Proyecto simple con Tkinter, tests, .exe y CI/CD en GitHub

0. Estructura del proyecto

```
mi_suma_app/
├─ app.py            # GUI Tkinter (entrada para ejecutar localmente)
├─ calc.py           # lógica: función sum_two(a,b) — fácil de testear
├─ requirements.txt  # generado automáticamente
├─ tests/
│  └─ test_calc.py   # tests unitarios con unittest
└─ .github/
   └─ workflows/
      ├─ ci.yml      # ejecutar tests en PRs / pushes
      └─ build-exe.yml  # compilar .exe cuando llegue a main
```

1. Crear virtual environment y generar requirements.txt

1. Desde la raíz del proyecto:

```
# en Windows (PowerShell)
python -m venv .venv
```

2. Activarlo:

   Windows (PowerShell):

   .\.venv\Scripts\Activate.ps1

3. Instalar dependencias mínimas:

```
pip install pyinstaller
```

unittest → viene con python

4. **Generar requirements.txt automáticamente** (guarda versiones actuales):

```
pip freeze > requirements.txt
```

pip freeze > requirements.txt es la forma estándar para crear el fichero de requerimientos con las versiones instaladas.

2. Código: app Tkinter + lógica separada + unit test local

Es buena práctica separar la **lógica** de la UI para poder testear sin arrancar la ventana.

calc.py — lógica (fácil de testear)

```
# calc.py
def sum_two(a, b):
    """
    Convierte a y b a float si es posible y devuelve la suma.
    Lanza ValueError si los valores no son números.
    """
    try:
        return float(a) + float(b)
    except Exception as e:
        raise ValueError("Entradas no numéricas") from e
```

app.py — GUI en Tkinter que usa sum_two

```
# app.py
import tkinter as tk
from tkinter import messagebox
from calc import sum_two

def on_sum():
    a = entry_a.get()
    b = entry_b.get()
    try:
        resultado = sum_two(a, b)
        label_result.config(text=f"Resultado: {resultado}")
    except ValueError:
        messagebox.showerror("Error", "Introduce dos números válidos")

root = tk.Tk()
root.title("Suma rápida")

frame = tk.Frame(root, padx=10, pady=10)
frame.pack()
```

```
tk.Label(frame, text="A:").grid(row=0, column=0)
entry_a = tk.Entry(frame)
entry_a.grid(row=0, column=1)

tk.Label(frame, text="B:").grid(row=1, column=0)
entry_b = tk.Entry(frame)
entry_b.grid(row=1, column=1)

btn = tk.Button(frame, text="Sumar", command=on_sum)
btn.grid(row=2, column=0, columnspan=2, pady=5)

label_result = tk.Label(frame, text="Resultado: ")
label_result.grid(row=3, column=0, columnspan=2)

if __name__ == "__main__":
  root.mainloop()
```

tests/test_calc.py — unit tests con unittest (local)

```
# tests/test_calc.py
import unittest
from calc import sum_two

class TestCalc(unittest.TestCase):
  def test_sum_integers(self):
    self.assertEqual(sum_two(2, 3), 5.0)

  def test_sum_strings_numbers(self):
    self.assertAlmostEqual(sum_two("1.5", "2.25"), 3.75)

  def test_invalid_input(self):
    with self.assertRaises(ValueError):
```

```
        sum_two("abc", 3)


if __name__ == "__main__":
    unittest.main()
```

Cómo ejecutar los tests localmente:

```
# desde la raíz (con el venv activado)
python -m unittest discover -v
# o para un fichero concreto:
python -m unittest tests.test_calc
```

3. Generar un .exe localmente usando PyInstaller

Con el venv activado e instaladas dependencias (pyinstaller en requirements.txt), desde la raíz:

```
# comando básico: un solo ejecutable, ventana sin consola
pyinstaller --onefile --windowed app.py
```

- Resultado: el .exe aparecerá en dist/app.exe.
- Opcional: añadir --name MiSumaApp para dar otro nombre, o --icon=app.ico para icono.
- Documentación oficial y opciones: pyinstaller [options] script (usa --onefile para un único .exe).

