

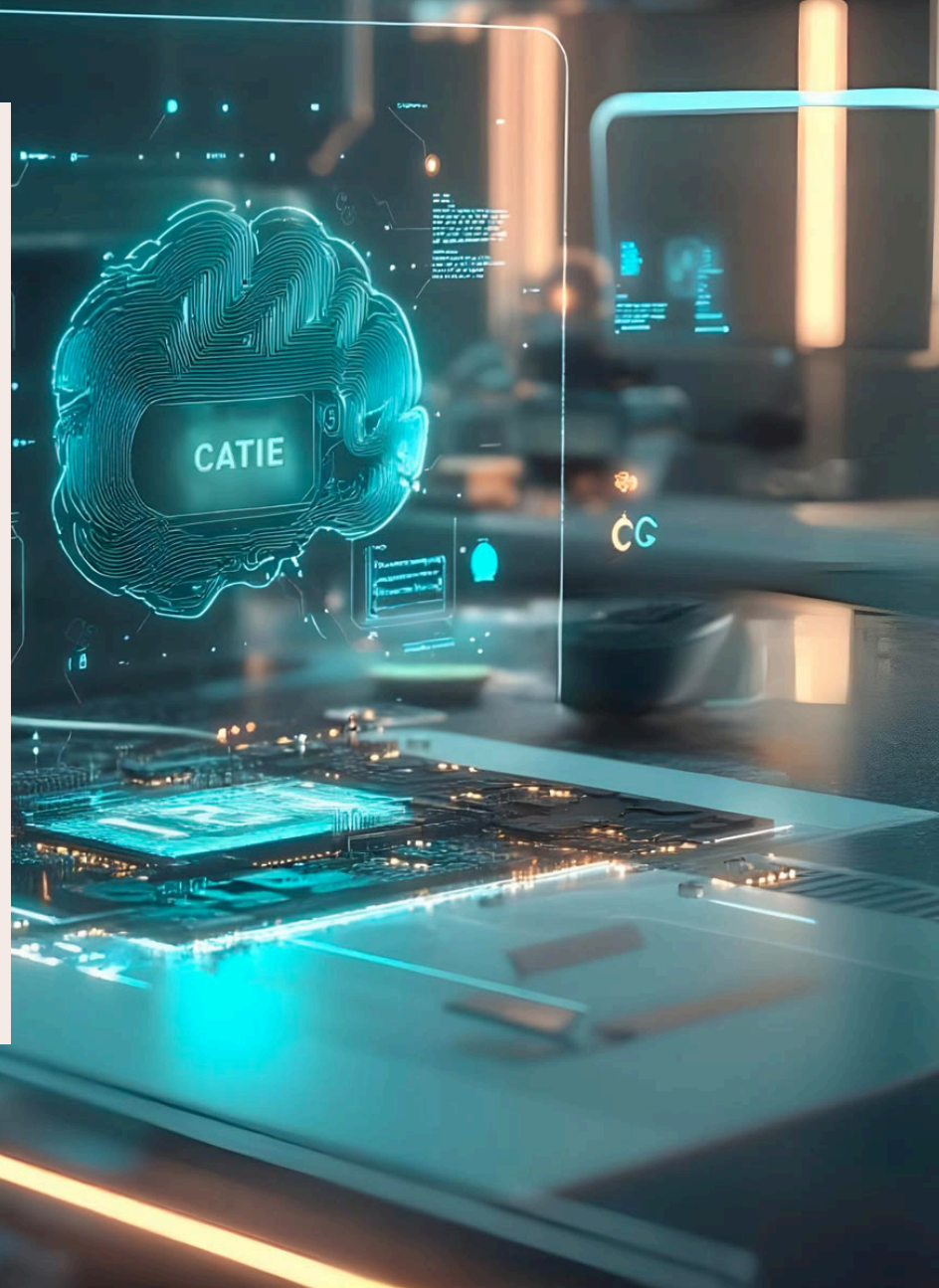


CATIE

Outils d'imagerie pour la
robotique

Projet Python avec uv, tests et qualité

20/10/2026



RÉGION
Nouvelle-
Aquitaine

Aujourd'hui : du script au projet

Le fil rouge fonctionne sur votre machine. Dès qu'on travaille à plusieurs, quatre questions se posent :

Question

Réponse du jour

Que faut-il installer, en quelle version, pour que ça tourne ailleurs ?

uv : projet, dépendances, lock

Après une modification, est-ce que tout marche encore ?

pytest

Le code est-il homogène et sans erreur évidente ?

