

# Engenharia de Software II

## Teste de Software

Profa. Thaiana Pereira dos Anjos Reis, Dra. Eng.  
[thaiana.anjos@ifsc.edu.br](mailto:thaiana.anjos@ifsc.edu.br)

# Agenda

- Instalação Python e Pytest
- Execução do Pytest no Prompt de Comando
- Execução do Pytest no Pycharm
- Exercícios
- Projeto de Teste



Pytest é um framework que facilita o desenvolvimento e execução de testes unitários em Python.

Ele é convenientemente integrado ao Pycharm e outras IDEs.



pytest

PHPUnit

 Jasmine

 Rspec

 + JUnit 

# Passo a passo



## 1) Instalação do Python

Verifique se você tem o Python instalado no seu computador.

Abra o CMD como Administrador e digite python.

Caso não esteja instalado, acesse o link abaixo e faça o download.

<https://www.python.org/downloads/>

# Passo a passo

Execute o CMD como **Administrador**.



# Passo a passo

Digite **python** para confirmar que ele está instalado.

```
C:\Users\Thaiana\Desktop>python
Python 3.10.8 (tags/v3.10.8:aaaf517, Oct 11 2022, 16:50:30
) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more
information.
>>> _
```

# Passo a passo

## 2) Instalação do PIP

Depois de instalar o Python, você precisa instalar o Pip.

**Pip é um gerenciador de pacotes para projetos em Python**, utilizado para instalar, remover e atualizar pacotes nos projetos.

Para instalar o Pip, siga as instruções:

- baixe o arquivo **get-pip.py** em uma pasta no seu computador;
- abra o Prompt de Comando e navegue para a pasta que possui o arquivo baixado.
- digite o comando **python get-pip.py**



# Passo a passo



## Tela de Sucesso da instalação do PIP

cmd Prompt de Comando

```
Microsoft Windows [versão 10.0.19044.2006]
(c) Microsoft Corporation. Todos os direitos reservados.

C:\Users\Thaiana>python get-pip.py
python: can't open file 'C:\\Users\\Thaiana\\get-pip.py': [Errno 2] No such file or directory

C:\Users\Thaiana>cd desktop

C:\Users\Thaiana\Desktop>python get-pip.py
Collecting pip
  Using cached pip-22.2.2-py3-none-any.whl (2.0 MB)
Collecting wheel
  Downloading wheel-0.37.1-py2.py3-none-any.whl (35 kB)
Installing collected packages: wheel, pip
  WARNING: The script wheel.exe is installed in 'C:\Users\Thaiana\AppData\Local\Programs\Python\Python310\Scripts' which
is not on PATH.
  Consider adding this directory to PATH or, if you prefer to suppress this warning, use --no-warn-script-location.
  Attempting uninstall: pip
  Found existing installation: pip 22.2.2
  Uninstalling pip-22.2.2:
    Successfully uninstalled pip-22.2.2
  WARNING: The scripts pip.exe, pip3.10.exe and pip3.exe are installed in 'C:\Users\Thaiana\AppData\Local\Programs\Pytho
n\Python310\Scripts' which is not on PATH.
  Consider adding this directory to PATH or, if you prefer to suppress this warning, use --no-warn-script-location.
Successfully installed pip-22.2.2 wheel-0.37.1
```

# Passo a passo

## 3) Instalação do Pytest

Depois de instalar o Pip, você precisa instalar o Pytest.

Para instalar o Pytest, você deve digitar a seguinte instrução no Prompt de Comando:

**pip install pytest**

```
C:\Users\Thaiana\Desktop>pip install pytest
Collecting pytest
  Downloading pytest-7.1.3-py3-none-any.whl (298 kB)
----- 298.2/298.2 kB 3.1 MB/s eta 0:00:00
Collecting colorama
  Using cached colorama-0.4.5-py2.py3-none-any.whl (16 kB)
Collecting iniconfig
  Using cached iniconfig-1.1.1-py2.py3-none-any.whl (5.0 kB)
Collecting packaging
  Using cached packaging-21.3-py3-none-any.whl (40 kB)
Collecting py>=1.8.2
  Using cached py-1.11.0-py2.py3-none-any.whl (98 kB)
Collecting tomli>=1.0.0
  Using cached tomli-2.0.1-py3-none-any.whl (12 kB)
Collecting attrs>=19.2.0
  Using cached attrs-22.1.0-py2.py3-none-any.whl (58 kB)
Collecting pluggy<2.0,>=0.12
  Using cached pluggy-1.0.0-py2.py3-none-any.whl (13 kB)
Collecting pyparsing!=3.0.5,>=2.0.2
  Using cached pyparsing-3.0.9-py3-none-any.whl (98 kB)
Installing collected packages: iniconfig, tomli, pyparsing, py, pluggy, colorama, attrs, packaging, pytest
Successfully installed attrs-22.1.0 colorama-0.4.5 iniconfig-1.1.1 packaging-21.3 p
luggy-1.0.0 py-1.11.0 pyparsing-3.0.9 pytest-7.1.3 tomli-2.0.1
```



**O ambiente de  
teste foi instalado  
com sucesso!**

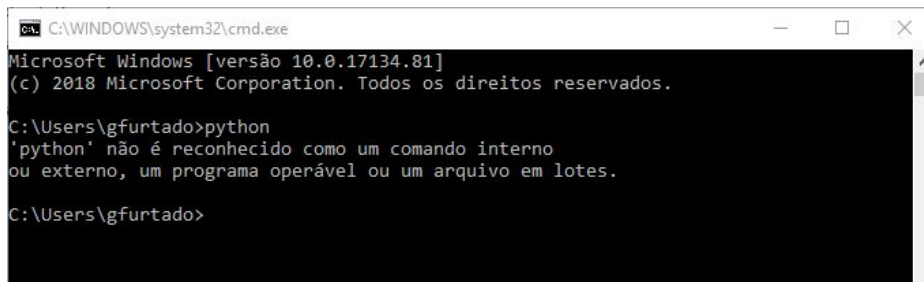


# PROBLEMAS NA INSTALAÇÃO

# PROBLEMAS NA INSTALAÇÃO

## 1) 'python' não é reconhecido como um comando interno

Pode acontecer de depois que você instalar o python no seu computador, você tente executar o comando python na linha de comando e apareça o seguinte erro no Windows:



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [versão 10.0.17134.81]
(c) 2018 Microsoft Corporation. Todos os direitos reservados.

