Linters, Formatters, and Other Tools and Practices for Python

PICSciE/RC software engineering summer school 2024

July 31st 2024, Rob Bierman, Research Software Engineer

With your permission I'd like to ask a few very personal questions

- (a) Mac or (b) PC?
- (a) Coke or (b) Pepsi?
- (a) Yankees or (b) anyone else?
- (a) Tabs or (b) spaces?
- (a) 'quotes' or (b) "quotes"?
- (a) y=ax+b or (b) y = a * x + b ?

Goals for this session

- Discussion of developer tools
- Format some code with `black`!
- Format and lint some code with `ruff`!
- Setup a pre-commit hook to automatically format and lint!

Everything we discuss today will be for python, but other programming languages have the same or similar tools and concepts!

Here's an example of what we'll accomplish with formatting and linting!

Code before:

```
def calculate_tip(cost, tip_fraction):
    try:
        tip_percent = float(tip_fraction)
        tip_percent >= 0
    except:
        print('Tip must be between 0 and 1')

    O =  cost*( 1+tip_percent );
    return O
```

Code after:

```
def calculate_tip(cost, tip_fraction):
    try:
        tip_percent = float(tip_fraction)
        assert tip_percent >= 0
    except ValueError:
        raise ValueError("Error tip < 0")

    total = cost * (1 + tip_percent)
    return total
```

We'll be using formatters and linters to change our code for us!

Following along and acknowledgements

If you want to follow along (it will be fun!) you'll need access to a terminal with `python` and `git` installed and the ability to `pip install` packages.

I learned quite a bit in preparing for this session and the following links were very helpful

- https://github.com/klieret/everything-you-didnt-now-you-needed
- https://www.slideshare.net/slideshow/embed_code/key/euNhpSgvuPL9kG
- https://github.com/henryiii/sqat-example

(I heavily borrowed from these resources, and if we have extra time we can look at them)

These materials were created by Kilian Lieret and Henry Schreiner who are excellent Research Software Engineers (RSEs) here at Princeton.
https://researchcomputing.princeton.edu/services/research-software-engineering

Development tool tradeoffs

Development tools solve problems for developers

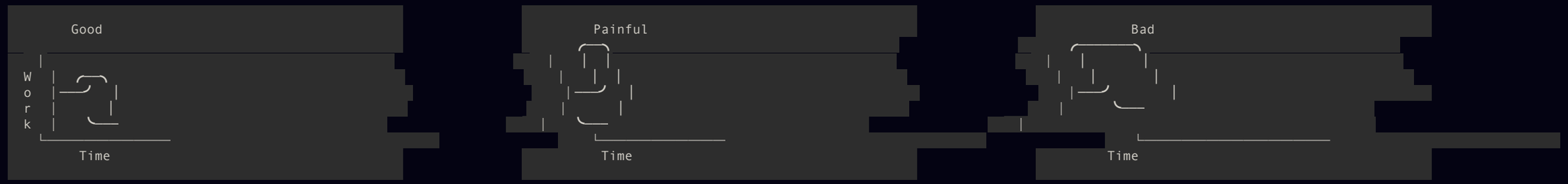
- Integrated Development Environments (IDEs)
  - Edit and view multiple files
- git/github
  - Rolling-back changes and collaborating with others
- Formatting and Linting
  - Standardize code style and enforce best practices
- Pre-commit hooks
  - Small checks before code is committed

Development tools add overhead to a project

- "New contributors welcome, no experience necessary!"
  - Just setup the dev environment...
  - Make sure the tests pass...
  - Add new tests for the code you've added...
  - Make sure you're using our coding style...

This can feel like a high barier to entry for new contributors.

Different types of dev tool learning experiences



By the end of this session, hopefully you'll agree that formatting and linting with pre-commit hooks provide enough value to outweigh the added "cruft"

This code works, but isn't formatted very nicely:

```
fruits = {"Pears":1   , "Apples": 4, "Banana":3}
print("Fruit counts:",fruits,'and thats it')
```

We could manually make the changes, OR we could use a formatter!

Let's use the black formatter

"The Uncompromising Code Formatter"

Fetching course materials and installing black

First fetch the class materials using `git clone`

```
git clone https://github.com/biermanr/formatters_and_linters
```

then `cd` into the directory and `ls` to see the files

```
cd formatters_and_linters
ls
```

Installing black

It would be better to use `pipx` to install/run `black`, but let's install it using `pip` for today in a venv.

```
python3 -m venv .venv      #create a venv virtual environment
source .venv/bin/activate  #activate the environment
which pip3                 #make sure venv pip3 is used
pip3 install black         #finally install black
```

If you have a successful `black` installation, you should be able to run `black --help` on the command line and get the help message.

