Code quality in the repository

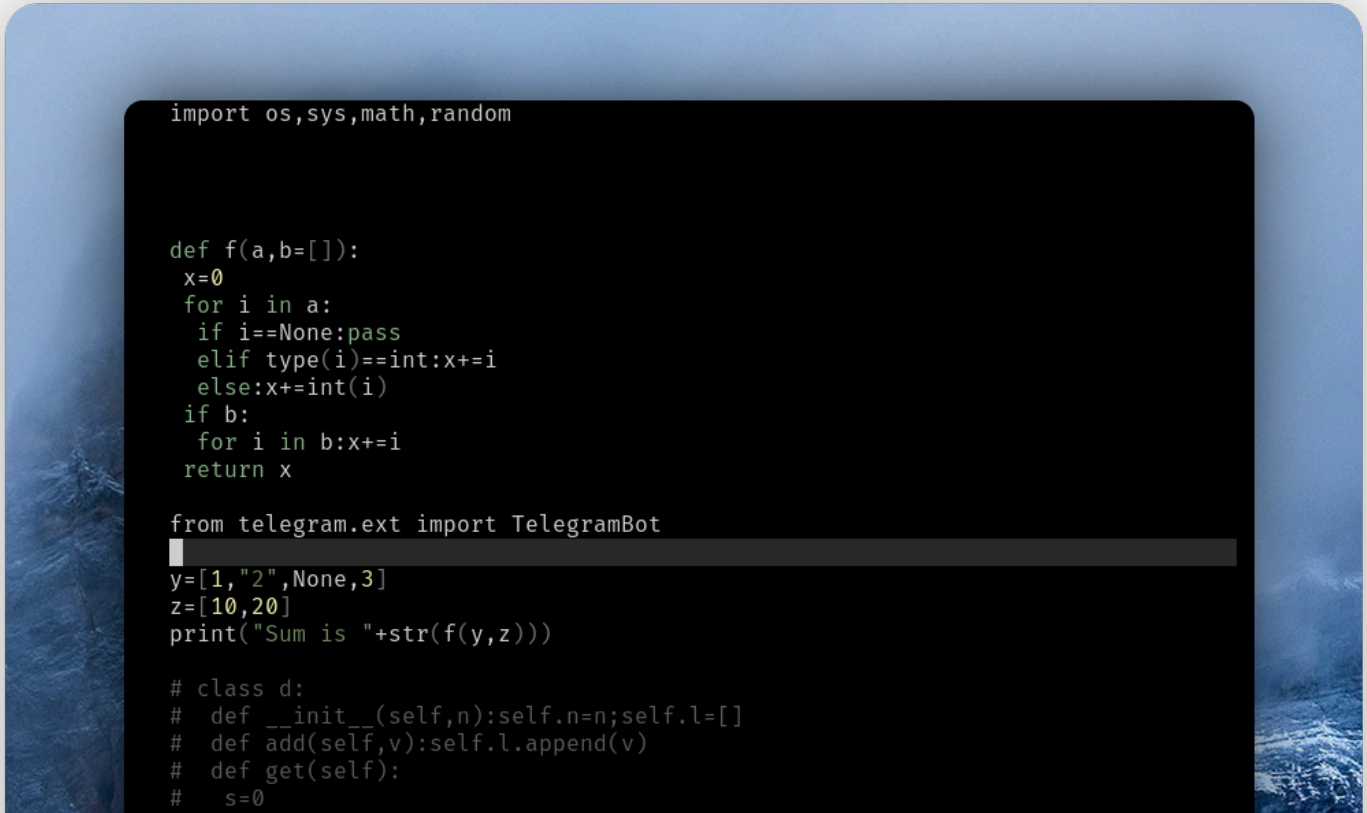
Anton Grishin (@alchemmist)

EOSP #L4, winter 2026. CU x CPM

Table of contents

`style`	How to control code style in repo? Formatters and syle-checkers.
`lint`	Set and follow some rules or best-practice in code base.
`types`	How to check consistency of types system.
`CI`	Automate all checks without installatino and configuring tools.
`make`	Entry point into repository. Useful tool.

