Testes de Unidade e Integração
Aplicação Prática - Calculadora

Luciano Rossi

08/Setembro/2025

1 Objetivo

Desenvolver e executar testes de unidade e integração para uma calculadora simples. O objetivo é aplicar diferentes tipos de teste aprendidos na teoria, com exemplos práticos e específicos para cada categoria.

2 Descrição do Sistema

A calculadora possui as seguintes operações básicas:

- Somar dois números
- Subtrair dois números
- Multiplicar dois números
- Dividir dois numeros (com tratamento de divisão por zero)
- Calcular potência
- Manter histórico das operações

Código base da calculadora:

```python
import math

class Calculadora:
  def __init__(self):
    self.historico = []
    self.resultado = 0

  def somar(self, a, b):
    if not isinstance(a, (int, float)) or not isinstance(b, (int, float)):
      raise TypeError("Argumentos devem ser numeros")
    resultado = a + b
    self.historico.append(f"{a} + {b} = {resultado}")
    self.resultado = resultado
    return resultado

  def subtrair(self, a, b):
    if not isinstance(a, (int, float)) or not isinstance(b, (int, float)):
      raise TypeError("Argumentos devem ser numeros")
    resultado = a - b
```

```
    self.historico.append(f"{a} - {b} = {resultado}")
    self.resultado = resultado
    return resultado

  def multiplicar(self, a, b):
    if not isinstance(a, (int, float)) or not isinstance(b, (int, float)):
      raise TypeError("Argumentos devem ser numeros")
    resultado = a * b
    self.historico.append(f"{a} * {b} = {resultado}")
    self.resultado = resultado
    return resultado

  def dividir(self, a, b):
    if not isinstance(a, (int, float)) or not isinstance(b, (int, float)):
      raise TypeError("Argumentos devem ser numeros")
    if b == 0:
      raise ValueError("Divisao por zero nao permitida")
    resultado = a / b
    self.historico.append(f"{a} / {b} = {resultado}")
    self.resultado = resultado
    return resultado

  def potencia(self, base, expoente):
    if not isinstance(base, (int, float)) or not isinstance(expoente, (int,
        float)):
      raise TypeError("Argumentos devem ser numeros")
    resultado = base ** expoente
    self.historico.append(f"{base} ^ {expoente} = {resultado}")
    self.resultado = resultado
    return resultado

  def limpar_historico(self):
    self.historico.clear()

  def obter_ultimo_resultado(self):
    return self.resultado
```

3 Testes de Unidade

Implemente os seguintes tipos de teste com os exemplos específicos fornecidos:

3.1 Testes de Entrada e Saída

Objetivo: Validar se os parâmetros são interpretados corretamente e os valores retornados estão corretos.

```
def test_entrada_saida_soma(self):
  calc = Calculadora()
  resultado = calc.somar(5, 3)
  self.assertEqual(resultado, 8)
  self.assertEqual(calc.obter_ultimo_resultado(), 8)
```

Implemente: Teste similar para subtração, multiplicação e divisão.

3.2 Testes de Tipagem

Objetivo: Confirmar que tipos incorretos são rejeitados.

```python
def test_tipagem_invalida(self):
  calc = Calculadora()
  with self.assertRaises(TypeError):
    calc.somar("5", 3)  # String no lugar de numero
  with self.assertRaises(TypeError):
    calc.dividir(10, None)  # None no lugar de numero
```

Implemente: Teste tipagem para todas as operações matemáticas.

3.3 Testes de Consistência

Objetivo: Verificar se os dados permanecem consistentes apos operações.

```python
def test_consistencia_historico(self):
  calc = Calculadora()
  calc.somar(2, 3)
  calc.multiplicar(4, 5)
  self.assertEqual(len(calc.historico), 2)
  self.assertIn("2 + 3 = 5", calc.historico)
  self.assertIn("4 * 5 = 20", calc.historico)
```

3.4 Testes de Inicialização

Objetivo: Garantir que a estrutura é inicializada corretamente.

```python
def test_inicializacao(self):
  calc = Calculadora()
  self.assertEqual(calc.resultado, 0)
  self.assertEqual(len(calc.historico), 0)
```

3.5 Testes de Modificação de Dados

Objetivo: Verificar se modificações são aplicadas corretamente.

```python
def test_modificacao_historico(self):
  calc = Calculadora()
  calc.somar(1, 1)
  self.assertEqual(len(calc.historico), 1)
  calc.limpar_historico()
  self.assertEqual(len(calc.historico), 0)
```

3.6 Testes de Limite Inferior

Objetivo: Testar comportamento com valores mínimos.

```python
def test_limite_inferior(self):
  calc = Calculadora()
  # Teste com zero
  resultado = calc.somar(0, 5)
  self.assertEqual(resultado, 5)
  # Teste com numeros negativos muito pequenos
  resultado = calc.multiplicar(-1e-10, 2)
  self.assertEqual(resultado, -2e-10)
```

3.7 Testes de Limite Superior

Objetivo: Testar comportamento com valores máximos.

```
def test_limite_superior(self):
  calc = Calculadora()
  # Teste com numeros grandes
  resultado = calc.somar(1e10, 1e10)
  self.assertEqual(resultado, 2e10)
```

Implemente: Teste com valores próximos ao limite de float do Python.

