 

TP2 – Pipeline CI/CD avec GitHub Actions

**Année universitaire :** 2024–2025

**Enseignant :** Arij Azzabi

**Table des matières**

1. [Objectif](#_bookmark0) 2
2. [Étape 1 : Configuration d’un projet Python](#_bookmark1) 2
   1. [Créer un projet Python](#_bookmark2) 2
   2. [Initialiser Git et GitHub](#_bookmark3) 2
3. [Étape 2 : Mise en place de GitHub Actions](#_bookmark4) 3
   1. [Créer un fichier de workflow GitHub Actions](#_bookmark5) 3
   2. [Vérifier l’exécution du workflow](#_bookmark6) 4
4. [Étape 3 : Ajouter des vérifications supplémentaires](#_bookmark7) 4
   1. [Ajouter Flake8 pour la qualité du code](#_bookmark8) 4
   2. [Lancer les tests en local avant d’utiliser GitHub Actions](#_bookmark9) 4
5. [Étape 4 : Aller plus loin avec les tests](#_bookmark10) 4
   1. [Modifier le code Python](#_bookmark11) 5
   2. [Ajouter un nouveau test](#_bookmark12) 5
   3. [Lancer les tests et observer les résultats](#_bookmark13) 5
6. [Étape 5 : Réalisez plusieurs scénarios de test](#_bookmark14) 6
   1. [Scénario 1 : Vérification du type de données](#_bookmark15) 6
   2. [Scénario 2 : Mesure de performance basique](#_bookmark16) 6
   3. [Scénario 3 : Gestion d’erreur](#_bookmark17) 6
   4. [Scénario 4 : Test avec plusieurs paramètres](#_bookmark18) 7
   5. [Validation](#_bookmark19) 7
7. [Conclusion](#_bookmark20) 7

# Objectif

Dans cet atelier, nous allons intégrer **GitHub Actions** dans un projet Python pour

**automatiser les tests et le déploiement**. Nous allons apprendre à :

* Configurer un projet Python avec GitHub
* Automatiser les tests unitaires avec GitHub Actions
* Assurer la qualité du code avant le déploiement

# Étape 1 : Configuration d’un projet Python

## Créer un projet Python

mkdir my\_python\_project && cd my\_python\_project

Création du dossier de projet

Contenu du fichier main.py

def hello():

return "Hello, GitHub Actions!"

Contenu du fichier test\_main.py

from main import hello

def test\_hello():

assert hello() == "Hello, GitHub Actions!"

Contenu du fichier requirements.txt

pytest

## Initialiser Git et GitHub

Initialisation du dépôt Git et configuration GitHub

git init git add .

git commit -m "Initial commit"

git remote add origin https://github.com/your\_username/my\_python\_project.git git branch -M main

git push -u origin main

# Étape 2 : Mise en place de GitHub Actions

## Créer un fichier de workflow GitHub Actions

Créer le fichier ‘.github/workflows/python-ci.yml‘ :

name: Python CI

on: [push, pull\_request]

jobs:

test:

runs-on: ubuntu-latest

steps:

* name: Checkout repository uses: actions/checkout@v3
* name: Set up Python

uses: actions/setup-python@v3 with:

python-version: 3.9

* name: Install dependencies run: |

python -m pip install –upgrade pip pip install -r requirements.txt

* name: Run tests run: pytest

Workflow CI pour GitHub Actions

Chaque ligne dans le fichier YAML a un but spécifique :

* + - **name : Python CI** : Nom du workflow, visible sur le tableau de bord GitHub Actions.
    - **on : [push, pull\_request]** : Déclencheur lors d’un push ou d’une pull request.
    - **jobs :** : Définit les tâches à exécuter.
    - **test :** : Tâche dédiée aux tests.
    - **runs-on : ubuntu-latest** : Utilisation de l’environnement Ubuntu le plus récent.
    - **steps :** : Séquence d’étapes à suivre.
    - **Checkout repository** : Clone du dépôt :
      * name : Checkout r e p o s i to r y uses : a c t i o n s / checkout@v3
    - **Set up Python** : Installation de la version de Python :
      * name : Set up Python

uses : a c t i o n s / setup−python@v3 with :

python−v e r s i o n : 3 . 9

* + - **Install dependencies** : Installation des dépendances :
      * name : I n s t a l l dependencies run : |

python −m pip i n s t a l l −−upgrade pip pip i n s t a l l −r requirements . txt

* + - **Run tests** : Exécution des tests :
      * name : Run t e s t s run : pyte s t

## Vérifier l’exécution du workflow

Une fois le fichier poussé, allez dans l’onglet **Actions** sur GitHub pour vérifier l’exécution.

