Fixit Documentation

Jimmy Lai, Josie Eshkenazi, Tim Hatch, John Reese, Benjamin W

Apr 01, 2023

TUTORIAL:

A lint framework that writes better Python code for you.

Fixit is a lint framework that complements Flake8. It's based on LibCST which makes it possible to provide **auto-fixes**. Lint rules are made easy to build through pattern matching, a test toolkit, and utility helpers (e.g. scope analysis) for non-trivial boilerplate. It is optimized for efficiency, easy to customize and comes with many builtin lint rules.

CHAPTER

ONE

GETTING STARTED

Use `pip install fixit` to install Fixit.

1.1 Analyze Code Issues and Autofix Issues

Given an example code (`example.py`) like this:

[2]:
```
%%writefile $file_path
from typing import Dict


class C(object):
    attr = "ab" "cd" "ef" "gh"

    def method(self) -> Dict[int, str]:
        filtered_char = []
        for char in self.attr:
            if char is not "a":
                filtered_char.append(char)

        index_to_char = dict([(idx, char) for idx, char in enumerate(filtered_char)])
        return index_to_char
```

```
Writing /tmp/tmp194jsdpw/example.py
```

We can run Fixit rules to check code issues:

[4]:
```
! python -m fixit.cli.run_rules
```

```
Scanning 1 files
Testing 33 rules

example.py:4:1
    NoInheritFromObjectRule: Inheriting from object is a no-op.  'class Foo:' is
    just fine =)
example.py:5:12
    UsePlusForStringConcatRule: Implicit string concatenation detected, please
    add '+' to be explicit. E.g. a tuple or a call ("a" "b") with a missing
    comma results in multiple strings being concatenated as one string and
    causes unexpected behaviour.
example.py:10:16
    ComparePrimitivesByEqualRule: Don't use `is` or `is not` to compare
    primitives, as they compare references. Use == or != instead.
example.py:13:25
```

(continues on next page)

(continued from previous page)

```
    RewriteToComprehensionRule: It's unnecessary to use a list comprehension
    inside a call to dict since there are equivalent comprehensions for this
    type

Found 4 reports in 1 files in 0.58 seconds.
```

Each warning shows the violation's position in the format of `file_name:starting_line_number:starting_column_number`, follwed by the lint rule name and lint message.

To fix the issues automatically:

[5]:
```
! python -m fixit.cli.apply_fix
```

```
Scanning 1 files
./example.py

example.py:4:1 [applied fix]
    NoInheritFromObjectRule: Inheriting from object is a no-op.  'class Foo:' is
    just fine =)
example.py:5:12 [applied fix]
    UsePlusForStringConcatRule: Implicit string concatenation detected, please
    add '+' to be explicit. E.g. a tuple or a call ("a" "b") with a missing
    comma results in multiple strings being concatenated as one string and
    causes unexpected behaviour.
example.py:10:16 [applied fix]
    ComparePrimitivesByEqualRule: Don't use `is` or `is not` to compare
    primitives, as they compare references. Use == or != instead.
example.py:13:25 [applied fix]
    RewriteToComprehensionRule: It's unnecessary to use a list comprehension
    inside a call to dict since there are equivalent comprehensions for this
    type
reformatted -

All done!
1 file reformatted.

Found 4 reports in 1 files in 3.16 seconds.
```

All the issues are automatically fixed!

[6]:
```
! git diff
```

```
diff --git a/example.py b/example.py
index aed4bb5..3f667f8 100644
--- a/example.py
+++ b/example.py
@@ -1,14 +1,14 @@
 from typing import Dict


-class C(object):
-    attr = "ab" "cd" "ef" "gh"
+class C:
+    attr = "ab" + "cd" + "ef" + "gh"

     def method(self) -> Dict[int, str]:
         filtered_char = []
         for char in self.attr:
```

(continues on next page)

(continued from previous page)

```
-            if char is not "a":
+            if char != "a":
                 filtered_char.append(char)

-        index_to_char = dict([(idx, char) for idx, char in enumerate(filtered_char)])
+        index_to_char = {idx: char for idx, char in enumerate(filtered_char)}
         return index_to_char
```

1.2 Configuration File

A Fixit configuration file allows you to configure Fixit settings for your codebase.

1. To initialize a configuration file populated with some defaults, run:

   ```
   python -m fixit.cli.init_config
   ```

   This will create a `.fixit.config.yaml` with default settings in the current working directory.

2. Next, you may wish to edit or add some specific settings. The available configurations are:

- `allow_list_rules`: A list of rules (whether custom of from Fixit) that should be applied to the repository. Omitting this setting allows all rules to run. For example:

  allow_list_rules: [Flake8PseudoLintRule]

`block_list_rules` takes precendence, so if a rule is in both `allow_list_rules` and `block_list_rules` the rule will not be run.

- `block_list_patterns`: A list of patterns that indicate that a file should not be linted. For example:

  ```
  block_list_patterns: ['@generated', '@nolint']
  ```

  will tell Fixit to skip linting any files that have `@generated` or `@nolint` in their contents.

- `block_list_rules`: A list of rules (whether custom or from Fixit) that should not be applied to the repository. For example:

  ```
  block_list_rules: [NoInheritFromObjectRule]
  ```

- `fixture_dir`: The directory in which fixture files required for unit testing are to be found. This is only necessary if you are testing rules that use a metadata cache (see *AwaitAsyncCallRule* for an example of such a rule). This can be an absolute path, or a path relative to *repo_root* (see below).
- `use_noqa`: Defaults to `False`. Use `True` to support Flake8 lint suppression comment: noqa. The noqa is not recommended because a bare noqa implicitly silences all lint errors which prevent other useful lint errors to show up. We recommend use `lint-fixme` or `lint-ignore` suppression comments.
- `formatter`: A list of the formatter commands to use after a lint is complete. These will be passed to the `args` parameter in subprocess.check_output in the order in which they appear. For example:

  ```
  formatter: [black, '-']
  ```

  Here, the formatter of choice would be Black and the added `-` tells it to read from standard input, and write to standard output so that it is compatible with Fixit's formatting logic.

- `packages`: The Python packages in which to search for lint rules. For example:

```
packages: [fixit.rules, my.custom.package]
```

- `repo_root`: The path to the repository root. This can be a path relative to the *.fixit.config.yaml* file or an absolute path. For example:

```
repo_root: .
```

- `rule_config`: Rule-specific configurations. For example:

```
ImportConstraintsRule:
    fixit:
        rules: [["*", "allow"]]
```

(see *ImportConstraintsRule* for more details on this example)

3. A *.fixit.config.yaml* example with populated settings:

```
block_list_patterns:
- '@generated'
- '@nolint'
block_list_rules:
- BlockListedRule
fixture_dir: ./tests/fixtures
formatter:
- black
- '-'
packages:
- fixit.rules
repo_root: .
rule_config:
    ImportConstraintsRule:
        fixit:
            rules: [["*", "allow"]]
```

1.3 Enforcing Custom Rules

After finishing up the configuration, you may wish to enforce some custom lint rules in your repository.

1. Start by creating a directory where your custom rules will live. Make sure to include an `__init__.py` file so that the directory is importable as a package. This can simply be an empty file. For example:

```
my_repo_root
    └── lint
        └── custom_rules
            └── __init__.py
```

2. Include the dotted name of the package in the *.fixit.config.yaml* file under the *packages* setting:

```
packages:
- fixit.rules
- lint.custom_rules
```

3. See the *Build a Lint Rule* page for more details on how to write the logic for a custom lint rule.

1.4 Running Lint Rules

You may also want to run some rules against your repository to see all current violations.

- To run only the pre-packaged Fixit rules against the entire repository, run:

```
python -m fixit.cli.run_rules --rules fixit.rules
```

- To run only your custom rules package against the entire repository, run:

```
python -m fixit.cli.run_rules --rules <dotted_name_of_custom_package>
```

- To run a specific rule against the entire repository, run:

```
python -m fixit.cli.run_rules --rules <rule_name>
```

- To run all the rule packages under the `packages` settings in the *.fixit.config.yaml* file against the entire repository, run:

```
python -m fixit.cli.run_rules
```

- To run all the rule packages under the `packages` settings in the *.fixit.config.yaml* file against a particular file or directory, run:

```
python -m fixit.cli.run_rules <file_or_directory>
```

- To run all the rule packages under the `packages` settings in the *.fixit.config.yaml* file against mutliple files or directories, run:

```
python -m fixit.cli.run_rules <file_or_directory> <file_or_directory2> <file_or_
↪directory3>
```

1.5 Applying Autofixes

Some rules come with provided autofix suggestions. We have provided a script to help you automatically apply these suggested fixes. To do this, run:

```
python -m fixit.cli.apply_fix <file_or_directory> --rules <rule_name_or_package>
```

This will apply one or more lint rules' autofix to the source code in the specified file(s) or directory.

- For more detailes on this script's usage, run:

```
python -m fixit.cli.apply_fix --help
```

1.6 Suppressing Violations

You may wish to suppress existing lint violations from the lint engine altogether. We have provided a script to help you automatically insert lint suppressions. To do this, run:

```
python -m fixit.cli.insert_suppressions <rule_name> <file_or_directory>
```

This will insert a suppression in the form of a `# lint-fixme` comment above lines in the source code that violate the specified rule.

- For more detailes on this script's usage, run:

```
python -m fixit.cli.insert_suppressions --help
```

CHAPTER

TWO

BUILD A LINT RULE

2.1 Example: NoInheritFromObjectRule

In Python 3, a class is inherited from `object` by default. Explicitly inheriting from `object` is redundant, so removing it keeps the code simpler. In this tutorial, we'd like to build a lint rule to identify cases when a class inherit from object and add an autofix to remove it.

[2]:
```
# an example with unnecessary object inheritance
class C(object):
    ...


# the above example can be simplified as this
class C:
    ...
```

2.2 Pick a Good Name

Before starting creating a new lint rule, let's figure out a good short name for it. **A good lint rule name should be short and actionable.** Instead of describing the issue, describe the action needs to be taken to fix it. So developers can easily learn how to fix the issue by just reading the name. A lint rule name is a class name in camel case and ends with `Rule`.

For example, to suggest gather await calls in a loop, it's better to name as `GatherSequentialAwaitRule` instead of `AwaitInLoopLintRule` (less actionable). If the action need is to remove/cleanup something, it can be named as `No...Rule`, e.g. `NoAssertEqualsRule`.

In this example, we name the rule as NoInheritFromObjectRule.

2.3 Lint Rule Scaffolding

A lint rule is a subclass of *`CstLintRule`* which inherits from *CSTVisitor* in LibCST. LibCST provides visitors for traversing the syntax tree. Defining a `visit_` or `leave_` functions for a specific types of CSTNode allows us to register a callback function to be called during the syntax tree traversal.

In this example, we can inspect all class definitions in a file by defining a `visit_ClassDef` function, which will get called once per class definition encountered during syntax tree traversal.

To add types, we'll need to import ClassDef from libcst, and annotate the function signature:

[3]:
```
from fixit import CstLintRule
import libcst as cst


class NoInheritFromObjectRule(CstLintRule):
    def visit_ClassDef(self, node: cst.ClassDef) -> None:
        ...
```

We don't need to perform any work after visiting our children, so we won't define `leave_ClassDef`. While not needed for this lint rule, if we wanted to visit specific attributes of a given node type, we could specify that attribute as part of the method name too:

[4]:
```
def visit_If(self, node: cst.If) -> None:
    # called first
    ...


def visit_If_test(self, node: cst.If) -> None:
    # called after visit_If, but before we visit the test attribute
    # `leave_If_test` would be called next, followed by `leave_If`.
    if check_something(node.test):
        ...
```

Iteration order of attributes is the same as the order they appear in the source code. In this case, that means visit_If_test is called before visit_If_body and visit_If_orelse.

Use fixit's cli to generate a skeleton of adding a new rule file:

```
$ python -m fixit.cli.add_new_rule # Creates new.py at fixit/rules/new.py
$ python -m fixit.cli.add_new_rule --path fixit/rules/my_rule.py --name rule_name #␣
↪Creates my_rule.py at path specified
```

This will generate a rule file used to create and add new rule to fixit module.

The *fixit.cli.add_new_rule* contains two argument, `-path` and `--name`

- `--path` is used to create rule file at path given in `--path`, defaults to `fixit/rules/new.py`
- `--name` is used to assign the name of the rule and should be in snake case, defaults to the rule file name if `path` provided else *new*. Otherwise, considers the value specified in `--name`

Please provide the name of the rule in snake case without the suffix `rule` as CLI will take care of adding `Rule` to end of the rule name.

