Simulação e Teste de Software (CC8550)

Aula 02 – Teste de unidade com pytest

Prof. Luciano Rossi

Ciência da Computação
Centro Universitário FEI

1º Semestre de 2026

Teste de Unidade com pytest

Aprofundamento técnico

No tópico anterior vimos:

- Instalar pytest
- Estrutura básica de testes
- Assertions simples
- Executar testes

Neste tópico vamos dominar:

- Princípios de bons testes unitários (FIRST)
- Fixtures para reutilização de código
- Parametrização de testes
- Testes de exceções
- Organização profissional de suítes

Objetivo: Escrever testes unitários de qualidade profissional!

centro universitário FEI

O que é um Bom Teste Unitário?

Um teste unitário de qualidade deve ser:

Rápido: Executa em milissegundos
Suítes lentas desencorajam execução frequente

Isolado: Não depende de outros testes
Pode executar em qualquer ordem

Repetível: Mesmo resultado sempre
Sem dependências externas (rede, BD, data/hora)

Auto-verificável: Passa ou falha automaticamente
Sem verificação manual de logs

Oportuno: Escrito no momento certo
Idealmente antes ou junto com o código

centro universitário FEI

Princípios FIRST

Mnemônico para testes de qualidade

Fast
Independent
Repeatable
Self-validating
Timely

Importante

Esses princípios guiam a escrita de testes que são **confiáveis**, **manuteníveis** e **úteis** ao longo do tempo.

Padrão AAA (Arrange-Act-Assert)

Estrutura clara de testes

Todo teste deve seguir três fases:

Arrange (Preparar): Setup dos dados e contexto

```
# Arrange
calculadora = Calculadora()
 x, y = 10, 5
```

Act (Agir): Executar a ação sendo testada

```
# Act
resultado = calculadora.dividir(x, y)
```

Assert (Verificar): Verificar o resultado esperado

```
# Assert
assert resultado == 2
```

centro universitário FEI

Exemplo: AAA na Prática

```
def test_adicionar_produto_carrinho():
# Arrange
carrinho = CarrinhoCompras()
produto = Produto("Notebook", 3000.00)

# Act
carrinho.adicionar(produto)

# Assert
assert len(carrinho.itens) == 1
assert carrinho.total == 3000.00
```

Dica

Use comentários ou linhas em branco para separar as três fases!

centro universitário FEI

Assertions Básicas

Igualdade e desigualdade:

```
assert x == 5
assert y != 0
```

Comparações:

```
assert idade >= 18
assert temperatura < 100
```

Pertencimento:

```
assert "Python" in linguagens
assert usuario not in bloqueados
```

Booleanos:

```
assert is_valid
assert not is_expired
```

centro universitário FEI

Testando Exceções

Verificar comportamento excepcional

Use pytest.raises para verificar que exceções são lançadas:

```
import pytest

def test_divisao_por_zero():
 calculadora = Calculadora()

 with pytest.raises(ZeroDivisionError):
  calculadora.dividir(10, 0)
```

Importante

Testar exceções é essencial para garantir que seu código falha **corretamente** quando deveria!

centro universitário FEI

Testando Mensagem de Exceção

Você pode verificar também a mensagem de erro:

```
def test_cpf_invalido():
 with pytest.raises(ValueError) as exc_info:
  validar_cpf("123.456.789-00")

  assert "CPF inválido" in str(exc_info.value)
```

Vantagens:

- Garante que a exceção correta foi lançada
- Valida que a mensagem é informativa
- Detecta se lógica de validação mudou

centro universitário FEI

Comparação de Floats

Cuidado com precisão de ponto flutuante

Problema:

```
def test_media():
 assert calcular_media([1, 2]) == 1.5
 # Pode falhar por erro de precisão!
```

Solução: usar pytest.approx

```
import pytest

def test_media():
 resultado = calcular_media([1, 2])
 assert resultado == pytest.approx(1.5)
```

Permite pequenas diferenças devido a arredondamento

O que são Fixtures?

Fixtures são funções que fornecem dados ou estado para testes.

