Apostila Pytest Definitiva:
Controle Total sobre o Código com Recursos Especiais

🎯 O Desafio de Testar o "Mundo Real"

Para que seus testes sejam confiáveis, é crucial controlar fatores externos:

1. **O Estado (SQLite3)**: Conexões com bancos de dados, onde as ações de um teste podem afetar o próximo (exigindo **Fixtures** e **Injeção de Dependência**).
2. **A Sorte (random)**: Geração de números aleatórios, que leva a resultados imprevisíveis (exigindo **Mocking**).

Módulo 1: Configuração Essencial e Estrutura

1.1. Organização do Projeto

Mantenha a separação entre o código a ser testado e os testes:

```
projeto_teste/
├── .venv/          (Ambiente Virtual)
├── codigo_producao.py  (O código do seu sistema)
└── test_recursos.py    (O código dos seus testes)
```

1.2. Instalação das Ferramentas

Instale o Pytest e os plugins necessários para as técnicas avançadas:

1. **Framework de Teste e Cobertura:**
   PowerShell
   pip install pytest pytest-cov

2. **Plugin de Simulação (Mocking):** Usado para controlar funções externas, como o módulo random.
   PowerShell
   pip install pytest-mock

Módulo 2: O Fator Sorte - Dominando o random com Mocking

O **Mocking** permite usar um **"Dublê"** no lugar de um componente real para isolar o código e garantir resultados previsíveis.

2.1. O Código a Ser Controlado (Tudo em codigo_producao.py)

```python
# ====================================================
# ARQUIVO: codigo_producao.py (Parte 1: Lógica Random)
# ====================================================
import random
import sqlite3

def jogar_dado_seis_lados() -> int:
    """Gera um número aleatório entre 1 e 6."""
    return random.randint(1, 6)

def verificar_numero_secreto(palpite: int) -> bool:
    """Tenta adivinhar um número entre 1 e 10, tratando erros internos."""
    try:
        numero_secreto = jogar_dado_seis_lados()
        if numero_secreto == 10:
            raise ValueError("Erro interno: dado fora do intervalo.")
        return palpite == numero_secreto
    except ValueError:
        return False

def numero_eh_valido(numero: int) -> bool:
    """Verifica se o número está no intervalo válido de um dado (1 a 6)."""
    return 1 <= numero <= 6

def gerar_id_composto() -> str:
    """Gera um ID composto de 4 dígitos aleatórios (1-9)."""
    d1 = str(random.randint(1, 9))
    d2 = str(random.randint(1, 9))
    d3 = str(random.randint(1, 9))
    d4 = str(random.randint(1, 9))
    return f"{d1}{d2}{d3}{d4}"
```

2.2. Teste A: Mocking de Valores Sequenciais (Em test_recursos.py)

```
# ======================================================
# ARQUIVO: test_recursos.py (Parte 1: Testes de Mocking)
# ======================================================
import pytest
from codigo_producao import (
    jogar_dado_seis_lados, verificar_numero_secreto,
    numero_eh_valido, gerar_id_composto
)
# ... (demais imports e código omitido por brevidade)

def test_dado_simulado_sequencial(mocker):
    # 1. Preparar o Dublê: Simula 3 lançamentos (retornos 3, 6, 1)
    mock_sequencia = mocker.Mock(side_effect=[3, 6, 1])

    # 2. Injetar (Patch): Substitui random.randint DENTRO de "codigo_producao"
    mocker.patch("codigo_producao.random.randint", new=mock_sequencia)

    # 3. Execução: As chamadas usam o mock, garantindo o resultado.
    assert jogar_dado_seis_lados() == 3
    assert jogar_dado_seis_lados() == 6
    assert jogar_dado_seis_lados() == 1

    assert mock_sequencia.call_count == 3
```

2.3. Teste B: Mocking + Parametrização (A Técnica Avançada!)

Aqui, usamos parametrize para fornecer a **lista de retorno do Mock** e o **resultado esperado** em uma única tabela, rodando o teste para múltiplos cenários.

```
# Teste Avançado: Parametrização para controlar Mocking
@pytest.mark.parametrize(
    "mock_sequence, expected_id",
    [
        # Cenário 1: Sequência simples (Mock retorna [1, 1, 1, 1])
        ([1, 1, 1, 1], "1111"),
        # Cenário 2: Sequência complexa (Mock retorna [9, 8, 7, 6])
        ([9, 8, 7, 6], "9876"),
        # Cenário 3: Sequência alternada (Mock retorna [2, 5, 2, 5])
        ([2, 5, 2, 5], "2525"),
    ]
)
def test_gerar_id_composto_com_parametrizacao(mocker, mock_sequence, expected_id):
    """
    Roda o mesmo teste 3 vezes, injetando uma sequência diferente no Mock a cada rodada.
    """
    # 1. Cria o Mock AGORA, usando a sequência fornecida pelo parametrize
    mock_randint = mocker.Mock(side_effect=mock_sequence)

    # 2. Patch: Substitui a função real pelo mock
    mocker.patch("codigo_producao.random.randint", new=mock_randint)

    # 3. Execução e Verificação
    assert gerar_id_composto() == expected_id
    # Garante que a função interna foi chamada 4 vezes (uma para cada dígito)
    assert mock_randint.call_count == 4
```

