

# **Guide pour un Pipeline de Qualité Python étape par étape Pour une équipe de développement**

**Document fourni par :** Mlle Metra Phirielle VANDJY BOUNDZANGA

Etudiante en Master II Programmation à GROUPE SUP'INFO

[metraphirielle@gmail.com](mailto:metraphirielle@gmail.com)

Module : Qualité Logicielle / Assurance Qualité

**Dirigé par :** M. Mohamed Bah Ingénieur Senior et Mentor technique

[batobad@yahoo.fr](mailto:batobad@yahoo.fr) / +221 77 567 76 33

**Lien du code :** <https://github.com/Moouh2/-Pipeline-de-Qualit-Python.git>

## Table des matières

<b>1</b>	Introduction.....	3
<b>1.1</b>	Prérequis .....	3
<b>1.2</b>	Configuration des fichiers .....	5
<b>1.3</b>	Les tests.....	24
<b>2</b>	Conclusion.....	27
<b>3</b>	Webographie.....	28

# 1 Introduction

Ce guide présente un pipeline de qualité **entièrement automatisé** complet pour les projets Python, combinant les pre-commit hooks pour des vérifications locales instantanées et GitHub Actions pour l'intégration continue (CI) et le déploiement continu (CD). L'objectif est d'automatiser, tester, déployer la qualité du code et de standardiser les pratiques de développement au sein de l'équipe. En d'autres termes, il couvre tout le processus depuis l'écriture du code sur la machine du développeur jusqu'au déploiement en production via GitHub Actions. L'automatisation élimine les tâches manuelles répétitives et garantit une qualité constante à chaque étape.

## Avantages du Pipeline

- **Qualité constante** : Application automatique des standards de code à chaque étape
- **Détection précoce** : Identification des problèmes avant le commit
- **Productivité** : Réduction du temps de review et de correction
- **Standardisation** : Uniformisation des pratiques de l'équipe
- **Automatisation Complète** : De l'écriture du code à la production sans intervention manuelle
- **Déploiement Sûr** : Tests automatiques avant chaque déploiement
- **Feedback Immédiat** : Notifications automatiques en cas de problème
- **Traçabilité** : Historique complet de tous les changements et déploiements

## 1.1 Prérequis

### ➤ Configuration Initiale du Projet

Avant de mettre en place les outils, assurez-vous de créer un environnement virtuel si vous voulez isoler les dépendances du projet, mais aussi assurez-vous que votre projet soit bien structuré :

#### **Créez un répertoire pour le projet et initialisez un dépôt Git :**

```
mkdir my_project
cd my_project
git init
```

#### **Environnement virtuel :**

```
python -m venv .venv
source .venv/bin/activate # Sur Windows : .venv\Scripts\activate
```

#### **Structure du projet :**

votre-projet/

```

├── .github/
│   ├── workflows/
│   │   ├── ci.yml
│   │   ├── cd.yml
│   │   └── release.yml
├── scripts/                # 📁 DOSSIER POUR TOUS LES SCRIPTS
│   ├── dev-workflow.sh      # Script principal de développement
│   ├── setup-dev-env.sh    # Configuration automatique environnement
│   ├── check_requirements.py # Vérification des dépendances
│   ├── build_docs.py       # Génération documentation
│   ├── deploy.sh           # Script de déploiement
│   ├── run-tests.sh        # Tests automatisés
│   └── release.sh          # Gestion des releases
├── src/                    # Code source
│   ├── votre_package/
│   │   ├── __init__.py
│   │   └── calculator.py
├── tests/                  # Tests unitaires
│   ├── __init__.py
│   └── test_calculator.py
├── docs/
├── .pre-commit-config.yaml
├── .flake8
├── .gitignore              # Fichiers à ignorer par Git
├── pytest.ini
├── mypy.ini
├── pyproject.toml          # Configuration des outils (Black, Flake8, etc.)
├── requirements-dev.txt    # Dépendances Python
└── README.md               # Documentation

```

➤ Version recommandées :

- Python 3.8+
- Git 2.25+
- pre-commit 3.0+

➤ **Installer les outils de qualité de code**

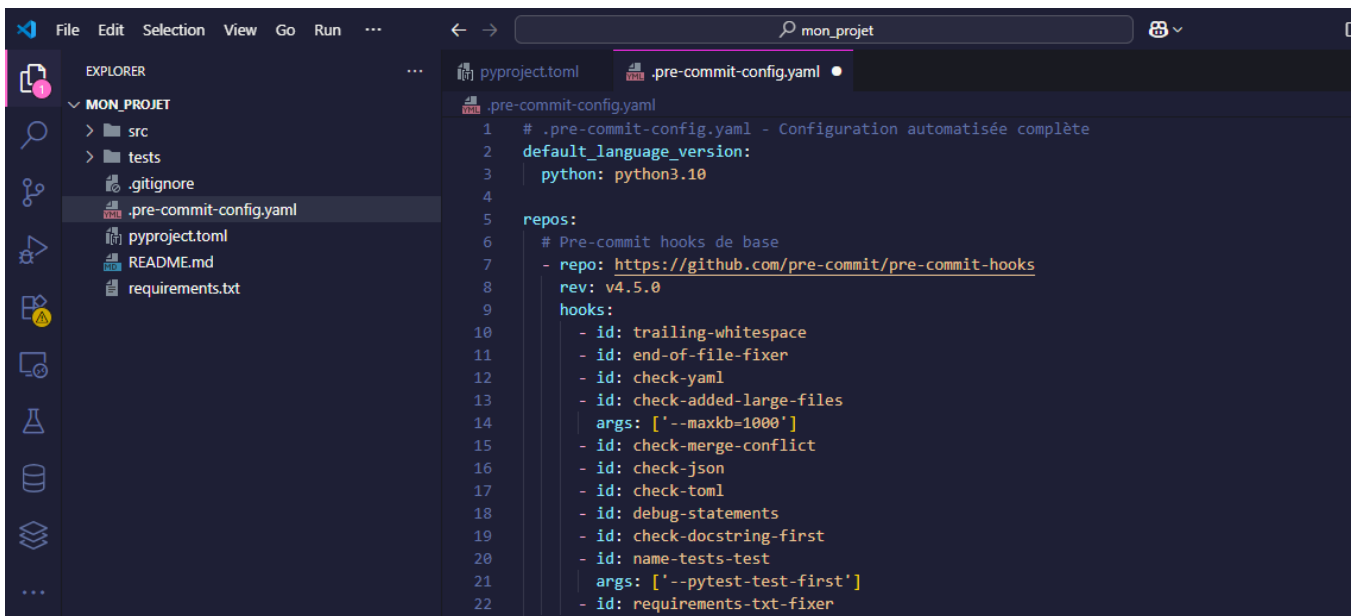
- ❖ `pip install black isort ruff flake8 mypy pytest bandit safety`
- ❖ `pip install pre-commit`

- ❖ pre-commit [install](#)
- ❖ pre-commit [install](#) --hook-type commit-msg
- ❖ pip [install](#) --upgrade pip setuptools wheel
- ❖ pip [install](#) -r requirements-dev.txt
- ❖ black src/ tests/
- ❖ flake8 src/ tests/ # ou ruff check src/ tests/
- ❖ mypy src/ tests/
- ❖ pytest tests/

## 1.2 Configuration des fichiers

- ❖ Créer un fichier **.pre-commit-config.yaml** à la racine du projet pour automatiser les vérifications de qualité avant chaque commit et aussi édité le fichier **pyproject.toml** pour standardiser les configurations des outils. Mais aussi .flake8, .gitignore, mypy.ini, pytest.ini, pyproject.toml, requirements.txt, requirements-dev.txt, README.md, CHANGELOG.md et le tout dans le fichier principal de votre projet.