Ruff : lint et formatage

Qui vérifie tout ça, à chaque fois ?

pre-commit et **CI**

30 min de cours, puis 1h30 de TP : l'étape 3 du fil rouge.



1. Un projet avec uv

Environnement isolé, dépendances déclarées,
versions figées.

Script, module, paquet

Bases

Mot	Ce que c'est	Dans le fil rouge
Script	Un fichier <code>.py</code> qu'on lance	<code>uv run pipeline.py</code>
Module	Un fichier <code>.py</code> qu'on importe	<code>from operations import Brighten</code>
Paquet	Un dossier de modules, avec un <code>__init__.py</code>	<code>from fil_rouge.operations import ...</code>
Distribution	Un paquet emballé avec nom, version et dépendances : il s'installe	<code>numpy, pillow, bientôt fil-rouge</code>

- Un même fichier peut être script et module : c'est le rôle de `if __name__ == "__main__":`
- Comme en électronique : le module est un composant, le paquet une carte qui en regroupe plusieurs, la distribution le produit emballé avec sa fiche technique
- Le nom installé et le nom importé peuvent différer : on installe `pillow`, on importe `PIL` ; `fil-rouge` s'importe `fil_rouge`

Gestionnaire de paquets et PyPI

Bases

- **PyPI** (pypi.org) : le catalogue public des distributions Python, plus de 600 000 projets. Le catalogue d'un distributeur de composants, en somme
- **Gestionnaire de paquets** : l'outil qui, à partir d'un nom, télécharge la distribution **et ses propres dépendances**, choisit des versions compatibles entre elles (la **résolution**), puis installe le tout
- Chaque langage a le sien : **pip** puis **uv** pour Python, **npm** pour JavaScript, **cargo** pour Rust ; **apt** fait de même pour le système

```
$ uv add matplotlib
```

```
Resolved 12 packages
```

```
Installed 11 packages
```

+ contourpy	+ numpy
+ cycler	+ packaging
+ fonttools	+ pillow
+ kiwisolver	+ pyparsing
+ matplotlib	+ python-dateutil
	+ six

Une ligne demandée, onze paquets installés : ce sont les dépendances **transitives**.

Du script au projet

	Script (séances 1 et 2)	Projet (aujourd'hui)
Dépendances	En-tête <code># /// script</code> du fichier	<code>pyproject.toml</code>
Versions	Libres : la plus récente au lancement	Figées dans <code>uv.lock</code> , commité
Lancer	<code>uv run pipeline.py</code>	<code>uv run fil-rouge data output</code>
Outils du développeur	—	Groupe <code>dev</code> : <code>pytest</code> , <code>Ruff</code> ...
Pour quoi	Un outil d'un seul fichier	Du code partagé, testé et livré

Le code ne change presque pas : on ajoute autour de lui ce qu'il faut pour qu'il s'installe et se vérifie pareil sur toutes les machines.

Dépendances et environnement virtuel

- Le fil rouge dépend de NumPy et Pillow, téléchargés depuis PyPI. Sur une autre machine : lesquelles, en quelle version ?
- Installées pour tout le système, deux projets aux versions incompatibles se cassent mutuellement
- **Environnement virtuel** : un dossier `.venv/` par projet, avec son interpréteur et ses paquets. Jamais commité : il se reconstruit à partir des déclarations. Un poste de manip par projet : ce qu'on y branche ne touche pas le poste voisin
- **uv** gère Python, `.venv`, la résolution et le lock. Vous l'avez déjà utilisé en séance 2 (`uv run pipeline.py`)

Commande	Rôle
<code>uv init</code>	Crée le projet
<code>uv add numpy</code>	Déclare et installe une dépendance
<code>uv add --dev pytest</code>	Idem, pour un outil de développement
<code>uv lock</code>	Résout et fige les versions dans <code>uv.lock</code>
<code>uv sync</code>	Rend <code>.venv</code> conforme à <code>uv.lock</code>
<code>uv run <cmd></code>	Synchronise si besoin, puis exécute dans <code>.venv</code>

On n'active jamais le venv à la main : toutes les commandes passent par `uv run`.