What is style? style



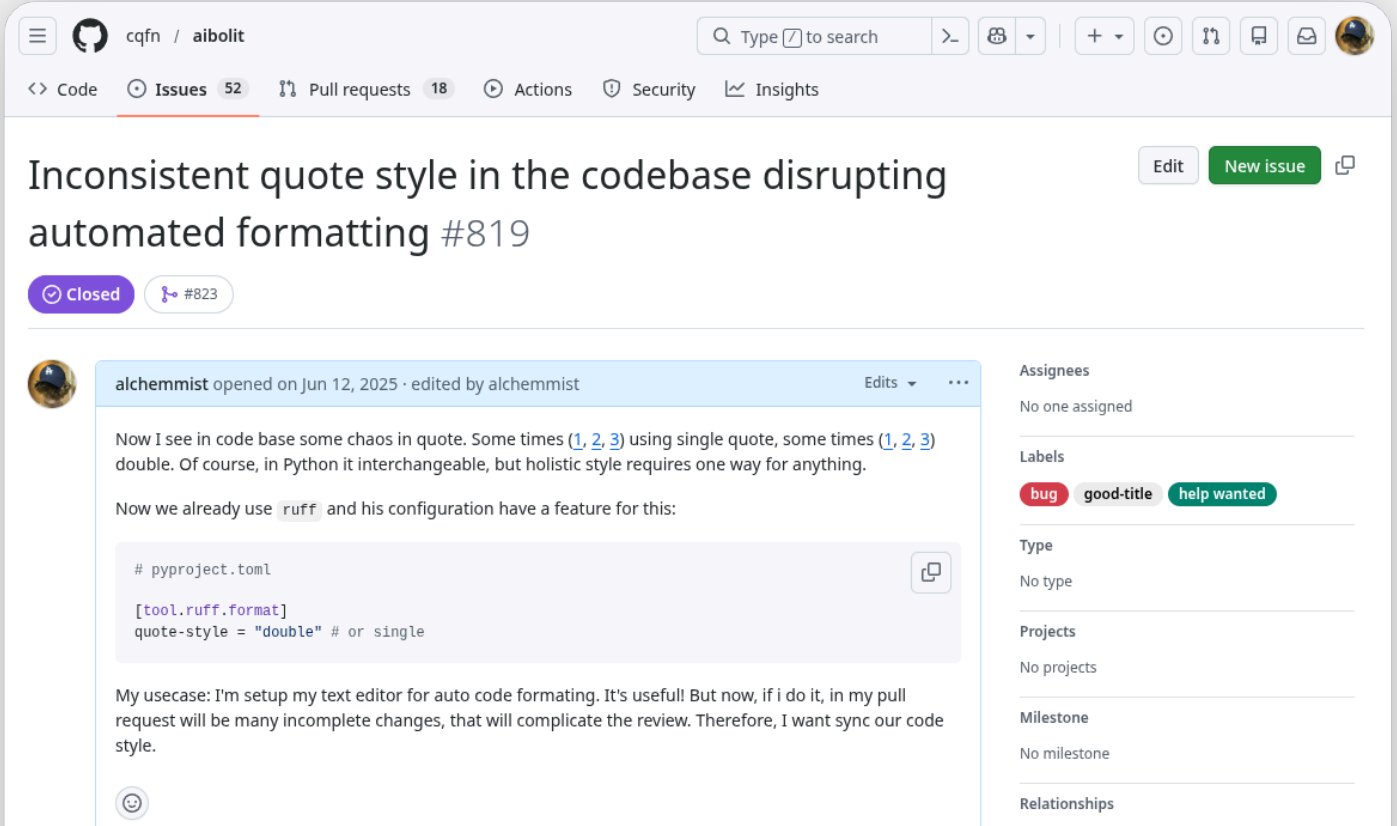
```
import os,sys,math,random



def f(a,b=[]):
 x=0
 for i in a:
  if i==None:pass
  elif type(i)==int:x+=i
  else:x+=int(i)
 if b:
  for i in b:x+=i
 return x

from telegram.ext import TelegramBot

y=[1,"2",None,3]
z=[10,20]
print("Sum is "+str(f(y,z)))

# class d:
#  def __init__(self,n):self.n=n;self.l=[]
#  def add(self,v):self.l.append(v)
#  def get(self):
#   s=0
```