Fichier **.pre-commit-config.yaml** :



```

1 # .pre-commit-config.yaml - Configuration automatisée complète
2 default_language_version:
3   python: python3.10
4
5 repos:
6   # Pre-commit hooks de base
7   - repo: https://github.com/pre-commit/pre-commit-hooks
8     rev: v4.5.0
9     hooks:
10       - id: trailing-whitespace
11       - id: end-of-file-fixer
12       - id: check-yaml
13       - id: check-added-large-files
14         args: ['--maxkb=1000']
15       - id: check-merge-conflict
16       - id: check-json
17       - id: check-toml
18       - id: debug-statements
19       - id: check-docstring-first
20       - id: name-tests-test
21         args: ['--pytest-test-first']
22       - id: requirements-txt-fixer

```

```

24 # Formatage automatique avec Black
25 - repo: https://github.com/psf/black
26   rev: 24.1.1
27   hooks:
28     - id: black
29       language_version: python3
30       args: [--line-length=88, --target-version=py310]
31
32 # Tri automatique des imports
33 - repo: https://github.com/pycqa/isort
34   rev: 5.13.2
35   hooks:
36     - id: isort
37       args: [--profile=black, --line-length=88, --multi-line=3]
38
39 # Linting avec flake8
40 - repo: https://github.com/pycqa/flake8
41   rev: 7.0.0
42   hooks:
43     - id: flake8
44       args:
45         - --max-line-length=88
46         - --extend-ignore=E203,W503,E501
47         - --max-complexity=10
48         - --statistics
49       additional_dependencies: [flake8-docstrings, flake8-bugbear]
50
51 # Vérification de types avec mypy
52 - repo: https://github.com/pre-commit/mirrors-mypy
53   rev: v1.8.0
54   hooks:
55     - id: mypy
56       additional_dependencies: [types-requests, types-PyYAML, types-setuptools]
57       args: [--ignore-missing-imports, --strict, --show-error-codes]

```

```

# Sécurité avec bandit
- repo: https://github.com/pycqa/bandit
  rev: 1.7.5
  hooks:
    - id: bandit
      args: ['-c', '.bandit', '-r', '.']
      exclude: ^tests/

# Vérification des dépendances
- repo: https://github.com/Lucas-C/pre-commit-hooks-safety
  rev: v1.3.2
  hooks:
    - id: python-safety-dependencies-check
      files: requirements.*\.txt$

# Documentation et docstrings
- repo: https://github.com/pycqa/pydocstyle
  rev: 6.3.0
  hooks:
    - id: pydocstyle
      args: [--convention=google, --add-ignore=D100,D101,D102,D103,D104,D105]

```

```

81 # Détection de code mort
82 - repo: https://github.com/jendrikseipp/vulture
83   rev: v2.10
84   hooks:
85     - id: vulture
86       args: [--min-confidence=80]
87
88 # Conventional commits
89 - repo: https://github.com/commitizen-tools/commitizen
90   rev: v3.13.0
91   hooks:
92     - id: commitizen
93       stages: [commit-msg]

```

```

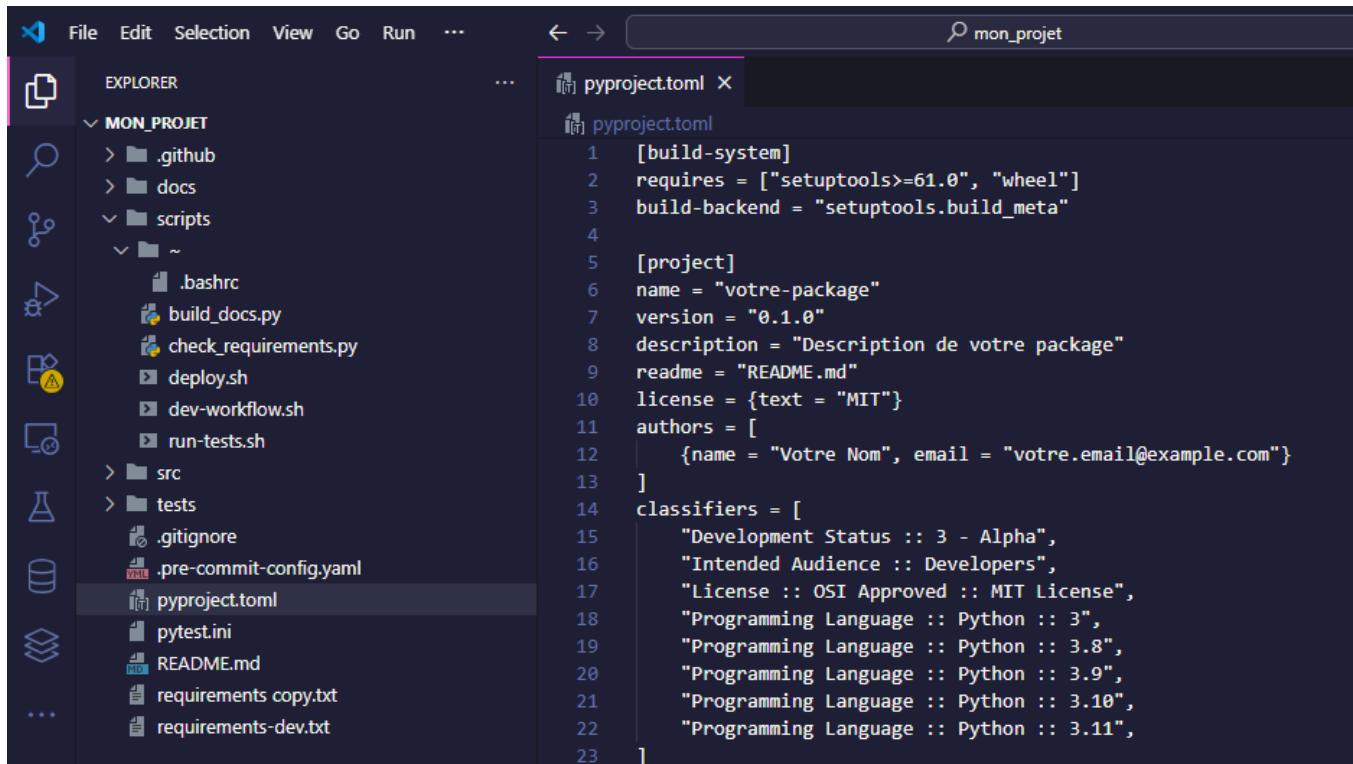
95 : Hooks personnalisés pour l'automatisation
96 : repo: local
97 : hooks:
98 :     # Exécution automatique des tests rapides
99 :     - id: pytest-quick
100 :       name: Tests rapides automatiques
101 :       entry: python -m pytest tests/unit/ -x --tb=short
102 :       language: system
103 :       types: [python]
104 :       pass_filenames: false
105 :
106 :     # Vérification automatique de la couverture
107 :     - id: coverage-check
108 :       name: Vérification couverture minimale
109 :       entry: python -m pytest tests/ --cov=src --cov-fail-under=80 --cov-report=term-missing:skip-covered
110 :       language: system
111 :       types: [python]
112 :       pass_filenames: false
113 :
114 :     # Mise à jour automatique des requirements
115 :     - id: requirements-update
116 :       name: Vérification des requirements
117 :       entry: python scripts/check_requirements.py
118 :       language: system
119 :       files: requirements.*\txt$
120 :
121 :     # Génération automatique de documentation
122 :     - id: docs-build
123 :       name: Build documentation automatique
124 :       entry: python scripts/build_docs.py
125 :       language: system
126 :       files: ^(src/|docs/).*\.(py|md|rst)$
127 :       pass_filenames: false
128 :
129 : # Configuration pour différents environnements
130 : ci:
131 :   autofix_commit_msg: |
132 :     [pre-commit.ci] auto fixes from pre-commit hooks
133 :
134 :   for more information, see https://pre-commit.ci
135 :   autofix_prs: true
136 :   autoupdate_branch: ''
137 :   autoupdate_commit_msg: '[pre-commit.ci] pre-commit autoupdate'
138 :   autoupdate_schedule: weekly
139 :   skip: [pytest-quick, coverage-check] # Skip sur CI pour éviter la duplication
140 :   submodules: false

