### **¿Qué son las Pruebas Unitarias?**

Las pruebas unitarias son una forma de asegurar la calidad del código al probar individualmente las unidades más pequeñas de código (funciones, métodos, clases). Estas pruebas verifican que cada unidad de código funcione correctamente de acuerdo a su especificación.

**Beneficios de las pruebas unitarias:**

* **Detección temprana de errores:** Se identifican errores en etapas tempranas del desarrollo.
* **Mayor confianza en el código:** Se reduce la probabilidad de introducir errores al realizar cambios.
* **Facilita el mantenimiento:** Las pruebas sirven como documentación del comportamiento esperado del código.
* **Promueve el diseño limpio:** Al escribir pruebas, se fomenta un diseño modular y bien estructurado.

### **Cómo Aplicar Pruebas Unitarias en Python**

Python ofrece un módulo estándar llamado unittest para escribir pruebas unitarias.

**Ejemplo básico:**

Python

import unittest

def suma(a, b):

return a + b

class TestSuma(unittest.TestCase):

def test\_suma\_positivos(self):

self.assertEqual(suma(2, 3), 5)

def test\_suma\_negativos(self):

self.assertEqual(suma(-2,

[1. github.com](https://github.com/Sapitorico/Development-with-Python)

[github.com](https://github.com/Sapitorico/Development-with-Python)

-3), -5) if \_\_name\_\_ == '\_\_main\_\_': unittest.main()

Use code [with caution.](file:///C:/faq#coding)

**Explicación:**

1. **Importar unittest:** Se importa el módulo unittest.
2. **Crear una clase de prueba:** Se crea una clase que hereda de unittest.TestCase.
3. **Definir métodos de prueba:** Cada método de prueba verifica una funcionalidad específica.
4. **Utilizar métodos de afirmación:** Métodos como assertEqual, assertTrue, assertFalse, etc., se utilizan para verificar si el resultado obtenido es el esperado.

### **Mejores Prácticas**

* **Escribir pruebas antes de escribir el código:** Esto te obliga a pensar en cómo se va a usar el código y a diseñar interfaces más claras.
* **Pruebas independientes:** Cada prueba debe ser independiente de las demás.
* **Pruebas exhaustivas:** Cubre todos los casos posibles, incluyendo valores límite y casos de error.
* **Pruebas legibles:** Utiliza nombres descriptivos para los métodos de prueba y los asserts.
* **Automatizar las pruebas:** Integra las pruebas en tu flujo de trabajo de desarrollo para ejecutarlas de forma automática.

### **Ejemplo Detallado: Aplicación de Tienda Online**

Python

# src/producto/models.pyclass Producto: def \_\_init\_\_(self, nombre, precio, stock): self.nombre = nombre self.precio = precio self.stock = stock # tests/test\_producto.pyimport unittest from producto.models import Producto class TestProducto(unittest.TestCase): def test\_crear\_producto(self): producto = Producto("Camiseta", 19.99, 10) self.assertEqual(producto.nombre, "Camiseta") self.assertEqual(producto.precio, 19.99) self.assertEqual(producto.stock, 10) def test\_stock\_negativo(self): with self.assertRaises(ValueError): Producto("Pantalón", 29.99, -5)

Use code [with caution.](file:///C:/faq#coding)

**¿Por qué este ejemplo es bueno?**

* **Pruebas para diferentes escenarios:** Se prueba la creación de un producto válido y se verifica que se lance una excepción cuando el stock es negativo.
* **Nomenclatura clara:** Los nombres de los métodos de prueba describen claramente lo que se está probando.
* **Uso de assertRaises:** Se utiliza para verificar que se lance la excepción esperada.

### **Herramientas Adicionales**

* **pytest:** Un framework de pruebas más flexible y completo que unittest.
* **nose2:** Otro framework de pruebas que ofrece características adicionales.
* **Coverage:** Una herramienta para medir la cobertura de código de tus pruebas.

**Conclusión**

Las pruebas unitarias son una inversión de tiempo que vale la pena. Al escribir pruebas unitarias de forma regular, mejorarás la calidad de tu código, reducirás errores y aumentarás tu confianza en tu aplicación.