Style it's about code formatting `style`

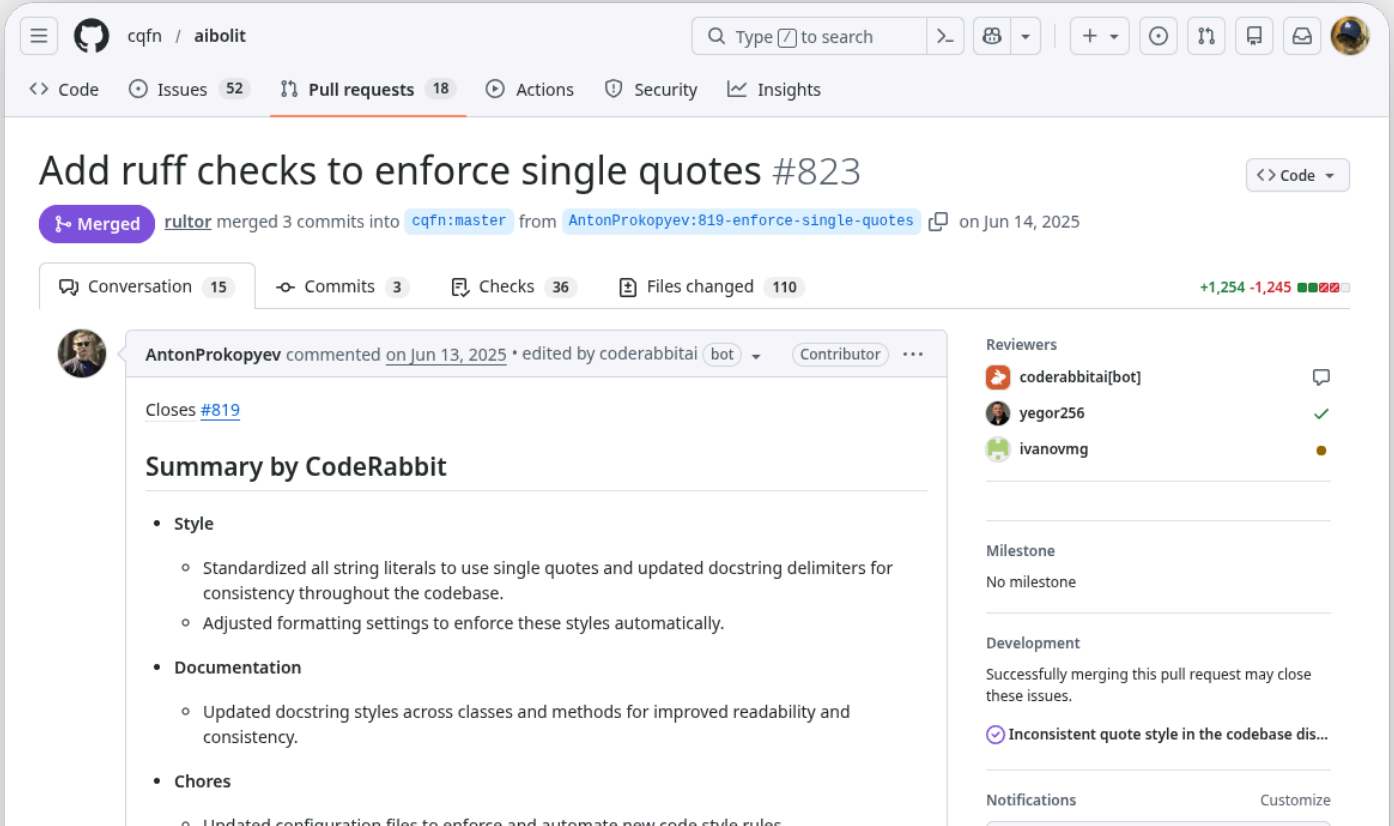
- The set of rules, that needs to be follow.
- For example: len of indent: tab, 4 space, 2 space, etc.
- Maximum line length (wrap long lines).
- Spaces around operators and after commas.
- Blank lines between functions/classes for readability.
- Order of imports (standard, third-party, local).
- The type of quotes (sigle or double).