```

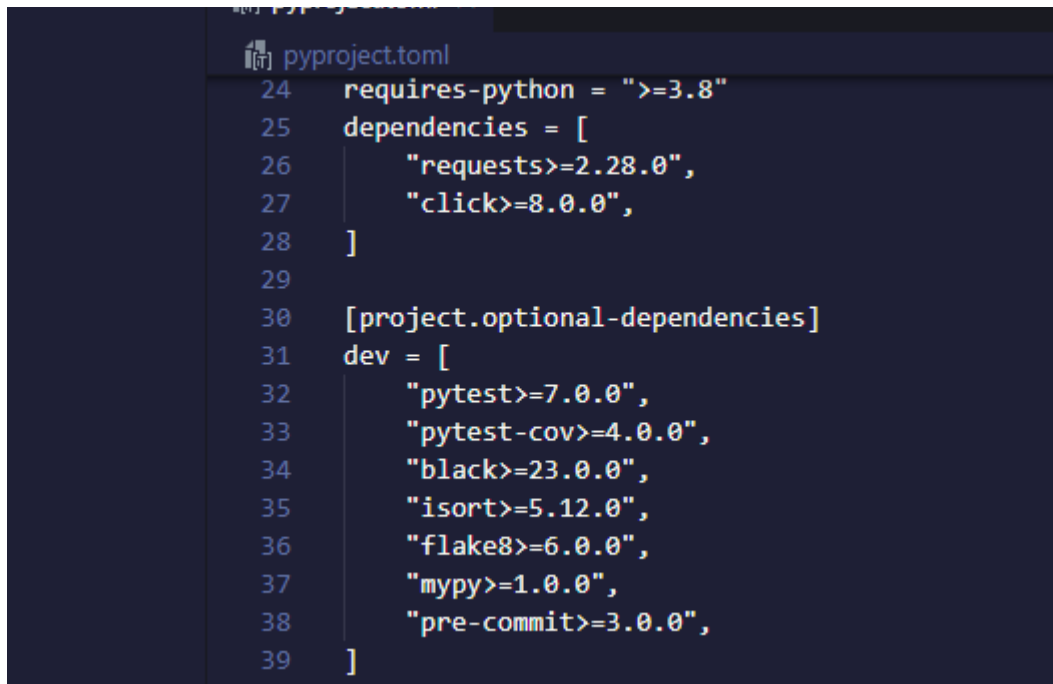
Désormais, avant chaque commit, les vérifications s'exécuteront automatiquement.



## Fichier `pyproject.toml` :



```
1 [build-system]
2   requires = ["setuptools>=61.0", "wheel"]
3   build-backend = "setuptools.build_meta"
4
5 [project]
6   name = "votre-package"
7   version = "0.1.0"
8   description = "Description de votre package"
9   readme = "README.md"
10  license = {text = "MIT"}
11  authors = [
12    {name = "Votre Nom", email = "votre.email@example.com"}
13  ]
14  classifiers = [
15    "Development Status :: 3 - Alpha",
16    "Intended Audience :: Developers",
17    "License :: OSI Approved :: MIT License",
18    "Programming Language :: Python :: 3",
19    "Programming Language :: Python :: 3.8",
20    "Programming Language :: Python :: 3.9",
21    "Programming Language :: Python :: 3.10",
22    "Programming Language :: Python :: 3.11",
23  ]
```



```
24 requires-python = ">=3.8"
25 dependencies = [
26     "requests>=2.28.0",
27     "click>=8.0.0",
28 ]
29
30 [project.optional-dependencies]
31 dev = [
32     "pytest>=7.0.0",
33     "pytest-cov>=4.0.0",
34     "black>=23.0.0",
35     "isort>=5.12.0",
36     "flake8>=6.0.0",
37     "mypy>=1.0.0",
38     "pre-commit>=3.0.0",
39 ]
```

```

1 [tool.black]
2 line-length = 88
3 target-version = ['py38', 'py39', 'py310', 'py311']
4 include = '\.pyi?$'
5
6 [tool.isort]
7 profile = "black"
8 line_length = 88
9 multi_line_output = 3
10
11 [tool.pytest.ini_options]
12 testpaths = ["tests"]
13 python_files = "test_*.py"
14 addopts = "--cov=src --cov-report=term-missing --cov-report=html --cov-fail-under=80"
15
16 [tool.mypy]
17 python_version = "3.8"
18 warn_return_any = true
19 warn_unused_configs = true
20 disallow_untyped_defs = true
21 ignore_missing_imports = true

```

Fichier **requirements-dev.txt** :

```

1 # requirements-dev.txt - Dépendances de développement automatisées
2
3 # Core development tools
4 pre-commit>=3.6.0
5 black>=24.1.0
6 isort>=5.13.0
7 flake8>=7.0.0
8 mypy>=1.8.0
9
10 # Testing framework
11 pytest>=8.0.0
12 pytest-cov>=4.0.0
13 pytest-xdist>=3.5.0 # Tests parallèles
14 pytest-mock>=3.12.0
15 coverage>=7.4.0
16
17 # Security tools
18 bandit>=1.7.5
19 safety>=3.0.0
20 semgrep>=1.45.0
21

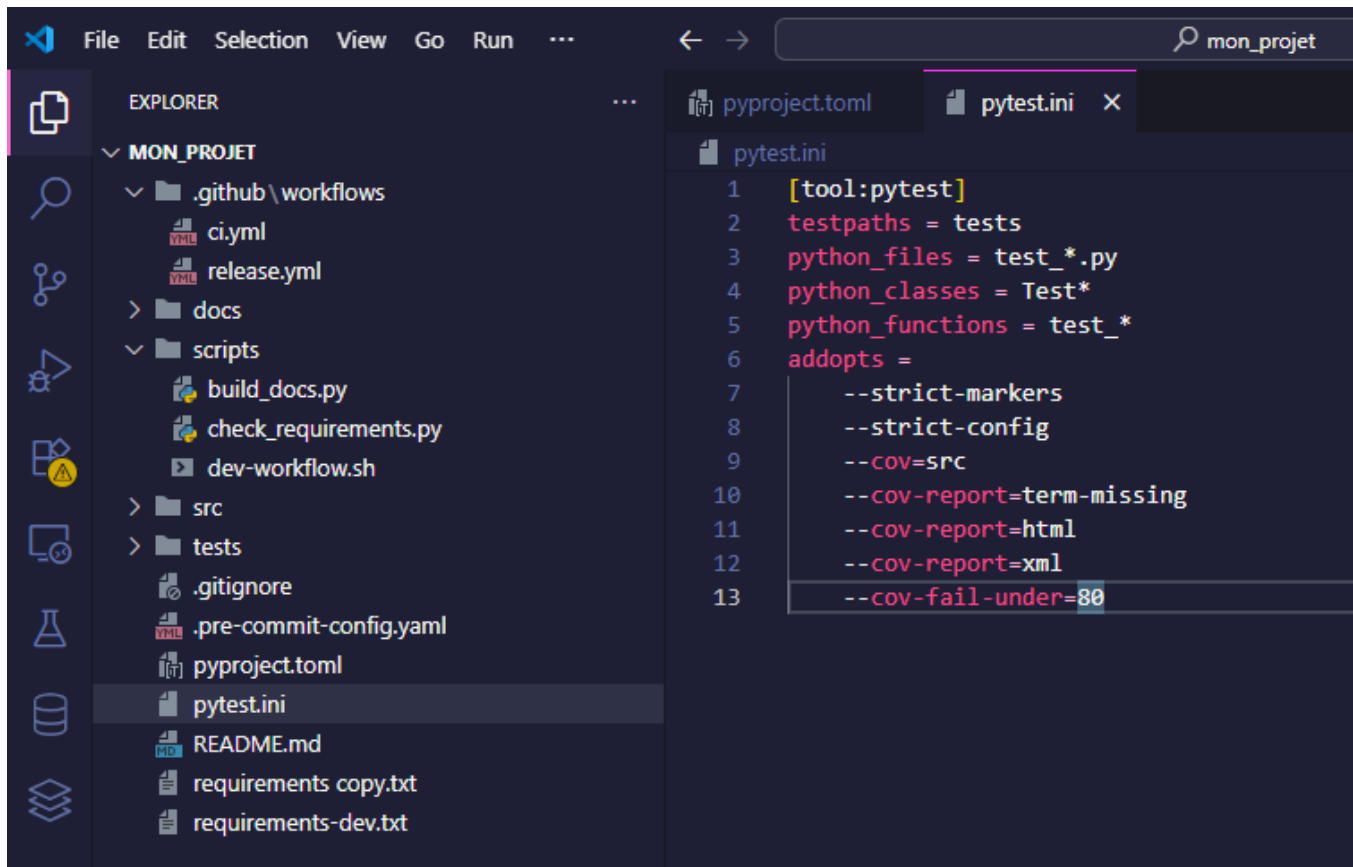
```

```

22 # Documentation
23 sphinx>=7.2.0
24 sphinx-rtd-theme>=2.0.0
25 myst-parser>=2.0.0
26
27 # Code quality
28 pydocstyle>=6.3.0
29 pylint>=3.0.0
30 vulture>=2.10 # Dead code detection
31
32 # Development utilities
33 ipython>=8.20.0
34 jupyter>=1.0.0
35 notebook>=7.0.0
36 tox>=4.12.0
37
38 # Build tools
39 build>=1.0.0
40 twine>=4.0.0
41 setuptools>=69.0.0
42 wheel>=0.42.0
43
44 # Git hooks and workflow
45 commitizen>=3.13.0 # Conventional commits
46 gitpython>=3.1.0
47
48 # Performance monitoring
49 py-spy>=0.3.14
50 memory-profiler>=0.61.0