[6]:
```
! python -m fixit.cli.add_new_rule --path fixit/rules/my_rule.py --name abcd
```

```
Traceback (most recent call last):
  File "/home/docs/.pyenv/versions/3.7.9/lib/python3.7/runpy.py", line 193, in _run_
↪module_as_main
    "__main__", mod_spec)
  File "/home/docs/.pyenv/versions/3.7.9/lib/python3.7/runpy.py", line 85, in _run_
↪code
    exec(code, run_globals)
  File "/home/docs/checkouts/readthedocs.org/user_builds/fixit/checkouts/v0.1.4/fixit/
↪cli/add_new_rule.py", line 135, in <module>
    register_subparser()
  File "/home/docs/checkouts/readthedocs.org/user_builds/fixit/checkouts/v0.1.4/fixit/
↪cli/add_new_rule.py", line 114, in register_subparser
    sys.exit(_main(add_rule_parser.parse_args()))
```

(continues on next page)

(continued from previous page)

```
  File "/home/docs/.pyenv/versions/3.7.9/lib/python3.7/argparse.py", line 1755, in␣
↪parse_args
    args, argv = self.parse_known_args(args, namespace)
  File "/home/docs/.pyenv/versions/3.7.9/lib/python3.7/argparse.py", line 1787, in␣
↪parse_known_args
    namespace, args = self._parse_known_args(args, namespace)
  File "/home/docs/.pyenv/versions/3.7.9/lib/python3.7/argparse.py", line 1993, in _
↪parse_known_args
    start_index = consume_optional(start_index)
  File "/home/docs/.pyenv/versions/3.7.9/lib/python3.7/argparse.py", line 1933, in␣
↪consume_optional
    take_action(action, args, option_string)
  File "/home/docs/.pyenv/versions/3.7.9/lib/python3.7/argparse.py", line 1845, in␣
↪take_action
    argument_values = self._get_values(action, argument_strings)
  File "/home/docs/.pyenv/versions/3.7.9/lib/python3.7/argparse.py", line 2376, in _
↪get_values
    value = self._get_value(action, arg_string)
  File "/home/docs/.pyenv/versions/3.7.9/lib/python3.7/argparse.py", line 2409, in _
↪get_value
    result = type_func(arg_string)
  File "/home/docs/checkouts/readthedocs.org/user_builds/fixit/checkouts/v0.1.4/fixit/
↪cli/add_new_rule.py", line 63, in is_path_exists
    raise FileExistsError(f"{filepath} already exists")
FileExistsError: fixit/rules/my_rule.py already exists
```

[7]:
```
! cat fixit/rules/my_rule.py
```

```
# Copyright (c) Facebook, Inc. and its affiliates.
#
# This source code is licensed under the MIT license found in the
# LICENSE file in the root directory of this source tree.


import libcst as cst
import libcst.matchers as m

from fixit import CstLintRule, InvalidTestCase as Invalid, ValidTestCase as Valid


"""
This is a model rule file for adding a new rule to fixit module
"""


class AbcdRule(CstLintRule):
    """
    docstring or new_rule description
    """

    MESSAGE = "Enter rule description message"

    VALID = [Valid("'example'")]

    INVALID = [Invalid("'example'")]
```

Now, you can add rule's functionality on top of above generated file. Also, make sure to check the class name in

generated file. Your class name should always be suffixed with `Rule`.

2.4 The Declarative Matcher API

Once we have a `ClassDef` node, we need to see if it contains a base class named `object`. We could implement by inspecting attributes of the node using equality and isinstance.

[8]:
```
# check if any of the base classes of this class def is "object"
def visit_ClassDef(self, node: cst.ClassDef):
    has_object_base = any(
        isinstance(arg.value, cst.Name) and arg.value.value == "object"
        for arg in node.bases
    )
```

Unfortunately, that imperative approach isn't easy to read or write, especially when matching a more complex syntax tree structure. LibCST has a declarative matcher API which allows you to define the shape of an object to match. It's like a regular expression, but for the CST.

[9]:
```
import libcst.matchers as m


def visit_ClassDef(self, node: cst.ClassDef):
    has_object_base = m.matches(
        node, m.ClassDef(bases=[m.AtLeastN(n=1, matcher=m.Arg(value=m.Name("object
↪")))])
    )
```

It makes the code easier to read and maintain.

2.5 Reporting Violations

To report a lint violation, simply call *`report()`* with a CSTNode. Define a lint message via the `MESSAGE` attribute in your lint class. Keep your lint descriptions brief but informative. Link to other documentation if you want to provide an extended explanation. Feedback that you provide to a developer should be clear and actionable. Add a docstring to the rule class to provide more context and the docstring will be included in the generated document.

[10]:
```
class NoInheritFromObjectRule(CstLintRule):
    """
    In Python 3, a class is inherited from ``object`` by default.
    Explicitly inheriting from ``object`` is redundant, so removing it keeps the code␣
↪simpler.
    """
    MESSAGE = "Inheriting from object is a no-op. 'class Foo:' is just fine =)"

    def visit_ClassDef(self, node: cst.ClassDef) -> None:
        new_bases = tuple(
            base for base in node.bases if not m.matches(base.value, m.Name("object"))
        )
        if tuple(node.bases) != new_bases:
            self.report(node)
```

2.6 Adding an Autofix

Warning a user about a problem is nice, but offering to fix it for them is even better! That's what you can do with an auto-fixer. Currently we support replacing a node (use with_changes to modify a CSTNode) or removing it (by passing a libcst.RemovalSentinel as the replacement). In our example, we want to remove any references to object in the base classes of a class definition:

[11]:
```
class NoInheritFromObjectRule(CstLintRule):
    MESSAGE = "Inheriting from object is a no-op. 'class Foo:' is just fine =)"

    def visit_ClassDef(self, node: cst.ClassDef) -> None:
        new_bases = tuple(
            base for base in node.bases if not m.matches(base.value, m.Name("object"))
        )

        if tuple(node.bases) != new_bases:
            # reconstruct classdef, removing parens if bases and keywords are empty
            new_classdef = node.with_changes(
                bases=new_bases,
                lpar=cst.MaybeSentinel.DEFAULT,
                rpar=cst.MaybeSentinel.DEFAULT,
            )

            # report warning and autofix
            self.report(node, replacement=new_classdef)
```

This example also makes use of libcst.MaybeSentinel to properly handle the rendering syntax. In this case, using a MaybeSentinel for the parens fields will remove the parentheses following the class name if removing object from bases means there are no other base classes or keywords in the class definition.

2.7 Skipping Files

Certain behaviors may be acceptable in a set of files, but not in others. We can avoid running the linter on some files by overriding should_skip_file. The properties provided by `self.context` (*BaseContext()*) are useful when implementing `should_skip_file()`.

[12]:
```
class MyRule(CstLintRule):
    def should_skip_file(self):
        # Assert statements are okay for tests.
        # We could check the self.context.file_path object (see pathlib.Path), but
        # Context has a helper property for tests, since this is a common use-case
        return self.context.in_tests
```

Your lint rule does something, but now you need to test it. Testing lint rules is easy, and every lint rule should include test cases. Continue to the next tutorial *Test a Lint Rule*

2.8 Reference

class fixit.**CstLintRule**

MESSAGE
a short message in one or two sentences show to user when the rule is violated.

METADATA_DEPENDENCIES = (<class 'libcst.metadata.position_provider.PositionProvider'>,

should_skip_file() → bool

report(*node: libcst._nodes.base.CSTNode*, *message: Optional[str] = None*, *, position: Optional[libcst._position.CodePosition] = None*, *replacement: Union[libcst._nodes.base.CSTNode, libcst._flatten_sentinel.FlattenSentinel, libcst._removal_sentinel.RemovalSentinel, None] = None*) → None

Report a lint violation for a given node. Optionally specify a custom position to report an error at or a replacement node for an auto-fix.

classmethod requires_metadata_caches() → bool

class fixit.common.base.**BaseContext**

property in_tests

property in_scripts

class fixit.**CstContext**

CHAPTER

THREE

TEST A LINT RULE

This guide assumes you've already started implementing your lint rule. If not, please read *Build a Lint Rule*.

You've written your rule, but now you need a test plan for it. Every lint rule should include some test cases.

3.1 Writing Unit Tests

While manually running your lint rule is useful, a unit test makes it easier to validate that your rule isn't failing to report certain violations, prevents future regressions, and helps your reviewers understand your rule. An ideal lint rule written should warn on every violation (avoiding false-negatives), and never warn where there's no violation (false-positives). For this reason, we require that every rule includes both `VALID` and `INVALID` test cases.

[2]:
```
from fixit import (
    CstLintRule,
    InvalidTestCase as Invalid,
    ValidTestCase as Valid,
)


class NoInheritFromObjectRule(CstLintRule):
    MESSAGE = "Inheriting from object is a no-op. 'class Foo:' is just fine =)"
    VALID = []
    INVALID = []
```

Valid test cases are defined using *`ValidTestCase`* (imported as a shorter name `Valid`), and invalid test cases are defined using *`InvalidTestCase`* (imported as a shorter name `Invalid`). You should add test cases until all potential edge-cases are covered.

[3]:
```
import libcst as cst
import libcst.matchers as m

class NoInheritFromObjectRule(CstLintRule):
    MESSAGE = "Inheriting from object is a no-op. 'class Foo:' is just fine =)"

    VALID = [Valid("class A(something):\n    pass"), Valid("class A:\n    pass")]
    INVALID = []
```

3.2 Adding Autofix Test Case

Testing an auto-fixer is as easy as adding the expected replacement code to your *Invalid* test case. Function strings can be multi-line, similar to docstrings, the leading whitespace is trimmed. The position of the lint suggestion is the

starting position of the reported CSTNode. (In our example, it's the start of `ClassDef` node which is the position of `c` in `class`) It can be specified by providing `line` and `column` numbers in an `Invalid` test case.

[4]:
```
class NoInheritFromObjectRule(CstLintRule):
    """
    In Python 3, a class is inherited from ``object`` by default.
    Explicitly inheriting from ``object`` is redundant, so removing it keeps the code␣
↪simpler.
    """
    MESSAGE = "Inheriting from object is a no-op. 'class Foo:' is just fine =)"
    VALID = [
        Valid("class A(something):    pass"),
        Valid(
            """
            class A:
                pass"""
        ),
    ]
    INVALID = [
        Invalid(
            """
            class B(object):
                pass""",
            line=1,
            column=1,
            expected_replacement="""
            class B:
                pass""",
        ),
        Invalid(
            """
            class B(object, A):
                pass""",
            line=1,
            column=1,
            expected_replacement="""
            class B(A):
                pass""",
        ),
    ]

    def visit_ClassDef(self, node: cst.ClassDef) -> None:
        new_bases = tuple(
            base for base in node.bases if not m.matches(base.value, m.Name("object"))
        )

        if tuple(node.bases) != new_bases:
            # reconstruct classdef, removing parens if bases and keywords are empty
            new_classdef = node.with_changes(
                bases=new_bases,
                lpar=cst.MaybeSentinel.DEFAULT,
                rpar=cst.MaybeSentinel.DEFAULT,
            )

            # report warning and autofix
            self.report(node, replacement=new_classdef)
```

3.3 Run Tests

Fixit provides *add_lint_rule_tests_to_module()* to automatically generate unittest test cases in a module. If you're contributing a new lint rule to Fixit, you can add the rule in `fixit/rules/` of fixit repository. The *add_lint_rule_tests_to_module()* is already configured in `fixit/tests/__init__.py`. Run the added tests by:

[5]:
```
! python -m unittest fixit.tests.NoInheritFromObjectRule
```

```
...
----------------------------------------------------------------------
Ran 4 tests in 0.055s

OK
```

If you're developing a custom lint rule in your codebase, you can configure *add_lint_rule_tests_to_module()* in your test module by passing *globals()* as the *module_attrs* argument and providing a collection of the rules you would like to test as the *rules* argument. E.g. `add_lint_rule_tests_to_module(globals(), {my_package.Rule1, my_package.Rule2})` Then run your test module by:

```
python -m unittest your_test_module
```

[6]:
```
from fixit import add_lint_rule_tests_to_module

add_lint_rule_tests_to_module(globals(), rules=[NoInheritFromObjectRule])
import unittest

unittest.main(argv=["first-arg-is-ignored"], exit=False)
```

```
...
----------------------------------------------------------------------
Ran 4 tests in 0.059s

OK
```

[6]:
```
<unittest.main.TestProgram at 0x7ff8706815d0>
```

3.4 Run Your Rule in a Codebase

The easiest way to see the effects of your lint rule is to run it against the entire codebase. This is pretty easy:

```
python -m fixit.cli.run_rules path --help  # see all the supported flags
python -m fixit.cli.run_rules path --rules RewriteToLiteralRule
```

This runs the linter in parallel across all the Python files in the path. It ignores `# noqa`, `# lint-fixme`, and `# lint-ignore` comments by default, but that can be overridden with the `--use-ignore-comments` flag (see `--help` for more details).

Check out a few of the reported violations to see that generated reports are accurate. If possible, you should consider cleaning up these existing violations before shipping your lint rule. Otherwise, consider that fixing these preexisting violations in future changes may slow other developers down.

3.5 Reference

class fixit.**ValidTestCase**

code : str

filename : str = "not/a/real/file/path.py"

config : common.base.LintConfig = LintConfig(allow_list_rules=[], block_list_patterns=

class fixit.**InvalidTestCase**

code : str

kind : typing.Union[str, NoneType]

line : typing.Union[int, NoneType]

column : typing.Union[int, NoneType]

expected_replacement : typing.Union[str, NoneType]

filename : str = "not/a/real/file/path.py"

config : common.base.LintConfig = LintConfig(allow_list_rules=[], block_list_patterns=

expected_message : typing.Union[str, NoneType]

property expected_str

fixit.**add_lint_rule_tests_to_module**(*module_attrs: Dict[str, Any], rules: Set[Union[Type[fixit.common.base.CstLintRule], Type[fixit.common.pseudo_rule.PseudoLintRule]]], test_case_type: Type[unittest.case.TestCase] = <class 'fixit.common.testing.LintRuleTestCase'>, custom_test_method_name: str = '_test_method', fixture_dir: pathlib.Path = PosixPath('.'), rules_package: str = ''*) → None

Generates classes inheriting from *unittest.TestCase* from the data available in *rules* and adds these to module_attrs. The goal is to facilitate unit test discovery by Python's *unittest* framework. This will provide the capability of testing your lint rules by running commands such as *python -m unittest <your testing module name>*.

module_attrs: A dictionary of attributes we want to add these test cases to. If adding to a module, you can pass *globals()* as the argument.

rules: A collection of classes extending *CstLintRule* to be converted to test cases.

test_case_type: A class extending Python's *unittest.TestCase* that implements a custom test method for testing lint rules to serve as a stencil for test cases. New classes will be generated, and named after each lint rule. They will inherit directly from the class passed into *test_case_type*. If argument is omitted, will default to the *LintRuleTestCase* class from fixit.common.testing.

custom_test_method_name: A member method of the class passed into *test_case_type* parameter that contains the logic around asserting success or failure of CstLintRule's *ValidTestCase* and *InvalidTestCase* test cases. The method will be dynamically renamed to *test_<VALID/INVALID>_<test case index>* for discovery by unittest. If argument is omitted, *add_lint_rule_tests_to_module* will look for a test method named *_test_method* member of *test_case_type*.

fixture_dir: The directory in which fixture files for the passed rules live. Necessary only if any lint rules require fixture data for testing.

rules_package: The name of the rules package. This will be used during the search for fixture files and provides insight into the structure of the fixture directory. The structure of the fixture directory is automatically assumed to mirror the structure of the rules package, eg: *<rules_package>.submodule.module.rule_class* should have fixture files in *<fixture_dir>/submodule/module/rule_class/*.