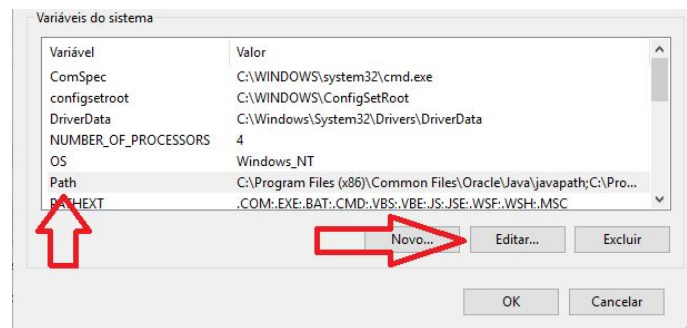
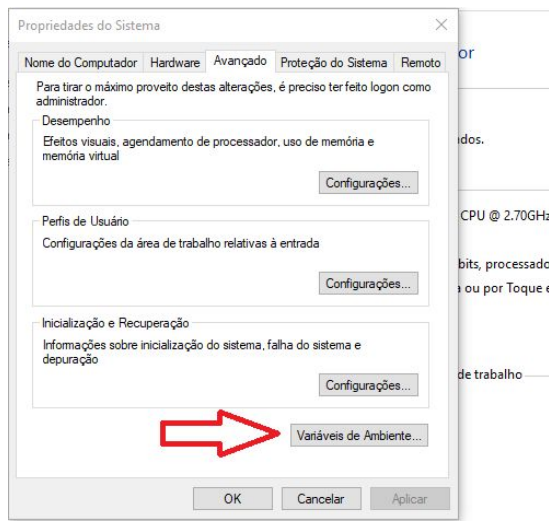
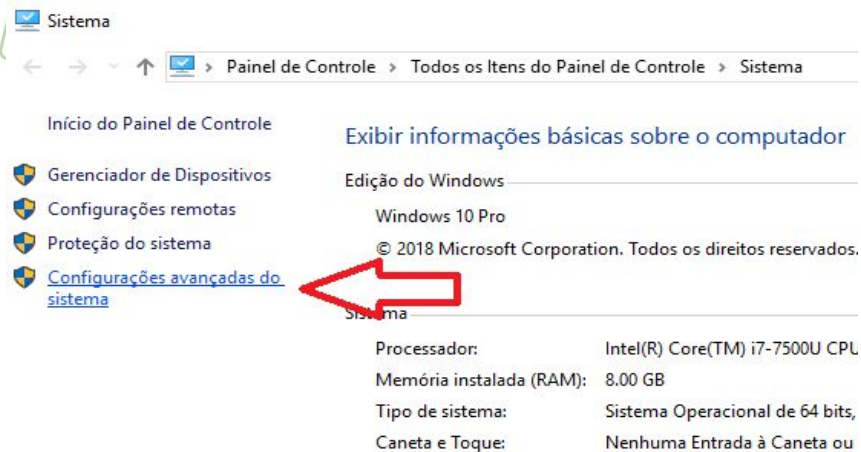
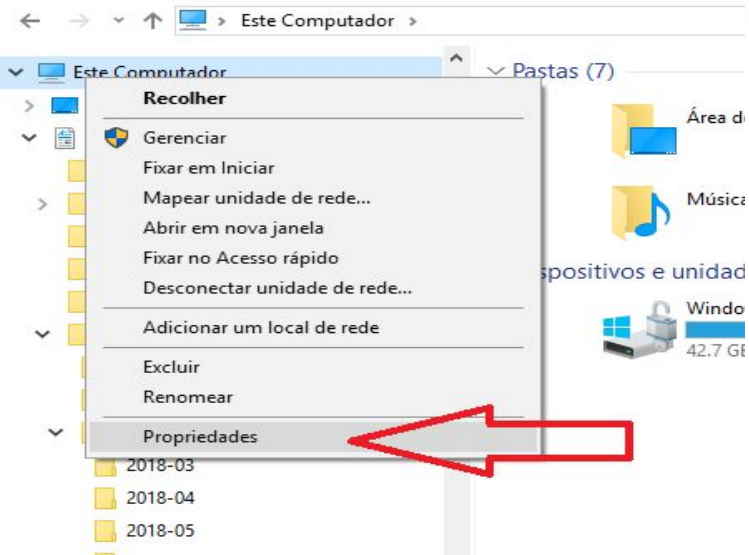
C:\Users\gfurtado>python
'python' não é reconhecido como um comando interno
ou externo, um programa operável ou um arquivo em lotes.

C:\Users\gfurtado>
```

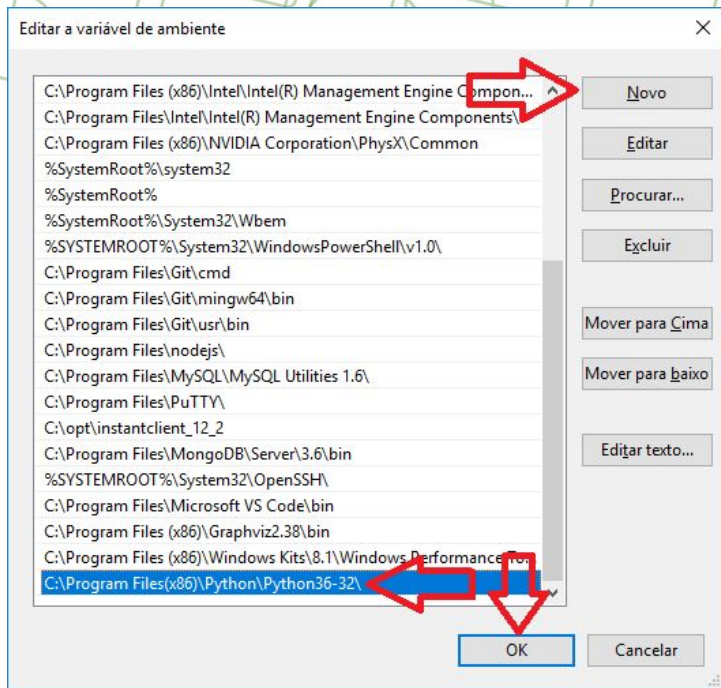
# PROBLEMAS NA INSTALAÇÃO

## 1) 'python' não é reconhecido como um comando interno

Solução: você colocar o caminho da instalação do python na **variável de ambiente PATH** e reiniciar o prompt de comando.







**Importante:** Pode ser que na sua versão do Windows esta janela seja diferente, apresentando apenas um campo de texto com vários caminhos de pastas separadas por **ponto-e-vírgula (;)**. Neste caso, o que você precisa fazer é justamente **adicionar outro ponto-e-vírgula (;) e o caminho da nova pasta** que você deseja incluir no PATH.

Próximo passo: **REINICIE o prompt de comando!**

Você precisa reiniciar o prompt de comandos para que as alterações de variáveis de ambiente sejam aplicadas.



# PROBLEMAS NA INSTALAÇÃO

## 2) 'pip' não é reconhecido como um comando interno

Pode acontecer de depois que você instalar o pip no seu computador, você tente executar o comando pip na linha de comando e apareça o mesmo erro do problema anterior, porém agora a respeito do pip.

Solução: você colocar o caminho da instalação do pip na **variável de ambiente PATH** e reiniciar o prompt de comando.



**O ambiente de  
teste foi instalado  
com sucesso!**

**Pronto!**  
**Agora podemos escrever o nosso primeiro teste!**

# Passo a passo

## 1) Criação do arquivo de teste

O Pytest utiliza um padrão para os testes.

O nome do arquivo deve iniciar com **test\_** ou terminar com **\_test.py**.

É assim que ele descobre os arquivos de teste.

# Passo a passo

## Exemplo:

Vamos criar um arquivo chamado **test\_iniciante.py**, que possui uma função **soma**.

Comece importando o pytest.

A função **soma** recebe dois números como parâmetros e retorna o valor da soma destes números.

# Passo a passo

## Exemplo:

Arquivo **test\_iniciante.py**

```
1 import pytest
2
3 # Função para testar
4 def soma(a, b):
5     return a + b
```

# Passo a passo

## Exemplo:

Crie a função **test\_soma**, para garantir que a função soma funciona corretamente.

**Observação:** o nome da nossa função de teste possui o prefixo **test\_**, pois o Pytest espera que as funções de teste tenham esse prefixo.

# Passo a passo

## Exemplo:

Arquivo **test\_iniciante.py**

```
1  import pytest
2
3  # Função para testar
4  def soma(a, b):
5      return a + b
6
7  def test_soma():
8      assert soma(1, 2) == 3
```

# Passo a passo



## Exemplo:

No prompt execute o comando **pytest test\_iniciante.py**

Teste realizado com sucesso!