```

Fichier **pytest.ini** :



- ❖ Créer dans le dossier scripts les fichiers **dev-workflow.sh**, **check\_requirements.py**, **build\_docs.py**, **run-tests.sh**, **deploy.sh**, . Le dossier scripts est là pour une meilleur organisation visible et claire qui permet une automatisation quotidienne.

Fichier **dev-workflow.sh** :

The image shows a Visual Studio Code editor window with a project named 'mon\_projet'. The Explorer sidebar on the left shows the project structure: 'MON\_PROJET' contains 'docs', 'scripts', 'src', and 'tests' folders, along with files like '.gitignore', '.pre-commit-config.yaml', 'pyproject.toml', 'README.md', 'requirements copy.txt', and 'requirements-dev.txt'. The 'scripts' folder is expanded, showing 'dev-workflow.sh' as the selected file. The main editor area displays the content of 'dev-workflow.sh', which is a bash script for automated development workflow. The script includes configuration variables for branch prefixes and main branches, utility functions for running tests and quality checks, and a function for creating feature branches. The script is currently open in a tab labeled 'dev-workflow.sh'.

```
1  #!/bin/bash
2  # scripts/dev-workflow.sh - Workflow de développement automatisé
3
4  # Configuration
5  BRANCH_PREFIX="feature/"
6  MAIN_BRANCH="main"
7
8  # Fonctions utilitaires
9  run_tests() {
10     echo "🚀 Exécution des tests..."
11     python -m pytest tests/ -v --cov=src --cov-report=term-missing
12 }
13
14 run_quality_checks() {
15     echo "🔍 Vérifications de qualité..."
16     pre-commit run --all-files
17 }
18
19 create_feature_branch() {
20     local feature_name=$1
21     if [ -z "$feature_name" ]; then
22         echo "❌ Nom de feature requis"
23         echo "Usage: $0 new-feature <nom-de-la-feature>"
24         exit 1
25     fi
26
27     echo "🌱 Création de la branche feature/$feature_name"
28     git checkout $MAIN_BRANCH
29     git pull origin $MAIN_BRANCH
30     git checkout -b "$BRANCH_PREFIX$feature_name"
31 }
```

```

3  commit_changes() {
4      local message=$1
5      if [ -z "$message" ]; then
6          echo "❌ Message de commit requis"
7          echo "Usage: $0 commit <message>"
8          exit 1
9      fi
10
11     echo "📁 Ajout des fichiers modifiés..."
12     git add .
13
14     echo "🔍 Vérifications pre-commit..."
15     if ! pre-commit run --all-files; then
16         echo "❌ Vérifications pre-commit échouées"
17         exit 1
18     fi
19
20     echo "📄 Commit avec message: $message"
21     git commit -m "$message"
22 }

```

```

54  push_and_pr() {
55      local current_branch=$(git branch --show-current)
56
57      echo "📌 Push de la branche $current_branch"
58      git push origin $current_branch
59
60      echo "🔗 Création automatique de la PR..."
61      if command -v gh &> /dev/null; then
62          gh pr create --title "$(git log -1 --pretty=%B)" --body "Automatiquement créée par dev-workflow."
63      else
64          echo "💡 Installez GitHub CLI (gh) pour la création automatique de PR"
65          echo "🌐 Créez manuellement la PR sur GitHub"
66      fi
67  }

```

```

69 # Workflow principal
70 case "$1" in
71     "setup")
72         echo "⚙️ Configuration de l'environnement de développement"
73         source venv/bin/activate 2>/dev/null || {
74             echo "🔄 Création de l'environnement virtuel"
75             python3 -m venv venv
76             source venv/bin/activate
77         }
78         pip install -r requirements-dev.txt
79         pre-commit install
80         echo "✅ Configuration terminée"
81         ;;
82     "test")
83         run_tests
84         ;;
85     "check")
86         run_quality_checks
87         ;;
88     "new-feature")
89         create_feature_branch $2
90         ;;
91     "commit")
92         shift

```

```

93         commit_changes "$*"
94         ;;
95     "push")
96         push_and_pr
97         ;;
98     "full-check")
99         echo "🔍 Vérification complète..."
100         run_quality_checks && run_tests
101         echo "✅ Toutes les vérifications sont passées"
102         ;;
103     "release")
104         echo "🚀 Préparation de la release..."
105         git checkout $MAIN_BRANCH
106         git pull origin $MAIN_BRANCH
107         run_quality_checks && run_tests
108         echo "✅ Prêt pour la release"
109         ;;
110 *)

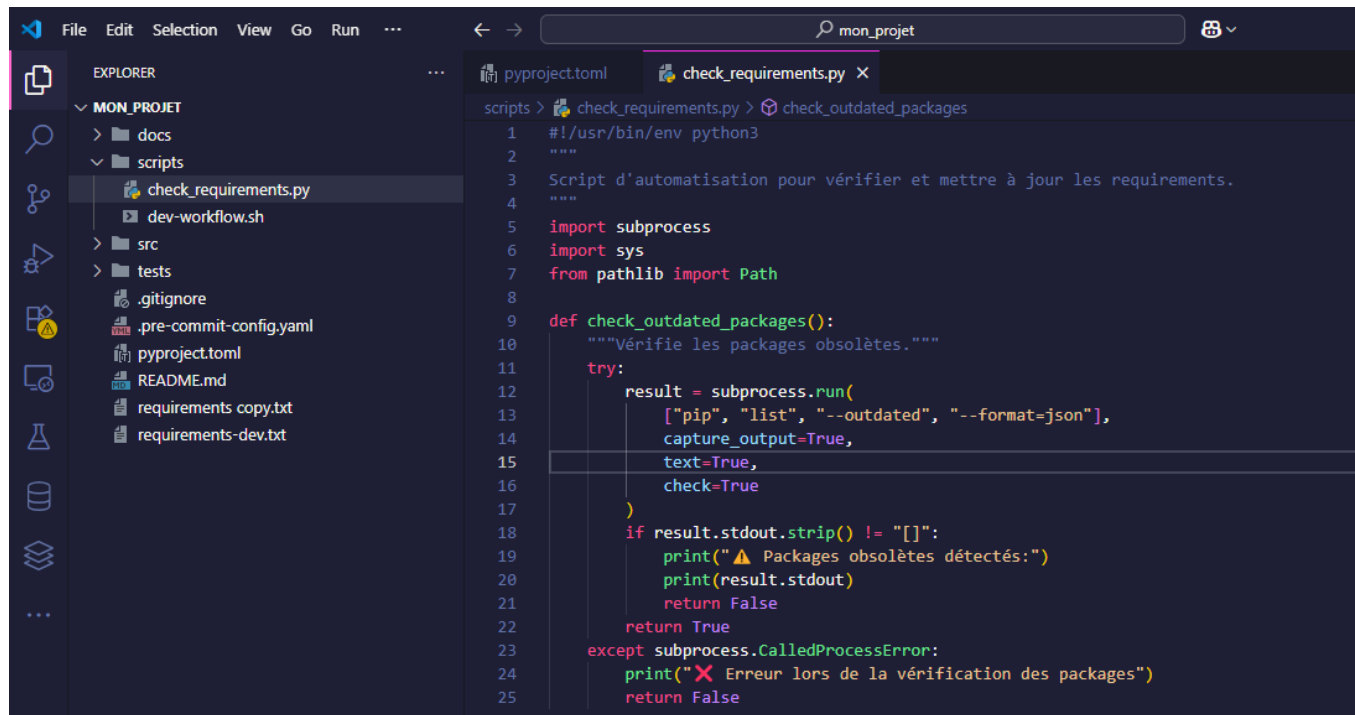
```

```

111 echo "🔧 Workflow de Développement Automatisé"
112 echo ""
113 echo "Usage: $0 <commande> [arguments]"
114 echo ""
115 echo "Commandes disponibles:"
116 echo "  setup          - Configuration initiale"
117 echo "  test           - Exécution des tests"
118 echo "  check          - Vérifications de qualité"
119 echo "  new-feature <nom> - Création d'une nouvelle branche feature"
120 echo "  commit <message> - Commit automatisé avec vérifications"
121 echo "  push           - Push et création de PR automatique"
122 echo "  full-check     - Vérification complète (qualité + tests)"
123 echo "  release        - Préparation de release"
124 echo ""
125 exit 1
126 ;;
127 esac

