(Pre-)Commit to Better Code

Stefanie Molin

Bio

- 👩‍💻 Software engineer at Bloomberg in NYC
- 🚀 Creator of the numpydoc-validation and exif-stripper pre-commit hooks
- ✍️ Author of "Hands-On Data Analysis with Pandas"
- 🎓 Bachelor's degree in operations research from Columbia University
- 🎓 Master's degree in computer science from Georgia Tech

Prerequisites

- Comfort writing Python code and working with Git on the command line using basic commands (*e.g.*, `clone`, `status`, `diff`, `add`, `commit`, and `push`)
- Have Python and Git installed on your computer, as well as a text editor for writing code (*e.g.*, Visual Studio Code)
- Fork and clone this repository: github.com/stefmolin/pre-commit-workshop
- Open up these slides in your browser and use the arrow keys to follow along: stefaniemolin.com/pre-commit-workshop

Agenda

1. Setting Up Pre-Commit Hooks
2. Creating a Pre-Commit Hook

Setting Up Pre-Commit Hooks

Overview of Git hooks

Overview of Git hooks

- Scripts triggered when taking certain actions on a repository (*e.g.*, committing changes)

Overview of Git hooks

- Scripts triggered when taking certain actions on a repository (*e.g.*, committing changes)
- Can be client-side or server-side

Overview of Git hooks

- Scripts triggered when taking certain actions on a repository (*e.g.*, committing changes)
- Can be client-side or server-side
- Stored in the `.git/hooks/` directory of your repository, but excluded from version control

```
.git
└── hooks
    ├── commit-msg.sample
    ├── pre-commit.sample
    ├── pre-merge-commit.sample
    ├── pre-push.sample
    └── [...]
```

Overview of Git hooks

- Scripts triggered when taking certain actions on a repository (*e.g.*, committing changes)
- Can be client-side or server-side
- Stored in the `.git/hooks/` directory of your repository, but excluded from version control

```
.git
└── hooks
    ├── commit-msg.sample
    ├── pre-commit.sample
    ├── pre-merge-commit.sample
    ├── pre-push.sample
    └── [...]
```

Overview of pre-commit hooks

Overview of pre-commit hooks

- One pre-commit hook executable per repository

Overview of pre-commit hooks

- One pre-commit hook executable per repository
- Triggered by Git when you run `git commit`

Overview of pre-commit hooks

- One pre-commit hook executable per repository
- Triggered by Git when you run `git commit`
- Often used to enforce coding standards (*e.g.*, linting and formatting)

Overview of pre-commit hooks

- One pre-commit hook executable per repository
- Triggered by Git when you run `git commit`
- Often used to enforce coding standards (*e.g.*, linting and formatting)
- Should only include checks that run quickly

Overview of pre-commit hooks

- One pre-commit hook executable per repository
- Triggered by Git when you run `git commit`
- Often used to enforce coding standards (*e.g.*, linting and formatting)
- Should only include checks that run quickly
- The commit fails if *any* of the checks fail

How to enable pre-commit hooks

How to enable pre-commit hooks

1. Create an executable script that runs any checks you want to include and save it as `.git/hooks/pre-commit`,

How to enable pre-commit hooks

1. Create an executable script that runs any checks you want to include and save it as `.git/hooks/pre-commit`,
2. Or, use a tool like `pre-commit` and include the hook configuration in version control.

Pre-commit hooks != `pre-commit`

Pre-commit hooks != `pre-commit`

- `pre-commit` is "a multi-language package manager for pre-commit hooks" (source).

Pre-commit hooks != `pre-commit`

- `pre-commit` is "a multi-language package manager for pre-commit hooks" (source).
- `pre-commit` installs its own script as the pre-commit hook executable for your repository.

Pre-commit hooks != `pre-commit`

- `pre-commit` is "a multi-language package manager for pre-commit hooks" (source).
- `pre-commit` installs its own script as the pre-commit hook executable for your repository.
- The installed pre-commit hook then runs `pre-commit`, which in turn runs any checks you specify using a YAML file.*

---

*Check out my "A Behind-the-Scenes Look at How Pre-Commit Works" article for more information.

Setup

Setup

1. Install the `pre-commit` package in your repository.

Setup

1. Install the `pre-commit` package in your repository.
2. Define the checks you want to use in a file called `.pre-commit-config.yaml`.

Setup

1. Install the `pre-commit` package in your repository.
2. Define the checks you want to use in a file called `.pre-commit-config.yaml`.
3. Have `pre-commit` install itself as your pre-commit hook.

Setup

1. Install the `pre-commit` package in your repository.
2. Define the checks you want to use in a file called `.pre-commit-config.yaml`.
3. Have `pre-commit` install itself as your pre-commit hook.
4. Work on your code as usual.

Package installation with `pip`

```
$ python3 -m pip install pre-commit
```

Package installation with `uv`

```
$ uv tool install pre-commit --with pre-commit-uv
```

---

* It's also possible to use `uv add --dev pre-commit` to install as a development dependency; however, you will then need to prefix each call to `pre-commit` with `uv run`, which will make it harder to follow along with the workshop. Installing it this way also rewires `pre-commit` to use `uv`, which will speed things up.

Basic configuration

Basic configuration

Defined in `.pre-commit-config.yaml` at the repository root:

```yaml
repos:
  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v5.0.0
    hooks:
      - id: check-toml
      - id: check-yaml
      - id: end-of-file-fixer
      - id: trailing-whitespace
```

Source: How to Set Up Pre-Commit Hooks by Stefanie Molin

Basic configuration

The `repos` section lists the repositories to use:

```yaml
repos:
  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v5.0.0
    hooks:
      - id: check-toml
      - id: check-yaml
      - id: end-of-file-fixer
      - id: trailing-whitespace
```

Source: How to Set Up Pre-Commit Hooks by Stefanie Molin

Basic configuration

This represents the configuration for a single repository:

```yaml
repos:
  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v5.0.0
    hooks:
      - id: check-toml
      - id: check-yaml
      - id: end-of-file-fixer
      - id: trailing-whitespace
```

Source: How to Set Up Pre-Commit Hooks by Stefanie Molin

Basic configuration

The URL of the repository to clone:

```yaml
repos:
  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v5.0.0
    hooks:
      - id: check-toml
      - id: check-yaml
      - id: end-of-file-fixer
      - id: trailing-whitespace
```

Source: How to Set Up Pre-Commit Hooks by Stefanie Molin

Basic configuration

The tag or commit hash to clone at:

```yaml
repos:
  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v5.0.0
    hooks:
      - id: check-toml
      - id: check-yaml
      - id: end-of-file-fixer
      - id: trailing-whitespace
```

Source: How to Set Up Pre-Commit Hooks by Stefanie Molin

Basic configuration

The hooks to run from that repository:

```yaml
repos:
  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v5.0.0
    hooks:
      - id: check-toml
      - id: check-yaml
      - id: end-of-file-fixer
      - id: trailing-whitespace
```

Source: How to Set Up Pre-Commit Hooks by Stefanie Molin

Git hook installation

While the `.pre-commit-config.yaml` file lives in version control, each contributor must run this locally:

```
$ pre-commit install
pre-commit installed at .git/hooks/pre-commit
```

Add the configuration to version control

The `check-yaml` hook we included will make sure that the `.pre-commit-config.yaml` file (and any other YAML files in this repository) is valid YAML. Try committing it:

```
$ git add .pre-commit-config.yaml
$ git commit -m "Add pre-commit config"
[INFO] Initializing environment for [...]/pre-commit-hooks.
[INFO] Installing environment for [...]/pre-commit-hooks.
[INFO] Once installed this environment will be reused.
[INFO] This may take a few minutes...
check toml...........................(no files to check)Skipped
check yaml...........................................Passed
fix end of files.....................................Passed
trim trailing whitespace.............................Passed
```

