# 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  
  3.2 flake8 34  

4 Plugin Developer Guide 37  
  4.1 Writing Plugins for Flake8 37  

5 Contributor Guide 49  
  5.1 Exploring Flake8’s Internals 49  

6 Release Notes and History 85  
  6.1 Release Notes and History 85  

7 General Indices 99  

Python Module Index 101
CHAPTER 1

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 *ignore* only one specific warning or error:

flake8 --ignore E24,W504 path/to/code/

Please read our user guide for more information about how to use and configure Flake8.
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 GitLab, 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:
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.
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
- called by Git or Mercurial on or around committing

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:

```bash
$ 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 2.7 and Python 3.5) and you need to call a specific version. In that case, you will have much better results using:

```bash
$ python2.7 -m flake8
```

Or
$ python3.5 -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 2.7 and Python 3.5. 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
- `-v, --verbose` Print more information about what is happening in flake8. This option is repeatable and will increase verbosity each time it is repeated.
- `-q, --quiet` Report only file names, or nothing. This option is repeatable.
- `--count` Print total number of errors and warnings to standard error and set the exit code to 1 if total is not empty.
- `--diff` Report changes only within line number ranges in the unified diff provided on standard in by the user.
- `--exclude=patterns` Comma-separated list of files or directories to exclude. (Default: .svn,CVS,.bzr,.hg,.git,.pycache, .tox)
- `--filename=patterns` Only check for filenames matching the patterns in this comma-separated list. (Default: *.py)
- `--format=format` Format errors according to the chosen formatter.
- `--hang-closing` Hang closing bracket instead of matching indentation of opening bracket's line.
- `--ignore=errors` Comma-separated list of errors and warnings to ignore
(or skip). For example, `--ignore=E4,E51,W234`.
(Default: E121,E123,E126,E226,E24,E704)

---extend-ignore=errors
Comma-separated list of errors and warnings to add to the list of ignored ones. For example, `--extend-ignore=E4,E51,W234`.

---max-line-length=n
Maximum allowed line length for the entirety of this run. (Default: 79)

---select=errors
Comma-separated list of errors and warnings to enable. For example, `--select=E4,E51,W234`.

---disable-noqa
Disable the effect of "# noqa". This will report errors on lines with "# noqa" at the end.

---show-source
Show the source generate each error or warning.

---statistics
Count errors and warnings.

---enabled-extensions=ENABLED_EXTENSIONS
Enable plugins and extensions that are otherwise disabled by default.

---exit-zero
Exit with status code "0" even if there are errors.

-j JOBS, --jobs=JOBS
Number of subprocesses to use to run checks in parallel. This is ignored on Windows. The default, "auto", will auto-detect the number of processors available to use. (Default: auto)

---output-file=OUTPUT_FILE
Redirect report to a file.

---tee
Write to stdout and output-file.

---append-config=APPEND_CONFIG
Provide extra config files to parse in addition to the files found by Flake8 by default. These files are the last ones read and so they take the highest precedence when multiple files provide the same option.

---config=CONFIG
Path to the config file that will be the authoritative config source. This will cause Flake8 to ignore all other configuration files.

---isolated
Ignore all configuration files.

---builtins=BUILTINS
define more built-ins, comma separated

---doctests
check syntax of the doctests

---include-in-doctest=INCLUDE_IN_DOCTEST
Run doctests only on these files

---exclude-from-doctest=EXCLUDE_FROM_DOCTEST
Skip these files when running doctests

Installed plugins: pyflakes: 1.0.0, pep8: 1.7.0

### 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
```

### 3.1. Using Flake8

---
Configuration Locations

Flake8 supports storing its configuration in the following places:

- Your top-level user directory
- In your project in one of `setup.cfg`, `tox.ini`, or `.flake8`.

“User” Configuration

Flake8 allows a user to use “global” configuration file to store preferences. The user configuration file is expected to be stored somewhere in the user’s “home” directory.

- On Windows the “home” directory will be something like `C:\Users\sigmavirus24`, a.k.a, `~`.
- On Linux and other Unix like systems (including OS X) we will look in `~`.

Note that Flake8 looks for `~\flake8` on Windows and `~/.config/flake8` on Linux and other Unix systems. User configuration files use the same syntax as Project Configuration files. Keep reading to see that syntax.

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
  |      |        |---- _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:

```ini
[flake8]
ignore = D203
exclude = .git, __pycache__, docs/source/conf.py, old, build, dist
max-complexity = 10
```

This is equivalent to:

```bash
$ flake8 --ignore D203 --exclude .git, __pycache__, docs/source/conf.py, old, build, dist
```

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

```ini
[flake8]
ignore = D203
exclude =
    .git,
    __pycache__,
    docs/source/conf.py,
    old,
    build,
    dist
max-complexity = 10
```

This would allow us to add comments for why we’re excluding items, e.g.,

```ini
[flake8]
ignore = D203
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:** 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`.  

3.1. Using **Flake8**
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`:

```ini
[flake8]
# it's not a bug that we aren't using all of hacking, ignore:
# F812: list comprehension redefines ...
# 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
```

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

```ini
[flake8]
# it's not a bug that we aren't using all of hacking
ignore =
    # F812: list comprehension redefines ...
    F812,
    # 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 --count`
• `flake8 --diff`
• `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 --select`
• `flake8 --disable-noqa`
• `flake8 --show-source`
• `flake8 --statistics`
• `flake8 --enable-extensions`
• `flake8 --exit-zero`
• `flake8 --install-hook`
• `flake8 --jobs`
• `flake8 --output-file`
• `flake8 --tee`
• `flake8 --append-config`
• `flake8 --config`
• `flake8 --isolated`
• `flake8 --builtins`
• `flake8 --doctest`
• `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 be specified in config files.
Example config file usage:

verbose = 2

-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

--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
**--diff**

*Go back to index*

Use the unified diff provided on standard in to only check the modified files and report errors included in the diff.

Command-line example:

```
$ git diff -u | flake8 --diff
```

This *can not* be specified in config files.

**--exclude=<patterns>**

*Go back to index*

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

This defaults to: `.svn, CVS, .bazaar, .hg, .git, __pycache__, .tox`

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__
```

**--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:
**--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
```

**--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`

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:

```bash
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.

```bash
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:

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

url = {  
    'http://...'
}
```

This defaults to: 79

Command-line example:

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

This can be specified in config files.

Example config file usage:

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

---

### 3.1. Using Flake8
--select=<errors>

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,C

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:

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

--disable-noqa

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:

```python
disable_noqa = True
disable_noqa = True
```

--show-source

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

--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 effectively add themselves to the default ignore list.
    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.

--install-hook=VERSION_CONTROL_SYSTEM
    Go back to index
    Install a hook for your version control system that is executed before or during commit.
    The available options are:
        • git
• mercurial

Command-line usage:

$ flake8 --install-hook=git
$ flake8 --install-hook=mercurial

This can not be specified in config files.

--jobs=<n>

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

**Note:** This option is ignored on Windows because multiprocessing does not support Windows across all supported versions of Python.

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:

```ini
jobs = 8
```

--output-file=<path>

Redirect all output to the specified file.

Command-line example:

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

This can be specified in config files.

Example config file usage:

```ini
output-file = output.txt
output_file = output.txt
```

--tee

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:

```yaml
output-file = output.txt
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,
    _LN
```

**--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
```

**--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:
flake8 Documentation, Release 3.7.4

```
exlude-in-doctest =
dir/subdir/file.py,
dir/other/file.py
exclude_in_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"
    },
    {
      "plugin": "pycodestyle",
      "version": "2.0.0"
    }
  ]
}
```

(continues on next page)
This can not be specified in config files.

\[--\text{max-complexity} = <n>\]

Go back to index

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

This option is provided by the \texttt{mccabe} dependency’s \texttt{Flake8} plugin.

Command-line usage:

\$ \texttt{flake8 --max-complexity 15 dir/}

This can be specified in config files.

Example config file usage:

\[
\text{max-complexity} = 15
\]

### 3.1.4 Error / Violation Codes

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

Flake8 installs \texttt{pycodestyle}, \texttt{pyflakes}, and \texttt{mccabe} by default and generates its own \textit{error codes} for \texttt{pyflakes}:

<table>
<thead>
<tr>
<th>Code</th>
<th>Example Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>F401</td>
<td>module imported but unused</td>
</tr>
<tr>
<td>F402</td>
<td>import module from line (N) shadowed by loop variable</td>
</tr>
<tr>
<td>F403</td>
<td>‘from module import *’ used; unable to detect undefined names</td>
</tr>
<tr>
<td>F404</td>
<td>future import(s) \texttt{name} after other statements</td>
</tr>
<tr>
<td>F405</td>
<td>\texttt{name} may be undefined, or defined from star imports: \texttt{module}</td>
</tr>
<tr>
<td>F406</td>
<td>‘from module import *’ only allowed at module level</td>
</tr>
<tr>
<td>F407</td>
<td>an undefined \texttt{<strong>future</strong>} feature name was imported</td>
</tr>
<tr>
<td>F601</td>
<td>dictionary key \texttt{name} repeated with different values</td>
</tr>
<tr>
<td>F602</td>
<td>dictionary key variable \texttt{name} repeated with different values</td>
</tr>
<tr>
<td>F621</td>
<td>too many expressions in an assignment with star-unpacking</td>
</tr>
<tr>
<td>F622</td>
<td>two or more starred expressions in an assignment ((a, *, b, *c = d))</td>
</tr>
<tr>
<td>F631</td>
<td>assertion test is a tuple, which are always \texttt{True}</td>
</tr>
<tr>
<td>F632</td>
<td>use \texttt{==/!=} to compare \texttt{str, bytes, int} literals</td>
</tr>
<tr>
<td>F633</td>
<td>assertion test is a tuple, which are always \texttt{True}</td>
</tr>
</tbody>
</table>

