Styling and Formatting Code

Styling and Formatting Code

```
Code is read more often than it is written. -- Guido Van Rossum (author of Python)
```

When we write a piece of code, it's almost never the last time we see it or the last time it's edited. So we need to make it easy to read. One of the easiest ways to make code more readable is to follow consistent style and formatting conventions. There are many options when it comes to Python style conventions to adhere to, but most are based on `PEP8` conventions. Different teams follow different conventions and that's perfectly alright.

Basics of styling

The most important aspects are:

- `consistency`: everyone follows the same standards.
- `automation`: formatting should be largely effortless after initial configuration.

Tools

We will be using two very popular tools for style and formatting conventions that makes some very opinionated decisions on our behalf (with configurable options).

- Ruff: An extremely fast Python linter and code formatter, written in Rust.
- isort: sorts and formats import statements inside Python scripts.

Install

```
# that way (--dev) it is installed as a dev dépendency, you don't need that in prod.
uv add ruff --dev
uv add isort --dev
```

usage

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

Wait ?!

But I don't want to run these commands one by one, I'm too lazy. Let's make a `makefile`. This file can be used to define a set of commands that can be executed with a single command.

Note: a `makefile` is the easiest way to create a file that store commands. But there are other softwares. For example:

- Task: Task is a task runner / build tool that aims to be simpler and easier to use than, for example, GNU Make. Written in Go.
- just written in Rust.

```
# Makefile
SHELL = /bin/bash

# Styling
.PHONY: style
style:
	ruff check . --fix
	isort .

# Cleaning
# the clean command depends on the style command
# which means that style will be executed first before clean is executed.
.PHONY: clean
clean: style
	find . -type f -name "*.DS_Store" -ls -delete
	find . | grep -E "(__pycache__|\.pyc|\.pyo)" | xargs rm -rf
	find . | grep -E ".pytest_cache" | xargs rm -rf
	find . | grep -E ".ipynb_checkpoints" | xargs rm -rf
	rm -rf .coverage*
```

Pre-commit

The best way to automate this thing is to make sure that each time you commit your code, all the style conventions are respected. That's what pre-commit is for.

We use **pre-commit hooks**, which will **automatically** be triggered when we try to perform a commit. These hooks can ensure that certain rules are followed or specific actions are executed successfully and if any of them fail, the commit will be aborted.

Install

```
uv add pre-commit --dev
```

Usage

```
pre-commit install
pre-commit autoupdate
pre-commit commit --all
```

But before running `pre-commit install` you need to define a YAML configuration file. An easy way to generate it is to use the pre-commit cli.

Simple config

```
pre-commit sample-config > .pre-commit-config.yaml
cat .pre-commit-config.yaml
```

```yaml
# See https://pre-commit.com for more information
# See https://pre-commit.com/hooks.html for more hooks
repos:
-   repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v3.2.0
    hooks:
    -   id: trailing-whitespace
    -   id: end-of-file-fixer
    -   id: check-yaml
    -   id: check-added-large-files
```

Where can I find hooks ?

- **Built-in**: Inside the sample configuration, we can see that pre-commit has added some default hooks from it's repository. We can read about the function of these hooks and add even more by exploring pre-commit's built-in hooks.
- **Custom**: Besides pre-commit's built-in hooks, there are also many custom, 3rd party popular hooks that we can choose from.

For example, hooks for ruff and isort.