Add the configuration to version control

The `check-yaml` hook we included will make sure that the `.pre-commit-config.yaml` file (and any other YAML files in this repository) is valid YAML. Try committing it:

```
$ git add .pre-commit-config.yaml
$ git commit -m "Add pre-commit config"
[INFO] Initializing environment for [...]/pre-commit-hooks.
[INFO] Installing environment for [...]/pre-commit-hooks.
[INFO] Once installed this environment will be reused.
[INFO] This may take a few minutes...
check toml...........................(no files to check)Skipped
check yaml...........................................Passed
fix end of files.....................................Passed
trim trailing whitespace.............................Passed
```

Add the configuration to version control

The `check-yaml` hook we included will make sure that the `.pre-commit-config.yaml` file (and any other YAML files in this repository) is valid YAML. Try committing it:

```
$ git add .pre-commit-config.yaml
$ git commit -m "Add pre-commit config"
[INFO] Initializing environment for [...]/pre-commit-hooks.
[INFO] Installing environment for [...]/pre-commit-hooks.
[INFO] Once installed this environment will be reused.
[INFO] This may take a few minutes...
check toml.....................(no files to check)Skipped
check yaml.....................................Passed
fix end of files...............................Passed
trim trailing whitespace.......................Passed
```

Exercise

There are more hooks provided by the repository that we are currently using. Take a moment to look through their documentation, and add some additional hooks to the setup that are relevant to your work. Be sure to commit your changes.

github.com/pre-commit/pre-commit-hooks

Example solution

Adding these two hooks ensures that your scripts are actually executable:

```
 repos:
   - repo: https://github.com/pre-commit/pre-commit-hooks
     rev: v5.0.0
     hooks:
+      - id: check-executables-have-shebangs
+      - id: check-shebang-scripts-are-executable
       - id: check-toml
       - id: check-yaml
       - id: end-of-file-fixer
       - id: trailing-whitespace
```

Third-party hooks

We are free to use hooks from multiple repositories. A non-exhaustive list of popular hooks along with tips for searching for compatible repositories can be found at pre-commit.com/hooks.html.

Third-party hooks

We are free to use hooks from multiple repositories. A non-exhaustive list of popular hooks along with tips for searching for compatible repositories can be found at pre-commit.com/hooks.html.

Let's add `ruff` to lint and format our Python code, since it is "10-100x faster than existing linters (like Flake8) and formatters (like Black),"* and speed very is important with pre-commit hooks.

---

* Source: `ruff` homepage as of June 16, 2024

Adding `ruff` to `.pre-commit-config.yaml`

Adding `ruff` to `.pre-commit-config.yaml`

Create a new entry under the `repos` key:

```yaml
repos:
  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v5.0.0
    hooks:
      - id: check-toml
      - id: check-yaml
      - id: end-of-file-fixer
      - id: trailing-whitespace

  - repo: https://github.com/astral-sh/ruff-pre-commit
    rev: v0.11.12
    hooks:
      - id: ruff-check
        args: [--fix, --exit-non-zero-on-fix, --show-fixes]
      - id: ruff-format
```

Adding `ruff` to `.pre-commit-config.yaml`

The `ruff` pre-commit hook is in a separate repository:

```yaml
repos:
  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v5.0.0
    hooks:
      - id: check-toml
      - id: check-yaml
      - id: end-of-file-fixer
      - id: trailing-whitespace

  - repo: https://github.com/astral-sh/ruff-pre-commit
    rev: v0.11.12
    hooks:
      - id: ruff-check
        args: [--fix, --exit-non-zero-on-fix, --show-fixes]
      - id: ruff-format
```

Adding `ruff` to `.pre-commit-config.yaml`

The tag or commit hash to clone at:

```yaml
repos:
  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v5.0.0
    hooks:
      - id: check-toml
      - id: check-yaml
      - id: end-of-file-fixer
      - id: trailing-whitespace

  - repo: https://github.com/astral-sh/ruff-pre-commit
    rev: v0.11.12
    hooks:
      - id: ruff-check
        args: [--fix, --exit-non-zero-on-fix, --show-fixes]
      - id: ruff-format
```

Adding `ruff` to `.pre-commit-config.yaml`

The hooks to run from that repository:

```yaml
repos:
  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v5.0.0
    hooks:
      - id: check-toml
      - id: check-yaml
      - id: end-of-file-fixer
      - id: trailing-whitespace

  - repo: https://github.com/astral-sh/ruff-pre-commit
    rev: v0.11.12
    hooks:
      - id: ruff-check
        args: [--fix, --exit-non-zero-on-fix, --show-fixes]
      - id: ruff-format
```

Adding `ruff` to `.pre-commit-config.yaml`

Use `args` to pass command line arguments to the hook:

```yaml
repos:
  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v5.0.0
    hooks:
      - id: check-toml
      - id: check-yaml
      - id: end-of-file-fixer
      - id: trailing-whitespace

  - repo: https://github.com/astral-sh/ruff-pre-commit
    rev: v0.11.12
    hooks:
      - id: ruff-check
        args: [--fix, --exit-non-zero-on-fix, --show-fixes]
      - id: ruff-format
```

Commit your changes

Commit your changes

This once again triggers an update to the environment used for running the checks:

```
$ git add .pre-commit-config.yaml
$ git commit -m "Add ruff pre-commit hooks"
[INFO] Initializing environment for [...]/ruff-pre-commit.
[INFO] Installing environment for [...]/ruff-pre-commit.
[INFO] Once installed this environment will be reused.
[INFO] This may take a few minutes...
check toml...........................(no files to check)Skipped
check yaml...........................................Passed
fix end of files.....................................Passed
trim trailing whitespace.............................Passed
ruff.................................(no files to check)Skipped
ruff-format..........................(no files to check)Skipped
```

Commit your changes

All hooks that were added or modified hooks need to be updated in the environment:

```
$ git add .pre-commit-config.yaml
$ git commit -m "Add ruff pre-commit hooks"
[INFO] Initializing environment for [...]/ruff-pre-commit.
[INFO] Installing environment for [...]/ruff-pre-commit.
[INFO] Once installed this environment will be reused.
[INFO] This may take a few minutes...
check toml.....................(no files to check)Skipped
check yaml.....................................Passed
fix end of files...............................Passed
trim trailing whitespace.......................Passed
ruff...........................(no files to check)Skipped
ruff-format....................(no files to check)Skipped
```

Commit your changes

This will be slower the first time, but the environment persists until modification:

```
$ git add .pre-commit-config.yaml
$ git commit -m "Add ruff pre-commit hooks"
[INFO] Initializing environment for [...]/ruff-pre-commit.
[INFO] Installing environment for [...]/ruff-pre-commit.
[INFO] Once installed this environment will be reused.
[INFO] This may take a few minutes...
check toml.....(no files to check)Skipped
check yaml.....Passed
fix end of files.....Passed
trim trailing whitespace.....Passed
ruff.....(no files to check)Skipped
ruff-format.....(no files to check)Skipped
```

Commit your changes

Hook order in the configuration file matters – hooks run in the order you specify and may make conflicting changes:

```
$ git add .pre-commit-config.yaml
$ git commit -m "Add ruff pre-commit hooks"
[INFO] Initializing environment for [...]/ruff-pre-commit.
[INFO] Installing environment for [...]/ruff-pre-commit.
[INFO] Once installed this environment will be reused.
[INFO] This may take a few minutes...
check toml...........................(no files to check)Skipped
check yaml...........................................Passed
fix end of files.....................................Passed
trim trailing whitespace.............................Passed
ruff.................................(no files to check)Skipped
ruff-format..........................(no files to check)Skipped
```

Testing the `ruff` pre-commit hooks

Create a new file called `example.py` with the following contents, and try to commit it:

```
import re

def my_function(a):
    """My function."""
    pass
```

The commit fails

The commit fails

No TOML or YAML files to check:

```
check toml.....................(no files to check)Skipped
check yaml.....................(no files to check)Skipped
fix end of files.....................................Passed
trim trailing whitespace.....................................Passed
ruff-check.....................................Failed
- hook id: ruff-check
- exit code: 1
- files were modified by this hook

Fixed 1 error:
- example.py:
    1 × F401 (unused-import)

Found 1 error (1 fixed, 0 remaining).

ruff-format.....................................Failed
```