cqfn/aibolit

style



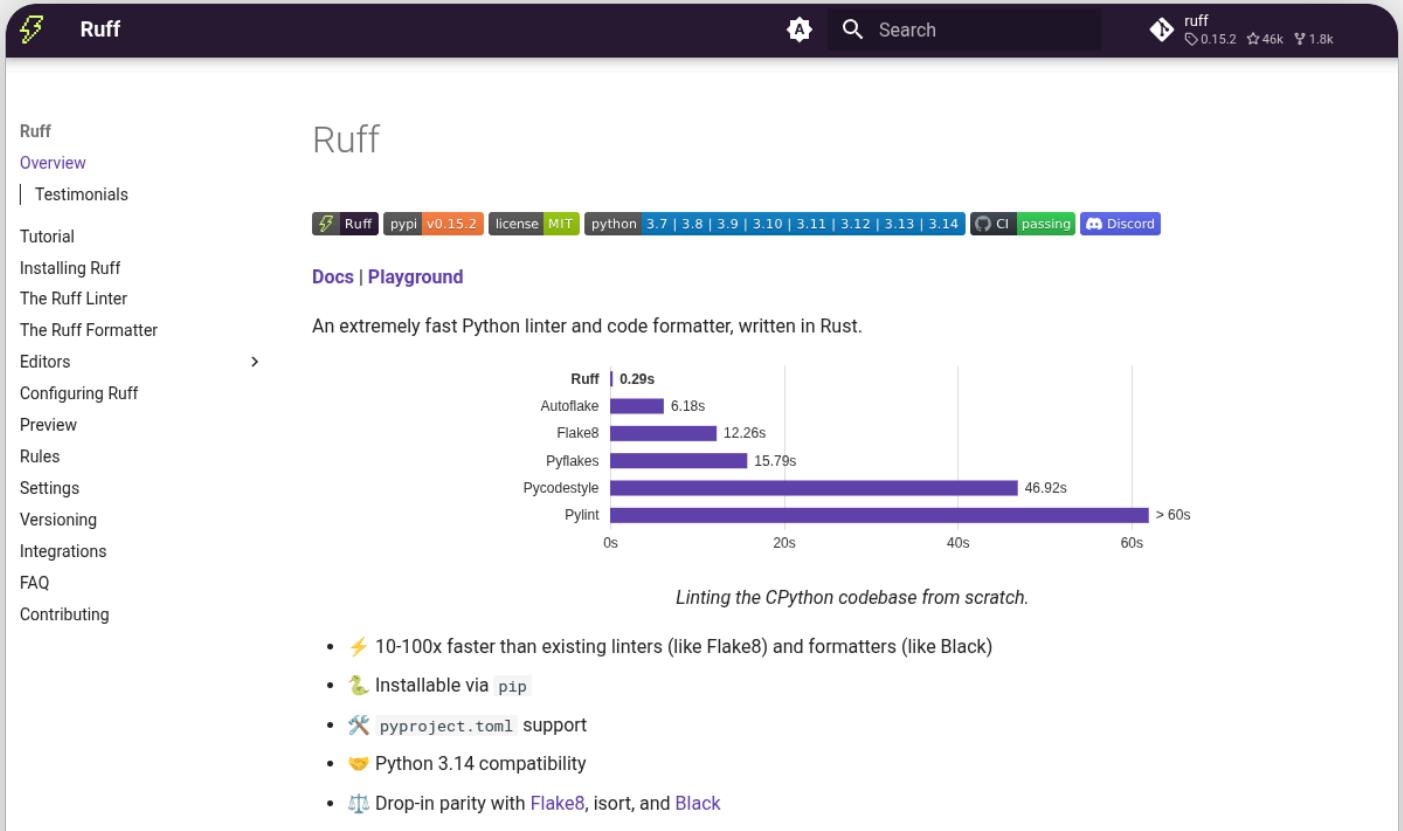
Painstacking work or magic? `style`



Code formatters for Python

- `black` — classic way (nondeterministic changes on edge cases).
- `blue` — more configurable, a little bit rules changed (writen on Python — slow on big projects).
- `isort` — sorts imports into standard/third-party/local groups
- `ruff` — linter + formatter, fast and all-in-one (writen on Rust ⚡)
- `autopep8` — fixes PEP8 violations automatically
- `yapf` — flexible formatting based on style config

Ruff `style`


Ruff
Search
ruff
0.15.2 46k 1.8k
Ruff
Overview
Testimonials
Tutorial
Installing Ruff
The Ruff Linter
The Ruff Formatter
Editors
Configuring Ruff
Preview
Rules
Settings
Versioning
Integrations
FAQ
Contributing
Ruff
Ruff pypi v0.15.2 license MIT python 3.7 | 3.8 | 3.9 | 3.10 | 3.11 | 3.12 | 3.13 | 3.14 CI passing Discord
Docs | Playground
An extremely fast Python linter and code formatter, written in Rust.
Ruff 0.29s
Autoflake 6.18s
Flake8 12.26s
Pyflakes 15.79s
Pycodestyle 46.92s
Pylint > 60s
0s
20s
40s
60s
Linting the CPython codebase from scratch.
⚡ 10-100x faster than existing linters (like Flake8) and formatters (like Black)
🐍 Installable via pip
🛠️ pyproject.toml support
🤝 Python 3.14 compatibility
⚖️ Drop-in parity with Flake8, isort, and Black

Why Ruff? `style`

- Very fast on real repositories, so it is easy to run constantly.
- Deterministic formatting with minimal configuration overhead.
- One ecosystem for both formatting and linting in the same toolchain.
- Simple `uv` workflow: install once, run everywhere (local, hooks, CI).
- Actively maintained and widely adopted in modern Python projects.

Ruff in action

