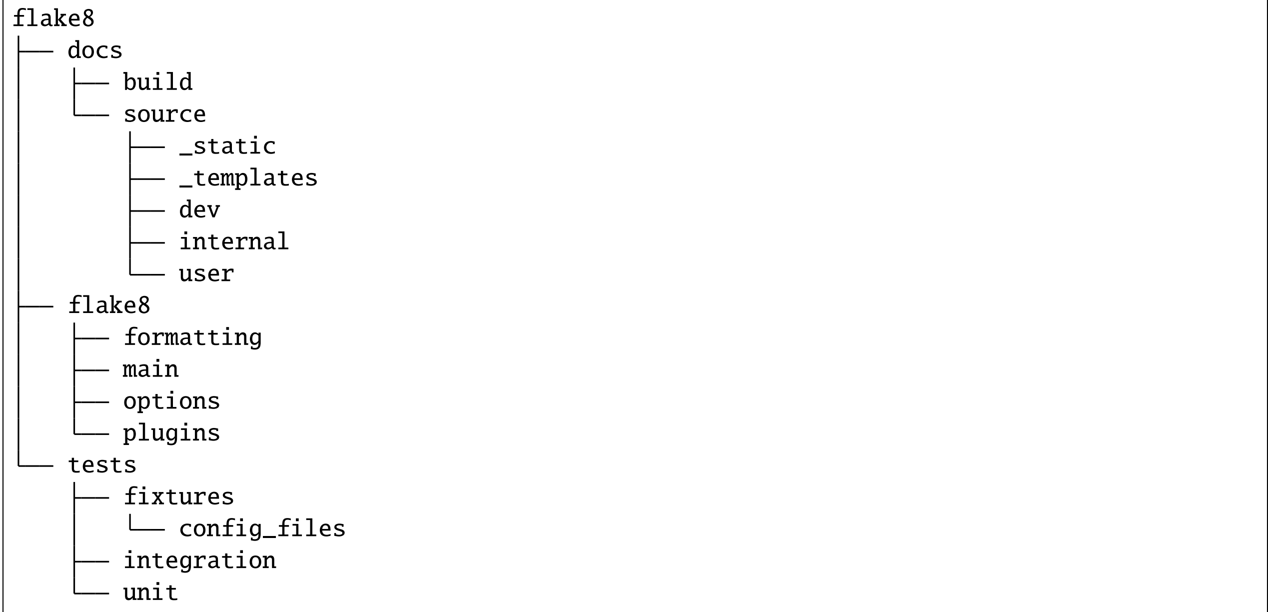
Configuration Locations

Flake8 supports storing its configuration in your project in one of `setup.cfg`, `tox.ini`, or `.flake8`.

Values set at the command line have highest priority, then those in the project configuration file, and finally there are the defaults. However, there are additional command line options which can alter this.

Project Configuration

Flake8 is written with the understanding that people organize projects into sub-directories. Let's take for example **Flake8**'s own project structure



```
flake8
├── docs
│   ├── build
│   └── source
│       ├── _static
│       ├── _templates
│       ├── dev
│       ├── internal
│       └── user
├── flake8
│   ├── formatting
│   ├── main
│   ├── options
│   └── plugins
└── tests
    ├── fixtures
    │   └── config_files
    ├── integration
    └── unit
```

In the top-level `flake8` directory (which contains `docs`, `flake8`, and `tests`) there's also `tox.ini` and `setup.cfg` files. In our case, we keep our **Flake8** configuration in `tox.ini`. Regardless of whether you keep your config in `.flake8`, `setup.cfg`, or `tox.ini` we expect you to use INI to configure **Flake8** (since each of these files already uses INI as a format). This means that any **Flake8** configuration you wish to set needs to be in the `flake8` section, which means it needs to start like so:

```
[flake8]
```

Each command-line option that you want to specify in your config file can be named in either of two ways:

1. Using underscores (_) instead of hyphens (-)
2. Simply using hyphens (without the leading hyphens)

Note: Not every **Flake8** command-line option can be specified in the configuration file. See *our list of options* to determine which options will be parsed from the configuration files.

Let's actually look at **Flake8**'s own configuration section:

```
[flake8]
extend-ignore = E203
exclude = .git,__pycache__,docs/source/conf.py,old,build,dist
max-complexity = 10
```

This is equivalent to:

```
$ flake8 --extend-ignore E203           --exclude .git,__pycache__,docs/source/conf.py,old,build,dist
```

In our case, if we wanted to, we could also do

```
[flake8]
extend-ignore = E203
exclude =
    .git,
    __pycache__,
    docs/source/conf.py,
    old,
    build,
    dist
max-complexity = 10
```

This allows us to add comments for why we're excluding items, e.g.

```
[flake8]
extend-ignore = E203
exclude =
    # No need to traverse our git directory
    .git,
    # There's no value in checking cache directories
    __pycache__,
    # The conf file is mostly autogenerated, ignore it
    docs/source/conf.py,
    # The old directory contains Flake8 2.0
    old,
    # This contains our built documentation
    build,
    # This contains builds of flake8 that we don't want to check
    dist
max-complexity = 10
```

Note: Following the recommended settings for Python's configparser, **Flake8** does not support inline comments for any of the keys. So while this is fine:

```
[flake8]
per-file-ignores =
    # imported but unused
    __init__.py: F401
```

this is not:

```
[flake8]
per-file-ignores =
    __init__.py: F401 # imported but unused
```

Note: If you're using Python 2, you will notice that we download the `configparser` backport from PyPI. That backport enables us to support this behaviour on all supported versions of Python.

Please do **not** open issues about this dependency to **Flake8**.

Note: You can also specify `--max-complexity` as `max_complexity = 10`.

This is also useful if you have a long list of error codes to ignore. Let's look at a portion of a project's Flake8 configuration in their `tox.ini`:

```
[flake8]
# it's not a bug that we aren't using all of hacking, ignore:
# H101: Use TODO(NAME)
# H202: assertRaises Exception too broad
# H233: Python 3.x incompatible use of print operator
# H301: one import per line
# H306: imports not in alphabetical order (time, os)
# H401: docstring should not start with a space
# H403: multi line docstrings should end on a new line
# H404: multi line docstring should start without a leading new line
# H405: multi line docstring summary not separated with an empty line
# H501: Do not use self.__dict__ for string formatting
extend-ignore = H101,H202,H233,H301,H306,H401,H403,H404,H405,H501
```

They use the comments to describe the check but they could also write this as:

```
[flake8]
# it's not a bug that we aren't using all of hacking
extend-ignore =
    # H101: Use TODO(NAME)
    H101,
    # H202: assertRaises Exception too broad
    H202,
    # H233: Python 3.x incompatible use of print operator
    H233,
    # H301: one import per line
    H301,
    # H306: imports not in alphabetical order (time, os)
    H306,
    # H401: docstring should not start with a space
    H401,
    # H403: multi line docstrings should end on a new line
    H403,
    # H404: multi line docstring should start without a leading new line
    H404,
    # H405: multi line docstring summary not separated with an empty line
    H405,
    # H501: Do not use self.__dict__ for string formatting
    H501
```

Or they could use each comment to describe **why** they've ignored the check. **`Flake8`** knows how to parse these lists and will appropriately handle these situations.

Using Local Plugins

New in version 3.5.0.

Flake8 allows users to write plugins that live locally in a project. These plugins do not need to use setuptools or any of the other overhead associated with plugins distributed on PyPI. To use these plugins, users must specify them in their configuration file (i.e., `.flake8`, `setup.cfg`, or `tox.ini`). This must be configured in a separate INI section named `flake8:local-plugins`.

Users may configure plugins that check source code, i.e., `extension` plugins, and plugins that report errors, i.e., `report` plugins.

An example configuration might look like:

```
[flake8:local-plugins]
extension =
    MC1 = project.flake8.checkers:MyChecker1
    MC2 = project.flake8.checkers:MyChecker2
report =
    MR1 = project.flake8.reporters:MyReporter1
    MR2 = project.flake8.reporters:MyReporter2
```

Flake8 will also, however, allow for commas to separate the plugins for example:

```
[flake8:local-plugins]
extension =
    MC1 = project.flake8.checkers:MyChecker1,
    MC2 = project.flake8.checkers:MyChecker2
report =
    MR1 = project.flake8.reporters:MyReporter1,
    MR2 = project.flake8.reporters:MyReporter2
```

These configurations will allow you to select your own custom reporter plugin that you've designed or will utilize your new check classes.

If your package is installed in the same virtualenv that **Flake8** will run from, and your local plugins are part of that package, you're all set; **Flake8** will be able to import your local plugins. However, if you are working on a project that isn't set up as an installable package, or **Flake8** doesn't run from the same virtualenv your code runs in, you may need to tell **Flake8** where to import your local plugins from. You can do this via the `paths` option in the `local-plugins` section of your config:

```
[flake8:local-plugins]
extension =
  MC1 = myflake8plugin:MyChecker1
paths =
  ./path/to
```

Relative paths will be interpreted relative to the config file. Multiple paths can be listed (comma separated just like `exclude`) as needed. If your local plugins have any dependencies, it's up to you to ensure they are installed in whatever Python environment **Flake8** runs in.

Note: These plugins otherwise follow the same guidelines as regular plugins.

3.1.3 Full Listing of Options and Their Descriptions

Index of Options

- *flake8 --version*
- *flake8 --help*
- *flake8 --verbose*
- *flake8 --quiet*
- *flake8 --color*
- *flake8 --count*
- *flake8 --exclude*
- *flake8 --filename*
- *flake8 --stdin-display-name*
- *flake8 --format*
- *flake8 --hang-closing*
- *flake8 --ignore*
- *flake8 --extend-ignore*
- *flake8 --per-file-ignores*
- *flake8 --max-line-length*
- *flake8 --max-doc-length*
- *flake8 --indent-size*
- *flake8 --select*
- *flake8 --extend-select*
- *flake8 --disable-noqa*
- *flake8 --show-source*
- *flake8 --statistics*
- *flake8 --require-plugins*
- *flake8 --enable-extensions*
- *flake8 --exit-zero*
- *flake8 --jobs*
- *flake8 --output-file*
- *flake8 --tee*
- *flake8 --append-config*
- *flake8 --config*
- *flake8 --isolated*
- *flake8 --builtins*
- *flake8 --doctests*

- *flake8 --include-in-doctest*
- *flake8 --exclude-from-doctest*
- *flake8 --benchmark*
- *flake8 --bug-report*
- *flake8 --max-complexity*

Options and their Descriptions

--version

Go back to index

Show **Flake8**'s version as well as the versions of all plugins installed.

Command-line usage:

```
$ flake8 --version
```

This **can not** be specified in config files.

-h, --help

Go back to index

Show a description of how to use **Flake8** and its options.

Command-line usage:

```
$ flake8 --help
$ flake8 -h
```

This **can not** be specified in config files.

-v, --verbose

Go back to index

Increase the verbosity of **Flake8**'s output. Each time you specify it, it will print more and more information.

Command-line example:

```
$ flake8 -vv
```

This **can not** be specified in config files.

-q, --quiet

Go back to index

Decrease the verbosity of **Flake8**'s output. Each time you specify it, it will print less and less information.

Command-line example:

```
$ flake8 -q
```

This **can** be specified in config files.

Example config file usage:

```
quiet = 1
```

--color

Go back to index

Whether to use color in output. Defaults to `auto`.

Possible options are `auto`, `always`, and `never`.

This **can not** be specified in config files.

When color is enabled, the following substitutions are enabled:

- `%(bold)s`
- `%(black)s`
- `%(red)s`
- `%(green)s`
- `%(yellow)s`
- `%(blue)s`
- `%(magenta)s`
- `%(cyan)s`
- `%(white)s`
- `%(reset)s`

--count

Go back to index

Print the total number of errors.

Command-line example:

```
$ flake8 --count dir/
```

This **can** be specified in config files.

Example config file usage:

```
count = True
```

--exclude=<patterns>

Go back to index

Provide a comma-separated list of glob patterns to exclude from checks.

This defaults to: `.svn,CVS,.bzr,.hg,.git,__pycache__,.tox,.nox,.eggs,*.egg`

Example patterns:

- `*.pyc` will match any file that ends with `.pyc`
- `__pycache__` will match any path that has `__pycache__` in it
- `lib/python` will look expand that using `os.path.abspath()` and look for matching paths

Command-line example:

```
$ flake8 --exclude=*.pyc dir/
```

This **can** be specified in config files.

Example config file usage:

```
exclude =
    .tox,
    __pycache__
```

--extend-exclude=<patterns>

Go back to index

New in version 3.8.0.

Provide a comma-separated list of glob patterns to add to the list of excluded ones. Similar considerations as in `--exclude` apply here with regard to the value.

The difference to the `--exclude` option is, that this option can be used to selectively add individual patterns without overriding the default list entirely.

Command-line example:

```
$ flake8 --extend-exclude=legacy/,vendor/ dir/
```

This **can** be specified in config files.

Example config file usage:

```
extend-exclude =
    legacy/,
    vendor/
extend-exclude = legacy/,vendor/
```

--filename=<patterns>

Go back to index

Provide a comma-separate list of glob patterns to include for checks.

This defaults to: `*.py`

Example patterns:

- `*.py` will match any file that ends with `.py`
- `__pycache__` will match any path that has `__pycache__` in it
- `lib/python` will look expand that using `os.path.abspath()` and look for matching paths

Command-line example:

```
$ flake8 --filename=*.py dir/
```

This **can** be specified in config files.

Example config file usage:

```
filename =
    example.py,
    another-example*.py
```

--stdin-display-name=<display_name>

Go back to index

Provide the name to use to report warnings and errors from code on stdin.

Instead of reporting an error as something like:

```
stdin:82:73 E501 line too long
```

You can specify this option to have it report whatever value you want instead of stdin.

This defaults to: `stdin`

Command-line example:

```
$ cat file.py | flake8 --stdin-display-name=file.py -
```

This **can not** be specified in config files.

--format=<format>

Go back to index

Select the formatter used to display errors to the user.

This defaults to: `default`

By default, there are two formatters available:

- default
- pylint

Other formatters can be installed. Refer to their documentation for the name to use to select them. Further, users can specify their own format string. The variables available are:

- code
- col
- path
- row
- text

The default formatter has a format string of:

```
'%(path)s:%(row)d:%(col)d: %(code)s %(text)s'
```

Command-line example:

```
$ flake8 --format=pylint dir/
$ flake8 --format='%(path)s::%(row)d,%(col)d::%(code)s::%(text)s' dir/
```

This **can** be specified in config files.

Example config file usage:

```
format=pylint
format=%(path)s::%(row)d,%(col)d::%(code)s::%(text)s
```

--hang-closing

Go back to index

Toggle whether pycodestyle should enforce matching the indentation of the opening bracket's line. When you specify this, it will prefer that you hang the closing bracket rather than match the indentation.

Command-line example:

```
$ flake8 --hang-closing dir/
```

This **can** be specified in config files.

Example config file usage:

```
hang_closing = True
hang-closing = True
```

--ignore=<errors>

Go back to index

Specify a list of codes to ignore. The list is expected to be comma-separated, and does not need to specify an error code exactly. Since **Flake8** 3.0, this **can** be combined with *--select*. See *--select* for more information.