Using black to reformat our fruits

When you git cloned the class materials you downloaded fruits.py as well as other files that we'll use later!

```
cat fruits.py #print the file contents to terminal
```

Which should just be these two lines:

```
fruit_counts = {"Pears":1       , "Apples": 4, "Banana":3}
print("Here are the fruits:",fruit_counts,'and thats it')
```

Now that we have black installed we can simply run:

```
black fruits.py
```

```
reformatted fruits.py
All done! ✨ 🍰 ✨
1 file reformatted.
```

If you `cat` the file again, you should see it has done the formatting for us!

```
fruit_counts = {"Pears": 1, "Apples": 4, "Banana": 3}
print("Here are the fruits:", fruit_counts, "and thats it")
```

Look at what it did to the quotes in the `print` statement!

Black had the audacity to change our code!

Try doing the same thing with `long_fruits.py`.

1. `cat long_fruits.py` to see what it looks like
2. `black long_fruits.py` to re-format it
3. `cat long_fruits.py` again to see the final output

```
fruit_counts = {"Pears":1       , "Apples": 4, "Banana":3, "Mango":1, "Grape":17, "Kiwi":1001}
```

```
fruit_counts = {
    "Pears": 1,
    "Apples": 4,
    "Banana": 3,
    "Mango": 1,
    "Grape": 17,
    "Kiwi": 1001,
}
```

How is `black` doing this? How does it know what "well-formatted" code looks like?

How is black working and how can it be configured?

Is black a benevolent AI?

No, it's an uncaring collection of opinionated rules

Turns out that black is pretty opinionated

> Black aims for consistency, generality, readability and reducing git diffs. Similar language constructs are formatted with similar rules. Style configuration options are deliberately limited and rarely added.

"Don't try and tell black what to do"

You can read about the black formatting style: https://black.readthedocs.io/en/stable/the_black_code_style/current_style.html

Which includes statements like:

> Pro-tip: If you're asking yourself "Do I need to configure anything?" the answer is "No". Black is all about sensible defaults. Applying those defaults will have your code in compliance with many other Black formatted projects.

I'm trying not to be offended. Let's look at the main other tool, ruff!

ruff is the hip new alternative to black

Ruff is new to the game, but gained immediate wide-spread appeal

GitHub Stars for Black vs. Ruff over time

Date	Black	Ruff
Mar 2018	0	N/A
Aug 2022	27,870	0
Feb 2023	30,390	7,800
Sep 2023	32,940	17,610
Jan 2024	35,460	21,510
Jul 2024	38,061	29,417

Let's install ruff and try it out ourselves!

```
which pip3
pip3 install ruff #install ruff
ruff help         #get the ruff help message
```

Ruff has multiple commands to choose from

```
(.venv) $ ruff help
Ruff: An extremely fast Python linter code formatter.

Usage: ruff [OPTIONS] <COMMAND>

Commands:
  check    Run Ruff on the given files or directories
  ...
  linter   List all supported upstream linters
  clean    Clear any caches in the current directory
  format   Run the Ruff formatter on the given files/dirs
  ...
```

ruff can format code similarly to black

Let's format the starting example tip-calculator code

Take a look with cat tip.py

```
1 def calculate_tip(cost, tip_fraction):
2     try:
3         tip_percent = float(tip_fraction)
4         tip_percent >= 0
5     except:
6         print("Non-numeric tip")
7
8     O =  cost*( 1+tip_percent );
9     return O
```

Format with ruff using

```
$ ruff format tip.py
  1 file reformatted
```

Running cat again:

```
1 def calculate_tip(cost, tip_fraction):
2     try:
3         tip_percent = float(tip_fraction)
4         tip_percent >= 0
5     except:
6         print('Non-numeric tip')
7
8     O = cost * (1 + tip_percent)
9     return O
```

It has fixed the spacing on line 8, and switched to single-quotes

`ruff` is also a linter

It was great that we improved the formatting of the tip.py code, but there are non-formatting issues with the code.

```
1 def calculate_tip(cost, tip_fraction):
2     try:
3         tip_percent = float(tip_fraction)
4         tip_percent >= 0
5     except:
6         print('The tip must be a number between 0 and 1')
7
8     O = cost * (1 + tip_percent)
9     return O
```

The main issue I see is on line `4` where the `tip_percent >= 0` comparison isn't being used anywhere

▒▒▒ This check is failing silently!

We're going to move into linting!