Continued on next page
Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>Code</th>
<th>Example Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>F701</td>
<td>a break statement outside of a while or for loop</td>
</tr>
<tr>
<td>F702</td>
<td>a continue statement outside of a while or for loop</td>
</tr>
<tr>
<td>F703</td>
<td>a continue statement in a finally block in a loop</td>
</tr>
<tr>
<td>F704</td>
<td>a yield or yield from statement outside of a function</td>
</tr>
<tr>
<td>F705</td>
<td>a return statement with arguments inside a generator</td>
</tr>
<tr>
<td>F706</td>
<td>a return statement outside of a function/method</td>
</tr>
<tr>
<td>F707</td>
<td>an except: block as not the last exception handler</td>
</tr>
<tr>
<td>F721</td>
<td>syntax error in doctest</td>
</tr>
<tr>
<td>F722</td>
<td>syntax error in forward annotation</td>
</tr>
<tr>
<td>F723</td>
<td>syntax error in type comment</td>
</tr>
<tr>
<td>F811</td>
<td>redefinition of unused name from line N</td>
</tr>
<tr>
<td>F812</td>
<td>list comprehension redefines name from line N</td>
</tr>
<tr>
<td>F821</td>
<td>undefined name name</td>
</tr>
<tr>
<td>F822</td>
<td>undefined name name in <strong>all</strong></td>
</tr>
<tr>
<td>F823</td>
<td>local variable name ... referenced before assignment</td>
</tr>
<tr>
<td>F831</td>
<td>duplicate argument name in function definition</td>
</tr>
<tr>
<td>F841</td>
<td>local variable name is assigned to but never used</td>
</tr>
<tr>
<td>F901</td>
<td>raise NotImplemented should be raise NotImplementedError</td>
</tr>
</tbody>
</table>

Note that some of these entries behave differently on Python 2 and Python 3, for example F812 is specific to Python 2 only.

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.,

```bash
$ 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

```python
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:

```python
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:

```bash
# 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.

**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:

```python
# 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
```

```
exa...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:

```python
# example.py
import os

def foo():
    var = 1
    print("Hello"
        "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 **plugins** 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-future**.

**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>
$ python2.7 --m pip install <plugin-name>
$ python3 --m pip install <plugin-name>
$ python3.4 --m pip install <plugin-name>
$ python3.5 --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 **errors**. 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`:

---

*3.1. Using Flake8*
- repo: https://gitlab.com/pycqa/flake8
  rev: ''  # pick a git hash / tag to point to
  hooks:
    - id: flake8
      exclude: ^testing/(data|examples)/

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.

Built-in Hook Integration

**Note:** It is strongly suggested to use Flak8 via pre-commit over the built-in hook mechanisms. pre-commit smooths out many of the rough edges of git and is much more battle-tested than the Flak8 hook implementation.

Flake8 can be integrated into your development workflow in many ways. A default installation of Flake8 can install pre-commit hooks for both Git and Mercurial. To install a built-in hook, you can use the flake8 --install-hook command-line option. For example, you can install a git pre-commit hook by running:

```
$ flake8 --install-hook git
```

This will install the pre-commit hook into .git/hooks/. Alternatively, you can install the mercurial commit hook by running

```
$ flake8 --install-hook mercurial
```

Preventing Commits

By default, Flak8 does not prevent you from creating a commit with these hooks. Both hooks can be configured to be strict easily.

Both our Git and Mercurial hooks check for the presence of flake8.strict in each VCS’ config. For example, you might configure this like so:

```
$ git config --bool flake8.strict true
$ hg config flake8.strict true
```

Checking All Modified Files Currently Tracked

**Note:** Mercurial does not have the concept of an index or “stage” as best as I understand.

---

30 Chapter 3. User Guide
Flake8 aims to make smart choices that keep things fast for users where possible. As a result, the Flake8 Git pre-commit will default to only checking files that have been staged (i.e., added to the index). If, however, you are keen to be lazy and not independently add files to your git index, you can set flake8.lazy to true (similar to how you would set flake8.strict above) and this will check all tracked files.

This is to support users who often find themselves doing things like:

```
$ git commit -a
```

**Note:** If you have files you have not yet added to the index, Flake8 will not see these and will not check them for you. You must git-add them first.

### 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:

```python
from flake8.api import legacy as flake8

style_guide = flake8.get_style_guide(ignores=['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.,

```python
style_guide = flake8.get_style_guide(
    ignores=['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

```
StyleGuide.check_files(paths=None)
Run collected checks on the files provided.
This will check the files passed in and return a Report instance.
Parameters paths (list) – List of filenames (or paths) to check.
Returns Object that mimic’s Flake8 2.0’s Reporter class.
Return type flake8.api.legacy.Report
```

```
StyleGuide.excluded(filename, parent=None)
Determine if a file is excluded.
Parameters
• filename (str) – Path to the file to check if it is excluded.
• parent (str) – Name of the parent directory containing the file.
Returns True if the filename is excluded, False otherwise.
Return type bool
```

```
StyleGuide.init_report(reporter=None)
Set up a formatter for this run of Flake8.
```

```
StyleGuide.input_file(filename, lines=None, expected=None, line_offset=0)
Run collected checks on a single file.
This will check the file passed in and return a Report instance.
Parameters
• filename (str) – The path to the file to check.
• lines (list) – Ignored since Flake8 3.0.
• expected – Ignored since Flake8 3.0.
• line_offset (int) – Ignored since Flake8 3.0.
Returns Object that mimic’s Flake8 2.0’s Reporter class.
Return type flake8.api.legacy.Report
```

**Warning:** These are not perfectly backwards compatible. Not all arguments are respected, 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

```
Report.get_statistics(violation)
Get the list of occurrences of a violation.
Returns List of occurrences of a violation formatted as: {Count} {Error Code} {Message}, e.g., 8 E531 Some error message about the error
Return type list
```
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 maleable.

**Autogenerated Legacy Documentation**

Module containing shims around Flake8 2.x behaviour.

Previously, users would import `get_style_guide()` from `flake8.engine`. In 3.0 we no longer have an “engine” module but we maintain the API from it.

```
flake8.api.legacy.get_style_guide(**kwargs)
```

Provision a StyleGuide for use.

**Parameters**

- `**kwargs` – Keyword arguments that provide some options for the StyleGuide.

**Returns**

- An initialized StyleGuide

**Return type**

*StyleGuide*

```
class flake8.api.legacy.StyleGuide(application)
```

Public facing object that mimic’s Flake8 2.0’s StyleGuide.

**Note:** There are important changes in how this object behaves compared to the StyleGuide object provided in Flake8 2.x.

**Warning:** This object should not be instantiated directly by users.

Changed in version 3.0.0.

```
options
```

Return application’s options.

An instance of `optparse.Values` containing parsed options.

```
paths
```

Return the extra arguments passed as paths.

```
class flake8.api.legacy.Report(application)
```

Public facing object that mimic’s Flake8 2.0’s API.

**Note:** There are important changes in how this object behaves compared to the object provided in Flake8 2.x.

**Warning:** This should not be instantiated by users.

Changed in version 3.0.0.

```
total_errors
```

Return the total number of errors.
3.2 flake8

3.2.1 SYNOPSIS

flake8 [options] [<path> <path> ...]
flake8 --help

3.2.2 DESCRIPTION

flake8 is a command-line utility for enforcing style consistency across Python projects. By default it includes lint checks provided by the PyFlakes project, PEP-0008 inspired style checks provided by the PyCodeStyle project, and McCabe complexity checking provided by the McCabe project. It will also run third-party extensions if they are found and installed.

3.2.3 OPTIONS

It is important to note that third-party extensions may add options which are not represented here. To see all options available in your installation, run:

flake8 --help

All options available as of Flake8 3.1.0:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--version</td>
<td>show program's version number and exit</td>
</tr>
<tr>
<td>-h, --help</td>
<td>show this help message and exit</td>
</tr>
<tr>
<td>-v, --verbose</td>
<td>Print more information about what is happening in flake8. This option is repeatable and will increase verbosity each time it is repeated.</td>
</tr>
<tr>
<td>-q, --quiet</td>
<td>Report only file names, or nothing. This option is repeatable.</td>
</tr>
<tr>
<td>--count</td>
<td>Print total number of errors and warnings to standard error and set the exit code to 1 if total is not empty.</td>
</tr>
<tr>
<td>--diff</td>
<td>Report changes only within line number ranges in the unified diff provided on standard in by the user.</td>
</tr>
<tr>
<td>--exclude=patterns</td>
<td>Comma-separated list of files or directories to exclude. (Default: .svn,CVS,.bzd,.hg,.git,.pycache,.tox,.eggs,*.*egg)</td>
</tr>
<tr>
<td>--filename=patterns</td>
<td>Only check for filenames matching the patterns in this comma-separated list. (Default: *.py)</td>
</tr>
<tr>
<td>--stdin-display-name=STDIN_DISPLAY_NAME</td>
<td>The name used when reporting errors from code passed via stdin. This is useful for editors piping the file contents to flake8. (Default: stdin)</td>
</tr>
<tr>
<td>--format=format</td>
<td>Format errors according to the chosen formatter.</td>
</tr>
<tr>
<td>--hang-closing</td>
<td>Hang closing bracket instead of matching indentation of opening bracket's line.</td>
</tr>
<tr>
<td>--ignore=errors</td>
<td>Comma-separated list of errors and warnings to ignore (or skip). For example, <code>--ignore=E4,E51,W234</code>. (Default: E121,E123,E126,E226,E24,E704,W503,W504)</td>
</tr>
<tr>
<td>--max-line-length=n</td>
<td>Maximum allowed line length for the entirety of this run. (Default: 79)</td>
</tr>
</tbody>
</table>
--select=errors  Comma-separated list of errors and warnings to enable.
   For example, `--select=E4,E51,W234`. (Default: E,F,W,C90)
--disable-noqa   Disable the effect of "# noqa". This will report
   errors on lines with "# noqa" at the end.
--show-source    Show the source generate each error or warning.
--statistics     Count errors and warnings.
--enable-extensions=ENABLE_EXTENSIONS
   Enable plugins and extensions that are otherwise
   disabled by default
--exit-zero      Exit with status code "0" even if there are errors.
--install-hook=INSTALL_HOOK
   Install a hook that is run prior to a commit for the
   supported version control system.
-j JOBS, --jobs=JOBS  Number of subprocesses to use to run checks in
   parallel. This is ignored on Windows. The default, "auto", will auto-detect the number of processors
   available to use. (Default: auto)
--output-file=OUTPUT_FILE
   Redirect report to a file.
--tee           Write to stdout and output-file.
--append-config=APPEND_CONFIG
   Provide extra config files to parse in addition to the
   files found by Flake8 by default. These files are the
   last ones read and so they take the highest precedence
   when multiple files provide the same option.
--config=CONFIG  Path to the config file that will be the authoritative
   config source. This will cause Flake8 to ignore all
   other configuration files.
--isolated       Ignore all configuration files.
--benchmark     Print benchmark information about this run of Flake8
--bug-report    Print information necessary when preparing a bug
   report
--builtins=BUILTINS  define more built-ins, comma separated
--doctests      check syntax of the doctests
--include-in-doctest=INCLUDE_IN_DOCTEST
   Run doctests only on these files
--exclude-from-doctest=EXCLUDE_FROM_DOCTEST
   Skip these files when running doctests
--max-complexity=MAX_COMPLEXITY
   McCabe complexity threshold