Notas prácticas:

- Si la app usa ficheros adicionales (iconos, imágenes, archivos), hay que incluirlos en el spec o pasarlos con --add-data.
- En Windows al ejecutar pyinstaller se obtiene un .exe nativo

4. GitHub: impedir push directo a la rama principal (obligar PR y revisión)

En GitHub (repositorio):

1. Ir a **Settings > Branches > Branch protection rules**.
2. Click **Add rule**, escribe main (o el nombre de la rama por defecto).
3. Marcar **Require a pull request before merging** (o "Require pull request reviews before merging").
4. Marcar **Require approvals** y poner el número de revisores (p.ej. 1).
5. (Opcional, recomendado) Marcar **Require status checks to pass before merging** y seleccionar los checks (por ejemplo: ci o el nombre del workflow que ejecuta

tests). Importante: Si no aparecen los checks revisa que tenga nombre los Jobs que se ejecutan.

6. Guardar la regla.

5. En GitHub: ejecutar unit tests automáticamente antes de poder subir/mergear

Esto se logra combinando:

- Un **GitHub Actions** workflow que ejecuta los tests en cada push y PR.
- Y en la regla de branch protection, exigir que dicho *status check* pase (seleccionando el nombre del job/acción).

Ejemplo de workflow para tests: .github/workflows/ci.yml

```
# .github/workflows/ci.yml

name: CI - tests


on:

 push:

  branches: ["**"]    # se ejecuta en pushes

 pull_request:

  branches: ["main"]  # y en PRs hacia main


jobs:

 test:

  name: Run unit tests

  runs-on: ubuntu-latest


  steps:

   - uses: actions/checkout@v4


   - name: Set up Python

    uses: actions/setup-python@v4

    with:

     python-version: "3.x"
```

```
    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        if [ -f requirements.txt ]; then pip install -r requirements.txt; fi

    - name: Run unit tests
      run: |
        python -m unittest discover -v
```

- El job test crea un *status check* con nombre test/CI - tests visible en GitHub.
- En las reglas de protección de rama, marca **Require status checks to pass before merging** y selecciona CI - tests (u opción similar) para impedir merges hasta que pasen.

6. Pipeline: cuando algo llegue a main, generar automáticamente el .exe

Flujo recomendado:

- Workflow A (ci.yml) corre tests en PRs y pushes. PRs no pueden mergear si no pasan.
- Workflow B (build-exe.yml) se dispara **cuando se hace push a main** (es decir, ya pasó el test) y compila el .exe en un runner Windows, luego sube el artefacto.

Ejemplo de workflow para compilar exe en Windows: .github/workflows/build-exe.yml

```
# .github/workflows/build-exe.yml
name: Build EXE

on:
  push:
    branches: ["main"]

jobs:
  build-windows:
    name: Build Windows EXE
    runs-on: windows-latest
```

```
  steps:
    - uses: actions/checkout@v4

    - name: Set up Python
      uses: actions/setup-python@v4
      with:
        python-version: "3.x"

    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        if (Test-Path requirements.txt) { pip install -r requirements.txt } else { pip install pyinstaller }

    - name: Build exe with PyInstaller
      run: |
        pyinstaller --onefile --windowed app.py
      shell: bash

    - name: Upload artifact
      uses: actions/upload-artifact@v4
      with:
        name: MiSumaApp-windows
        path: dist/*.exe
```

Resumen de pasos operativos

Local:

python -m venv .venv → activar venv.

1. pip install pyinstaller (y otras libs si hace falta).
2. Crear calc.py, app.py, tests/test_calc.py.
3. Ejecutar python -m unittest discover -v para comprobar tests.
4. pip freeze > requirements.txt.

5. pyinstaller --onefile --windowed app.py → dist/app.exe.

En GitHub:

1. Subir repo con la estructura y requirements.txt.
2. Añadir .github/workflows/ci.yml (tests) y .github/workflows/build-exe.yml (build en main).
3. En Settings → Branches → Add rule para main:
    - Requerir PRs y al menos 1 aprobación.
    - Requerir paso de status checks y seleccionar el workflow de tests.
4. PR → checks automáticos (tests) → solo si pasan y hay revisión → merge → workflow build-exe genera .exe automáticamente.

Documentación y referencias oficiales útiles:

- pip freeze > requirements.txt (forma estándar para generar requirements).
- PyInstaller — uso y opciones (--onefile, --windowed).
- Branch protection rules en GitHub (require PR reviews, require status checks).
- GitHub Actions — ejemplo de workflow para Python (tests) y publicar artefactos (PyInstaller ejemplos).