main.py

```python
import os, sys, math, random


def calc_sum(a, b=[]):
    total = 0
    for i in a:
        if i == None:
            pass
        else:
            total += int(i)
    if b:
        total += sum(b)
    return total


x = [1, "2", None, 3]
y = [10, 20]
print("Sum is " + str(calc_sum(x, y)))
print(math.pi, random.randint(1, 100))
```

We can check format of this file with:

```
$ uv run ruff format --check main.py
```

```
Would reformat: main.py
1 file would be reformatted
```

And format it autmatically:

```
$ uv run ruff format --check main.py
```

```
1 file reformatted
```

Running on project `style`

```
$ uv run ruff format . --check
```

```
Would reformat: src/lib_demo/report.py
12 files would be reformatted, 38 files already formatted
```

```
$ uv run ruff format .
```

```
12 files reformatted, 37 files left unchanged
```

Configure Ruff formatter `style`

pyproject.toml

```toml
[tool.ruff]
line-length = 88
target-version = "py312"

[tool.ruff.format]
quote-style = "double"
indent-style = "space"
line-ending = "auto"
```

Install `uv` as devdependency:

```
uv add --dev ruff
```

Format entire project:

```
uv run ruff format .
```

Check without write changes:

```
uv run ruff format . --check
```

What is linter? How it differs from formatter? `lint`

- Linter analyses code and reports violations of rules and best practices.
- Formatter rewrites code style automatically (spaces, quotes, wraps, etc.).
- Linter focuses on quality risks: unused imports, bugs, complexity, anti-patterns.
- Formatter answer: “How code looks?” Linter answer: “Is code safe and clean?”
- Some linter rules are auto-fixable, but many need developer decision.

