Testes e Exceções em Python

Aula 14 e 15

Ana Júlia Lima

Desenvolvido para
Young 1 - sexta #5228

Ctrl Play
Brasil
Outubro 2025

Sumário

1 Testes em Python (unittest)

1.1 Por que testar?

Testes são pequenos programas que exercitam o seu código com entradas controladas e verificam se a saída está correta. Eles evitam regressões, documentam o comportamento esperado e dão confiança para refatorar.

1.2 O que são testes?

Em Python, a biblioteca padrão `unittest` fornece:

- **Casos de teste** (`TestCase`): agrupam métodos de teste;
- **Asserts**: verificam resultados;
- **Runner**: executa e relata o status dos testes.

1.3 Como pensar em testes (AAA)

Siga o padrão **Arrange–Act–Assert**:

1. **Arrange**: preparo dos dados/objetos;
2. **Act**: chamada da função/método sob teste;
3. **Assert**: verificação com asserts.

Cubra casos felizes, bordas (vazio, `None`, extremos), erros esperados e tipos inválidos (quando fizer sentido).

1.4 Testes de unidade com `unittest`

Crie classes que herdam de `unittest.TestCase`. Cada método que começa com `test_` é um teste. Principais asserts:

- `self.assertEqual(a, b)`, `self.assertNotEqual(a, b)`
- `self.assertTrue(x)`, `self.assertFalse(x)`
- `self.assertIn(item, colecao)`, `self.assertNotIn(item, colecao)`

Dica (notebooks) Use `unittest.main(argv=[''], exit=False)` para evitar sair do interpretador ao finalizar os testes.

1.5 Como vemos a função passando no teste?

Na execução, o runner mostra `.` para *pass*, `F` para *fail* e `E` para *error*. Ao final, um sumário indica quantos testes passaram/falharam, com mensagens e *tracebacks* úteis.

1.6 Testando funções

```python
# arquivo: math_utils.py
def somar(a, b):
    return a + b

def eh_par(n):
    return n % 2 == 0
```

```python
# arquivo: test_math_utils.py
import unittest
from math_utils import somar, eh_par

class TestMathUtils(unittest.TestCase):
    def test_somar_basico(self):
        # Arrange & Act
        resultado = somar(2, 2)
        # Assert
        self.assertEqual(resultado, 4)
        self.assertNotEqual(resultado, 5)

    def test_eh_par_exemplos(self):
        # Várias entradas em um único teste
        for n, esperado in [(0, True), (1, False), (2, True), (-3, False)]:
            with self.subTest(n=n):
                self.assertEqual(eh_par(n), esperado)

    def test_asserts_varios(self):
        numeros = [1, 2, 3]
        self.assertTrue(eh_par(2))
        self.assertFalse(eh_par(3))
        self.assertIn(2, numeros)
        self.assertNotIn(4, numeros)

if __name__ == '__main__':
    # Em ambientes interativos (ex.: Jupyter), use:
    unittest.main(argv=[''], exit=False)
    # Em scripts normais, pode ser apenas: unittest.main()
```

1.7 Como testar uma classe? `setUp()`

```python
# arquivo: conta.py
class ContaBancaria:
    def __init__(self, titular, saldo_inicial=0.0):
        self.titular = titular
        self.saldo = float(saldo_inicial)

    def depositar(self, valor):
        if valor <= 0:
            raise ValueError("Valor do depósito deve ser positivo.")
        self.saldo += valor

    def sacar(self, valor):
        if valor <= 0:
            raise ValueError("Valor do saque deve ser positivo.")
        if valor > self.saldo:
            raise ValueError("Saldo insuficiente.")
        self.saldo -= valor
```

```python
# arquivo: test_conta.py
import unittest
from conta import ContaBancaria

class TestContaBancaria(unittest.TestCase):
```

```
    def setUp(self):
        # Cria uma nova conta antes de CADA teste (estado limpo)
        self.conta = ContaBancaria("Ana", saldo_inicial=100.0)

    def test_depositar_incrementa_saldo(self):
        self.conta.depositar(50.0)       # Act
        self.assertEqual(self.conta.saldo, 150.0)  # Assert

    def test_sacar_decrementa_saldo(self):
        self.conta.sacar(40.0)
        self.assertEqual(self.conta.saldo, 60.0)

    def test_sacar_maior_que_saldo_lanca_erro(self):
        with self.assertRaises(ValueError):
            self.conta.sacar(999.0)

    def test_depositar_valor_invalido_lanca_erro(self):
        for invalido in [0, -10]:
            with self.subTest(valor=invalido):
                with self.assertRaises(ValueError):
                    self.conta.depositar(invalido)

if __name__ == '__main__':
    unittest.main(argv=[''], exit=False)
```

1.8 Boas práticas rápidas

- Nomeie testes claramente: `test_acao_condicao_resultado`.
- Um cenário por teste; use `subTest` para variações.
- Evite dependência entre testes; isole com `setUp()`.
- Cubra casos de borda e erros esperados (`assertRaises`).

2 Tratamento de Exceções em Python

2.1 O que são exceções?

Durante a execução de um programa, podem ocorrer situações inesperadas — como tentar dividir por zero, acessar um arquivo inexistente ou converter texto em número. Essas situações geram **exceções**, que são interrupções controladas no fluxo do programa.

2.2 Por que tratar exceções?

- Evita que o programa quebre de forma abrupta;
- Melhora a experiência do usuário;
- Facilita a depuração e manutenção;
- Permite prever comportamentos e garantir estabilidade.

2.3 Erros comuns em Python

- `SyntaxError` — erro de sintaxe (esquecendo `:` ou `)`);
- `ZeroDivisionError` — divisão por zero;
- `ValueError` — tipo de valor incorreto;
- `NameError` — variável inexistente;
- `TypeError` — operação com tipos incompatíveis;
- `IndexError` — índice fora do alcance da lista;
- `FileNotFoundError` — arquivo inexistente.

2.4 try, except e else

O bloco `try` é usado para envolver o código que pode gerar exceções. O bloco `except` captura e trata o erro. O bloco `else` é executado apenas se não ocorrer erro.

```
try:
    n1 = int(input("Digite um número: "))
    n2 = int(input("Digite outro número: "))
    resultado = n1 / n2
except ValueError:
    print("Erro: você precisa digitar apenas números inteiros.")
except ZeroDivisionError:
    print("Erro: não é possível dividir por zero.")
else:
    print(f"Resultado: {resultado}")
```

Se ocorrer erro, o bloco `except` é executado. Se tudo ocorrer bem, o bloco `else` é executado.

2.5 finally

O bloco `finally` é executado sempre, independentemente de ter ocorrido erro ou não. É útil para liberar recursos, como fechar arquivos, desconectar banco de dados, etc.

```
try:
    arquivo = open("dados.txt", "r")
    conteudo = arquivo.read()
    print(conteudo)
except FileNotFoundError:
    print("Arquivo não encontrado.")
finally:
    print("Encerrando operação...")
    if 'arquivo' in locals():
        arquivo.close()
```

2.6 Resumo rápido

- `try`: tenta executar um bloco de código;
- `except`: trata o erro, se ocorrer;
- `else`: executa se não houver erro;
- `finally`: executa sempre, com ou sem erro.