The line between formatting and linting can be a little blury

- Formatting is concerned with how the code looks
- Linting is concerned with "best practices" and identifying potential bugs

```python
def calculate_tip(cost, tip_fraction):
    try:
        tip_percent = float(tip_fraction)
        tip_percent >= 0
    except:
        print('Non-numeric tip')

    O = cost * (1 + tip_percent)
    return O
```

Run `ruff check tip.py`

We get a lot of output, let's just start with the first complaint

```
tip.py.hold:4:9: B015 Pointless comparison.
Did you mean to assign a value?
Otherwise, prepend `assert` or remove it.
  |
2 |     try:
3 |         tip_percent = float(tip_fraction)
4 |         tip_percent >= 0
  |         ^^^^^^^^^^^^^^^^ B015
5 |     except:
6 |         print('Non-numeric tip')
  |
```

Ahah! `ruff check` found the bug I complained about before (what a coincidence!)

What does B015 mean?

```
tip.py.hold:4:9: B015 Pointless comparison.
Did you mean to assign a value?
Otherwise, prepend `assert` or remove it.
  |
2 |     try:
3 |         tip_percent = float(tip_fraction)
4 |         tip_percent >= 0
  |         ^^^^^^^^^^^^^^^^ B015
5 |     except:
6 |         print('Non-numeric tip')
```

B015 is a Linting Rule!

We can ask ruff to explain this rule to us with ruff rule B015

```
# useless-comparison (B015)

Derived from the **flake8-bugbear** linter.

## What it does
Checks for useless comparisons.

## Why is this bad?
Useless comparisons have no effect on the program,
and are often included by mistake. If the comparison
is intended to enforce an invariant, prepend the
comparison with an `assert`. Otherwise, remove it entirely.

## Example
foo == bar

Use instead:
assert foo == bar, "`foo` and `bar` should be equal."
```

Fixing the comparison linting error

I'm going to use vim to make the suggested change to tip.py, of course feel free to use any text editor, even emacs I guess.

```python
def calculate_tip(cost, tip_fraction):
    try:
        tip_percent = float(tip_fraction)
        assert tip_percent >= 0
    except:
        print('Non-numeric tip')

    O = cost * (1 + tip_percent)
    return O
```

Now when you run the same ruff check tip.py you'll see its no longer upset about B015, but there are a few remaining issues.

```
tip.py.hold:5:5: E722 Do not use bare `except`
  |
3 |         tip_percent = float(tip_fraction)
4 |         assert tip_percent >= 0
5 |     except:
  |     ^^^^^^ E722
6 |         print('Non-numeric tip')
  |

tip.py.hold:8:5: E741 Ambiguous variable name: `O`
  |
6 |         print('Non-numeric tip')
7 |
8 |     O = cost * (1 + tip_percent)
  |     ^ E741
9 |     return O
  |
```

Take a look at ruff rule E741

Final form of tip.py to make ruff happy

```python
def calculate_tip(cost, tip_fraction):
    try:
        tip_percent = float(tip_fraction)
        assert tip_percent >= 0
    except ValueError:
        print('The tip must be between 0 and 1')

    total = cost * (1 + tip_percent)
    return total


print(calculate_tip(100, 0.18))
```

and now when we run ruff check tip.py we see

```
All checks passed!
```

Where do these Linting Rules come from?

Linting rules come from a few different places!

`ruff linting` shows the full list of linting rules that ruff knows

```
    F Pyflakes
  E/W pycodestyle
  C90 mccabe
    I isort
    N pep8-naming
    D pydocstyle
    ...
 FURB refurb
  DOC pydoclint
  RUF Ruff-specific rules
```

Unlike `black` we are encouraged to configure `ruff's` behavior

This is controlled either in a `ruff.toml` or a `pyproject.toml` if you are writing a python package. Let's just use `ruff.toml` for today.

Take a look at the `ruff.toml` in the course materials

```
[lint]
# Remember `B015`? It's in the `B` lints.
select = ["E4", "E7", "E9", "F", "B"]

[format]
quote-style = "single"
```

Let's try adding `D` for `pydoclint` suite of and linting again with `ruff check tip.py`

Oh right! Docstrings...

Let's skip the `D` lints actually. I'll change `ruff.toml` back to how it was

Let's lint an old friend ("andres_tricky_bug.py")

```python
from math import prod

def add_to_list(*elements, starting_list=[]):
    starting_list.extend(elements)
    return starting_list

def sylvester(n):
    sequence = add_to_list(2)
    for _ in range(n-1):
        new_num = 1 + prod(sequence)
        sequence = add_to_list(
            new_num,
            starting_list=sequence,
        )
    return sequence

print(sylvester(2))
print(sylvester(3))
print(sylvester(4))
```

Let's run the `ruff linter` with `ruff check andres_tricky_bug.py`

```
andres_tricky_bug.py:3:42: B006
Mutable data structures for argument defaults
  |
1 | from math import prod
2 |
3 | def add_to_list(*elements, starting_list=[]):
  |                                          ^^ B006
4 |     starting_list.extend(elements)
5 |     return starting_list
  |
  = help: Replace with `None`; initialize within function

Found 1 error.
No fixes available (1 hidden fix can be enabled with
the `--unsafe-fixes` option).
```