3.2.4 EXAMPLES

Simply running flake8 against the current directory:

```
flake8
flake8 .
```

Running flake8 against a specific path:

```
flake8 path/to/file.py
```

Ignoring violations from flake8:
Only report certain violations:

```
flake8 --ignore E101
flake8 --ignore E1,E202
```

Analyzing only a diff:

```
git diff -U0 | flake8 --diff -
```

Generate information for a bug report:

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

### 3.2.5 SEE ALSO

Flake8 documentation: http://flake8.pycqa.org


PyCodeStyle documentation: http://pycodestyle.pycqa.org

PyFlakes: https://github.com/pycqa/pyflakes

McCabe: https://github.com/pycqa/mccabe

### 3.2.6 BUGS

Please report all bugs to https://gitlab.com/pycqa/flake8
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

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:

```python
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://gitlab.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 :: 2",
        "Programming Language :: Python :: 3",
        "Topic :: Software Development :: Libraries :: Python Modules",
        "Topic :: Software Development :: Quality Assurance",
    ],
)
```

(continues on next page)
Note specifically these lines:

```python
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 three 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:

```python
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:

```python
X101 = flake8_example.ExamplePlugin
```

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

```python
X10 = flake8_example.ExamplePlugin
```

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

```python
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. Consider using an entry point with 3 letters followed by 3 numbers (i.e. ABC123).
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.utils.parameters_for()`. `flake8.plugins.manager.Plugin.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 `optparse` 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`.

```python
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. (Default: %default),
    )
```

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 `optparse`'s willingness to display that in the help text with `%default`.

```python
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.'
    )
```

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.

### 4.1. Writing Plugins for Flake8
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 `optparse`.

### 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 `optparse.Values`, and the extra arguments that were not parsed by the `OptionManager`. If that fails, we will just pass the `optparse.Values`. In other words, your `parse_options` callable will have one of the following signatures:

```python
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:

```python
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.

```python
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**

```python
class flake8.formatting.base.BaseFormatter(options)
    Class defining the formatter interface.

    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.

    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.

    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.

    format(error)
        Format an error reported by Flake8.

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

            Parameters error (flake8.style_guide.Violation) – This will be an instance of Violation.

            Returns The formatted error string.

            Return type str

    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 (flake8.style_guide.Violation) – This will be an instance of Violation.

    show_benchmarks(benchmarks)
        Format and print the benchmarks.

    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.
```

4.1. Writing Plugins for Flake8
Parameters **error** *(flake8.style_guide.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

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

**start** *
Prepare the formatter to receive input.
This defaults to initializing `output_fd` if `filename`.

**stop** *
Clean up after reporting is finished.

**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)* – The formatted string to print or write.
- **source** *(str)* – The source code that has been formatted and associated with the line of output.

### Writing Plugins For Flake8 2 and 3

Plugins have existed for Flake8 2.x for a few years. There are a number of these on PyPI already. While it did not seem reasonable for Flake8 to attempt to provide a backwards compatible shim for them, we did decide to try to document the easiest way to write a plugin that’s compatible across both versions.

**Note:** If your plugin does not register options, it should Just Work.

The only two breaking changes in Flake8 3.0 is the fact that we no longer check the option parser for a list of strings to parse from a config file and we no longer patch pep8 or pycodestyle’s stdin_get_value functions. On Flake8 2.x, to have an option parsed from the configuration files that Flake8 finds and parses you would have to do something like:

```python
parser.add_option('-X', '--example-flag', type='string',
                 help='...')
parser.config_options.append('example-flag')
```

For Flake8 3.0, we have added three arguments to the `add_option()` method you will call on the parser you receive:

- **parse_from_config** which expects True or False
  When True, Flake8 will parse the option from the config files Flake8 finds.
- **comma_separated_list** which expects True or False
  When True, Flake8 will split the string intelligently and handle extra whitespace. The parsed value will be a list.
- normalize_paths which expects True or False

When True, Flake8 will:

- remove trailing path separators (i.e., os.path.sep)
- return the absolute path for values that have the separator in them

All three of these options can be combined or used separately.

### Parsing Options from Configuration Files

The example from Flake8 2.x now looks like:

```python
parser.add_option('-X', '--example-flag', type='string',
                  parse_from_config=True,
                  help='...')
```

### Parsing Comma-Separated Lists

Now let’s imagine that the option we want to add is expecting a comma-separated list of values from the user (e.g., `--select E123,W503,F405`). Flake8 2.x often forced users to parse these lists themselves since pep8 special-cased certain flags and left others on their own. Flake8 3.0 adds `comma_separated_list` so that the parsed option is already a list for plugin authors. When combined with `parse_from_config` this means that users can also do something like:

```python
example-flag =
    first,
    second,
    third,
    fourth,
    fifth
```

And Flake8 will just return the list:

```python
['first', 'second', 'third', 'fourth', 'fifth']
```

### Normalizing Values that Are Paths

Finally, let’s imagine that our new option wants a path or list of paths. To ensure that these paths are semi-normalized (the way Flake8 2.x used to work) we need only pass `normalize_paths=True`. If you have specified `comma_separated_list=True` then this will parse the value as a list of paths that have been normalized. Otherwise, this will parse the value as a single path.

### Option Handling on Flake8 2 and 3

To ease the transition, the Flake8 maintainers have released flake8-polyfill. flake8-polyfill provides a convenience function to help users transition between Flake8 2 and 3 without issue. For example, if your plugin has to work on Flake8 2.x and 3.x but you want to take advantage of some of the new options to `add_option`, you can do
from flake8_polyfill import options

class MyPlugin(object):
    @classmethod
    def add_options(cls, parser):
        options.register(
            parser,
            '--application-names', default='', type='string',
            help='Names of the applications to be checked.',
            parse_from_config=True,
            comma_separated_list=True,
        )
        options.register(
            parser,
            '--style-name', default='', type='string',
            help='The name of the style convention you want to use',
            parse_from_config=True,
        )
        options.register(
            parser,
            '--application-paths', default='', type='string',
            help='Locations of the application code',
            parse_from_config=True,
            comma_separated_list=True,
            normalize_paths=True,
        )

    @classmethod
    def parse_options(cls, parsed_options):
        cls.application_names = parsed_options.application_names
        cls.style_name = parsed_options.style_name
        cls.application_paths = parsed_options.application_paths

flake8-polyfill will handle these extra options using callbacks to the option parser. The project has direct
replications of the functions that Flake8 uses to provide the same functionality. This means that the values
you receive should be identically parsed whether you’re using Flake8 2.x or 3.x.

flake8_polyfill.options.register(parser, *args, **kwargs)

Register an option for the Option Parser provided by Flake8.

Parameters

- **parser** – The option parser being used by Flake8 to handle command-line options.
- **args** – Positional arguments that you might otherwise pass to add_option.
- **kwargs** – Keyword arguments you might otherwise pass to add_option.

Standard In Handling on Flake8 2.5, 2.6, and 3

After releasing Flake8 2.6, handling standard-in became a bit trickier for some plugins. Flake8 2.5 and earlier
had started monkey-patching pep8’s stdin_get_value function. 2.6 switched to pycodestyle and only monkey-
patched that. 3.0 has its own internal implementation and uses that but does not directly provide anything for plugins
using pep8 and pycodestyle’s stdin_get_value function. flake8-polyfill provides this functionality for plugin developers via its flake8_polyfill.stdin module.

If a plugin needs to read the content from stdin, it can do the following:
from flake8_polyfill import stdin

stdin.monkey_patch('pep8')  # To monkey-patch only pep8
stdin.monkey_patch('pycodestyle')  # To monkey-patch only pycodestyle
stdin.monkey_patch('all')  # To monkey-patch both pep8 and pycodestyle

Further, when using all, flake8-polyfill does not require both packages to be installed but will attempt to monkey-patch both and will silently ignore the fact that pep8 or pycodestyle is not installed.

flake8_polyfill.stdin.monkey_patch(which)
    Monkey-patch the specified module with the appropriate stdin.

    On Flake8 2.5 and lower, Flake8 would monkey-patch pep8.stdin_get_value for everyone. This avoided problems where stdin might be exhausted.

    On Flake8 2.6, Flake8 stopped patching pep8 and started monkey-patching pycodestyle.
    stdin_get_value.

    On Flake8 3.x, Flake8 has no need to monkey-patch either pep8 or pycodestyle.