For example, if you wish to only ignore `W234`, then you can specify that. But if you want to ignore all codes that start with `W23` you need only specify `W23` to ignore them. This also works for `W2` and `W` (for example).

This defaults to: `E121,E123,E126,E226,E24,E704,W503,W504`

Command-line example:

```
$ flake8 --ignore=E121,E123 dir/
$ flake8 --ignore=E24,E704 dir/
```

This **can** be specified in config files.

Example config file usage:

```
ignore =
    E121,
    E123
ignore = E121,E123
```

--extend-ignore=<errors>

Go back to index

New in version 3.6.0.

Specify a list of codes to add to the list of ignored ones. Similar considerations as in *--ignore* apply here with regard to the value.

The difference to the *--ignore* option is, that this option can be used to selectively add individual codes without overriding the default list entirely.

Command-line example:

```
$ flake8 --extend-ignore=E4,E51,W234 dir/
```

This **can** be specified in config files.

Example config file usage:

```
extend-ignore =
    E4,
    E51,
    W234
extend-ignore = E4,E51,W234
```

--per-file-ignores=<filename:errors>[<filename:errors>]

Go back to index

New in version 3.7.0.

Specify a list of mappings of files and the codes that should be ignored for the entirety of the file. This allows for a project to have a default list of violations that should be ignored as well as file-specific violations for files that have not been made compliant with the project rules.

This option supports syntax similar to `--exclude` such that glob patterns will also work here.

This can be combined with both `--ignore` and `--extend-ignore` to achieve a full flexibility of style options.

Command-line usage:

```
$ flake8 --per-file-ignores='project/__init__.py:F401 setup.py:E121'
$ flake8 --per-file-ignores='project/*/__init__.py:F401 setup.py:E121'
```

This **can** be specified in config files.

```
per-file-ignores =
    project/__init__.py:F401
    setup.py:E121
    other_project/*:W9
```

--max-line-length=<n>

Go back to index

Set the maximum length that any line (with some exceptions) may be.

Exceptions include lines that are either strings or comments which are entirely URLs. For example:

```
# https://some-super-long-domain-name.com/with/some/very/long/path

url = '''\
    https://...
'''
```

This defaults to: `79`

Command-line example:

```
$ flake8 --max-line-length 99 dir/
```

This **can** be specified in config files.

Example config file usage:

```
max-line-length = 79
```

--max-doc-length=<n>

Go back to index

Set the maximum length that a comment or docstring line may be.

By default, there is no limit on documentation line length.

Command-line example:

```
$ flake8 --max-doc-length 99 dir/
```

This **can** be specified in config files.

Example config file usage:

```
max-doc-length = 79
```

--indent-size=<n>

Go back to index

Set the number of spaces used for indentation.

By default, 4.

Command-line example:

```
$ flake8 --indent-size 2 dir/
```

This **can** be specified in config files.

Example config file usage:

```
indent-size = 2
```

--select=<errors>

Go back to index

Specify the list of error codes you wish **Flake8** to report. Similarly to *--ignore*. You can specify a portion of an error code to get all that start with that string. For example, you can use E, E4, E43, and E431.

This defaults to: E,F,W,C90

Command-line example:

```
$ flake8 --select=E431,E5,W,F dir/
$ flake8 --select=E,W dir/
```

This can also be combined with *--ignore*:

```
$ flake8 --select=E --ignore=E432 dir/
```

This will report all codes that start with E, but ignore E432 specifically. This is more flexibly than the **Flake8** 2.x and 1.x used to be.

This **can** be specified in config files.

Example config file usage:

```
select =
    E431,
    W,
    F
```

--extend-select=<errors>

Go back to index

New in version 4.0.0.

Specify a list of codes to add to the list of selected ones. Similar considerations as in `--select` apply here with regard to the value.

The difference to the `--select` option is, that this option can be used to selectively add individual codes without overriding the default list entirely.

Command-line example:

```
$ flake8 --extend-select=E4,E51,W234 dir/
```

This **can** be specified in config files.

Example config file usage:

```
extend-select =
    E4,
    E51,
    W234
```

--disable-noqa

Go back to index

Report all errors, even if it is on the same line as a `# NOQA` comment. `# NOQA` can be used to silence messages on specific lines. Sometimes, users will want to see what errors are being silenced without editing the file. This option allows you to see all the warnings, errors, etc. reported.

Command-line example:

```
$ flake8 --disable-noqa dir/
```

This **can** be specified in config files.

Example config file usage:

```
disable_noqa = True
disable-noqa = True
```

--show-source

Go back to index

Print the source code generating the error/warning in question.

Command-line example:

```
$ flake8 --show-source dir/
```

This **can** be specified in config files.

Example config file usage:

```
show_source = True
show-source = True
```

--statistics

Go back to index

Count the number of occurrences of each error/warning code and print a report.

Command-line example:

```
$ flake8 --statistics
```

This **can** be specified in config files.

Example config file usage:

```
statistics = True
```

--require-plugins=<names>

Go back to index

Require specific plugins to be installed before running.

This option takes a list of distribution names (usually the name you would use when running `pip install`).

Command-line example:

```
$ flake8 --require-plugins=flake8-2020,flake8-typing-extensions dir/
```

This **can** be specified in config files.

Example config file usage:

```
require-plugins =
    flake8-2020
    flake8-typing-extensions
```

--enable-extensions=<errors>

Go back to index

Enable *off-by-default* extensions.

Plugins to **Flake8** have the option of registering themselves as off-by-default. These plugins will not be loaded unless enabled by this option.

Command-line example:

```
$ flake8 --enable-extensions=H111 dir/
```

This **can** be specified in config files.

Example config file usage:

```
enable-extensions =
    H111,
    G123
enable_extensions =
    H111,
    G123
```

--exit-zero

Go back to index

Force **Flake8** to use the exit status code 0 even if there are errors.

By default **Flake8** will exit with a non-zero integer if there are errors.

Command-line example:

```
$ flake8 --exit-zero dir/
```

This **can not** be specified in config files.

--jobs=<n>

Go back to index

Specify the number of subprocesses that **Flake8** will use to run checks in parallel.

Note: This option is ignored on platforms where `fork` is not a supported `multiprocessing` method.

This defaults to: `auto`

The default behaviour will use the number of CPUs on your machine as reported by `multiprocessing.cpu_count()`.

Command-line example:

```
$ flake8 --jobs=8 dir/
```

This **can** be specified in config files.

Example config file usage:

```
jobs = 8
```

--output-file=<path>

Go back to index

Redirect all output to the specified file.

Command-line example:

```
$ flake8 --output-file=output.txt dir/
$ flake8 -vv --output-file=output.txt dir/
```

--tee

Go back to index

Also print output to stdout if output-file has been configured.

Command-line example:

```
$ flake8 --tee --output-file=output.txt dir/
```

This **can** be specified in config files.

Example config file usage:

```
tee = True
```

--append-config=<config>

Go back to index

New in version 3.6.0.

Provide extra config files to parse in after and in addition to the files that **Flake8** found on its own. Since these files are the last ones read into the Configuration Parser, so it has the highest precedence if it provides an option specified in another config file.

Command-line example:

```
$ flake8 --append-config=my-extra-config.ini dir/
```

This **can not** be specified in config files.

--config=<config>

Go back to index

Provide a path to a config file that will be the only config file read and used. This will cause **Flake8** to ignore all other config files that exist.

Command-line example:

```
$ flake8 --config=my-only-config.ini dir/
```

This **can not** be specified in config files.

--isolated

Go back to index

Ignore any config files and use **Flake8** as if there were no config files found.

Command-line example:

```
$ flake8 --isolated dir/
```

This **can not** be specified in config files.

--builtins=<builtins>

Go back to index

Provide a custom list of builtin functions, objects, names, etc.

This allows you to let pyflakes know about builtins that it may not immediately recognize so it does not report warnings for using an undefined name.

This is registered by the default PyFlakes plugin.

Command-line example:

```
$ flake8 --builtins=_,_LE,_LW dir/
```

This **can** be specified in config files.

Example config file usage:

```
builtins =
    _,
    _LE,
    _LW
```

--doctests

Go back to index

Enable PyFlakes syntax checking of doctests in docstrings.

This is registered by the default PyFlakes plugin.

Command-line example:

```
$ flake8 --doctests dir/
```

This **can** be specified in config files.

Example config file usage:

```
doctests = True
```

--include-in-doctest=<paths>

Go back to index

Specify which files are checked by PyFlakes for doctest syntax.

This is registered by the default PyFlakes plugin.

Command-line example:

```
$ flake8 --include-in-doctest=dir/subdir/file.py,dir/other/file.py dir/
```

This **can** be specified in config files.

Example config file usage:

```
include-in-doctest =
    dir/subdir/file.py,
    dir/other/file.py
include_in_doctest =
    dir/subdir/file.py,
    dir/other/file.py
```

--exclude-from-doctest=<paths>

Go back to index

Specify which files are not to be checked by PyFlakes for doctest syntax.

This is registered by the default PyFlakes plugin.

Command-line example:

```
$ flake8 --exclude-from-doctest=dir/subdir/file.py,dir/other/file.py dir/
```

This **can** be specified in config files.

Example config file usage:

```
exclude-from-doctest =
    dir/subdir/file.py,
    dir/other/file.py
exclude_from_doctest =
    dir/subdir/file.py,
    dir/other/file.py
```

--benchmark

Go back to index

Collect and print benchmarks for this run of **Flake8**. This aggregates the total number of:

- tokens
- physical lines
- logical lines
- files

and the number of elapsed seconds.

Command-line usage:

```
$ flake8 --benchmark dir/
```

This **can not** be specified in config files.

--bug-report

Go back to index

Generate information necessary to file a complete bug report for Flake8. This will pretty-print a JSON blob that should be copied and pasted into a bug report for Flake8.

Command-line usage:

```
$ flake8 --bug-report
```

The output should look vaguely like:

```
{
  "dependencies": [
    {
      "dependency": "setuptools",
      "version": "25.1.1"
    }
  ],
  "platform": {
    "python_implementation": "CPython",
    "python_version": "2.7.12",
    "system": "Darwin"
  },
  "plugins": [
    {
      "plugin": "mccabe",
      "version": "0.5.1"
    },
```

(continues on next page)

(continued from previous page)

```
    {
      "plugin": "pycodestyle",
      "version": "2.0.0"
    },
    {
      "plugin": "pyflakes",
      "version": "1.2.3"
    }
  ],
  "version": "3.1.0.dev0"
}
```

This **can not** be specified in config files.

--max-complexity=<n>

Go back to index

Set the maximum allowed McCabe complexity value for a block of code.

This option is provided by the `mccabe` dependency's **Flake8** plugin.

Command-line usage:

```
$ flake8 --max-complexity 15 dir/
```

This **can** be specified in config files.

Example config file usage:

```
max-complexity = 15
```

3.1.4 Error / Violation Codes

Flake8 and its plugins assign a code to each message that we refer to as an *error code* (or *violation*). Most plugins will list their error codes in their documentation or README.

Flake8 installs `pycodestyle`, `pyflakes`, and `mccabe` by default and generates its own *error codes* for `pyflakes`:

Code	Example Message
F401	`module` imported but unused
F402	import `module` from line `N` shadowed by loop variable
F403	'from `module` import *' used; unable to detect undefined names
F404	future import(s) `name` after other statements
F405	`name` may be undefined, or defined from star imports: `module`
F406	'from `module` import *' only allowed at module level
F407	an undefined `__future__` feature name was imported
F501	invalid `%` format literal
F502	`%` format expected mapping but got sequence
F503	`%` format expected sequence but got mapping
F504	`%` format unused named arguments
F505	`%` format missing named arguments

continues on next page

Table 1 – continued from previous page

Code	Example Message
F506	% format mixed positional and named arguments
F507	% format mismatch of placeholder and argument count
F508	% format with * specifier requires a sequence
F509	% format with unsupported format character
F521	.format(...) invalid format string
F522	.format(...) unused named arguments
F523	.format(...) unused positional arguments
F524	.format(...) missing argument
F525	.format(...) mixing automatic and manual numbering
F541	f-string without any placeholders
F601	dictionary key name repeated with different values
F602	dictionary key variable name repeated with different values
F621	too many expressions in an assignment with star-unpacking
F622	two or more starred expressions in an assignment (a, *b, *c = d)
F631	assertion test is a tuple, which is always True
F632	use ==/!= to compare str, bytes, and int literals
F633	use of >> is invalid with print function
F634	if test is a tuple, which is always True
F701	a break statement outside of a while or for loop
F702	a continue statement outside of a while or for loop
F703	a continue statement in a finally block in a loop
F704	a yield or yield from statement outside of a function
F706	a return statement outside of a function/method
F707	an except: block as not the last exception handler
F721	syntax error in doctest
F722	syntax error in forward annotation
F723	syntax error in type comment
F811	redefinition of unused name from line N
F821	undefined name name
F822	undefined name name in __all__
F823	local variable name … referenced before assignment
F831	duplicate argument name in function definition
F841	local variable name is assigned to but never used
F901	raise NotImplemented should be raise NotImplementedError

We also report one extra error: E999. We report E999 when we fail to compile a file into an Abstract Syntax Tree for the plugins that require it.

mccabe only ever reports one *violation* - C901 based on the complexity value provided by the user.

Users should also reference pycodestyle's list of error codes.

3.1.5 Selecting and Ignoring Violations

It is possible to select and ignore certain violations reported by **Flake8** and the plugins we've installed. It's also possible as of **Flake8** 3.0 to combine usage of *flake8 --select* and *flake8 --ignore*. This chapter of the User Guide aims to educate about how Flake8 will report errors based on different inputs.

Ignoring Violations with Flake8

By default, **Flake8** has a list of error codes that it ignores. The list used by a version of **Flake8** may be different than the list used by a different version. To see the default list, *flake8 --help* will show the output with the current default list.

Extending the Default Ignore List

If we want to extend the default list of ignored error codes, we can use *flake8 --extend-ignore* to specify a comma-separated list of codes for a specific run on the command line, e.g.,

```
$ flake8 --extend-ignore=E1,E23 path/to/files/ path/to/more/files
```

This tells **Flake8** to ignore any error codes starting with `E1` and `E23`, in addition the default ignore list. To view the default error code ignore list, run *flake8 --help* and refer to the help text for *flake8 --ignore*.

Overriding the Default Ignore List

If we want to *completely* override the default list of ignored error codes, we can use *flake8 --ignore* to specify a comma-separated list of codes for a specific run on the command-line, e.g.,

```
$ flake8 --ignore=E1,E23,W503 path/to/files/ path/to/more/files/
```

This tells **Flake8** to *only* ignore error codes starting with `E1`, `E23`, or `W503` while it is running.

Note: The documentation for *flake8 --ignore* shows examples for how to change the ignore list in the configuration file. See also *Configuring Flake8* as well for details about how to use configuration files.

In-line Ignoring Errors

In some cases, we might not want to ignore an error code (or class of error codes) for the entirety of our project. Instead, we might want to ignore the specific error code on a specific line. Let's take for example a line like

```
example = lambda: 'example'
```

Sometimes we genuinely need something this simple. We could instead define a function like we normally would. Note, in some contexts this distracts from what is actually happening. In those cases, we can also do:

```
example = lambda: 'example'  # noqa: E731
```

This will only ignore the error from pycodestyle that checks for lambda assignments and generates an `E731`. If there are other errors on the line then those will be reported. `# noqa` is case-insensitive, without the colon the part after `# noqa` would be ignored.

