Tools to setup great Python projects

PyCon Portugal, 2023

Introduction

I'll talk about my experience with tools that can:

- **Improve quality** of a python project
- **Reduce effort** for some tasks

My suggestion: keep it iterative!

Table of Contents

About me

Duarte Pompeu

Software Engineer
@ xgeeks

linkedin.com/in/duartepompeu

1

Package managers

Package managers

Goal: **reproducible application**

Using requirements.txt: a good start.

Using a package manager: even better.

requirements.txt

Requirements file:

```
fastapi==0.103.1
```

Problems:

- Easy to make mistakes:
  - forget to activate virtual env
  - forget to write package down
- No sub-dependencies
- No separation between deployed and development packages

Poetry

You define your desired dependencies in `pyproject.toml`:

```
[tool.poetry.dependencies]
python = "^3.11"
Flask = "2.2.*"

[tool.poetry.group.dev.dependencies]
pytest = "^7.3.1"
```

Poetry

Poetry keeps track of all dependencies in `poetry.lock`:

```
# This file is automatically @generated by Poetry 1.5.1 and
should not be changed by hand.

[[package]]
name = "certifi"
version = "2023.7.22"
description = "Python package for providing Mozilla's CA
Bundle."
optional = false
python-versions = ">=3.6"
files = [
    {file = "certifi-2023.7.22-py3-none-any.whl", hash =
"sha256:92d6037539857d8206b8f6ae472e8b77db8058fec5937a1ef3f54
304089edbb9"},
    {file = "certifi-2023.7.22.tar.gz", hash =
```

2

Code formatters

Formatting

Goal: **consistent code style**

A formatter can enforce rules to improve consistency.

My preference: black + isort

Black

“Any color you want, as long as it’s black”

It calls itself “the uncompromising code formatter”

Usage: `black .`

Isort

“Isort your imports, so you don’t have to.”

Usage: `isort --profile=black .`

Before

```
import random
import logging
def a_very_long_function_name(a_very_long_parameter_name,yet_another_very_long_parameter_name):
      a_very_long_variable_name = a_very_long_parameter_name yet_another_very_long_parameter_name
      logging.info(a_very_long_variable_name)
      return a_very_long_variable_name
```

After

```
import logging
import random


def a_very_long_function_name(
      a_very_long_parameter_name, yet_another_very_long_parameter_name
):
      a_very_long_variable_name = (
          a_very_long_parameter_name + yet_another_very_long_parameter_name
      )
      logging.info(a_very_long_variable_name)
      return a_very_long_variable_name
```

3

Code linters

Linting

Goal: **reduce basic errors**

A code linter can help detecting defects.

My preference: ruff with flake8 rules.

Alternatives: pylint, flake8, etc.

Ruff

“An extremely fast Python linter, written in Rust.”

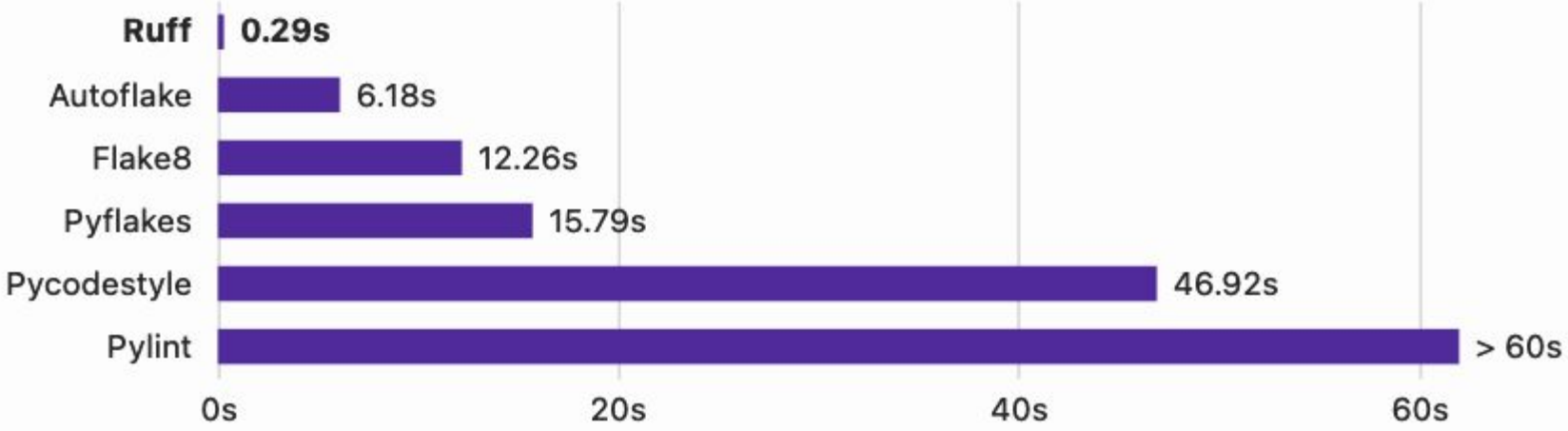
Over 600 built-in rules, can replace linters such as flake8.