This function accepts three parameters:

- pep8
- pycodestyle
- all

“all” is a special value that will monkey-patch both “pep8” and “pycodestyle”.

**Parameters** which (**str**) – The name of the module to patch.

**Returns** Nothing.

**Return type** NoneType
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 GitLab. 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 [GitLab](https://gitlab.com). 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*

**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 documenting 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>`.
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 minimum 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.

```python
# src/flake8/main/git.py
def hook(lazy=False, strict=False):
    
    # 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 bool 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.
```

(continues on next page)
Note that because the parameters `hook` and `strict` are simply boolean parameters, we inline the type declaration for those parameters, e.g.,

```python
:param bool strict:
    If True, return the total number of errors/violations found by Flake8. This will cause the hook to fail.

:param bool lazy:
    Also note that we begin the description of the parameter on a new-line and indented 4 spaces.

On the other hand, we also separate the parameter type declaration in some places where the name is a little longer, e.g.,

```python
# src/flake8/formatting/base.py
def format(self, error):
    """Format an error reported by Flake8. This method **must** be implemented by subclasses.

:param error:
    This will be an instance of `flake8.style_guide.Error`.
:type error:
    `flake8.style_guide.Error`

:returns:
    The formatted error string.
:rtype:
    str
"""
```

Here we've separated `:param error:` and `:type error:`.

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:

5.1. Exploring Flake8's Internals
• 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():
        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('{count:<5} {error_code} {message}'.format(
            count=count,

(continues on next page)"""
```
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,

```python
# 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)
- Subtly breaking CLI changes (3.0)
- 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/releases/` 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 2.7 (a.k.a., `tox -e py27`)
- Python 3.6 (a.k.a., `tox -e py36`)
- Python 3.7 (a.k.a., `tox -e py37`)
- PyPy (a.k.a., `tox -e pypy`)
- 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:

```python
#!/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`.

via Setuptools

If you’re invoking Flake8 from your `setup.py` then you actually end up in `flake8.main.setuptools_command.Flake8.run()`. This then collects the files that are included in the package information and creates an instance of `Application`.

via Git or Mercurial

In both cases, they call their respective `hook` functions which create instances 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()`
Our Git hook, however, runs these individually.

**Application Initialization**

`initialize()` loads all of our plugins, 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 violations 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 sub-processes 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

class flake8.checker.FileChecker (filename, checks, options)
Manage running checks for a file and aggregate the results.

check_physical_eol (token)
Run physical checks if and only if it is at the end of the line.

handle_newline (token_type)
Handle the logic when encountering a newline token.

process_tokens ()
Process tokens and trigger checks.

This can raise a flake8.exceptions.InvalidSyntax exception. Instead of using this directly, you should use flake8.checker.FileChecker.run_checks().

report (error_code, line_number, column, text, line=None)
Report an error by storing it in the results list.

run_ast_checks ()
Run all checks expecting an abstract syntax tree.

run_check (plugin, **arguments)
Run the check in a single plugin.

run_checks ()
Run checks against the file.

run_logical_checks ()
Run all checks expecting a logical line.

run_physical_checks (physical_line, override_error_line=None)
Run all checks for a given physical line.

A single physical check may return multiple errors.

class flake8.checker.Manager (style_guide, arguments, checker_plugins)
Manage the parallelism and checker instances for each plugin and file.

This class will be responsible for the following:

• Determining the parallelism of Flake8, e.g.:
  – Do we use multiprocessing or is it unavailable?
  – Do we automatically decide on the number of jobs to use or did the user provide that?

• Falling back to a serial way of processing files if we run into an OSError related to multiprocessing

• Organizing the results of each checker so we can group the output together and make our output deterministic.

is_path_excluded (path)
Check if a path is excluded.
**Parameters** path *(str)* – Path to check against the exclude patterns.

**Returns** True if there are exclude patterns and the path matches, otherwise False.

**Return type** bool

**make_checkers** *(paths=None)*
Create checkers for each file.

**report** ()
Report all of the errors found in the managed file checkers.

This iterates over each of the checkers and reports the errors sorted by line number.

**Returns** A tuple of the total results found and the results reported.

**Return type** tuple(int, int)

**run** ()
Run all the checkers.

This will intelligently decide whether to run the checks in parallel or whether to run them in serial.

If running the checks in parallel causes a problem (e.g., https://gitlab.com/pycqa/flake8/issues/74) this also implements fallback to serial processing.

**run_parallel** ()
Run the checkers in parallel.

**run_serial** ()
Run the checkers in serial.

**start** *(paths=None)*
Start checking files.

**Parameters** paths *(list)* – Path names to check. This is passed directly to

**make_checkers()**.

**stop** ()
Stop checking files.

**class** flake8.processor.FileProcessor *(filename, options, lines=None)*
Processes a file and holds state.

This processes a file by generating tokens, logical and physical lines, and AST trees. This also provides a way of passing state about the file to checks expecting that state. Any public attribute on this object can be requested by a plugin. The known public attributes are:

- *blank_before*
- *blank_lines*
- *checker_state*
- *indent_char*
- *indent_level*
- *line_number*
- *logical_line*
- *max_line_length*
- *max_doc_length*
- *multiline*
- `noqa`
- `previous_indent_level`
- `previous_logical`
- `previous_unindented_logical_line`
- `tokens`
- `file_tokens`
- `total_lines`
- `verbose`

`blank_before = None`
Number of preceding blank lines

`blank_lines = None`
Number of blank lines

`build_ast()`
Build an abstract syntax tree from the list of lines.

`build_logical_line()`
Build a logical line from the current tokens list.

`build_logical_line_tokens()`
Build the mapping, comments, and logical line lists.

`check_physical_error(error_code, line)`
Update attributes based on error code and line.

`checker_state = None`
Current checker state

`delete_first_token()`
Delete the first token in the list of tokens.

`file_tokens`
Return the complete set of tokens for a file.
Accessing this attribute may raise an InvalidSyntax exception.

- **Raises** flake8.exceptions.InvalidSyntax

`generate_tokens()`
Tokenize the file and yield the tokens.

- **Raises** flake8.exceptions.InvalidSyntax – If a `tokenize.TokenError` is raised while generating tokens.

`hang_closing = None`
User provided option for hang closing

`indent_char = None`
Character used for indentation

`indent_level = None`
Current level of indentation

`inside_multiline(line_number)`
Context-manager to toggle the multiline attribute.

5.1. Exploring Flake8’s Internals
keyword_arguments_for(parameters, arguments=None)
    Generate the keyword arguments for a list of parameters.

line_for(line_number)
    Retrieve the physical line at the specified line number.

line_number = None
    Line number in the file

logical_line = None
    Current logical line

max_doc_length = None
    Maximum docstring / comment line length as configured by the user

max_line_length = None
    Maximum line length as configured by the user

multiline = None
    Whether the current physical line is multiline

next_line()
    Get the next line from the list.

next_logical_line()
    Record the previous logical line.
    This also resets the tokens list and the blank_lines count.

noqa = None
    Whether or not we’re observing NoQA

previous_indent_level = None
    Previous level of indentation

previous_logical = None
    Previous logical line

previous_unindented_logical_line = None
    Previous unindented (i.e. top-level) logical line

read_lines()
    Read the lines for this file checker.

read_lines_from_filename()
    Read the lines for a file.

read_lines_from_stdin()
    Read the lines from standard in.

reset_blank_before()
    Reset the blank_before attribute to zero.

should_ignore_file()
    Check if flake8: noqa is in the file to be ignored.
    Returns True if a line matches defaults.NOQA_FILE, otherwise False
    Return type bool

split_line(token)
    Split a physical line’s line based on new-lines.
    This also auto-increments the line number for the caller.
statistics = None
Statistics dictionary

strip_utf_bom()
Strip the UTF bom from the lines of the file.

tokens = None
Current set of tokens

total_lines = None
Total number of lines in the file

update_checker_state_for(plugin)
Update the checker_state attribute for the plugin.

update_state(mapping)
Update the indent level based on the logical line mapping.

verbose = None
Verbosity level of Flake8

visited_new_blank_line()
Note that we visited a new blank line.

Utility Functions

flake8.processor.count_parentheses(current_parentheses_count, token_text)
Count the number of parentheses.

flake8.processor.expand_indent(line)
Return the amount of indentation.

Tabs are expanded to the next multiple of 8.

```python
>>> expand_indent('  
4
>>> expand_indent('
\t')
8
>>> expand_indent('    
\t')
8
>>> expand_indent('      
\t')
16
```

flake8.processor.is_eol_token(token)
Check if the token is an end-of-line token.

flake8.processor.is_multiline_string(token)
Check if this is a multiline string.

flake8.processor.log_token(log, token)
Log a token to a provided logging object.

flake8.processor.mutate_string(text)
Replace contents with ‘xxx’ to prevent syntax matching.

```python
>>> mute_string("abc")
'"xxx"
>>> mute_string("'''abc'''")
"'''xxx'''"
```

(continues on next page)
flake8 Documentation, Release 3.7.4

>>> mute_string("r'abc'")
"r'xxx'"

flake8.processor.token_is_newline(token)
Check if the token type is a newline token type.

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

flake8.main.cli.main(argv=None)
Execute the main bit of the application.

Parameters

argv (list) – The arguments to be passed to the application for parsing.

class flake8.main.application.Application (program='flake8', version='3.7.4')
Abstract our application into a class.

args = None
The left over arguments that were not parsed by option_manager

catastrophic_failure = None
Whether or not something catastrophic happened and we should exit with a non-zero status code

check_plugins = None
The instance of flake8.plugins.manager.Checkers

config_finder = None
The instance of flake8.options.config.ConfigFileFinder

date_time = None
The timestamp when the Application finished reported errors.

exit()
Handle finalization and exiting the program.

This should be the last thing called on the application instance. It will check certain options and exit appropriately.

file_checker_manager = None
The flake8.checker.Manager that will handle running all of the checks selected by the user.

find_plugins()
Find and load the plugins for this application.