CHAPTER

FOUR

WHY FIXIT?

There are many Python linter tools. Why do we need another?

Linters are built to help developers write better code and many good linters have been built. In a large codebase, tons of lint suggestions can slow developers down. Developers may spend too much time on fixing code quality suggestions instead of focusing on important progress. At some point, they may start to ignore lint suggestions, even the important ones.

We'd like to help developers move faster by **auto-fixing** lint violations. The first challenge is that most tools analyze source code using AST which doesn't preserve formatting information (comments, whitespaces, etc) and so it is hard for them to suggest high-quality autofixes. We built LibCST to make parsing and modifying source code as a Concrete Syntax Tree easy. Fixit rules take advantage of LibCST APIs to find problematic code and transform it to eliminate the problems.

4.1 Learning from building a Flake8 plugin

Many of our old lint rules are implemented in a monolithic single-file Flake8 plugin. This offered a lot of flexibility and helped facilitate a shared structure for the rules, but it led to severe performance and reliability problems:

- Rules were highly coupled, with a significant amount of shared state and helper functions. It was common to make a small change to one lint rule which broke the others. Because of poor testing practices, these breakages could be missed.
- It wasn't possible to run rules in isolation, making benchmarking and testing more difficult. Disabling a rule in Flake8 consists of running the whole linter and filtering out the disabled results.
- Visitors were responsible for visiting their children. Forgetting to do this could break lint coverage inside of the given construct.
- To work around some of these issues, developers would often define their own node visitor. Traversing the AST multiple times slows down the linter.

The result was a large file with lots of visitors and helper functions which was hard to develop upon.

4.2 Design Principles

When designing Fixit, we used the following list of principles.

- **Autofix.** Lint rules should provide autofix whenever possible to help developers write better code faster. The autofix can run as a codemod on an existing codebase to avoid introducing lots of lint violations when adding a new lint rule.

- **Modular.** A lint rule is a standalone class which keeps its own logic and states from other rules. That makes developing a lint rule simple and less likely to break others.
- **Fast.** All lint rules are applied during a single traversal of the syntax tree and reuse shared metadata to provide lint suggestions fast.
- **Configurable.** Each lint rule is configurable in the config file. That makes lint rules highly customizable. When a lint rule is disabled, it doesn't run at all.
- **Testable.** Testing a lint rule is as simple as providing code examples. These are also used as documentation to help developers understand the line rule.

CHAPTER

FIVE

CONTRIBUTING TO FIXIT

You're very welcome to contribute to Fixit.

5.1 Call for Contribution

1. New Autofixer Rules that help developers write simpler, safer and more efficient code.
2. IDE and workflow integration to build Fixit into development process for better usability.
3. Bug and document fixes.
4. New feature requests.

We want to make contributing to this project as easy and transparent as possible.

5.1.1 Our Development Process

This Github repo is the source of truth and all changes need to be reviewed in pull requests.

5.2 Pull Requests

We actively welcome your pull requests.

1. Fork the repo and create your branch from `master`.
2. If you've added code that should be tested, add tests.
3. If you've changed APIs, update the documentation.
4. Ensure the test suite passes by `tox test`.
5. Make sure your code lints.
6. If you haven't already, complete the Contributor License Agreement ("CLA").

5.3 Contributor License Agreement ("CLA")

In order to accept your pull request, we need you to submit a CLA. You only need to do this once to work on any of Facebook's open source projects.

Complete your CLA here: <https://code.facebook.com/cla>

5.4 Issues

We use GitHub issues to track public bugs. Please ensure your description is clear and has sufficient instructions to be able to reproduce the issue.

Facebook has a bounty program for the safe disclosure of security bugs. In those cases, please go through the process outlined on that page and do not file a public issue.

5.5 Coding Style

We use Fixit, flake8, isort and black to enforce coding style. Code can be autoformatted by `tox -e autofix`.

5.6 License

By contributing to Fixit, you agree that your contributions will be licensed under the MIT LICENSE file in the root directory of this source tree.

CHAPTER

SIX

ADDMISSINGHEADERRULE

Verify if required header comments exist in a module. Configuration is required in `.fixit.config.yaml` in order to enable this rule:

```
rule_config:
   AddMissingHeaderRule:
       path: pkg/*.py
       header: |-
           # header line 1
           # header line 2
```

(Use `|-` to keep newlines and add no new line at the end of header comments.) `path` is a glob-style `str` used in Path.match(). The specified `header` is a newline-separated `str` to be enforced in files whose path matches.

6.1 Message

A required header comment for this file is missing.

6.2 Has Autofix: Yes

6.3 VALID Code Examples

1:

```
import libcst
```

2:

config:

```
repo_root: .
rule_config:
  AddMissingHeaderRule:
    header: '# header line 1

      # header line 2'
    path: '*.py'
```

```
# header line 1
# header line 2
import libcst
```

3:

config:

```
repo_root: .
rule_config:
  AddMissingHeaderRule:
    header: '# header line 1

      # header line 2'
    path: '*.py'
```

```
# header line 1
# header line 2
# An extra header line is ok
import libcst
```

4:

config:

```
repo_root: .
rule_config:
  AddMissingHeaderRule:
    header: '# header line 1

      # header line 2'
    path: a/*.py
```

path: `b/m.py`

```
# other header in an unrelated file
import libcst
```

6.4 INVALID Code Examples

1:

config:

```
repo_root: .
rule_config:
  AddMissingHeaderRule:
    header: '# header line'
    path: '*.py'
```

```
# wrong header
```

Autofix:

```
---
+++
@@ -1 +1,2 @@
+# header line
 # wrong header
```

2:

config:

```
repo_root: .
rule_config:
  AddMissingHeaderRule:
    header: '# header line'
    path: '*.py'
```

```
#!/usr/bin/env python
# wrong header
```

Autofix:

```
---
+++
@@ -1,3 +1,3 @@
-
 #!/usr/bin/env python
+# header line
 # wrong header
```

CHAPTER

SEVEN

AVOIDORINEXCEPTRULE

Discourages use of `or` in except clauses. If an except clause needs to catch multiple exceptions, they must be expressed as a parenthesized tuple, for example: `except (ValueError, TypeError)` (https://docs.python.org/3/tutorial/errors.html#handling-exceptions)

When `or` is used, only the first operand exception type of the conditional statement will be caught. For example:

```
In [1]: class Exc1(Exception):
    ...:     pass
    ...:

In [2]: class Exc2(Exception):
    ...:     pass
    ...:

In [3]: try:
    ...:     raise Exception()
    ...: except Exc1 or Exc2:
    ...:     print("caught!")
    ...:
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-3-3340d66a006c> in <module>
    1 try:
----> 2     raise Exception()
    3 except Exc1 or Exc2:
    4     print("caught!")
    5

Exception:

In [4]: try:
    ...:     raise Exc1()
    ...: except Exc1 or Exc2:
    ...:     print("caught!")
    ...:
    caught!

In [5]: try:
    ...:     raise Exc2()
    ...: except Exc1 or Exc2:
    ...:     print("caught!")
    ...:
---------------------------------------------------------------------------
Exc2                                      Traceback (most recent call last)
```

(continues on next page)

(continued from previous page)

```
<ipython-input-5-5d29c1589cc0> in <module>
    1 try:
----> 2     raise Exc2()
    3 except Exc1 or Exc2:
    4     print("caught!")
    5

Exc2:
```

7.1 Message

Avoid using 'or' in an except block. For example:'except ValueError or TypeError' only catches 'ValueError'. Instead, use parentheses, 'except (ValueError, TypeError)'

7.2 Has Autofix: No

7.3 VALID Code Examples

1:

```
try:
    print()
except (ValueError, TypeError) as err:
    pass
```

7.4 INVALID Code Examples

1:

```
try:
    print()
except ValueError or TypeError:
    pass
```

CHAPTER

EIGHT

AWAITASYNCCALLRULE

Enforces calls to coroutines are preceeded by the `await` keyword. Awaiting on a coroutine will execute it while simply calling a coroutine returns a coroutine object (https://docs.python.org/3/library/asyncio-task.html#coroutines).

8.1 Message

Async function call will only be executed with *await* statement. Did you forget to add *await*? If you intend to not await, please add comment to disable this warning: # lint-fixme: AwaitAsyncCallRule

8.2 Has Autofix: Yes

8.3 VALID Code Examples

1:

```
async def async_func():
    await async_foo()
```

2:

```
def foo(): pass
foo()
```

3:

```
async def foo(): pass
async def bar():
    await foo()
```

4:

```
async def foo(): pass
async def bar():
    x = await foo()
```

5:

```
async def foo() -> bool: pass
async def bar():
    while not await foo(): pass
```

6:

```
import asyncio
async def foo(): pass
asyncio.run(foo())
```

8.4 INVALID Code Examples

1:

```
async def foo(): pass
async def bar():
    foo()
```

Autofix:

```
---
+++
@@ -1,4 +1,4 @@

 async def foo(): pass
 async def bar():
-    foo()
+    await foo()
```

2:

```
class Foo:
    async def _attr(self): pass
obj = Foo()
obj._attr
```

Autofix:

```
---
+++
@@ -2,4 +2,4 @@
 class Foo:
     async def _attr(self): pass
 obj = Foo()
-obj._attr
+await obj._attr
```

3:

```
class Foo:
    async def _method(self): pass
obj = Foo()
obj._method()
```

Autofix:

```
---
+++
@@ -2,4 +2,4 @@
```

(continues on next page)

(continued from previous page)

```
 class Foo:
     async def _method(self): pass
 obj = Foo()
-obj._method()
+await obj._method()
```

4:

```
class Foo:
    async def _method(self): pass
obj = Foo()
result = obj._method()
```

Autofix:

```
---
+++
@@ -2,4 +2,4 @@
 class Foo:
     async def _method(self): pass
 obj = Foo()
-result = obj._method()
+result = await obj._method()
```

5:

```
class Foo:
    async def bar(): pass
class NodeUser:
    async def get():
        do_stuff()
        return Foo()
user = NodeUser.get().bar()
```

Autofix:

```
---
+++
@@ -5,4 +5,4 @@
     async def get():
         do_stuff()
         return Foo()
-user = NodeUser.get().bar()
+user = await NodeUser.get().bar()
```

6:

```
class Foo:
    async def _attr(self): pass
obj = Foo()
attribute = obj._attr
```

Autofix:

```
---
+++
```

(continues on next page)

(continued from previous page)

```
@@ -2,4 +2,4 @@
 class Foo:
     async def _attr(self): pass
 obj = Foo()
-attribute = obj._attr
+attribute = await obj._attr
```

7:

```
async def foo() -> bool: pass
x = True
if x and foo(): pass
```

Autofix:

```
---
+++
@@ -1,4 +1,4 @@

 async def foo() -> bool: pass
 x = True
-if x and foo(): pass
+if x and await foo(): pass
```

8:

```
async def foo() -> bool: pass
x = True
are_both_true = x and foo()
```

Autofix:

```
---
+++
@@ -1,4 +1,4 @@

 async def foo() -> bool: pass
 x = True
-are_both_true = x and foo()
+are_both_true = x and await foo()
```

9:

```
async def foo() -> bool: pass
if foo():
    do_stuff()
```

Autofix:

```
---
+++
@@ -1,4 +1,4 @@

 async def foo() -> bool: pass
-if foo():
+if await foo():
     do_stuff()
```

10:

```
async def foo() -> bool: pass
if not foo():
    do_stuff()
```

Autofix:

```
---
+++
@@ -1,4 +1,4 @@

 async def foo() -> bool: pass
-if not foo():
+if not await foo():
     do_stuff()
```

11:

```
class Foo:
    async def _attr(self): pass
    def bar(self):
        if self._attr: pass
```

Autofix:

```
---
+++
@@ -2,4 +2,4 @@
 class Foo:
     async def _attr(self): pass
     def bar(self):
-        if self._attr: pass
+        if await self._attr: pass
```

12:

```
class Foo:
    async def _attr(self): pass
    def bar(self):
        if not self._attr: pass
```

Autofix:

```
---
+++
@@ -2,4 +2,4 @@
 class Foo:
     async def _attr(self): pass
     def bar(self):
-        if not self._attr: pass
+        if not await self._attr: pass
```

13:

```
class Foo:
    async def _attr(self): pass
def bar() -> Foo:
```

(continues on next page)

(continued from previous page)

```
    return Foo()
attribute = bar()._attr
```

Autofix:

```
---
+++
@@ -3,4 +3,4 @@
     async def _attr(self): pass
 def bar() -> Foo:
     return Foo()
-attribute = bar()._attr
+attribute = await bar()._attr
```

14:

```
class Foo:
    def _attr(self): pass
async def bar():
    await do_stuff()
    return Foo()
attribute = bar()._attr
```

Autofix:

```
---
+++
@@ -4,4 +4,4 @@
 async def bar():
     await do_stuff()
     return Foo()
-attribute = bar()._attr
+attribute = await bar()._attr
```

15:

```
async def bar() -> bool: pass
while bar(): pass
```

Autofix:

```
---
+++
@@ -1,3 +1,3 @@

 async def bar() -> bool: pass
-while bar(): pass
+while await bar(): pass
```

16:

```
async def bar() -> bool: pass
while not bar(): pass
```

Autofix:

```
---
+++
@@ -1,3 +1,3 @@

 async def bar() -> bool: pass
-while not bar(): pass
+while not await bar(): pass
```

17:

```
class Foo:
    @classmethod
    async def _method(cls): pass
Foo._method()
```

Autofix:

```
---
+++
@@ -2,4 +2,4 @@
 class Foo:
     @classmethod
     async def _method(cls): pass
-Foo._method()
+await Foo._method()
```

18:

```
class Foo:
    @staticmethod
    async def _method(self): pass
Foo._method()
```

Autofix:

```
---
+++
@@ -2,4 +2,4 @@
 class Foo:
     @staticmethod
     async def _method(self): pass
-Foo._method()
+await Foo._method()
```

CHAPTER

NINE

COLLAPSEISINSTANCECHECKSRULE

The built-in `isinstance` function, instead of a single type, can take a tuple of types and check whether given target suits any of them. Rather than chaining multiple `isinstance` calls with a boolean-or operation, a single `isinstance` call where the second argument is a tuple of all types can be used.