```
C:\Users\Thaiana\PycharmProjects\test>pytest test_iniciante.py
===== test session starts =====
platform win32 -- Python 3.10.8, pytest-7.1.3, pluggy-1.0.0
rootdir: C:\Users\Thaiana\PycharmProjects\test
collected 1 item

test_iniciante.py . [100%]

===== 1 passed in 0.05s =====
```



# Passo a passo



## Exemplo:

Se você alterar o valor do assert para 6, o teste falhará.

```
C:\Users\Thaiana\PycharmProjects\test>pytest test_iniciante.py
===== test session starts =====
platform win32 -- Python 3.10.8, pytest-7.1.3, pluggy-1.0.0
rootdir: C:\Users\Thaiana\PycharmProjects\test
collected 1 item

test_iniciante.py F [100%]

===== FAILURES =====
_____ test_soma _____

    def test_soma():
>     assert soma(1, 2) == 6
E       assert 3 == 6
E       + where 3 = soma(1, 2)

test_iniciante.py:8: AssertionError
===== short test summary info =====
FAILED test_iniciante.py::test_soma - assert 3 == 6
===== 1 failed in 0.09s =====
```

Teste falho!



# Anotações e Asserções

# Frameworks de testes



**Anotações:** indicam a funcionalidade do método para o framework.



**Asserções:** são métodos de ensaio que possibilitam a validação de informações.

# Frameworks de testes



**Anotações:** indicam a funcionalidade do método para o framework.

Essas anotações indicam se um método é de teste ou não, se um método deve ser executado antes da classe e/ou depois da classe, indicam também se o teste deve ou não ser ignorado e se a classe em questão é uma suíte de teste, ou seja, se a partir desta classe é disparada a execução das demais classes de teste, entre outras anotações menos utilizadas.

# Pytest

## Anotações:

- `@pytest.fixture`
- `@pytest.mark.parametrize`

Essas anotações permitem que você aumente a cobertura de código adicionando facilmente valores de entrada.

# JUnit

@Teste: Identifica que o método é um teste;

@Before: Indica que o método deve ser executado antes do teste, pode ser utilizado para preparar o ambiente de teste;

@After: Indica que o método deve ser executado depois de cada teste, pode ser utilizado para limpar o ambiente de teste;

@BeforeClass: Indica que o método será executado antes do início do ensaio;

@AfterClass: Indica que o método será executado ao finalizar o ensaio;

# Frameworks de testes



**Assertões:** são métodos de ensaio que possibilitam a validação de informações.

Métodos de assertões são utilizados para validar informações. Com base nas assertões teremos o resultado de nosso teste como falho ou OK.

# Pytest

A palavra **assert** é uma palavra reservada no Python que serve para validar condições de teste.

As asserções são usadas para **validar condições de teste e retornam True ou False.**

Se o assert de um método de teste falhar, a execução do mesmo para. O código restante do método não é executado e o Pytest irá continuar a execução com o próximo método de teste.



# Pytest