If check_plugins, or formatting_plugins are None then this method will update them with the appropriate plugin manager instance. Given the expense of finding plugins (via entrypoints) we want this to be idempotent and so only update those attributes if they are None.
formatter = None
    The user-selected formatter from formatting_plugins

formatter_for(self, formatter_plugin_name)
    Retrieve the formatter class by plugin name.

formatting_plugins = None
    The instance of flake8.plugins.manager.ReportFormatters

guide = None
    The flake8.style_guide.StyleGuideManager built from the user’s options

initialize(argv)
    Initialize the application to be run.
    This finds the plugins, registers their options, and parses the command-line arguments.

local_plugins = None
    The flake8.options.config.LocalPlugins found in config

make_config_finder()
    Make our ConfigFileFinder based on preliminary opts and args.

make_file_checker_manager()
    Initialize our FileChecker Manager.

make_formatter(formatter_class=None)
    Initialize a formatter based on the parsed options.

make_guide()
    Initialize our StyleGuide.

option_manager = None
    The instance of flake8.options.manager.OptionManager used to parse and handle the options
    and arguments passed by the user

options = None
    The user-supplied options parsed into an instance of optparse.Values

parse_configuration_and_cli(argv=None)
    Parse configuration files and the CLI options.

    Parameters argv (list) – Command-line arguments passed in directly.

parse_preliminary_options_and_args(argv=None)
    Get preliminary options and args from CLI, pre-plugin-loading.
    We need to know the values of a few standard options and args now, so that we can find config files and
    configure logging.
    Since plugins aren’t loaded yet, there may be some as-yet-unknown options; we ignore those for now,
    they’ll be parsed later when we do real option parsing.
    Sets self.prelim_opts and self.prelim_args.

    Parameters argv (list) – Command-line arguments passed in directly.

parsed_diff = None
    The parsed diff information

prelim_args = None
    The preliminary arguments parsed from CLI before plugins are loaded

prelim_opts = None
    The preliminary options parsed from CLI before plugins are loaded, into a optparse.Values instance
**program** = None
   The name of the program being run

**register_plugin_options**()
   Register options provided by plugins to our option manager.

**report**()
   Report errors, statistics, and benchmarks.

**report_benchmarks**()
   Aggregate, calculate, and report benchmarks for this run.

**report_errors**()
   Report all the errors found by flake8 3.0.
   This also updates the **result_count** attribute with the total number of errors, warnings, and other messages found.

**report_statistics**()
   Aggregate and report statistics from this run.

**result_count** = None
   The number of errors, warnings, and other messages after running flake8 and taking into account ignored errors and lines.

**run**(**argv**=None)
   Run our application.
   This method will also handle KeyboardInterrupt exceptions for the entirety of the flake8 application. If it sees a KeyboardInterrupt it will forcibly clean up the Manager.

**run_checks**(**files**=None)
   Run the actual checks with the FileChecker Manager.
   This method encapsulates the logic to make a Manager instance run the checks it is managing.

   **Parameters** **files** (**list**) – List of filenames to process

**running_against_diff** = None
   Whether the program is processing a diff or not

**start_time** = None
   The timestamp when the Application instance was instantiated.

**total_result_count** = None
   The total number of errors before accounting for ignored errors and lines.

**version** = None
   The version of the program being run

**flake8.main.options.register_default_options**(**option_manager**)  
Register the default options on our OptionManager.

   The default options include:
   • -v/--verbose
   • -q/--quiet
   • --count
   • --diff
   • --exclude
   • --filename
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.

```python
class flake8.formatting.default.Default(options)
    Default formatter for Flake8.
    This also handles backwards compatibility for people specifying a custom format string.
    after_init()
        Check for a custom format string.
```

**Pylint Formatter**

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

```python
class flake8.formatting.default.Pylint(options)
    Pylint formatter for Flake8.
```

## 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:

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

```plaintext
\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:

```python
```

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`:

```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:

```ini
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.

5.1. Exploring Flake8's Internals
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. As part of managing this ourselves, we decided to change management/discovery for 3.0.0. We have done the following:

- User files (files stored in a user’s home directory or in the XDG directory inside their home directory) are the first files read. For example, if the user has a ~/.flake8 file, we will read that first.

- 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’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 is managed by the ConfigFileFinder object. This object needs to know information about the program’s name, any extra arguments passed to it, and any configuration files that should be appended to the list of discovered files. It provides methods for finding the files and similar methods for parsing those files. For example, it provides local_config_files() to find known local config files (and append the extra configuration files) and it also provides local_configs() to parse those configuration files.

Note: local_config_files also filters out non-existent files.

Configuration file merging and management is controlled by the MergedConfigParser. This requires the instance of OptionManager that the program is using, the list of appended config files, and the list of extra arguments. This object is currently the sole user of the ConfigFileFinder object. It appropriately initializes the object and uses it in each of

- parse_cli_config()
- parse_local_config()
- parse_user_config()

Finally, merge_user_and_local_config() takes the user and local configuration files that are parsed by parse_local_config() and parse_user_config(). The main usage of the MergedConfigParser is 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 passed by the user necessary for creating an instance of MergedConfigParser.

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

flake8.options.aggregator.aggregate_options (manager, config_finder, arglist=None, values=None)

Aggregate and merge CLI and config file options.

Parameters

- **manager** (flake8.options.manager.OptionManager) – The instance of the OptionManager that we’re presently using.
- **config_finder** (flake8.options.config.ConfigFileFinder) – The config file finder to use.
- **arglist** (list) – The list of arguments to pass to manager.parse_args. In most cases this will be None so parse_args uses sys.argv. This is mostly available to make testing easier.
- **values** (optparse.Values) – Previously parsed set of parsed options.

Returns Tuple of the parsed options and extra arguments returned by manager.parse_args.

Return type tuple(optparse.Values, list)

class flake8.options.manager.Option (short_option_name=None, long_option_name=None, action=None, default=None, type=None, dest=None, nargs=None, const=None, choices=None, callback=None, callback_args=None, callback_kwargv=None, help=None, metavar=None, parse_from_config=False, comma_separated_list=False, normalize_paths=False)

Our wrapper around an optparse.Option object to add features.

__init__ (short_option_name=None, long_option_name=None, action=None, default=None, type=None, dest=None, nargs=None, const=None, choices=None, callback=None, callback_args=None, callback_kwargv=None, help=None, metavar=None, parse_from_config=False, comma_separated_list=False, normalize_paths=False)

Initialize an Option instance wrapping optparse.Option.

The following are all passed directly through to optparse.

Parameters

- **short_option_name** (str) – The short name of the option (e.g., -X). This will be the first argument passed to Option.
- **long_option_name** (str) – The long name of the option (e.g., --xtra-long-option). This will be the second argument passed to Option.
- **action** (str) – Any action allowed by optparse.
- **default** – Default value of the option.
- **type** – Any type allowed by optparse.
- **dest** – Attribute name to store parsed option value as.
- **nargs** – Number of arguments to parse for this option.
• **const** – Constant value to store on a common destination. Usually used in conjunction with action="store_const".

• **choices**(iterable) – Possible values for the option.

• **callback**(callable) – Callback used if the action is "callback".

• **callback_args**(iterable) – Additional positional arguments to the callback callable.

• **callback_kwargs**(dictionary) – Keyword arguments to the callback callable.

• **help**(str) – Help text displayed in the usage information.

• **metavar**(str) – Name to use instead of the long option name for help text.

The following parameters are for Flake8’s option handling alone.

**Parameters**

• **parse_from_config**(bool) – Whether or not this option should be parsed out of config files.

• **comma_separated_list**(bool) – Whether the option is a comma separated list when parsing from a config file.

• **normalize_paths**(bool) – Whether the option is expecting a path or list of paths and should attempt to normalize the paths to absolute paths.

**normalize**(value, *normalize_args)

Normalize the value based on the option configuration.

**to_optparse**( )

Convert a Flake8 Option to an optparse Option.

**class** flake8.options.manager.OptionManager**(prog=None, version=None, usage='%prog [options] file file ...')**

Manage Options and OptionParser while adding post-processing.

**__init__**(prog=None, version=None, usage='%prog [options] file file ...')

Initialize an instance of an OptionManager.

**Parameters**

• **prog**(str) – Name of the actual program (e.g., flake8).

• **version**(str) – Version string for the program.

• **usage**(str) – Basic usage string used by the OptionParser.

**__weakref__**

list of weak references to the object (if defined)

**add_option**( *args, **kwargs)

Create and register a new option.

See parameters for Option for acceptable arguments to this method.

---

**Note**: short_option_name and long_option_name may be specified positionally as they are with optparse normally.

**extend_default_ignore**(error_codes)

Extend the default ignore list with the error codes provided.
Parameters **error_codes** *(list)* – List of strings that are the error/warning codes with which to extend the default ignore list.

**extend_default_select** *(error_codes)*
Extend the default select list with the error codes provided.

Parameters **error_codes** *(list)* – List of strings that are the error/warning codes with which to extend the default select list.

**static format_plugin** *(plugin)*
Convert a PluginVersion into a dictionary mapping name to value.

**generate_epilog** *
Create an epilog with the version and name of each of plugin.

**generate_versions** *(format_str='%(name)s: %(version)s', join_on=',')*
Generate a comma-separated list of versions of plugins.

**parse_args** *(args=None, values=None)*
Proxy to calling the OptionParser’s parse_args method.

**parse_known_args** *(args=None, values=None)*
Parse only the known arguments from the argument values.
Replicate a little argparse behaviour while we’re still on optparse.

**register_plugin** *(name, version, local=False)*
Register a plugin relying on the OptionManager.

Parameters

* **name** *(str)* – The name of the checker itself. This will be the name attribute of the class or function loaded from the entry-point.
* **version** *(str)* – The version of the checker that we’re using.
* **local** *(bool)* – Whether the plugin is local to the project/repository or not.

**remove_from_default_ignore** *(error_codes)*
Remove specified error codes from the default ignore list.

Parameters **error_codes** *(list)* – List of strings that are the error/warning codes to attempt to remove from the extended default ignore list.

**update_version_string** *
Update the flake8 version string.

**class** `flake8.options.config.ConfigFileFinder` *(program_name, args, extra_config_files)*
Encapsulate the logic for finding and reading config files.

**__init__** *(program_name, args, extra_config_files)*
Initialize object to find config files.

Parameters

* **program_name** *(str)* – Name of the current program (e.g., flake8).
* **args** *(list)* – The extra arguments passed on the command-line.
* **extra_config_files** *(list)* – Extra configuration files specified by the user to read.

**__weakref__**
list of weak references to the object (if defined)

**cli_config** *(files)*
Read and parse the config file specified on the command-line.

5.1. Exploring Flake8’s Internals 75
**generate_possible_local_files()**  
Find and generate all local config files.

**local_config_files()**  
Find all local config files which actually exist.

Filter results from `generate_possible_local_files()` based on whether the filename exists or not.

**Returns** List of files that exist that are local project config files with extra config files appended to that list (which also exist).

**Return type** [str]

**local_configs()**  
Parse all local config files into one config object.

**local_configs_with_files()**  
Parse all local config files into one config object.

Return (config, found_config_files) tuple.

**user_config()**  
Parse the user config file into a config object.

**user_config_file()**  
Find the user-level config file.

---

**flake8.options.config.MergedConfigParser**(option_manager, config_finder)  
Encapsulate merging different types of configuration files.

This parses out the options registered that were specified in the configuration files, handles extra configuration files, and returns dictionaries with the parsed values.

**GETBOOL_ACTIONS** = {'store_false', 'store_true'}  
Set of actions that should use the `getbool()` method.

**GETINT_TYPES** = {'count', 'int'}  
Set of types that should use the `getint()` method.

**__init__**(option_manager, config_finder)  
Initialize the MergedConfigParser instance.

**Parameters**

- **option_manager**(flake8.options.manager.OptionManager) – Initialized OptionManager.


**__weakref__**  
list of weak references to the object (if defined)

**config_finder** = None  
Our instance of our ConfigFileFinder

**config_options** = None  
Mapping of configuration option names to Option instances

**is_configured_by**(config)  
Check if the specified config parser has an appropriate section.

**merge_user_and_local_config()**  
Merge the parsed user and local configuration files.
Returns Dictionary of the parsed and merged configuration options.

Return type dict

option_manager = None
Our instance of flake8.options.manager.OptionManager

parse(cli_config=None, isolated=False)
Parse and return the local and user config files.
First this copies over the parsed local configuration and then iterates over the options in the user configuration and sets them if they were not set by the local configuration file.

Parameters

• cli_config (str) – Value of –config when specified at the command-line. Overrides all other config files.

• isolated (bool) – Determines if we should parse configuration files at all or not. If running in isolated mode, we ignore all configuration files

Returns Dictionary of parsed configuration options

Return type dict

parse_cli_config(config_path)
Parse and return the file specified by –config.

parse_local_config()
Parse and return the local configuration files.

parse_user_config()
Parse and return the user configuration files.

program_name = None
The prog value for the cli parser

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

To facilitate this, Flake8 needed a more mature way of managing plugins. Thus, we developed the PluginManager which accepts a namespace and will load the plugins for that namespace. A PluginManager creates and manages many Plugin instances.

A Plugin lazily loads the underlying entry-point provided by setuptools. The entry-point will be loaded either by calling load_plugin() or accessing the plugin attribute. We also use this abstraction to retrieve options that the plugin wishes to register and parse.

The only public method the PluginManager provides is map(). This will accept a function (or other callable) and call it with each plugin as the first parameter.

We build atop the PluginManager with the PluginTypeManager. It is expected that users of the PluginTypeManager will subclass it and specify the namespace, e.g.,
This provides a few extra methods via the `PluginManager`’s `map` method.

Finally, we create two classes of plugins:

- **Checkers**
- **ReportFormatters**

These are used to interact with each of the types of plugins individually.

**Note:** Our inspiration for our plugin handling comes from the author’s extensive experience with `stevedore`.

### 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 `pep8`. Each check in `pep8` 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 `pep8` checks roughly like so:

```python
pep8.<check-name> = pep8:<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 report `pep8` and check `pep8` the module for a `__version__` attribute. We only report it once to avoid confusing users.

### API Documentation

```python
class flake8.plugins.manager.PluginManager(namespace, local_plugins=None)
Find and manage plugins consistently.
__init__(namespace, local_plugins=None)
Initialize the manager.
Parameters
• namespace (str) – Namespace of the plugins to manage, e.g., ‘flake8.extension’.
• local_plugins (list) – Plugins from config (as “X = path.to:Plugin” strings).
map(func, *args, **kwargs)
Call func with the plugin and *args and **kwargs after.
This yields the return value from func for each plugin.
Parameters
• func (collections.Callable) – Function to call with each plugin. Signature should at least be:
  def myfunc(plugin):
    pass
```

```
Any extra positional or keyword arguments specified with map will be passed along to this function after the plugin. The plugin passed is a `Plugin`.

- **args** – Positional arguments to pass to `func` after each plugin.
- **kwargs** – Keyword arguments to pass to `func` after each plugin.

```python
versions()

    Generate the versions of plugins.

    Returns  Tuples of the plugin_name and version

    Return type  tuple
```

```python
class flake8.plugins.manager.Plugin(name, entry_point, local=False)

Wrap an EntryPoint from setuptools and other logic.

    __init__(name, entry_point, local=False)

    Initialize our Plugin.

    Parameters

    - **name** *(str)* – Name of the entry-point as it was registered with setuptools.
    - **entry_point** *(setuptools.EntryPoint)* – EntryPoint returned by setuptools.
    - **local** *(bool)* – Is this a repo-local plugin?

    disable(optmanager)

Add the plugin name to the default ignore list.

```python
def disable(optmanager)
    Add the plugin name to the default ignore list.
```

```python
def enable(optmanager, options=None)

    Remove plugin name from the default ignore list.

    execute(*args, **kwargs)

    Call the plugin with *args and **kwargs.
```

```python
def execute(*args, **kwargs)

    Call the plugin with *args and **kwargs.
```

```python
def group()

    Find and parse the group the plugin is in.

    ```python
def group()
        Find and parse the group the plugin is in.
```

```python
def is_in_a_group()

    Determine if this plugin is in a group.

    Returns  True if the plugin is in a group, otherwise False.

    Return type  bool
```

```python
def load_plugin()

    Retrieve the plugin for this entry-point.

    This loads the plugin, stores it on the instance and then returns it. It does not reload it after the first time, it merely returns the cached plugin.

    Returns  Nothing
```

```python
def off_by_default

    Return whether the plugin is ignored by default.
```

```python
def parameter_names

    List of argument names that need to be passed to the plugin.
```

```python
def parameters

    List of arguments that need to be passed to the plugin.
```

5.1. Exploring Flake8's Internals
plugin
Load and return the plugin associated with the entry-point.
This property implicitly loads the plugin and then caches it.

plugin_name
Return the name of the plugin.

provide_options (optmanager, options, extra_args)
Pass the parsed options and extra arguments to the plugin.

register_options (optmanager)
Register the plugin’s command-line options on the OptionManager.

Parameters
optmanager (flake8.options.manager.OptionManager) – Instantiated OptionManager to register options on.

Returns
Nothing

to_dictionary ()
Convert this plugin to a dictionary.

version
Return the version of the plugin.

class flake8.plugins.manager.PluginTypeManager (local_plugins=None)
Parent class for most of the specific plugin types.

class flake8.plugins.manager.Checkers (local_plugins=None)
All of the checkers registered through entry-points or config.

ast_plugins
List of plugins that expect the AST tree.
checks_expecting (argument_name)
Retrieve checks that expect an argument with the specified name.
Find all checker plugins that are expecting a specific argument.

logical_line_plugins
List of plugins that expect the logical lines.

physical_line_plugins
List of plugins that expect the physical lines.

register_options (optmanager)
Register all of the checkers’ options to the OptionManager.
This also ensures that plugins that are not part of a group and are enabled by default are enabled on the
option manager.

to_dictionary()
Return a dictionary of AST and line-based plugins.

class flake8.plugins.manager.ReportFormatters (local_plugins=None)
All of the report formatters registered through entry-points/config.

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 considera-
tion they may be added to the documented stable API.

flake8.utils.parse_comma_separated_list (value, regexp=re.compile(r'[,\s]'))
Parse a comma-separated list.

Parameters

- value – String or list of strings to be parsed and normalized.
- regexp (_sre.SRE_Pattern) – Compiled regular expression used to split the value
  when it is a string.

Returns List of values with whitespace stripped.

Return type list

parse_comma_separated_list () takes either a string like

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

Or it will take a list of strings (potentially with whitespace) such as

[" E121\n", "\tW123 ", "\nF904\n "]

And converts it to a list that looks as follows

5.1. Exploring Flake8's Internals
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.

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

Normalize a single-path.

- **Returns**: The normalized path.
- **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`.

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

Parse a comma-separated list of paths.

- **Returns**: The normalized paths.
- **Return type**: [str]

This function utilizes `parse_comma_separated_list()` and `normalize_path()` to normalize its input to a list of strings that should be paths.

```python
flake8.utils.stdin_get_value()
```

Get and cache it so plugins can use it.

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

```python
flake8.utils.is_windows()
```

Determine if we’re running on Windows.

- **Returns**: True if running on Windows, otherwise False
- **Return type**: bool

This provides a convenient and explicitly named function that checks if we are currently running on a Windows (or nt) operating system.

```python
flake8.utils.can_run_multiprocessing_on_windows()
```

Determine if we can use multiprocessing on Windows.