9.1 Message

Multiple isinstance calls with the same target but different types can be collapsed into a single call with a tuple of types.

9.2 Has Autofix: Yes

9.3 VALID Code Examples

1:

```
foo() or foo()
```

2:

```
foo(x, y) or foo(x, z)
```

3:

```
foo(x, y) or foo(x, z) or foo(x, q)
```

4:

```
isinstance() or isinstance()
```

5:

```
isinstance(x) or isinstance(x)
```

6:

```
isinstance(x, y) or isinstance(x)
```

7:

```
isinstance(x) or isinstance(x, y)
```

8:

```
isinstance(x, y) or isinstance(t, y)
```

9:

```
isinstance(x, y) and isinstance(x, z)
```

10:

```
isinstance(x, y) or isinstance(x, (z, q))
```

11:

```
isinstance(x, (y, z)) or isinstance(x, q)
```

12:

```
isinstance(x, a) or isinstance(y, b) or isinstance(z, c)
```

13:

```
def foo():
    def isinstance(x, y):
        return _foo_bar(x, y)
    if isinstance(x, y) or isinstance(x, z):
        print("foo")
```

9.4 INVALID Code Examples

1:

```
isinstance(x, y) or isinstance(x, z)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-isinstance(x, y) or isinstance(x, z)
+isinstance(x, (y, z))
```

2:

```
isinstance(x, y) or isinstance(x, z) or isinstance(x, q)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-isinstance(x, y) or isinstance(x, z) or isinstance(x, q)
+isinstance(x, (y, z, q))
```

3:

```
something or isinstance(x, y) or isinstance(x, z) or another
```

Autofix:

```
---
+++
@@ -1 +1 @@
-something or isinstance(x, y) or isinstance(x, z) or another
+something or isinstance(x, (y, z)) or another
```

4:

```
isinstance(x, y) or isinstance(x, z) or isinstance(x, q) or isinstance(x, w)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-isinstance(x, y) or isinstance(x, z) or isinstance(x, q) or isinstance(x, w)
+isinstance(x, (y, z, q, w))
```

5:

```
isinstance(x, a) or isinstance(x, b) or isinstance(y, c) or isinstance(y, d)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-isinstance(x, a) or isinstance(x, b) or isinstance(y, c) or isinstance(y, d)
+isinstance(x, (a, b)) or isinstance(y, (c, d))
```

6:

```
isinstance(x, a) or isinstance(x, b) or isinstance(y, c) or isinstance(y, d) or␣
↪isinstance(z, e)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-isinstance(x, a) or isinstance(x, b) or isinstance(y, c) or isinstance(y, d) or␣
↪isinstance(z, e)
+isinstance(x, (a, b)) or isinstance(y, (c, d)) or isinstance(z, e)
```

7:

```
isinstance(x, a) or isinstance(x, b) or isinstance(y, c) or isinstance(y, d) or␣
↪isinstance(z, e) or isinstance(q, f) or isinstance(q, g) or isinstance(q, h)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-isinstance(x, a) or isinstance(x, b) or isinstance(y, c) or isinstance(y, d) or␣
↪isinstance(z, e) or isinstance(q, f) or isinstance(q, g) or isinstance(q, h)
+isinstance(x, (a, b)) or isinstance(y, (c, d)) or isinstance(z, e) or isinstance(q,␣
↪(f, g, h))
```

CHAPTER

TEN

COMPAREPRIMITIVESBYEQUALRULE

Enforces the use of == and != in comparisons to primitives rather than `is` and `is not`. The == operator checks equality (https://docs.python.org/3/reference/datamodel.html#object.__eq__), while `is` checks identity (https://docs.python.org/3/reference/expressions.html#is).

10.1 Message

Don't use *is* or *is not* to compare primitives, as they compare references. Use == or != instead.

10.2 Has Autofix: Yes

10.3 VALID Code Examples

1:

```
a == 1
```

2:

```
a == '1'
```

3:

```
a != '1'
```

4:

```
'3' == '1'
```

5:

```
3 == '1'
```

6:

```
3 > 2 > 1
```

7:

```
3 > 2 > '1'
```

8:

```
a is b > 1
```

9:

```
a is b is c
```

10:

```
1 > b is c
```

10.4 INVALID Code Examples

1:

```
a is 1
```

Autofix:

```
---
+++
@@ -1 +1 @@
-a is 1
+a == 1
```

2:

```
a is '1'
```

Autofix:

```
---
+++
@@ -1 +1 @@
-a is '1'
+a == '1'
```

3:

```
a is f'1{b}'
```

Autofix:

```
---
+++
@@ -1 +1 @@
-a is f'1{b}'
+a == f'1{b}'
```

4:

```
a is not f'1{d}'
```

Autofix:

```
---
+++
@@ -1 +1 @@
-a is not f'1{d}'
+a != f'1{d}'
```

5:

```
1 is a
```

Autofix:

```
---
+++
@@ -1 +1 @@
-1 is a
+1 == a
```

6:

```
'2' > '1' is a
```

Autofix:

```
---
+++
@@ -1 +1 @@
-'2' > '1' is a
+'2' > '1' == a
```

7:

```
3 > a is 2
```

Autofix:

```
---
+++
@@ -1 +1 @@
-3 > a is 2
+3 > a == 2
```

8:

```
1  is   2
```

Autofix:

```
---
+++
@@ -1 +1 @@
-1  is   2
+1  ==   2
```

CHAPTER

ELEVEN

COMPARESINGLETONPRIMITIVESBYISRULE

Enforces the use of *is* and *is not* in comparisons to singleton primitives (None, True, False) rather than == and !=. The == operator checks equality, when in this scenario, we want to check identity. See Flake8 rules E711 (https://www.flake8rules.com/rules/E711.html) and E712 (https://www.flake8rules.com/rules/E712.html).

11.1 Message

Comparisons to singleton primitives should not be done with == or !=, as they check equality rather than identiy. Use *is* or *is not* instead.

11.2 Has Autofix: Yes

11.3 VALID Code Examples

1:

```
if x: pass
```

2:

```
if not x: pass
```

3:

```
x is True
```

4:

```
x is False
```

5:

```
x is None
```

6:

```
x is not None
```

7:

```
x is True is not y
```

8:

```
y is None is not x
```

9:

```
None is y
```

10:

```
True is x
```

11:

```
False is x
```

12:

```
x == 2
```

13:

```
2 != x
```

11.4 INVALID Code Examples

1:

```
x != True
```

Autofix:

```
---
+++
@@ -1 +1 @@
-x != True
+x is not True
```

2:

```
x != False
```

Autofix:

```
---
+++
@@ -1 +1 @@
-x != False
+x is not False
```

3:

```
x == False
```

Autofix:

```
---
+++
@@ -1 +1 @@
-x == False
+x is False
```

4:

```
x == None
```

Autofix:

```
---
+++
@@ -1 +1 @@
-x == None
+x is None
```

5:

```
x != None
```

Autofix:

```
---
+++
@@ -1 +1 @@
-x != None
+x is not None
```

6:

```
False == x
```

Autofix:

```
---
+++
@@ -1 +1 @@
-False == x
+False is x
```

7:

```
x is True == y
```

Autofix:

```
---
+++
@@ -1 +1 @@
-x is True == y
+x is True is y
```

CHAPTER

TWELVE

EXPLICITFROZENDATACLASSRULE

Encourages the use of frozen dataclass objects by telling users to specify the kwarg.

Without this lint rule, most users of dataclass won't know to use the kwarg, and may unintentionally end up with mutable objects.

12.1 Message

When using dataclasses, explicitly specify a frozen keyword argument. Example: *@dataclass(frozen=True)* or *@dataclass(frozen=False)*. Docs: https://docs.python.org/3/library/dataclasses.html

12.2 Has Autofix: Yes

12.3 VALID Code Examples

1:

```
@some_other_decorator
class Cls: pass
```

2:

```
from dataclasses import dataclass
@dataclass(frozen=False)
class Cls: pass
```

3:

```
import dataclasses
@dataclasses.dataclass(frozen=False)
class Cls: pass
```

4:

```
import dataclasses as dc
@dc.dataclass(frozen=False)
class Cls: pass
```

5:

```
from dataclasses import dataclass as dc
@dc(frozen=False)
class Cls: pass
```

12.4 INVALID Code Examples

1:

```
from dataclasses import dataclass
@some_unrelated_decorator
@dataclass  # not called as a function
@another_unrelated_decorator
class Cls: pass
```

Autofix:

```
---
+++
@@ -1,6 +1,6 @@

 from dataclasses import dataclass
 @some_unrelated_decorator
-@dataclass  # not called as a function
+@dataclass(frozen=True)  # not called as a function
 @another_unrelated_decorator
 class Cls: pass
```

2:

```
from dataclasses import dataclass
@dataclass()  # called as a function, no kwargs
class Cls: pass
```

Autofix:

```
---
+++
@@ -1,4 +1,4 @@

 from dataclasses import dataclass
-@dataclass()  # called as a function, no kwargs
+@dataclass(frozen=True)  # called as a function, no kwargs
 class Cls: pass
```

3:

```
from dataclasses import dataclass
@dataclass(other_kwarg=False)
class Cls: pass
```

Autofix:

```
---
+++
```

(continues on next page)

(continued from previous page)

```
@@ -1,4 +1,4 @@

 from dataclasses import dataclass
-@dataclass(other_kwarg=False)
+@dataclass(other_kwarg=False, frozen=True)
 class Cls: pass
```

4:

```
import dataclasses
@dataclasses.dataclass
class Cls: pass
```

Autofix:

```
---
+++
@@ -1,4 +1,4 @@

 import dataclasses
-@dataclasses.dataclass
+@dataclasses.dataclass(frozen=True)
 class Cls: pass
```

5:

```
import dataclasses
@dataclasses.dataclass()
class Cls: pass
```

Autofix:

```
---
+++
@@ -1,4 +1,4 @@

 import dataclasses
-@dataclasses.dataclass()
+@dataclasses.dataclass(frozen=True)
 class Cls: pass
```

6:

```
import dataclasses
@dataclasses.dataclass(other_kwarg=False)
class Cls: pass
```

Autofix:

```
---
+++
@@ -1,4 +1,4 @@

 import dataclasses
-@dataclasses.dataclass(other_kwarg=False)
+@dataclasses.dataclass(other_kwarg=False, frozen=True)
 class Cls: pass
```

7:

```
from dataclasses import dataclass as dc
@dc
class Cls: pass
```

Autofix:

```
---
+++
@@ -1,4 +1,4 @@

 from dataclasses import dataclass as dc
-@dc
+@dc(frozen=True)
 class Cls: pass
```

8:

```
from dataclasses import dataclass as dc
@dc()
class Cls: pass
```

Autofix:

```
---
+++
@@ -1,4 +1,4 @@

 from dataclasses import dataclass as dc
-@dc()
+@dc(frozen=True)
 class Cls: pass
```

9:

```
from dataclasses import dataclass as dc
@dc(other_kwarg=False)
class Cls: pass
```

Autofix:

```
---
+++
@@ -1,4 +1,4 @@

 from dataclasses import dataclass as dc
-@dc(other_kwarg=False)
+@dc(other_kwarg=False, frozen=True)
 class Cls: pass
```

10:

```
import dataclasses as dc
@dc.dataclass
class Cls: pass
```

Autofix:

```
---
+++
@@ -1,4 +1,4 @@

 import dataclasses as dc
-@dc.dataclass
+@dc.dataclass(frozen=True)
 class Cls: pass
```

11:

```
import dataclasses as dc
@dc.dataclass()
class Cls: pass
```

Autofix:

```
---
+++
@@ -1,4 +1,4 @@

 import dataclasses as dc
-@dc.dataclass()
+@dc.dataclass(frozen=True)
 class Cls: pass
```

12:

```
import dataclasses as dc
@dc.dataclass(other_kwarg=False)
class Cls: pass
```

Autofix:

```
---
+++
@@ -1,4 +1,4 @@

 import dataclasses as dc
-@dc.dataclass(other_kwarg=False)
+@dc.dataclass(other_kwarg=False, frozen=True)
 class Cls: pass
```

CHAPTER

THIRTEEN

GATHERSEQUENTIALAWAITRULE

Discourages awaiting coroutines in a loop as this will run them sequentially. Using `asyncio.gather()` will run them concurrently.

13.1 Message

Using await in a loop will run async function sequentially. Use asyncio.gather() to run async functions concurrently.