The commit fails

No issues with these checks:

```
check toml.....................(no files to check)Skipped
check yaml.....................(no files to check)Skipped
fix end of files.....................................Passed
trim trailing whitespace.............................Passed
ruff-check...........................................Failed
- hook id: ruff-check
- exit code: 1
- files were modified by this hook

Fixed 1 error:
- example.py:
    1 × F401 (unused-import)

Found 1 error (1 fixed, 0 remaining).

ruff-format..........................................Failed
```

The commit fails

`ruff` modified a file, which is an automatic failure:

```
check toml.....(no files to check)Skipped
check yaml.....(no files to check)Skipped
fix end of files.....Passed
trim trailing whitespace.....Passed
ruff-check.....Failed
- hook id: ruff-check
- exit code: 1
- files were modified by this hook

Fixed 1 error:
- example.py:
    1 × F401 (unused-import)

Found 1 error (1 fixed, 0 remaining).

ruff-format.....Failed
- hook id: ruff-format
- files were modified by this hook
```

The commit fails

`ruff` found an unused import when linting:

```
fix end of files.....................................Passed
trim trailing whitespace.............................Passed
ruff-check...........................................Failed
- hook id: ruff-check
- exit code: 1
- files were modified by this hook

Fixed 1 error:
- example.py:
    1 × F401 (unused-import)

Found 1 error (1 fixed, 0 remaining).

ruff-format..........................................Failed
- hook id: ruff-format
- files were modified by this hook
```

The commit fails

`ruff` summarizes its findings (sometimes we have to fix it):

```
ruff-check.....................................Failed
- hook id: ruff-check
- exit code: 1
- files were modified by this hook

Fixed 1 error:
- example.py:
    1 × F401 (unused-import)

Found 1 error (1 fixed, 0 remaining).

ruff-format....................................Failed
- hook id: ruff-format
- files were modified by this hook

1 file reformatted
```

The commit fails

`ruff` also made a change while formatting the file:

```
ruff-check.....................................Failed
- hook id: ruff-check
- exit code: 1
- files were modified by this hook

Fixed 1 error:
- example.py:
    1 × F401 (unused-import)

Found 1 error (1 fixed, 0 remaining).

ruff-format....................................Failed
- hook id: ruff-format
- files were modified by this hook

1 file reformatted
```

Verify the changes before trying again

```
$ git diff example.py
diff --git a/example.py b/example.py
index ebb29eb..ad4c89e 100644
--- a/example.py
+++ b/example.py
@@ -1,5 +1,3 @@
-import re
-
def my_function(a):
    """My function."""
    pass
```

The commit succeeds now

```
$ git add example.py
$ git commit -m "Add example.py"
check toml...........................(no files to check)Skipped
check yaml...........................(no files to check)Skipped
fix end of files.....................................Passed
trim trailing whitespace.............................Passed
ruff.................................................Passed
ruff-format..........................................Passed
[example 79b0f28] Add example.py
 1 file changed, 3 insertions(+)
 create mode 100644 example.py
```

Exercise

Look at the setup information for the `numpydoc-validation` hook at numpydoc.readthedocs.io/en/stable/validation.html, and add it to your configuration. Be sure to commit your changes.

Tip: You can run the following to check for any formatting errors:

```
$ pre-commit validate-config .pre-commit-config.yaml
```

Example solution

Adding the `numpydoc-validation` hook:

```yaml
- repo: https://github.com/numpy/numpydoc
  rev: v1.8.0
  hooks:
    - id: numpydoc-validation
```

Running hooks on demand

Running hooks on demand

Run all hooks on staged changes:

```
$ pre-commit run
$ pre-commit run --files example.py
$ pre-commit run numpydoc-validation --all-files
```

Running hooks on demand

Run all hooks on `example.py`:

```
$ pre-commit run
$ pre-commit run --files example.py
$ pre-commit run numpydoc-validation --all-files
```

Running hooks on demand

Run only the `numpydoc-validation` hook on all files:

```
$ pre-commit run
$ pre-commit run --files example.py
$ pre-commit run numpydoc-validation --all-files
```

Validating docstrings

Validating docstrings

Run `numpydoc-validation` hook on `example.py`:

```
$ pre-commit run numpydoc-validation --files example.py
numpydoc-validation.....................................Failed
- hook id: numpydoc-validation
- exit code: 1

+----------------------+-------+-----------------------------+
| item                 | check | description                 |
+======================+=======+=============================+
| example              | GL08  | The object missing docstring |
+----------------------+-------+-----------------------------+
| example.my_function | ES01  | No extended summary found    |
+----------------------+-------+-----------------------------+
| example.my_function | PR01  | Param {'a'} not documented   |
+----------------------+-------+-----------------------------+
| example.my_function | SA01  | See Also section not found   |
+----------------------+-------+-----------------------------+
```

Output abbreviated.

Validating docstrings

`example.py` fails the docstring checks:

```
$ pre-commit run numpydoc-validation --files example.py
numpydoc-validation.....................................Failed
- hook id: numpydoc-validation
- exit code: 1

+----------------------+-------+------------------------------+
| item                 | check | description                  |
+======================+=======+==============================+
| example              | GL08  | The object missing docstring |
+----------------------+-------+------------------------------+
| example.my_function | ES01  | No extended summary found    |
+----------------------+-------+------------------------------+
| example.my_function | PR01  | Param {'a'} not documented   |
+----------------------+-------+------------------------------+
| example.my_function | SA01  | See Also section not found   |
+----------------------+-------+------------------------------+
```

Output abbreviated.

Validating docstrings

Without any additional configuration, all numpydoc checks are run:

```
- hook id: numpydoc-validation
- exit code: 1

+---------------------+-------+-----------------------------+
| item                | check | description                 |
+=====================+=======+=============================+
| example             | GL08  | The object missing docstring |
+---------------------+-------+-----------------------------+
| example.my_function | ES01  | No extended summary found    |
+---------------------+-------+-----------------------------+
| example.my_function | PR01  | Param {'a'} not documented   |
+---------------------+-------+-----------------------------+
| example.my_function | SA01  | See Also section not found   |
+---------------------+-------+-----------------------------+
| example.my_function | EX01  | No examples section found    |
+---------------------+-------+-----------------------------+
```

Output abbreviated.

Modifying hook behavior

Two options:

Modifying hook behavior

Two options:

- via command line arguments

Modifying hook behavior

Two options:

- via command line arguments
- via a configuration file

Modifying hook behavior with command line arguments

For behavior that you only want to happen when the tool is run as a pre-commit hook (we want the issues fixed automatically here, but perhaps when we run `ruff` on our own, we just want it to report on what it would have changed):

```yaml
- repo: https://github.com/astral-sh/ruff-pre-commit
  rev: v0.11.12
  hooks:
    - id: ruff-check
      args: [--fix, --exit-non-zero-on-fix, --show-fixes]
    - id: ruff-format
```

Modifying hook behavior with a configuration file

For settings that should be applied for every invocation of the tool:

```toml
# pyproject.toml (the tool must support this file)
[tool.ruff]
line-length = 88

[tool.ruff.format]
indent-style = "space"
quote-style = "single"

[tool.ruff.lint]
select = [
    "B", # flake8-bugbear rules
    "E", # pycodestyle error rules
    "F", # pyflakes rules
    "I", # isort rules
    "W", # pycodestyle warning rules
]
```

Modifying hook behavior with a configuration file

For settings that should be applied for every invocation of the tool:

```toml
# pyproject.toml (the tool must support this file)
[tool.ruff]
line-length = 88

[tool.ruff.format]
indent-style = "space"
quote-style = "single"

[tool.ruff.lint]
select = [
    "B", # flake8-bugbear rules
    "E", # pycodestyle error rules
    "F", # pyflakes rules
    "I", # isort rules
    "W", # pycodestyle warning rules
]
```

Modifying hook behavior with a configuration file

For settings that should be applied for every invocation of the tool:

```
[tool.ruff.format]
indent-style = "space"
quote-style = "single"

[tool.ruff.lint]
select = [
    "B", # flake8-bugbear rules
    "E", # pycodestyle error rules
    "F", # pyflakes rules
    "I", # isort rules
    "W", # pycodestyle warning rules
]
ignore = [
    "E501", # line-too-long
]
```