# Étape 3 : Ajouter des vérifications supplémentaires

## Ajouter Flake8 pour la qualité du code

Flake8 est un outil de vérification de style respectant la norme **PEP 8**. Il détecte :

* + - **Erreurs de syntaxe** (ex. : indentation incorrecte, absence de deux-points)
    - **Incohérences de style** (espaces, lignes vides manquantes)
    - **Erreurs logiques** potentielles

Installer ‘flake8‘ et l’ajouter dans ‘requirements.txt‘ :

pip install flake8

echo ’flake8’ » requirements.txt

Installation de Flake8

Ajouter cette étape dans ‘.github/workflows/python-ci.yml‘ :

- name: Run linter run: flake8 .

Étape GitHub Actions : exécuter le linter

## Lancer les tests en local avant d’utiliser GitHub Actions

pyte s t

# Étape 4 : Aller plus loin avec les tests

Expérimentation

Dans cette étape, vous allez modifier votre fonction pour y introduire un nouveau comportement, puis créer un test pour valider ce changement.

## Modifier le code Python

Modifiez votre fonction dans main.py pour qu’elle accepte un nom en paramètre : Listing 1 – Nouvelle version de la fonction hello()

def h e l l o ( name="GitHub Actions " ) : return f " Hello , {name } ! "

## Ajouter un nouveau test

Dans test\_main.py, ajoutez un second test pour valider le comportement avec un argu- ment :

Listing 2 – Ajout d’un test personnalisé from main import h e l l o

def test\_hello\_custom\_name ( ) :

a s s e r t h e l l o ( " EPSI " ) == " Hello , EPSI ! "

## Lancer les tests et observer les résultats

Instructions

* Exécutez les tests localement avec pytest
* Si tout est bon, poussez vos changements sur GitHub
* Vérifiez que le workflow GitHub Actions passe avec succès

# Étape 5 : Réalisez plusieurs scénarios de test

Objectif de cette étape

Vous allez renforcer vos compétences en tests automatisés en implémentant plusieurs types de scénarios réalistes. Chaque scénario couvre une problématique fréquente en développement.

## Scénario 1 : Vérification du type de données

Modifiez la fonction hello() pour qu’elle lève une exception si l’entrée n’est pas une chaîne de caractères, puis ajoutez un test qui vérifie ce comportement.

def hello(name="GitHub Actions"): if not isinstance(name, str):

raise TypeError("Le nom doit être une chaîne") return f"Hello, name!"

Ajout d’une vérification de type dans la fonction hello()

import pytest

from main import hello

def test\_hello\_type\_error(): with pytest.raises(TypeError):

hello(123)

Test du comportement en cas de type incorrect

## Scénario 2 : Mesure de performance basique

Ajoutez un test qui appelle la fonction 1000 fois pour vérifier qu’elle reste rapide et stable.

import time

def test\_hello\_performance(): start = time.time()

for \_ in range(1000): hello("EPSI")

duration = time.time() - start assert duration < 1

Test de performance simple

## Scénario 3 : Gestion d’erreur

Provoquez volontairement une erreur dans la fonction et assurez-vous que le test échoue bien (utile pour valider votre CI).

Instruction temporaire

Commentez la ligne return f"Hello, name!" et remplacez-la par raise ValueError("Erreur volontaire"), puis lancez le test. Ensuite, remettez la fonc- tion en état.

## Scénario 4 : Test avec plusieurs paramètres

Faites évoluer la fonction pour accepter un prénom et un nom, puis testez le formatage correct du message.

def hello(firstname="John", lastname="Doe"): return f"Hello, firstname lastname!"

Nouvelle version multi-paramètres de la fonction hello()

def test\_hello\_full\_name():

assert hello("Jane", "Smith") == "Hello, Jane Smith!"

Test de la fonction hello() avec deux paramètres

## Validation

Vérification finale

Une fois tous les tests ajoutés et passés localement, poussez vos modifications pour déclencher le pipeline GitHub Actions. Vérifiez que le workflow s’exécute avec suc- cès.

# Conclusion

Au terme de ce TP, vous avez été en mesure de :

* Créer et versionner un projet Python avec Git et GitHub
* Mettre en place un pipeline CI avec GitHub Actions pour automatiser les tests
* Intégrer un outil de qualité de code (Flake8) dans votre workflow
* Concevoir, écrire et valider plusieurs scénarios de tests unitaires
* Comprendre l’intérêt d’un pipeline automatisé dans un projet collaboratif

Résumé des apprentissages