
Refurb

A tool for refurbishing and modernizing Python codebases.

Example

```
1 # main.py
2
3 for filename in ["file1.txt", "file2.txt"]:
4     with open(filename) as f:
5         contents = f.read()
6
7     lines = contents.splitlines()
8
9     for line in lines:
10        if not line or line.startswith("# ") or line.startswith("// "):
11            continue
12
13        for word in line.split():
14            print(f"[{word}]", end="")
15
16        print("")
```

Running:

```
1 $ refurb main.py
2 main.py:3:17 [FURB109]: Use `in (x, y, z)` instead of `in [x, y, z]`
3 main.py:4:5 [FURB101]: Use `y = Path(x).read_text()` instead of `with
  open(x, ...) as f: y = f.read()`
4 main.py:10:40 [FURB102]: Replace `x.startswith(y) or x.startswith(z)`
  with `x.startswith((y, z))`
5 main.py:16:9 [FURB105]: Use `print()` instead of `print("")`
```

Installing

```
1 $ pipx install refurb
2 $ refurb file.py folder/
```

Note Refurb must be run on Python 3.10+, though it can check Python 3.7+ code by setting the `--python-version` flag.

Explanations For Checks

You can use `refurb --explain FURB123`, where `FURB123` is the error code you are trying to look up. For example:

```
1 $ refurb --explain FURB123
2 Don't cast a variable or literal if it is already of that type. For
3 example:
4
5 Bad:
6
7 ```
8 name = str("bob")
9 num = int(123)
10 ```
11
12 Good:
13
14 ```
15 name = "bob"
16 num = 123
17 ```
```

An online list of all available checks can be viewed [here](#).

Ignoring Errors

Use `--ignore 123` to ignore error 123. The error code can be in the form `FURB123` or `123`. This flag can be repeated.

The `FURB` prefix indicates that this is a built-in error. The `FURB` prefix is optional, but for all other errors (ie, `ABC123`), the prefix is required.

You can also use inline comments to disable errors:

```
1 x = int(0) # noqa: FURB123
2 y = list() # noqa
```

Here, `noqa: FURB123` specifically ignores the `FURB123` error for that line, and `noqa` ignores all errors on that line.

You can also specify multiple errors to ignore by separating them with a comma/space:

```
1 x = not not int(0) # noqa: FURB114, FURB123
2 x = not not int(0) # noqa: FURB114 FURB123
```

Enabling/Disabling Checks

Certain checks are disabled by default, and need to be enabled first. You can do this using the `--enable ERR` flag, where `ERR` is the error code of the check you want to enable. A disabled check differs from an ignored check in that a disabled check will never be loaded, whereas an ignored check will be loaded, an error will be emitted, and the error will be suppressed.

Use the `--verbose/-v` flag to get a full list of enabled checks.

The opposite of `--enable` is `--disable`, which will disable a check. When `--enable` and `--disable` are both specified via the command line, whichever one comes last will take precedence. When using `enable` and `disable` via the config file, `disable` will always take precedence.

Use the `--disable-all` flag to disable all checks. This allows you to incrementally `--enable` checks as you see fit, as opposed to adding a bunch of `--ignore` flags. To use this in the config file, set `disable_all` to `true`.

Use the `--enable-all` flag to enable all checks by default. This allows you to opt into all checks that Refurb (and Refurb plugins) have to offer. This is a good option for new codebases. To use this in a config file, set `enable_all` to `true`.

In the config file, `disable_all/enable_all` is applied first, and then the `enable` and `disable` fields are applied afterwards.

Note that `disable_all` and `enable_all` are mutually exclusive, both on the command line and in the config file. You will get an error if you try to specify both.

You can also disable checks by category using the `#category` syntax. For example, `--disable "#readability"` will disable all checks with the `readability` category. The same applies for `enable` and `ignore`. Also, if you disable an entire category you can still explicitly re-enable a check in that category.

Note that `#readability` is wrapped in quotes because your shell will interpret the `#` as the start of a comment.

Setting Python Version

Use the `--python-version` flag to tell Refurb which version of Python your codebase is using. This should allow for better detection of language features, and allow for better error messages. The argument for this flag must be in the form `x.y`, for example, `3.10`.

The syntax for using this in the config file is `python_version = "3.10"`.