Popular Python linters `lint`

- `flake8` — classic plugin-based linter, many teams still use it.
- `pylint` — very strict analyzer, checks style and design smells deeply.
- `ruff` — very fast linter with large ruleset, drop-in for many flake8 plugins.
- `bandit` — security-oriented static checks for common Python vulnerabilities.

Why Ruff is special `lint`

- `ruff` combines both roles: formatter (`ruff format`) and linter (`ruff check`).
- Single tool, single config, single install path via `uv`.
- Ruff implements rules inspired by multiple tools and plugins.
- Examples: Pyflakes (`F`), pycodestyle (`E/W`), isort (`I`), bugbear (`B`), pyupgrade (`UP`).
- Result: fewer tools in repo, same or stronger quality gates.

Ruff in action again! lint

main.py

```python
import math
import random


def calc_sum(a, b=[]):
    total = 0
    for i in a:
        if i == None:
            pass
        else:
            total += int(i)
    if b:
        total += sum(b)
    return total


x = [1, "2", None, 3]
y = [10, 20]
print("Sum is " + str(calc_sum(x, y)))
print(math.pi, random.randint(1, 100))
```

Run checks on your code:

```
$ uv run ruff check main.py
```

```
main.py:7:17: E711 Comparison to `None` should be `cond is None`
  |
5 |     total = 0
6 |     for i in a:
7 |         if i == None:
  |                 ^^^^ E711
8 |             pass
9 |         elif type(i) == int:
  |
  = help: Replace with `cond is None`
```

Try to fix someone:

```
$ uv run ruff check main.py --fix
```

```
Found 4 errors (3 fixed, 1 remaining).
No fixes available (1 hidden fix can be enabled with the `--unsa
```

Configure Ruff linter lint

pyproject.toml

```toml
[tool.ruff.lint]
select = ["ALL"]
ignore = ["D203", "D212", "COM812", "PLR0913"]

[tool.ruff.lint.per-file-ignores]
"tests/**/*.py" = ["S101", "D", "INP001", "ARG001", "ARG002"]
```

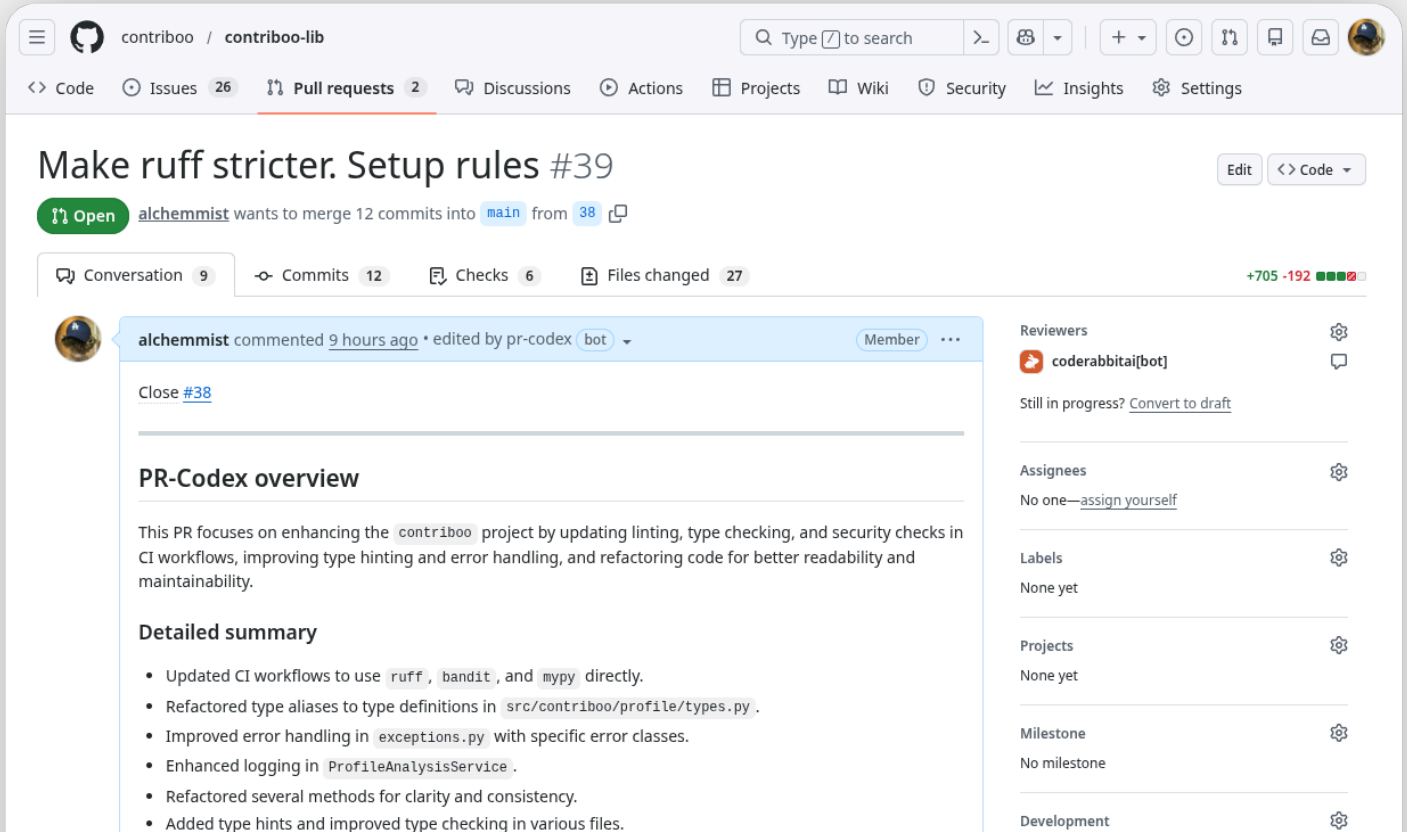
Install and run:

```
uv add --dev ruff
uv run ruff check .
uv run ruff check . --fix
```