Note: If we ever want to disable **Flake8** respecting `# noqa` comments, we can refer to *flake8 --disable-noqa*.

If we instead had more than one error that we wished to ignore, we could list all of the errors with commas separating them:

```
# noqa: E731,E123
```

Finally, if we have a particularly bad line of code, we can ignore every error using simply `# noqa` with nothing after it.

Contents before and after the `# noqa: ...` portion are ignored so multiple comments may appear on one line. Here are several examples:

```
# mypy requires `# type: ignore` to appear first
x = 5  # type: ignore  # noqa: ABC123

# can use to add useful user information to a noqa comment
y = 6  # noqa: ABC456  # TODO: will fix this later
```

Ignoring Entire Files

Imagine a situation where we are adding **Flake8** to a codebase. Let's further imagine that with the exception of a few particularly bad files, we can add **Flake8** easily and move on with our lives. There are two ways to ignore the file:

1. By explicitly adding it to our list of excluded paths (see: *flake8 --exclude*)
2. By adding `# flake8: noqa` to the file

The former is the **recommended** way of ignoring entire files. By using our exclude list, we can include it in our configuration file and have one central place to find what files aren't included in **Flake8** checks. The latter has the benefit that when we run **Flake8** with *flake8 --disable-noqa* all of the errors in that file will show up without having to modify our configuration. Both exist so we can choose which is better for us.

Selecting Violations with Flake8

Flake8 has a default list of violation classes that we use. This list is:

- `C90`

  All `C90` class violations are reported when the user specifies *flake8 --max-complexity*

- `E`

  All `E` class violations are "errors" reported by pycodestyle

- `F`

  All `F` class violations are reported by pyflakes

- `W`

  All `W` class violations are "warnings" reported by pycodestyle

This list can be overridden by specifying *flake8 --select*. Just as specifying *flake8 --ignore* will change the behaviour of **Flake8**, so will *flake8 --select*.

Let's look through some examples using this sample code:

```
# example.py
def foo():
    print(
                "Hello"
        "World"
        )
```

By default, if we run `flake8` on this file we'll get:

```
$ flake8 example.py
```

```
example.py:4:9: E131 continuation line unaligned for hanging indent
```

Now let's select all `E` class violations:

```
$ flake8 --select E example.py
```

```
example.py:3:17: E126 continuation line over-indented for hanging indent
example.py:4:9: E131 continuation line unaligned for hanging indent
example.py:5:9: E121 continuation line under-indented for hanging indent
```

Suddenly we now have far more errors that are reported to us. Using `--select` alone will override the default `--ignore` list. In these cases, the user is telling us that they want all `E` violations and so we ignore our list of violations that we ignore by default.

We can also be highly specific. For example, we can do

```
$ flake8 --select E121 example.py
```

```
example.py:5:9: E121 continuation line under-indented for hanging indent
```

We can also specify lists of items to select both on the command-line and in our configuration files.

```
$ flake8 --select E121,E131 example.py
```

```
example.py:4:9: E131 continuation line unaligned for hanging indent
example.py:5:9: E121 continuation line under-indented for hanging indent
```

Selecting and Ignoring Simultaneously For Fun and Profit

Prior to **Flake8** 3.0, all handling of *flake8 --select* and *flake8 --ignore* was delegated to pycodestyle. Its handling of the options significantly differs from how **Flake8** 3.0 has been designed.

pycodestyle has always preferred `--ignore` over `--select` and will ignore `--select` if the user provides both. **Flake8** 3.0 will now do its best to intuitively combine both options provided by the user. Let's look at some examples using:

```
# example.py
import os


def foo():
    var = 1
    print(
                "Hello"
```

(continues on next page)

(continued from previous page)

```
        "World"
        )
```

If we run **Flake8** with its default settings we get:

```
$ flake8 example.py
```

```
example.py:1:1: F401 'os' imported but unused
example.py:5:5: F841 local variable 'var' is assigned to but never used
example.py:8:9: E131 continuation line unaligned for hanging indent
```

Now let's select all `E` and `F` violations including those in the default ignore list.

```
$ flake8 --select E,F example.py
```

```
example.py:1:1: F401 'os' imported but unused
example.py:5:5: F841 local variable 'var' is assigned to but never used
example.py:7:17: E126 continuation line over-indented for hanging indent
example.py:8:9: E131 continuation line unaligned for hanging indent
example.py:9:9: E121 continuation line under-indented for hanging indent
```

Now let's selectively ignore some of these while selecting the rest:

```
$ flake8 --select E,F --ignore F401,E121 example.py
```

```
example.py:5:5: F841 local variable 'var' is assigned to but never used
example.py:7:17: E126 continuation line over-indented for hanging indent
example.py:8:9: E131 continuation line unaligned for hanging indent
```

Via this example, we can see that the *most specific* **user-specified** rule will win. So in the above, we had very vague select rules and two very specific ignore rules. Let's look at a different example:

```
$ flake8 --select F401,E131 --ignore E,F example.py
```

```
example.py:1:1: F401 'os' imported but unused
example.py:8:9: E131 continuation line unaligned for hanging indent
```

In this case, we see that since our selected violation codes were more specific those were reported.

3.1.6 Using Plugins For Fun and Profit

Flake8 is useful on its own but a lot of **Flake8**'s popularity is due to its extensibility. Our community has developed *plugin*s that augment **Flake8**'s behaviour. Most of these plugins are uploaded to PyPI. The developers of these plugins often have some style they wish to enforce.

For example, flake8-docstrings adds a check for **PEP 257** style conformance. Others attempt to enforce consistency, like flake8-quotes.

Note: The accuracy or reliability of these plugins may vary wildly from plugin to plugin and not all plugins are guaranteed to work with **Flake8** 3.0.

To install a third-party plugin, make sure that you know which version of Python (or pip) you used to install **Flake8**. You can then use the most appropriate of:

```
$ pip install <plugin-name>
$ pip3 install <plugin-name>
$ python -m pip install <plugin-name>
$ python3 -m pip install <plugin-name>
$ python3.9 -m pip install <plugin-name>
```

To install the plugin, where `<plugin-name>` is the package name on PyPI. To verify installation use:

```
$ flake8 --version
$ python<version> -m flake8 --version
```

To see the plugin's name and version in the output.

See also:

How to Invoke Flake8

After installation, most plugins immediately start reporting *error*s. Check the plugin's documentation for which error codes it returns and if it disables any by default.

Note: You can use both *flake8 --select* and *flake8 --ignore* with plugins.

Some plugins register new options, so be sure to check *flake8 --help* for new flags and documentation. These plugins may also allow these flags to be specified in your configuration file. Hopefully, the plugin authors have documented this for you.

See also:

Configuring Flake8

3.1.7 Using Version Control Hooks

Usage with the pre-commit git hooks framework

Flake8 can be included as a hook for pre-commit. The easiest way to get started is to add this configuration to your `.pre-commit-config.yaml`:

```
-   repo: https://github.com/pycqa/flake8
    rev: ''  # pick a git hash / tag to point to
    hooks:
    -   id: flake8
```

See the pre-commit docs for how to customize this configuration.

Checked-in python files will be passed as positional arguments. `flake8` will always lint explicitly passed arguments (*flake8 --exclude* has no effect). Instead use `pre-commit`'s `exclude: ...` regex to exclude files. `pre-commit` won't ever pass untracked files to `flake8` so excluding `.git` / `.tox` / etc. is unnecessary.

```
-   id: flake8
    exclude: ^testing/(data|examples)/
```

`pre-commit` creates an isolated environment for hooks. To use `flake8` plugins, use the `additional_dependencies` setting.

```
-   id: flake8
    additional_dependencies: [flake8-docstrings]
```

3.1.8 Public Python API

Flake8 3.0.0 presently does not have a public, stable Python API.

When it does it will be located in `flake8.api` and that will be documented here.

Legacy API

When **Flake8** broke its hard dependency on the tricky internals of pycodestyle, it lost the easy backwards compatibility as well. To help existing users of that API we have `flake8.api.legacy`. This module includes a couple classes (which are documented below) and a function.

The main usage that the developers of Flake8 observed was using the `get_style_guide()` function and then calling `check_files()`. To a lesser extent, people also seemed to use the `get_statistics()` method on what `check_files` returns. We then sought to preserve that API in this module.

Let's look at an example piece of code together:

```
from flake8.api import legacy as flake8


style_guide = flake8.get_style_guide(ignore=['E24', 'W503'])
report = style_guide.check_files([...])
assert report.get_statistics('E') == [], 'Flake8 found violations'
```

This represents the basic universal usage of all existing Flake8 2.x integrations. Each example we found was obviously slightly different, but this is kind of the gist, so let's walk through this.

Everything that is backwards compatible for our API is in the `flake8.api.legacy` submodule. This is to indicate, clearly, that the old API is being used.

We create a `flake8.api.legacy.StyleGuide` by calling `flake8.api.legacy.get_style_guide()`. We can pass options to `flake8.api.legacy.get_style_guide()` that correspond to the command-line options one might use. For example, we can pass `ignore`, `select`, `exclude`, `format`, etc. Our legacy API, does not enforce legacy behaviour, so we can combine `ignore` and `select` like we might on the command-line, e.g.,

```
style_guide = flake8.get_style_guide(
    ignore=['E24', 'W5'],
    select=['E', 'W', 'F'],
    format='pylint',
)
```

Once we have our `flake8.api.legacy.StyleGuide` we can use the same methods that we used before, namely

> **Warning:** These are not *perfectly* backwards compatible. Not all arguments are respsected, and some of the types necessary for something to work have changed.

Most people, we observed, were using `check_files()`. You can use this to specify a list of filenames or directories to check. In **Flake8** 3.0, however, we return a different object that has similar methods. We return a `flake8.api.legacy.Report` which has the method

Most usage of this method that we noted was as documented above. Keep in mind, however, that it provides a list of strings and not anything more malleable.

Autogenerated Legacy Documentation

CHAPTER

FOUR

PLUGIN DEVELOPER GUIDE

If you're maintaining a plugin for **Flake8** or creating a new one, you should read this section of the documentation. It explains how you can write your plugins and distribute them to others.

4.1 Writing Plugins for Flake8

Since **Flake8** 2.0, the **Flake8** tool has allowed for extensions and custom plugins. In **Flake8** 3.0, we're expanding that ability to customize and extend **and** we're attempting to thoroughly document it. Some of the documentation in this section may reference third-party documentation to reduce duplication and to point you, the developer, towards the authoritative documentation for those pieces.

4.1.1 Getting Started

To get started writing a **Flake8** *plugin* you first need:

- An idea for a plugin
- An available package name on PyPI
- One or more versions of Python installed
- A text editor or IDE of some kind
- An idea of what *kind* of plugin you want to build:
  - Formatter
  - Check

Once you've gathered these things, you can get started.

All plugins for **Flake8** must be registered via entry points. In this section we cover:

- How to register your plugin so **Flake8** can find it
- How to make **Flake8** provide your check plugin with information (via command-line flags, function/class parameters, etc.)
- How to make a formatter plugin
- How to write your check plugin so that it works with **Flake8** 2.x and 3.x

4.1.2 Video Tutorial

Here's a tutorial which goes over building an ast checking plugin from scratch:

Registering a Plugin with Flake8

To register any kind of plugin with **Flake8**, you need:

1. A way to install the plugin (whether it is packaged on its own or as part of something else). In this section, we will use a `setup.py` written for an example plugin.
2. A name for your plugin that will (ideally) be unique.
3. A somewhat recent version of setuptools (newer than 0.7.0 but preferably as recent as you can attain).

Flake8 relies on functionality provided by setuptools called Entry Points. These allow any package to register a plugin with **Flake8** via that package's `setup.py` file.

Let's presume that we already have our plugin written and it's in a module called `flake8_example`. We might have a `setup.py` that looks something like:

```
import setuptools

requires = [
    "flake8 > 3.0.0",
]

flake8_entry_point = # ...

setuptools.setup(
    name="flake8_example",
    license="MIT",
    version="0.1.0",
    description="our extension to flake8",
    author="Me",
    author_email="example@example.com",
    url="https://github.com/me/flake8_example",
    packages=[
        "flake8_example",
    ],
    install_requires=requires,
    entry_points={
        flake8_entry_point: [
            'X = flake8_example:ExamplePlugin',
        ],
    },
    classifiers=[
        "Framework :: Flake8",
        "Environment :: Console",
        "Intended Audience :: Developers",
        "License :: OSI Approved :: MIT License",
        "Programming Language :: Python",
        "Programming Language :: Python :: 3",
        "Topic :: Software Development :: Libraries :: Python Modules",
        "Topic :: Software Development :: Quality Assurance",
```

(continues on next page)

(continued from previous page)

```
    ],
)
```

Note specifically these lines:

```
flake8_entry_point = # ...

setuptools.setup(
    # snip ...
    entry_points={
        flake8_entry_point: [
            'X = flake8_example:ExamplePlugin',
        ],
    },
    # snip ...
)
```

We tell setuptools to register our entry point `X` inside the specific grouping of entry-points that flake8 should look in.

Flake8 presently looks at two groups:

- `flake8.extension`
- `flake8.report`

If your plugin is one that adds checks to **Flake8**, you will use `flake8.extension`. If your plugin performs extra report handling (formatting, filtering, etc.) it will use `flake8.report`.

If our `ExamplePlugin` is something that adds checks, our code would look like:

```
setuptools.setup(
    # snip ...
    entry_points={
        'flake8.extension': [
            'X = flake8_example:ExamplePlugin',
        ],
    },
    # snip ...
)
```

The `X` in checking plugins define what error codes it is going to report. So if the plugin reports only the error code `X101` your entry-point would look like:

```
X101 = flake8_example:ExamplePlugin
```

In the above case, the entry-point name and the error code produced by your plugin are the same.

If your plugin reports several error codes that all start with `X10`, then it would look like:

```
X10 = flake8_example:ExamplePlugin
```

In this case as well as the following case, your entry-point name acts as a prefix to the error codes produced by your plugin.

If all of your plugin's error codes start with `X1` then it would look like:

```
X1 = flake8_example:ExamplePlugin
```

Finally, if all of your plugin's error codes start with just `X` then it would look like the original example.

Flake8 requires each entry point to be unique amongst all plugins installed in the users environment. Selecting an entry point that is already used can cause plugins to be deactivated without warning!

Please Note: Your entry point does not need to be exactly 4 characters as of **Flake8** 3.0. Single letter entry point prefixes (such as the 'X' in the examples above) have caused issues in the past. As such, please consider using a 2 or 3 character entry point prefix, i.e., `ABC` is better than `A` but `ABCD` is invalid. *A 3 letters entry point prefix followed by 3 numbers (i.e.* `ABC123` *) is currently the longest allowed entry point name.*

If your plugin is intended to be opt-in, it can set the attribute `off_by_default = True`. Users of your plugin will then need to utilize *enable-extensions* with your plugin's entry point.