```
1 import pytest
2
3 # Função para testar
4 def soma(a, b):
5     return a + b
6
7 def test_soma():
8     assert soma(1, 2) == 3
```

# JUnit

`fail(String)`: Deixa o método falhar, normalmente utilizado para verificar se uma determinada parte do código não está sendo atingido;

`assertTrue(true)/(false)`: Será sempre verdadeiro ou falso, pode ser utilizado para pré-definir um resultado de um teste, se o teste ainda não foi implementado;

`assertEquals(esperado, real)`: Testa dois valores verificando se são iguais.

`assertEquals(esperado, tolerância, real)`: Teste de valores do tipo `double` ou `float`, a tolerância, neste caso, é o número de casas

# JUnit

`assertNull(objeto)`: Verifica se o objeto é nulo;

`assertNotNull(objeto)`: Verifica se o objeto não é nulo;

`assertSame(String, esperado, real)`: Verifica se as duas variáveis se referem ao mesmo objeto;

`assertNotSame(String, esperado, real)`: Verifica se as duas variáveis se refere a objetos diferentes;

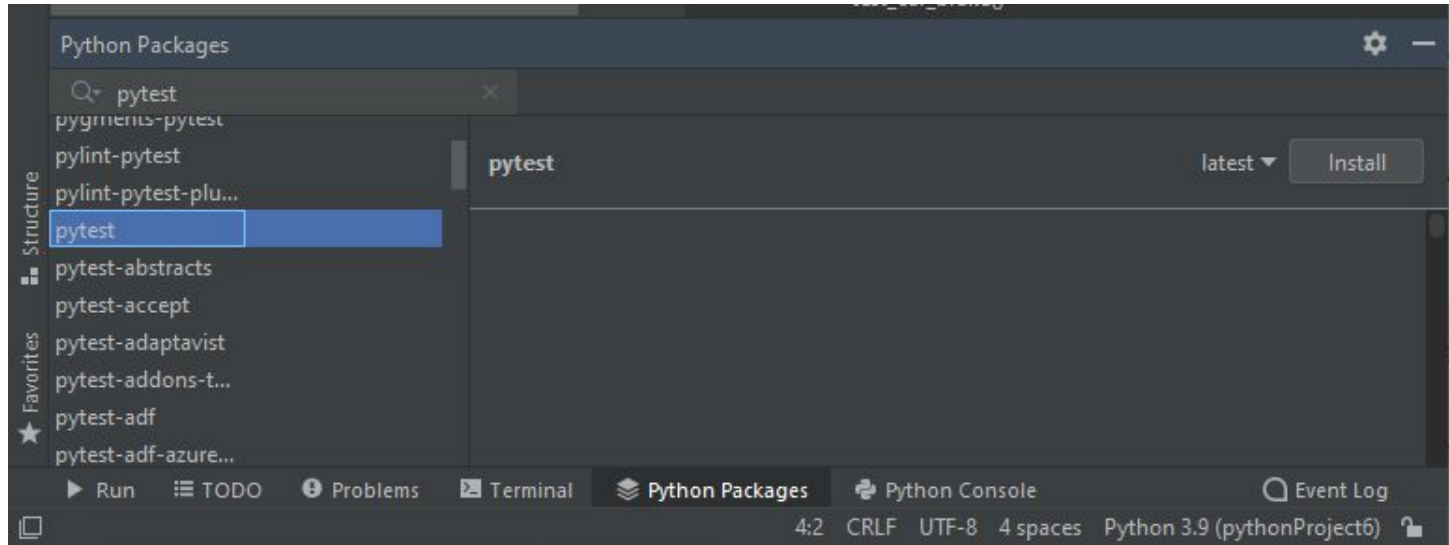


