## **Ejemplos de Configuración de Pipelines CI/CD con GitHub Actions para Python**

**1. Flujo de trabajo básico de CI:**

YAML

name: CI

on:

push:

branches: [ main ]

pull\_request:

branches: [ main ]

jobs:

build:

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v3

- name:

Set up Python uses: actions/setup-python@v2

with:

python-version: '3.9'

- name: Install dependencies

run: pip install -r requirements.txt

- name: Test with pytest

run: pytest

Use code [with caution.](file:///C:/faq#coding)

* **Explicación:**
* El flujo se activa en cada push a la rama main o en cada pull request.
* Se verifica el código, se instala el entorno de Python y las dependencias, y finalmente se ejecutan los tests con pytest.

**2. Flujo de trabajo más complejo con formato y envío:**

YAML

name: CI/CDon: push: branches: [ main ] jobs:

build:

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v3

- name: Set up Python

uses: actions/setup-python@v2

with:

python-version:

'3.9'

- name: Install dependencies

run: pip install -r requirements.txt

- name: Format code with black run: black . - name: Lint code with flake8 run: flake8 . - name: Test with pytest run: pytest deploy: runs-on: ubuntu-latest needs: build steps: - uses: actions/checkout@v3 - name: Deploy to Heroku run: git push heroku main

Use code [with caution.](file:///C:/faq#coding)

* **Explicación:**
* Además de las pruebas, se incluye formateo de código con black y linteo con flake8 para mantener la calidad del código.
* Se añade una etapa de despliegue que envía el código a Heroku.

## **Técnicas de Pruebas Automatizadas con Python y GitHub Actions**

* **Pytest:** El framework de pruebas más popular en Python. Permite escribir tests parametrizados, fixtures, y generar reportes detallados.
* **Unittests:** El framework de pruebas estándar de Python. Es más básico que pytest pero sigue siendo muy útil.
* **Mock:** Para aislar las unidades de código y simular comportamientos de dependencias externas.
* **Code coverage:** Para medir qué parte del código se ejecuta durante las pruebas.
* **Integración con otras herramientas:** GitHub Actions se integra con muchas otras herramientas como SonarQube para análisis estático de código, Codecov para cobertura de código, y más.

**Ejemplos de uso de pytest:**

Python

import pytest def test\_suma(): assert 2 + 2 == 4def test\_resta(): assert 5 - 3 == 2

Use code [with caution.](file:///C:/faq#coding)

**Personalización y Extensión**

* **Matrices de trabajo:** Ejecutar el mismo flujo de trabajo con diferentes configuraciones (e.g., diferentes versiones de Python, diferentes entornos).
* **Acciones de terceros:** Utilizar acciones creadas por la comunidad para tareas comunes (e.g., despliegue en AWS, notificaciones).
* **Scripts personalizados:** Crear scripts personalizados para tareas específicas.

**Consideraciones Adicionales**

* **Organización:** Mantén tus flujos de trabajo bien organizados y fáciles de entender.
* **Velocidad:** Optimiza tus flujos de trabajo para que se ejecuten rápidamente.
* **Fiabilidad:** Asegúrate de que tus pruebas sean fiables y consistentes.
* **Seguridad:** Protege tus credenciales y secretos.

**Recursos Adicionales**

* **Documentación oficial de GitHub Actions:** <https://docs.github.com/en/actions>
* **Documentación de pytest:** <https://docs.pytest.org/en/latest/>
* **Ejemplos de flujos de trabajo:** <https://github.com/actions>

**¿Quieres un ejemplo más específico?**

Si tienes un proyecto Python en mente y quieres saber cómo configurar un pipeline CI/CD con GitHub Actions, no dudes en compartir más detalles. Por ejemplo, puedes indicar:

* **Tipo de proyecto:** Web, API, aplicación de línea de comandos, etc.
* **Framework:** Django, Flask, FastAPI, etc.
* **Herramientas de despliegue:** Heroku, AWS, GCP, etc.
* **Pruebas específicas:** Unit tests, integración, end-to-end, etc.