This presently will always return False due to a bug in the multiprocessing module on Windows. Once fixed, we will check to ensure that the version of Python contains that fix (via version inspection) and conditionally re-enable support on Windows.

- **Returns**: True if the version of Python is modern enough, otherwise False
- **Return type**: bool

This provides a separate and distinct check from `is_windows()` that allows us to check if the version of Python we’re using can actually use multiprocessing on Windows.

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

Determine if we’re going to read from stdin.

- **Parameters**: `paths` (list) – 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.

```python
flake8.utils.filenames_from(arg, predicate=None)
```

Generate filenames from an argument.

**Parameters**

- `arg` *(str)* – Parameter from the command-line.
- `predicate` *(callable)* – Predicate to use to filter out filenames. If the predicate returns
  `True` we will exclude the filename, otherwise we will yield it. By default, we include every
  filename generated.

**Returns** Generator of paths

When provided an argument to Flake8, we need to be able to traverse directories in a convenient manner. For
example, if someone runs

```
$ flake8 flake8/
```

Then they want us to check all of the files in the directory `flake8/`. This function will handle that while also handling
the case where they specify a file like:

```
$ flake8 flake8/__init__.py
```

```python
flake8.utils.fnmatch(filename, patterns, default=True)
```

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

**Parameters**

- `filename` *(str)* – Name of the file we’re trying to match.
- `patterns` *(list)* – Patterns we’re using to try to match the filename.
- `default` *(bool)* – The default value if patterns is empty

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

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.

```python
flake8.utils.parameters_for(plugin)
```

Return the parameters for the plugin.

This will inspect the plugin and return either the function parameters if the plugin is a function or the parameters
for `__init__` after `self` if the plugin is a class.

**Parameters**

- `plugin` *(flake8.plugins.manager.Plugin)* – The internal plugin object.

**Returns** A dictionary mapping the parameter name to whether or not it is required (a.k.a., is posi-
tional only/does not have a default).

**Return type** `dict([(str, bool)])`

Flake8 analyzes the parameters to plugins to determine what input they are expecting. Plugins may expect one of
the following:

- `physical_line` to receive the line as it appears in the file
- `logical_line` to receive the logical line (not as it appears in the file)
- `tree` to receive the abstract syntax tree (AST) for the file
We also analyze the rest of the parameters to provide more detail to the plugin. This function will return the parameters in a consistent way across versions of Python and will handle both classes and functions that are used as plugins. Further, if the plugin is a class, it will strip the `self` argument so we can check the parameters of the plugin consistently.

```python
flake8.utils.parse_unified_diff(diff=None)
```

Parse the unified diff passed on stdin.

Returns dictionary mapping file names to sets of line numbers

Return type `dict`

To handle usage of `flake8 --diff`, Flask8 needs to be able to parse the name of the files in the diff as well as the ranges indicated the sections that have been changed. This function either accepts the diff as an argument or reads the diff from standard-in. It then returns a dictionary with filenames as the keys and sets of line numbers as the value.
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 3.x Release Series

3.7.4 – 2019-01-31

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

Bugs Fixed

• Fix performance regression with lots of per-file-ignores and errors (See also GitLab!299, GitLab#501)

3.7.3 – 2019-01-30

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

Bugs Fixed

• Fix imports of typing in python 3.5.0 / 3.5.1 (See also GitLab!294, GitLab#498)
• Fix flake8 --statistics (See also GitLab!295, GitLab#499)
• Gracefully ignore flake8-per-file-ignores plugin if installed (See also GitLab!297, GitLab#495)
• Improve error message for malformed per-file-ignores (See also GitLab!298, GitLab#489)
3.7.2 – 2019-01-30

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

Bugs Fixed

- Fix broken `flake8 --diff` (regressed in 3.7.0) (See also GitLab!292, GitLab#490)
- Fix typo in plugin exception reporting (See also GitLab!275, GitLab#491)
- Fix `AttributeError` while attempting to use the legacy api (regressed in 3.7.0) (See also GitLab!293, GitLab#497)

3.7.1 – 2019-01-30

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

Bugs Fixed

- Fix capitalized filenames in `per-file-ignores` setting (See also GitLab!290, GitLab#488)

3.7.0 – 2019-01-29

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

New Dependency Information

- Add dependency on `entrypoints` >= 0.3, < 0.4 (See also GitLab!264, GitLab!288)
- Pyflakes has been updated to >= 2.1.0, < 2.2.0 (See also GitLab!283, GitLab!285)
- pycodestyle has been updated to >= 2.5.0, < 2.6.0 (See also GitLab!287)

Features

- Add support for `per-file-ignores` (See also GitLab!259, GitLab!156, GitLab!281, GitLab#471)
- Enable use of `float` and `complex` option types (See also GitLab!261, GitLab#452)
- Improve startup performance by switching from `pkg_resources` to `entrypoints` (See also GitLab!264)
- Add metadata for use through the `pre-commit` git hooks framework (See also GitLab!268, GitLab!284)
- Allow physical line checks to return more than one result (See also GitLab!269)
- Allow `# noqa:X123` comments without space between the colon and codes list (See also GitLab!273, GitLab#470)
- Remove broken and unused `flake8.listen` plugin type (See also GitLab!274, GitLab#480)

3.6.0 – 2018-10-23

You can view the 3.6.0 milestone on GitLab for more details.
New Dependency Information

- `pycodestyle` has been updated to \( \geq 2.4.0, < 2.5.0 \) (See also GitLab#381, GitLab#415, GitLab!212, GitLab!230, GitLab!231)
- `Pyflakes` has been updated to \( \geq 2.0.0, < 2.1.0 \) (See also GitLab#422, GitLab!239)
- `flake8` requires python 2.x \( \geq 2.7 \) or python 3.x \( \geq 3.4 \) (See also GitLab!225)

Features

- Add `paths` to allow local plugins to exist outside of `sys.path` (See also GitLab#379, GitLab!211)
- Copy `setup.cfg` files to the temporary git hook execution directory (See also GitLab!215)
- Only skip a file if `# flake8: noqa` is on a line by itself (See also GitLab#453, GitLab!219)
- Provide a better user experience for broken plugins (See also GitLab!221)
- Report E902 when a file passed on the command line does not exist (See also GitLab#405, GitLab!227)
- Add `--extend-ignore` for extending the default `ignore` instead of overriding it (See also GitLab#365, GitLab!233)

Bugs Fixed

- Respect a formatter’s newline setting when printing (See also GitLab!222)
- Fix leaking of processes in the legacy api (See also GitLab#410, GitLab!228)
- Fix a SyntaxWarning for an invalid escape sequence (See also GitLab!244)
- Fix DeprecationWarning due to import of abc classes from the collections module (See also GitLab#249)
- Defer setuptools import to improve flake8 startup time (See also GitLab!250)
- Fix inconsistent line endings in `FileProcessor.lines` when running under python 3.x (See also GitLab#457, GitLab!255)

3.5.0 – 2017-10-23

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

New Dependency Information

- Allow for PyFlakes 1.6.0 (See also GitLab#359)
- Start using new PyCodeestyle checks for bare excepts and ambiguous identifier (See also GitLab#361)

Features

- Print out information about configuring VCS hooks (See also GitLab#335)
- Allow users to develop plugins “local” to a repository without using setuptools. See our documentation on local plugins for more information. (See also GitLab#357)
Bugs Fixed

- Catch and helpfully report `UnicodeDecodeError` when parsing configuration files. (See also GitLab#358)

3.4.1 – 2017-07-28

You can view the 3.4.1 milestone on GitLab for more details.
- Fix minor regression when users specify only a `--select` list with items in the enabled/extended select list. (See also GitLab#354)

3.4.0 – 2017-07-27

You can view the 3.4.0 milestone on GitLab for more details.
- Refine logic around `--select` and `--ignore` when combined with the default values for each. (See also GitLab#318)
- Handle spaces as an alternate separate for error codes, e.g., `--ignore 'E123 E234'`. (See also GitLab#329)
- Filter out empty select and ignore codes, e.g., `--ignore E123,,E234`. (See also GitLab#330)
- Specify dependencies appropriately in setup.py (See also Gitlab#341)
- Fix bug in parsing `--quiet` and `--verbose` from config files. (See also GitLab#193)
- Remove unused import of os in the git hook template (See also GitLab#194)

3.3.0 – 2017-02-06

You can view the 3.3.0 milestone on GitLab 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 GitLab#156)
- Display the local file path instead of the temporary file path when using the git hook. (See also GitLab#244)
- Add methods to Report class that will be called when Flake8 starts and finishes processing a file. (See also GitLab#251)
- Fix problem where hooks should only check *.py files. (See also GitLab#268)
- Fix handling of SyntaxErrors that do not include physical line information. (See also GitLab#279)
- Update upper bound on PyFlakes to allow for PyFlakes 1.5.0. (See also GitLab#290)
- Update setuptools integration to less eagerly deduplicate packages. (See also GitLab#295)
- Force `flake8 --version` to be repeatable between invocations. (See also GitLab#297)
3.2.1 – 2016-11-21

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

- Fix subtle bug when deciding whether to report an on-by-default’s violation (See also GitLab#257)
- Fix another bug around SyntaxErrors not being reported at the right column and row (See also GitLab#259 and GitLab#237 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 GitLab#266)

3.2.0 – 2016-11-14

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

- Allow for pycodestyle 2.2.0 which fixes a bug in E305 (See also GitLab#256)

3.1.1 – 2016-11-14

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

- Do not attempt to install/distribute a man file with the Python package; leave this for others to do. (See also GitLab#254)
- Fix packaging bug where wheel version constraints specified in setup.cfg did not match the constraints in setup.py. (See also GitLab#255)