```

Fichier `check_requirements.py` :



```

1  #!/usr/bin/env python3
2  """
3  Script d'automatisation pour vérifier et mettre à jour les requirements.
4  """
5  import subprocess
6  import sys
7  from pathlib import Path
8
9  def check_outdated_packages():
10     """Vérifie les packages obsolètes."""
11     try:
12         result = subprocess.run(
13             ["pip", "list", "--outdated", "--format=json"],
14             capture_output=True,
15             text=True,
16             check=True
17         )
18         if result.stdout.strip() != "[]":
19             print("⚠ Packages obsolètes détectés:")
20             print(result.stdout)
21             return False
22         return True
23     except subprocess.CalledProcessError:
24         print("❌ Erreur lors de la vérification des packages")
25         return False

```

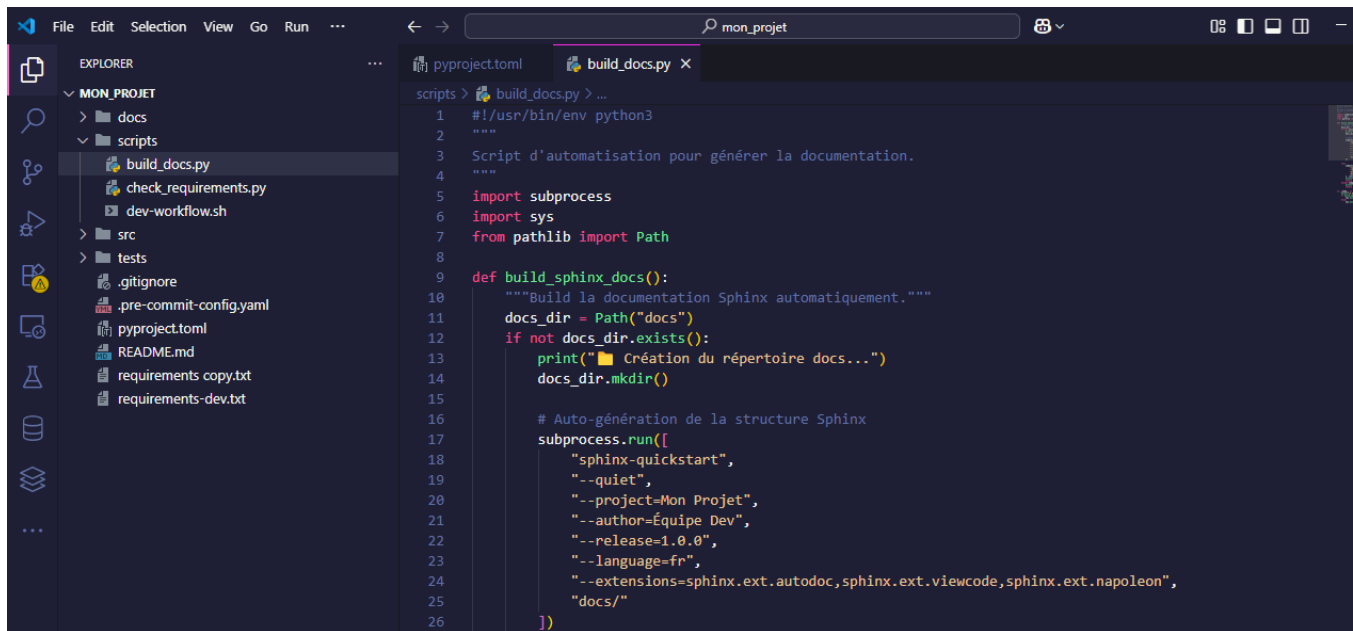


```

20
27 def verify_requirements_sync():
28     """Vérifie la synchronisation des fichiers requirements."""
29     requirements_files = [
30         "requirements.txt",
31         "requirements-dev.txt"
32     ]
33
34     for req_file in requirements_files:
35         if Path(req_file).exists():
36             try:
37                 subprocess.run(
38                     ["pip-check-reqs", req_file],
39                     check=True,
40                     capture_output=True
41                 )
42                 print(f"✅ {req_file} est synchronisé")
43             except subprocess.CalledProcessError as e:
44                 print(f"❌ {req_file} n'est pas synchronisé:")
45                 print(e.stdout.decode())
46             return False
47     return True
48
49 if __name__ == "__main__":
50     print("🔍 Vérification des requirements...")
51
52     checks = [
53         check_outdated_packages(),
54         verify_requirements_sync()
55     ]
56
57     if all(checks):
58         print("✅ Toutes les vérifications sont passées")
59         sys.exit(0)
60     else:
61         print("❌ Certaines vérifications ont échoué")
62         sys.exit(1)

```

Fichier **build\_docs.py** :



```
1 #!/usr/bin/env python3
2 """
3 Script d'automatisation pour générer la documentation.
4 """
5 import subprocess
6 import sys
7 from pathlib import Path
8
9 def build_sphinx_docs():
10     """Build la documentation Sphinx automatiquement."""
11     docs_dir = Path("docs")
12     if not docs_dir.exists():
13         print("📁 Création du répertoire docs...")
14         docs_dir.mkdir()
15
16     # Auto-génération de la structure Sphinx
17     subprocess.run([
18         "sphinx-quickstart",
19         "--quiet",
20         "--project=Mon Projet",
21         "--author=Équipe Dev",
22         "--release=1.0.0",
23         "--language=fr",
24         "--extensions=sphinx.ext.autodoc,sphinx.ext.viewcode,sphinx.ext.napoleon",
25         "docs/"
26     ])
```

```
28     try:
29         subprocess.run([
30             "sphinx-build",
31             "-b", "html",
32             "docs/",
33             "docs/_build/html/"
34         ], check=True)
35         print("✅ Documentation générée avec succès")
36         return True
37     except subprocess.CalledProcessError:
38         print("❌ Erreur lors de la génération de la documentation")
39         return False
40
41 if __name__ == "__main__":
42     print("📄 Génération automatique de la documentation...")
43     if build_sphinx_docs():
44         sys.exit(0)
45     else:
46         sys.exit(1)
```

Fichier **run-tests.sh** :

The screenshot shows the Visual Studio Code interface. On the left, the Explorer sidebar displays the project structure: `MON_PROJET` containing `.github`, `docs`, `scripts`, `src`, and `tests`. The `scripts` directory is expanded, showing `build_docs.py`, `check_requirements.py`, `dev-workflow.sh`, and `run-tests.sh`. The main editor window shows the content of `run-tests.sh` with the following code:

```
1  #!/bin/bash
2  # scripts/run-tests.sh - Tests automatisés avancés
3
4  set -e
5
6  # Configuration
7  TEST_DIR="tests"
8  SRC_DIR="src"
9  COVERAGE_THRESHOLD=80
10
11  echo "🚀 Exécution de la suite de tests complète"
12
13  # Tests unitaires
14  echo "📁 Tests unitaires..."
15  python -m pytest $TEST_DIR/unit/ -v --tb=short
16
17  # Tests d'intégration
18  if [ -d "$TEST_DIR/integration" ]; then
19      echo "🔗 Tests d'intégration..."
20      python -m pytest $TEST_DIR/integration/ -v --tb=short
21  fi
```

