O arcabouço de teste - unittest

Jailma Januário da Silva

Leonardo Massayuki Takuno

Resumo

Os objetivos desta parte são: (I) Apresentar maneiras de realizar testes de unidade com arcabouço de testes unittest. (II) Construir casos de testes que recebem os valores obtidos da função e confrontar com os valores esperados no teste. (III) Tratar casos de testes que esperam uma exceção como resultado.

Introdução

O assunto de testes automatizados no Python é amplo e pode ter vários níveis de complexidade, porém não é preciso ser complicado. É possível iniciar a atividade de testes criando pequenos testes na sua aplicação, e aos poucos construir uma base de conhecimento de testes para qualquer projeto de software.

Este texto pretende mostrar como criar alguns testes básicos, executar e encontrar falhas antes de ir para a produção. Você irá aprender a utilizar ferramentas disponíveis para escrever e executar testes automatizados, e após isso, analisar os resultados dessas ferramentas e corrigir o seu programa caso seja necessário.

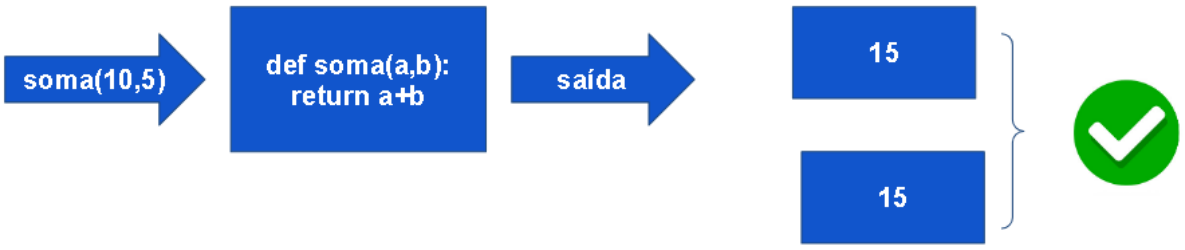
A parte 01 mostrou como realizar testes de unidade utilizando a instrução assert e realizando o tratamento da exceção AssertionError, o que indica que houve algum tipo de problema na execução da função. Esta parte apresenta outras maneiras de se realizar testes de unidades através dos chamados *test runners*. Um *test runner* é um arcabouço de testes automatizado (do inglês, *framework*) que contém ferramentas para realizar a execução dos testes e apresentar os resultados indicando para cada teste a ocorrência de sucesso

ou falha. Segundo a Python Software Foundation (2001), um *test runner* orquestra a execução de testes e pode fornecer o resultado em uma interface gráfica ou textual. Entre os *test runners* disponíveis para a linguagem Python, dois se destacam: o unittest e o pytest. Ambos serão objeto de estudo nesta disciplina.

O arcabouço unittest

Segundo a Python Software Foundation (2001), o arcabouço de testes de unidade unittest foi inspirado em uma ferramenta para testes na linguagem Java chamada JUnit, e possui estrutura muito semelhante e adaptado para a linguagem Python. Esta seção apresenta um exemplo simples de utilização do unittest, porém, apesar de simples, este exemplo será muito útil para atender às necessidades de vários desenvolvedores de software. Seguindo o mesmo raciocínio da parte 01, a Figura 2.1 ilustra um teste de unidade para a função soma.

Figura 2.1. Teste de unidade da função soma



Fonte: do autor, 2022.

Crie um arquivo chamado test_ex01.py e escreva o teste de acordo com a Codificação 2.1. Observe que a função soma é a unidade a ser testada, além disso, este arquivo contém, também, uma classe chamada TesteSoma herdada de unittest.TestCase, a qual é uma classe que pode conter uma coleção de casos de testes. Um caso de teste, por sua vez, é uma unidade de teste individual que verifica uma resposta específica a um determinado conjunto de entradas (Python Software Foundation, 2001). Ao estender a

classe unittest.TestCase, é possível criar novos casos de testes específicos para o elemento que se deseja testar.