Usage:

```
ruff check .
```

```
ruff check . --fix
```

Ruff



Linting the CPython codebase from scratch.

Ruff

“An extremely fast Python linter, written in Rust.”

Over 600 built-in rules, can replace linters such as flake8.

Usage:

```
ruff check .
```

```
ruff check . --fix
```

Before

```
import logging
import random


def f(a, b):
     x = 5
     return a + b
```

After

```
def f(a, b):
     return a + b
```

4

Test libraries

Testing

Goal: **reduce bugs / manual testing**

A test library / tool can help write tests, check coverage, etc.

Pytest

"... makes it easy to write small tests, yet scales to support complex functional testing..."

Reduces boilerplate code, compared to `unittest` from the standard library.

Supports features like parametrization, dynamic generation of tests, etc.

Using unittest

```
import unittest

from demo.main import f


class TestF(unittest.TestCase):
    def test_f(self):
        self.assertEqual(5, f(2, 3))


if __name__ == "__main__":
    unittest.main()
```

Using pytest

```
from demo.main import f


def test_f():
     assert f(2, 3) == 5
```

Using advanced pytest

```
import pytest
from demo.main import f


@pytest.mark.parametrize(
  “a,b,expected”,
[
   pytest.parameter(2,3,5, id=”positives”),
   pytest.parameter(0,0,0, id=”zeroes”),
   pytest.parameter(-2,-3,-5, id=”negatives”),
   pytest.parameter(2,-3,-1, id=”mixed”),

])
def test_f(a, b, expected):
    assert f(a, b) == expected
```

Using advanced pytest

Others:

- fixtures
- Conditionals: only run this test if the other passes
- Dynamic generation of tests

Other test tools

`coverage`: reports statement coverage

`hypothesis`: test generic properties

`locust`: load testing

5

Centralized configs

Centralized configs

Goal: **re-use configuration** (CLI, IDE, CI/CD, ...)

How? Store configs in `pyproject.toml`

pyproject.toml

```
[tool.isort]
profile = "black"

[tool.ruff]
select = [
     "E",  # pycodestyle
     "F",  # pyflakes
     "B",  # bugbear: finds potential bugs
     "UP", # pyupgrade: warns about deprecated features, eg typing
     "S",  # flake8-bandit: security warnings
]
ignore = [
     "E501", # line too long -> handled by formatter
]
```

6

Python manager

An useful extra

Python manager

The goal: **manage different python versions**

Why: working on projects with **different python versions** can be **complex**

My preference: Use `pyenv`

pyenv

"...lets you easily switch between multiple versions of Python."

Usage:

- `pyenv install 3.11.4`
- `poetry env use ~./pyenv/versions/3.11.4/bin/python`

Conclusion

Goals:

- **Improve quality** of a python project
- **Reduce effort** for some tasks

Package manager: reproducible application

Formatter: consistent code style

Linter: avoid basic errors

Test library: fewer bugs, less manual testing

Centralized configurations: avoid split/outdated configuration

Python manager: use different Python versions

That’s all folks!

Questions? Comments? Other interesting tools?

Go ahead!

Demo: github.com/duarte-pompeu/great-tools-pyconpt23

linkedin.com/in/duartepompeu