Receiving Information For A Check Plugin

Plugins to **Flake8** have a great deal of information that they can request from a `FileProcessor` instance. Historically, **Flake8** has supported two types of plugins:

1. classes that accept parsed abstract syntax trees (ASTs)
2. functions that accept a range of arguments

Flake8 now does not distinguish between the two types of plugins. Any plugin can accept either an AST or a range of arguments. Further, any plugin that has certain callable attributes can also register options and receive parsed options.

Indicating Desired Data

Flake8 inspects the plugin's signature to determine what parameters it expects using `flake8.plugins.finder._parameters_for()`. `flake8.plugins.finder.LoadedPlugin.parameters` caches the values so that each plugin makes that fairly expensive call once per plugin. When processing a file, a plugin can ask for any of the following:

- `blank_before`
- `blank_lines`
- `checker_state`
- `indent_char`
- `indent_level`
- `line_number`
- `logical_line`
- `multiline`
- `noqa`
- `previous_indent_level`
- `previous_logical`
- `previous_unindented_logical_line`
- `tokens`

Some properties are set once per file for plugins which iterate itself over the data instead of being called on each physical or logical line.

- `filename`
- `file_tokens`
- `lines`
- `max_line_length`
- `max_doc_length`
- `total_lines`
- `verbose`

These parameters can also be supplied to plugins working on each line separately.

Plugins that depend on `physical_line` or `logical_line` are run on each physical or logical line once. These parameters should be the first in the list of arguments (with the exception of `self`). Plugins that need an AST (e.g., PyFlakes and McCabe) should depend on `tree`. These plugins will run once per file. The parameters listed above can be combined with `physical_line`, `logical_line`, and `tree`.

Registering Options

Any plugin that has callable attributes `add_options` and `parse_options` can parse option information and register new options.

Your `add_options` function should expect to receive an instance of `OptionManager`. An `OptionManager` instance behaves very similarly to `optparse.OptionParser`. It, however, uses the layer that **Flake8** has developed on top of `argparse` to also handle configuration file parsing. `add_option()` creates an `Option` which accepts the same parameters as `optparse` as well as three extra boolean parameters:

- `parse_from_config`

  The command-line option should also be parsed from config files discovered by **Flake8**.

  Note: This takes the place of appending strings to a list on the `optparse.OptionParser`.

- `comma_separated_list`

  The value provided to this option is a comma-separated list. After parsing the value, it should be further broken up into a list. This also allows us to handle values like:

  ```
  E123,E124,
  E125,
    E126
  ```

- `normalize_paths`

  The value provided to this option is a path. It should be normalized to be an absolute path. This can be combined with `comma_separated_list` to allow a comma-separated list of paths.

Each of these options works individually or can be combined. Let's look at a couple examples from **Flake8**. In each example, we will have `option_manager` which is an instance of `OptionManager`.

```
option_manager.add_option(
    '--max-line-length', type='int', metavar='n',
    default=defaults.MAX_LINE_LENGTH, parse_from_config=True,
    help='Maximum allowed line length for the entirety of this run. '
```

(continues on next page)

(continued from previous page)

```
        '(Default: %(default)s)',
)
```

Here we are adding the `--max-line-length` command-line option which is always an integer and will be parsed from the configuration file. Since we provide a default, we take advantage of `argparse`'s willingness to display that in the help text with `%(default)s`.

```
option_manager.add_option(
    '--select', metavar='errors', default='',
    parse_from_config=True, comma_separated_list=True,
    help='Comma-separated list of errors and warnings to enable.'
         ' For example, ``--select=E4,E51,W234``. (Default: %(default)s)',
)
```

In adding the `--select` command-line option, we're also indicating to the `OptionManager` that we want the value parsed from the config files and parsed as a comma-separated list.

```
option_manager.add_option(
    '--exclude', metavar='patterns', default=defaults.EXCLUDE,
    comma_separated_list=True, parse_from_config=True,
    normalize_paths=True,
    help='Comma-separated list of files or directories to exclude.'
         '(Default: %(default)s)',
)
```

Finally, we show an option that uses all three extra flags. Values from `--exclude` will be parsed from the config, converted from a comma-separated list, and then each item will be normalized.

For information about other parameters to `add_option()` refer to the documentation of `argparse`.

Accessing Parsed Options

When a plugin has a callable `parse_options` attribute, **Flake8** will call it and attempt to provide the `OptionManager` instance, the parsed options which will be an instance of `argparse.Namespace`, and the extra arguments that were not parsed by the `OptionManager`. If that fails, we will just pass the `argparse.Namespace`. In other words, your `parse_options` callable will have one of the following signatures:

```
def parse_options(option_manager, options, args):
    pass
# or
def parse_options(options):
    pass
```

Developing a Formatting Plugin for Flake8

Flake8 allowed for custom formatting plugins in version 3.0.0. Let's write a plugin together:

```
from flake8.formatting import base


class Example(base.BaseFormatter):
    """Flake8's example formatter."""

    pass
```

We notice, as soon as we start, that we inherit from **Flake8**'s *BaseFormatter* class. If we follow the *instructions to register a plugin* and try to use our example formatter, e.g., `flake8 --format=example` then **Flake8** will fail because we did not implement the `format` method. Let's do that next.

```
class Example(base.BaseFormatter):
    """Flake8's example formatter."""

    def format(self, error):
        return 'Example formatter: {0!r}'.format(error)
```

With that we're done. Obviously this isn't a very useful formatter, but it should highlight the simplicity of creating a formatter with Flake8. If we wanted to instead create a formatter that aggregated the results and returned XML, JSON, or subunit we could also do that. **Flake8** interacts with the formatter in two ways:

1. It creates the formatter and provides it the options parsed from the configuration files and command-line
2. It uses the instance of the formatter and calls `handle` with the error.

By default *flake8.formatting.base.BaseFormatter.handle()* simply calls the `format` method and then `write`. Any extra handling you wish to do for formatting purposes should override the `handle` method.

API Documentation

class flake8.formatting.base.**BaseFormatter**(*options*)

Class defining the formatter interface.

Parameters

options (*argparse.Namespace*) –

options

The options parsed from both configuration files and the command-line.

filename

If specified by the user, the path to store the results of the run.

output_fd

Initialized when the *start()* is called. This will be a file object opened for writing.

newline

The string to add to the end of a line. This is only used when the output filename has been specified.

after_init()

Initialize the formatter further.

Return type
None

`beginning`(*filename*)
Notify the formatter that we're starting to process a file.

Parameters
filename (*str*) – The name of the file that Flake8 is beginning to report results from.

Return type
None

`finished`(*filename*)
Notify the formatter that we've finished processing a file.

Parameters
filename (*str*) – The name of the file that Flake8 has finished reporting results from.

Return type
None

`format`(*error*)
Format an error reported by Flake8.

This method **must** be implemented by subclasses.

Parameters
error (*Violation*) – This will be an instance of `Violation`.

Returns
The formatted error string.

Return type
str | None

`handle`(*error*)
Handle an error reported by Flake8.

This defaults to calling `format()`, `show_source()`, and then `write()`. To extend how errors are handled, override this method.

Parameters
error (*Violation*) – This will be an instance of `Violation`.

Return type
None

`show_benchmarks`(*benchmarks*)
Format and print the benchmarks.

Parameters
benchmarks (*list[tuple[str, float]]*) –

Return type
None

`show_source`(*error*)
Show the physical line generating the error.

This also adds an indicator for the particular part of the line that is reported as generating the problem.

Parameters
error (*Violation*) – This will be an instance of `Violation`.

Returns
The formatted error string if the user wants to show the source. If the user does not want to show the source, this will return `None`.

Return type
str | None

`show_statistics`(*statistics*)
Format and print the statistics.

Parameters
statistics (*Statistics*) –

Return type
None

`start`()
Prepare the formatter to receive input.

This defaults to initializing *output_fd* if *filename*

Return type
None

`stop`()
Clean up after reporting is finished.

Return type
None

`write`(*line*, *source*)
Write the line either to the output file or stdout.

This handles deciding whether to write to a file or print to standard out for subclasses. Override this if you want behaviour that differs from the default.

Parameters

- **line** (*str | None*) – The formatted string to print or write.
- **source** (*str | None*) – The source code that has been formatted and associated with the line of output.

Return type
None

CHAPTER

FIVE

CONTRIBUTOR GUIDE

If you are reading **Flake8**'s source code for fun or looking to contribute, you should read this portion of the documentation. This is a mix of documenting the internal-only interfaces **Flake8** and documenting reasoning for Flake8's design.

5.1 Exploring Flake8's Internals

While writing **Flake8** 3.0, the developers attempted to capture some reasoning and decision information in internal documentation meant for future developers and maintainers. Most of this information is unnecessary for users and plugin developers. Some of it, however, is linked to from the plugin development documentation.

Keep in mind that not everything will be here and you may need to help pull information out of the developers' heads and into these documents. Please pull gently.

5.1.1 Contributing to Flake8

There are many ways to contribute to **Flake8**, and we encourage them all:

- contributing bug reports and feature requests
- contributing documentation (and yes that includes this document)
- reviewing and triaging bugs and merge requests

Before you go any further, please allow me to reassure you that I do want *your* contribution. If you think your contribution might not be valuable, I reassure you that any help you can provide *is* valuable.

Code of Conduct

Flake8 adheres to the Python Code Quality Authority's Code of Conduct. Any violations of the Code of Conduct should be reported to Ian Stapleton Cordasco (graffatcolmingov [at] gmail [dot] com).

Setting Up A Development Environment

To contribute to **Flake8**'s development, you simply need:

- Python (one of the versions we support)
- tox

  We suggest installing this like:

  ```
  $ pip install --user tox
  ```

  Or

  ```
  $ python<version> -m pip install --user tox
  ```

- your favorite editor

Filing a Bug

When filing a bug against **Flake8**, please fill out the issue template as it is provided to you by GitHub. If your bug is in reference to one of the checks that **Flake8** reports by default, please do not report them to **Flake8** unless **Flake8** is doing something to prevent the check from running or you have some reason to believe **Flake8** is inhibiting the effectiveness of the check.

Please search for closed and open bug reports before opening new ones.

All bug reports about checks should go to their respective projects:

- Error codes starting with `E` and `W` should be reported to pycodestyle.
- Error codes starting with `F` should be reported to pyflakes
- Error codes starting with `C` should be reported to mccabe

Requesting a New Feature

When requesting a new feature in **Flake8**, please fill out the issue template. Please also note if there are any existing alternatives to your new feature either via plugins, or combining command-line options. Please provide example use cases. For example, do not ask for a feature like this:

> I need feature frobulate for my job.

Instead ask:

> I need **Flake8** to frobulate these files because my team expects them to frobulated but **Flake8** currently does not frobulate them. We tried using `--filename` but we could not create a pattern that worked.

The more you explain about *why* you need a feature, the more likely we are to understand your needs and help you to the best of our ability.

Contributing Documentation

To contribute to **Flake8**'s documentation, you might want to first read a little about reStructuredText or Sphinx. **Flake8** has a *guide of best practices* when contributing to our documentation. For the most part, you should be fine following the structure and style of the rest of **Flake8**'s documentation.

All of **Flake8**'s documentation is written in reStructuredText and rendered by Sphinx. The source (reStructuredText) lives in `docs/source/`. To build the documentation the way our Continuous Integration does, run:

```
$ tox -e docs
```

To view the documentation locally, you can also run:

```
$ tox -e serve-docs
```

You can run the latter in a separate terminal and continuously re-run the documentation generation and refresh the documentation you're working on.

Note: We lint our documentation just like we lint our code. You should also run:

```
$ tox -e linters
```

After making changes and before pushing them to ensure that they will pass our CI tests.

Contributing Code

Flake8 development happens on GitHub. Code contributions should be submitted there.

Merge requests should:

- Fix one issue and fix it well

  Fix the issue, but do not include extraneous refactoring or code reformatting. In other words, keep the diff short, but only as short as is necessary to fix the bug appropriately and add sufficient testing around it. Long diffs are fine, so long as everything that it includes is necessary to the purpose of the merge request.

- Have descriptive titles and descriptions

  Searching old merge requests is made easier when a merge request is well described.

- Have commits that follow this style:

```
Create a short title that is 50 characters long

Ensure the title and commit message use the imperative voice. The
commit and you are doing something. Also, please ensure that the
body of the commit message does not exceed 72 characters.

The body may have multiple paragraphs as necessary.

The final line of the body references the issue appropriately.
```

- Follow the guidelines in *Writing Code for Flake8*
- Avoid having `.gitignore` file in your PR

  Changes to `.gitignore` will rarely be accepted.

  If you need to add files to `.gitignore` you have multiple options

- Create a global `.gitignore` file
- Create/update `.git/info/exclude` file.

Both these options are explained in detail here

Reviewing and Triaging Issues and Merge Requests

When reviewing other people's merge requests and issues, please be **especially** mindful of how the words you choose can be read by someone else. We strive for professional code reviews that do not insult the contributor's intelligence or impugn their character. The code review should be focused on the code, its effectiveness, and whether it is appropriate for **Flake8**.

If you have the ability to edit an issue or merge request's labels, please do so to make search and prioritization easier.

Flake8 uses milestones with both issues and merge requests. This provides direction for other contributors about when an issue or merge request will be delivered.

5.1.2 Writing Documentation for Flake8

The maintainers of **Flake8** believe strongly in benefit of style guides. Hence, for all contributors who wish to work on our documentation, we've put together a loose set of guidelines and best practices when adding to our documentation.

View the docs locally before submitting

You can and should generate the docs locally before you submit a pull request with your changes. You can build the docs by running:

```
$ tox -e docs
```

From the directory containing the `tox.ini` file (which also contains the `docs/` directory that this file lives in).

Note: If the docs don't build locally, they will not build in our continuous integration system. We will generally not merge any pull request that fails continuous integration.

Run the docs linter tests before submitting

You should run the `doc8` linter job before you're ready to commit and fix any errors found.

Capitalize Flake8 in prose

We believe that by capitalizing **Flake8** in prose, we can help reduce confusion between the command-line usage of `flake8` and the project.

We also have defined a global replacement `|Flake8|` that should be used and will replace each instance with `:program:`Flake8``.

Use the prompt directive for command-line examples

When documenting something on the command-line, use the `.. prompt::` directive to make it easier for users to copy and paste into their terminal.

Example:

```
.. prompt:: bash

    flake8 --select E123,W503 dir/
    flake8 --ignore E24,W504 dir
```

Wrap lines around 79 characters

We use a maximum line-length in our documentation that is similar to the default in **Flake8**. Please wrap lines at 79 characters (or less).

Use two new-lines before new sections

After the final paragraph of a section and before the next section title, use two new-lines to separate them. This makes reading the plain-text document a little nicer. Sphinx ignores these when rendering so they have no semantic meaning.

Example:

```
Section Header
==============

Paragraph.


Next Section Header
===================

Paragraph.
```

Surround document titles with equal symbols

To indicate the title of a document, we place an equal number of = symbols on the lines before and after the title. For example:

```
==================================
 Writing Documentation for Flake8
==================================
```

Note also that we "center" the title by adding a leading space and having extra = symbols at the end of those lines.

Use the option template for new options

All of **Flake8**'s command-line options are documented in the User Guide. Each option is documented individually using the `.. option::` directive provided by Sphinx. At the top of the document, in a reStructuredText comment, is a template that should be copied and pasted into place when documening new options.

Note: The ordering of the options page is the order that options are printed in the output of:

```
$ flake8 --help
```

Please insert your option documentation according to that order.