Créer le projet

```
uv init --package --name fil-rouge \  
  --python 3.12 etape_3
```

```
etape_3/  
├── .python-version  
├── pyproject.toml  
├── README.md  
└── src/  
    ├── fil_rouge/  
    └── __init__.py
```

- `--package` : projet installable, avec une commande
- `--name` : sinon le projet prendrait le nom du dossier, `etape_3`
- `--python 3.12` : sinon uv prend le Python le plus récent de la machine, et l'image Docker (3.12) ne pourra plus installer le projet
- Disposition `src/` : le code n'est importable qu'une fois installé, les tests exercent donc le paquet tel qu'un utilisateur le verrait

```
[project]  
name = "fil-rouge"  
version = "0.1.0"  
requires-python = ">=3.12"  
dependencies = [  
    "numpy>=2.5.3",  
    "pillow>=12.3.0",  
]
```

```
[project.scripts]  
fil-rouge = "fil_rouge:main"
```

```
[dependency-groups]  
dev = ["pytest>=9.1.1", "ruff>=0.16.9",  
    "pre-commit>=4.6.2"]
```

[project.scripts] : la commande `fil-rouge` appelle `fil_rouge.main()`.

Dépendances, lock et synchronisation

```
uv add numpy pillow          # dependencies
uv add --dev pytest ruff      # dev group
uv run fil-rouge data output
uv run pytest
```

```
uv sync --locked      # CI: fails if the lock
                      # is behind pyproject
uv sync --no-dev      # production
```

- `pyproject.toml` porte des **contraintes** (`numpy>=2.5.3`) ; `uv.lock` les versions **exactes** retenues, dépendances transitives comprises, pour toutes les plateformes. Les deux sont commités
- Le premier est le cahier des charges (« un régulateur 5 V »), le second la nomenclature avec les références exactes, y compris celles des sous-ensembles achetés
- Groupe **dev** : outils du développeur, installés en local, absents du paquet publié
- `uv add` met à jour `pyproject.toml`, `uv.lock` et `.venv` d'un coup. Après une édition à la main : `uv lock`

Même lock, mêmes versions : en local, chez le binôme, en CI et dans l'image Docker de la séance 5.

Ce que uv remplace

Pour aller plus loin

Besoin	Outils historiques	uv
Installer une version de Python	pyenv, paquets système, conda	<code>uv python install 3.12</code>
Isoler les dépendances d'un projet	venv, virtualenv, conda	automatique (<code>.venv</code>)
Déclarer et figer les dépendances	pip + <code>requirements.txt</code> , pip-tools, Poetry, Pipenv	<code>uv add</code> , <code>uv.lock</code>
Installer un outil en ligne de commande	pipx	<code>uv tool</code> , <code>uvx</code>
Script autonome avec dépendances	—	<code>uv run script.py</code> (PEP 723)
Construire et publier un paquet	build, twine, Poetry	<code>uv build</code> , <code>uv publish</code>

- Un seul exécutable Rust (Astral, les auteurs de Ruff), 10 à 100 fois plus rapide que pip
- Python téléchargé sans `sudo` ; `uv python pin 3.12` écrit `.python-version`, commité
- Standards respectés (`pyproject.toml`, PEP 621, PEP 735) : le projet reste utilisable avec pip

Versions et résolution

Pour aller plus loin

Contrainte	Versions acceptées
<code>==2.5.3</code>	exactement 2.5.3
<code>>=2.5</code>	2.5 ou plus récente
<code>>=2.5,<3</code>	au moins 2.5, pas la 3
<code>~=2.5</code>	≥ 2.5 et < 3

MAJEUR.MINEUR.CORRECTIF : un saut de majeure peut casser l'API.

- Résolution : trouver un ensemble de versions qui satisfait toutes les contraintes, les vôtres et celles de vos dépendances, pour toutes les plateformes cibles. Comme choisir des composants quand chaque sous-ensemble impose ses propres tolérances
- Contraindre trop large expose aux ruptures ; trop serré rend la résolution impossible avec d'autres paquets

```
uv tree                                # tree
uv lock --upgrade-package numpy       # upgrade numpy
uv add --group docs mkdocs            # another group
uv remove pillow
```

Outils, scripts, paquets

Pour aller plus loin

Outils en ligne de commande

```
uvx ruff check .      # one-off, no install
uv tool install ruff  # permanent
```

Script autonome (PEP 723, séance 2)

```
uv init --script tool.py
uv add --script tool.py requests
uv run tool.py
```

Construire le paquet

```
uv build
# dist/fil_rouge-0.1.0-py3-none-any.whl
# dist/fil_rouge-0.1.0.tar.gz
```

Venant de Poetry

Poetry	uv
poetry add --group dev	uv add --dev
poetry install	uv sync
poetry run	uv run
poetry.lock	uv.lock

[build-system] dans `pyproject.toml` désigne l'outil qui construit le paquet (`uv_build`). Sans `--package`, `uv init` crée une simple application, sans commande ni paquet à construire.



