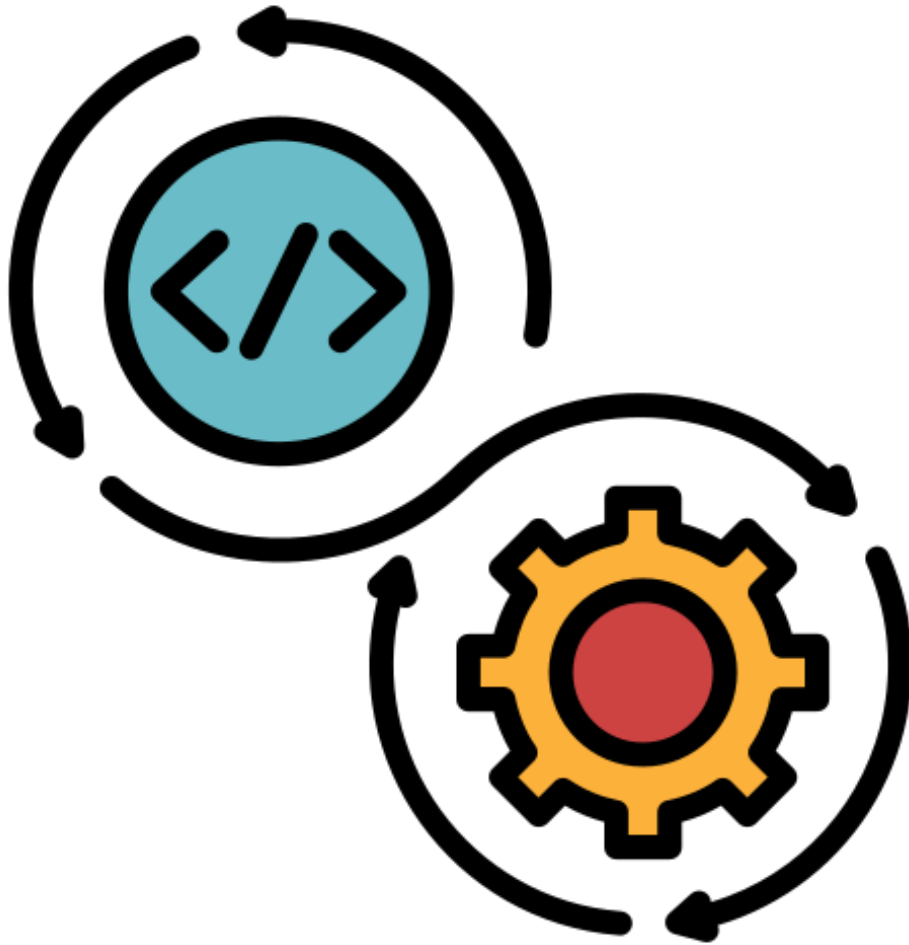


DevOps

Sprint 1 Lab 5



Felipe Izquierdo Romero

Índice

<u>1</u>	<u>Descripción del problema</u>
<u>2</u>	<u>Desarrollo</u>

1. Descripción del problema

Para el laboratorio de DevOps se deberá realizar un sencillo módulo de Python con cuatro funciones (suma, resta, multiplicación y división). Estos cuatro módulos deberán estar subidos a un repositorio de GitHub.

Además, habrá que realizar unas pruebas unitarias para cada función y automatizarlas en el repositorio de GitHub con la herramienta GitActions.