This block shows the continuation of the `run-tests.sh` script. The code includes sections for tests with coverage and performance tests, followed by a final success message.

```
22
23  # Tests avec couverture
24  echo "📊 Tests avec mesure de couverture..."
25  python -m pytest $TEST_DIR/ \
26      --cov=$SRC_DIR \
27      --cov-report=term-missing \
28      --cov-report=html \
29      --cov-report=xml \
30      --cov-fail-under=$COVERAGE_THRESHOLD
31
32  # Tests de performance (si présents)
33  if [ -d "$TEST_DIR/performance" ]; then
34      echo "⚡ Tests de performance..."
35      python -m pytest $TEST_DIR/performance/ -v --benchmark-only
36  fi
37
38  echo "✅ Tous les tests sont passés avec succès!"
```

Fichier **deploy.sh** :

```

1 #!/bin/bash
2 # scripts/deploy.sh - Déploiement automatisé
3
4 set -e
5
6 ENVIRONMENT=${1:-staging}
7 VERSION=${2:-latest}
8
9 echo "🚀 Déploiement automatique vers $ENVIRONMENT (version: $VERSION)"
10
11 # Vérifications pré-déploiement
12 echo "🔍 Vérifications pré-déploiement..."
13 make full-check
14
15 # Build du package
16 echo "📦 Construction du package..."
17 python -m build
18
19 # Tests du package construit
20 echo "🧪 Test du package construit..."
21 python -m pip install dist/*.whl --force-reinstall
22 python -c "import $(basename $(pwd) | tr '-' '_'); print('Package installé avec succès!'"

```

```

24 # Déploiement selon l'environnement
25 case $ENVIRONMENT in
26     "staging")
27         echo "🚀 Déploiement vers staging..."
28         # Commandes spécifiques au staging
29         ;;
30     "production")
31         echo "🏢 Déploiement vers production..."
32         # Commandes spécifiques à la production
33         # Avec vérifications supplémentaires
34         ;;
35     *)
36         echo "❌ Environnement non reconnu: $ENVIRONMENT"
37         exit 1
38         ;;
39 esac
40
41 echo "✅ Déploiement terminé avec succès!"

```

❖ Créer dans le dossier `.github/workflows/` les fichiers **ci.yml**, **release.yml**,

Fichier **ci.yml** :

The image shows a Visual Studio Code editor window with a project named 'mon\_projet'. The Explorer sidebar on the left shows the project structure, including a '.github/workflows' directory containing 'ci.yml' and 'release.yml'. The main editor area displays the content of 'ci.yml', which is a GitHub Actions workflow. The workflow is named 'CI Pipeline' and is triggered on 'push' to 'main' or 'develop' branches, and on 'pull\_request' to 'main' or 'develop' branches. It consists of several jobs: 'quality-checks' (which runs on 'ubuntu-latest' and uses a matrix for Python versions 3.8, 3.9, 3.10, and 3.11), 'Cache pip dependencies', 'Install dependencies', 'Run pre-commit hooks', and 'Run tests with pytest'.

```
1 name: CI Pipeline
2
3 on:
4   push:
5     branches: [ main, develop ]
6   pull_request:
7     branches: [ main, develop ]
8
9 jobs:
10  quality-checks:
11    runs-on: ubuntu-latest
12    strategy:
13      matrix:
14        python-version: [3.8, 3.9, "3.10", "3.11"]
15
16    steps:
17      - name: Checkout code
18        uses: actions/checkout@v4
19
20      - name: Set up Python ${ matrix.python-version }
21        uses: actions/setup-python@v4
22        with:
23          python-version: ${ matrix.python-version }
24
25      - name: Cache pip dependencies
26        uses: actions/cache@v3
27        with:
28          path: ~/.cache/pip
29          key: ${ runner.os }}-pip-${ hashFiles('**/requirements*.txt') }}
30          restore-keys: |
31            ${ runner.os }}-pip-
32
33      - name: Install dependencies
34        run: |
35          python -m pip install --upgrade pip
36          pip install -r requirements.txt
37          pip install -r requirements-dev.txt
38
39      - name: Run pre-commit hooks
40        uses: pre-commit/action@v3.0.0
41
42      - name: Run tests with pytest
43        run: |
44          pytest tests/ --cov=src --cov-report=xml --cov-report=html
45
```

```

46 - name: Upload coverage to Codecov
47   uses: codecov/codecov-action@v3
48   with:
49     file: ./coverage.xml
50     flags: unittests
51     name: codecov-umbrella
52
53 - name: Security scan with bandit
54   run: |
55     bandit -r src/ -f json -o bandit-report.json
56
57 - name: Dependency security check
58   run: |
59     safety check --json --output safety-report.json
60
61 - name: Archive security reports
62   uses: actions/upload-artifact@v3
63   with:
64     name: security-reports-${{ matrix.python-version }}
65     path: |
66       bandit-report.json
67       safety-report.json
68

```

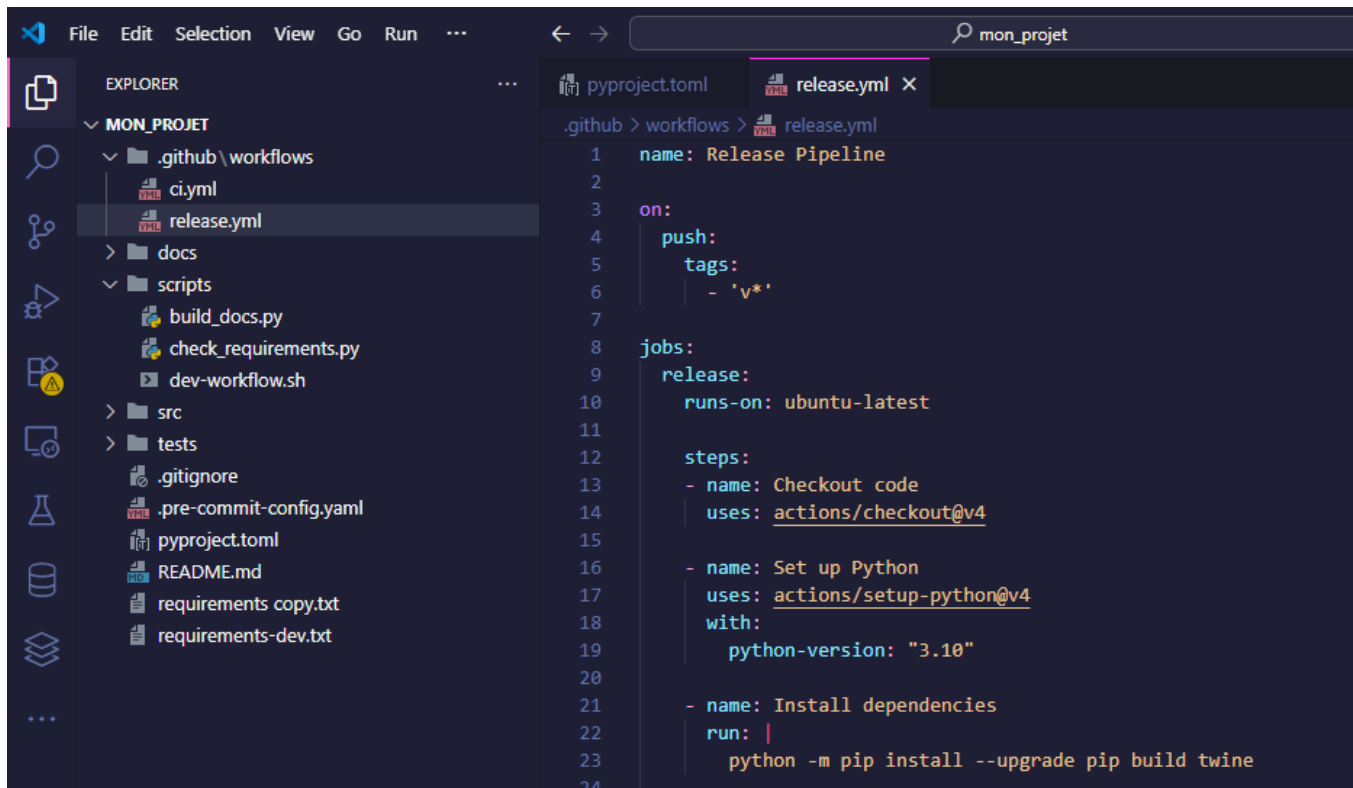
```

69 build-and-test:
70   needs: quality-checks
71   runs-on: ubuntu-latest
72
73   steps:
74   - name: Checkout code
75     uses: actions/checkout@v4
76
77   - name: Set up Python
78     uses: actions/setup-python@v4
79     with:
80       python-version: "3.10"
81
82   - name: Build package
83     run: |
84       python -m pip install --upgrade pip build
85       python -m build
86
87   - name: Test installation
88     run: |
89       pip install dist/*.whl
90       python -c "import your_package; print('Package installed successfully')"