# Como usar o Pytest no Pycharm

# Passo a passo

## 1) Instalar o Pytest

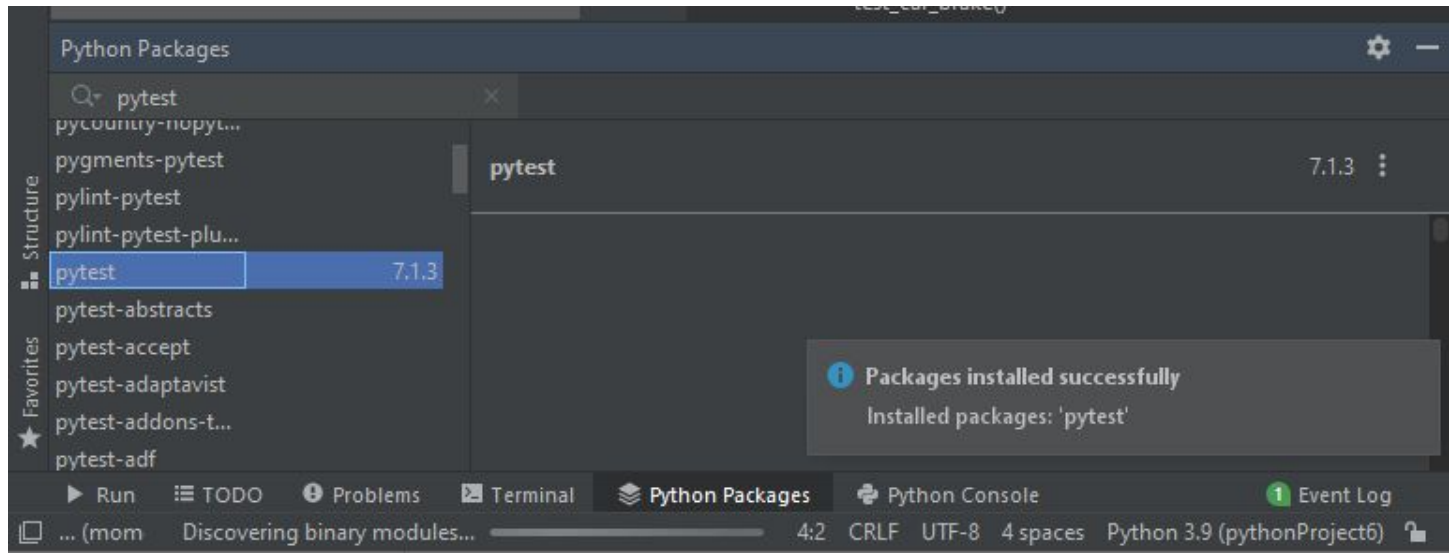
Na aba **Python Packages**, digite **pytest** no campo de pesquisa e clique no botão **Install**.



# Passo a passo

## 1) Instalar o Pytest

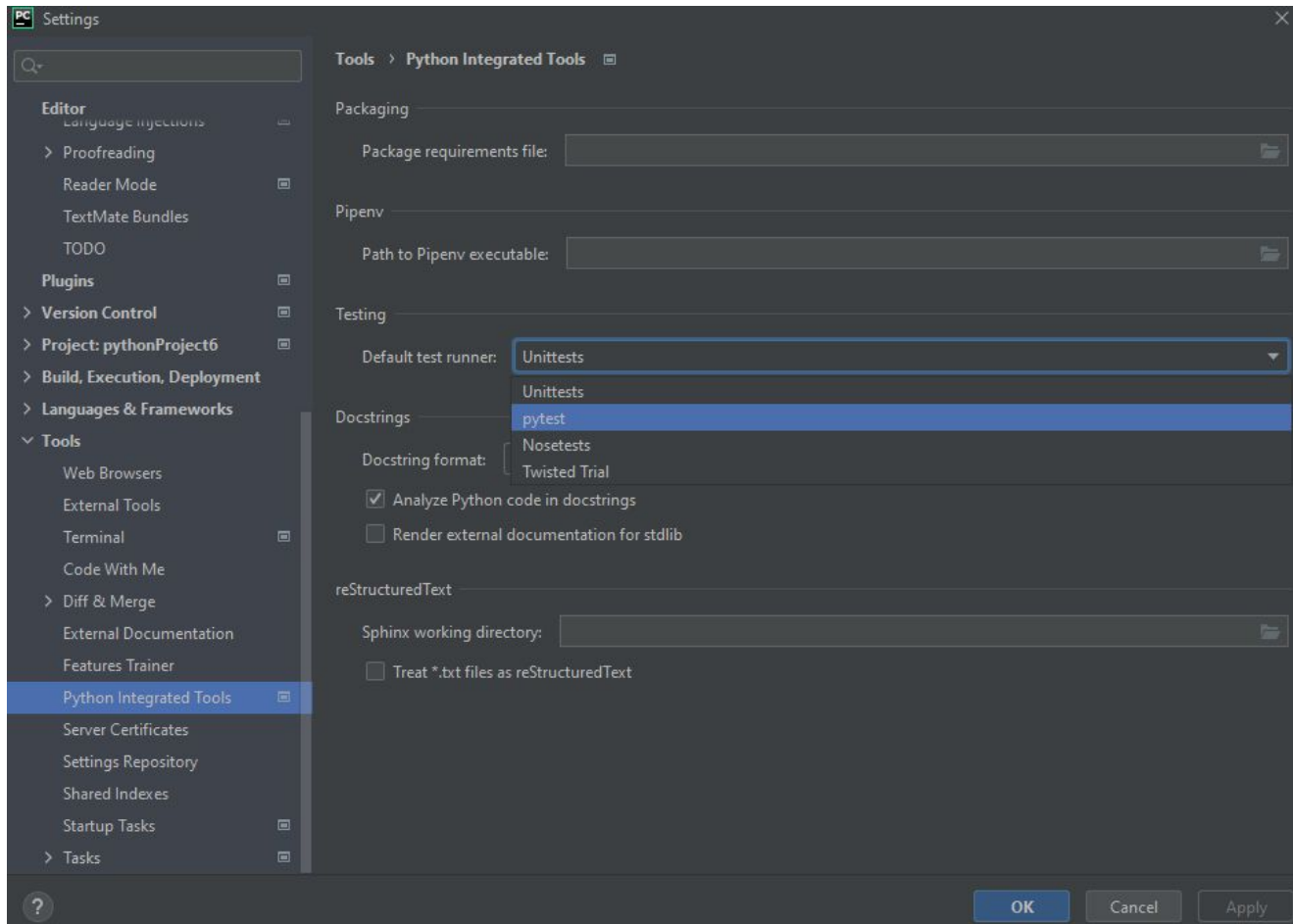
A mensagem “**Packages installed successfully**” será mostrada para informar que o Pytest foi instalado com sucesso.



# Passo a passo

## 2) Definir o Pytest como o framework padrão de teste

Para garantir que todos os recursos específicos do Pytest estejam disponíveis, defina o executor de teste manualmente. Pressione as teclas Ctrl+Alt+S para abrir as configurações da IDE e selecione **Tools | Python Integrated Tools**, e selecione **pytest** na lista do campo **Default test runner**.







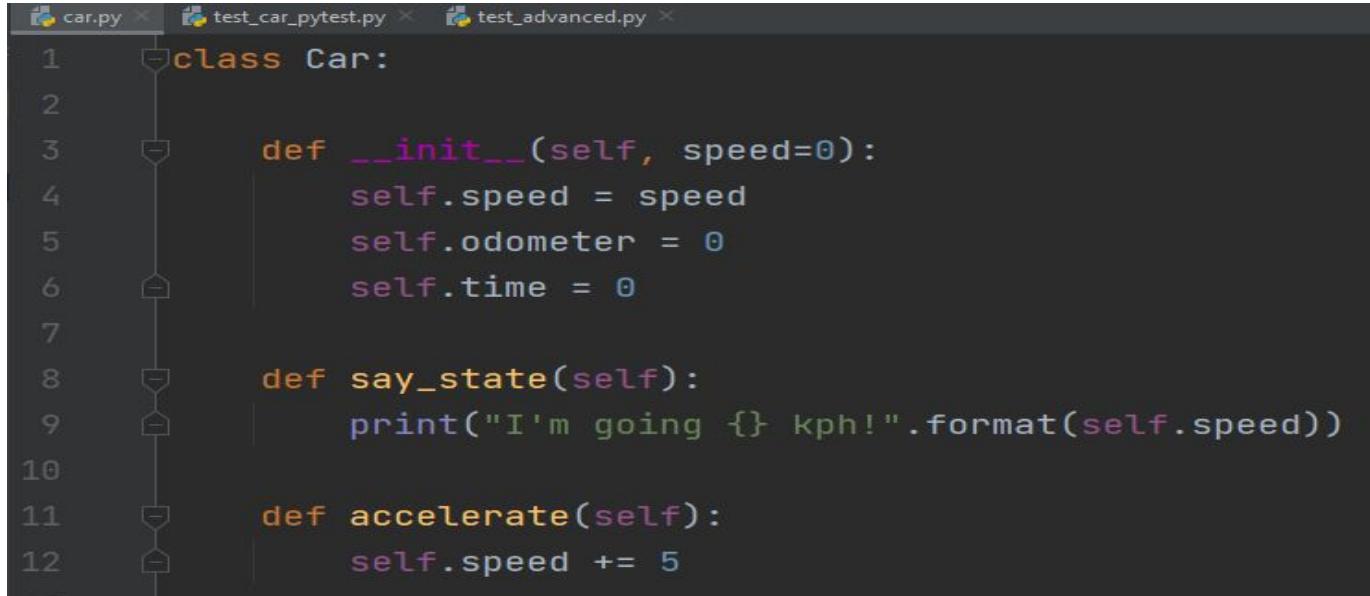
# Exemplo 1

## Carro

# Passo a passo

## 3) Criar uma classe

Crie o arquivo **car.py** com a classe Car

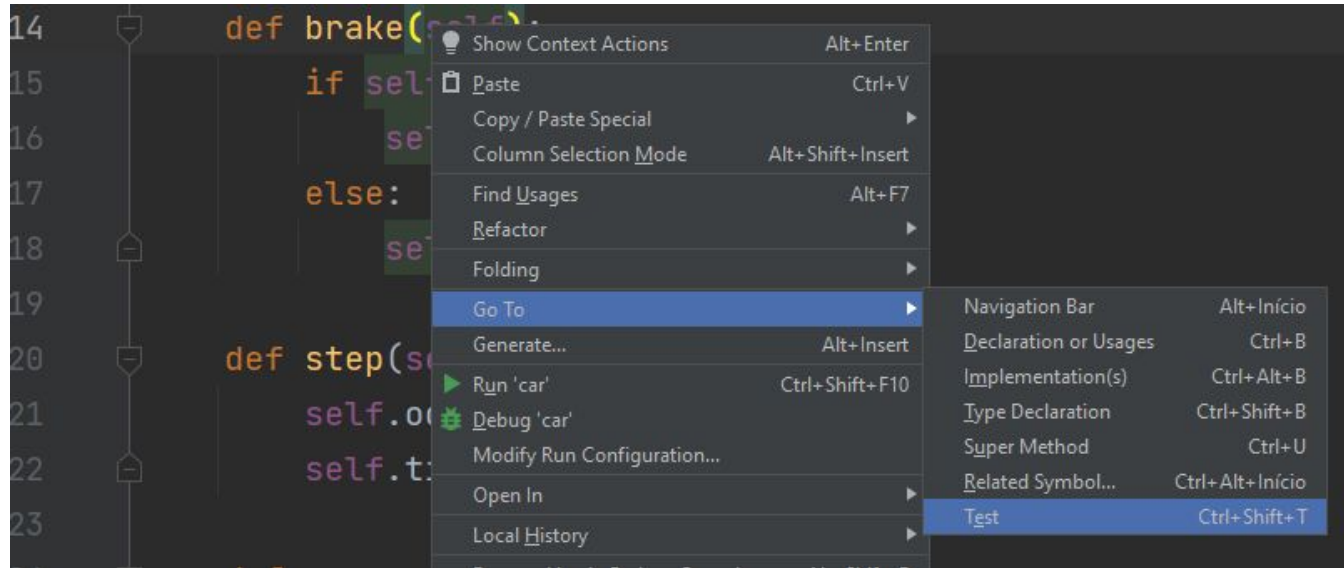