With formatter in one flow:

```
uv run ruff check . --fix --unsafe-fixes
uv run ruff format .
```

Let's see new pull reqeust `live-demo`



Static analyzers (recap) `types`

We already covered theory in previous lecture, here is a practical map.

- `mypy`: reference checker, huge ecosystem and many plugins/extensions.
- `pyright` / `basedpyright`: very fast, strict defaults, strong IDE feedback.
- `pyre`: Meta checker, good performance, mostly used in specific stacks.
- `pytype`: Google checker with deep inference, less common in OSS tooling today.
- `ty` (Astral team): promising new checker from the same team, but still early and raw.
- In practice: choose one primary checker and run it everywhere via `uv` + CI.

Why mypy in this course? `types`

- Mature and stable: predictable behavior for students and CI.
- Rich ecosystem of plugins/extensions for real frameworks and libraries.
- Lots of docs, examples, and community answers for every typical issue.
- Strict mode gives clear quality gate for typed Python in educational projects.
- We can add `pyright` as second opinion, but keep `mypy` as baseline.
- `ty` is interesting, but currently too early to be course baseline.

Type checker configs (short) types

`pyproject.toml` (`mypy`):

```toml
[tool.mypy]
python_version = "3.12"
strict = true
warn_unused_ignores = true
warn_return_any = true
disallow_untyped_defs = true
```

`pyrightconfig.json`:

```json
{
  "pythonVersion": "3.12",
  "typeCheckingMode": "strict",
  "include": ["src"],
  "exclude": ["tests/fixtures"]
}
```

Run static analysis with `uv` `run`

```
$ uv add --dev mypy basedpyright
$ uv run mypy src
Success: no issues found in 27 source files

$ uv run basedpyright
0 errors, 0 warnings, 0 notes
```

What is `make`? `make`

- `make` is a task runner built around named targets in `Makefile`.
- Target is a command alias: `make lint`, `make test`, `make check`.
- Targets can depend on each other: one command can run a full pipeline.
- It standardizes local workflow, onboarding, and CI commands.