Codificação 2.1. Primeiro teste com unittest

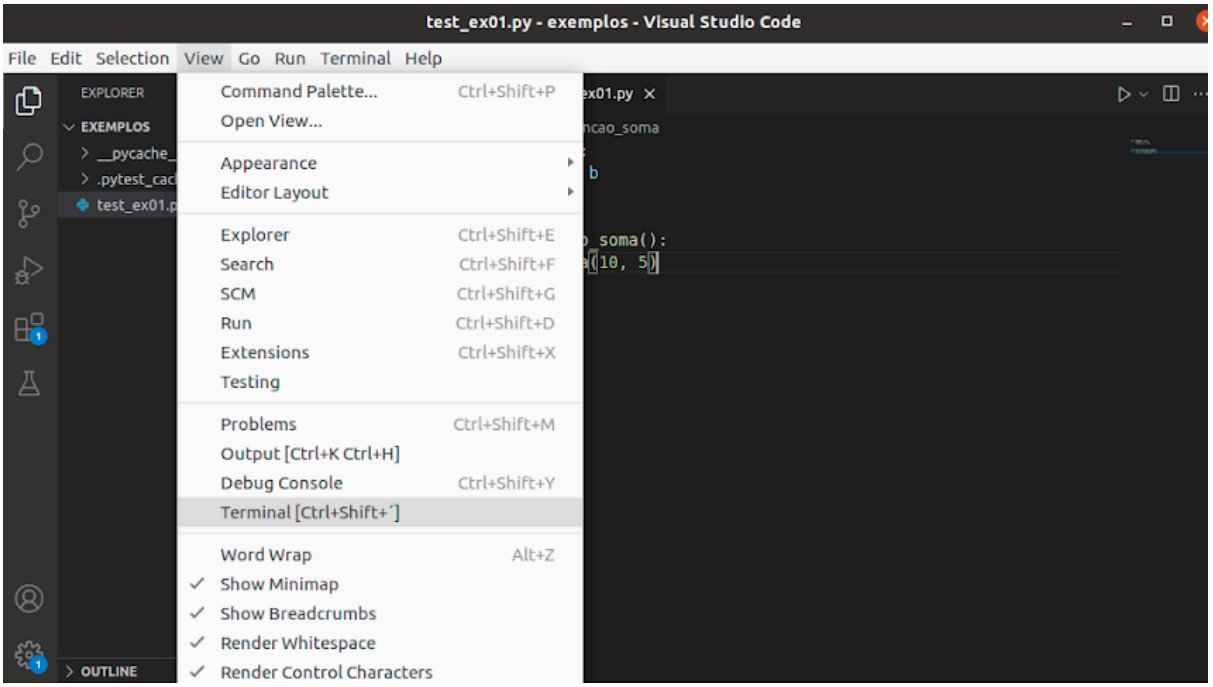
```
import unittest
def soma(a, b):
        return a + b
class TesteSoma(unittest.TestCase):
        def test_funcao_soma(self):
                self.assertEqual(soma(10, 5), 15)
if __name__ == '__main__':
        unittest.main()
```

Fonte: do autor, 2022.

Para verificar se o valor obtido da função soma é igual ao valor esperado, o unittest fornece a instrução assertEqual(). Por convenção do unittest, todo caso de teste consiste em um método cujo nome inicia-se pelo prefixo test. Assim, o test runner consegue identificar quais são os métodos de teste. Na Codificação 2.1 é possível observar o primeiro caso de teste, denominado test_funcao_soma(), que ao passar os valores 10 e 5 para a função soma, o teste aguarda como resultado o valor 15. Caso o valor devolvido esteja correto, então a função executou de maneira para este cenário. Caso contrário, a função irá gerar uma exceção de AssertionError, e o teste detectou uma falha na função que deverá ser corrigida. É possível realizar quantos casos de testes forem necessários.

No Visual Studio Code, que é a IDE utilizada neste curso, acesse o menu view/terminal, conforme apresenta a Figura 2.2.