Exercise

Configure the `numpydoc-validation` hook to ignore the checks ES01, EX01, SA01, and SS06. Then, address any issues in `example.py` so that it passes the checks.

numpydoc.readthedocs.io/en/stable/validation.html

Example solution

First, specify which checks to report in the configuration file:

```
# pyproject.toml (this tool supports pyproject.toml)
[tool.numpydoc_validation]
checks = [
    "all",  # report on all checks
    "ES01", # but don't require an extended summary
    "EX01", # or examples
    "SA01", # or a see also section
    "SS06", # and don't require the summary to fit on one line
]
```

Then, update the docstrings in `example.py`:

```
"""Utility functions."""


def my_function(a):
    """
    My function.

    Parameters
    ----------
    a : int
        The value to use.
    """
    pass
```

Excluding files

The docstring checks are strict, but things like the test suite won't end up in our documentation, so we are creating extra work. Let's restrict when the docstring checks will run:

```
  - repo: https://github.com/numpy/numpydoc
    rev: v1.8.0
    hooks:
      - id: numpydoc-validation
+       exclude: (tests|docs)/.*
```

Excluding files

The docstring checks are strict, but things like the test suite won't end up in our documentation, so we are creating extra work. Let's restrict when the docstring checks will run:

```
  - repo: https://github.com/numpy/numpydoc
    rev: v1.8.0
    hooks:
      - id: numpydoc-validation
+       exclude: (tests|docs)/.*
```

Tip: Inclusive filtering on the file name is also supported with `files`. File type filtering is also available. See the pre-commit documentation for all supported options.

Keeping hooks up-to-date

Run `pre-commit autoupdate` to update all hooks to their latest versions (the `rev` key in the YAML):

```
$ pre-commit autoupdate
[...]/pre-commit-hooks ... already up to date!
[...]/ruff-pre-commit ... updating v0.11.12 -> v0.11.13
[...]/numpy/numpydoc ... already up to date!
```

Keeping hooks up-to-date

Run `pre-commit autoupdate` to update all hooks to their latest versions (the `rev` key in the YAML):

```
$ pre-commit autoupdate
[...]/pre-commit-hooks ... already up to date!
[...]/ruff-pre-commit ... updating v0.11.12 -> v0.11.13
[...]/numpy/numpydoc ... already up to date!
```

Warning: Make sure these new versions work with your setup by running `pre-commit run --all-files`.

Good to know

Good to know

- Pass `--no-verify` when committing to bypass the checks.

Good to know

- Pass `--no-verify` when committing to bypass the checks.
- Hooks can be configured to run at different Git stages (*e.g.*, pre-push): pre-commit.com/#config-stages.

Good to know

- Pass `--no-verify` when committing to bypass the checks.
- Hooks can be configured to run at different Git stages (*e.g.*, pre-push): pre-commit.com/#config-stages.
- Check out pre-commit.com/hooks.html for tips on finding hooks.

Good to know

- Pass `--no-verify` when committing to bypass the checks.
- Hooks can be configured to run at different Git stages (*e.g.*, pre-push): pre-commit.com/#config-stages.
- Check out pre-commit.com/hooks.html for tips on finding hooks.
- Use `pre-commit run --all-files` to run your hooks in CI/CD workflows (or you can use pre-commit.ci).

Creating a Pre-Commit Hook

Recipe

Recipe

1. Implement the check(s) in a function.

Recipe

1. Implement the check(s) in a function.
2. Wrap the function in a CLI.

Recipe

1. Implement the check(s) in a function.
2. Wrap the function in a CLI.
3. Make it installable.

Recipe

1. Implement the check(s) in a function.
2. Wrap the function in a CLI.
3. Make it installable.
4. Configure it as a hook.

1. Implement the check(s) in a function

```python
from typing import Sequence

def perform_check(filenames: Sequence[str]) -> int:
    """
    Given a sequence of filenames,
    perform the check(s),
    return 1 if there is a failure, 0 otherwise.
    """
    failure = ?  # your logic

    return 1 if failure else 0
```

1. Implement the check(s) in a function

```python
from typing import Sequence

def perform_check(filenames: Sequence[str]) -> int:
    """
    Given a sequence of filenames,
    perform the check(s),
    return 1 if there is a failure, 0 otherwise.
    """
    failure = ?  # your logic

    return 1 if failure else 0
```

1. Implement the check(s) in a function

```python
from typing import Sequence

def perform_check(filenames: Sequence[str]) -> int:
    """
    Given a sequence of filenames,
    perform the check(s),
    return 1 if there is a failure, 0 otherwise.
    """
    failure = ?  # your logic

    return 1 if failure else 0
```

1. Implement the check(s) in a function

```python
from typing import Sequence

def perform_check(filenames: Sequence[str]) -> int:
    """
    Given a sequence of filenames,
    perform the check(s),
    return 1 if there is a failure, 0 otherwise.
    """
    failure = ?  # your logic

    return 1 if failure else 0
```

Toy example

Only allow committing one file at a time:

```python
from typing import Sequence

def perform_check(filenames: Sequence[str]) -> int:
    """
    Given a sequence of filenames,
    check the number of files,
    return 1 if there is a failure, 0 otherwise.
    """
    failure = len(filenames) > 1

    return 1 if failure else 0
```

Simplified example from `exif-stripper`

This hook removes metadata from images, and the check is implemented to run on a single file at once.

```python
from PIL import Image, UnidentifiedImageError

def process_image(filename: str) -> bool:  # given a file
    has_changed = False
    try:
        with Image.open(filename) as im:
            if exif := im.getexif():  # run check
                exif.clear()  # edit file to fix the issue
                im.save(filename)
                has_changed = True
                print(f'Stripped metadata from {filename}')
    except (FileNotFoundError, UnidentifiedImageError):
        pass  # not an image

    return has_changed  # report result
```

Simplified to only strip EXIF data. *Source: stefmolin/exif-stripper*

Simplified example from `exif-stripper`

This hook removes metadata from images, and the check is implemented to run on a single file at once.

```python
from PIL import Image, UnidentifiedImageError

def process_image(filename: str) -> bool:  # given a file
    has_changed = False
    try:
        with Image.open(filename) as im:
            if exif := im.getexif():  # run check
                exif.clear()  # edit file to fix the issue
                im.save(filename)
                has_changed = True
                print(f'Stripped metadata from {filename}')
    except (FileNotFoundError, UnidentifiedImageError):
        pass  # not an image

    return has_changed  # report result
```

Simplified to only strip EXIF data. *Source: stefmolin/exif-stripper*

Simplified example from `exif-stripper`

This hook removes metadata from images, and the check is implemented to run on a single file at once.

```python
from PIL import Image, UnidentifiedImageError

def process_image(filename: str) -> bool:  # given a file
    has_changed = False
    try:
        with Image.open(filename) as im:
            if exif := im.getexif():  # run check
                exif.clear()  # edit file to fix the issue
                im.save(filename)
                has_changed = True
                print(f'Stripped metadata from {filename}')
    except (FileNotFoundError, UnidentifiedImageError):
        pass  # not an image

    return has_changed  # report result
```

Simplified to only strip EXIF data. *Source: stefmolin/exif-stripper*

Simplified example from `exif-stripper`

This hook removes metadata from images, and the check is implemented to run on a single file at once.

```python
from PIL import Image, UnidentifiedImageError

def process_image(filename: str) -> bool:  # given a file
    has_changed = False
    try:
        with Image.open(filename) as im:
            if exif := im.getexif():  # run check
                exif.clear()  # edit file to fix the issue
                im.save(filename)
                has_changed = True
                print(f'Stripped metadata from {filename}')
    except (FileNotFoundError, UnidentifiedImageError):
        pass  # not an image

    return has_changed  # report result
```

Simplified to only strip EXIF data. *Source: stefmolin/exif-stripper*

What makes a helpful hook?

What makes a helpful hook?

1. Runs quickly

What makes a helpful hook?

1. Runs quickly
2. Tells you what is wrong and where the issue is (the file and, potentially, line number)

What makes a helpful hook?