2. Tester avec pytest

Vérifier automatiquement, à chaque modification, que le code produit le résultat attendu.

pytest : assertions, erreurs, cas multiples

```
# tests/test_operations.py
import numpy as np
import pytest
from fil_rouge.config import Config
from fil_rouge.operations import Brighten

def test_brighten_saturates_at_255():
    image = np.full((2, 2, 3), 200, dtype=np.uint8)
    result = Brighten(2.0)(image)
    assert result.dtype == np.uint8
    assert (result == 255).all()

@pytest.mark.parametrize(
    "field",
    [{"factor": 0}, {"width": -1}, {"quality": 100}],
)
def test_config_rejects_invalid_values(
    tmp_path, field
):
    with pytest.raises(ValueError):
        Config(tmp_path, tmp_path, **field)
```

- Découverte : `tests/test_*.py`, fonctions `test_*`. Un simple `assert`, pytest affiche les valeurs comparées en cas d'échec
- Sans le `np.minimum(..., 255)`, 200×2 déborde en `uint8` et donne 144 : ce test l'attrape
- `pytest.raises` : le test échoue si l'erreur n'est pas levée
- `parametrize` : une fonction, trois cas, trois lignes dans le rapport
- `pytest.approx` pour comparer des flottants
- `-x` : arrêt au premier échec ; `-k config` : filtre par nom

Fixtures et test de bout en bout

```
# tests/test_pipeline.py
@pytest.fixture
def input_dir(tmp_path):
    folder = tmp_path / "input"
    folder.mkdir()
    for i in range(3):
        pixels = np.full((24, 32, 3), 40 * i, dtype=np.uint8)
        Image.fromarray(pixels).save(folder / f"image_{i}.png")
    (folder / "corrupted.png").write_bytes(b"not an image")
    return folder

@pytest.mark.parametrize(
    "executor", ["sequential", "threads", "processes"]
)
def test_full_pipeline(tmp_path, input_dir, executor):
    config = Config(input_dir, tmp_path / "output")
    results = run(config, executor, workers=2)
    assert len(results) == 3          # corrupted one skipped
    ...                               # stats.csv, archive.tar
```

- **Fixture** : prépare un contexte ; pytest l'injecte dans chaque test qui la nomme en paramètre. C'est le montage de test, câblé avant chaque mesure et démonté après
- `tmp_path` (fournie par pytest) : un dossier neuf par test, rien ne fuit d'un test à l'autre
- Bout en bout : le pipeline complet sur de vraies images, pour les trois exécuteurs de la séance 3
- Tester les cas limites : saturation, fichier corrompu, dossier vide

pytest : organiser et mesurer

Pour aller plus loin

- `conftest.py` : fixtures partagées par tous les fichiers de test d'un dossier
- Portée d'une fixture :
`@pytest.fixture(scope="module")` pour préparer une seule fois une ressource coûteuse
- Marqueurs : `@pytest.mark.slow`, puis `uv run pytest -m "not slow"` pour la boucle rapide
- Couverture : quelles lignes les tests exécutent réellement

```
[tool.pytest.ini_options]
testpaths = ["tests"]
markers = ["slow: long-running tests"]

uv add --dev pytest-cov
uv run pytest --cov=fil_rouge \
              --cov-report=term-missing
```

Une couverture élevée ne prouve pas que les assertions sont pertinentes : elle montre surtout ce qui n'est jamais testé.



3. Qualité et CI

Ruff pour le lint et le formatage, pre-commit en local, GitHub Actions à chaque push.