`contriboo-lib` Makefile structure `make`

Core quality flow:

```
check: lint types format-check test

lint:
  uv run ruff check src tests && markdownlint-cli2 .

types:
  uv run mypy --strict ... src tests

format:
  uv run ruff format && uv run ruff check --fix -s

test:
  uv run pytest -q
```

Other groups:

- Security: `security`
- Tests: `test-cov`
- Setup: `venv`, `requirements`
- Hooks: `pre-commit`, `pre-commit-install`, `pre-commit-uninstall`
- Utility: `clean`, `help`

Important internals:

```
.DEFAULT_GOAL := help
SHELL := /bin/bash
.SHELLFLAGS := -eu -o pipefail -c
UV := uv
RUFF := $(UV) run ruff
```

Make targets map (short) `make`

Use case	Targets
Main quality gate	`check`
Style and lint	`format`, `format-check`, `lint`
Typing	`types`
Tests	`test`, `test-cov`
Security	`security`
Setup	`venv`, `requirements`
Hooks	`pre-commit`, `pre-commit-install`, `pre-commit-uninstall`
Utility	`clean`, `help`

Example `make` runs from `contriboo-lib` run

```
$ make check
Running linters...
uv run ruff check src tests
markdownlint-cli2 .
Lint completed!

Checking types...
uv run mypy --strict --disallow-untyped-defs --disallow-incomplete-defs src tests
Success: no issues found in 19 source files
Type check completed!

Checking formatting...
uv run ruff format --check
50 files already formatted
Format check completed!

Running tests...
uv run pytest -q
16 passed in 0.52s
Tests completed!
```

```
$ make security
uv run bandit -r src/
No issues identified.
```

CI strategy in `contriboo-lib` ci

Workflows split by responsibility:

```
.github/workflows/ruff.yaml
.github/workflows/mypy.yaml
.github/workflows/pytest.yaml
.github/workflows/coverage.yaml
.github/workflows/security.yaml
.github/workflows/markdownlint.yaml
```

Each one runs on `pull_request`.

Why split instead of one huge job:

- Fast feedback per signal (lint/types/tests/security).
- Easy to rerun failed check only.
- Simple ownership of pipeline parts.
- Clear mapping from badge/status to action.

Real GitHub Actions examples ci

`ruff.yaml`:

```yaml
name: ruff
on:
  pull_request:
jobs:
  quality:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v6
      - uses: astral-sh/setup-uv@v7
      - run: uv sync
      - run: uv run ruff check src tests
```

`mypy.yaml`:

```yaml
name: mypy
on:
  pull_request:
jobs:
  types:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v6
      - uses: astral-sh/setup-uv@v7
      - run: uv sync
      - run: uv run mypy --strict --disallow-untyped-defs --disa
```

Coverage + security workflows ci

```yaml
name: pytest-coverage
on:
  pull_request:
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: astral-sh/setup-uv@v7
      - run: uv sync
      - run: |
          export PYTHONPATH=$PYTHONPATH:$(pwd)/src
          uv run pytest --cov=src --cov-report=term-missing:skip-covered --cov-fail-under=70 tests/
```

```yaml
name: security
on:
  pull_request:
jobs:
  security:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v6
      - uses: astral-sh/setup-uv@v7
      - run: uv sync
```

Local `make` and CI mapping `ci`

Local command	GitHub Actions workflow
`make lint`	`ruff.yaml` + `markdownlint.yaml`
`make types`	`mypy.yaml`
`make test`	`pytest.yaml`
`make test-cov`	`coverage.yaml`
`make security`	`security.yaml`
`make check`	aggregate local gate before push

Summary

- `ruff format` gives deterministic style with minimal setup.
- Linters catch quality issues early: Ruff for Python, markdownlint for docs.
- Static analyzers are short in theory, strict in practice: `uv run mypy --strict ...`.
- `contriboo-lib` Makefile covers full dev lifecycle: setup, checks, security, hooks, cleanup.
- CI in GitHub Actions mirrors local commands with focused workflows.

END