Problemas que resolvem:

- Código duplicado de setup em múltiplos testes
- Preparação complexa de dados
- Limpeza de recursos após testes
- Compartilhamento de objetos entre testes

Vantagens:

- Reutilização de código
- Testes mais limpos e legíveis
- Setup e teardown automático
- Injeção de dependências elegante

centro universitário FEI

Criando Fixtures

Use o decorator @pytest.fixture:

```
import pytest

@pytest.fixture
def usuario_valido():
 return {
  "nome": "João Silva",
  "email": "joao@example.com",
  "idade": 25
 }

def test_criar_conta(usuario_valido):
 conta = criar_conta(usuario_valido)
 assert conta.nome == "João Silva"
```

pytest injeta automaticamente a fixture como parâmetro!

centro universitário FEI

Reutilizando Fixtures

Múltiplos testes podem usar a mesma fixture:

```
@pytest.fixture
def carrinho():
 return CarrinhoCompras()

def test_carrinho_vazio(carrinho):
 assert len(carrinho.itens) == 0

def test_adicionar_item(carrinho):
 carrinho.adicionar(Produto("Livro", 50))
 assert len(carrinho.itens) == 1

def test_remover_item(carrinho):
 produto = Produto("Livro", 50)
 carrinho.adicionar(produto)
 carrinho.remover(produto)
 assert len(carrinho.itens) == 0
```

centro universitário FEI

Escopo de Fixtures

Controle quando fixtures são criadas/destruídas:

```
@pytest.fixture(scope="function")  # Padrão
def fixture1():
 # Nova instância para cada teste
 pass

@pytest.fixture(scope="class")
def fixture2():
 # Uma instância por classe de testes
 pass

@pytest.fixture(scope="module")
def fixture3():
 # Uma instância por arquivo
 pass

@pytest.fixture(scope="session")
def fixture4():
 # Uma instância para toda execução
 pass
```

centro universitário FEI

Fixture com Setup e Teardown

Use yield para executar código após o teste:

```
@pytest.fixture
def conexao_bd():
 # Setup: executado antes do teste
 conn = conectar_banco()

 yield conn  # Fornece ao teste

 # Teardown: executado após o teste
 conn.close()

def test_inserir_usuario(conexao_bd):
 conexao_bd.execute("INSERT INTO ...")
 # Conexão será fechada automaticamente
```

centro universitário FEI

Fixtures Built-in: tmp_path

pytest fornece fixtures prontas. Exemplo: tmp_path

```
def test_salvar_arquivo(tmp_path):
    # tmp_path é um diretório temporário único
    arquivo = tmp_path / "dados.txt"

    salvar_dados(arquivo, "conteúdo")

    assert arquivo.exists()
    assert arquivo.read_text() == "conteúdo"
    # Diretório é limpo automaticamente
```

Vantagem

Não precisa gerenciar limpeza de arquivos temporários!

O Problema da Repetição

Testar múltiplos casos similares

Imagine testar uma função com 10 entradas diferentes:

Abordagem ingênua: 10 funções quase idênticas

- `test_caso_1()`
- `test_caso_2()`
- `test_caso_3()`
- `...`
- `test_caso_10()`

Problema

Código duplicado, difícil de manter, suíte poluída!

Parametrização de Testes

Solução elegante

Use @pytest.mark.parametrize:

```
@pytest.mark.parametrize("entrada,esperado", [
(2, 4),
(3, 9),
(4, 16),
(5, 25),
])
def test_quadrado(entrada, esperado):
 assert quadrado(entrada) == esperado
```

pytest executa o teste 4 vezes, uma para cada par de valores!

centro universitário FEI

Parametrização: Output

```
$ pytest -v test_math.py

test_math.py::test_quadrado[2-4] PASSED
test_math.py::test_quadrado[3-9] PASSED
test_math.py::test_quadrado[4-16] PASSED
test_math.py::test_quadrado[5-25] PASSED

============== 4 passed in 0.01s ==============
```

Vantagem

Uma função, múltiplos casos, relatório detalhado de qual caso falhou!