Ruff : lint et formatage

```
[tool.ruff]
line-length = 88

[tool.ruff.lint]
# E/F: basic errors, I: import sorting,
# UP: modern Python syntax, B: likely bugs,
# SIM: simplifications,
# PL: rules taken from pylint.
select = ["E", "F", "I", "UP", "B", "SIM", "PL"]

[tool.ruff.lint.per-file-ignores]
# Hard-coded expected values are normal in a test.
"tests/*" = ["PLR2004"]

uv run ruff format          # layout
uv run ruff check --fix     # lint
```

- **Lint** : analyse statique du code sans l'exécuter (imports inutilisés, variables mortes, bugs probables). **Formatage** : mise en page déterministe
- Remplace flake8, isort, pyupgrade et black, en quelques millisecondes
- Un format unique : plus de débat de style en revue

Sur `config.py` de l'étape 2 :

```
PLR2004 Magic value used in comparison,
consider replacing `95` with a constant
--> config.py:30:37
```

Correction : `MAX_QUALITY = 95`. Chaque règle est documentée : `ruff rule PLR2004`.

Vérifier les types : mypy, ty

- Les annotations de la séance 1 ne sont pas vérifiées à l'exécution : un vérificateur de types les contrôle statiquement
- **mypy** : la référence ; **ty** (Astral, bêta) : beaucoup plus rapide
- **strict** d'emblée sur un projet neuf ; sur un code existant, durcir module par module

```
[tool.mypy]
strict = true
files = ["src"]
```

```
uv add --dev mypy
uv run mypy
uvx ty check
```

```
error: Argument 1 to "Brighten" has
incompatible type "str"; expected "float"
[arg-type]
```

À ajouter en CI et en hook pre-commit, comme Ruff.

pre-commit : vérifier avant chaque commit

```
# .pre-commit-config.yaml
repos:
- repo: https://github.com/pre-commit/pre-commit-hooks
  rev: v6.0.0
  hooks:
    - id: trailing-whitespace
    - id: end-of-file-fixer
    - id: check-yaml
    - id: check-added-large-files
- repo: https://github.com/astral-sh/ruff-pre-commit
  rev: v0.16.9
  hooks:
    - id: ruff-check
      args: [--fix]
    - id: ruff-format
- repo: https://github.com/astral-sh/uv-pre-commit
  rev: 0.11.8
  hooks:
    - id: uv-lock
```

`uv run pre-commit install`

`uv run pre-commit run --all-files`

- `install` (une fois par clone) branche les hooks sur `git commit`
- Un hook échoue ou modifie un fichier : commit refusé. On relit, `git add`, on recommence
- `uv-lock` : `uv.lock` doit suivre `pyproject.toml`
- `rev` : version de chaque hook, figée
- Filet local, contournable (`--no-verify`) : la CI reste la référence

Intégration continue avec GitHub Actions

Pour aller plus loin

```
# .github/workflows/ci.yml
name: CI
on:
  push:
    branches: [main]
  pull_request:

jobs:
  quality:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v5
      - uses: astral-sh/setup-uv@v10
        with:
          enable-cache: true
      - run: uv sync --locked
      - run: uv run ruff check
      - run: uv run ruff format --check
      - run: uv run pytest
```

- L'intégration continue de la séance 1, concrètement : à chaque push sur **main** et chaque Pull Request
- Machine neuve à chaque exécution : c'est l'essai chez le client, pas sur votre poste
- Le workflow doit être à la racine du dépôt, dans **.github/workflows/**
- **setup-uv** installe uv, qui installe lui-même la bonne version de Python
- **--locked** : un lock périmé fait échouer la CI
- **format --check** : vérifie sans modifier
- Résultat affiché sur la PR : on ne fusionne pas une CI rouge

CI : cache, matrice, protection de branche

Pour aller plus loin

- **enable-cache: true** : le cache de paquets uv est conservé entre exécutions, la CI gagne le temps de téléchargement
- Matrice : un job par valeur, en parallèle. Utile pour une bibliothèque qui promet plusieurs versions de Python
- Protection de branche (réglages du dépôt) : fusion bloquée tant que les checks requis ne sont pas verts, relecture obligatoire
- **concurrency** : annule l'exécution précédente d'une même branche quand on repousse

```
jobs:
  quality:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        python: ["3.12", "3.13", "3.14"]
    steps:
      - uses: actions/checkout@v5
      - uses: astral-sh/setup-uv@v10
        with:
          python-version: ${ matrix.python }
      - run: uv run pytest
```