```
car.py x test_car_pytest.py x test_advanced.py x
1 class Car:
2
3     def __init__(self, speed=0):
4         self.speed = speed
5         self.odometer = 0
6         self.time = 0
7
8     def say_state(self):
9         print("I'm going {} kph!".format(self.speed))
10
11     def accelerate(self):
12         self.speed += 5
```

# Passo a passo

## 4) Criar um teste

Para criar um arquivo de teste para testar o método `brake()`, clique no botão direito no método e selecione **Go To | Test**

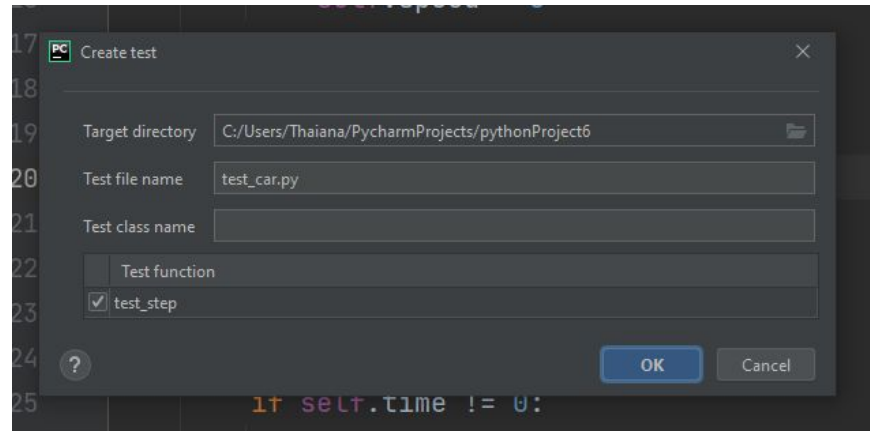


# Passo a passo

## 4) Criar um teste

Selecione a opção **Create New Test** e informe o nome do arquivo de teste no campo **Test File Name**.

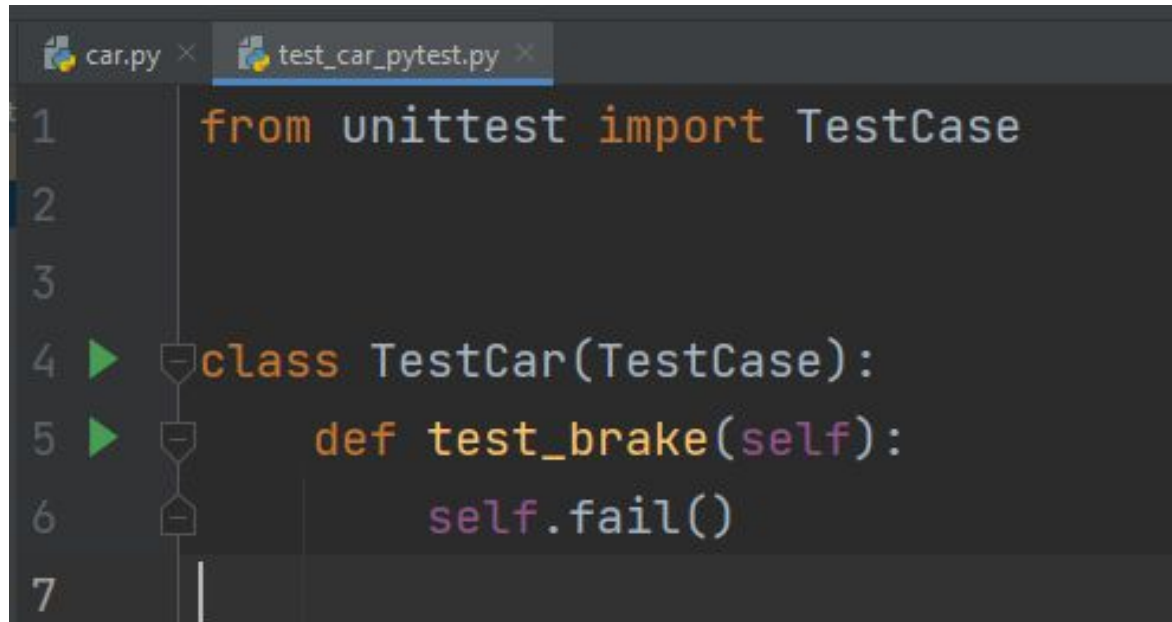
Digite **test\_car\_pytest.py**



# Passo a passo

## 5) Alterar arquivo de teste

O arquivo **test\_car\_pytest.py** é criado.

A screenshot of a code editor with two tabs: 'car.py' and 'test\_car\_pytest.py'. The 'test\_car\_pytest.py' tab is active and shows the following Python code:

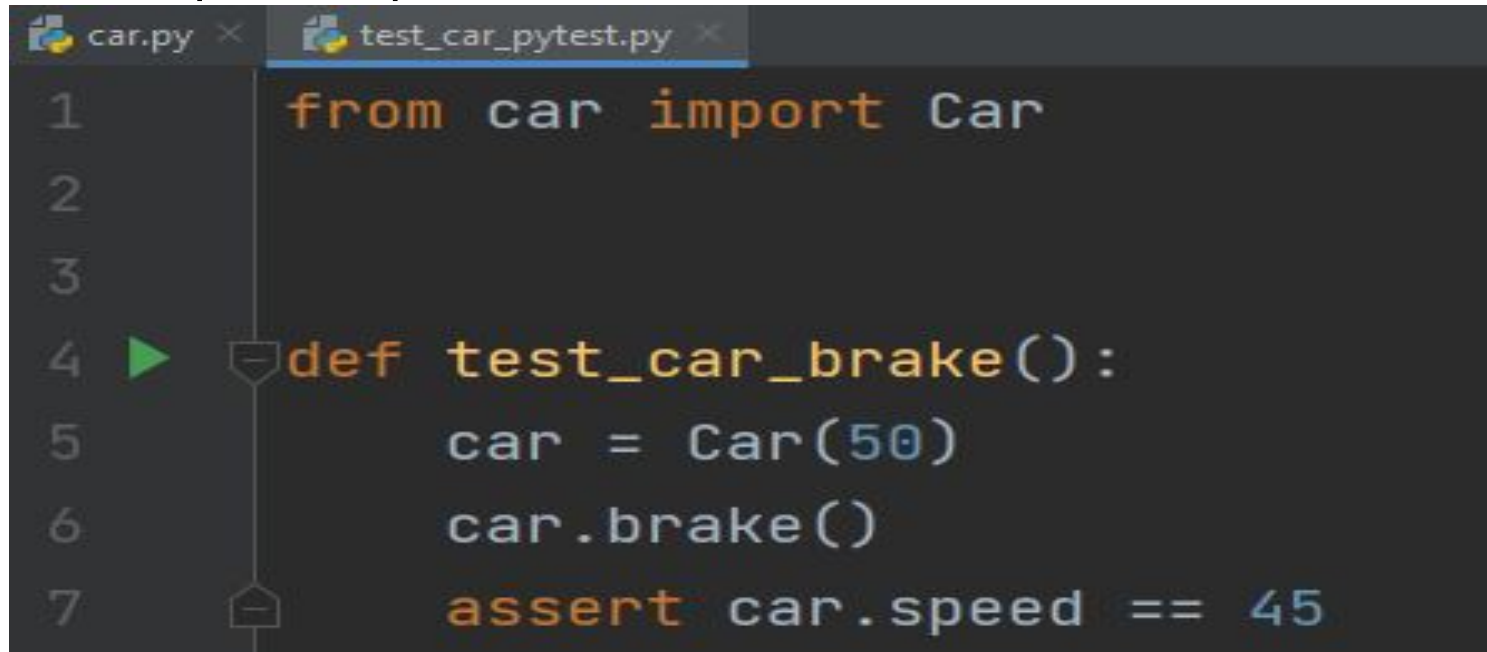
```
1  from unittest import TestCase
2
3
4  class TestCar(TestCase):
5      def test_brake(self):
6          self.fail()
7
```

Line numbers 1 through 7 are visible on the left. The code defines a class 'TestCar' that inherits from 'TestCase' and contains a method 'test\_brake' which calls 'self.fail()'. There are green play icons and minus signs in the left margin next to lines 4 and 5.

# Passo a passo

## 5) Alterar arquivo de teste

Modifique o arquivo informando o teste a ser realizado.

A screenshot of a code editor with two tabs: 'car.py' and 'test\_car\_pytest.py'. The 'test\_car\_pytest.py' tab is active. The code in the editor is as follows:

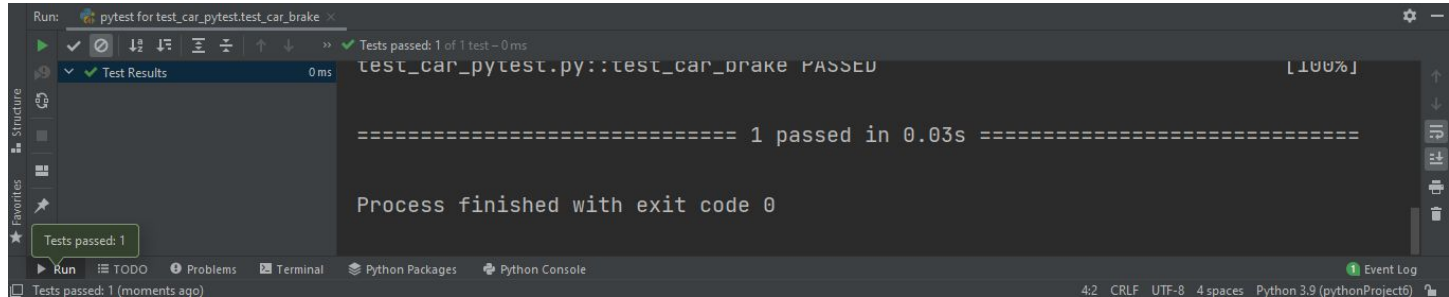
```
1 from car import Car
2
3
4 def test_car_brake():
5     car = Car(50)
6     car.brake()
7     assert car.speed == 45
```

The code is written in a dark-themed editor. Line numbers 1 through 7 are visible on the left. A green play button icon is next to line 4. A vertical line with a diamond-shaped breakpoint icon is positioned between lines 4 and 5.

# Passo a passo

## 6) Executar o teste

Rode o teste e verifique o resultado.



The screenshot shows an IDE interface with a dark theme. The top toolbar indicates a successful run: a green play button, a checkmark, and the text "Tests passed: 1 of 1 test - 0 ms". Below this, a "Test Results" panel on the left shows a green checkmark and "0 ms". The main editor area displays the output of the test run:

```
test_car_pytest.py::test_car_brake PASSED [100%]  
  
===== 1 passed in 0.03s =====  
  
Process finished with exit code 0
```

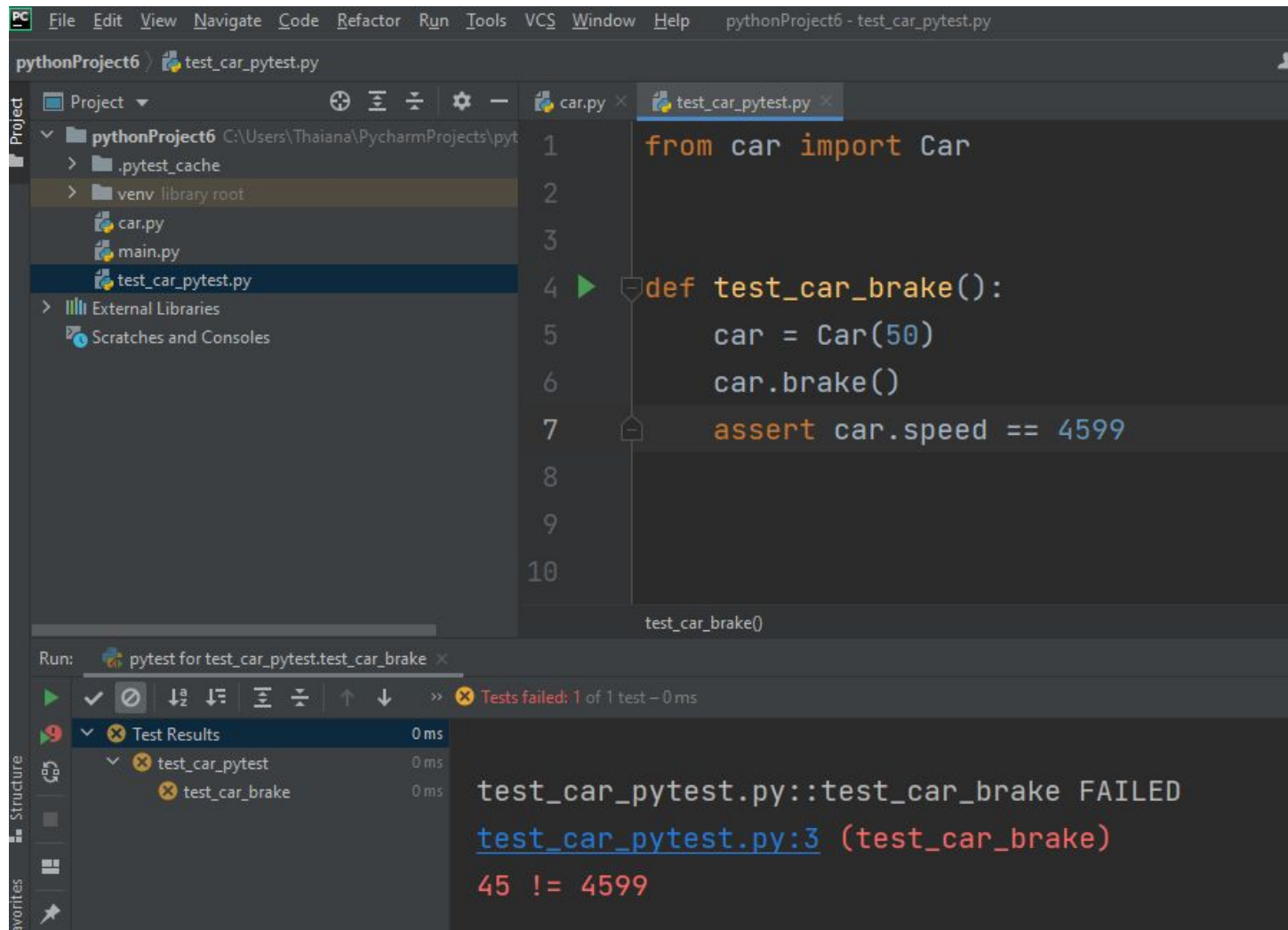
The bottom status bar shows "Tests passed: 1 (moments ago)" on the left and "4:2 CRLF UTF-8 4 spaces Python 3.9 (pythonProject6)" on the right. The left sidebar contains icons for Structure, Favorites, and a star icon.

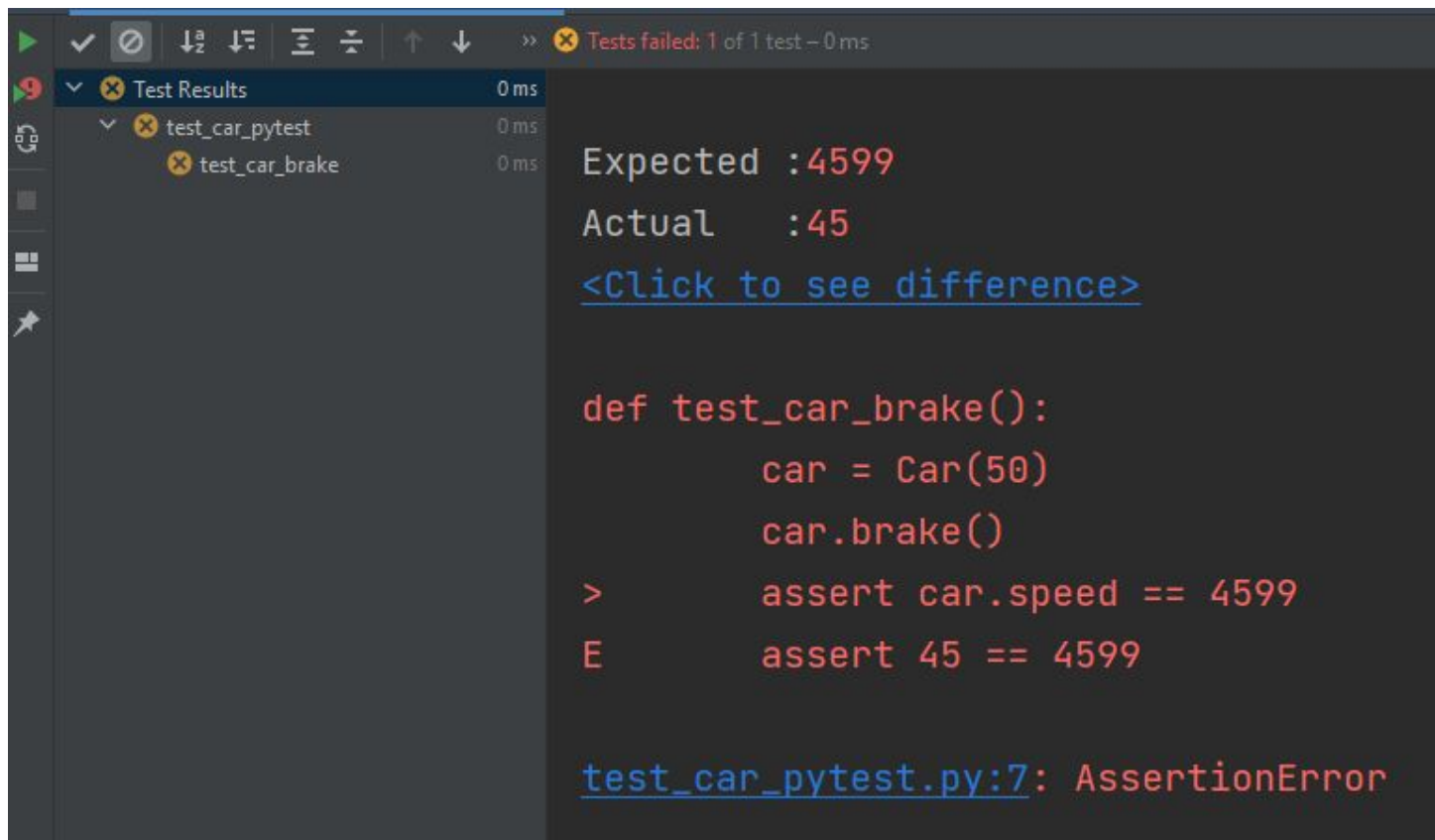
# Passo a passo

## **7) Faça o teste falhar**

Altere o teste (informe um valor para que o teste falhe). Rode o teste novamente e verifique o resultado.







# Passo a passo

## 8) Utilizar a anotação `@pytest.fixture`

É possível criar pequenas unidades de teste que podem ser reutilizadas no módulo de teste. Para isso é necessário marcar uma unidade reutilizável com `@pytest.fixture`.

Vamos criar o método `my_car()`, o qual cria uma instância `Car` com o valor de velocidade igual a 50.

O método `my_car()` é usado em `test_car_accelerate()` e `test_car_brake()` para verificar a execução correta das funções correspondentes na classe `Car`.