Figura 2.2. Iniciar o terminal a partir do Visual Studio Code



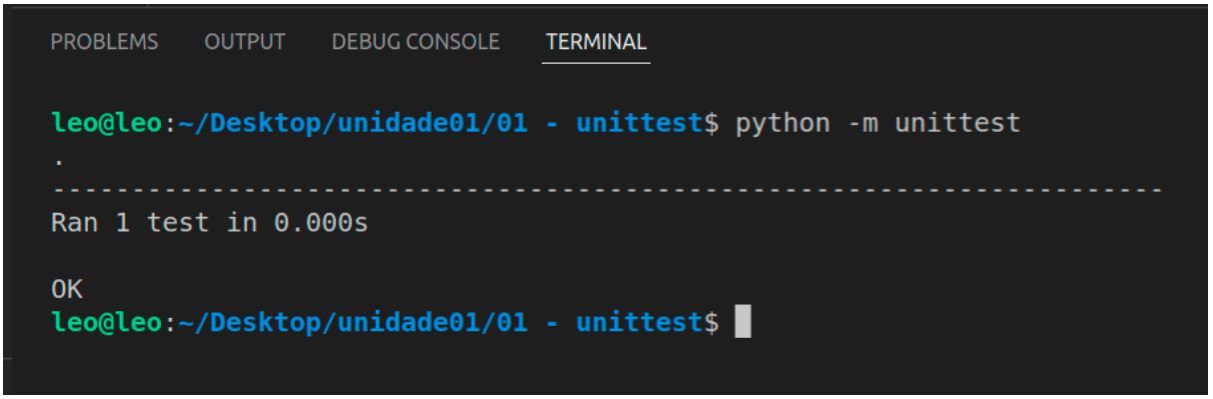
Fonte: do autor, 2022.

Para executar o teste execute a instrução:

$ python -m unittest

A Figura 2.3 apresenta a execução do teste de unidade sobre o arquivo test_ex01.py.

Figura 2.3. Executando o teste de unidade com unittest

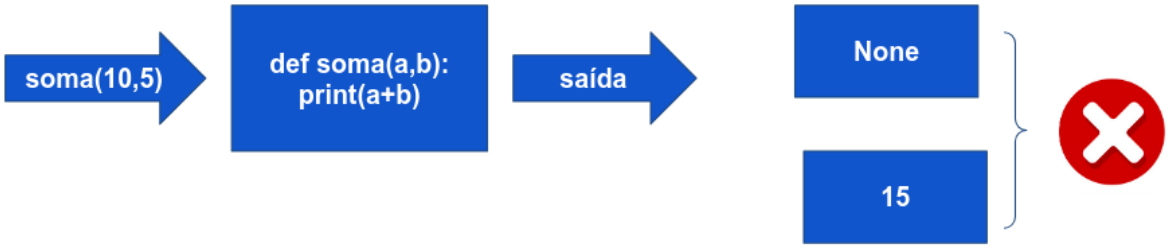


Fonte: do autor, 2022.

Observe que, de acordo com a saída fornecida pela execução do programa, o resultado indica que a execução de um teste foi finalizada corretamente. Para cada teste realizado com sucesso, o unittest apresenta um ponto (.), além de indicar a quantidade de testes executados, que neste exemplo, foi apenas um.

O próximo exemplo, de acordo com a Figura 2.4, ilustra que a função soma que recebe como parâmetros os valores 10 e 5, e imprime um valor que é o resultado da soma através da função print, mas não devolve o valor algum de resultado. O caso de teste recebe um valor None como valor de saída, e ao comparar com o valor o valor esperado 15, ocorre um erro indicando que a função soma não está executando da maneira adequada.

Figura 2.4. Teste de unidade da função soma, com erro



Fonte: do autor, 2022.

Para verificar este cenário utilizando o unittest, crie um arquivo chamado test_ex02.py e escreva o teste de acordo com a Codificação 2.2.

Codificação 2.2. Teste da função soma com falha

```
import unittest
def soma(a, b):
        print(a + b)
class TesteSoma(unittest.TestCase):
        def test_funcao_soma(self):
                self.assertEqual(soma(10, 5), 15)
if __name__ == '__main__':
        unittest.main()
```

