

«Talento Tech»

# Automation Testing

Clase 4



# Clase N° 4 | Introducción a Pytest y Automatización Básica

## Temario

- Instalación de Pytest y preparación del proyecto
- Creación de casos de prueba
- Decoradores y aserciones
- Fixtures: qué son, para qué sirven y variantes
- Parametrización y markers personalizados
- Reporte HTML nativo

## Objetivos de la clase

En esta clase nos adentraremos en el mundo de las pruebas automatizadas con Pytest, una herramienta clave para garantizar la calidad del código. Aprenderemos a instalar Pytest y a ejecutarlo sobre nuestros proyectos locales, comprendiendo el rol fundamental que cumplen los decoradores en su funcionamiento. Construiremos pruebas unitarias sólidas mediante aserciones claras y reutilizables, y utilizaremos fixtures para preparar datos de forma ordenada. Además, exploraremos cómo ampliar la cobertura con `@pytest.mark.parametrize`, cómo clasificar nuestros tests con markers personalizados (como *smoke*, *slow* o *exception*), y cómo filtrarlos desde la línea de comandos. Finalmente, generaremos un reporte HTML autosuficiente con los resultados, ideal para documentar y compartir el estado de las pruebas.

# Pytest



**Pytest** es el *framework de testing* más popular en Python. Se utiliza para automatizar pruebas de software, especialmente aquellas que validan funciones y comportamientos dentro de un proyecto.

Hoy lo instalaremos, convertiremos nuestro módulo `calculadora.py` en una mini-suite de pruebas y aprenderemos a usar *fixtures*, *markers* y parametrización para mantener el código de tests limpio y escalable. Al final generarás un reporte HTML que cualquiera podrá leer sin instalar nada extra.

## ¿Para qué se usa?

- Para verificar automáticamente que las funciones del código (por ejemplo: sumar(), dividir(), etc.) devuelven lo esperado.
- Para detectar errores temprano en el desarrollo.
- Para reducir el riesgo de que algo se rompa al modificar código.
- Para documentar comportamientos esperados, lo cual también ayuda a otros desarrolladores.

## Instalación y preparación del proyecto

### Paso 1: Ejercicios previos

Asegurate de tener tu archivo `calculadora.py` con las funciones básicas.

Si no lo tenés podés copiarlo de aca:

<https://github.com/emilianospinoso/pre-entrega-automation-testing/blob/main/calculadora.py>

### Paso 2: Instalación

Podés instalar Pytest (y opcionalmente el plugin para generar reportes HTML) usando `pip`:

```
pip install pytest pytest-html
```

```
emilianospinoso@ARJRF6K3:~/Escritorio/automation_project$ pip install pytest pytest-html
Defaulting to user installation because normal site-packages is not writeable
Collecting pytest
  Downloading pytest-8.3.5-py3-none-any.whl (343 kB)
    343.6/343.6 KB 5.3 MB/s eta 0:00:00
Collecting pytest-html
  Downloading pytest_html-4.1.1-py3-none-any.whl (23 kB)
Collecting iniconfig
  Downloading iniconfig-2.1.0-py3-none-any.whl (6.0 kB)
Collecting exceptiongroup<=1.0.0rc8
  Downloading exceptiongroup-1.3.0-py3-none-any.whl (16 kB)
Collecting pluggy<2,>=1.5
  Downloading pluggy-1.6.0-py3-none-any.whl (20 kB)
Collecting tomli>=1
  Downloading tomli-2.2.1-py3-none-any.whl (14 kB)
Requirement already satisfied: packaging in /home/emilianospinoso/.local/lib/python3.10/site-packages (from pytest) (24.2)
Collecting Jinja2>=3.0.0
  Downloading Jinja2-3.1.6-py3-none-any.whl (134 kB)
    134.9/134.9 KB 51.3 MB/s eta 0:00:00
Collecting pytest-metadata>=2.0.0
  Downloading pytest_metadata-3.1.1-py3-none-any.whl (11 kB)
Collecting typing-extensions>=4.6.0
  Downloading typing_extensions-4.13.2-py3-none-any.whl (45 kB)
```

`pytest-html` es útil si querés compartir los resultados de los tests en formato visual (HTML), por ejemplo al adjuntarlos en un Pull Request.

### Paso 3: Estructura del proyecto recomendada.

Para que Pytest funcione correctamente, es importante organizar tu proyecto. Una estructura básica sería:

- Crea una carpeta `tests/` y dentro un archivo `test_calculadora.py`.