```

- À chaque push ou pull request, le workflow s'exécute sur GitHub.
- Si une étape échoue, une notification est envoyée (email, Slack, etc.).

### Fichier **release.yml** :



The screenshot shows the Visual Studio Code interface. On the left, the Explorer view displays the project structure for 'MON\_PROJET'. The file '.github/workflows/release.yml' is selected. The main editor area shows the content of 'release.yml' with line numbers 1 through 24. The file is titled 'name: Release Pipeline' and is triggered on 'push' events for tags matching the pattern 'v\*'. It defines a single job named 'release' that runs on 'ubuntu-latest'. The job consists of three steps: 1. 'Checkout code' using 'actions/checkout@v4'. 2. 'Set up Python' using 'actions/setup-python@v4' with 'python-version: "3.10"'. 3. 'Install dependencies' running the command 'python -m pip install --upgrade pip build twine'.

```
1 name: Release Pipeline
2
3 on:
4   push:
5     tags:
6       - 'v*'
7
8 jobs:
9   release:
10    runs-on: ubuntu-latest
11
12    steps:
13      - name: Checkout code
14        uses: actions/checkout@v4
15
16      - name: Set up Python
17        uses: actions/setup-python@v4
18        with:
19          python-version: "3.10"
20
21      - name: Install dependencies
22        run: |
23          python -m pip install --upgrade pip build twine
24
```

```

25     - name: Build package
26       run: |
27         python -m build
28
29     - name: Check package
30       run: |
31         twine check dist/*
32
33     - name: Create GitHub Release
34       uses: actions/create-release@v1
35       env:
36         GITHUB_TOKEN: ${ secrets.GITHUB_TOKEN }
37       with:
38         tag_name: ${ github.ref }
39         release_name: Release ${ github.ref }
40         draft: false
41         prerelease: false

```

❖ jhe

### 1.3 Les tests

Structure des Tests :

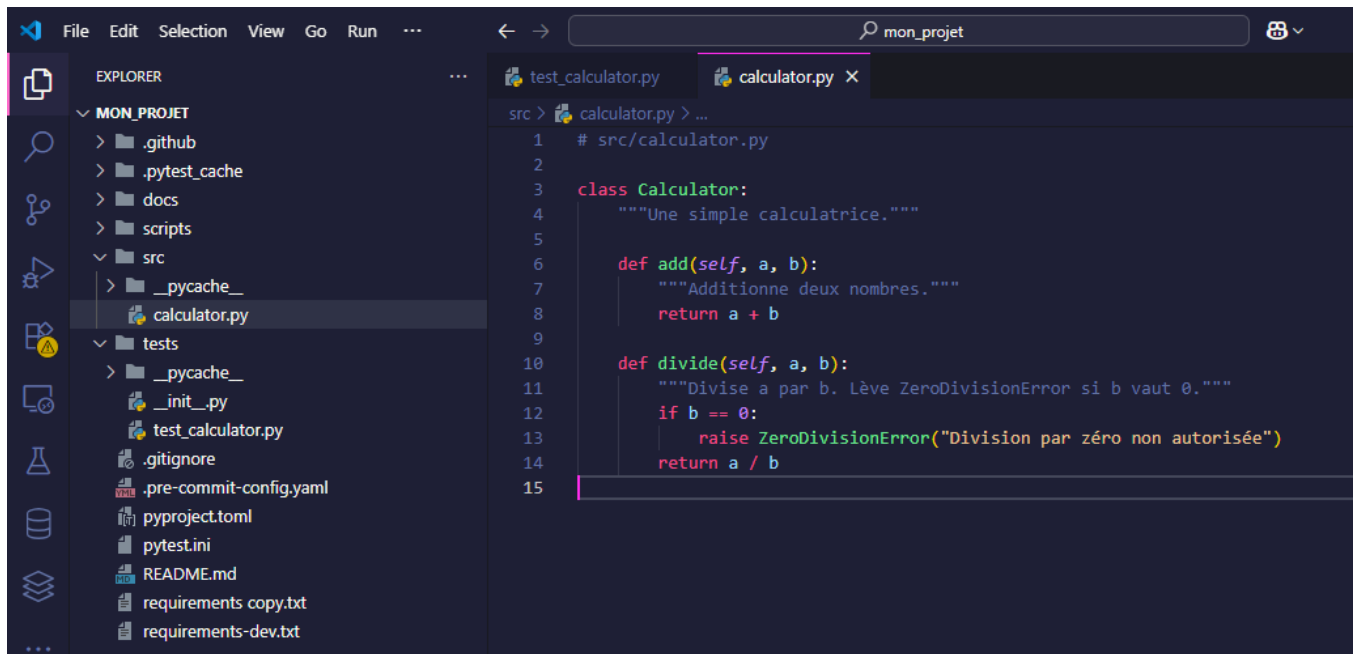
```

mon_projet/
|
├── src/
│   └── calculator.py
├── pytest.ini
├── tests/
│   ├── test_calculator.py
│   └── __init__.py

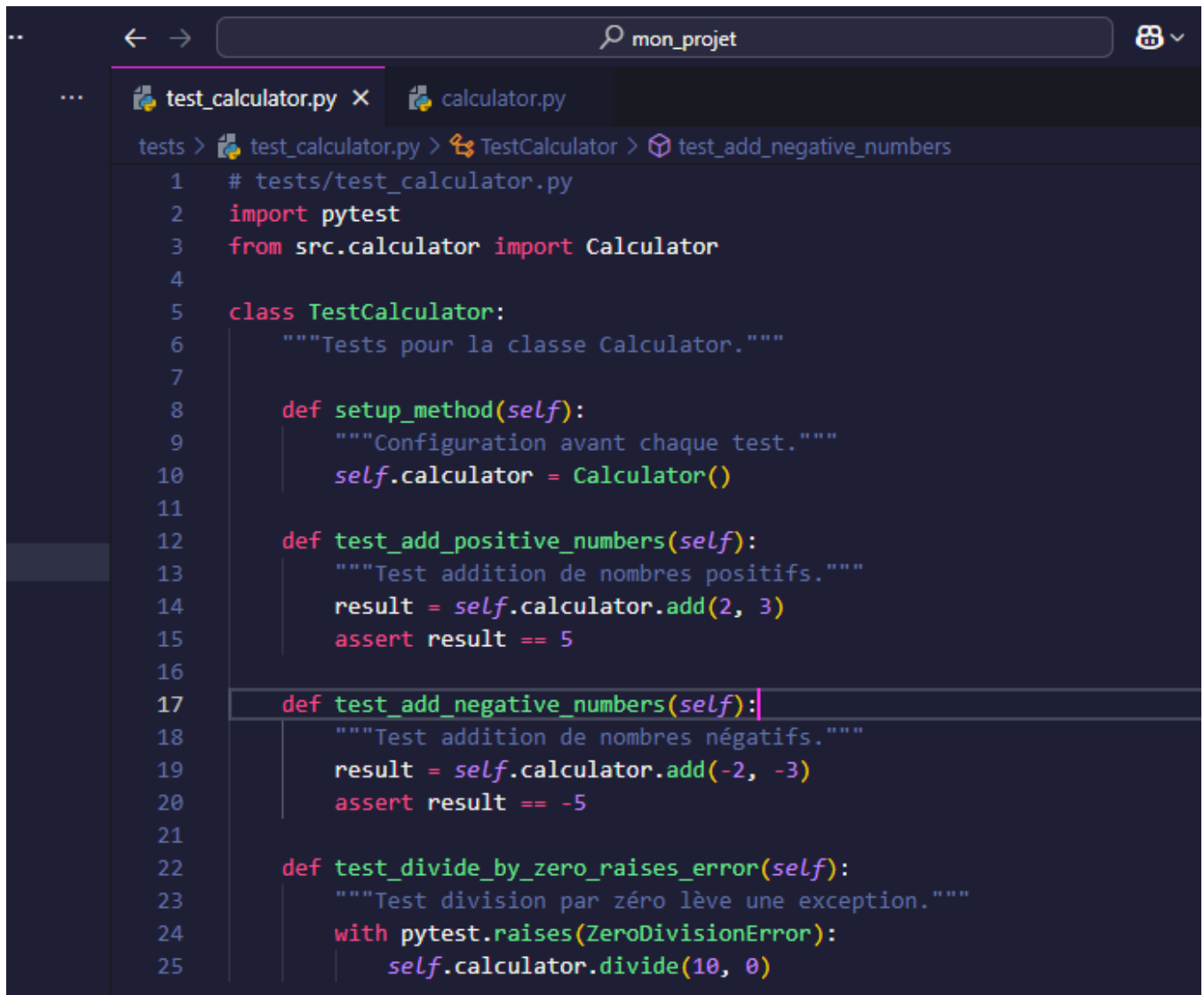
```