When the Python version is unspecified, Refurb uses whatever version your local Python installation uses. For example, if your `python --version` is `3.11.5`, Refurb uses `3.11`, dropping the 5 patch version.

Changing Output Formats

By default everything is outputted as plain text:

```
1 file.py:1:5 [FURB123]: Replace int(x) with x
```

Here are all of the available formats:

- `text`: The default
- `github`: Print output for use with GitHub Annotations
- More to come!

To change the default format use `--format XYZ` on the command line, or `format = "XYZ"` in the config file.

Changing Sort Order

By default errors are sorted by filename, then by error code. To change this, use the `--sort XYZ` flag on the command line, or `sort_by = "XYZ"` in the config file, where `XYZ` is one of the following sort modes:

- `filename`: Sort files in alphabetical order (the default)
- `error`: Sort by error first, then by filename

Overriding Mypy Flags

This is typically used for development purposes, but can also be used to better fine-tune Mypy from within Refurb. Any command line arguments after `--` are passed to Mypy. For example:

```
1 $ refurb files -- --show-traceback
```

This tells Mypy to show a traceback if it crashes.

You can also use this in the config file by assigning an array of values to the `mypy_args` field. Note that any Mypy arguments passed via the command line arguments will override the `mypy_args` field in the config file.

Configuring Refurb

In addition to the command line arguments, you can also add your settings in the `pyproject.toml` file. For example, the following command line arguments:

```
1 refurb file.py --ignore 100 --load some_module --quiet
```

Corresponds to the following in your `pyproject.toml` file:

```
1 [tool.refurb]
2 ignore = [100]
3 load = ["some_module"]
4 quiet = true
```

Now all you need to type is `refurb file.py`!

Note that the values in the config file will be merged with the values specified via the command line. In the case of boolean arguments like `--quiet`, the command line arguments take precedence. All other arguments (such as `ignore` and `load`) will be combined.

You can use the `--config-file` flag to tell Refurb to use a different config file from the default `pyproject.toml` file. Note that it still must be in the same form as the normal `pyproject.toml` file.

[Click here](#) to see some example config files.

Ignore Checks Per File/Folder

If you have a large codebase you might want to ignore errors for certain files or folders, which allows you to incrementally fix errors as you see fit. To do that, add the following to your `pyproject.toml` file:

```
1 # these settings will be applied globally
2 [tool.refurb]
3 enable_all = true
4
5 # these will only be applied to the "src" folder
6 [[tool.refurb.amend]]
7 path = "src"
8 ignore = ["FURB123", "FURB120"]
9
10 # these will only be applied to the "src/util.py" file
11 [[tool.refurb.amend]]
12 path = "src/util.py"
13 ignore = ["FURB125", "FURB148"]
```

Note that only the `ignore` field is available in the `amend` sections. This is because a check can only be enabled/disabled for the entire codebase, and cannot be selectively enabled/disabled on a per-file basis. Assuming a check is enabled though, you can simply `ignore` the errors for the files of your choosing.

Using Refurb With `pre-commit`

You can use Refurb with `pre-commit` by adding the following to your `.pre-commit-config.yaml` file:

```
1 - repo: https://github.com/dosisod/refurb
2   rev: REVISION
3   hooks:
4     - id: refurb
```

Replacing `REVISION` with a version or SHA of your choosing (or leave it blank to let `pre-commit` find the most recent one for you).

Plugins

Installing plugins for Refurb is very easy:

```
1 $ pip install refurb-plugin-example
```

Where `refurb-plugin-example` is the name of the plugin. Refurb will automatically load any installed plugins.

To make your own Refurb plugin, see the `refurb-plugin-example` repository for more info.

Writing Your Own Check

If you want to extend Refurb but don't want to make a full-fledged plugin, you can easily create a one-off check file with the `refurb gen` command.

Note that this command uses the `fzf` fuzzy-finder for getting user input, so you will need to install `fzf` before continuing.

Here is the basic overview for creating a new check using the `refurb gen` command:

1. First select the node type you want to accept
2. Then type in where you want to save the auto generated file

3. Add your code to the new file

To get an idea of what you need to add to your check, use the `--debug` flag to see the AST representation for a given file (ie, `refurb --debug file.py`). Take a look at the files in the `refurb/checks/` folder for some examples.

Then, to load your new check, use `refurb file.py --load your.path.here`

Note that when using `--load`, you need to use dots in your argument, just like importing a normal python module. If `your.path.here` is a directory, all checks in that directory will be loaded. If it is a file, only that file will be loaded.

