Caso práctico 4

Integración continua con GitHub Actions en Python

Se pide

1. Crear (o usar) un repositorio con una función `es_primo` y sus pruebas unitarias.
2. Configurar un flujo de trabajo en GitHub Actions que:
   - se ejecute en `push` y `pull_request` sobre `main`,
   - instale dependencias,
   - pase `pylint`,
   - ejecute `pytest`.
3. Verificar en la pestaña **Actions** que el flujo se ejecuta correctamente.
4. Forzar un error y comprobar que el pipeline falla.

Claves de resolución

- El workflow debe ir en `.github/workflows/` para que GitHub Actions lo detecte.
- Para `push` y `pull_request` puedes filtrar por rama (`branches`).
- Conviene usar acciones actuales: `actions/checkout@v6` (la serie v6 es la vigente).
- Para Python, usa `actions/setup-python@v6`, indicando explícitamente `python-version`; también admite caché de `pip`.
- En runners de GitHub, hoy puedes usar `ubuntu-latest` o fijar `ubuntu-24.04`.

Paso 1. Preparar estructura del proyecto

Crea (o deja) esta estructura:

```
mi-repo/
├─ es_primo.py
├─ test_es_primo.py
├─ requirements.txt
└─ .github/
   └─ workflows/
      └─ python-app.yml
```

Explicación:
Separar código, tests y dependencias facilita que CI sea reproducible (mismo comportamiento en local y en GitHub).

Paso 2. Código Python (revisado)

es_primo.py

```
def es_primo(numero: int) -> bool:
    if numero < 2:
        return False
    if numero == 2:
        return True
    if numero % 2 == 0:
        return False

    limite = int(numero ** 0.5) + 1
    for n in range(3, limite, 2):
        if numero % n == 0:
            return False
    return True
```

Explicación:
Esta versión evita falsos positivos y es más eficiente (solo comprueba divisores impares hasta la raíz).

Paso 3. Tests unitarios

test_es_primo.py

```
from es_primo import es_primo

def test_2_es_primo():
    assert es_primo(2) is True

def test_4_no_es_primo():
    assert es_primo(4) is False

def test_9_no_es_primo():
    assert es_primo(9) is False

def test_13_es_primo():
    assert es_primo(13) is True

def test_menores_que_2_no_son_primos():
    assert es_primo(0) is False
    assert es_primo(1) is False
    assert es_primo(-5) is False
```

Explicación:
Añadimos casos límite (`0`, `1`, negativos) para mejorar la calidad de pruebas.

Paso 4. Dependencias del proyecto

`requirements.txt`

```
pytest>=8.0,<10
pylint>=3.0,<5
```

Explicación:
Usar rangos de versión evita "romper" por cambios mayores inesperados, pero mantiene el proyecto actualizado.

Paso 5. Workflow actualizado (`python-app.yml`)

Este sería el reemplazo de tu workflow antiguo (v2/v1 y rama `master`).

```
name: CI Python (actualizado)

on:
  push:
    branches: ["main"]
  pull_request:
    branches: ["main"]
  workflow_dispatch:

permissions:
  contents: read

jobs:
  lint-and-test:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        python-version: ["3.12", "3.13"]

    steps:
      - name: Checkout del repositorio
        uses: actions/checkout@v6

      - name: Configurar Python
        uses: actions/setup-python@v6
        with:
          python-version: ${{ matrix.python-version }}
          cache: "pip"

      - name: Instalar dependencias
        run: |
          python -m pip install --upgrade pip
          pip install -r requirements.txt

      - name: Análisis estático (pylint)
        run: pylint --fail-under=8 es_primo.py

      - name: Ejecutar pruebas (pytest)
        run: pytest -q
```

Explicación de puntos clave:

- `workflow_dispatch` te permite lanzarlo manualmente desde Actions.
- `matrix` prueba varias versiones de Python.
- `permissions: contents: read` sigue el principio de mínimo privilegio recomendado.
- `--fail-under=8` hace fallar el pipeline si el score de pylint baja de 8. `--fail-under` está soportado oficialmente.

Sobre las versiones y recomendaciones del workflow: checkout v6, setup-python v6, versión explícita y caché de pip, además de permisos recomendados.

Paso 6. Comprobar en local antes de subir

```
python -m pip install -r requirements.txt
pylint --fail-under=8 es_primo.py
pytest -q
```

Explicación:
Si pasa en local, reduces errores de CI al subir.

Paso 7. Subir cambios y revisar Actions

1. `git add .`
2. `git commit -m "Configuro CI actualizado con GitHub Actions"`
3. `git push origin main`
4. Abrir pestaña **Actions** y revisar el job.

Nota didáctica:
Si `pytest` falla, devuelve código de error y GitHub marca la ejecución como fallida (por ejemplo, exit code 1 cuando hay tests fallidos).

Paso 8. Prueba de fallo controlada

Cambia intencionalmente una aserción en `test_es_primo.py` para que falle, sube commit y comprueba:

- Estado rojo en Actions,
- logs con el test fallido.

Después corrige y vuelve a subir.