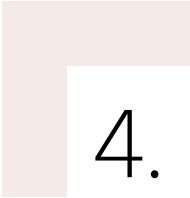
À retenir : le projet

Besoin	Réflexe	En une ligne
Créer un projet installable	<code>uv init --package</code>	<code>uv init --package --python 3.12 app</code>
Ajouter une dépendance	<code>uv add</code> , jamais <code>pip install</code>	<code>uv add numpy</code> <code>uv add --dev pytest</code>
Lancer le code ou un outil du projet	<code>uv run</code>	<code>uv run pytest</code>
Installer à l'identique ailleurs	<code>uv.lock</code> commité, <code>--locked</code>	<code>uv sync --locked</code>
Lancer un outil ponctuellement	<code>uvx</code>	<code>uvx ruff check</code>

À retenir : la qualité

Besoin	Réflexe	En une ligne
Vérifier un comportement	Test unitaire, simple <code>assert</code>	<code>assert (result == 255).all()</code>
Tester plusieurs cas	<code>parametrize</code>	<code>@pytest.mark.parametrize(...)</code>
Vérifier qu'une erreur est levée	<code>pytest.raises</code>	<code>with pytest.raises(ValueError):</code>
Travailler dans un dossier jetable	Fixture <code>tmp_path</code>	<code>def test_pipeline(tmp_path): ...</code>
Uniformiser le code	Ruff : format et lint	<code>uv run ruff check --fix</code>
Vérifier avant chaque commit et merge	pre-commit, puis CI	<code>uv run pre-commit install</code>

Les mêmes vérifications partout : en local, en pre-commit et en CI.



4. TP : fil rouge, étape 3

Trois parties d'environ 30 min : le projet, les tests, la qualité et la CI. Point de départ : votre étape 2, ou le corrigé de la séance 2, sur le site.

TP 1 : transformer les scripts en projet

1. `uv init --package --name fil-rouge --python 3.12 etape_3` à côté de `depart/`, puis `cd etape_3`
2. Déplacez `config.py`, `operations.py` et `pipeline.py` dans `src/fil_rouge/`. Retirez l'en-tête `# /// script` de `pipeline.py` : les dépendances passent dans `pyproject.toml`
3. Imports : `from config import Config` devient `from fil_rouge.config import Config`, idem pour `operations`
4. `uv init` a déjà déclaré `fil-rouge = "fil_rouge:main"` et généré un `main()` factice dans `__init__.py` : remplacez-le par `from fil_rouge.pipeline import main`
5. `uv add numpy pillow`, puis `uv add --dev pytest ruff pre-commit`

Checkpoint : `uv run fil-rouge ../depart/data output produit stats.csv` et `archive.tar`.

TP 2 : les tests

6. `tests/test_operations.py` :

- `Brighten` : multiplication ($100 \times 1,5 = 150$) et saturation à 255
- `Resize(160)` : une image (240, 320, 3) devient (120, 160, 3)
- `mean` : un pixel noir et un blanc donnent `pytest.approx(127.5)`
- `Config` refuse `factor=0, width=-1, quality=100` (`parametrize + raises`)

7. `tests/test_pipeline.py` : fixture avec 3 PNG et un fichier corrompu, test de bout en bout paramétré sur les trois exécuteurs. Vérifiez les 3 résultats, les 4 lignes de `stats.csv` et les 3 JPEG de `archive.tar`

8. Retirez `np.minimum` dans `Brighten` : un test doit échouer. Remettez-le

Checkpoint : `uv run pytest` passe, 10 tests. Ils resservent tels quels à l'étape 4.

TP 3 : qualité, hooks et CI

9. Ajoutez la configuration Ruff dans `pyproject.toml`, puis `uv run ruff format` et `uv run ruff check --fix` jusqu'à zéro erreur (il reste un **PLR2004** à corriger à la main)
10. `.pre-commit-config.yaml`, `uv run pre-commit install`, puis un commit
11. `.github/workflows/ci.yml`. Créez un dépôt GitHub dont **etape_3/** est la racine, poussez, ouvrez une Pull Request

Checkpoint : la CI est verte sur votre Pull Request.

Bonus : `uv build` et le contenu de `dist/` ; `uvx ty check` ou `mypy` en CI ; couverture avec `pytest-cov` ; matrice de versions de Python.

À retenir

Reproductible

`pyproject.toml` et `uv.lock` commités, lock vérifié en CI par `--locked`

Homogène

Ruff en local, en pre-commit et en CI

Testé

Unitaires et bout en bout, `tmp_path`, `parametrize`

Pour le projet

Même structure : `src/`, `tests/`, CI verte avant chaque merge

QUESTIONS ?