Use anchors for easy reference linking

Use link anchors to allow for other areas of the documentation to use the `:ref:` role for intralinking documentation. Example:

```
.. _use-anchors:

Use anchors for easy reference linking
======================================
```

```
Somewhere in this paragraph we will :ref:`reference anchors
<use-anchors>`.
```

Note: You do not need to provide custom text for the `:ref:` if the title of the section has a title that is sufficient.

Keep your audience in mind

Flake8's documentation has three distinct (but not separate) audiences:

1. Users
2. Plugin Developers
3. Flake8 Developers and Contributors

At the moment, you're one of the third group (because you're contributing or thinking of contributing).

Consider that most Users aren't very interested in the internal working of **Flake8**. When writing for Users, focus on how to do something or the behaviour of a certain piece of configuration or invocation.

Plugin developers will only care about the internals of **Flake8** as much as they will have to interact with that. Keep discussions of internal to the mininmum required.

Finally, Flake8 Developers and Contributors need to know how everything fits together. We don't need detail about every line of code, but cogent explanations and design specifications will help future developers understand the Hows and Whys of **Flake8**'s internal design.

5.1.3 Writing Code for Flake8

The maintainers of **Flake8** unsurprisingly have some opinions about the style of code maintained in the project.

At the time of this writing, **Flake8** enables all of PyCodeStyle's checks, all of PyFlakes' checks, and sets a maximum complexity value (for McCabe) of 10. On top of that, we enforce PEP-0257 style doc-strings via PyDocStyle (disabling only D203) and Google's import order style using flake8-import-order.

The last two are a little unusual, so we provide examples below.

PEP-0257 style doc-strings

Flake8 attempts to document both internal interfaces as well as our API and doc-strings provide a very convenient way to do so. Even if a function, class, or method isn't included specifically in our documentation having a doc-string is still preferred. Further, **Flake8** has some style preferences that are not checked by PyDocStyle.

For example, while most people will never read the doc-string for `flake8.main.git.hook()` that doc-string still provides value to the maintainers and future collaborators. They (very explicitly) describe the purpose of the function, a little of what it does, and what parameters it accepts as well as what it returns.

```
# src/flake8/main/git.py
def hook(lazy: bool = False, strict: bool = False) -> int:
    """Execute Flake8 on the files in git's index.

    Determine which files are about to be committed and run Flake8 over them
    to check for violations.

    :param lazy:
        Find files not added to the index prior to committing. This is useful
        if you frequently use ``git commit -a`` for example. This defaults to
        False since it will otherwise include files not in the index.
    :param strict:
        If True, return the total number of errors/violations found by Flake8.
        This will cause the hook to fail.
    :returns:
        Total number of errors found during the run.
    """
    # NOTE(sigmavirus24): Delay import of application until we need it.
    from flake8.main import application
    app = application.Application()
    with make_temporary_directory() as tempdir:
        filepaths = list(copy_indexed_files_to(tempdir, lazy))
        app.initialize(['.'])
        app.options.exclude = update_excludes(app.options.exclude, tempdir)
        app.run_checks(filepaths)

    app.report_errors()
    if strict:
        return app.result_count
    return 0
```

Note that we begin the description of the parameter on a new-line and indented 4 spaces.

Following the above examples and guidelines should help you write doc-strings that are stylistically correct for **Flake8**.

Imports

Flake8 follows the import guidelines that Google published in their Python Style Guide. In short this includes:

- Only importing modules
- Grouping imports into
  - standard library imports
  - third-party dependency imports
  - local application imports
- Ordering imports alphabetically

In practice this would look something like:

```
import configparser
import logging
from os import path

import requests

from flake8 import exceptions
from flake8.formatting import base
```

As a result, of the above, we do not:

- Import objects into a namespace to make them accessible from that namespace
- Import only the objects we're using
- Add comments explaining that an import is a standard library module or something else

Other Stylistic Preferences

Finally, **Flake8** has a few other stylistic preferences that it does not presently enforce automatically.

Multi-line Function/Method Calls

When you find yourself having to split a call to a function or method up across multiple lines, insert a new-line after the opening parenthesis, e.g.,

```
# src/flake8/main/options.py
add_option(
    '-v', '--verbose', default=0, action='count',
    parse_from_config=True,
    help='Print more information about what is happening in flake8.'
         ' This option is repeatable and will increase verbosity each '
         'time it is repeated.',
)

# src/flake8/formatting/base.py
def show_statistics(self, statistics):
    """Format and print the statistics."""
    for error_code in statistics.error_codes():
```

(continues on next page)

(continued from previous page)

```
        stats_for_error_code = statistics.statistics_for(error_code)
        statistic = next(stats_for_error_code)
        count = statistic.count
        count += sum(stat.count for stat in stats_for_error_code)
        self._write(f'{count:<5} {error_code} {statistic.message}')
```

In the first example, we put a few of the parameters all on one line, and then added the last two on their own. In the second example, each parameter has its own line. This particular rule is a little subjective. The general idea is that putting one parameter per-line is preferred, but sometimes it's reasonable and understandable to group a few together on one line.

Comments

If you're adding an important comment, be sure to sign it. In **Flake8** we generally sign comments by preceding them with `NOTE(<name>)`. For example,

```
# NOTE(sigmavirus24): The format strings are a little confusing, even
# to me, so here's a quick explanation:
# We specify the named value first followed by a ':' to indicate we're
# formatting the value.
# Next we use '<' to indicate we want the value left aligned.
# Then '10' is the width of the area.
# For floats, finally, we only want only want at most 3 digits after
# the decimal point to be displayed. This is the precision and it
# can not be specified for integers which is why we need two separate
# format strings.
float_format = '{value:<10.3} {statistic}'.format
int_format = '{value:<10} {statistic}'.format
```

Ian is well known across most websites as `sigmavirus24` so he signs his comments that way.

Verbs Belong in Function Names

Flake8 prefers that functions have verbs in them. If you're writing a function that returns a generator of files then `generate_files` will always be preferable to `make_files` or `files`.

5.1.4 Releasing Flake8

There is not much that is hard to find about how **Flake8** is released.

- We use **major** releases (e.g., 2.0.0, 3.0.0, etc.) for big, potentially backwards incompatible, releases.
- We use **minor** releases (e.g., 2.1.0, 2.2.0, 3.1.0, 3.2.0, etc.) for releases that contain features and dependency version changes.
- We use **patch** releases (e.g., 2.1.1, 2.1.2, 3.0.1, 3.0.10, etc.) for releases that contain *only* bug fixes.

In this sense we follow semantic versioning. But we follow it as more of a set of guidelines. We're also not perfect, so we may make mistakes, and that's fine.

Major Releases

Major releases are often associated with backwards incompatibility. **Flake8** hopes to avoid those, but will occasionally need them.

Historically, **Flake8** has generated major releases for:

- Unvendoring dependencies (2.0)
- Large scale refactoring (2.0, 3.0, 5.0, 6.0)
- Subtly breaking CLI changes (3.0, 4.0, 5.0, 6.0)
- Breaking changes to its plugin interface (3.0)

Major releases can also contain:

- Bug fixes (which may have backwards incompatible solutions)
- New features
- Dependency changes

Minor Releases

Minor releases often have new features in them, which we define roughly as:

- New command-line flags
- New behaviour that does not break backwards compatibility
- New errors detected by dependencies, e.g., by raising the upper limit on PyFlakes we introduce F405
- Bug fixes

Patch Releases

Patch releases should only ever have bug fixes in them.

We do not update dependency constraints in patch releases. If you do not install **Flake8** from PyPI, there is a chance that your packager is using different requirements. Some downstream redistributors have been known to force a new version of PyFlakes, pep8/PyCodestyle, or McCabe into place. Occasionally this will cause breakage when using **Flake8**. There is little we can do to help you in those cases.

Process

To prepare a release, we create a file in `docs/source/release-notes/` named: `{{ release_number }}.rst` (e.g., `3.0.0.rst`). We note bug fixes, improvements, and dependency version changes as well as other items of note for users.

Before releasing, the following tox test environments must pass:

- Python 3.6 (a.k.a., `tox -e py36`)
- Python 3.7 (a.k.a., `tox -e py37`)
- PyPy 3 (a.k.a., `tox -e pypy3`)
- Linters (a.k.a., `tox -e linters`)

We tag the most recent commit that passes those items and contains our release notes.

Finally, we run `tox -e release` to build source distributions (e.g., `flake8-3.0.0.tar.gz`), universal wheels, and upload them to PyPI with Twine.

5.1.5 What Happens When You Run Flake8

Given **Flake8** 3.0's new organization and structure, it might be a bit much for some people to understand what happens from when you call `flake8` on the command-line to when it completes. This section aims to give you something of a technical overview of what exactly happens.

Invocation

The exact way that we end up in our `main` function for Flake8 depends on how you invoke it. If you do something like:

```
$ flake8
```

Then your shell looks up where `flake8` the executable lives and executes it. In almost every case, this is a tiny python script generated by `setuptools` using the console script entry points that **Flake8** declares in its `setup.py`. This might look something like:

```
#!/path/to/python<version>
# EASY-INSTALL-ENTRY-SCRIPT: 'flake8==3.0.0','console_scripts','flake8'
__requires__ = 'flake8==3.0.0'
import sys
from pkg_resources import load_entry_point

if __name__ == '__main__':
    sys.exit(
        load_entry_point('flake8==3.0.0', 'console_scripts', 'flake8')()
    )
```

If instead you invoke it like:

```
$ python -m flake8
```

Then you're relying on Python to find `flake8.__main__` and run that. In both cases, however, you end up in `flake8.main.cli.main()`. This is the primary way that users will end up starting Flake8. This function creates an instance of `Application`.

Application Logic

When we create our `Application` instance, we record the start time and parse our command-line arguments so we can configure the verbosity of **Flake8**'s logging. For the most part, every path then calls `run()` which in turn calls:

- `initialize()`
- `run_checks()`
- `report_errors()`
- `report_benchmarks()`

Our Git hook, however, runs these individually.

Application Initialization

`initialize()` loads all of our *plugin*s, registers the options for those plugins, parses the command-line arguments, makes our formatter (as selected by the user), makes our `StyleGuide` and finally makes our `file checker manager`.

Running Our Checks

`run_checks()` then creates an instance of `flake8.checker.FileChecker` for each file to be checked after aggregating all of the files that are not excluded and match the provided file-patterns. Then, if we're on a system that supports `multiprocessing` **and** *flake8 --jobs* is either `auto` or a number greater than 1, we will begin processing the files in subprocesses. Otherwise, we'll run the checks in parallel.

After we start running the checks, we start aggregating the reported *violation*s in the main process. After the checks are done running, we record the end time.

Reporting Violations

Next, the application takes the violations from the file checker manager, and feeds them through the `StyleGuide`. This relies on a `DecisionEngine` instance to determine whether the particular *error code* is selected or ignored and then appropriately sends it to the formatter (or not).

Reporting Benchmarks

Finally, if the user has asked to see benchmarks (i.e., *flake8 --benchmark*) then we print the benchmarks.

Exiting

Once `run()` has finished, we then call `exit()` which looks at how many errors were reported and whether the user specified *flake8 --exit-zero* and exits with the appropriate exit code.

5.1.6 How Checks are Run

In **Flake8** 2.x, **Flake8** delegated check running to pep8. In 3.0 **Flake8** takes on that responsibility. This has allowed for simpler handling of the `--jobs` parameter (using `multiprocessing`) and simplified our fallback if something goes awry with concurrency. At the lowest level we have a `FileChecker`. Instances of `FileChecker` are created for *each* file to be analyzed by **Flake8**. Each instance, has a copy of all of the plugins registered with setuptools in the `flake8.extension` entry-point group.

The `FileChecker` instances are managed by an instance of `Manager`. The `Manager` instance handles creating subprocesses with `multiprocessing` module and falling back to running checks in serial if an operating system level error arises. When creating `FileChecker` instances, the `Manager` is responsible for determining if a particular file has been excluded.

Processing Files

Unfortunately, since **Flake8** took over check running from pep8/pycodestyle, it also had to take over parsing and processing files for the checkers to use. Since it couldn't reuse pycodestyle's functionality (since it did not separate cleanly the processing from check running) that function was isolated into the `FileProcessor` class. We moved several helper functions into the `flake8.processor` module (see also *Processor Utility Functions*).

API Reference

Utility Functions

5.1.7 Command Line Interface

The command line interface of **Flake8** is modeled as an application via `Application`. When a user runs `flake8` at their command line, `main()` is run which handles management of the application.

User input is parsed *twice* to accommodate logging and verbosity options passed by the user as early as possible. This is so as much logging can be produced as possible.

The default **Flake8** options are registered by `register_default_options()`. Trying to register these options in plugins will result in errors.

API Documentation

5.1.8 Built-in Formatters

By default **Flake8** has two formatters built-in, `default` and `pylint`. These correspond to two classes *`Default`* and *`Pylint`*.

In **Flake8** 2.0, pep8 handled formatting of errors and also allowed users to specify an arbitrary format string as a parameter to `--format`. In order to allow for this backwards compatibility, **Flake8** 3.0 made two choices:

1. To not limit a user's choices for `--format` to the format class names
2. To make the default formatter attempt to use the string provided by the user if it cannot find a formatter with that name.

Default Formatter

The *`Default`* continues to use the same default format string as pep8: `'%(path)s:%(row)d:%(col)d:  %(code)s %(text)s'`.

To provide the default functionality it overrides two methods:

1. `after_init`
2. `format`

The former allows us to inspect the value provided to `--format` by the user and alter our own format based on that value. The second simply uses that format string to format the error.

class `flake8.formatting.default.`**`Default`**(*options*)

Default formatter for Flake8.

This also handles backwards compatibility for people specifying a custom format string.

Parameters
options (*argparse.Namespace*) –

after_init()

Check for a custom format string.

Return type
None

Pylint Formatter

The *Pylint* simply defines the default Pylint format string from pep8: `'%(path)s:%(row)d: [%(code)s] %(text)s'`.

class flake8.formatting.default.**Pylint**(*options*)

Pylint formatter for Flake8.

Parameters
options (*argparse.Namespace*) –

5.1.9 Option and Configuration Handling

Option Management

Command-line options are often also set in configuration files for **Flake8**. While not all options are meant to be parsed from configuration files, many default options are also parsed from configuration files as well as most plugin options.

In **Flake8** 2, plugins received a `optparse.OptionParser` instance and called `optparse.OptionParser.add_option()` to register options. If the plugin author also wanted to have that option parsed from config files they also had to do something like:

```
parser.config_options.append('my_config_option')
parser.config_options.extend(['config_opt1', 'config_opt2'])
```

This was previously undocumented and led to a lot of confusion about why registered options were not automatically parsed from configuration files.

Since **Flake8** 3 was rewritten from scratch, we decided to take a different approach to configuration file parsing. Instead of needing to know about an undocumented attribute that pep8 looks for, **Flake8** 3 now accepts a parameter to `add_option`, specifically `parse_from_config` which is a boolean value.

