Testes unitários

📌 Passos para criar um teste unitário:

1. Criar uma função/classe que queremos testar.
2. Criar um arquivo de teste separado.
3. Usar `unittest.TestCase` para definir os testes.
4. Rodar os testes com `unittest.main()`.

🎯 Exemplo Simples

🔷 Código que será testado (`calculadora.py`):

```
def soma(a, b):
    return a + b

def subtrai(a, b):
    return a - b
```

🔷 Teste unitário (`test_calculadora.py`):

```
import unittest
from calculadora import soma, subtrai  # Importamos as funções que queremos testar

class TestCalculadora(unittest.TestCase):  # Criamos uma classe que herda de unittest.TestCase

    def test_soma(self):
        self.assertEqual(soma(2, 3), 5)  # Testa se 2 + 3 é igual a 5
        self.assertEqual(soma(-1, 1), 0)  # Testa se -1 + 1 é igual a 0
```

```
    def test_subtrai(self):
        self.assertEqual(subtrai(5, 3), 2)  # Testa se 5 - 3 é igual a 2
        self.assertEqual(subtrai(10, 4), 6)  # Testa se 10 - 4 é igual a 6

if __name__ == "__main__":
    unittest.main()  # Executa os testes
```

🛠️ Explicação linha por linha:

1. `import unittest` → Importa o módulo de testes.
2. `from calculadora import soma, subtrai` → Importa as funções que serão testadas.
3. `class TestCalculadora(unittest.TestCase):` → Cria a classe de testes, que herda de `unittest.TestCase`.
4. `def test_soma(self):` → Define um método de teste para a função `soma`.
5. `self.assertEqual(soma(2, 3), 5)` → Compara se `soma(2, 3)` retorna `5`, caso contrário, o teste falha.
6. `self.assertEqual(soma(-1, 1), 0)` → Outro teste para `soma()`.
7. `def test_subtrai(self):` → Define um método de teste para `subtrai()`.
8. `unittest.main()` → Executa todos os testes dentro da classe.

No terminal, digite:

```
python test_calculadora.py
```

`TestCase` é uma classe do módulo `unittest` em Python que serve como base para a criação de testes unitários.

Ela fornece métodos prontos para verificar se o código retorna os resultados esperados, como:

✅ `assertEqual(a, b)`: Verifica se `a == b`

✅ `assertNotEqual(a, b)`: Verifica se `a != b`

✅ `assertTrue(x)` : Verifica se `x` é `True`

✅ `assertFalse(x)` : Verifica se `x` é `False`

✅ `assertRaises(Exception, func, *args)` : Verifica se `func(*args)` levanta a exceção `Exception`

🔷 Exemplo de `TestCase`

```
python
CopiarEditar
import unittest

# Função que será testada
def divide(a, b):
    if b == 0:
        raise ValueError("Não pode dividir por zero!")
    return a / b

# Criando uma classe de teste que herda de unittest.TestCase
class TesteDivisao(unittest.TestCase):

    def test_divisao_normal(self):
        self.assertEqual(divide(10, 2), 5)  # 10 / 2 = 5

    def test_divisao_zero(self):
        with self.assertRaises(ValueError):  # Testa se a exceção é levantada
            divide(10, 0)

# Executando os testes
if __name__ == "__main__":
    unittest.main()
```

🔍 O que acontece aqui?

1. `TestCase` **é herdado** → Criamos `TesteDivisao(unittest.TestCase)`, permitindo usar os métodos de teste.
2. **Verificação de valores** → `assertEqual(divide(10, 2), 5)` testa se o retorno é 5.

3. **Testando exceções** → `assertRaises(ValueError)` verifica se a função lança um erro ao dividir por zero.
4. **Execução automática** → `unittest.main()` roda os testes.

💡 **Ou seja, `TestCase` estrutura os testes e facilita a validação do código!** 🚀