## Creación de casos de prueba

Un test case o un caso de prueba en Pytest es **cualquier función cuyo nombre comienza con `test_`**. Esto permite que Pytest las descubra automáticamente.

Link al repo:

[https://github.com/emilianospinoso/pre-entests/test\\_calculadora.py](https://github.com/emilianospinoso/pre-entests/test_calculadora.py)

Supongamos que tenemos un archivo `calculadora.py`

Ahora creamos `tests/test_calculadora.py`

```
from ..calculadora import sumar, restar, multiplicar, dividir
import pytest

def test_sumar_positivo():
    assert sumar(2, 3) == 5

def test_sumar_negativo():
    assert sumar(-2, -3) == -5

def test_dividir_por_cero():
    with pytest.raises(ValueError):
        dividir(1, 0)
```



## Explicación:

### 1. Importación de funciones

```
from ..calculadora import sumar, restar, multiplicar, dividir
```

Este import trae las funciones definidas en calculadora.py. Si la estructura del proyecto está bien organizada, Pytest podrá acceder a esas funciones desde el archivo de test.

### 2. Test de suma positiva

```
def test_sumar_positivo():
    assert sumar(2, 3) == 5
```

Este test verifica que sumar 2 + 3 da como resultado 5.

Usamos assert para decirle a Pytest: “Esto debería ser verdadero”.

Si no lo es, el test falla y se muestra la diferencia.

### 3. Test de suma negativa

```
def test_sumar_negativo():
    assert sumar(-2, -3) == -5
```

Igual que el anterior, pero con números negativos.

Es importante validar distintos escenarios para aumentar la cobertura del código.

### 4. Test de división por cero (manejo de errores)

```
def test_dividir_por_cero():
    with pytest.raises(ValueError):
        dividir(1, 0)
```

Este test comprueba que dividir entre cero lanza un error, como debería.

Pytest tiene esta forma especial de validar excepciones con `pytest.raises(...)`.

Si el error no ocurre, el test falla.

### 5. ¿Cómo corro estos tests?

Estando dentro de la carpeta raíz del proyecto, desde consola:

Ejecutá `pytest -v` y verás algo similar:

```
emilianospinoso@ARJRFFGK3:~/Escritorio/automation_project$ pytest -v
===== test session starts =====
platform linux -- Python 3.10.12, pytest-8.3.5, pluggy-1.6.0 -- /usr/bin/python3
cachedir: .pytest_cache
metadata: {'Python': '3.10.12', 'Platform': 'Linux-6.8.0-59-generic-x86_64-with-glibc2.35', 'Packages': {'pytest': '8.3.5', 'pluggy': '1.6.0'}, 'Plugins': {'html': '4.1.1', 'metadata': '3.1.1'}, 'JAVA_HOME': '/usr/lib/jvm/amazon-corretto-11.0.6.10.1-linux-x64'}
rootdir: /home/emilianospinoso/Escritorio/automation_project
plugins: html-4.1.1, metadata-3.1.1
collected 3 items

tests/test_calculadora.py::test_sumar_positivo PASSED [ 33%]
tests/test_calculadora.py::test_sumar_negativo PASSED [ 66%]
tests/test_calculadora.py::test_dividir_por_cero PASSED [100%]

===== 3 passed in 0.01s =====
```

- **PASSED** (✓) indica éxito.
- Si un assert falla, Pytest mostrará **FAILED** (✗) y resaltará la diferencia entre valores esperados y reales.

**Tip:** Si cuando corres el test te da un error diciendo que no encuentra el módulo probablemente sea porque te falte el archivo `__init__.py`. Puedes crearlos desde el VsCode o bien ejecutando estos comandos:

```
/Escritorio/automation_project$ touch tests/__init__.py
```

```
/Escritorio/automation_project$ touch __init__.py
```

## Decoradores y aserciones

### ¿Qué es un decorador?

Los **decoradores** en Python son funciones que “envuelven” a otra para modificar su comportamiento sin tocar su código interno.

Pytest se apoya en decoradores para tres tareas claves:

Decorador	¿Para qué sirve?
@pytest.fixture	Declarar fixtures: piezas reutilizables que preparan datos u objetos antes del test y, opcionalmente, los limpian después. Evitan código duplicado y centralizan la configuración.
@pytest.mark.parametrize	Parametrizar un mismo test con varias combinaciones de datos. Pytest crea un sub-test por cada fila del parámetro, aumentando la cobertura sin duplicar funciones.
@pytest.mark.<etiqueta>	Etiquetar (marcar) tests para agruparlos o alterar su ejecución (p. ej. saltar, esperar fallo). Permite filtrar subconjuntos desde la CLI (pytest -m smoke).