13.2 Has Autofix: No

13.3 VALID Code Examples

1:

```
async def async_foo():
    return await async_bar()
```

2:

path: `foo/tests/test_foo.py`

```
# await in a loop is fine if it's a test.
async def async_check_call():
    for _i in range(0, 2):
        await async_foo()
```

13.4 INVALID Code Examples

1:

```
async def async_check_call():
    for _i in range(0, 2):
        await async_foo()
```

2:

```
async def async_check_assignment():
    for _i in range(0, 2):
        x = await async_foo()
```

3:

```
async def async_check_list_comprehension():
    [await async_foo() for _i in range(0, 2)]
```

CHAPTER

FOURTEEN

IMPORTCONSTRAINTSRULE

Rule to impose import constraints in certain directories to improve runtime performance. The directories specified in the ImportConstraintsRule setting in the `.fixit.config.yaml` file's `rule_config` section can impose import constraints for that directory and its children as follows:

```
rule_config:
    ImportConstraintsRule:
        dir_under_repo_root:
            rules: [
                ["module_under_repo_root", "allow"],
                ["another_module_under_repo_root, "deny"],
                ["*", "deny"]
            ]
            ignore_tests: True
            ignore_types: True
            message: "'{imported}' cannot be imported from within '{current_file}'."
```

Each rule under `rules` is evaluated in order from top to bottom and the last rule for each directory should be a wildcard rule. Rules can be `"allow"`, `"deny"`, `"allow_global"`, `"allow_local"`, `"deny_global"` or `"deny_local"`. `ignore_tests` and *ignore_types* should carry boolean values and can be omitted. They are both set to *True* by default. If `ignore_types` is True, this rule will ignore imports inside `if TYPE_CHECKING` blocks since those imports do not have an affect on runtime performance. If `ignore_tests` is True, this rule will not lint any files found in a testing module. If `message` is passed, it must be a string containing a custom message. The string will be formatted passing `imported` and `current_file` variables to be used if needed, with the symbol being imported and the current file, respectively.

14.1 Message

According to the settings for this directory in the .fixit.config.yaml configuration file, '{imported}' cannot be imported from within '{current_file}'.

14.2 Has Autofix: No

14.3 VALID Code Examples

1:

```
import common
```

2:

config:

```
rule_config:
  ImportConstraintsRule:
    some_dir:
      rules:
      - - '*'
        - allow
```

path: `some_dir/file.py`

```
import common
```

3:

config:

```
rule_config:
  ImportConstraintsRule:
    some_dir:
      rules:
      - - common
        - allow
      - - '*'
        - deny
```

path: `some_dir/file.py`

```
import common
```

4:

config:

```
rule_config:
  ImportConstraintsRule:
    some_dir:
      rules:
      - - common
        - allow
      - - '*'
        - deny
```

path: `some_dir/file.py`

```
from common.foo import bar
```

5:

config:

```
rule_config:
  ImportConstraintsRule:
    some_dir:
      rules:
      - - common.foo.bar
        - allow
```

(continues on next page)

(continued from previous page)

```
      - - common
        - deny
      - - '*'
        - deny
```

path: `some_dir/file.py`

```
from common.foo import bar
```

6:

config:

```
rule_config:
  ImportConstraintsRule:
    some_dir:
      rules:
      - - '*'
        - deny
```

path: `some_dir/file.py`

```
import ast
```

7:

config:

```
rule_config:
  ImportConstraintsRule:
    .:
      rules:
      - - common.safe
        - allow
      - - '*'
        - deny
```

path: `common/safe/file.py`

```
from . import module
```

8:

config:

```
rule_config:
  ImportConstraintsRule:
    common:
      rules:
      - - common.safe
        - allow
      - - '*'
        - deny
```

path: `common/unsafe/file.py`

```
from ..safe import module
```

9:

config:

```
rule_config:
  ImportConstraintsRule:
    .:
      rules:
      - - '*'
        - deny
```

path: `file.py`

```
from ..................................... import module
```

10:

config:

```
rule_config:
  ImportConstraintsRule:
    dir_1:
      rules:
      - - common.foo.bar
        - deny
      - - '*'
        - deny
    dir_1/dir_2:
      rules:
      - - common.foo.bar
        - allow
      - - '*'
        - deny
```

path: `dir_1/dir_2/file.py`

```
from common.foo import bar
```

11:

config:

```
rule_config:
  ImportConstraintsRule:
    dir_1:
      rules:
      - - common.foo.bar
        - deny
      - - '*'
        - deny
    dir_1/dir_2:
      rules:
      - - common.foo.bar
        - allow
      - - '*'
        - deny
```

path: `dir_1/dir_2/file.py`

```
from common.foo import bar
```

12:

config:

```
rule_config:
  ImportConstraintsRule:
    dir:
      rules:
      - - '*'
        - deny_global
```

path: `dir/file.py`

```
def local_scope():
    import common
```

13:

config:

```
rule_config:
  ImportConstraintsRule:
    dir:
      rules:
      - - '*'
        - deny_local
```

path: `dir/file.py`

```
import common
```

14:

config:

```
rule_config:
  ImportConstraintsRule:
    dir:
      rules:
      - - '*'
        - allow_global
```

path: `dir/file.py`

```
import common
```

15:

config:

```
rule_config:
  ImportConstraintsRule:
    dir:
      rules:
      - - '*'
        - allow_local
```

path: `dir/file.py`

```
def local_scope():
    import common
```

14.4 INVALID Code Examples

1:

config:

```
rule_config:
  ImportConstraintsRule:
    some_dir:
      rules:
      - - '*'
        - deny
```

path: `some_dir/file.py`

```
import common
```

2:

config:

```
rule_config:
  ImportConstraintsRule:
    '*/file.py':
      rules:
      - - '*'
        - deny
```

path: `some_dir/file.py`

```
import common
```

3:

config:

```
rule_config:
  ImportConstraintsRule:
    some_dir:
      rules:
      - - common.foo.bar
        - deny
      - - common
        - allow
      - - '*'
        - allow
```

path: `some_dir/file.py`

```
from common.foo import bar
```

4:

config:

```
rule_config:
  ImportConstraintsRule:
    some_dir:
      rules:
      - - common
        - deny
      - - '*'
        - allow
```

path: `some_dir/file.py`

```
import common as not_common
```

5:

config:

```
rule_config:
  ImportConstraintsRule:
    some_dir:
      rules:
      - - common.bar
        - deny
      - - '*'
        - allow
```

path: `some_dir/file.py`

```
from common import bar as not_bar
```

6:

config:

```
rule_config:
  ImportConstraintsRule:
    common:
      rules:
      - - '*'
        - deny
```

path: `common/a.py`

```
from . import b
```

7:

config:

```
rule_config:
  ImportConstraintsRule:
    dir_1:
      rules:
      - - common.foo.bar
        - allow
```

(continues on next page)

(continued from previous page)

```
      - - '*'
        - deny
    dir_1/dir_2:
      rules:
      - - '*'
        - deny
```

path: `dir_1/dir_2/file.py`

```
from common.foo import bar
```

8:

config:

```
rule_config:
  ImportConstraintsRule:
    dir_1:
      rules:
      - - common
        - allow
      - - '*'
        - deny
    dir_1/dir_2:
      rules:
      - - '*'
        - deny
```

path: `dir_1/dir_2/file.py`

```
import common
```

9:

config:

```
rule_config:
  ImportConstraintsRule:
    dir:
      rules:
      - - '*'
        - deny_global
```

path: `dir/file.py`

```
import common
```

10:

config:

```
rule_config:
  ImportConstraintsRule:
    dir:
      rules:
      - - '*'
        - deny_local
```

path: `dir/file.py`

```
def local_scope():
    import common
```

11:

config:

```
rule_config:
  ImportConstraintsRule:
    dir:
      rules:
      - - '*'
        - allow_global
```

path: `dir/file.py`

```
def local_scope():
    import common
```

12:

config:

```
rule_config:
  ImportConstraintsRule:
    dir:
      rules:
      - - '*'
        - allow_local
```

path: `dir/file.py`

```
import common
```

13:

config:

```
rule_config:
  ImportConstraintsRule:
    dir:
      message: '''{imported}'' cannot be imported from ''{current_file}'''
      rules:
      - - '*'
        - deny
```

path: `dir/file.py`

```
import common
```

CHAPTER

FIFTEEN

NOASSERTEQUALSRULE

Discourages use of `assertEquals` as it is deprecated (see https://docs.python.org/2/library/unittest.html#deprecated-aliases and https://bugs.python.org/issue9424). Use the standardized `assertEqual` instead.

15.1 Message

“assertEquals” is deprecated, use “assertEqual” instead. See https://docs.python.org/2/library/unittest.html#deprecated-aliases and https://bugs.python.org/issue9424.

15.2 Has Autofix: Yes

15.3 VALID Code Examples

1:

```
self.assertEqual(a, b)
```

15.4 INVALID Code Examples

1:

```
self.assertEquals(a, b)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-self.assertEquals(a, b)
+self.assertEqual(a, b)
```

CHAPTER

SIXTEEN

NOASSERTTRUEFORCOMPARISONSRULE

Finds incorrect use of `assertTrue` when the intention is to compare two values. These calls are replaced with `assertEqual`. Comparisons with True, False and None are replaced with one-argument calls to `assertTrue`, `assertFalse` and `assertIsNone`.

16.1 Message

“assertTrue” does not compare its arguments, use “assertEqual” or other appropriate functions.

16.2 Has Autofix: Yes

16.3 VALID Code Examples

1:

```
self.assertTrue(a == b)
```

2:

```
self.assertTrue(data.is_valid(), "is_valid() method")
```

3:

```
self.assertTrue(validate(len(obj.getName(type=SHORT))))
```

4:

```
self.assertTrue(condition, message_string)
```

16.4 INVALID Code Examples

1:

```
self.assertTrue(a, 3)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-self.assertTrue(a, 3)
+self.assertEqual(a, 3)
```

2:

```
self.assertTrue(hash(s[:4]), 0x1234)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-self.assertTrue(hash(s[:4]), 0x1234)
+self.assertEqual(hash(s[:4]), 0x1234)
```

3:

```
self.assertTrue(list, [1, 3])
```

Autofix:

```
---
+++
@@ -1 +1 @@
-self.assertTrue(list, [1, 3])
+self.assertEqual(list, [1, 3])
```

4:

```
self.assertTrue(optional, None)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-self.assertTrue(optional, None)
+self.assertIsNone(optional)
```

5:

```
self.assertTrue(b == a, True)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-self.assertTrue(b == a, True)
+self.assertTrue(b == a)
```

6:

```
self.assertTrue(b == a, False)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-self.assertTrue(b == a, False)
+self.assertFalse(b == a)
```

CHAPTER

SEVENTEEN

NOINHERITFROMOBJECTRULE

In Python 3, a class is inherited from `object` by default. Explicitly inheriting from `object` is redundant, so removing it keeps the code simpler.

17.1 Message

Inheriting from object is a no-op. 'class Foo:' is just fine =)

17.2 Has Autofix: Yes

17.3 VALID Code Examples

1:

```
class A(something):    pass
```

2:

```
class A:
    pass
```

17.4 INVALID Code Examples

1:

```
class B(object):
    pass
```

Autofix:

```
---
+++
@@ -1,3 +1,3 @@

-class B(object):
+class B:
     pass
```

2:

```
class B(object, A):
    pass
```

Autofix:

```
---
+++
@@ -1,3 +1,3 @@

-class B(object, A):
+class B(A):
     pass
```

CHAPTER

EIGHTEEN

NONAMEDTUPLERULE

Enforce the use of `dataclasses.dataclass` decorator instead of `NamedTuple` for cleaner customization and inheritance. It supports default value, combining fields for inheritance, and omitting optional fields at instantiation. See PEP 557. `@dataclass` is faster at reading an object's nested properties and executing its methods. (benchmark)

18.1 Message

Instead of NamedTuple, consider using the @dataclass decorator from dataclasses instead for simplicity, efficiency and consistency.

18.2 Has Autofix: Yes

18.3 VALID Code Examples

1:

```
@dataclass(frozen=True)
class Foo:
    pass
```

2:

```
@dataclass(frozen=False)
class Foo:
    pass
```

3:

```
class Foo:
    pass
```

4:

```
class Foo(SomeOtherBase):
    pass
```

5:

```
@some_other_decorator
class Foo:
    pass
```

6:

```
@some_other_decorator
class Foo(SomeOtherBase):
    pass
```

18.4 INVALID Code Examples

1:

```
from typing import NamedTuple

class Foo(NamedTuple):
    pass
```

Autofix:

```
---
+++
@@ -1,5 +1,6 @@

 from typing import NamedTuple

-class Foo(NamedTuple):
+@dataclass(frozen=True)
+class Foo:
     pass
```

2:

```
from typing import NamedTuple as NT

class Foo(NT):
    pass
```

Autofix:

```
---
+++
@@ -1,5 +1,6 @@

 from typing import NamedTuple as NT

-class Foo(NT):
+@dataclass(frozen=True)
+class Foo:
     pass
```

3:

```
import typing as typ

class Foo(typ.NamedTuple):
    pass
```

Autofix:

```
---
+++
@@ -1,5 +1,6 @@

 import typing as typ

-class Foo(typ.NamedTuple):
+@dataclass(frozen=True)
+class Foo:
     pass
```

4:

```
from typing import NamedTuple

class Foo(NamedTuple, AnotherBase, YetAnotherBase):
    pass
```

Autofix:

```
---
+++
@@ -1,5 +1,6 @@

 from typing import NamedTuple

-class Foo(NamedTuple, AnotherBase, YetAnotherBase):
+@dataclass(frozen=True)
+class Foo(AnotherBase, YetAnotherBase):
     pass
```

5:

```
from typing import NamedTuple

class OuterClass(SomeBase):
    class InnerClass(NamedTuple):
        pass
```

Autofix:

```
---
+++
@@ -2,5 +2,6 @@
 from typing import NamedTuple

 class OuterClass(SomeBase):
-    class InnerClass(NamedTuple):
+    @dataclass(frozen=True)
+    class InnerClass:
         pass
```

6:

```
from typing import NamedTuple

@some_other_decorator
class Foo(NamedTuple):
    pass
```

Autofix:

```
---
+++
@@ -2,5 +2,6 @@
 from typing import NamedTuple

 @some_other_decorator
-class Foo(NamedTuple):
+@dataclass(frozen=True)
+class Foo:
     pass
```

CHAPTER

NINETEEN

NOREDUNDANTARGUMENTSSUPERRULE

Remove redundant arguments when using super for readability.