Módulo 3: O Fator Estado - Isolamento com Fixtures e Injeção de Dependência

3.1. O Código de Produção (Pronto para Injeção)

```
# ====================================================
# ARQUIVO: codigo_producao.py (Parte 2: Lógica SQLite)
# ====================================================
# ... (imports no topo do arquivo)

# 1. Dependência (Sabe como conectar)
class ConexaoDB:
    """Gerencia a conexão bruta com o SQLite."""
    def __init__(self, db_path: str):
        self.conn = sqlite3.connect(db_path)
        self._criar_tabela()

    def _criar_tabela(self):
        self.conn.execute("CREATE TABLE IF NOT EXISTS tarefas (id INTEGER PRIMARY KEY, descricao
TEXT, concluida INTEGER)")
        self.conn.commit()

    def obter_conexao(self):
        return self.conn

    def fechar(self):
        self.conn.close()
```

```python
# 2. Classe Principal (Recebe a dependência)
class GerenciadorTarefas:
    """Gerencia a lógica de negócio, usando a ConexaoDB."""
    def __init__(self, db_conn: ConexaoDB): # <--- INJEÇÃO DE DEPENDÊNCIA
        self.db_conn = db_conn
        self.conn = db_conn.obter_conexao()

    def adicionar_tarefa(self, descricao: str):
        self.conn.execute("INSERT INTO tarefas (descricao, concluida) VALUES (?, 0)", (descricao,))
        self.conn.commit()

    def contar_tarefas(self) -> int:
        cursor = self.conn.cursor()
        resultado = cursor.execute("SELECT COUNT(*) FROM tarefas").fetchone()
        return resultado[0]
```

3.2. Teste Aninhado: Fixtures para Setup e Composição (Em test_recursos.py)

```
# ======================================================
# ARQUIVO: test_recursos.py (Parte 2: Testes de SQLite)
# ======================================================
from codigo_producao import ConexaoDB, GerenciadorTarefas

# FIXTURE 1: Cria e Limpa a Dependência
@pytest.fixture
def conexao_db_limpa():
    """Cria um DB temporário (:memory:) e garante o fechamento (Teardown)."""

    # SETUP
    conexao = ConexaoDB(db_path=":memory:")

    yield conexao # Entrega o objeto de conexão

    # TEARDOWN: Código executado após o teste
    conexao.fechar()

# FIXTURE 2: Monta a Classe a Ser Testada
@pytest.fixture
def gerenciador_tarefas_isolado(conexao_db_limpa):
    """Monta o GerenciadorTarefas, INJETANDO a ConexaoDB limpa."""
    # Ocorre a Composição
    return GerenciadorTarefas(db_conn=conexao_db_limpa)


# Testes Utilizando o Recurso Limpo
def test_contagem_inicia_em_zero(gerenciador_tarefas_isolado):
    """Verifica o estado inicial (DB limpo)."""
    assert gerenciador_tarefas_isolado.contar_tarefas() == 0

def test_adicionar_tarefa_incrementa_contador(gerenciador_tarefas_isolado):
    """Verifica a adição de dados no DB isolado."""
    gerenciador_tarefas_isolado.adicionar_tarefa("Revisar Fixtures")
    assert gerenciador_tarefas_isolado.contar_tarefas() == 1
```

Módulo 4: Parametrização para Múltiplos Cenários

Usamos parametrize para injetar múltiplos dados em um teste de lógica simples.

```
# ====================================================
# ARQUIVO: test_recursos.py (Parte 3: Parametrização)
# ====================================================

# Parametrização de sucesso
@pytest.mark.parametrize("valor_entrada", [
    (1),   # Mínimo
    (3),   # Meio
    (6),   # Máximo
])
def test_numero_valido_sucesso(valor_entrada):
    """Roda o mesmo teste injetando valores válidos (1, 3 e 6)."""
    assert numero_eh_valido(valor_entrada) is True

# Parametrização de falha
@pytest.mark.parametrize("valor_entrada", [
    (0),   # Fora do limite inferior
    (7),   # Fora do limite superior
    (-10), # Negativo
])
def test_numero_valido_falha(valor_entrada):
    """Roda o mesmo teste injetando valores inválidos (0, 7 e -10)."""
    assert numero_eh_valido(valor_entrada) is False
```

Módulo 5: Comando Final e Análise

Para rodar todos os testes de Mocking, Fixtures e Parametrização, e verificar a cobertura de código:

```
pytest --cov=codigo_producao -v
```

Esta apostila fornece a base completa para testes avançados no Pytest, garantindo o controle total sobre a aleatoriedade e o estado do sistema.