3.1.0 – 2016-11-14

You can view the 3.1.0 milestone on GitLab 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 GitLab#210, GitLab#218, GitLab#223)
- Avoid unhandled exceptions when dealing with SyntaxErrors. (See also GitLab#214, GitLab#238)
- Exit early if the value for --diff is empty. (See also GitLab#226)
- Handle empty --stdin-display-name values. (See also GitLab#235)
- 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 GitLab#237)
- 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 GitLab#239)
- Fix problems around --select and --ignore behaviour that prevented codes that were neither explicitly selected nor explicitly ignored from being reported. (See also GitLab#242)
- 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 GitLab#245)
- Add new File Processor attribute, previous_unindented_logical_line to accommodate pycodestyle 2.1.0. (See also GitLab#246)
• When something goes wrong, exit non-zero. (See also GitLab#248, GitLab#209)
• 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 GitLab#164)
• Fix an Attribute Error raised when dealing with Invalid Syntax. (See also GitLab#203)
• Fix an unhandled Syntax Error when tokenizing files. (See also GitLab#205)

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 GitLab#164)
• Fix # noqa comments followed by a : and explanation broken by 3.0.0 (See also GitLab#178)
• Always open our output file in append mode so we do not overwrite log messages. (See also GitLab#193)
• When normalizing path values read from configuration, keep in context the directory where the configuration was found so that relative paths work. (See also GitLab#194)
• Fix issue where users were unable to ignore plugin errors that were on by default. (See also GitLab#195)
• Fix our legacy API StyleGuide’s init_report method to actually override the previous formatter. (See also GitLab#200)

3.0.2 – 2016-07-26

• Fix local config file discovery. (See also GitLab#181)
• 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 GitLab#186)

3.0.1 – 2016-07-25

• Fix regression in handling of # noqa for multiline strings. (See also GitLab#177)
• Fix regression in handling of --output-file when not also using --verbose. (See also GitLab#180)
• Fix regression in handling of --quiet. (See also GitLab#180)
• Fix regression in handling of --statistics. (See also GitLab#180)
3.0.0 – 2016-07-25

- Rewrite our documentation from scratch! ([http://flake8.pycqa.org](http://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,E,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.

6.1.2 2.x Release Series

2.6.2 - 2016-06-25

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

---

6.1. Release Notes and History
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 ([GitLab#103](#))
- **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. ([GitLab#105](#), [GitLab#107](#))

2.5.0 - 2015-10-26

- **Improvement** Raise cap on PyFlakes for Python 3.5 support
- **Improvement** Avoid deprecation warnings when loading extensions ([GitLab#59](#), [GitLab#90](#))
- **Improvement** Separate logic to enable “off-by-default” extensions ([GitLab#67](#))
- **Bug** Properly parse options to setuptools Flake8 command ([GitLab#41](#))
- **Bug** Fix exceptions when output on stdout is truncated before Flake8 finishes writing the output ([GitLab#69](#))
• **Bug** Fix error on OS X where Flake8 can no longer acquire or create new semaphores (GitLab#74)

2.4.1 - 2015-05-18

• **Bug** Do not raise a `SystemError` unless there were errors in the setuptools command. (GitLab#39, GitLab#23)
• **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. (GitLab#31)
• **Bug** Put upper cap on dependencies. The caps for 2.4.0 are:
  - pep8 < 1.6 (Related to GitLab#35)
  - mccabe < 0.4
  - pyflakes < 0.9
See also GitLab#32
• **Bug** Files excluded in a config file were not being excluded when flake8 was run from a git hook. (GitHub#2)
• **Improvement** Print warnings for users who are providing mutually exclusive options to flake8. (GitLab#8, GitLab#18)
• **Feature** Allow git hook configuration to live in `.git/config`. See the updated VCS hooks docs for more details. (GitLab#20)

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 (GitLab#17)

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 multiprocesssing 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

6.1.3 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.

6.1.4 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 7

General Indices

- genindex
- Index of Documented Public Modules
- *Glossary of terms*
flake8.api.legacy, 33
<table>
<thead>
<tr>
<th>Method/Attribute</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong>()</td>
<td>78</td>
</tr>
<tr>
<td><strong>weakref</strong></td>
<td>75</td>
</tr>
<tr>
<td><strong>weakref</strong></td>
<td>76</td>
</tr>
<tr>
<td><strong>weakref</strong></td>
<td>74</td>
</tr>
<tr>
<td>add_option()</td>
<td>74</td>
</tr>
<tr>
<td>after_init()</td>
<td>43</td>
</tr>
<tr>
<td>after_init()</td>
<td>70</td>
</tr>
<tr>
<td>aggregate_options()</td>
<td>73</td>
</tr>
<tr>
<td>Application</td>
<td>66</td>
</tr>
<tr>
<td>args</td>
<td>66</td>
</tr>
<tr>
<td>ast_plugins</td>
<td>74</td>
</tr>
<tr>
<td>BaseFormatter</td>
<td>43</td>
</tr>
<tr>
<td>beginning()</td>
<td>43</td>
</tr>
<tr>
<td>blank_before</td>
<td>63</td>
</tr>
<tr>
<td>blank_lines</td>
<td>63</td>
</tr>
<tr>
<td>build_ast()</td>
<td>63</td>
</tr>
<tr>
<td>build_logical_line()</td>
<td>63</td>
</tr>
<tr>
<td>build_logical_line_tokens()</td>
<td>63</td>
</tr>
<tr>
<td>C</td>
<td>73</td>
</tr>
<tr>
<td>can_run_multiprocessing_on_windows()</td>
<td>82</td>
</tr>
<tr>
<td>catastrophic_failure</td>
<td>66</td>
</tr>
<tr>
<td>check_files()</td>
<td>32</td>
</tr>
<tr>
<td>check_physical_eol()</td>
<td>61</td>
</tr>
<tr>
<td>check_physical_error()</td>
<td>61</td>
</tr>
<tr>
<td>check_plugins</td>
<td>66</td>
</tr>
<tr>
<td>checker_state</td>
<td>66</td>
</tr>
<tr>
<td>Checks</td>
<td>80</td>
</tr>
<tr>
<td>checks_expecting()</td>
<td>80</td>
</tr>
<tr>
<td>cli_config()</td>
<td>75</td>
</tr>
<tr>
<td>config_finder</td>
<td>66</td>
</tr>
<tr>
<td>config_options</td>
<td>76</td>
</tr>
<tr>
<td>ConfigFileFinder</td>
<td>75</td>
</tr>
<tr>
<td>count_parentheses()</td>
<td>65</td>
</tr>
<tr>
<td>D</td>
<td>70</td>
</tr>
<tr>
<td>Default</td>
<td>70</td>
</tr>
<tr>
<td>delete_first_token()</td>
<td>63</td>
</tr>
<tr>
<td>disable()</td>
<td>79</td>
</tr>
<tr>
<td>enable()</td>
<td>79</td>
</tr>
<tr>
<td>end_time</td>
<td>66</td>
</tr>
<tr>
<td>error</td>
<td>4</td>
</tr>
<tr>
<td>error class</td>
<td>4</td>
</tr>
<tr>
<td>error code</td>
<td>4</td>
</tr>
<tr>
<td>excluded()</td>
<td>32</td>
</tr>
<tr>
<td>execute()</td>
<td>79</td>
</tr>
<tr>
<td>exit()</td>
<td>66</td>
</tr>
<tr>
<td>extend_default_ignore()</td>
<td>74</td>
</tr>
<tr>
<td>extend_default_select()</td>
<td>75</td>
</tr>
<tr>
<td>F</td>
<td>83</td>
</tr>
<tr>
<td>file_checker_manager</td>
<td>66</td>
</tr>
<tr>
<td>file_tokens</td>
<td>63</td>
</tr>
<tr>
<td>FileChecker</td>
<td>61</td>
</tr>
<tr>
<td>filename</td>
<td>43</td>
</tr>
<tr>
<td>filenames_from()</td>
<td>83</td>
</tr>
<tr>
<td>FileProcessor</td>
<td>62</td>
</tr>
<tr>
<td>find_plugins</td>
<td>66</td>
</tr>
<tr>
<td>finished()</td>
<td>43</td>
</tr>
<tr>
<td>flake8 command line option</td>
<td>43</td>
</tr>
<tr>
<td>--append-config=&lt;config&gt;,</td>
<td>21</td>
</tr>
<tr>
<td>--benchmark,</td>
<td>23</td>
</tr>
<tr>
<td>--bug-report,</td>
<td>23</td>
</tr>
<tr>
<td>--builtins=&lt;builtins&gt;,</td>
<td>21</td>
</tr>
</tbody>
</table>
stdin_get_value() (in module flake8Utils), 82
stop() (flake8.checker.Manager method), 62
stop() (flake8.formatting.base.BaseFormatter method), 44
strip_utf_bom() (flake8.processor.FileProcessor method), 65
StyleGuide (class in flake8.api.legacy), 33

to_dictionary() (flake8.plugins.manager.Checkers method), 81
to_dictionary() (flake8.plugins.manager.Plugin method), 80
to_optparse() (flake8.options.manager.Option method), 74
token_is_newline() (in module flake8.processor), 66
tokens (flake8.processor.FileProcessor attribute), 65
total_errors (flake8.api.legacy.Report attribute), 33
total_lines (flake8.processor.FileProcessor attribute), 65
total_result_count (flake8.main.application.Application attribute), 68

update_checker_state_for() (flake8.processor.FileProcessor method), 65
update_state() (flake8.processor.FileProcessor method), 65
update_version_string() (flake8.options.manager.OptionManager method), 75
user_config() (flake8.options.config.ConfigFileFinder method), 76
user_config_file() (flake8.options.config.ConfigFileFinder method), 76

verbose (flake8.processor.FileProcessor attribute), 65
version (flake8.main.application.Application attribute), 68
version (flake8.plugins.manager.Plugin attribute), 80
versions() (flake8.plugins.manager.PluginManager method), 79
violation, 4
visited_new_blank_line() (flake8.processor.FileProcessor method), 65

warning, 4
write() (flake8.formatting.base.BaseFormatter method), 44