19.1 Message

Do not use arguments when calling super for the parent class. See https://www.python.org/dev/peps/pep-3135/

19.2 Has Autofix: Yes

19.3 VALID Code Examples

1:

```
class Foo(Bar):
    def foo(self, bar):
        super().foo(bar)
```

2:

```
class Foo(Bar):
    def foo(self, bar):
        super(Bar, self).foo(bar)
```

3:

```
class Foo(Bar):
    @classmethod
    def foo(cls, bar):
        super(Bar, cls).foo(bar)
```

4:

```
class Foo:
    class InnerBar(Bar):
        def foo(self, bar):
            pass

    class InnerFoo(InnerBar):
        def foo(self, bar):
            super(InnerBar, self).foo(bar)
```

19.4 INVALID Code Examples

1:

```
class Foo(Bar):
    def foo(self, bar):
        super(Foo, self).foo(bar)
```

Autofix:

```
---
+++
@@ -1,4 +1,4 @@

 class Foo(Bar):
     def foo(self, bar):
-        super(Foo, self).foo(bar)
+        super().foo(bar)
```

2:

```
class Foo(Bar):
    @classmethod
    def foo(cls, bar):
        super(Foo, cls).foo(bar)
```

Autofix:

```
---
+++
@@ -2,4 +2,4 @@
 class Foo(Bar):
     @classmethod
     def foo(cls, bar):
-        super(Foo, cls).foo(bar)
+        super().foo(bar)
```

3:

```
class Foo:
    class InnerFoo(Bar):
        def foo(self, bar):
            super(Foo.InnerFoo, self).foo(bar)
```

Autofix:

```
---
+++
@@ -2,4 +2,4 @@
 class Foo:
     class InnerFoo(Bar):
         def foo(self, bar):
-            super(Foo.InnerFoo, self).foo(bar)
+            super().foo(bar)
```

4:

```
class Foo:
    class InnerFoo(Bar):
        class InnerInnerFoo(Bar):
            def foo(self, bar):
                super(Foo.InnerFoo.InnerInnerFoo, self).foo(bar)
```

Autofix:

```
---
+++
@@ -3,4 +3,4 @@
     class InnerFoo(Bar):
         class InnerInnerFoo(Bar):
             def foo(self, bar):
-                super(Foo.InnerFoo.InnerInnerFoo, self).foo(bar)
+                super().foo(bar)
```

CHAPTER

TWENTY

NOREDUNDANTFSTRINGRULE

Remove redundant f-string without placeholders.

20.1 Message

f-string doesn't have placeholders, remove redundant f-string.

20.2 Has Autofix: Yes

20.3 VALID Code Examples

1:

```
good: str = "good"
```

2:

```
good: str = f"with_arg{arg}"
```

3:

```
good = "good{arg1}".format(1234)
```

4:

```
good = "good".format()
```

5:

```
good = "good" % {}
```

6:

```
good = "good" % ()
```

7:

```
good = rf"good  +{bar}"
```

20.4 INVALID Code Examples

1:

```
bad: str = f"bad" + "bad"
```

Autofix:

```
---
+++
@@ -1 +1 @@
-bad: str = f"bad" + "bad"
+bad: str = "bad" + "bad"
```

2:

```
bad: str = f'bad'
```

Autofix:

```
---
+++
@@ -1 +1 @@
-bad: str = f'bad'
+bad: str = 'bad'
```

3:

```
bad: str = rf'bad   +'
```

Autofix:

```
---
+++
@@ -1 +1 @@
-bad: str = rf'bad  +'
+bad: str = r'bad   +'
```

4:

```
bad: str = f"no args but messing up {{ braces }}"
```

Autofix:

```
---
+++
@@ -1 +1 @@
-bad: str = f"no args but messing up {{ braces }}"
+bad: str = "no args but messing up { braces }"
```

CHAPTER

TWENTYONE

NOREDUNDANTLAMBDARULE

A lamba function which has a single objective of passing all it is arguments to another callable can be safely replaced by that callable.

21.1 Has Autofix: Yes

21.2 VALID Code Examples

1:

```
lambda x: foo(y)
```

2:

```
lambda x: foo(x, y)
```

3:

```
lambda x, y: foo(x)
```

4:

```
lambda *, x: foo(x)
```

5:

```
lambda x = y: foo(x)
```

6:

```
lambda x, y: foo(y, x)
```

7:

```
lambda self: self.func()
```

8:

```
lambda x, y: foo(y=x, x=y)
```

9:

```
lambda x, y, *z: foo(x, y, z)
```

10:

```
lambda x, y, **z: foo(x, y, z)
```

21.3 INVALID Code Examples

1:

```
lambda: self.func()
```

Autofix:

```
---
+++
@@ -1 +1 @@
-lambda: self.func()
+self.func
```

2:

```
lambda x: foo(x)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-lambda x: foo(x)
+foo
```

3:

```
lambda x, y, z: (t + u).math_call(x, y, z)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-lambda x, y, z: (t + u).math_call(x, y, z)
+(t + u).math_call
```

CHAPTER

TWENTYTWO

NOREDUNDANTLISTCOMPREHENSIONRULE

A derivative of flake8-comprehensions's C407 rule.

22.1 Has Autofix: Yes

22.2 VALID Code Examples

1:

```
any(val for val in iterable)
```

2:

```
all(val for val in iterable)
```

3:

```
frozenset([val for val in iterable])
```

4:

```
max([val for val in iterable])
```

5:

```
min([val for val in iterable])
```

6:

```
sorted([val for val in iterable])
```

7:

```
sum([val for val in iterable])
```

8:

```
tuple([val for val in iterable])
```

22.3 INVALID Code Examples

1:

```
any([val for val in iterable])
```

Autofix:

```
---
+++
@@ -1 +1 @@
-any([val for val in iterable])
+any(val for val in iterable)
```

2:

```
all([val for val in iterable])
```

Autofix:

```
---
+++
@@ -1 +1 @@
-all([val for val in iterable])
+all(val for val in iterable)
```

CHAPTER

TWENTYTHREE

NOSTATICIFCONDITIONRULE

Discourages `if` conditions which evaluate to a static value (e.g. `or True`, `and False`, etc).

23.1 Message

Your if condition appears to evaluate to a static value (e.g. *or True*, *and False*). Please double check this logic and if it is actually temporary debug code.

23.2 Has Autofix: No

23.3 VALID Code Examples

1:

```
if my_func() or not else_func():
    pass
```

2:

```
if function_call(True):
    pass
```

3:

```
# ew who would this???
def true():
    return False
if true() and else_call():  # True or False
    pass
```

4:

```
# ew who would this???
if False or some_func():
    pass
```

23.4 INVALID Code Examples

1:

```
if True:
    do_something()
```

2:

```
if crazy_expression or True:
    do_something()
```

3:

```
if crazy_expression and False:
    do_something()
```

4:

```
if crazy_expression and not True:
    do_something()
```

5:

```
if crazy_expression or not False:
    do_something()
```

6:

```
if crazy_expression or (something() or True):
    do_something()
```

7:

```
if crazy_expression and (something() and (not True)):
    do_something()
```

8:

```
if crazy_expression and (something() and (other_func() and not True)):
    do_something()
```

9:

```
if (crazy_expression and (something() and (not True))) or True:
    do_something()
```

10:

```
async def some_func() -> none:
    if (await expression()) and False:
        pass
```

CHAPTER

TWENTYFOUR

NOSTRINGTYPEANNOTATIONRULE

Enforce the use of type identifier instead of using string type hints for simplicity and better syntax highlighting. Starting in Python 3.7, `from __future__ import annotations` can postpone evaluation of type annotations PEP 563 and thus forward references no longer need to use string annotation style.

24.1 Message

String type hints are no longer necessary in Python, use the type identifier directly.

24.2 Has Autofix: Yes

24.3 VALID Code Examples

1:

```
from a.b import Class

def foo() -> Class:
    return Class()
```

2:

```
import typing
from a.b import Class

def foo() -> typing.Type[Class]:
    return Class
```

3:

```
import typing
from a.b import Class
from c import func

def foo() -> typing.Optional[typing.Type[Class]]:
    return Class if func() else None
```

4:

```
from a.b import Class

def foo(arg: Class) -> None:
    pass

foo(Class())
```

5:

```
from a.b import Class

module_var: Class = Class()
```

6:

```
from typing import Literal

def foo() -> Literal["a", "b"]:
    return "a"
```

7:

```
import typing

def foo() -> typing.Optional[typing.Literal["a", "b"]]:
    return "a"
```

24.4 INVALID Code Examples

1:

```
from __future__ import annotations

from a.b import Class

def foo() -> "Class":
    return Class()
```

Autofix:

```
---
+++
@@ -3,5 +3,5 @@

 from a.b import Class

-def foo() -> "Class":
+def foo() -> Class:
     return Class()
```

2:

```
from __future__ import annotations
```

(continues on next page)

(continued from previous page)

```
from a.b import Class

async def foo() -> "Class":
    return await Class()
```

Autofix:

```
---
+++
@@ -3,5 +3,5 @@

 from a.b import Class

-async def foo() -> "Class":
+async def foo() -> Class:
     return await Class()
```

3:

```
from __future__ import annotations

import typing
from a.b import Class

def foo() -> typing.Type["Class"]:
    return Class
```

Autofix:

```
---
+++
@@ -4,5 +4,5 @@
 import typing
 from a.b import Class

-def foo() -> typing.Type["Class"]:
+def foo() -> typing.Type[Class]:
     return Class
```

4:

```
from __future__ import annotations

import typing
from a.b import Class
from c import func

def foo() -> Optional[typing.Type["Class"]]:
    return Class if func() else None
```

Autofix:

```
---
+++
@@ -5,5 +5,5 @@
 from a.b import Class
```

(continues on next page)

(continued from previous page)

```
 from c import func

-def foo() -> Optional[typing.Type["Class"]]:
+def foo() -> Optional[typing.Type[Class]]:
     return Class if func() else None
```

5:

```
from __future__ import annotations

from a.b import Class

def foo(arg: "Class") -> None:
    pass

foo(Class())
```

Autofix:

```
---
+++
@@ -3,7 +3,7 @@

 from a.b import Class

-def foo(arg: "Class") -> None:
+def foo(arg: Class) -> None:
     pass

 foo(Class())
```

6:

```
from __future__ import annotations

from a.b import Class

module_var: "Class" = Class()
```

Autofix:

```
---
+++
@@ -3,4 +3,4 @@

 from a.b import Class

-module_var: "Class" = Class()
+module_var: Class = Class()
```

7:

```
from __future__ import annotations

import typing
from typing_extensions import Literal
```

(continues on next page)

(continued from previous page)

```
from a.b import Class

def foo() -> typing.Tuple[Literal["a", "b"], "Class"]:
    return Class()
```

Autofix:

```
---
+++
@@ -5,5 +5,5 @@
 from typing_extensions import Literal
 from a.b import Class

-def foo() -> typing.Tuple[Literal["a", "b"], "Class"]:
+def foo() -> typing.Tuple[Literal["a", "b"], Class]:
     return Class()
```

CHAPTER

TWENTYFIVE

REPLACEUNIONWITHOPTIONALRULE

Enforces the use of `Optional[T]` over `Union[T, None]` and `Union[None, T]`. See https://docs.python.org/3/library/typing.html#typing.Optional to learn more about Optionals.

25.1 Message

Optional[T] is preferred over *Union[T, None]* or *Union[None, T]*. Learn more: https://docs.python.org/3/library/typing.html#typing.Optional

25.2 Has Autofix: Yes

25.3 VALID Code Examples

1:

```
def func() -> Optional[str]:
    pass
```

2:

```
def func() -> Optional[Dict]:
    pass
```

3:

```
def func() -> Union[str, int, None]:
    pass
```

25.4 INVALID Code Examples

1:

```
def func() -> Union[str, None]:
    pass
```

2:

```
from typing import Optional
def func() -> Union[Dict[str, int], None]:
    pass
```

Autofix:

```
---
+++
@@ -1,4 +1,4 @@

 from typing import Optional
-def func() -> Union[Dict[str, int], None]:
+def func() -> Optional[Dict[str, int]]:
     pass
```

3:

```
from typing import Optional
def func() -> Union[str, None]:
    pass
```

Autofix:

```
---
+++
@@ -1,4 +1,4 @@

 from typing import Optional
-def func() -> Union[str, None]:
+def func() -> Optional[str]:
     pass
```

4:

```
from typing import Optional
def func() -> Union[Dict, None]:
    pass
```

Autofix:

```
---
+++
@@ -1,4 +1,4 @@

 from typing import Optional
-def func() -> Union[Dict, None]:
+def func() -> Optional[Dict]:
     pass
```

CHAPTER

TWENTYSIX

REWRITETOCOMPREHENSIONRULE

A derivative of flake8-comprehensions's C400-C402 and C403-C404. Comprehensions are more efficient than functions calls. This C400-C402 suggest to use *dict/set/list* comprehensions rather than respective function calls whenever possible. C403-C404 suggest to remove unnecessary list comprehension in a set/dict call, and replace it with set/dict comprehension.