3.8 Testes de Valores Fora do Intervalo

Objetivo: Verificar comportamento com valores inválidos.

```
def test_divisao_por_zero(self):
  calc = Calculadora()
  with self.assertRaises(ValueError):
    calc.dividir(10, 0)
```

3.9 Testes de Fluxos de Controle

Objetivo: Testar diferentes caminhos do código.

```
def test_fluxos_divisao(self):
  calc = Calculadora()
  # Caminho normal
  resultado = calc.dividir(10, 2)
  self.assertEqual(resultado, 5)
  # Caminho de erro
  with self.assertRaises(ValueError):
    calc.dividir(10, 0)
```

3.10 Testes de Mensagens de Erro

Objetivo: Verificar se mensagens de erro são claras.

```
def test_mensagens_erro(self):
  calc = Calculadora()
  try:
    calc.dividir(5, 0)
  except ValueError as e:
    self.assertEqual(str(e), "Divisao por zero nao permitida")
```

4 Parte 2: Testes de Integração

4.1 Teste de Operações Sequenciais

Objetivo: Verificar se múltiplas operações funcionam em conjunto.

```
def test_operacoes_sequenciais(self):
  calc = Calculadora()
  # Sequencia: 2 + 3 = 5, depois 5 * 4 = 20, depois 20 / 2 = 10
  calc.somar(2, 3)
  resultado1 = calc.obter_ultimo_resultado()

  calc.multiplicar(resultado1, 4)
```

```
    resultado2 = calc.obter_ultimo_resultado()

    calc.dividir(resultado2, 2)
    resultado_final = calc.obter_ultimo_resultado()

    self.assertEqual(resultado_final, 10)
    self.assertEqual(len(calc.historico), 3)
```

4.2 Teste de Interface entre Métodos

Objetivo: Verificar se diferentes métodos se comunicam corretamente.

```
def test_integracao_historico_resultado(self):
    calc = Calculadora()
    calc.potencia(2, 3)  # 2^3 = 8
    calc.somar(calc.obter_ultimo_resultado(), 2)  # 8 + 2 = 10

    self.assertEqual(calc.obter_ultimo_resultado(), 10)
    self.assertEqual(len(calc.historico), 2)
    self.assertIn("2 ^ 3 = 8", calc.historico)
    self.assertIn("8 + 2 = 10", calc.historico)
```

5 Tarefas

1. **Complete os exemplos**: Implemente os testes marcados como "Implemente" seguindo os padrões mostrados.
2. **Adicione testes extras**: Para cada categoria, crie pelo menos um teste adicional não mostrado nos exemplos.
3. **Execute e documente**: Execute todos os testes e documente:
   - Quantos testes passaram/falharam
   - Qual foi a cobertura de código (use `coverage.py`)
   - Quais problemas foram encontrados (se houver)
4. **Melhore o código**: Se encontrar problemas no código da calculadora, corrija-os e documente as correções.

6 Entrega

Entregue o endereço de um repositório (GitHub) contendo:

- O código da calculadora (corrigido, se necessário)
- Todos os testes implementados
- Comentários explicando cada tipo de teste
- Um relatório como comentário no final do arquivo com:
  - Resultado da execução dos testes
  - Cobertura de código obtida
  - Problemas encontrados e soluções aplicadas
  - Lições aprendidas sobre cada tipo de teste

7 Estrutura de Arquivos Sugerida

Organize seu projeto da seguinte forma:

```
projeto_calculadora/
|-- src/
|   |-- calculadora.py          # Codigo da calculadora
|-- tests/
|   |-- __init__.py             # Arquivo vazio
|   |-- test_unidade.py         # Testes de unidade
|   |-- test_integracao.py      # Testes de integracao
|-- requirements.txt            # Dependencias
|-- README.md                   # Documentacao
|-- relatorio.md                # Relatorio dos testes
```

7.1 Conteúdo dos Arquivos

requirements.txt:

```
coverage>=7.0.0
```

Comandos para execução:

```
# Instalar dependencias
pip install -r requirements.txt

# Executar todos os testes
python -m unittest discover tests -v

# Executar com cobertura
coverage run -m unittest discover tests
coverage report
coverage html

# Executar teste especifico
python -m unittest tests.test_unidade.TestCalculadora.test_soma -v
```

8 Critérios de Avaliação

- **Completude (30%):** Todos os tipos de teste implementados
- **Correcao (25%):** Testes executam e validam corretamente
- **Qualidade (25%):** Testes bem estruturados e documentados
- **Analise (20%):** Relatório demonstra compreensão dos conceitos