1. Runs quickly
2. Tells you what is wrong and where the issue is (the file and, potentially, line number)
3. Fixes the file for you, if possible, or, if not, guides you to the fix

Exercise

Come up with your own check. It doesn't matter if something already exists for your idea. If you can't think of anything, try creating a check that enforces a file naming convention that you follow.

Write your code in the `src/custom_hook/check.py` file, renaming both the directory and file as you see fit.

Example solution

Require filenames to be at least three characters (without the extension):

```python
from pathlib import Path

def validate_filename(filename: str, min_len: int = 3) -> int:
    # extract the name so that `/my/repo/x.py` becomes `x`
    name = Path(filename).stem

    # check the length
    if failure := len(name) < min_len:
        print(f'Name too short ({min_len=}): {filename}')

    # convert to an exit code for later
    return int(failure)
```

Contents of `src/filename_validation/validate_filename.py`

Example solution

Require filenames to be at least three characters (without the extension):

```python
from pathlib import Path

def validate_filename(filename: str, min_len: int = 3) -> int:
    # extract the name so that `/my/repo/x.py` becomes `x`
    name = Path(filename).stem

    # check the length
    if failure := len(name) < min_len:
        print(f'Name too short ({min_len=}): {filename}')

    # convert to an exit code for later
    return int(failure)
```

Contents of `src/filename_validation/validate_filename.py`

Example solution

Require filenames to be at least three characters (without the extension):

```python
from pathlib import Path

def validate_filename(filename: str, min_len: int = 3) -> int:
    # extract the name so that `/my/repo/x.py` becomes `x`
    name = Path(filename).stem

    # check the length
    if failure := len(name) < min_len:
        print(f'Name too short ({min_len=}): {filename}')

    # convert to an exit code for later
    return int(failure)
```

Contents of `src/filename_validation/validate_filename.py`

Example solution

Require filenames to be at least three characters (without the extension):

```python
from pathlib import Path

def validate_filename(filename: str, min_len: int = 3) -> int:
    # extract the name so that `/my/repo/x.py` becomes `x`
    name = Path(filename).stem

    # check the length
    if failure := len(name) < min_len:
        print(f'Name too short ({min_len=}): {filename}')

    # convert to an exit code for later
    return int(failure)
```

Contents of `src/filename_validation/validate_filename.py`

Example solution

Require filenames to be at least three characters (without the extension):

```python
from pathlib import Path

def validate_filename(filename: str, min_len: int = 3) -> int:
    # extract the name so that `/my/repo/x.py` becomes `x`
    name = Path(filename).stem

    # check the length
    if failure := len(name) < min_len:
        print(f'Name too short ({min_len=}): {filename}')

    # convert to an exit code for later
    return int(failure)
```

Contents of `src/filename_validation/validate_filename.py`

2. Wrap your function in a CLI

2. Wrap your function in a CLI

We will use `argparse` for this example:

```python
import argparse
from typing import Sequence

def main(argv: Sequence[str] | None = None) -> int:
    parser = argparse.ArgumentParser(prog='your-hook')
    parser.add_argument(
        'filenames',
        nargs='*',
        help='Filenames to process.',
    )

    args = parser.parse_args(argv)

    failures = ?  # run your check(s) on `args.filenames`
    return 1 if failures else 0  # must be 0 or 1 this time
```

2. Wrap your function in a CLI

At a minimum, we need to accept filenames:

```python
import argparse
from typing import Sequence

def main(argv: Sequence[str] | None = None) -> int:
    parser = argparse.ArgumentParser(prog='your-hook')
    parser.add_argument(
        'filenames',
        nargs='*',
        help='Filenames to process.',
    )

    args = parser.parse_args(argv)

    failures = ?  # run your check(s) on `args.filenames`
    return 1 if failures else 0  # must be 0 or 1 this time
```

2. Wrap your function in a CLI

We can then parse the received arguments:

```python
import argparse
from typing import Sequence

def main(argv: Sequence[str] | None = None) -> int:
    parser = argparse.ArgumentParser(prog='your-hook')
    parser.add_argument(
        'filenames',
        nargs='*',
        help='Filenames to process.',
    )

    args = parser.parse_args(argv)

    failures = ?  # run your check(s) on `args.filenames`
    return 1 if failures else 0  # must be 0 or 1 this time
```

2. Wrap your function in a CLI

Run your check(s) on each item in `args.filenames`:

```python
import argparse
from typing import Sequence

def main(argv: Sequence[str] | None = None) -> int:
    parser = argparse.ArgumentParser(prog='your-hook')
    parser.add_argument(
        'filenames',
        nargs='*',
        help='Filenames to process.',
    )

    args = parser.parse_args(argv)

    failures = ?  # run your check(s) on `args.filenames`
    return 1 if failures else 0  # must be 0 or 1 this time
```

2. Wrap your function in a CLI

Return 1 if there were any failures, 0 otherwise:

```python
import argparse
from typing import Sequence

def main(argv: Sequence[str] | None = None) -> int:
    parser = argparse.ArgumentParser(prog='your-hook')
    parser.add_argument(
        'filenames',
        nargs='*',
        help='Filenames to process.',
    )

    args = parser.parse_args(argv)

    failures = ?  # run your check(s) on `args.filenames`
    return 1 if failures else 0  # must be 0 or 1 this time
```

Simplified example from `exif-stripper`

Simplified example from `exif-stripper`

Create the `ArgumentParser`:

```
import argparse
from typing import Sequence

def main(argv: Sequence[str] | None = None) -> int:
    parser = argparse.ArgumentParser(prog='exif-stripper')
    parser.add_argument(
        'filenames',
        nargs='*',
        help='Filenames to process.',
    )

    args = parser.parse_args(argv)

    results = [
        process_image(filename)
        for filename in args.filenames
```

Source: stefmolin/exif-stripper

Simplified example from `exif-stripper`

Parse the received arguments:

```python
import argparse
from typing import Sequence

def main(argv: Sequence[str] | None = None) -> int:
    parser = argparse.ArgumentParser(prog='exif-stripper')
    parser.add_argument(
        'filenames',
        nargs='*',
        help='Filenames to process.',
    )

    args = parser.parse_args(argv)

    results = [
        process_image(filename)
        for filename in args.filenames
```

Source: stefmolin/exif-stripper

Simplified example from `exif-stripper`

Process each file, storing the outcomes (these are Boolean values):

```python
def main(argv: Sequence[str] | None = None) -> int:
    parser = argparse.ArgumentParser(prog='exif-stripper')
    parser.add_argument(
        'filenames',
        nargs='*',
        help='Filenames to process.',
    )

    args = parser.parse_args(argv)

    results = [
        process_image(filename)
        for filename in args.filenames
    ]
    return int(any(results))
```

Source: stefmolin/exif-stripper

Simplified example from `exif-stripper`

Aggregate the results, returning 1 if there were any failures, 0 otherwise:

```python
def main(argv: Sequence[str] | None = None) -> int:
    parser = argparse.ArgumentParser(prog='exif-stripper')
    parser.add_argument(
        'filenames',
        nargs='*',
        help='Filenames to process.',
    )

    args = parser.parse_args(argv)

    results = [
        process_image(filename)
        for filename in args.filenames
    ]
    return int(any(results))
```

Source: stefmolin/exif-stripper

Exercise

Create a CLI for the check function you created in the previous exercise. Add an optional command line argument to modify how your check behaves.

You can put this in the same file from the previous exercise or in another file in the same directory (*e.g.*, `src/custom_hook/cli.py`).