26.1 Has Autofix: Yes

26.2 VALID Code Examples

1:

```
[val for val in iterable]
```

2:

```
{val for val in iterable}
```

3:

```
{val: val+1 for val in iterable}
```

4:

```
dict(line.strip().split('=', 1) for line in attr_file)
```

26.3 INVALID Code Examples

1:

```
list(val for val in iterable)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-list(val for val in iterable)
+[val for val in iterable]
```

2:

```
list(val for row in matrix for val in row)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-list(val for row in matrix for val in row)
+[val for row in matrix for val in row]
```

3:

```
set(val for val in iterable)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-set(val for val in iterable)
+{val for val in iterable}
```

4:

```
dict((x, f(x)) for val in iterable)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-dict((x, f(x)) for val in iterable)
+{x: f(x) for val in iterable}
```

5:

```
dict((x, y) for y, x in iterable)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-dict((x, y) for y, x in iterable)
+{x: y for y, x in iterable}
```

6:

```
dict([val, val+1] for val in iterable)
```

Autofix:

```
---
+++
@@ -1 +1 @@
```

(continues on next page)

(continued from previous page)

```
-dict([val, val+1] for val in iterable)
+{val: val+1 for val in iterable}
```

7:

```
dict((x["name"], json.loads(x["data"])) for x in responses)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-dict((x["name"], json.loads(x["data"])) for x in responses)
+{x["name"]: json.loads(x["data"]) for x in responses}
```

8:

```
dict((k, v) for k, v in iter for iter in iters)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-dict((k, v) for k, v in iter for iter in iters)
+{k: v for k, v in iter for iter in iters}
```

9:

```
set([val for val in iterable])
```

Autofix:

```
---
+++
@@ -1 +1 @@
-set([val for val in iterable])
+{val for val in iterable}
```

10:

```
dict([[val, val+1] for val in iterable])
```

Autofix:

```
---
+++
@@ -1 +1 @@
-dict([[val, val+1] for val in iterable])
+{val: val+1 for val in iterable}
```

11:

```
dict([(x, f(x)) for x in foo])
```

Autofix:

```
---
+++
@@ -1 +1 @@
-dict([(x, f(x)) for x in foo])
+{x: f(x) for x in foo}
```

12:

```
dict([(x, y) for y, x in iterable])
```

Autofix:

```
---
+++
@@ -1 +1 @@
-dict([(x, y) for y, x in iterable])
+{x: y for y, x in iterable}
```

13:

```
set([val for row in matrix for val in row])
```

Autofix:

```
---
+++
@@ -1 +1 @@
-set([val for row in matrix for val in row])
+{val for row in matrix for val in row}
```

CHAPTER

TWENTYSEVEN

REWRITETOLITERALRULE

A derivative of flake8-comprehensions' C405-C406 and C409-C410. It's unnecessary to use a list or tuple literal within a call to tuple, list, set, or dict since there is literal syntax for these types.

27.1 Has Autofix: Yes

27.2 VALID Code Examples

1:

```
(1, 2)
```

2:

```
()
```

3:

```
[1, 2]
```

4:

```
[]
```

5:

```
{1, 2}
```

6:

```
set()
```

7:

```
{1: 2, 3: 4}
```

8:

```
{}
```

27.3 INVALID Code Examples

1:

```
tuple([1, 2])
```

Autofix:

```
---
+++
@@ -1 +1 @@
-tuple([1, 2])
+(1, 2)
```

2:

```
tuple((1, 2))
```

Autofix:

```
---
+++
@@ -1 +1 @@
-tuple((1, 2))
+(1, 2)
```

3:

```
tuple([])
```

Autofix:

```
---
+++
@@ -1 +1 @@
-tuple([])
+()
```

4:

```
list([1, 2, 3])
```

Autofix:

```
---
+++
@@ -1 +1 @@
-list([1, 2, 3])
+[1, 2, 3]
```

5:

```
list((1, 2, 3))
```

Autofix:

```
---
+++
@@ -1 +1 @@
-list((1, 2, 3))
+[1, 2, 3]
```

6:

```
list([])
```

Autofix:

```
---
+++
@@ -1 +1 @@
-list([])
+[]
```

7:

```
set([1, 2, 3])
```

Autofix:

```
---
+++
@@ -1 +1 @@
-set([1, 2, 3])
+{1, 2, 3}
```

8:

```
set((1, 2, 3))
```

Autofix:

```
---
+++
@@ -1 +1 @@
-set((1, 2, 3))
+{1, 2, 3}
```

9:

```
set([])
```

Autofix:

```
---
+++
@@ -1 +1 @@
-set([])
+set()
```

10:

```
dict([(1, 2), (3, 4)])
```

Autofix:

```
---
+++
@@ -1 +1 @@
-dict([(1, 2), (3, 4)])
+{1: 2, 3: 4}
```

11:

```
dict(((1, 2), (3, 4)))
```

Autofix:

```
---
+++
@@ -1 +1 @@
-dict(((1, 2), (3, 4)))
+{1: 2, 3: 4}
```

12:

```
dict([[1, 2], [3, 4], [5, 6]])
```

Autofix:

```
---
+++
@@ -1 +1 @@
-dict([[1, 2], [3, 4], [5, 6]])
+{1: 2, 3: 4, 5: 6}
```

13:

```
dict([])
```

Autofix:

```
---
+++
@@ -1 +1 @@
-dict([])
+{}
```

14:

```
tuple()
```

Autofix:

```
---
+++
@@ -1 +1 @@
-tuple()
+()
```

15:

```
list()
```

Autofix:

```
---
+++
@@ -1 +1 @@
-list()
+[]
```

16:

```
dict()
```

Autofix:

```
---
+++
@@ -1 +1 @@
-dict()
+{}
```

CHAPTER

TWENTYEIGHT

SORTEDATTRIBUTESRULE

Ever wanted to sort a bunch of class attributes alphabetically? Well now it's easy! Just add "@sorted-attributes" in the doc string of a class definition and lint will automatically sort all attributes alphabetically.

Feel free to add other methods and such – it should only affect class attributes.

28.1 Message

It appears you are using the @sorted-attributes directive and the class variables are unsorted. See the lint autofix suggestion.

28.2 Has Autofix: Yes

28.3 VALID Code Examples

1:

```
class MyConstants:
    """
    @sorted-attributes
    """
    A = 'zzz123'
    B = 'aaa234'

class MyUnsortedConstants:
    B = 'aaa234'
    A = 'zzz123'
```

28.4 INVALID Code Examples

1:

```
class MyUnsortedConstants:
    """
    @sorted-attributes
    """
    z = "hehehe"
```

(continues on next page)

(continued from previous page)

```
    B = 'aaa234'
    A = 'zzz123'
    cab = "foo bar"
    Daaa = "banana"

    @classmethod
    def get_foo(cls) -> str:
        return "some random thing"
```

Autofix:

```
---
+++
@@ -3,11 +3,11 @@
     """
     @sorted-attributes
     """
+    A = 'zzz123'
+    B = 'aaa234'
+    Daaa = "banana"
+    cab = "foo bar"
     z = "hehehe"
-    B = 'aaa234'
-    A = 'zzz123'
-    cab = "foo bar"
-    Daaa = "banana"

     @classmethod
     def get_foo(cls) -> str:
```

CHAPTER

TWENTYNINE

USEASSERTINRULE

Discourages use of `assertTrue(x in y)` and `assertFalse(x in y)` as it is deprecated (https://docs.python.org/3.8/library/unittest.html#deprecated-aliases). Use `assertIn(x, y)` and `assertNotIn(x, y))` instead.

29.1 Message

Use assertIn/assertNotIn instead of assertTrue/assertFalse for inclusion check. See https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertIn)

29.2 Has Autofix: Yes

29.3 VALID Code Examples

1:

```
self.assertIn(a, b)
```

2:

```
self.assertIn(f(), b)
```

3:

```
self.assertIn(f(x), b)
```

4:

```
self.assertIn(f(g(x)), b)
```

5:

```
self.assertNotIn(a, b)
```

6:

```
self.assertNotIn(f(), b)
```

7:

```
self.assertNotIn(f(x), b)
```

8:

```
self.assertNotIn(f(g(x)), b)
```

29.4 INVALID Code Examples

1:

```
self.assertTrue(a in b)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-self.assertTrue(a in b)
+self.assertIn(a, b)
```

2:

```
self.assertTrue(f() in b)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-self.assertTrue(f() in b)
+self.assertIn(f(), b)
```

3:

```
self.assertTrue(f(x) in b)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-self.assertTrue(f(x) in b)
+self.assertIn(f(x), b)
```

4:

```
self.assertTrue(f(g(x)) in b)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-self.assertTrue(f(g(x)) in b)
+self.assertIn(f(g(x)), b)
```

5:

```
self.assertTrue(a not in b)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-self.assertTrue(a not in b)
+self.assertNotIn(a, b)
```

6:

```
self.assertTrue(not a in b)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-self.assertTrue(not a in b)
+self.assertNotIn(a, b)
```

7:

```
self.assertFalse(a in b)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-self.assertFalse(a in b)
+self.assertNotIn(a, b)
```

CHAPTER

THIRTY

USEASSERTISNOTNONERULE

Discourages use of `assertTrue(x is not None)` and `assertFalse(x is not None)` as it is deprecated (https://docs.python.org/3.8/library/unittest.html#deprecated-aliases). Use `assertIsNotNone(x)` and `assertIsNone(x)`) instead.

30.1 Message

“assertTrue” and “assertFalse” are deprecated. Use “assertIsNotNone” and “assertIsNone” instead. See https://docs.python.org/3.8/library/unittest.html#deprecated-aliases

30.2 Has Autofix: Yes

30.3 VALID Code Examples

1:

```
self.assertIsNotNone(x)
```

2:

```
self.assertIsNone(x)
```

3:

```
self.assertIsNone(None)
```

4:

```
self.assertIsNotNone(f(x))
```

5:

```
self.assertIsNone(f(x))
```

6:

```
self.assertIsNone(object.key)
```

7:

```
self.assertIsNotNone(object.key)
```

30.4 INVALID Code Examples

1:

```
self.assertTrue(a is not None)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-self.assertTrue(a is not None)
+self.assertIsNotNone(a)
```

2:

```
self.assertTrue(not x is None)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-self.assertTrue(not x is None)
+self.assertIsNotNone(x)
```

3:

```
self.assertTrue(f() is not None)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-self.assertTrue(f() is not None)
+self.assertIsNotNone(f())
```

4:

```
self.assertTrue(not x is not None)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-self.assertTrue(not x is not None)
+self.assertIsNone(x)
```

5:

```
self.assertTrue(f(x) is not None)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-self.assertTrue(f(x) is not None)
+self.assertIsNotNone(f(x))
```

6:

```
self.assertTrue(x is None)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-self.assertTrue(x is None)
+self.assertIsNone(x)
```

7:

```
self.assertFalse(x is not None)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-self.assertFalse(x is not None)
+self.assertIsNone(x)
```

8:

```
self.assertFalse(not x is None)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-self.assertFalse(not x is None)
+self.assertIsNone(x)
```

9:

```
self.assertFalse(f() is not None)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-self.assertFalse(f() is not None)
+self.assertIsNone(f())
```

10:

```
self.assertFalse(not x is not None)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-self.assertFalse(not x is not None)
+self.assertIsNotNone(x)
```

11:

```
self.assertFalse(f(x) is not None)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-self.assertFalse(f(x) is not None)
+self.assertIsNone(f(x))
```

12:

```
self.assertFalse(x is None)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-self.assertFalse(x is None)
+self.assertIsNotNone(x)
```

CHAPTER

THIRTYONE

USECLASSNAMEASCODERULE

Meta lint rule which checks that codes of lint rules are migrated to new format in lint rule class definitions.

31.1 Message

IG-series codes are deprecated. Use class name as code instead.

31.2 Has Autofix: Yes

31.3 VALID Code Examples

1:

```
MESSAGE = "This is a message"
```

2:

```
from fixit.common.base import CstLintRule
class FakeRule(CstLintRule):
    MESSAGE = "This is a message"
```

3:

```
from fixit.common.base import CstLintRule
class FakeRule(CstLintRule):
    INVALID = [
        Invalid(
            code=""
        )
    ]
```

31.4 INVALID Code Examples

1:

```
MESSAGE = "IG90000 Message"
```

Autofix:

```
---
+++
@@ -1,2 +1,2 @@

-MESSAGE = "IG90000 Message"
+MESSAGE = "Message"
```

2:

```
from fixit.common.base import CstLintRule
class FakeRule(CstLintRule):
    INVALID = [
        Invalid(
            code="",
            kind="IG000"
        )
    ]
```

Autofix:

```
---
+++
@@ -4,6 +4,5 @@
     INVALID = [
         Invalid(
             code="",
-            kind="IG000"
-        )
+            )
     ]
```

CHAPTER

THIRTYTWO

USECLSINCLASSMETHODRULE

Enforces using `cls` as the first argument in a `@classmethod`.

32.1 Message

When using @classmethod, the first argument must be *cls*.