### @pytest.fixture Preparar datos reutilizables

Un **fixture** es una función especial (decorada con `@pytest.fixture`) que se ejecuta **antes** de tu prueba para preparar datos, objetos o estado, y, opcionalmente, **después** para limpiarlos. Reduce repetición y centraliza la configuración.

```
@pytest.fixture
def numeros_enteros():
    """Prepara dos enteros comunes."""
    return 20, 5
```

- Esto crea una fixture que devuelve una tupla (20, 5).
- Sirve para evitar repetir estos datos en múltiples tests.

Usamos la fixture así:

```
def test_dividir_enteros(numeros_enteros):
    a, b = numeros_enteros
    assert dividir(a, b) == 4
```

## @pytest.mark.parametrize Parametrización y markers: Probar múltiples valores en un solo test

Permite probar múltiples combinaciones sin duplicar funciones.

```
@pytest.mark.parametrize(
    "a,b,esperado",
    [
        (1, 2, 3),
        (-1, -1, -2),
        (2.5, 0.5, 3),
        (0, 0, 0)
    ]
)
def test_sumar_varios(a, b, esperado):
    assert sumar(a, b) == esperado
```

## @pytest.mark.<etiqueta> Agrupar o filtrar tests

```
@pytest.mark.smoke
def test_sumar_smoke():
    assert sumar(5, 5) == 10

@pytest.mark.exception
def test_dividir_por_cero():
    with pytest.raises(ValueError):
        dividir(1, 0)
```

Las etiquetas (**smoke**, **exception**, etc.) permiten **filtrar qué tests ejecutar**.

Por ejemplo:

```
pytest -v -m smoke          # solo ejecuta tests críticos
pytest -v -m exception      # solo ejecuta tests que validan errores
```

Para que estas etiquetas funcionen sin advertencias, agregá un archivo **pytest.ini**:

```
# pytest.ini
[pytest]
markers =
    smoke: pruebas críticas y rápidas
    exception: casos que validan manejo de errores
```

## Implementación en la calculadora.

Vamos ahora a realizar un ejemplo mucho mas avanzado con la misma calculadora que tenemos. En él vamos a aplicar lo visto en los puntos 3, 4 y 5.

Para realizar el ejemplo deberás crear un archivo `pytest.ini` en la raíz de tu proyecto para registrar las etiquetas personalizadas y evitar advertencias:

[Link al archivo](#)

```
[pytest]

markers =

    smoke: pruebas críticas y rápidas

    exception: casos que validan manejo de errores
```

### Código

Este es el nuevo archivo de pruebas que implementa todos los conceptos mencionados:

Podés obtenerlo del repo:

[https://github.com/emilianospinoso/pre-entrega-automation-testing/blob/main/tests/test\\_calculadora\\_avanzado.py](https://github.com/emilianospinoso/pre-entrega-automation-testing/blob/main/tests/test_calculadora_avanzado.py)

## Explicación detallada del código

### a. Fixtures con `@pytest.fixture`

Creamos dos fixtures:

- `numeros_enteros`: Devuelve una tupla con dos enteros (20, 5) que se pueden usar en múltiples tests.
- `numeros_decimales`: Devuelve una tupla con dos decimales (0.1, 0.2) para pruebas de precisión.

Las fixtures preparan los datos una sola vez y pueden ser inyectadas en varios tests, reduciendo la duplicación de código.

### b. Parametrización con `@pytest.mark.parametrize`

Implementamos tests parametrizados para las funciones `sumar` y `restar`. Cada combinación de parámetros crea un test independiente. Esto permite:

- Probar múltiples escenarios con una sola función de test
- Aumentar la cobertura sin duplicar código
- Ver claramente qué valores concretos fallaron si hay un problema



### c. Etiquetas con `@pytest.mark.<etiqueta>`

Para qué sirven las siguientes etiquetas:

`@pytest.mark.smoke`: Para tests rápidos y críticos que verifican la funcionalidad básica.

`@pytest.mark.exception`: Para tests que verifican el manejo correcto de excepciones.