Troubleshooting

If Refurb is running slow, use the `--timing-stats` flag to diagnose why:

```
1 $ refurb file --timing-stats /tmp/stats.json
```

This will output a JSON file with the following information:

- Total time Mypy took to parse the modules (a majority of the time usually).
- Time Mypy spent parsing each module. Useful for finding very large/unused files.
- Time Refurb spent checking each module. These numbers should be very small (less than 100ms).

Larger files naturally take longer to check, but files that take way too long should be looked into, as an issue might only manifest themselves when a file reaches a certain size.

Disable Color

Color output is enabled by default in Refurb. To disable it, do one of the following:

- Set the `NO_COLOR` env var.
- Use the `--no-color` flag.
- Set `color = false` in the config file.
- Pipe/redirect Refurb output to another program or file.

Developing / Contributing

Setup

To setup locally run:

```
1 $ git clone https://github.com/dosisod/refurb
2 $ cd refurb
3 $ make install
```

Tests can be ran all at once using `make`, or you can run each tool on its own using `make black`, `make flake8`, and so on.

Unit tests can be ran with `pytest` or `make test`.

Since the end-to-end (e2e) tests are slow, they are not ran when running `make`. You will need to run `make test-e2e` to run them.

Updating Documentation

We encourage people to update the documentation when they see typos and other issues!

With that in mind though, don't directly modify the `docs/checks.md` file. It is auto-generated and will be overridden when new checks are added. The documentation for checks can be updated by changing the docstrings of in the checks themselves. For example, to update `FURB100`, change the docstring of the `ErrorInfo` class in the `refurb/checks/pathlib/with_suffix.py` file. You can find the file for a given check by grep-ing for `code = XYZ`, where `XYZ` is the check you are looking for but with the `FURB` prefix removed.

Use the `--verbose` flag with `--explain` to find the filename for a given check. For example:

```
1 $ refurb --explain FURB123 --verbose
2 Filename: refurb/checks/readability/no_unnecessary_cast.py
3
4 FURB123: no-redundant-cast [readability]
5
6 ...
```

Why Does This Exist?

I love doing code reviews: I like taking something and making it better, faster, more elegant, and so on. Lots of static analysis tools already exist, but none of them seem to be focused on making code more elegant, more readable, or more modern. That is where Refurb comes in.

Refurb is heavily inspired by `clippy`, the built-in linter for Rust.

What Refurb Is Not

Refurb is not a style/type checker. It is not meant as a first-line of defense for linting and finding bugs, it is meant for making good code even better.

Comparison To Other Tools

There are already lots of tools out there for linting and analyzing Python code, so you might be wondering why Refurb exists (skepticism is good!). As mentioned above, Refurb checks for code which can be made more elegant, something that no other linters (that I have found) specialize in. Here is a list of similar linters and analyzers, and how they differ from Refurb:

Black: is more focused on the formatting and styling of the code (line length, trailing comas, indentation, and so on). It does a really good job of making other projects using Black look more or less the same. It doesn't do more complex things such as type checking or code smell/anti-pattern detection.

flake8: flake8 is also a linter, is very extensible, and performs a lot of semantic analysis-related checks as well, such as "unused variable", "break outside of a loop", and so on. It also checks PEP8 conformance. Refurb won't try and replace flake8, because chances are you are already using flake8 anyways.

Pylint has a lot of checks which cover a lot of ground, but in general, are focused on bad or buggy code, things which you probably didn't mean to do. Refurb assumes that you know what you are doing, and will try to cleanup what is already there the best it can.

Mypy, Pyright, Pyre, and Pytype are all type checkers, and basically just enforce types, ensures arguments match, functions are called in a type safe manner, and so on. They do much more than that, but that is the general idea. Refurb actually is built on top of Mypy, and uses its AST parser so that it gets good type information.

pyupgrade: Pyupgrade has a lot of good checks for upgrading your older Python code to the newer syntax, which is really useful. Where Refurb differs is that Pyupgrade is more focused on upgrading your code to the newer version, whereas Refurb is more focused on cleaning up and simplifying what is already there.

In conclusion, Refurb doesn't want you to throw out your old tools, since they cover different areas of your code, and all serve a different purpose. Refurb is meant to be used in conjunction with the above tools.