IDs Customizados

Torne os nomes dos testes mais descritivos:

```
@pytest.mark.parametrize("cpf,valido", [
("111.222.333-44", True),
("000.000.000-00", False),
("123.456.789-00", False),
], ids=["cpf_valido", "cpf_zeros", "cpf_invalido"])
def test_validar_cpf(cpf, valido):
 assert validar_cpf(cpf) == valido
```

Output:

```
test_cpf.py::test_validar_cpf[cpf_valido] PASSED
test_cpf.py::test_validar_cpf[cpf_zeros] FAILED
```

centro universitário FEI

Múltiplos Parâmetros

Parametrize vários argumentos:

```
@pytest.mark.parametrize("x,y,operacao,esperado", [
(10, 5, "soma", 15),
(10, 5, "subtracao", 5),
(10, 5, "multiplicacao", 50),
(10, 5, "divisao", 2),
])
def test_calculadora(x, y, operacao, esperado):
 calc = Calculadora()
 resultado = calc.executar(x, y, operacao)
 assert resultado == esperado
```

centro universitário FEI

Combinando Parametrização e Fixtures

Parametrize pode usar fixtures:

```
@pytest.fixture
def calculadora():
return Calculadora()

@pytest.mark.parametrize("a,b,esperado", [
(2, 3, 5),
(10, 20, 30),
(-5, 5, 0),
])
def test_somar(calculadora, a, b, esperado):
 resultado = calculadora.somar(a, b)
 assert resultado == esperado
```

Fixture é criada para cada combinação de parâmetros!

centro universitário FEI

Organizando Testes com Classes

Agrupe testes relacionados:

```
class TestCarrinhoCompras:

def test_carrinho_vazio(self):
 carrinho = CarrinhoCompras()
 assert len(carrinho.itens) == 0

def test_adicionar_item(self):
 carrinho = CarrinhoCompras()
 carrinho.adicionar(Produto("Livro", 50))
 assert len(carrinho.itens) == 1

def test_calcular_total(self):
 carrinho = CarrinhoCompras()
 carrinho.adicionar(Produto("Livro", 50))
 assert carrinho.total == 50
```

centro universitário FEI

Fixtures em Nível de Classe

Compartilhe fixture entre métodos de uma classe:

```
class TestCarrinhoCompras:

@pytest.fixture
def carrinho(self):
 return CarrinhoCompras()

def test_carrinho_vazio(self, carrinho):
 assert len(carrinho.itens) == 0

def test_adicionar_item(self, carrinho):
 carrinho.adicionar(Produto("Livro", 50))
 assert len(carrinho.itens) == 1
```

Nova instância para cada método (isolamento garantido)!

centro universitário FEI

Arquivo conftest.py

Compartilhar fixtures entre módulos

Problema: Fixtures definidas em um arquivo só funcionam nele

Solução: Crie conftest.py no diretório de testes

Estrutura:

```
tests/
conftest.py       # Fixtures compartilhadas
test_carrinho.py
test_produto.py
test_usuario.py
```

Importante

Fixtures em conftest.py ficam disponíveis para **todos** os testes do diretório e subdiretórios!

centro universitário FEI

Exemplo: conftest.py

conftest.py:

```
import pytest

@pytest.fixture
def usuario_admin():
 return Usuario("admin", "admin@test.com", role="admin")

@pytest.fixture
def usuario_comum():
 return Usuario("user", "user@test.com", role="user")
```

test_permissoes.py:

```
def test_admin_pode_deletar(usuario_admin):
 assert usuario_admin.pode_deletar() == True

def test_usuario_nao_pode_deletar(usuario_comum):
 assert usuario_comum.pode_deletar() == False
```

centro universitário FEI

Nomenclatura de Testes

Testes devem ser autodescritivos

Ruim:

- `test_1()`
- `test_funcao()`
- `test_erro()`

Bom:

- `test_carrinho_vazio_tem_zero_itens()`
- `test_adicionar_produto_duplicado_aumenta_quantidade()`
- `test_divisao_por_zero_levanta_excecao()`

Dica

Nome deve descrever **o que** está sendo testado e **qual** comportamento esperado!

Test Smells

Sinais de testes problemáticos

Testes Dependentes:
Um teste depende do resultado de outro – viola princípio Independent

Testes Lentos:
Leva segundos/minutos – viola princípio Fast

Setup Excessivo:
Muitas linhas de preparação – use fixtures!

Assertions Vagas:
assert x vs assert x == valor_esperado

Teste Testando Múltiplas Coisas:
Dificulta identificar o que falhou – divida em testes menores

centro universitário FEI

Um Assert por Teste?