Flake8 does this by creating its own abstractions on top of `argparse`. The first abstraction is the `flake8.options.manager.Option` class. The second is the `flake8.options.manager.OptionManager`. In fact, we add three new parameters:

- `parse_from_config`
- `comma_separated_list`
- `normalize_paths`

The last two are not specifically for configuration file handling, but they do improve that dramatically. We found that there were options that, when specified in a configuration file, often necessitated being split across multiple lines and those options were almost always comma-separated. For example, let's consider a user's list of ignored error codes for a project:

```
[flake8]
ignore =
    # Reasoning
    E111,
    # Reasoning
    E711,
    # Reasoning
    E712,
    # Reasoning
    E121,
    # Reasoning
    E122,
    # Reasoning
    E123,
    # Reasoning
    E131,
    # Reasoning
    E251
```

It makes sense here to allow users to specify the value this way, but, the standard library's `configparser.RawConfigParser` class does returns a string that looks like

```
"\nE111,  \nE711,  \nE712,  \nE121,  \nE122,  \nE123,  \nE131,  \nE251  "
```

This means that a typical call to `str.split()` with `','` will not be sufficient here. Telling **Flake8** that something is a comma-separated list (e.g., `comma_separated_list=True`) will handle this for you. **Flake8** will return:

```
["E111", "E711", "E712", "E121", "E122", "E123", "E131", "E251"]
```

Next let's look at how users might like to specify their `exclude` list. Presently OpenStack's Nova project has this line in their tox.ini:

```
exclude = .venv,.git,.tox,dist,doc,*openstack/common/*,*lib/python*,*egg,build,tools/
↪xenserver*,releasenotes
```

We think we can all agree that this would be easier to read like this:

```
exclude =
    .venv,
    .git,
    .tox,
    dist,
    doc,
    *openstack/common/*,
    *lib/python*,
    *egg,
    build,
    tools/xenserver*,
    releasenotes
```

In this case, since these are actually intended to be paths, we would specify both `comma_separated_list=True` and `normalize_paths=True` because we want the paths to be provided to us with some consistency (either all absolute paths or not).

Now let's look at how this will actually be used. Most plugin developers will receive an instance of `OptionManager`

so to ease the transition we kept the same API as the optparse.OptionParser object. The only difference is that add_option() accepts the three extra arguments we highlighted above.

Configuration File Management

In **Flake8** 2, configuration file discovery and management was handled by pep8. In pep8's 1.6 release series, it drastically broke how discovery and merging worked (as a result of trying to improve it). To avoid a dependency breaking **Flake8** again in the future, we have created our own discovery and management in 3.0.0. In 4.0.0 we have once again changed how this works and we removed support for user-level config files.

- Project files (files stored in the current directory) are read next and merged on top of the user file. In other words, configuration in project files takes precedence over configuration in user files.
- **New in 3.0.0** The user can specify `--append-config <path-to-file>` repeatedly to include extra configuration files that should be read and take precedence over user and project files.
- **New in 3.0.0** The user can specify `--config <path-to-file>` to so this file is the only configuration file used. This is a change from **Flake8** 2 where pep8 would simply merge this configuration file into the configuration generated by user and project files (where this takes precedence).
- **New in 3.0.0** The user can specify `--isolated` to disable configuration via discovered configuration files.

To facilitate the configuration file management, we've taken a different approach to discovery and management of files than pep8. In pep8 1.5, 1.6, and 1.7 configuration discovery and management was centralized in 66 lines of very terse python which was confusing and not very explicit. The terseness of this function (**Flake8** 3.0.0's authors believe) caused the confusion and problems with pep8's 1.6 series. As such, **Flake8** has separated out discovery, management, and merging into a module to make reasoning about each of these pieces easier and more explicit (as well as easier to test).

Configuration file discovery and raw ini reading is managed by load_config(). This produces a loaded RawConfigParser and a config directory (which will be used later to normalize paths).

Next, parse_config() parses options using the types in the OptionManager.

Most of this is done in aggregate_options().

Aggregating Configuration File and Command Line Arguments

aggregate_options() accepts an instance of OptionManager and does the work to parse the command-line arguments.

After parsing the configuration file, we determine the default ignore list. We use the defaults from the OptionManager and update those with the parsed configuration files. Finally we parse the user-provided options one last time using the option defaults and configuration file values as defaults. The parser merges on the command-line specified arguments for us so we have our final, definitive, aggregated options.

API Documentation

5.1.10 Plugin Handling

Plugin Management

Flake8 3.0 added support for other plugins besides those which define new checks. It now supports:

- extra checks
- alternative report formatters

Default Plugins

Finally, **Flake8** has always provided its own plugin shim for Pyflakes. As part of that we carry our own shim in-tree and now store that in `flake8.plugins.pyflakes`.

Flake8 also registers plugins for pycodestyle. Each check in pycodestyle requires different parameters and it cannot easily be shimmed together like Pyflakes was. As such, plugins have a concept of a "group". If you look at our `setup.py` you will see that we register pycodestyle checks roughly like so:

```
pycodestyle.<check-name> = pycodestyle:<check-name>
```

We do this to identify that `<check-name>`> is part of a group. This also enables us to special-case how we handle reporting those checks. Instead of reporting each check in the `--version` output, we only report `pycodestyle` once.

API Documentation

5.1.11 Utility Functions

Flake8 has a few utility functions that it uses internally.

> **Warning:** As should be implied by where these are documented, these are all **internal** utility functions. Their signatures and return types may change between releases without notice.
>
> Bugs reported about these **internal** functions will be closed immediately.
>
> If functions are needed by plugin developers, they may be requested in the bug tracker and after careful consideration they *may* be added to the *documented* stable API.

`flake8.utils.`**`parse_comma_separated_list`**(*value*, *regexp=re.compile('[,\\s]')*)

Parse a comma-separated list.

Parameters

- **value** (*str*) – String to be parsed and normalized.
- **regexp** (*Pattern[str]*) – Compiled regular expression used to split the value when it is a string.

Returns

List of values with whitespace stripped.

Return type

list[str]

parse_comma_separated_list() takes either a string like

```
"E121,W123,F904"
"E121,\nW123,\nF804"
" E121,\n\tW123,\n\tF804 "
" E121\n\tW123 \n\tF804"
```

And converts it to a list that looks as follows

```
["E121", "W123", "F904"]
```

This function helps normalize any kind of comma-separated input you or **Flake8** might receive. This is most helpful when taking advantage of **Flake8**'s additional parameters to `Option`.

`flake8.utils.`**`normalize_path`**(*path*, *parent='.'*)

Normalize a single-path.

Returns
The normalized path.

Parameters

- **path** (*str*) –
- **parent** (*str*) –

Return type
str

This utility takes a string that represents a path and returns the absolute path if the string has a `/` in it. It also removes trailing `/`s.

`flake8.utils.`**`normalize_paths`**(*paths*, *parent='.'*)

Normalize a list of paths relative to a parent directory.

Returns
The normalized paths.

Parameters

- **paths** (*Sequence[str]*) –
- **parent** (*str*) –

Return type
list[str]

This function utilizes `normalize_path()` to normalize a sequence of paths. See `normalize_path()` for what defines a normalized path.

`flake8.utils.`**`stdin_get_value`**()

Get and cache it so plugins can use it.

Return type
str

This function retrieves and caches the value provided on `sys.stdin`. This allows plugins to use this to retrieve `stdin` if necessary.

`flake8.utils.`**`is_using_stdin`**(*paths*)

Determine if we're going to read from stdin.

Parameters
paths (*list[str]*) – The paths that we're going to check.

Returns
True if stdin (-) is in the path, otherwise False

Return type
bool

Another helpful function that is named only to be explicit given it is a very trivial check, this checks if the user specified `-` in their arguments to **`Flake8`** to indicate we should read from stdin.

`flake8.utils.`**`fnmatch`**(*filename*, *patterns*)

Wrap `fnmatch.fnmatch()` to add some functionality.

Parameters

- **filename** (*str*) – Name of the file we're trying to match.
- **patterns** (*Sequence[str]*) – Patterns we're using to try to match the filename.
- **default** – The default value if patterns is empty

Returns
True if a pattern matches the filename, False if it doesn't. `True` if patterns is empty.

Return type
bool

The standard library's `fnmatch.fnmatch()` is excellent at deciding if a filename matches a single pattern. In our use case, however, we typically have a list of patterns and want to know if the filename matches any of them. This function abstracts that logic away with a little extra logic.

CHAPTER

SIX

RELEASE NOTES AND HISTORY

6.1 Release Notes and History

All of the release notes that have been recorded for Flake8 are organized here with the newest releases first.

6.1.1 6.x Release Series

6.0.0 – 2022-11-23

You can view the 6.0.0 milestone on GitHub for more details.

Backwards Incompatible Changes

- Remove `--diff` option (See also #1389, #1720).
- Produce an error when invalid codes are specified in configuration (See also #1689, #1713).
- Produce an error if the file specified in `--extend-config` does not exist (See also #1729, #1732).
- Remove `optparse` compatibility support (See also #1739).

New Dependency Information

- pycodestyle has been updated to >= 2.10.0, < 2.11.0 (See also #1746).
- Pyflakes has been updated to >= 3.0.0, < 3.1.0 (See also #1748).

Features

- Require python >= 3.8.1 (See also #1633, #1741).
- List available formatters in for `--format` option in `--help` (See also #223, #1624).
- Improve multiprocessing performance (See also #1723).
- Enable multiprocessing on non-fork platforms (See also #1723).
- Ensure results are sorted when discovered from files (See also #1670, #1726).

6.1.2 5.x Release Series

5.0.4 – 2022-08-03

You can view the 5.0.4 milestone on GitHub for more details.

Bugs Fixed

- Set a lower bound on `importlib-metadata` to prevent `RecursionError` (See also #1650, #1653).

5.0.3 – 2022-08-01

You can view the 5.0.3 milestone on GitHub for more details.

Bugs Fixed

- Work around partial reads of configuration files with syntax errors (See also #1647, #1648).

5.0.2 – 2022-08-01

You can view the 5.0.2 milestone on GitHub for more details.

Bugs Fixed

- Fix execution on python == 3.8.0 (See also #1637, #1641).
- Fix config discovery when home does not exist (See also #1640, #1642).

5.0.1 – 2022-07-31

You can view the 5.0.1 milestone on GitHub for more details.

Bugs Fixed

- Fix duplicate plugin discovery on misconfigured pythons (See also #1627, #1631).

5.0.0 – 2022-07-30

You can view the 5.0.0 milestone on GitHub for more details.

Backwards Incompatible Changes

- Remove `indent_size_str` (See also #1411).
- Remove some dead code (See also #1453, #1540, #1541).
- Missing explicitly-specified configuration is now an error (See also #1497, #1498).
- Always read configuration files as UTF-8 (See also #1532, #1533).
- Remove manpage from docs – use `help2man` or related tools instead (See also #1557).
- Forbid invalid plugin codes (See also #325, #1579).

Deprecations

- Deprecate `--diff` option (See also #1389, #1441).

New Dependency Information

- pycodestyle has been updated to >= 2.9.0, < 2.10.0 (See also #1626).
- Pyflakes has been updated to >= 2.5.0, < 2.6.0 (See also #1625).
- mccabe has been updated to >= 0.7.0, < 0.8.0 (See also #1542).

Features

- Add colors to output, configurable via `--color` (See also #1379, #1440).
- Add `.nox` to the default exclude list (See also #1442, #1443).
- Don't consider a config file which does not contain flake8 settings (See also #199, #1472).
- Duplicate `local-plugins` names are now allowed (See also #362, #1504).
- Consider `.` to be a path in config files (See also #1494, #1508)
- Add `--require-plugins` option taking distribution names (See also #283, #1535).
- Improve performance by removing debug logs (See also #1537, #1544).
- Include failing file path in plugin execution error (See also #265, #1543).
- Improve performance by pre-generating a `pycodestyle` plugin (See also #1545).
- Properly differentiate between explicitly ignored / selected and default ignored / selected options (See also #284, #1576, #1609).

Bugs Fixed

- Fix physical line plugins not receiving all lines in the case of triple-quoted strings (See also #1534, #1536).
- Fix duplicate error logging in the case of plugin issues (See also #1538).
- Fix inconsistent ordering of `--ignore` in `--help` (See also #1550, #1552).
- Fix memory leak of style guides by avoiding `@lru_cache` of a method (See also #1573).
- Fix ignoring of configuration files exactly in the home directory (See also #1617, #1618).

6.1.3 4.x Release Series

4.0.1 – 2021-10-11

You can view the 4.0.1 milestone on GitHub for more details.

Bugs Fixed

- Fix parallel execution collecting a `SyntaxError` (See also #1410 #1408).

4.0.0 – 2021-10-10

You can view the 4.0.0 milestone on GitHub for more details.

Backwards Incompatible Changes

- Remove `--install-hook` vcs integration (See also #1008).
- Remove `setuptools` command (See also #1009).
- Migrate from GitLab to GitHub (See also #1305).
- Due to constant confusion by users, user-level **Flake8** configuration files are no longer supported. Files will not be searched for in the user's home directory (e.g., `~/.flake8`) nor in the XDG config directory (e.g., `~/.config/flake8`). (See also #1404).

New Dependency Information

- pycodestyle has been updated to >= 2.8.0, < 2.9.0 (See also #1406).
- Pyflakes has been updated to >= 2.4.0, < 2.5.0 (See also #1406).
- flake8 requires python >= 3.6 (See also #1010).

Features

- Add `--extend-select` option (See also #1312 #1061).
- Automatically create directories for output files (See also #1329).

Bugs Fixed

- `ast` parse before tokenizing to improve `SyntaxError` errors (See also #1320 #740).
- Fix warning in `--indent-size` argparse help (See also #1367).
- Fix handling `SyntaxError` in python 3.10+ (See also #1374 #1372).
- Fix writing non-cp1252-encodable when output is piped on windows (See also #1382 #1381).

6.1.4 3.x Release Series

3.9.2 – 2021-05-08

You can view the 3.9.2 milestone on GitHub for more details.

Bugs Fixed

- Fix error message for `E111` in `pycodestyle` (See also #1328, #1327).

Deprecations

- `indent_size_str` is deprecated, use `str(indent_size)` instead (See also #1328, #1327).

3.9.1 – 2021-04-15

You can view the 3.9.1 milestone on GitHub for more details.

Bugs Fixed

- Fix codes being ignored by plugins utilizing `extend_default_ignore` (See also #1317)

3.9.0 – 2021-03-14

You can view the 3.9.0 milestone on GitHub for more details.

New Dependency Information

- Pyflakes has been updated to >= 2.3.0, < 2.4.0 (See also #1006)
- pycodestyle has been updated to >= 2.7.0, < 2.8.0 (See also #1007)

Deprecations

- Drop support for python 3.4 (See also #1283)

Features

- Add `--no-show-source` option to disable `--show-source` (See also #995)

Bugs Fixed

- Fix handling of `crlf` line endings when linting stdin (See also #1002)

3.8.4 – 2020-10-02

You can view the 3.8.4 milestone on GitHub for more details.

Bugs Fixed

- Fix multiprocessing errors on platforms without `sem_open` syscall. (See also #1282)
- Fix skipping of physical checks on the last line of a file which does not end in a newline (See also #997)

3.8.3 – 2020-06-08

You can view the 3.8.3 milestone on GitHub for more details.