Fichier **calculator.py** :





Fichier **test\_calculator.py** :



The image shows a code editor window with a dark theme. At the top, there's a search bar with the text "mon\_projet" and a magnifying glass icon. Below the search bar, there are two tabs: "test\_calculator.py" (active) and "calculator.py". The breadcrumb navigation shows the path: "tests > test\_calculator.py > TestCalculator > test\_add\_negative\_numbers". The code is written in Python and defines a class "TestCalculator" with several methods. The method "test\_add\_negative\_numbers" is currently selected, indicated by a light blue highlight. The code includes comments in French and uses "pytest" for testing.

```
1 # tests/test_calculator.py
2 import pytest
3 from src.calculator import Calculator
4
5 class TestCalculator:
6     """Tests pour la classe Calculator."""
7
8     def setup_method(self):
9         """Configuration avant chaque test."""
10        self.calculator = Calculator()
11
12    def test_add_positive_numbers(self):
13        """Test addition de nombres positifs."""
14        result = self.calculator.add(2, 3)
15        assert result == 5
16
17    def test_add_negative_numbers(self):
18        """Test addition de nombres négatifs."""
19        result = self.calculator.add(-2, -3)
20        assert result == -5
21
22    def test_divide_by_zero_raises_error(self):
23        """Test division par zéro lève une exception."""
24        with pytest.raises(ZeroDivisionError):
25            self.calculator.divide(10, 0)
```

Résultat du test :

```
1 # tests/test_calculator.py
===== test session starts =====
platform win32 -- Python 3.13.2, pytest-8.4.0, pluggy-1.6.0 -- C:\Users\hp\AppData\Local\Programs\Python\Python313\python.exe
PS C:\Users\hp\Desktop\mon_projet> pytest -v tests\test_calculator.py
===== test session starts =====
platform win32 -- Python 3.13.2, pytest-8.4.0, pluggy-1.6.0 -- C:\Users\hp\AppData\Local\Programs\Python\Python313\python.exe
cachedir: .pytest_cache
rootdir: C:\Users\hp\Desktop\mon_projet
PS C:\Users\hp\Desktop\mon_projet> pytest -v tests\test_calculator.py
===== test session starts =====
platform win32 -- Python 3.13.2, pytest-8.4.0, pluggy-1.6.0 -- C:\Users\hp\AppData\Local\Programs\Python\Python313\python.exe
cachedir: .pytest_cache
rootdir: C:\Users\hp\Desktop\mon_projet
configfile: pytest.ini
plugins: anyio-4.9.0
collected 3 items

tests/test_calculator.py::TestCalculator::test_add_positive_numbers PASSED [ 33%]
tests/test_calculator.py::TestCalculator::test_add_negative_numbers PASSED [ 66%]
tests/test_calculator.py::TestCalculator::test_divide_by_zero_raises_error PASSED [100%]

===== 3 passed in 0.03s =====
PS C:\Users\hp\Desktop\mon_projet>
```

## 2 Conclusion

Ce pipeline de qualité Python offre une base solide pour maintenir un code de haute qualité tout en améliorant la productivité de l'équipe. L'automatisation des vérifications via les pre-commit hooks et GitHub Actions permet de détecter et corriger les problèmes tôt dans le cycle de développement.

La clé du succès réside dans l'adoption progressive de ces outils par toute l'équipe et l'adaptation continue du pipeline selon les besoins spécifiques du projet. N'hésitez pas à personnaliser les configurations selon vos standards et contraintes techniques.

### Points Clés à Retenir :

- Les pre-commit hooks empêchent les problèmes de qualité d'atteindre le repository
- GitHub Actions assure la validation continue sur toutes les plateformes
- La documentation et la formation sont essentielles pour l'adoption
- La maintenance régulière du pipeline garantit son efficacité à long terme

### 3 Webographie

- Documentation officielle de pytest : : <https://docs.pytest.org/>
- Tutoriel pytest : [Effective Python Testing With pytest – Real Python](#)
- Grok, ChatGpt, claude et Deeseek
- Automatic Python Quality Code QA test Pre-commit – Medium  
<https://medium.com/@marcdomenechvila/automatic-qa-code-pre-commit-b6dbe9332e01>
- Continuous Integration and Deployment for Python With GitHub Actions  
<https://realpython.com/github-actions-python/>
- Automate Python workflow using pre-commits: black and flake8  
<https://lvmiranda921.github.io/notebook/2018/06/21/precommits-using-black-and-flake8/>
- Effortless Code Quality: Ultimate Pre-Commit Hooks Guide for 2025  
<https://gatlenculp.medium.com/effortless-code-quality-the-ultimate-pre-commit-hooks-guide-for-2025-57ca501d9835>
- Create a GitHub Actions CI workflow for a Python Package  
<https://dedreira.medium.com/create-a-github-actions-ci-pipeline-for-a-python-package-4c4c02dc5f2e>
- Automating Python Code Quality: By Integrating Pre-Commit with GitHub Actions  
<https://medium.com/@othmane.ghandi/automating-python-code-quality-by-integrating-pre-commit-with-github-actions-3d29c9bb8067>
- Elevate Your Code Quality: Building an Effective Pre-Commit Pipeline for Python Project : <https://www.linkedin.com/pulse/elevate-your-code-quality-building-effective-pipeline-ansari-qskfc/>
- Integrating CI/CD Pipeline in Python Project with GitHub Actions  
<https://www.youtube.com/watch?v=T-l00oT4yfA>
- Creating CI/CD Pipeline using GitHub Actions for Python Project  
<https://www.youtube.com/watch?v=WTofttoD2xg>
- Modern Python part 3: run a CI pipeline & publish your package to PiPy  
<https://www.adaltas.com/en/2021/06/28/pypi-tox-cicd-github-actions/>
- Testing your Python Project with GitHub Actions  
[https://github.com/mwouts/github\\_actions\\_python](https://github.com/mwouts/github_actions_python)
- Boost Your Python Code Quality: Pre-commit Tutorial (UV & Ruff)  
<https://www.youtube.com/watch?v=xhg1dJHLqSM>
- Comment choisir le meilleur workflow GitHub Actions pour votre projet ?  
<https://www.linkedin.com/advice/0/how-do-you-choose-best-github-actions-workflow-your-project?lang=fr>