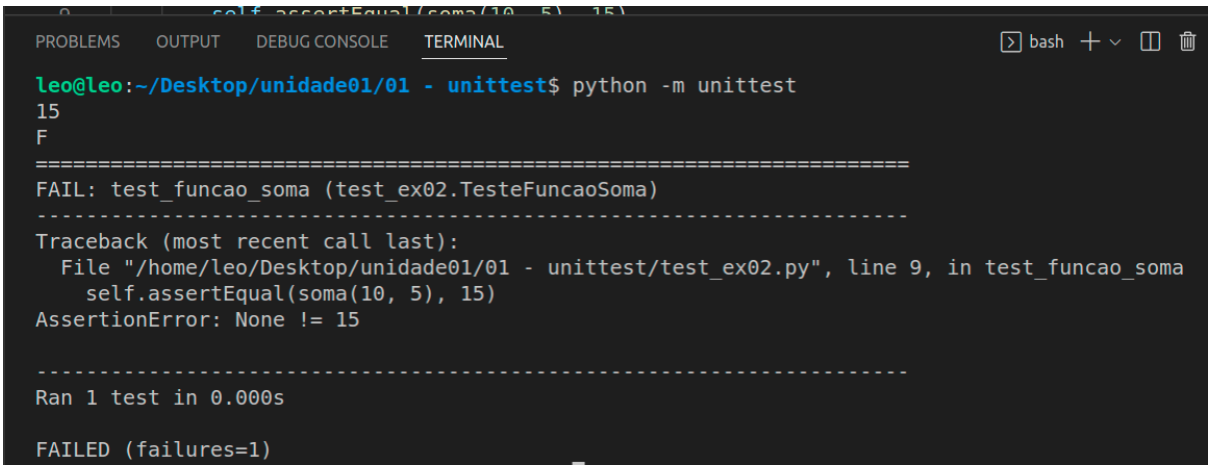
Fonte: do autor, 2022.

Para executar o teste execute a instrução:

$ python -m unittest

A Figura 2.5 apresenta a execução do teste de unidade sobre o arquivo test_ex02.py.

Figura 2.5. Teste de unidade apresentando erro de asserção



Fonte: do autor, 2022.

Observe pela Figura 2.5 que ao executar o teste, a primeira linha apresenta o número 15, que é o resultado da instrução print que foi executada dentro da função soma. Na linha seguinte, há a indicação da letra F , o que significa que houve falha. Na sequência, o python apresenta qual foi a falha ocorrida, que neste caso foi um AssertionError indicando que o valor None é diferente de 15.

Testes de funções que geram Exceções

Até agora, foi possível realizar um teste para verificar se dados dois números fornecidos para a função soma, ela devolve o resultado corretamente. O que acontece se a função receber como parâmetro valores inesperados? É comum, nestes casos, que uma exceção seja lançada. Então, se a função soma foi projetada para receber valores do tipo inteiro, caso a função receba uma string, ou booleano, ou float como parâmetro, então a ela deve devolver uma exceção.

O pacote do unittest fornece uma verificação de lançamento de exceções por meio do método assertRaises. Observe a Codificação 2.3, em que a função soma realiza uma verificação do tipo de dados do parâmetro, e devolve uma exceção do tipo TypeError caso o tipo de dados do parâmetro a ou do parâmetro b seja diferente de int.

Codificação 2.3. Teste de funções que geram exceções

```
import unittest
def soma(a, b):
  if type(a) == int and type(b) == int:
    return a + b
  else:
    raise TypeError(f'Tipos incompatíveis')
class TesteSoma(unittest.TestCase):
  def test_funcao_soma(self):
    self.assertEqual(soma(10, 5), 15)
  def test_excecao_tipos_incompativeis(self):
    self.assertRaises(TypeError, soma, 10, '5')
  def test_excecao_tipos_string(self):
    self.assertRaises(TypeError, soma, '10', '5')
  def test_excecao_tipos_float(self):
    self.assertRaises(TypeError, soma, 10.555, 5.12)
  def test_excecao_tipos_boolean(self):
    self.assertRaises(TypeError, soma, True, False)
if __name__ == '__main__':
      unittest.main()
```

Fonte: do autor, 2022.

Em resumo, o teste da Codificação 2.3 garante que a função soma só deve receber números inteiros como parâmetro de entrada e qualquer tipo de dados diferente do esperado será rejeitada pela função.

Separando o arquivo de teste do módulo a ser testado

Para fins didáticos, os exemplos apresentados até o momento definem a função a ser testada e a classe de testes em um só arquivo. Porém, em um projeto de desenvolvimento de software, o arquivo de teste deve ser separado

da função. A Codificação 2.4 apresenta a função soma em um arquivo separado chamado aritmetica.py.