Example solution

Example solution

Here, we continue with the hook requiring filenames of a certain length:

```python
import argparse
from typing import Sequence

from .validate_filename import validate_filename

def main(argv: Sequence[str] | None = None) -> int:
    parser = argparse.ArgumentParser(
        prog='validate-filename'
    )
    parser.add_argument(
        'filenames',
        nargs='*',
        help='Filenames to process.',
    )
    parser.add_argument(
        '--min-len',
```

Contents of `src/filename_validation/cli.py`

Example solution

Most of this is identical to the example:

```python
import argparse
from typing import Sequence

from .validate_filename import validate_filename

def main(argv: Sequence[str] | None = None) -> int:
    parser = argparse.ArgumentParser(
        prog='validate-filename'
    )
    parser.add_argument(
        'filenames',
        nargs='*',
        help='Filenames to process.',
    )
    parser.add_argument(
        '--min-len',
```

Contents of `src/filename_validation/cli.py`

Example solution

But, this time we add the minimum length as an optional argument:

```
    parser.add_argument(
        'filenames',
        nargs='*',
        help='Filenames to process.',
    )
    parser.add_argument(
        '--min-len',
        default=3,
        type=int,
        help='Minimum length for a filename.',
    )

    args = parser.parse_args(argv)

    results = [
        validate_filename(filename, args.min_len)
        for filename in args.filenames
```

Contents of `src/filename_validation/cli.py`

Example solution

When we run the checks on each file, we also pass it in:

```
        help='Filenames to process.',
    )
    parser.add_argument(
        '--min-len',
        default=3,
        type=int,
        help='Minimum length for a filename.',
    )

    args = parser.parse_args(argv)

    results = [
        validate_filename(filename, args.min_len)
        for filename in args.filenames
    ]
    return int(any(results))
```

Contents of `src/filename_validation/cli.py`

3. Make it installable

3. Make it installable

- It can be a package or standalone script.

3. Make it installable

- It can be a package or standalone script.
- You must state all dependencies.

3. Make it installable

- It can be a package or standalone script.
- You must state all dependencies.
- Installation should create an executable.

Package example with `exif-stripper`

Package example with `exif-stripper`

Let's look at the file structure for a Python package:

```
exif-stripper
├── .pre-commit-config.yaml
├── .pre-commit-hooks.yaml
├── LICENSE
├── README.md
├── pyproject.toml
├── src
│   └── exif_stripper
│       ├── __init__.py
│       └── cli.py
└── tests
    └── test_cli.py
```

Based on stefmolin/exif-stripper

Package example with `exif-stripper`

For this example, the repository name is `exif-stripper`:

```
exif-stripper
├── .pre-commit-config.yaml
├── .pre-commit-hooks.yaml
├── LICENSE
├── README.md
├── pyproject.toml
├── src
│   └── exif_stripper
│       ├── __init__.py
│       └── cli.py
└── tests
    └── test_cli.py
```

Based on stefmolin/exif-stripper

Package example with `exif-stripper`

Configuration file to have `pre-commit` run hooks on the code:

```
exif-stripper
├── .pre-commit-config.yaml
├── .pre-commit-hooks.yaml
├── LICENSE
├── README.md
├── pyproject.toml
├── src
│   └── exif_stripper
│       ├── __init__.py
│       └── cli.py
└── tests
    └── test_cli.py
```

Based on stefmolin/exif-stripper

Package example with `exif-stripper`

Configuration file for the hook (for the `strip-exif` hook, here):

```
exif-stripper
├── .pre-commit-config.yaml
├── .pre-commit-hooks.yaml
├── LICENSE
├── README.md
├── pyproject.toml
├── src
│   └── exif_stripper
│       ├── __init__.py
│       └── cli.py
└── tests
    └── test_cli.py
```

Based on stefmolin/exif-stripper

Package example with `exif-stripper`

Important files for any codebase:

```
exif-stripper
├── .pre-commit-config.yaml
├── .pre-commit-hooks.yaml
├── LICENSE
├── README.md
├── pyproject.toml
├── src
│   └── exif_stripper
│       ├── __init__.py
│       └── cli.py
└── tests
    └── test_cli.py
```

Based on stefmolin/exif-stripper

Package example with `exif-stripper`

This package uses the `src` layout:

```
exif-stripper
├── .pre-commit-config.yaml
├── .pre-commit-hooks.yaml
├── LICENSE
├── README.md
├── pyproject.toml
├── src
│   └── exif_stripper
│       ├── __init__.py
│       └── cli.py
└── tests
    └── test_cli.py
```

Based on stefmolin/exif-stripper

Package example with `exif-stripper`

Tests for the logic to make this easier to maintain:

```
exif-stripper
├── .pre-commit-config.yaml
├── .pre-commit-hooks.yaml
├── LICENSE
├── README.md
├── pyproject.toml
├── src
│   └── exif_stripper
│       ├── __init__.py
│       └── cli.py
└── tests
    └── test_cli.py
```

Based on stefmolin/exif-stripper

Package example with `exif-stripper`

The `pyproject.toml` file is used to install this:

```
exif-stripper
├── .pre-commit-config.yaml
├── .pre-commit-hooks.yaml
├── LICENSE
├── README.md
├── pyproject.toml
├── src
│   └── exif_stripper
│       ├── __init__.py
│       └── cli.py
└── tests
    └── test_cli.py
```

Based on stefmolin/exif-stripper

`pyproject.toml`

`pyproject.toml`

This is a template `pyproject.toml` file:

```toml
[project]
name = "custom-hook"
version = "0.1.0"
authors = [{name = "TODO", email = "TODO@TODO"},]
description = "A pre-commit hook created during the workshop."
readme = "README.md"

requires-python = ">=3.10"
dependencies = []  # TODO: declare any dependencies

# TODO: update `your-script-name` with the name of your choice
# TODO: update `pkg.module:function` to point to your function
scripts.your-script-name = "pkg.module:function"

[dependency-groups]
dev = [
```

`pyproject.toml`

Name your distribution (likely the name of your repository):

```toml
[project]
name = "custom-hook"
version = "0.1.0"
authors = [{name = "TODO", email = "TODO@TODO"},]
description = "A pre-commit hook created during the workshop."
readme = "README.md"

requires-python = ">=3.10"
dependencies = []  # TODO: declare any dependencies

# TODO: update `your-script-name` with the name of your choice
# TODO: update `pkg.module:function` to point to your function
scripts.your-script-name = "pkg.module:function"

[dependency-groups]
dev = [
```

`pyproject.toml`

Provide a version (can also be done dynamically):

```toml
[project]
name = "custom-hook"
version = "0.1.0"
authors = [{name = "TODO", email = "TODO@TODO"},]
description = "A pre-commit hook created during the workshop."
readme = "README.md"

requires-python = ">=3.10"
dependencies = []  # TODO: declare any dependencies

# TODO: update `your-script-name` with the name of your choice
# TODO: update `pkg.module:function` to point to your function
scripts.your-script-name = "pkg.module:function"

[dependency-groups]
dev = [
```

`pyproject.toml`

Provide information about your project:

```toml
[project]
name = "custom-hook"
version = "0.1.0"
authors = [{name = "TODO", email = "TODO@TODO"},]
description = "A pre-commit hook created during the workshop."
readme = "README.md"

requires-python = ">=3.10"
dependencies = []  # TODO: declare any dependencies

# TODO: update `your-script-name` with the name of your choice
# TODO: update `pkg.module:function` to point to your function
scripts.your-script-name = "pkg.module:function"

[dependency-groups]
dev = [
```

`pyproject.toml`

Include all dependencies required to use your hook here:

```toml
[project]
name = "custom-hook"
version = "0.1.0"
authors = [{name = "TODO", email = "TODO@TODO"},]
description = "A pre-commit hook created during the workshop."
readme = "README.md"

requires-python = ">=3.10"
dependencies = []  # TODO: declare any dependencies

# TODO: update `your-script-name` with the name of your choice
# TODO: update `pkg.module:function` to point to your function
scripts.your-script-name = "pkg.module:function"

[dependency-groups]
dev = [
    "pre-commit",  # so you can run hooks on the codebase
```

`pyproject.toml`

Optional dependencies for development of the hook go here, instead:

```
requires-python = ">=3.10"
dependencies = []  # TODO: declare any dependencies

# TODO: update `your-script-name` with the name of your choice
# TODO: update `pkg.module:function` to point to your function
scripts.your-script-name = "pkg.module:function"

[dependency-groups]
dev = [
    "pre-commit",  # so you can run hooks on the codebase
    # "pytest",  # remember to add tests for your hook
]

[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"
```