Estas etiquetas permiten ejecutar subconjuntos de tests con comandos como:

```
pytest -m smoke -v
```

## Aserciones de precisión

He utilizado `pytest.approx()` para realizar comparaciones con tolerancia:

- En `test_multiplicar_preciso`: Maneja la imprecisión inherente de multiplicar números flotantes.
- En `test_dividir_preciso`: Maneja la imprecisión de representar un número periódico.

### ¿Cómo ejecutar estos tests?

Para ejecutar todos los tests:

```
pytest -v
```

```
tests/test_calculadora.py::test_sumar_positivo PASSED [ 5%]
tests/test_calculadora.py::test_sumar_negativo PASSED [ 11%]
tests/test_calculadora.py::test_dividir_por_cero PASSED [ 16%]
tests/test_calculadora_avanzado.py::test_dividir_enteros PASSED [ 22%]
tests/test_calculadora_avanzado.py::test_multiplicar_enteros PASSED [ 27%]
tests/test_calculadora_avanzado.py::test_sumar_varios[1-2-3] PASSED [ 33%]
tests/test_calculadora_avanzado.py::test_sumar_varios[-1--1--2] PASSED [ 38%]
tests/test_calculadora_avanzado.py::test_sumar_varios[2.5-0.5-3] PASSED [ 44%]
tests/test_calculadora_avanzado.py::test_sumar_varios[0-0-0] PASSED [ 50%]
tests/test_calculadora_avanzado.py::test_restar_varios[10-5-5] PASSED [ 55%]
tests/test_calculadora_avanzado.py::test_restar_varios[-1--2-1] PASSED [ 61%]
tests/test_calculadora_avanzado.py::test_restar_varios[3.5-1.5-2] PASSED [ 66%]
tests/test_calculadora_avanzado.py::test_restar_varios[0-0-0] PASSED [ 72%]
tests/test_calculadora_avanzado.py::test_restar_smoke PASSED [ 77%]
tests/test_calculadora_avanzado.py::test_sumar_smoke PASSED [ 83%]
tests/test_calculadora_avanzado.py::test_dividir_por_cero PASSED [ 88%]
tests/test_calculadora_avanzado.py::test_multiplicar_preciso PASSED [ 94%]
tests/test_calculadora_avanzado.py::test_dividir_preciso PASSED [100%]
```

Para ejecutar solo los tests marcados como "smoke":

```
pytest -v -m smoke
```

Para ejecutar solo los tests marcados como "exception":

```
pytest -v -m exception
```

Para ver un reporte detallado de tests parametrizados:

```
pytest -v tests/test_calculadora_avanzado.py::test_sumar_varios
```

Con estos archivos tenemos ya los conceptos clave:

1. Fixtures para reutilización de datos
2. Parametrización para múltiples casos de prueba
3. Etiquetas para agrupar tests
4. Aserciones avanzadas para comparaciones precisas

## Reporte HTML nativo

Para compartir resultados con tu equipo genera un informe HTML autosuficiente:

Ejecutá la suite con:

```
pytest --html=report.html --self-contained-html
```

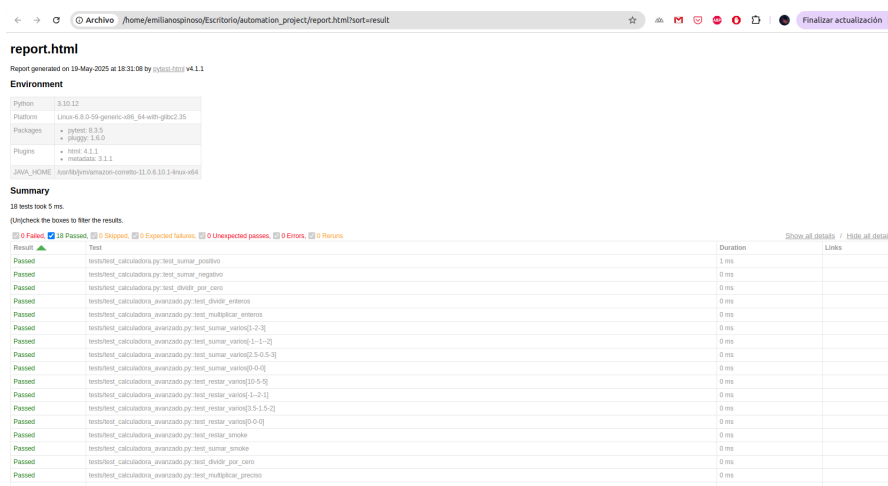
- `--html=report.html` establece el nombre del archivo.
- `--self-contained-html` incrusta CSS y JS, de modo que el archivo funcione sin recursos externos.