Bugs Fixed

- Also catch `SyntaxError` when tokenizing (See also #992, #747)
- Fix `--jobs` default display in `flake8 --help` (See also #1272, #750)

3.8.2 – 2020-05-22

You can view the 3.8.2 milestone on GitHub for more details.

Bugs Fixed

- Improve performance by eliminating unnecessary sort (See also #991)
- Improve messaging of `--jobs` argument by utilizing `argparse` (See also #1269, #1110)
- Fix file configuration options to be relative to the config passed on the command line (See also #442, #736)
- Fix incorrect handling of `--extend-exclude` by treating its values as files (See also #1271, #738)

3.8.1 – 2020-05-11

You can view the 3.8.1 milestone on GitHub for more details.

Bugs Fixed

- Fix `--output-file` (regression in 3.8.0) (See also #990, #725)

3.8.0 – 2020-05-11

You can view the 3.8.0 milestone on GitHub for more details.

Bugs Fixed

- Fix logical checks which report positions out of bounds (See also #987, #723)
- Fix `--exclude=.*` accidentally matching `.` and `..` (See also #441, #360)

Deprecations

- Add deprecation message for vcs hooks (See also #985, #296)

3.8.0a2 – 2020-04-24

You can view the 3.8.0 milestone on GitHub for more details.

Bugs Fixed

- Fix `type="str"` optparse options (See also #984)

3.8.0a1 – 2020-04-24

You can view the 3.8.0 milestone on GitHub for more details.

New Dependency Information

- Remove dependency on `entrypoints` and add dependency on `importlib-metadata` (only for `python<3.8`) (See also #1297, #297)
- Pyflakes has been updated to >= 2.2.0, < 2.3.0 (See also #982)
- pycodestyle has been updated to >= 2.6.0a1, < 2.7.0 (See also #983)

Features

- Add `--extend-exclude` option to add to `--exclude` without overwriting (See also #1211, #1091)
- Move argument parsing from `optparse` to `argparse` (See also #939
- Group plugin options in `--help` (See also #1219, #294)
- Remove parsing of `verbose` from configuration files as it was not consistently applied (See also #1245, #245)
- Remove parsing of `output_file` from configuration files as it was not consistently applied (See also #1246)
- Resolve configuration files relative to `cwd` instead of common prefix of passed filenames. You may need to change `flake8 subproject` to `cd subproject && flake8 .` (See also #952)
- Officially support python3.8 (See also #963)
- `--disable-noqa` now also disables `# flake8:  noqa` (See also #1296, #318)
- Ensure that a missing file produces a `E902` error (See also #1262, #328)
- `# noqa` comments now apply to all of the lines in an explicit `\` continuation or in a line continued by a multi-line string (See also #1266, #621)

Bugs Fixed

- Fix `--exclude=./t.py` to only match `t.py` at the top level (See also #1208, #628)
- Fix `--show-source` when a file is indented with tabs (See also #1218, #719)
- Fix crash when `--max-line-length` is given a non-integer (See also #939, #704)
- Prevent flip-flopping of `indent_char` causing extra `E101` errors (See also #949, pycodestyle#886)
- Only enable multiprocessing when the method is `fork` fixing issues on macos with python3.8+ (See also #955, #315) (note: this fix also landed in 3.7.9)
- `noqa` is now only handled by flake8 fixing specific-noqa. Plugins requesting this parameter will always receive `False` (See also #1214, #1104)
- Fix duplicate loading of plugins when invoked via `python -m flake8` (See also #1297)
- Fix early exit when `--exit-zero` and `--diff` are provided and the diff is empty (See also #970)
- Consistently split lines when `\f` is present when reading from stdin (See also #976, #202)

Deprecations

- `python setup.py flake8` (setuptools integration) is now deprecated and will be removed in a future version (See also #935, #1098)
- `type='string'` (optparse) types are deprecated, use `type=callable` (argparse) instead. Support for `type='string'` will be removed in a future version (See also #939)
- `%default` in plugin option help text is deprecated, use `%(default)s` instead. Support for `%default` will be removed in a future version (See also #939)
- optparse-style `action='callback'` setting for options is deprecated, use argparse action classes instead. This will be removed in a future version (See also #939)

3.7.9 – 2019-10-28

You can view the 3.7.9 milestone on GitHub for more details.

Bugs Fixed

- Disable multiprocessing when the multiprocessing method is `spawn` (such as on macos in python3.8) (See also #956, #315)

3.7.8 – 2019-07-08

You can view the 3.7.8 milestone on GitHub for more details.

Bugs Fixed

- Fix handling of `Application.parse_preliminary_options_and_args` when argv is an empty list (See also #1303, #694)
- Fix crash when a file parses but fails to tokenize (See also #1210, #1088)
- Log the full traceback on plugin exceptions (See also #926)
- Fix `# noqa: ...` comments with multi-letter codes (See also #931, #1101)

3.7.7 – 2019-02-25

You can view the 3.7.7 milestone on GitHub for more details.

Bugs Fixed

- Fix crashes in plugins causing `flake8` to hang while unpickling errors (See also #1206, #681)

3.7.6 – 2019-02-18

You can view the 3.7.6 milestone on GitHub for more details.

Bugs Fixed

- Fix `--per-file-ignores` for multi-letter error codes (See also #1203, #683)
- Improve flake8 speed when only 1 filename is passed (See also #1204)

3.7.5 – 2019-02-04

You can view the 3.7.5 milestone on GitHub for more details.

Bugs Fixed

- Fix reporting of pyflakes "referenced before assignment" error (See also #923, #679)

3.7.4 – 2019-01-31

You can view the 3.7.4 milestone on GitHub for more details.

Bugs Fixed

- Fix performance regression with lots of `per-file-ignores` and errors (See also #922, #677)

3.7.3 – 2019-01-30

You can view the 3.7.3 milestone on GitHub for more details.

Bugs Fixed

- Fix imports of `typing` in python 3.5.0 / 3.5.1 (See also #1199, #674)
- Fix `flake8 --statistics` (See also #920, #675)
- Gracefully ignore `flake8-per-file-ignores` plugin if installed (See also #1201, #671)
- Improve error message for malformed `per-file-ignores` (See also #921, #288)

3.7.2 – 2019-01-30

You can view the 3.7.2 milestone on GitHub for more details.

Bugs Fixed

- Fix broken `flake8 --diff` (regressed in 3.7.0) (See also #919, #667)
- Fix typo in plugin exception reporting (See also #908, #668)
- Fix `AttributeError` while attempting to use the legacy api (regressed in 3.7.0) (See also #1198, #673)

3.7.1 – 2019-01-30

You can view the 3.7.1 milestone on GitHub for more details.

Bugs Fixed

- Fix capitalized filenames in `per-file-ignores` setting (See also #917, #287)

3.7.0 – 2019-01-29

You can view the 3.7.0 milestone on GitHub for more details.

New Dependency Information

- Add dependency on `entrypoints` >= 0.3, < 0.4 (See also #897, #1197)
- Pyflakes has been updated to >= 2.1.0, < 2.2.0 (See also #912, #913)
- pycodestyle has been updated to >= 2.5.0, < 2.6.0 (See also #915)

Features

- Add support for `per-file-ignores` (See also #892, #511, #911, #277)
- Enable use of `float` and `complex` option types (See also #894, #258)
- Improve startup performance by switching from `pkg_resources` to `entrypoints` (See also #897)
- Add metadata for use through the pre-commit git hooks framework (See also #901, #1196)
- Allow physical line checks to return more than one result (See also #902)
- Allow `# noqa:X123` comments without space between the colon and codes list (See also #906, #276)
- Remove broken and unused `flake8.listen` plugin type (See also #907, #663)

3.6.0 – 2018-10-23

You can view the 3.6.0 milestone on GitHub for more details.

New Dependency Information

- pycodestyle has been updated to >= 2.4.0, < 2.5.0 (See also #1068, #652, #869, #881, #1239)
- Pyflakes has been updated to >= 2.0.0, < 2.1.0 (See also #655, #883)
- flake8 requires python 2.x >= 2.7 or python 3.x >= 3.4 (See also #876)

Features

- Add `paths` to allow local plugins to exist outside of `sys.path` (See also #1067, #1237)
- Copy `setup.cfg` files to the temporary git hook execution directory (See also #1299)
- Only skip a file if `# flake8:  noqa` is on a line by itself (See also #259, #873)
- Provide a better user experience for broken plugins (See also #1178)
- Report `E902` when a file passed on the command line does not exist (See also #645, #878)
- Add `--extend-ignore` for extending the default `ignore` instead of overriding it (See also #1061, #1180)

Bugs Fixed

- Respect a formatter's newline setting when printing (See also #1238)
- Fix leaking of processes in the legacy api (See also #650, #879)
- Fix a `SyntaxWarning` for an invalid escape sequence (See also #1186)
- Fix `DeprecationWarning` due to import of `abc` classes from the `collections` module (See also #887)
- Defer `setuptools` import to improve flake8 startup time (See also #1190)
- Fix inconsistent line endings in `FileProcessor.lines` when running under python 3.x (See also #263, #889)

3.5.0 – 2017-10-23

You can view the 3.5.0 milestone on GitHub for more details.

New Dependency Information

- Allow for PyFlakes 1.6.0 (See also #1058)
- Start using new PyCodestyle checks for bare excepts and ambiguous identifier (See also #611)

Features

- Print out information about configuring VCS hooks (See also #586)
- Allow users to develop plugins "local" to a repository without using setuptools. See our documentation on local plugins for more information. (See also #608)

Bugs Fixed

- Catch and helpfully report `UnicodeDecodeError`s when parsing configuration files. (See also #609)

3.4.1 – 2017-07-28

You can view the 3.4.1 milestone on GitHub for more details.

- Fix minor regression when users specify only a `--select` list with items in the enabled/extended select list. (See also #605)

3.4.0 – 2017-07-27

You can view the 3.4.0 milestone on GitHub for more details.

- Refine logic around `--select` and `--ignore` when combined with the default values for each. (See also #572)
- Handle spaces as an alternate separate for error codes, e.g., `--ignore 'E123 E234'`. (See also #580)
- Filter out empty select and ignore codes, e.g., `--ignore E123,,E234`. (See also #581)
- Specify dependencies appropriately in `setup.py` (See also #592)
- Fix bug in parsing `--quiet` and `--verbose` from config files. (See also #1169)
- Remove unused import of `os` in the git hook template (See also #1170)

3.3.0 – 2017-02-06

You can view the 3.3.0 milestone on GitHub for more details.

- Add support for Python 3.6 (via dependencies). **Note** Flake8 does not guarantee that all plugins will support Python 3.6.
- Added unique error codes for all missing PyFlakes messages. (14 new codes, see "Error / Violation Codes")
- Dramatically improve the performance of Flake8. (See also #829)
- Display the local file path instead of the temporary file path when using the git hook. (See also #176)
- Add methods to Report class that will be called when Flake8 starts and finishes processing a file. (See also #183)
- Fix problem where hooks should only check *.py files. (See also #200)
- Fix handling of SyntaxErrors that do not include physical line information. (See also #542)
- Update upper bound on PyFlakes to allow for PyFlakes 1.5.0. (See also #549)
- Update setuptools integration to less eagerly deduplicate packages. (See also #552)
- Force `flake8 --version` to be repeatable between invocations. (See also #554)

3.2.1 – 2016-11-21

You can view the 3.2.1 milestone on GitHub for more details.

- Fix subtle bug when deciding whether to report an on-by-default's violation (See also #189)
- Fix another bug around SyntaxErrors not being reported at the right column and row (See also #191 and #169 for a related, previously fixed bug)
- Fix regression from 2.x where we run checks against explicitly provided files, even if they don't match the filename patterns. (See also #198)

3.2.0 – 2016-11-14

You can view the 3.2.0 milestone on GitHub for more details.

- Allow for pycodestyle 2.2.0 which fixes a bug in E305 (See also #188)

3.1.1 – 2016-11-14

You can view the 3.1.1 milestone on GitHub for more details.

- Do not attempt to install/distribute a `man` file with the Python package; leave this for others to do. (See also #186)
- Fix packaging bug where wheel version constraints specified in setup.cfg did not match the constraints in setup.py. (See also #187)

3.1.0 – 2016-11-14

You can view the 3.1.0 milestone on GitHub for more details.

- Add `--bug-report` flag to make issue reporters' lives easier.
- Collect configuration files from the current directory when using our Git hook. (See also #142, #150, #155)
- Avoid unhandled exceptions when dealing with SyntaxErrors. (See also #146, #170)
- Exit early if the value for `--diff` is empty. (See also #158)
- Handle empty `--stdin-display-name` values. (See also #167)
- Properly report the column number of Syntax Errors. We were assuming that all reports of column numbers were 0-indexed, however, SyntaxErrors report the column number as 1-indexed. This caused us to report a column number that was 1 past the actual position. Further, when combined with SyntaxErrors that occur at a newline, this caused the position to be visually off by two. (See also #169)
- Fix the behaviour of `--enable-extensions`. Previously, items specified here were still ignored due to the fact that the off-by-default extension codes were being left in the `ignore` list. (See also #171)
- Fix problems around `--select` and `--ignore` behaviour that prevented codes that were neither explicitly selected nor explicitly ignored from being reported. (See also #174)
- Truly be quiet when the user specifies `-q` one or more times. Previously, we were showing the if the user specified `-q` and `--show-source`. We have fixed this bug. (See also #177)
- Add new File Processor attribute, `previous_unindented_logical_line` to accommodate pycodestyle 2.1.0. (See also #178)
- When something goes wrong, exit non-zero. (See also #180, #141)
- Add `--tee` as an option to allow use of `--output-file` and printing to standard out.

- Allow the git plugin to actually be lazy when collecting files.
- Allow for pycodestyle 2.1 series and pyflakes 1.3 series.