`pyproject.toml`

Build dependencies will be installed by the build front end (*e.g.*, `pip`):

```
requires-python = ">=3.10"
dependencies = []  # TODO: declare any dependencies

# TODO: update `your-script-name` with the name of your choice
# TODO: update `pkg.module:function` to point to your function
scripts.your-script-name = "pkg.module:function"

[dependency-groups]
dev = [
    "pre-commit",  # so you can run hooks on the codebase
    # "pytest",  # remember to add tests for your hook
]

[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"
```

`pyproject.toml`

Create an executable for calling your hook's CLI:

```
description = "A pre-commit hook created during the workshop."
readme = "README.md"

requires-python = ">=3.10"
dependencies = []  # TODO: declare any dependencies

# TODO: update `your-script-name` with the name of your choice
# TODO: update `pkg.module:function` to point to your function
scripts.your-script-name = "pkg.module:function"

[dependency-groups]
dev = [
    "pre-commit",  # so you can run hooks on the codebase
    # "pytest",  # remember to add tests for your hook
]

[build-system]
```

Example of specifying the entry point

When the `exif_stripper` package is installed, an entry point (executable) called, `exif-stripper`, is created automatically, which, when run, calls the `main()` function in the `exif_stripper.cli` module:

```
scripts.exif-stripper = "exif_stripper.cli:main"
```

Source: stefmolin/exif-stripper

Exercise

Using the provided template, populate the `pyproject.toml` file for your hook from the previous exercise. Be sure to create an entry point that will hit the CLI.

Confirm that you can install it: `pip install -e .` or `uv sync`. Note that you will need to declare any dependencies outside the standard library and create any files referenced in the `pyproject.toml`, such as `README.md`.

Example solution

Replace the existing `[project]` section with the following:

```toml
[project]
name = "filename-validation"
version = "0.1.0"
authors = [
  {name = "Stefanie Molin", email = "email@example.com"},
]
description = "Validates that filenames meet criteria."
readme = "README.md"

requires-python = ">=3.10"
dependencies = []

scripts.validate-filename = "filename_validation.cli:main"
```

Example solution

Replace the existing `[project]` section with the following:

```toml
[project]
name = "filename-validation"
version = "0.1.0"
authors = [
  {name = "Stefanie Molin", email = "email@example.com"},
]
description = "Validates that filenames meet criteria."
readme = "README.md"

requires-python = ">=3.10"
dependencies = []

scripts.validate-filename = "filename_validation.cli:main"
```

Example solution

Replace the existing `[project]` section with the following:

```toml
[project]
name = "filename-validation"
version = "0.1.0"
authors = [
  {name = "Stefanie Molin", email = "email@example.com"},
]
description = "Validates that filenames meet criteria."
readme = "README.md"

requires-python = ">=3.10"
dependencies = []

scripts.validate-filename = "filename_validation.cli:main"
```

Example solution

Replace the existing `[project]` section with the following:

```
[project]
name = "filename-validation"
version = "0.1.0"
authors = [
  {name = "Stefanie Molin", email = "email@example.com"},
]
description = "Validates that filenames meet criteria."
readme = "README.md"

requires-python = ">=3.10"
dependencies = []

scripts.validate-filename = "filename_validation.cli:main"
```

4. Configure it as a hook

4. Configure it as a hook

Configuration goes in `.pre-commit-hooks.yaml` as a list of hooks:

```yaml
- id: strip-exif
  name: strip EXIF metadata
  description: This hook strips image metadata.
  entry: exif-stripper
  language: python
  types: [image]
```

Source: stefmolin/exif-stripper

4. Configure it as a hook

The hook ID (we can use this to run just this hook):

```yaml
- id: strip-exif
  name: strip EXIF metadata
  description: This hook strips image metadata.
  entry: exif-stripper
  language: python
  types: [image]
```

Source: stefmolin/exif-stripper

4. Configure it as a hook

Display name for the hook when the checks are run:

```yaml
- id: strip-exif
  name: strip EXIF metadata
  description: This hook strips image metadata.
  entry: exif-stripper
  language: python
  types: [image]
```

Source: stefmolin/exif-stripper

4. Configure it as a hook

(Optional) Description of what your hook does:

```yaml
- id: strip-exif
  name: strip EXIF metadata
  description: This hook strips image metadata.
  entry: exif-stripper
  language: python
  types: [image]
```

Source: stefmolin/exif-stripper

4. Configure it as a hook

The entry point or executable to run, which can also include arguments:

```yaml
- id: strip-exif
  name: strip EXIF metadata
  description: This hook strips image metadata.
  entry: exif-stripper
  language: python
  types: [image]
```

Source: stefmolin/exif-stripper

4. Configure it as a hook

The language your hook is written in (so `pre-commit` can install it):

```yaml
- id: strip-exif
  name: strip EXIF metadata
  description: This hook strips image metadata.
  entry: exif-stripper
  language: python
  types: [image]
```

Source: stefmolin/exif-stripper

4. Configure it as a hook

(Optional) Restrict this hook to only run on certain file types:

```
- id: strip-exif
  name: strip EXIF metadata
  description: This hook strips image metadata.
  entry: exif-stripper
  language: python
  types: [image]
```

Source: stefmolin/exif-stripper

Exercise

Create a `.pre-commit-hooks.yaml` file for configuring your hook from the previous exercise. Note that, by default, hooks will run in parallel, if you need to run in serial, specify `require_serial: true`. Be sure to consult pre-commit.com/#creating-new-hooks for a listing of the configuration options.

Tip: You can run the following to check for any formatting errors:

```
$ pre-commit validate-manifest .pre-commit-hooks.yaml
```

Example solution

Continuing the example from previous exercises, we can use the following as our `.pre-commit-hooks.yaml` file for the hook to validate the length of the filename:

```yaml
- id: validate-filename
  name: validate-filename
  description: This hook checks filename length.
  entry: validate-filename
  language: python
```

Testing the hook configuration

Testing the hook configuration

- Run `pre-commit try-repo`. (pre-commit.com/#developing-hooks-interactively)

Testing the hook configuration

- Run `pre-commit try-repo`. (pre-commit.com/#developing-hooks-interactively)
- Add your new hook to the `.pre-commit-config.yaml` file in another repository, and set the `rev` value to the most recent commit hash. Then, you can run your hook with `pre-commit run`.

Exercise

Test that your hook is properly configured.

Example solution

Example solution

Start by creating a file that will fail the check:

```
$ touch x.py
$ pre-commit try-repo . --files x.py
[INFO] Initializing environment for ..
==========================================================
Using config:
==========================================================
repos:
-   repo: .
    rev: e11041f74c8a0f074f0633138506dce2efe9c5e7
    hooks:
    -   id: validate-filename
==========================================================
[INFO] Installing environment for ..
[INFO] Once installed this environment will be reused.
[INFO] This may take a few minutes...
validate-filename.....................................Failed
```

Example solution

Run `pre-commit try-repo` passing in the file:

```
$ touch x.py
$ pre-commit try-repo . --files x.py
[INFO] Initializing environment for ..
==========================================================
Using config:
==========================================================
repos:
-   repo: .
    rev: e11041f74c8a0f074f0633138506dce2efe9c5e7
    hooks:
    -   id: validate-filename
==========================================================
[INFO] Installing environment for ..
[INFO] Once installed this environment will be reused.
[INFO] This may take a few minutes...
validate-filename.....................................Failed
```

Example solution

Now, `pre-commit` will generate a configuration for us:

```
$ touch x.py
$ pre-commit try-repo . --files x.py
[INFO] Initializing environment for ..
=====================================================
Using config:
=====================================================
repos:
-   repo: .
    rev: e11041f74c8a0f074f0633138506dce2efe9c5e7
    hooks:
    -   id: validate-filename
=====================================================
[INFO] Installing environment for ..
[INFO] Once installed this environment will be reused.
[INFO] This may take a few minutes...
validate-filename.....................................Failed
```

Example solution

Think of this as the `.pre-commit-config.yaml` file that will be used:

```
$ touch x.py
$ pre-commit try-repo . --files x.py
[INFO] Initializing environment for ..
==========================================================
Using config:
==========================================================
repos:
-   repo: .
    rev: e11041f74c8a0f074f0633138506dce2efe9c5e7
    hooks:
    -   id: validate-filename
==========================================================
[INFO] Installing environment for ..
[INFO] Once installed this environment will be reused.
[INFO] This may take a few minutes...
validate-filename.....................................Failed
- hook id: validate-filename
```