Codificação 2.4. Módulo aritmetica.py

```
def soma(a, b):
  if type(a) == int and type(b) == int:
    return a + b
  else:

    raise TypeError(f'Tipos incompatíveis')
```

Fonte: do autor, 2022.

A Codificação 2.5 apresenta o código para o teste da função soma, como segue.

Codificação 2.5. Arquivo de testes separado da função a ser testada

```
import unittest
from aritmetica import soma
class TesteSoma(unittest.TestCase):
  def test_funcao_soma(self):
    self.assertEqual(soma(10, 5), 15)
  def test_excecao_tipos_incompativeis(self):
    self.assertRaises(TypeError, soma, 10, '5')
  def test_excecao_tipos_string(self):
    self.assertRaises(TypeError, soma, '10', '5')
  def test_excecao_tipos_float(self):
    self.assertRaises(TypeError, soma, 10.555, 5.12)
  def test_excecao_tipos_boolean(self):
    self.assertRaises(TypeError, soma, True, False)
if __name__ == '__main__':
      unittest.main()
```

Fonte: do autor, 2022.

Métodos de asserção

A classe TestCase fornece vários métodos de asserção para verificar e relatar as falhas. Observe pela Tabela 2.1 a lista de métodos mais utilizados.

Tabela 2.1. Métodos de asserção

Método	Avalia se	Novo em
`assertEqual(a, b)`	a == b	
`assertNotEqual(a, b)`	a != b	
`assertTrue(x)`	bool(x) is True	
`assertFalse(x)`	boll(x) is False	
`assertIs(a, b)`	a is b	3.1
`assertIsNot(a, b)`	a is not b	3.1
`assertIsNone(x)`	x is None	3.1
`assertIsNotNone(x)`	x is not None	3.1

Fonte: PYTHON SOFTWARE FOUNDATION, 2001.

Existem outros métodos de asserção, das quais incluem o método assertRaises que foi anteriormente utilizado na Codificação 2.5. Para maiores detalhes acesse a URL disponível em: <https://docs.python.org/pt-br/3/library/unittest.html>.

Considerações adicionais

Os testes de unidade são importantes para melhorar a confiança no código, e existem algumas oportunidades para se criar testes de unidade.

- Durante o desenvolvimento: crie testes para o módulo que você está desenvolvendo.
- Durante a correção de defeitos: crie testes para reproduzir o erro antes de corrigi-lo.

Além disso, é desejável levar em consideração as recomendações a seguir:

- Um teste de unidade deve se concentrar em um conjunto muito reduzido de funcionalidades e verificar se tudo está correto.
- Cada teste de unidade deve ser independente.
- O código do teste deve ser simples. Escreva código limpo para os testes de unidade.
- Use nomes que revelem a intenção do teste, mesmo que seja um nome muito longo.
- Escreva testes unidades que ajudam a testar o código, melhorar sua clareza e a corrigir eventuais falhas.

Vamos praticar?

1) Suponha que você foi contratado para realizar testes automatizados para um sistema escolar. O módulo para avaliar a nota de um aluno recebe como entrada três notas obtidas pelo aluno em provas, e também, a média dos exercícios e calcula a média de aproveitamento de acordo com a fórmula abaixo:

$$media\_aproveitamento = \frac{n_1 + 2*n_2 + 3*n_3 + media\_exercicios}{7}$$

A atribuição de conceitos obedece a tabela abaixo:

média de aproveitamento	Conceito
>= 9,0	A
>= 7,5 e < 9	B
>= 6,0 e < 7,5	C
< 6,0	D

Importe o módulo escolar.py abaixo para um programa de teste e escreva testes unitários para as funções do módulo:

```
'''
função: avaliar_notas
entrada: Três notas obtidas por um aluno em provas e
       a média dos exercícios.
saída: conceito (A, B, C ou D)
Restrição: as notas n1, n2 e n3, assim como a média são
        valores entre 0,0 e 10,0.
'''
def avaliar_notas(n1, n2, n3, media_exercicios):
  if n1 < 0 or n1 > 10:
     raise ValueError('Valor inválido para n1')
  if n2 < 0 or n2 > 10:
     raise ValueError('Valor inválido para n2')
  if n3 < 0 or n3 > 10:
     raise ValueError('Valor inválido para n3')
  if media_exercicios < 0 or media_exercicios > 10:
     raise ValueError('Valor inválido para media_exercicios')
  media_aproveitamento = (n1 + 2*n2 + 3*n3 + media_exercicios)/7
  if media_aproveitamento >= 9.0:
     return 'A'
  elif media_aproveitamento >= 7.5 and media_aproveitamento < 9:
     return 'B'
  elif media_aproveitamento >= 6.0 and media_aproveitamento < 7.5:
     return 'C'
  elif media_aproveitamento < 6.0:

     return 'D'
```