Let's lint an old friend ("andres_tricky_bug.py") with `--unsafe-fixes`

```
ruff check --unsafe-fixes andres_tricky_bug.py
```

```
andres_tricky_bug.py:3:42: B006 [*] Do not use mutable data structures for argument defaults
  |
1 | from math import prod
2 |
3 | def add_to_list(*elements, starting_list=[]):
  |                                          ^^ B006
4 |     starting_list.extend(elements)
5 |     return starting_list
  |
  = help: Replace with `None`; initialize within function

Found 1 error.
[*] 1 fixable with the --fix option.
```

Lets try with the `--fix` option!

```
ruff check --unsafe-fixes --fix andres_tricky_bug.py
```

ruff fixed the problem for us!

```
cat andres_tricky_bug.py
```

```python
from math import prod

def add_to_list(*elements, starting_list=None):
    if starting_list is None:
        starting_list = []
    starting_list.extend(elements)
    return starting_list

def sylvester(n):
    sequence = add_to_list(2)
    for _ in range(n-1):
        new_num = 1 + prod(sequence)
        sequence = add_to_list(new_num, starting_list=sequence)
    return sequence

print(sylvester(2))
print(sylvester(3))
print(sylvester(4))
```

ruff is pretty cool!

Also remember when ruff told us that it was extremely fast?

ruff is written in rust and is VERY FAST

Here's how long various linters take to lint the CPython codebase

Tool	Time(s)
Ruff	00.29
Autoflake	06.18
Flake8	12.26
Pyflakes	15.97
Pycodestyle	46.92
Pylint	60.00+

CPython has 600,000 lines of python code (citation needed)

If you're like me and writing <<< 600,000 lines of code then lint speed might not be your biggest concern.

But I find ruff is user-friendly anyway, so for me it's just a bonus that it's fast!

Also people praise ruff for consolidating functionality from many tools.

- Formatting
- Linting
- import sorting (with ISORT which we won't talk about)

Now that we're all convinced ruff is great, how can we best add it to our developer workflow?

We'll just add another developer tool!

I can hear you say "Oh boy, another developer tool..."

```
W
o
r
k

Time
```

`pre-commit` makes it easy to run code quality checks (like `ruff`!) before you commit your code.

It would be both annoying and error prone to have to remember to run `ruff` or `black` every single time we change our code and make a git commit.

In order to follow along, you need to have `git`, so try `git status`

It will list files that you've changed since running `git clone`

```
On branch main
Your branch is up to date with 'origin/main'.

Changes not staged for commit:
    modified:   fruits.py
    modified:   long_fruits.py
    modified:   tip.py

no changes added to commit...
```

Nobody stops you from committing "non-ideal" code

Let's make some code that we aren't proud of 😈

Let's make a big file

```
yes > no
```

Oops CTRL+C, CTRL+C! Hmm, no that's too big. Let's try:

```
yes | head -n 10000000 > large_file.txt
```

Finally lets make ugly.py

```
echo "print('yes','no',             'maybe')" > ugly.py
```

Now we're ready to add these to our commit! No-one is stopping us! (don't follow along here, just watch~)

```
git add large_file.txt ugly.py
git commit
```

Ok, I hope I made my point! How can we get some "parental supervision?"

Pre-commit can take a look at you "non-ideal" code before it makes it in a commit

Let's install pre-commit using our .venv

```
pip install pre-commit
pre-commit --version
cat .pre-commit-config.yaml
```

```yaml
repos:
-   repo: https://github.com/pre-commit/pre-commit-hooks
    ...
- repo: https://github.com/astral-sh/ruff-pre-commit
  # Ruff version.
  rev: v0.5.5
  hooks:
    # Run the linter with --fix on py and jup
    - id: ruff
      types_or: [ python, pyi, jupyter ]
      args: [ --fix ]
    # Run the formatter.
    - id: ruff-format
      types_or: [ python, pyi, jupyter ]
```

Then let's try to commit some "non ideal" code

```
pre-commit install #important
cat ugly.py
git add large_file.txt ugly.py
git commit
```

Pre-commit is NOT happy with us

```
$ git commit
Check Yaml..........................Passed
...
```

But it did the work for us!

So now we can just `git add` and `git commit` again

Summary of what we've discussed

- Discussion of developer tools (they are friends sometimes)
- Format some code with `black`!
- Format and lint some code with `ruff`! (probably ok to just use ruff, not black)
- Setup a pre-commit hook to automatically format and lint! (QC checking)

Thanks for listening! Again thanks to Henry and Kilian for slides!

Good luck incorporating these tool into your workflow!

You don't have to do everything at once, but pre-commit is a good place to start!

Yes, this is the last slide