**Proves Unitàries en Python**

*Les proves unitàries són una tècnica de verificació del codi en què es prova cada part d'un programa de manera individual (normalment funcions o mètodes) per assegurar-se que funciona correctament de manera aïllada. Cada "unitat" del codi es prova per si mateixa per garantir que es comporta de la manera esperada sota diferents condicions.*

*Les proves unitàries tenen diversos beneficis en el desenvolupament de programari:*

*1. Detectar errors: Ajuden a trobar errors en el codi més ràpidament, abans que es desplegui o s'integri amb altres parts del sistema.*

*2. Millorar la qualitat del codi: Quan implementes proves unitàries, tens una millor visió del comportament del teu codi i t'assegures que cada part funciona correctament.*

*3. Facilitar els canvis: Les proves unitàries permeten comprovar ràpidament si els canvis en el codi trenquen alguna funcionalitat existent.*

*4. Documentació viva: Les proves mostren exemples clars de com s'utilitzen les funcions o mètodes i serveixen com a documentació.*

*Objectiu de les Proves Unitàries*

*L'objectiu és provar funcions o mètodes individuals en diversos escenaris, incloent-hi:*

*- Condicions normals: On les dades d'entrada són les esperades.*

*- Condicions extremes: Per veure què passa amb valors inusuals (com números negatius o zero).*

*- Condicions d'error: Per verificar que la funció es comporta correctament amb dades incorrectes o inadequades.*

*Implementació de Proves Unitàries*

*Podem utilitzar dues eines principals: `unittest` (integrada a ) o `pytest` (una eina més avançada i flexible). Veurem com implementar proves amb totes dues eines.*

**1. Proves Unitàries amb `unittest`**

- Crearem un fitxer anomenat `matematica.py` amb algunes funcions matemàtiques bàsiques:

**# matematica.py**

**import math**

**def potencia(base, exponent):**

**return base \*\* exponent**

**def arrel\_quadrada(valor):**

**if valor < 0:**

**return "Valor negatiu no vàlid"**

**return math.sqrt(valor)**

**def factorial(n):**

**if n < 0:**

**return "Valor negatiu no vàlid"**

**elif n == 0:**

**return 1**

**else:**

**return math.prod(range(1, n + 1))**

- Creació de Proves amb `unittest`

Creem un fitxer de proves anomenat `test\_matematica.py` amb el següent contingut:

**import unittest**

**from matematica import potencia, arrel\_quadrada, factorial**

**class TestMatematica(unittest.TestCase):**

**def test\_potencia(self):**

**self.assertEqual(potencia(2, 3), 8, "La potència de 2^3 hauria de ser 8")**

**self.assertEqual(potencia(5, 0), 1, "La potència de 5^0 hauria de ser 1")**

**def test\_arrel\_quadrada(self):**

**self.assertEqual(arrel\_quadrada(4), 2, "L'arrel quadrada de 4 hauria de ser 2")**

**self.assertEqual(arrel\_quadrada(-4), "Valor negatiu no vàlid", "No s'accepten valors negatius")**

**def test\_factorial(self):**

**self.assertEqual(factorial(5), 120, "El factorial de 5 hauria de ser 120")**

**self.assertEqual(factorial(0), 1, "El factorial de 0 hauria de ser 1")**

**self.assertEqual(factorial(-3), "Valor negatiu no vàlid", "No s'accepten valors negatius")**

**if \_\_name\_\_ == '\_\_main\_\_':**

**unittest.main()**

Explicació:

*- assertEqual: Comprova si el resultat de la funció és igual al valor esperat. Si la condició no es compleix, la prova falla.*

*- if \_\_name\_\_ == '\_\_main\_\_': Permet executar les proves si el fitxer es crida directament.*

Com Executar les Proves:

Podem executar el fixer directament si estem dins del VS Code o des del terminal, executa:

**python -m unittest test\_matematica.py**

Aquest procés executarà totes les proves i mostrarà si han passat o fallat.

**2. Proves Unitàries amb `pytest`**

`pytest` és un framework de proves unitàries més flexible i fàcil d’utilitzar que `unittest`. Primer, instal·la `pytest`:

**pip install pytest**

- Exemple de Proves amb `pytest`

Utilitzarem el mateix exemple del fitxer `matematica.py`.

Creem el fitxer de proves `test\_matPytest.py` amb el següent contingut:

from matematica import potencia, arrel\_quadrada, factorial

def test\_potencia():

assert potencia(2, 3) == 8

assert potencia(5, 0) == 1

def test\_arrel\_quadrada():

assert arrel\_quadrada(4) == 2

assert arrel\_quadrada(-4) == "Valor negatiu no vàlid"

def test\_factorial():

assert factorial(5) == 120

assert factorial(0) == 1

assert factorial(-3) == "Valor negatiu no vàlid"

Explicació:

*- assert: Comprova si la condició és certa. Si no es compleix, `pytest` reportarà un error.*

*- Simplicitat: No cal definir classes ni fer servir mètodes d'assert específics.*

Com Executar les Proves amb `pytest`:

Per executar les proves amb `pytest`, ens hem de situar al directori on hi ha els arxius i executar:

**pytest**

`pytest` detectarà automàticament tots els fitxers que comencin per `test\_` i executarà les proves.

- Exemple de Comportament d'Error

Modifica el codi a ` test\_matPytest.py ` introduint un error i executa `pytest` per veure com detecta l’error.

**Comparativa entre `unittest` i `pytest` per a Proves Unitàries en Python**

1. Simplicitat:

- `unittest`: Utilitza mètodes com `assertEqual` i requereix estructurar les proves dins de classes.

- `pytest`: Permet utilitzar directament `assert` sense definir classes.

2. Configuració:

- `unittest`: Integrat en , sense instal·lació addicional.

- `pytest`: Requereix instal·lar-lo (`pip install pytest`), però és més flexible.

3. Capacitats Addicionals:

- `unittest`: Inclou funcionalitats per a proves estructurades.

- `pytest`: Pot executar proves complexes amb menys codi i suporta fixtures i plugins.

4. Execució de Proves:

- `unittest`: Les proves es defineixen dins de classes i s'executen amb ` -m unittest`.

- `pytest`: Simplement defineixes funcions de prova i executes `pytest`.

Resum

`unittest` és ideal per a projectes grans amb estructures formals de prova, mentre que `pytest` és més adequat per a desenvolupadors que volen fer proves ràpidament amb menys codi.