3.0.4 – 2016-08-08

- Side-step a Pickling Error when using Flake8 with multiprocessing on Unix systems. (See also #1014)
- Fix an Attribute Error raised when dealing with Invalid Syntax. (See also #539)
- Fix an unhandled Syntax Error when tokenizing files. (See also #540)

3.0.3 – 2016-07-30

- Disable `--jobs` for any version of Python on Windows. (See also this Python bug report)
- Raise exception when entry_point in plugin not callable. This raises an informative error when a plugin fails to load because its entry_point is not callable, which can happen with a plugin which is buggy or not updated for the current version of flake8. This is nicer than raising a *PicklingError* about failing to pickle a module (See also #1014)
- Fix `# noqa` comments followed by a `:` and explanation broken by 3.0.0 (See also #1025)
- Always open our output file in append mode so we do not overwrite log messages. (See also #535)
- When normalizing path values read from configuration, keep in context the directory where the configuration was found so that relative paths work. (See also #1036)
- Fix issue where users were unable to ignore plugin errors that were on by default. (See also #1037)
- Fix our legacy API StyleGuide's `init_report` method to actually override the previous formatter. (See also #136)

3.0.2 – 2016-07-26

- Fix local config file discovery. (See also #528)
- Fix indexing of column numbers. We accidentally were starting column indices at 0 instead of 1.
- Fix regression in handling of errors like E402 that rely on a combination of attributes. (See also #530)

3.0.1 – 2016-07-25

- Fix regression in handling of `# noqa` for multiline strings. (See also #1024)
- Fix regression in handling of `--output-file` when not also using `--verbose`. (See also #1026)
- Fix regression in handling of `--quiet`. (See also #1026)
- Fix regression in handling of `--statistics`. (See also #1026)

3.0.0 – 2016-07-25

- Rewrite our documentation from scratch! (https://flake8.pycqa.org)
- Drop explicit support for Pythons 2.6, 3.2, and 3.3.
- Remove dependence on pep8/pycodestyle for file processing, plugin dispatching, and more. We now control all of this while keeping backwards compatibility.
- `--select` and `--ignore` can now both be specified and try to find the most specific rule from each. For example, if you do `--select E --ignore E123` then we will report everything that starts with `E` except for `E123`. Previously, you would have had to do `--ignore E123,F,W` which will also still work, but the former should be far more intuitive.
- Add support for in-line `# noqa` comments to specify **only** the error codes to be ignored, e.g., `# noqa: E123, W503`
- Add entry-point for formatters as well as a base class that new formatters can inherit from. See the documentation for more details.
- Add detailed verbose output using the standard library logging module.
- Enhance our usage of optparse for plugin developers by adding new parameters to the `add_option` that plugins use to register new options.
- Update `--install-hook` to require the name of version control system hook you wish to install a Flake8.
- Stop checking sub-directories more than once via the setuptools command
- When passing a file on standard-in, allow the caller to specify `--stdin-display-name` so the output is properly formatted
- The Git hook now uses `sys.executable` to format the shebang line. This allows Flake8 to install a hook script from a virtualenv that points to that virtualenv's Flake8 as opposed to a global one (without the virtualenv being sourced).
- Print results in a deterministic and consistent ordering when used with multiprocessing
- When using `--count`, the output is no longer written to stderr.
- AST plugins can either be functions or classes and all plugins can now register options so long as there are callable attributes named as we expect.
- Stop forcibly re-adding `.tox`, `.eggs`, and `*.eggs` to `--exclude`. Flake8 2.x started always appending those three patterns to any exclude list (including the default and any user supplied list). Flake8 3 has stopped adding these in, so you may see errors when upgrading due to these patterns no longer being forcibly excluded by default if you have your own exclude patterns specified.

  To fix this, add the appropriate patterns to your exclude patterns list.

  Note: This item was added in November of 2016, as a result of a bug report.

6.1.5 2.x Release Series

2.6.2 - 2016-06-25

- **Bug** Fix packaging error during release process.

2.6.1 - 2016-06-25

- **Bug** Update the config files to search for to include `setup.cfg` and `tox.ini`. This was broken in 2.5.5 when we stopped passing `config_file` to our Style Guide

2.6.0 - 2016-06-15

- **Requirements Change** Switch to pycodestyle as all future pep8 releases will use that package name
- **Improvement** Allow for Windows users on *select* versions of Python to use `--jobs` and multiprocessing
- **Improvement** Update bounds on McCabe
- **Improvement** Update bounds on PyFlakes and blacklist known broken versions
- **Improvement** Handle new PyFlakes warning with a new error code: F405

2.5.5 - 2016-06-14

- **Bug** Fix setuptools integration when parsing config files
- **Bug** Don't pass the user's config path as the config_file when creating a StyleGuide

2.5.4 - 2016-02-11

- **Bug** Missed an attribute rename during the v2.5.3 release.

2.5.3 - 2016-02-11

- **Bug** Actually parse `output_file` and `enable_extensions` from config files

2.5.2 - 2016-01-30

- **Bug** Parse `output_file` and `enable_extensions` from config files
- **Improvement** Raise upper bound on mccabe plugin to allow for version 0.4.0

2.5.1 - 2015-12-08

- **Bug** Properly look for `.flake8` in current working directory (#458)
- **Bug** Monkey-patch `pep8.stdin_get_value` to cache the actual value in stdin. This helps plugins relying on the function when run with multiprocessing. (#460, #462)

2.5.0 - 2015-10-26

- **Improvement** Raise cap on PyFlakes for Python 3.5 support
- **Improvement** Avoid deprecation warnings when loading extensions (#102, #445)
- **Improvement** Separate logic to enable “off-by-default” extensions (#110)
- **Bug** Properly parse options to setuptools Flake8 command (#408)
- **Bug** Fix exceptions when output on stdout is truncated before Flake8 finishes writing the output (#112)
- **Bug** Fix error on OS X where Flake8 can no longer acquire or create new semaphores (#117)

2.4.1 - 2015-05-18

- **Bug** Do not raise a `SystemError` unless there were errors in the setuptools command. (#82, #390)
- **Bug** Do not verify dependencies of extensions loaded via entry-points.
- **Improvement** Blacklist versions of pep8 we know are broken

2.4.0 - 2015-03-07

- **Bug** Print filenames when using multiprocessing and `-q` option. (#74)
- **Bug** Put upper cap on dependencies. The caps for 2.4.0 are:
  - `pep8 < 1.6` (Related to #78)
  - `mccabe < 0.4`
  - `pyflakes < 0.9`

  See also #75
- **Bug** Files excluded in a config file were not being excluded when flake8 was run from a git hook. (#2)
- **Improvement** Print warnings for users who are providing mutually exclusive options to flake8. (#51, #386)
- **Feature** Allow git hook configuration to live in `.git/config`. See the updated VCS hooks docs for more details. (#387)

2.3.0 - 2015-01-04

- **Feature**: Add `--output-file` option to specify a file to write to instead of `stdout`.
- **Bug** Fix interleaving of output while using multiprocessing (#60)

2.2.5 - 2014-10-19

- Flush standard out when using multiprocessing
- Make the check for "# flake8: noqa" more strict

2.2.4 - 2014-10-09

- Fix bugs triggered by turning multiprocessing on by default (again)

  Multiprocessing is forcibly disabled in the following cases:

  - Passing something in via stdin
  - Analyzing a diff
  - Using windows
- Fix –install-hook when there are no config files present for pep8 or flake8.
- Fix how the setuptools command parses excludes in config files
- Fix how the git hook determines which files to analyze (Thanks Chris Buccella!)

2.2.3 - 2014-08-25

- Actually turn multiprocessing on by default

2.2.2 - 2014-07-04

- Re-enable multiprocessing by default while fixing the issue Windows users were seeing.

2.2.1 - 2014-06-30

- Turn off multiple jobs by default. To enable automatic use of all CPUs, use `--jobs=auto`. Fixes #155 and #154.

2.2.0 - 2014-06-22

- New option `doctests` to run Pyflakes checks on doctests too
- New option `jobs` to launch multiple jobs in parallel
- Turn on using multiple jobs by default using the CPU count
- Add support for `python -m flake8` on Python 2.7 and Python 3
- Fix Git and Mercurial hooks: issues #88, #133, #148 and #149
- Fix crashes with Python 3.4 by upgrading dependencies
- Fix traceback when running tests with Python 2.6

- Fix the setuptools command `python setup.py flake8` to read the project configuration

2.1.0 - 2013-10-26

- Add FLAKE8_LAZY and FLAKE8_IGNORE environment variable support to git and mercurial hooks
- Force git and mercurial hooks to repsect configuration in setup.cfg
- Only check staged files if that is specified
- Fix hook file permissions
- Fix the git hook on python 3
- Ignore non-python files when running the git hook
- Ignore .tox directories by default
- Flake8 now reports the column number for PyFlakes messages

2.0.0 - 2013-02-23

- Pyflakes errors are prefixed by an `F` instead of an `E`
- McCabe complexity warnings are prefixed by a `C` instead of a `W`
- Flake8 supports extensions through entry points
- Due to the above support, we **require** setuptools
- We publish the documentation
- Fixes #13: pep8, pyflakes and mccabe become external dependencies
- Split run.py into main.py, engine.py and hooks.py for better logic
- Expose our parser for our users
- New feature: Install git and hg hooks automagically
- By relying on pyflakes (0.6.1), we also fixed #45 and #35

6.1.6 1.x Release Series

1.7.0 - 2012-12-21

- Fixes part of #35: Exception for no WITHITEM being an attribute of Checker for Python 3.3
- Support stdin
- Incorporate @phd's builtins pull request
- Fix the git hook
- Update pep8.py to the latest version

1.6.2 - 2012-11-25

- fixed the NameError: global name 'message' is not defined (#46)

1.6.1 - 2012-11-24

- fixed the mercurial hook, a change from a previous patch was not properly applied
- fixed an assumption about warnings/error messages that caused an exception to be thrown when McCabe is used

1.6 - 2012-11-16

- changed the signatures of the `check_file` function in flake8/run.py, `skip_warning` in flake8/util.py and the `check`, `checkPath` functions in flake8/pyflakes.py.
- fix `--exclude` and `--ignore` command flags (#14, #19)
- fix the git hook that wasn't catching files not already added to the index (#29)
- pre-emptively includes the addition to pep8 to ignore certain lines. Add `# nopep8` to the end of a line to ignore it. (#37)
- `check_file` can now be used without any special prior setup (#21)
- unpacking exceptions will no longer cause an exception (#20)
- fixed crash on non-existent file (#38)

1.5 - 2012-10-13

- fixed the stdin
- make sure mccabe catches the syntax errors as warnings
- pep8 upgrade
- added max_line_length default value
- added Flake8Command and entry points if setuptools is around
- using the setuptools console wrapper when available

1.4 - 2012-07-12

- git_hook: Only check staged changes for compliance
- use pep8 1.2

1.3.1 - 2012-05-19

- fixed support for Python 2.5

1.3 - 2012-03-12

- fixed false W402 warning on exception blocks.

1.2 - 2012-02-12

- added a git hook
- now Python 3 compatible
- mccabe and pyflakes have warning codes like pep8 now

1.1 - 2012-02-14

- fixed the value returned by –version
- allow the flake8: header to be more generic
- fixed the “hg hook raises ‘physical lines’” bug
- allow three argument form of raise
- now uses setuptools if available, for ‘develop’ command

1.0 - 2011-11-29

- Deactivates by default the complexity checker
- Introduces the complexity option in the HG hook and the command line.

6.1.7 0.x Release Series

0.9 - 2011-11-09

- update pep8 version to 0.6.1
- mccabe check: gracefully handle compile failure

0.8 - 2011-02-27

- fixed hg hook
- discard unexisting files on hook check

0.7 - 2010-02-18

- Fix pep8 initialization when run through Hg
- Make pep8 short options work when run through the command line
- Skip duplicates when controlling files via Hg

0.6 - 2010-02-15

- Fix the McCabe metric on some loops

CHAPTER

SEVEN

GENERAL INDICES

- genindex
- Index of Documented Public Modules
- *Glossary of terms*

INDEX

Symbols

--append-config
    flake8 command line option, 22
--benchmark
    flake8 command line option, 24
--bug-report
    flake8 command line option, 24
--builtins
    flake8 command line option, 22
--color
    flake8 command line option, 13
--config
    flake8 command line option, 22
--count
    flake8 command line option, 13
--disable-noqa
    flake8 command line option, 19
--doctests
    flake8 command line option, 23
--enable-extensions
    flake8 command line option, 20
--exclude
    flake8 command line option, 13
--exclude-from-doctest
    flake8 command line option, 23
--exit-zero
    flake8 command line option, 20
--extend-exclude
    flake8 command line option, 14
--extend-ignore
    flake8 command line option, 16
--extend-select
    flake8 command line option, 19
--filename
    flake8 command line option, 14
--format
    flake8 command line option, 15
--hang-closing
    flake8 command line option, 15
--help
    flake8 command line option, 12
--ignore
    flake8 command line option, 16
--include-in-doctest
    flake8 command line option, 23
--indent-size
    flake8 command line option, 18
--isolated
    flake8 command line option, 22
--jobs
    flake8 command line option, 21
--max-complexity
    flake8 command line option, 25
--max-doc-length
    flake8 command line option, 17
--max-line-length
    flake8 command line option, 17
--output-file
    flake8 command line option, 21
--per-file-ignores
    flake8 command line option, 17
--quiet
    flake8 command line option, 12
--require-plugins
    flake8 command line option, 20
--select
    flake8 command line option, 18
--show-source
    flake8 command line option, 19
--statistics
    flake8 command line option, 20
--stdin-display-name
    flake8 command line option, 14
--tee
    flake8 command line option, 21
--verbose
    flake8 command line option, 12
--version
    flake8 command line option, 12
-h
    flake8 command line option, 12
-q
    flake8 command line option, 12
-v

flake8 command line option, 12

A

after_init() (*flake8.formatting.base.BaseFormatter method*), 41
after_init() (*flake8.formatting.default.Default method*), 58

B

BaseFormatter (*class in flake8.formatting.base*), 41
beginning() (*flake8.formatting.base.BaseFormatter method*), 42

C

check, **4**
class, **4**

D

Default (*class in flake8.formatting.default*), 57

E

error, **4**
error class, **4**
error code, **4**

F

filename (*flake8.formatting.base.BaseFormatter attribute*), 41
finished() (*flake8.formatting.base.BaseFormatter method*), 42
flake8 command line option
- --append-config, 22
- --benchmark, 24
- --bug-report, 24
- --builtins, 22
- --color, 13
- --config, 22
- --count, 13
- --disable-noqa, 19
- --doctests, 23
- --enable-extensions, 20
- --exclude, 13
- --exclude-from-doctest, 23
- --exit-zero, 20
- --extend-exclude, 14
- --extend-ignore, 16
- --extend-select, 19
- --filename, 14
- --format, 15
- --hang-closing, 15
- --help, 12
- --ignore, 16
- --include-in-doctest, 23
- --indent-size, 18
- --isolated, 22
- --jobs, 21
- --max-complexity, 25
- --max-doc-length, 17
- --max-line-length, 17
- --output-file, 21
- --per-file-ignores, 17
- --quiet, 12
- --require-plugins, 20
- --select, 18
- --show-source, 19
- --statistics, 20
- --stdin-display-name, 14
- --tee, 21
- --verbose, 12
- --version, 12
- -h, 12
- -q, 12
- -v, 12

fnmatch() (*in module flake8.utils*), 62
format() (*flake8.formatting.base.BaseFormatter method*), 42
formatter, **4**

H

handle() (*flake8.formatting.base.BaseFormatter method*), 42

I

is_using_stdin() (*in module flake8.utils*), 62

M

mccabe, **4**

N

newline (*flake8.formatting.base.BaseFormatter attribute*), 41
normalize_path() (*in module flake8.utils*), 61
normalize_paths() (*in module flake8.utils*), 62

O

options (*flake8.formatting.base.BaseFormatter attribute*), 41
output_fd (*flake8.formatting.base.BaseFormatter attribute*), 41

P

parse_comma_separated_list() (*in module flake8.utils*), 61
plugin, **4**
pycodestyle, **4**
pyflakes, **4**

Pylint (*class in flake8.formatting.default*), 58
Python Enhancement Proposals
    PEP 257, 30
    PEP 8, 4

S

show_benchmarks() (*flake8.formatting.base.BaseFormatter method*), 42
show_source() (*flake8.formatting.base.BaseFormatter method*), 42
show_statistics() (*flake8.formatting.base.BaseFormatter method*), 43
start() (*flake8.formatting.base.BaseFormatter method*), 43
stdin_get_value() (*in module flake8.utils*), 62
stop() (*flake8.formatting.base.BaseFormatter method*), 43

V

violation, **4**

W

warning, **4**
write() (*flake8.formatting.base.BaseFormatter method*), 43