Example solution

The `repo` points to the path we passed to `pre-commit try-repo`:

```
$ touch x.py
$ pre-commit try-repo . --files x.py
[INFO] Initializing environment for ..
=====================================================
Using config:
=====================================================
repos:
-   repo: .
    rev: e11041f74c8a0f074f0633138506dce2efe9c5e7
    hooks:
    -   id: validate-filename
=====================================================
[INFO] Installing environment for ..
[INFO] Once installed this environment will be reused.
[INFO] This may take a few minutes...
validate-filename.....................................Failed
```

Example solution

The `rev` is set to the most recent commit hash:

```
$ touch x.py
$ pre-commit try-repo . --files x.py
[INFO] Initializing environment for ..
==========================================================
Using config:
==========================================================
repos:
-   repo: .
    rev: e11041f74c8a0f074f0633138506dce2efe9c5e7
    hooks:
    -   id: validate-filename
==========================================================
[INFO] Installing environment for ..
[INFO] Once installed this environment will be reused.
[INFO] This may take a few minutes...
validate-filename.....................................Failed
- hook id: validate-filename
```

Example solution

The hooks are pulled out of `.pre-commit-hooks.yaml`:

```
[INFO] Initializing environment for ..
==========================================================
Using config:
==========================================================
repos:
-   repo: .
    rev: e11041f74c8a0f074f0633138506dce2efe9c5e7
    hooks:
    -   id: validate-filename
==========================================================
[INFO] Installing environment for ..
[INFO] Once installed this environment will be reused.
[INFO] This may take a few minutes...
validate-filename.....................................Failed
- hook id: validate-filename
- exit code: 1
```

Example solution

Just like before, `pre-commit` will need to set up the environment:

```
Using config:
=========================================================
repos:
-   repo: .
    rev: e11041f74c8a0f074f0633138506dce2efe9c5e7
    hooks:
    -   id: validate-filename
=========================================================
[INFO] Installing environment for ..
[INFO] Once installed this environment will be reused.
[INFO] This may take a few minutes...
validate-filename.....................................Failed
- hook id: validate-filename
- exit code: 1

Name too short (min_len=3): x.py
```

Example solution

As expected, `x.py` fails the `validate-filename` check:

```
Using config:
===============================================================
repos:
-   repo: .
    rev: e11041f74c8a0f074f0633138506dce2efe9c5e7
    hooks:
    -   id: validate-filename
===============================================================
[INFO] Installing environment for ..
[INFO] Once installed this environment will be reused.
[INFO] This may take a few minutes...
validate-filename.....................................Failed
- hook id: validate-filename
- exit code: 1

Name too short (min_len=3): x.py
```

Maintaining your hook

Maintaining your hook

There are several points of failure when building and maintaining a hook, so it is best to set up testing in layers:

Maintaining your hook

There are several points of failure when building and maintaining a hook, so it is best to set up testing in layers:

- Build a test suite that exercises the underlying logic and the CLI.

Maintaining your hook

There are several points of failure when building and maintaining a hook, so it is best to set up testing in layers:

- Build a test suite that exercises the underlying logic and the CLI.
- Run your test suite on different operating systems and Python versions.

Maintaining your hook

There are several points of failure when building and maintaining a hook, so it is best to set up testing in layers:

- Build a test suite that exercises the underlying logic and the CLI.
- Run your test suite on different operating systems and Python versions.
- Use `pre-commit` to run your hooks in your CI/CD workflows.

Maintaining your hook

There are several points of failure when building and maintaining a hook, so it is best to set up testing in layers:

- Build a test suite that exercises the underlying logic and the CLI.
- Run your test suite on different operating systems and Python versions.
- Use `pre-commit` to run your hooks in your CI/CD workflows.

Tip: See the references slide at the end for some examples.

Document your hook

Make it easy for people to get started with your hook by including a setup snippet in your `README`. If you have command line arguments and/or support configuration files, mention some common configurations. Note that you should create tags so people don't have to use commit hashes.

Usage

Add the following to your `.pre-commit-config.yaml` file:

```yaml
- repo: https://github.com/stefmolin/exif-stripper
  rev: 0.3.0
  hooks:
    - id: strip-exif
```

Source: stefmolin/exif-stripper

Sharing your hook with the world

Sharing your hook with the world

- Once your repository is public on GitHub (or similar), your hook is available for use.

Sharing your hook with the world

- Once your repository is public on GitHub (or similar), your hook is available for use.
- Help users find your hook by talking about it: conference talks, blog posts, tweets, *etc.*

Sharing your hook with the world

- Once your repository is public on GitHub (or similar), your hook is available for use.
- Help users find your hook by talking about it: conference talks, blog posts, tweets, *etc.*
- Consider publishing it to PyPI if it is useful beyond just being a hook. Note that you will need to add additional information to your `pyproject.toml` if you choose to do so.

Good to know

Good to know

- Adjust your hook's behavior by supporting command line arguments or reading settings from a configuration file like `pyproject.toml`.

Good to know

- Adjust your hook's behavior by supporting command line arguments or reading settings from a configuration file like `pyproject.toml`.
- Multiple hooks can be defined in a single repository.

Good to know

- Adjust your hook's behavior by supporting command line arguments or reading settings from a configuration file like `pyproject.toml`.
- Multiple hooks can be defined in a single repository.
- Hooks can run at different stages of the Git workflow (not just the pre-commit stage): pre-commit.com/#confining-hooks-to-run-at-certain-stages.

Good to know

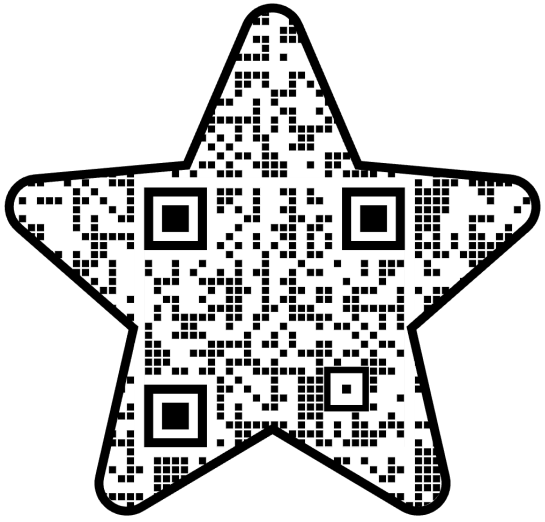
- Adjust your hook's behavior by supporting command line arguments or reading settings from a configuration file like `pyproject.toml`.
- Multiple hooks can be defined in a single repository.
- Hooks can run at different stages of the Git workflow (not just the pre-commit stage): pre-commit.com/#confining-hooks-to-run-at-certain-stages.
- Hooks can be written in any language supported by `pre-commit` (see the list at pre-commit.com/#supported-languages).

References

- Pre-commit articles on my website, including
  - How to Set Up Pre-Commit Hooks
  - Pre-Commit Hook Creation Guide
- Official documentation: pre-commit.com
  - `.pre-commit-config.yaml`
  - `.pre-commit-hooks.yaml`
  - CLI reference
- Example of a published pre-commit hook: `exif-stripper`
  - Test suite using `pytest`: `test_cli.py`
  - Testing workflow with GitHub Actions: `ci.yml`
- Example of finding and parsing configuration files like `pyproject.toml` (from the numpydoc-validation hook)
- Python Packaging User Guide: Publishing package distribution releases using GitHub Actions CI/CD workflows

Thank you!

I hope you enjoyed the workshop. You can follow my work on these platforms:



- stefaniemolin.com
- github.com/stefmolin
- bsky.app/profile/stefaniemolin.com
- x.com/StefanieMolin
- linkedin.com/in/stefanie-molin

This was generated at 2025-07-08T19:07:13.991+00:00. The latest version is at stefaniemolin.com/pre-commit-workshop/.