2. Desarrollo

Empezaremos con la creación del repositorio en GitHub

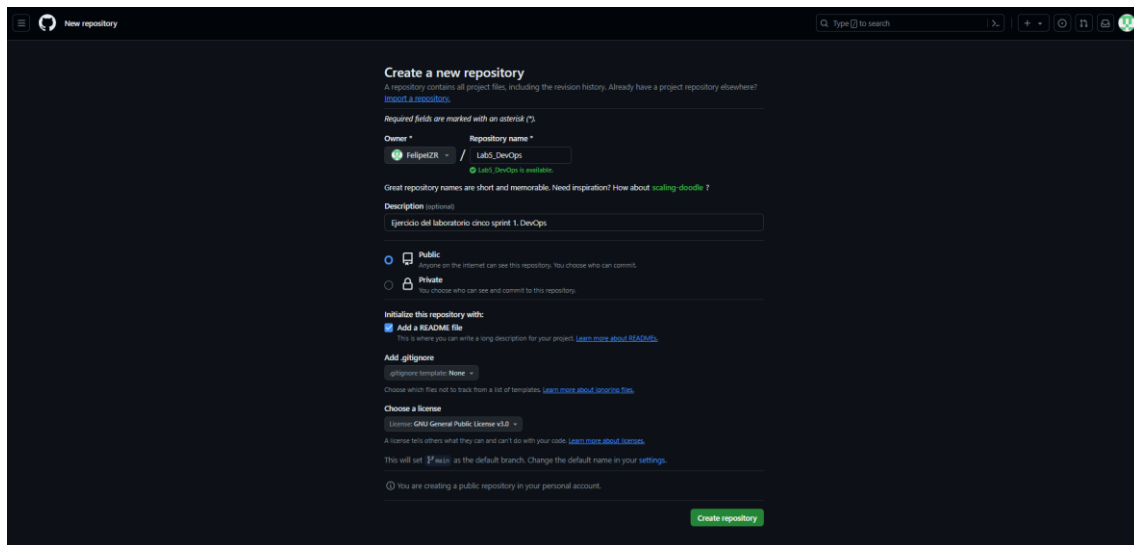


Ilustración 1- Creación de repositorio en GitHub

A continuación, clonamos el repositorio en local

```
MINGW64:/c/Users/felix/SprintsBootcamp/Sprint_1/Lab_5/Rep_Lab5_DevOps
felix@FelipeIzRo MINGW64 ~
$ cd "C:\Users\felix\SprintsBootcamp\Sprint_1\Lab_5\Rep_Lab5_DevOps"

felix@FelipeIzRo MINGW64 ~/SprintsBootcamp/Sprint_1/Lab_5/Rep_Lab5_DevOps
$ git clone https://github.com/FelipeIZR/Lab5_DevOps.git
Cloning into 'Lab5_DevOps'...
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (4/4), 12.77 KiB | 1006.00 KiB/s, done.

felix@FelipeIzRo MINGW64 ~/SprintsBootcamp/Sprint_1/Lab_5/Rep_Lab5_DevOps
$
```

Ilustración 2- Clonación del repositorio

Como paso intermedio crearemos una plantilla del archivo .gitignore en la web <https://www.toptal.com/developers/gitignore>



Ilustración 3- Generador web de .gitignore

Después en el repositorio creado añadimos un archivo en la raíz con el nombre .gitignore y copiamos el texto que nos generó esta web.

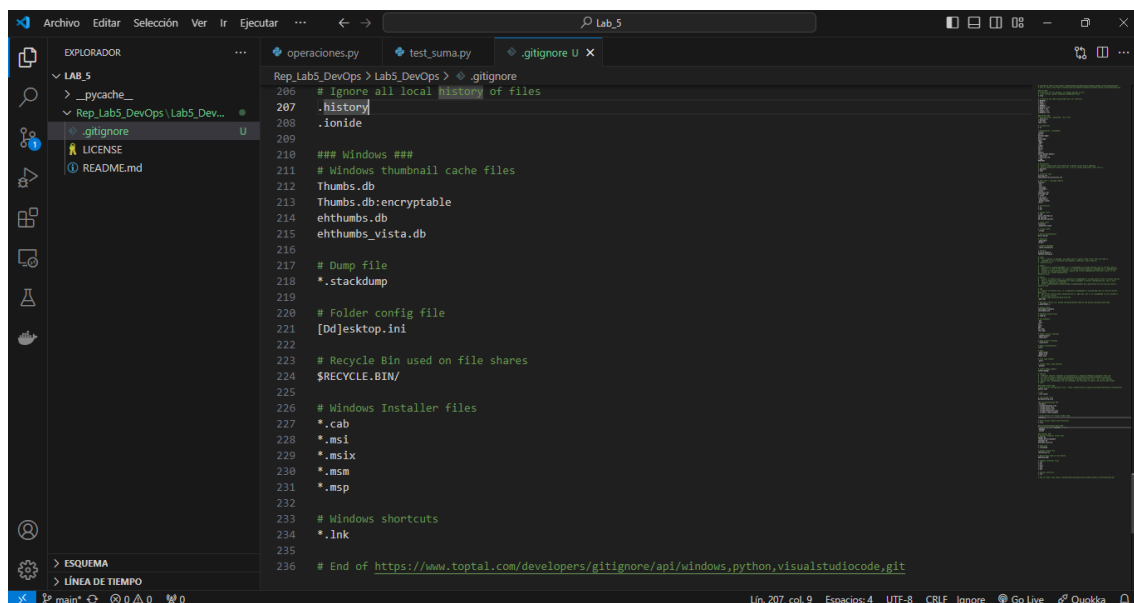


Ilustración 4- Creación de .gitignore

En el repositorio local crearemos el módulo de operaciones.py y test.py.