Debate

Argumento PRÓ um assert:

- Testes focados em um comportamento
- Fácil identificar causa da falha
- Seguem SRP (Single Responsibility Principle)

Argumento CONTRA (múltiplos asserts):

- Às vezes múltiplos asserts verificam **um comportamento**
- Evita duplicação de setup
- Pragmático em certos casos

Recomendação

Um conceito por teste, não necessariamente um assert.
Use bom senso!

centro universitário FEI

Exemplo Guiado 1: Fixture Básica

Demonstração

```
@pytest.fixture
def produtos():
 return [
 Produto("Notebook", 3000),
 Produto("Mouse", 50),
 Produto("Teclado", 150)
 ]

def test_buscar_produto_existente(produtos):
 resultado = buscar_produto(produtos, "Mouse")
 assert resultado.preco == 50

def test_buscar_produto_inexistente(produtos):
 resultado = buscar_produto(produtos, "Monitor")
 assert resultado is None
```

centro universitário FEI

Exemplo Guiado 2: Parametrização

Demonstração ao vivo

Testando validação de email com múltiplos casos:

```
@pytest.mark.parametrize("email,valido", [
("usuario@example.com", True),
("usuario@dominio.com.br", True),
("usuario", False),
("@example.com", False),
("usuario@", False),
("usuario @example.com", False),
], ids=["valido_simples", "valido_br", "sem_arroba",
"sem_usuario", "sem_dominio", "com_espaco"])
def test_validar_email(email, valido):
assert validar_email(email) == valido
```

centro universitário FEI

Exemplo Guiado 3: Testando Exceções

Demonstração ao vivo

Validando que função lança exceção apropriada:

```
def test_saque_acima_do_saldo():
conta = ContaBancaria(saldo=100)

with pytest.raises(SaldoInsuficienteError) as exc:
conta.sacar(200)

assert "Saldo insuficiente" in str(exc.value)
assert conta.saldo == 100  # Saldo não mudou

def test_saque_valor_negativo():
conta = ContaBancaria(saldo=100)

with pytest.raises(ValueError):
conta.sacar(-50)
```

centro universitário FEI

Exercício Prático: Sistema de Validação de CPF

Projeto completo

Implementar em cpf.py:

1. validar_cpf(cpf: str) -> bool
   - Valida formato (11 dígitos)
   - Valida dígitos verificadores
   - Retorna True se válido, False caso contrário
2. formatar_cpf(cpf: str) -> str
   - Recebe CPF sem formatação: "12345678901"
   - Retorna formatado: "123.456.789-01"
   - Levanta ValueError se CPF inválido

Tempo: 60 minutos

centro universitário FEI

Exercício: Requisitos de Teste

Criar em `test_cpf.py` **usando:**

1. **Fixtures (conftest.py ou no arquivo):**
   - `cpfs_validos`: lista de CPFs válidos
   - `cpfs_invalidos`: lista de CPFs inválidos
2. **Parametrização:**
   - Testar múltiplos CPFs válidos
   - Testar múltiplos CPFs inválidos
3. **Testes de Exceção:**
   - `formatar_cpf` com entrada inválida
4. **Padrão AAA em todos os testes**

centro universitário FEI

Exercício: Casos de Teste Sugeridos

Mínimo de 10 testes cobrindo:

- CPF válido padrão
- CPF válido com zeros
- CPF inválido (dígitos verificadores errados)
- CPF com todos dígitos iguais (111.111.111-11)
- CPF com menos de 11 dígitos
- CPF com mais de 11 dígitos
- CPF com letras
- Formatação de CPF válido
- Formatação de CPF inválido (exceção)
- CPF None ou string vazia

Execute com: `pytest -v test_cpf.py`

centro universitário FEI

Resumo do Tópico 2

Teste de Unidade com pytest – Aprofundamento

Conceitos dominados:

- Princípios FIRST para testes de qualidade
- Padrão AAA (Arrange-Act-Assert)
- Assertions avançadas e testes de exceções

Técnicas pytest:

- Fixtures para reutilização (function, class, module, session)
- Parametrização com @pytest.mark.parametrize
- Fixtures built-in (tmp_path, etc.)
- Arquivo conftest.py

Boas práticas:

- Nomenclatura descritiva
- Organização com classes
- Evitar test smells

Próximo tópico: Técnicas de Caixa-Preta

centro universitário FEI

Simulação e Teste de Software (CC8550)

Aula 02 – Teste de unidade com pytest

Prof. Luciano Rossi

Ciência da Computação
Centro Universitário FEI

1º Semestre de 2026