Al finalizar verás en consola:

```
----- Generated html report:
file:///home/emilianospinoso/Escritorio/automation_project/report.html
-----
```

**Abre `report.html`** con doble clic o arrastrándolo a tu navegador para revisar los resultados y compártelo con tu equipo adjuntándolo en el Pull Request.

Vas a ver algo como esto:



Result	Test	Duration	Links
Passed	testtest_calculadora.py::test_sumar_positivo	1 ms	
Passed	testtest_calculadora.py::test_sumar_negativo	0 ms	
Passed	testtest_calculadora.py::test_dividir_por_cero	0 ms	
Passed	testtest_calculadora_avanzado.py::test_dividir_enteros	0 ms	
Passed	testtest_calculadora_avanzado.py::test_multiplicar_enteros	0 ms	
Passed	testtest_calculadora_avanzado.py::test_sumar_varios[1-1-2]	0 ms	
Passed	testtest_calculadora_avanzado.py::test_sumar_varios[5-4-5-3]	0 ms	
Passed	testtest_calculadora_avanzado.py::test_sumar_varios[0-0]	0 ms	
Passed	testtest_calculadora_avanzado.py::test_restar_varios[10-5-5]	0 ms	
Passed	testtest_calculadora_avanzado.py::test_restar_varios[4-2-2]	0 ms	
Passed	testtest_calculadora_avanzado.py::test_restar_varios[5-1-5-2]	0 ms	
Passed	testtest_calculadora_avanzado.py::test_restar_varios[0-0]	0 ms	
Passed	testtest_calculadora_avanzado.py::test_restar_varios	0 ms	
Passed	testtest_calculadora_avanzado.py::test_sumar_varios	0 ms	
Passed	testtest_calculadora_avanzado.py::test_dividir_por_cero	0 ms	
Passed	testtest_calculadora_avanzado.py::test_multiplicar_preciso	0 ms	
Passed	testtest_calculadora_avanzado.py::test_dividir_preciso	0 ms	

## Preguntas para reflexionar

- ¿Cómo te ayuda un fixture a centralizar datos de prueba?
- ¿Qué beneficios aporta dividir tests en grupos (`smoke`, `slow`)?
- ¿Cuándo parametrizar tests ahorra mantenimiento?

## Próximos pasos

En la **Clase N° 5** profundizaremos en **HTML y la estructura de páginas web**. Estudiaremos la anatomía de un documento HTML, sus elementos principales (`div`, `form`, `input`, `button`) y aprenderemos a inspeccionar elementos con las DevTools del navegador para preparar la automatización web con Selenium.

## Automatizando en TalentoLab



Talento Lab debe lanzar un microservicio de cálculo.

Silvia (PO) necesita evidencia automatizada de que las operaciones básicas funcionan.



Matías (Automation Lead) te pide:

"Cubrí todas las funciones de la calculadora, parametrizá la suma con varios valores y marca las operaciones clave como `smoke`. El reporte HTML debe adjuntarse al Pull Request."

# Ejercicios prácticos

## Pytest

### 1. Casos de prueba por operación

Crea **dos tests para cada función** (`sumar`, `restar`, `multiplicar`, `dividir`):

- **Éxito** : resultado correcto.
- **Error** : comportamiento esperado ante fallo (p. ej. `dividir` debe lanzar `ValueError` al dividir por 0)

### 2. Fixtures

- Conserva el fixture de enteros existente.
- Añade un segundo fixture con **valores flotantes** (`0.1`, `0.2`) y utilízalo donde corresponda.

### 3. Parametrización

Para `sumar` y `restar`, usa `@pytest.mark.parametrize` con **al menos tres datasets** distintos.

### 4. Markers

- Etiqueta los tests de `sumar` con `@pytest.mark.smoke`.
- Etiqueta los tests de `dividir` que validan la excepción con `@pytest.mark.exception`.

***Muestra cómo filtrar la ejecución:***

```
pytest -m smoke           # solo tests "sumar"
```

```
pytest -m exception       # solo tests "dividir" con error
```

### 5. Reporte HTML

- Genera el informe con: `pytest --html=report.html`.
- Sube `report.html` junto con tu Pull Request.



**Buenos Aires**  
*aprende*  
Agencia de Habilidades para el Futuro

**BA** Buenos  
Aires  
Ciudad