Ambientes Virtuais, Pytest e Cobertura de Código

Módulo 1: Organizando o Desenvolvimento no Windows

1.1 O Conceito de Ambiente Virtual (.venv)

Um ambiente virtual é uma cópia isolada do interpretador Python e de seu gerenciador de pacotes (pip). Ele é essencial para o desenvolvimento profissional, pois garante que as dependências de um projeto (bibliotecas e suas versões) não entrem em conflito com as de outro projeto.

1.2 Criação e Ativação do Ambiente

Usaremos o módulo venv nativo do Python, o que dispensa a instalação de softwares adicionais.

Passo 1: Criar o Ambiente

Na pasta raiz do seu projeto, use o comando abaixo. A pasta .venv será criada.

PowerShell

```
python -m venv .venv
```

Passo 2: Ativar o Ambiente

Para começar a usar o Python isolado, o ambiente deve ser ativado. O comando varia conforme o terminal:

Terminal Utilizado	Comando de Ativação
PowerShell	.venv\Scripts\Activate.ps1
Prompt de Comando (CMD)	.venv\Scripts\activate

Confirmação: O prefixo (.venv) aparecerá na linha de comando, indicando que o ambiente está ativo.

Módulo 2: O Framework Pytest

2.1 Instalação do Pytest

Com o ambiente (.venv) **ATIVADO**, instale o framework de testes e o plugin de cobertura de código.

PowerShell

```
pip install pytest pytest-cov
```

2.2 Convenções e Estrutura

O Pytest encontra automaticamente os testes seguindo estas regras:

1. **Arquivos de Teste:** Devem começar com test_ (ex: test_funcoes.py).
2. **Funções de Teste:** Devem ser funções e começar com def test_ (ex: def test_somar_numeros():).

2.3 Código de Produção (funcoes.py)

Este é o arquivo que contém todo o código que será testado (somente exemplos).

Python

```python
# funcoes.py

def somar(a, b):
    return a + b

def subtrair(a, b):
    return a - b

def processar_lista(lista):
    """Ordena uma lista, levantando erro se vazia."""
    if not lista:
        raise ValueError("Lista não pode ser vazia.")
    # Supondo que a implementação correta seja ordenar
    return sorted(lista)

def buscar_usuario(usuarios: list, id_usuario: int):
    """Busca um usuário em uma lista de dicionários pelo ID."""
    for usuario in usuarios:
        if usuario.get('id') == id_usuario:
            return usuario
    return None
```

Módulo 3: Exemplos Completos de Testes

Todos os exemplos devem ser colocados no arquivo **test_funcoes.py**.

3.1 Testes de Funções Matemáticas Simples

Python

```python
# test_funcoes.py (Parte 1)

from funcoes import somar, subtrair

def test_somar_positivos():
    # Assert: 10 + 5 DEVE ser 15
    assert somar(10, 5) == 15

def test_subtrair_resultado_negativo():
    # Assert: 5 - 10 DEVE ser -5
    assert subtrair(5, 10) == -5
```

3.2 Testes de Estruturas de Dados (Listas e Dicionários)

Python

```python
# test_funcoes.py (Parte 2)

from funcoes import processar_lista, buscar_usuario

def test_processar_lista_ordenacao():
    lista_desordenada = [3, 1, 2, 4]
    lista_esperada = [1, 2, 3, 4]

    # O Pytest compara elemento por elemento para garantir a ordem
    assert processar_lista(lista_desordenada) == lista_esperada

def test_buscar_usuario_encontrado_completo():
    USUARIOS_TESTE = [
        {'id': 100, 'nome': 'Charlie', 'cargo': 'Dev'},
        {'id': 101, 'nome': 'Diana', 'cargo': 'Manager'}
    ]
    ID_EXISTENTE = 101
    usuario_esperado = {'id': 101, 'nome': 'Diana', 'cargo': 'Manager'}

    # Assert que o dicionário retornado é idêntico ao esperado
    assert buscar_usuario(USUARIOS_TESTE, ID_EXISTENTE) == usuario_esperado

def test_buscar_usuario_nao_encontrado():
    # Cenário de teste: Uma lista com dados, mas o ID não existe.
    USUARIOS_TESTE = [{'id': 50, 'nome': 'Ana', 'email': 'ana@exemplo.com'}]
    ID_INEXISTENTE = 999

    # Assert: O retorno DEVE ser o valor None (ausência de valor)
    resultado = buscar_usuario(USUARIOS_TESTE, ID_INEXISTENTE)
    assert resultado is None
```

3.3 Teste de Exceções (Falhas Esperadas)

Usamos o recurso pytest.raises para afirmar que uma função deve falhar sob determinadas condições.

Python

```
# test_funcoes.py (Parte 3)

import pytest
from funcoes import processar_lista

def test_processar_lista_levanta_value_error():

    # Afirma que a exceção ValueError DEVE ser levantada
    with pytest.raises(ValueError) as e:
        processar_lista([])

    # Verifica se a mensagem de erro da exceção está correta
    assert "Lista não pode ser vazia." in str(e.value)
```

Módulo 4: Qualidade do Código com pytest-cov

4.1 Medindo a Cobertura de Código

O pytest-cov é um plugin que mede a porcentagem do seu código de produção que foi executada (testada) durante a rodada de testes.

4.2 Comando de Execução

Em vez de usar apenas pytest, você adiciona o parâmetro --cov seguido do nome do seu módulo de código (funcoes):

PowerShell

```
pytest --cov=funcoes -v
```

4.3 Análise do Relatório

O relatório de cobertura será exibido ao final da execução, indicando linhas testadas e linhas faltando:

Módulo	Stmts (Declarações)	Miss (Linhas Faltando)	Cover (Cobertura)
funcoes.py	12	0	**100%**
TOTAL	12	0	**100%**

- Se houver um número em **Miss**, significa que você precisa escrever um teste que execute aquelas linhas de código (geralmente blocos if ou else não cobertos).

Módulo 5: Finalização e Recursos Avançados

5.1 Desativando o Ambiente

Sempre que terminar o trabalho, use o comando:

```
PowerShell
```

```
deactivate
```

5.2 Recursos Avançados para Estudo

Recurso	Descrição
@pytest.mark.parametrize	Permite rodar o mesmo teste várias vezes com diferentes entradas (tuplas de dados), economizando código e cobrindo diversos casos de teste rapidamente.
Fixtures	Usadas para configurar recursos complexos (ex: lista de usuários muito longa, conexão com banco de dados) que são injetados em múltiplos testes, garantindo que o ambiente de teste esteja sempre limpo.
Relatório HTML	Gere um relatório interativo da cobertura de código para visualização no navegador com: pytest --cov=funcoes --cov-report=html.