Utilize os valores abaixo como parâmetros de entrada e saída da função:

avaliar_notas(n1, n2, n3, media_exercicios)	
Entrada	**Saída**
n1 = -1 n2 = 0 n3 = 0 media_exercicios = 0	exceção: ValueError
n1 = 0 n2 = -1 n3 = 0 media_exercicios = 0	exceção: ValueError
n1 = 0 n2 = 0 n3 = -1 media_exercicios = 0	exceção: ValueError
n1 = 10 n2 = 10 n3 = 10 media_exercicios = 10	'A'
n1 = 9 n2 = 9 n3 = 9 media_exercicios = 9	'A'
n1 = 8.9 n2 = 8.9 n3 = 8.9 media_exercicios = 8.9	'B'
n1 = 7.5 n2 = 7.5 n3 = 7.5 media_exercicios = 7.5	'B'
n1 = 7.4 n2 = 7.4 n3 = 7.4 media_exercicios = 7.4	'C'

2) Uma empresa quer verificar se um empregado está qualificado para a aposentadoria ou não. Para estar em condições, um dos seguintes requisitos deve ser satisfeito:

- Ter no mínimo 65 anos de idade.
- Ter trabalhado no mínimo 30 anos.
- Ter no mínimo 60 anos e ter trabalhado no mínimo 25 anos.

Importe o módulo aposentadoria.py abaixo para um programa de teste e escreva testes unitários para as funções do módulo:

```
'''
função: verificar_qualificacao_empregado()
entrada: idade e tempo_de_servico
saída: 'Requerer aposentadoria' ou 'Não requerer'
Critérios para verificar se o funcionário está
qualificado para aposentadoria:
- Ter no mínimo 65 anos de idade.
- Ter trabalhado no mínimo 30 anos.
- Ter no mínimo 60 anos e ter trabalhado no mínimo 25 anos.
Restrição: os parâmetros de entrada, idade e tempo_de_servico, devem,
necessariamente, ser números inteiros e maiores que zero.
'''
REQUERER = 'Requerer aposentadoria'
NAO_REQUERER = 'Não requerer'
def verificar_qualificacao_empregado(idade, tempo_de_servico):
  if idade >= 65:
    return REQUERER
  elif tempo_de_servico >= 30:
    return REQUERER
  elif idade >= 60 and tempo_de_servico >= 25:
    return REQUERER

  return NAO_REQUERER
```

Utilize os valores abaixo como parâmetros de entrada e saída da função e corrija a função caso seja necessário:

`verificar_qualificacao_empregado(idade, tempo_de_servico)`	
Entrada	**Saída**
`idade = 0` `tempo_de_servico = 10`	`exceção: ValueError`
`idade = 20` `tempo_de_servico = 0`	`exceção: ValueError`
`idade = '65'` `tempo_de_servico = 30`	`exceção: TypeError`
`idade = 65` `tempo_de_servico = '30'`	`exceção: TypeError`
`idade = 65` `tempo_de_servico = 20`	`'Requerer aposentadoria'`
`idade = 59` `tempo_de_servico = 30`	`'Requerer aposentadoria'`
`idade = 60` `tempo_de_servico = 25`	`'Requerer aposentadoria'`
`idade = 59` `tempo_de_servico = 24`	`'Não requerer'`

Referências

PYTHON SOFTWARE FOUNDATION (org.). **Documentação python 3.10.4**. 2001. Disponível em: <https://docs.python.org/pt-br/3/library/unittest.html>. Acesso em: 01 jun. 2022.

SALE, D. **Testing python:** applying unit testing, TDD, BDD, and accepting testing. Wiley, 2014.