En operaciones estará el código necesario para crear las cuatro funciones necesarias para completar el laboratorio.

```
1  def suma (a,b):
2      try:
3          if type(a) == bool or type(b) == bool:
4              return 'error'
5              return float(a) + float(b)
6      except:
7          return 'error'
8  def resta (a,b):
9      try:
10         if type(a) == bool or type(b) == bool:
11             return 'error'
12             return float(a) - float(b)
13         except:
14             return 'error'
15
16  def multiplicacion(a,b):
17      try:
18         if type(a) == bool or type(b) == bool:
19             return 'error'
20             return float(a) * float(b)
21         except:
22             return 'error'
23  def division(a,b):
24      try:
25         if type(a) == bool or type(b) == bool:
26             return 'error'
27             return float(a) / float(b)
28         except:
29             return 'error'
30
```

Ilustración 5- Operaciones

En test.py los test unitarios que deberán pasar mis cuatro funciones

```
1 import unittest
2 import operaciones as operaciones
3
4 class TestSumar (unittest.TestCase):
5     def test_suma(self):
6         # Prueba de sumar
7         self.assertEqual(operaciones.suma(1,1),2)
8         self.assertEqual(operaciones.suma(5,-1),4)
9         self.assertEqual(operaciones.suma(-3,-3),-6)
10        self.assertEqual(operaciones.suma(3,-4),-1)
11        # Prueba parametros incorrectos
12        self.assertEqual(operaciones.suma("a",-4),'error')
13        self.assertEqual(operaciones.suma(True,-4),'error')
14
15    def test_resta(self):
16        # Prueba de restar
17        self.assertEqual(operaciones.resta(1,1),0)
18        self.assertEqual(operaciones.resta(5,-1),6)
19        self.assertEqual(operaciones.resta(-1,-1),0)
20        self.assertEqual(operaciones.resta(5,10),-5)
21        # Prueba parametros incorrectos
22        self.assertEqual(operaciones.resta("a",10),'error')
23        self.assertEqual(operaciones.resta(True,10),'error')
24
25    def test_multiplicacion(self):
26        # Prueba de multiplicar
27        self.assertEqual(operaciones.multiplicacion(1,1),1)
28        self.assertEqual(operaciones.multiplicacion(2,5),10)
29        self.assertEqual(operaciones.multiplicacion(3,-3),-9)
30        self.assertEqual(operaciones.multiplicacion(20,0),0)
31        # Prueba parametros incorrectos
32        self.assertEqual(operaciones.multiplicacion("a",2),'error')
33        self.assertEqual(operaciones.multiplicacion(True,4),'error')
34
35    def test_division(self):
36        # Prueba de dividir
37        self.assertEqual(operaciones.division(1,1),1.0)
38        self.assertEqual(operaciones.division(5,2),2.5)
39        self.assertEqual(operaciones.division(20,10),2.0)
40        # Prueba parametros incorrectos
41        self.assertEqual(operaciones.division(1,0),'error')
42        self.assertEqual(operaciones.division("a",1),'error')
43        self.assertEqual(operaciones.division(True,1),'error')
44
45 if __name__ == '__main__':
46     unittest.main()
47
```

Ilustración 6- Test de operaciones

Después crearemos dos subdirectorios .github en la raíz del repositorio y dentro workflows para que GitHub reconozca el fichero de configuración YAML que será el encargado de definir el WorkFlow que queremos que pase nuestro código en GitHub de manera automática.

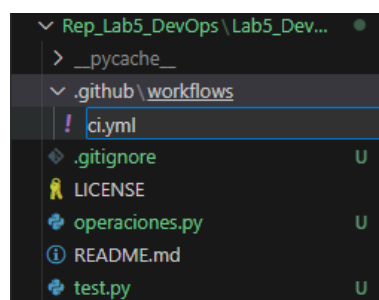


Ilustración 7- Árbol de directorios

El fichero YAML contiene las siguientes instrucciones

```
1  name: Python CI
2
3  on:
4    push:
5      branches: [ main ]
6    pull_request:
7      branches: [ main ]
8
9  jobs:
10   build:
11     runs-on: ubuntu-latest
12     steps:
13       - name: Checkout repository
14         uses: actions/checkout@v4
15
16       - name: Set up Python
17         uses: actions/setup-python@v4
18         with:
19           python-version: 3.11.16
20
21       - name: Run tests
22         run:
23           python -m unittest test.py
24
```

Ilustración 8- Fichero ci.yml

Para probar la ejecución del workflow bastaría con hacer un commit y hacer un push al repositorio de GitHub, para ver el proceso haz click en el link

[Link Video](#)