32.2 Has Autofix: Yes

32.3 VALID Code Examples

1:

```
class foo:
    # classmethod with cls first arg.
    @classmethod
    def cm(cls, a, b, c):
        pass
```

2:

```
class foo:
    # non-classmethod with non-cls first arg.
    def nm(self, a, b, c):
        pass
```

3:

```
class foo:
    # staticmethod with non-cls first arg.
    @staticmethod
    def sm(a):
        pass
```

32.4 INVALID Code Examples

1:

```
class foo:
    # No args at all.
    @classmethod
    def cm():
        pass
```

Autofix:

```
---
+++
@@ -2,5 +2,5 @@
 class foo:
     # No args at all.
     @classmethod
-    def cm():
+    def cm(cls):
         pass
```

2:

```
class foo:
    # Single arg + reference.
    @classmethod
    def cm(a):
        return a
```

Autofix:

```
---
+++
@@ -2,5 +2,5 @@
 class foo:
     # Single arg + reference.
     @classmethod
-    def cm(a):
-        return a
+    def cm(cls):
+        return cls
```

3:

```
class foo:
    # Another "cls" exists: do not autofix.
    @classmethod
    def cm(a):
        cls = 2
```

4:

```
class foo:
    # Multiple args + references.
    @classmethod
    async def cm(a, b):
        b = a
        b = a.__name__
```

Autofix:

```
---
+++
@@ -2,6 +2,6 @@
 class foo:
     # Multiple args + references.
     @classmethod
-    async def cm(a, b):
-        b = a
-        b = a.__name__
+    async def cm(cls, b):
+        b = cls
+        b = cls.__name__
```

5:

```
class foo:
    # Do not replace in nested scopes.
    @classmethod
    async def cm(a, b):
        b = a
        b = lambda _: a.__name__
        def g():
            return a.__name__

        # Same-named vars in sub-scopes should not be replaced.
        b = [a for a in [1,2,3]]
        def f(a):
            return a + 1
```

Autofix:

```
---
+++
@@ -2,11 +2,11 @@
 class foo:
     # Do not replace in nested scopes.
     @classmethod
-    async def cm(a, b):
-        b = a
-        b = lambda _: a.__name__
+    async def cm(cls, b):
+        b = cls
+        b = lambda _: cls.__name__
         def g():
-            return a.__name__
+            return cls.__name__

         # Same-named vars in sub-scopes should not be replaced.
         b = [a for a in [1,2,3]]
```

6:

```
# Do not replace in surrounding scopes.
a = 1

class foo:
    a = 2
```

(continues on next page)

(continued from previous page)

```
    def im(a):
        a = a

    @classmethod
    def cm(a):
        a[1] = foo.cm(a=a)
```

Autofix:

```
---
+++
@@ -9,5 +9,5 @@
         a = a

     @classmethod
-    def cm(a):
-        a[1] = foo.cm(a=a)
+    def cm(cls):
+        cls[1] = foo.cm(a=cls)
```

7:

```
def another_decorator(x): pass

class foo:
    # Multiple decorators.
    @another_decorator
    @classmethod
    @another_decorator
    async def cm(a, b, c):
        pass
```

Autofix:

```
---
+++
@@ -6,5 +6,5 @@
     @another_decorator
     @classmethod
     @another_decorator
-    async def cm(a, b, c):
+    async def cm(cls, b, c):
         pass
```

CHAPTER

THIRTYTHREE

USEFSTRINGRULE

Encourages the use of f-string instead of %-formatting or .format() for high code quality and efficiency.

Following two cases not covered:

1. **arguments length greater than 30 characters: for better readibility reason** For example:

   1: this is the answer: %d" % (a_long_function_call() + b_another_long_function_call()) 2: f"this is the answer: {a_long_function_call() + b_another_long_function_call()}" 3: result = a_long_function_call() + b_another_long_function_call() f"this is the answer: {result}"

   Line 1 is more readable than line 2. Ideally, we'd like developers to manually fix this case to line 3

2. **only %s placeholders are linted against for now. We leave it as future work to support other placeholders.** For example, %d raises TypeError for non-numeric objects, whereas f"{x:d}" raises ValueError. This discrepancy in the type of exception raised could potentially break the logic in the code where the exception is handled

33.1 Message

Do not use printf style formatting or .format(). Use f-string instead to be more readable and efficient. See https://www.python.org/dev/peps/pep-0498/

33.2 Has Autofix: Yes

33.3 VALID Code Examples

1:

```
somebody='you'; f"Hey, {somebody}."
```

2:

```
"hey"
```

3:

```
"hey" + "there"
```

4:

```
b"a type %s" % var
```

33.4 INVALID Code Examples

1:

```
"Hey, {somebody}.".format(somebody="you")
```

2:

```
"%s" % "hi"
```

Autofix:

```
---
+++
@@ -1 +1 @@
-"%s" % "hi"
+f"{'hi'}"
```

3:

```
"a name: %s" % name
```

Autofix:

```
---
+++
@@ -1 +1 @@
-"a name: %s" % name
+f"a name: {name}"
```

4:

```
"an attribute %s ." % obj.attr
```

Autofix:

```
---
+++
@@ -1 +1 @@
-"an attribute %s ." % obj.attr
+f"an attribute {obj.attr} ."
```

5:

```
r"raw string value=%s" % val
```

Autofix:

```
---
+++
@@ -1 +1 @@
-r"raw string value=%s" % val
+fr"raw string value={val}"
```

6:

```
"{%s}" % val
```

Autofix:

```
---
+++
@@ -1 +1 @@
-"{%s}" % val
+f"{{{val}}}"
```

7:

```
"{%s" % val
```

Autofix:

```
---
+++
@@ -1 +1 @@
-"{%s" % val
+f"{{{val}"
```

8:

```
"The type of var: %s" % type(var)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-"The type of var: %s" % type(var)
+f"The type of var: {type(var)}"
```

9:

```
"%s" % obj.this_is_a_very_long_expression(parameter)["a_very_long_key"]
```

10:

```
"type of var: %s, value of var: %s" % (type(var), var)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-"type of var: %s, value of var: %s" % (type(var), var)
+f"type of var: {type(var)}, value of var: {var}"
```

11:

```
'%s" double quote is used' % var
```

Autofix:

```
---
+++
@@ -1 +1 @@
-'%s" double quote is used' % var
+f'{var}" double quote is used'
```

12:

```
"var1: %s, var2: %s, var3: %s, var4: %s" % (class_object.attribute, dict_lookup["some_
↪key"], some_module.some_function(), var4)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-"var1: %s, var2: %s, var3: %s, var4: %s" % (class_object.attribute, dict_lookup[
↪"some_key"], some_module.some_function(), var4)
+f"var1: {class_object.attribute}, var2: {dict_lookup['some_key']}, var3: {some_
↪module.some_function()}, var4: {var4}"
```

13:

```
"a list: %s" % " ".join(var)
```

Autofix:

```
---
+++
@@ -1 +1 @@
-"a list: %s" % " ".join(var)
+f"a list: {' '.join(var)}"
```

CHAPTER

THIRTYFOUR

USEISNONEONOPTIONALRULE

Enforces explicit use of `is None` or `is not None` when checking whether an Optional has a value. Directly testing an object (e.g. `if x`) implicitely tests for a truth value which returns `True` unless the object's `__bool__()` method returns False, its `__len__()` method returns '0', or it is one of the constants `None` or `False`. (https://docs.python.org/3.8/library/stdtypes.html#truth-value-testing).

34.1 Message

When checking if an *Optional* has a value, avoid using it as a boolean since it implicitly checks the object's *__bool__()*, *__len__()* is not *0*, or the value is not *None*. Instead, use *is None* or *is not None* to be explicit.

34.2 Has Autofix: Yes

34.3 VALID Code Examples

1:

```
from typing import Optional
a: Optional[str]
if a is not None:
    pass
```

2:

```
a: bool
if a:
    pass
```

34.4 INVALID Code Examples

1:

```
from typing import Optional

a: Optional[str] = None
if a:
    pass
```

Autofix:

```
---
+++
@@ -2,5 +2,5 @@
 from typing import Optional

 a: Optional[str] = None
-if a:
+if a is not None:
     pass
```

2:

```
from typing import Optional
a: Optional[str] = None
x: bool = False
if x and a:
    ...
```

Autofix:

```
---
+++
@@ -2,5 +2,5 @@
 from typing import Optional
 a: Optional[str] = None
 x: bool = False
-if x and a:
+if x and a is not None:
     ...
```

3:

```
from typing import Optional
a: Optional[str] = None
x: bool = False
if a and x:
    ...
```

Autofix:

```
---
+++
@@ -2,5 +2,5 @@
 from typing import Optional
 a: Optional[str] = None
 x: bool = False
-if a and x:
+if a is not None and x:
     ...
```

4:

```
from typing import Optional
a: Optional[str] = None
x: bool = not a
```

Autofix:

```
---
+++
@@ -1,4 +1,4 @@

 from typing import Optional
 a: Optional[str] = None
-x: bool = not a
+x: bool = a is None
```

5:

```
from typing import Optional
a: Optional[str]
x: bool
if x or a: pass
```

Autofix:

```
---
+++
@@ -2,4 +2,4 @@
 from typing import Optional
 a: Optional[str]
 x: bool
-if x or a: pass
+if x or a is not None: pass
```

6:

```
from typing import Optional
a: Optional[str]
x: bool
if x: pass
elif a: pass
```

Autofix:

```
---
+++
@@ -3,4 +3,4 @@
 a: Optional[str]
 x: bool
 if x: pass
-elif a: pass
+elif a is not None: pass
```

7:

```
from typing import Optional
a: Optional[str] = None
b: Optional[str] = None
if a: pass
elif b: pass
```

Autofix:

```
---
+++
@@ -2,5 +2,5 @@
 from typing import Optional
 a: Optional[str] = None
 b: Optional[str] = None
-if a: pass
-elif b: pass
+if a is not None: pass
+elif b is not None: pass
```

CHAPTER

THIRTYFIVE

USELINTFIXMECOMMENTRULE

To silence a lint warning, use `lint-fixme` (when plans to fix the issue later) or `lint-ignore` (when the lint warning is not valid) comments. The comment requires to be in a standalone comment line and follows the format `lint-fixme: RULE_NAMES EXTRA_COMMENTS`. It suppresses the lint warning with the RULE_NAMES in the next line. RULE_NAMES can be one or more lint rule names separated by comma. `noqa` is deprecated and not supported because explicitly providing lint rule names to be suppressed in lint-fixme comment is preferred over implicit noqa comments. Implicit noqa suppression comments sometimes accidentally silence warnings unexpectedly.

35.1 Message

noqa is deprecated. Use *lint-fixme* or *lint-ignore* instead.

35.2 Has Autofix: No

35.3 VALID Code Examples

1:

```
# lint-fixme: UseFstringRule
"%s" % "hi"
```

2:

```
# lint-ignore: UsePlusForStringConcatRule
'ab' 'cd'
```

35.4 INVALID Code Examples

1:

```
fn() # noqa
```

2:

```
(
 1,
 2,  # noqa
)
```

3:

```
class C:
    # noqa
    ...
```

CHAPTER

THIRTYSIX

USEPLUSFORSTRINGCONCATRULE

Enforces use of explicit string concatenations using a + sign where an implicit concatenation is detected. An implicit concatenation is a tuple or a call with multiple strings and a missing comma, e.g: `("a" "b")`, and may have unexpected results.

36.1 Message

Implicit string concatenation detected, please add '+' to be explicit. E.g. a tuple or a call ("a" "b") with a missing comma results in multiple strings being concatenated as one string and causes unexpected behaviour.

36.2 Has Autofix: Yes

36.3 VALID Code Examples

1:

```
'abc'
```

2:

```
'abc' + 'def'
```

3:

```
f'abc'
```

36.4 INVALID Code Examples

1:

```
'ab' 'cd'
```

Autofix:

```
---
+++
@@ -1 +1 @@
-'ab' 'cd'
+('ab' + 'cd')
```

2:

```
'ab' 'cd' 'ef' 'gh'
```

Autofix:

```
---
+++
@@ -1 +1 @@
-'ab' 'cd' 'ef' 'gh'
+('ab' + 'cd' + 'ef' + 'gh')
```

3:

```
f'ab' f'cd'
```

Autofix:

```
---
+++
@@ -1 +1 @@
-f'ab' f'cd'
+(f'ab' + f'cd')
```

4:

```
(
    # comment
    'ab'  # middle comment
    'cd'  # trailing comment
)
```

Autofix:

```
---
+++
@@ -2,5 +2,5 @@
 (
     # comment
     'ab'  # middle comment
-    'cd'  # trailing comment
+    + 'cd'  # trailing comment
 )
```

CHAPTER

THIRTYSEVEN

USETYPESFROMTYPINGRULE

Enforces the use of types from the `typing` module in type annotations in place of `builtins.{builtin_type}` since the type system doesn't recognize the latter as a valid type.

37.1 Has Autofix: Yes

37.2 VALID Code Examples

1:

```
def fuction(list: List[str]) -> None:
    pass
```

2:

```
def function() -> None:
    thing: Dict[str, str] = {}
```

3:

```
def function() -> None:
    thing: Tuple[str]
```

4:

```
from typing import Dict, List
def function() -> bool:
        return Dict == List
```

5:

```
from typing import List as list
from graphene import List

def function(a: list[int]) -> List[int]:
        return []
```

37.3 INVALID Code Examples

1:

```
from typing import List
def whatever(list: list[str]) -> None:
    pass
```

Autofix:

```
---
+++
@@ -1,4 +1,4 @@

 from typing import List
-def whatever(list: list[str]) -> None:
+def whatever(list: List[str]) -> None:
     pass
```

2:

```
def function(list: list[str]) -> None:
    pass
```

3:

```
def func() -> None:
    thing: dict[str, str] = {}
```

4:

```
def func() -> None:
    thing: tuple[str]
```

5:

```
from typing import Dict
def func() -> None:
    thing: dict[str, str] = {}
```

Autofix:

```
---
+++
@@ -1,4 +1,4 @@

 from typing import Dict
 def func() -> None:
-    thing: dict[str, str] = {}
+    thing: Dict[str, str] = {}
```

CHAPTER

THIRTYEIGHT

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

THIRTYNINE

PRIVACY POLICY AND TERMS OF USE

INDEX