```
1  import pytest
2  from car import Car
3
4  @pytest.fixture
5  def my_car():
6      return Car(50)
7
8  ▶ def test_car_accelerate(my_car):
9      my_car.accelerate()
10     assert my_car.speed == 55
11
12  ▶ def test_car_brake(my_car):
13     my_car.brake()
14     assert my_car.speed == 45
```

# Passo a passo

## 9) Executar o teste

Rode o teste e verifique o resultado.

```
===== test session starts =====  
collecting ... collected 2 items  
  
test_car_pytest.py::test_car_accelerate PASSED [ 50%]  
test_car_pytest.py::test_car_brake PASSED [100%]  
  
===== 2 passed in 0.02s =====  
  
Process finished with exit code 0
```

# Parametrização

A parametrização de testes pode reduzir a repetição de código enquanto permite uma cobertura de testes maior e melhor.

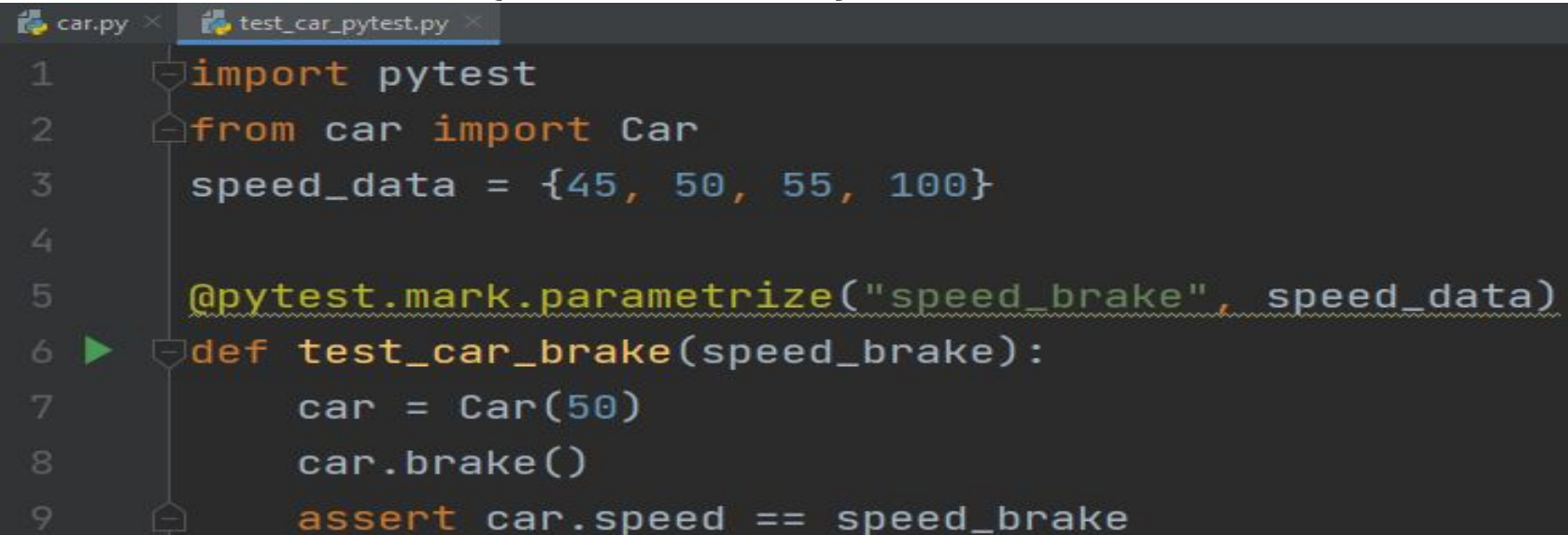
A parametrização permite que você aumente a cobertura de código adicionando facilmente **mais valores de entrada.**

# Passo a passo

## 10) Utilizar a anotação `@pytest.mark.parametrize`

Crie um conjunto de valores de velocidade para testar as funções `car.accelerate()` e `car.brake()`.

`speed_data = {45, 50, 55, 100}`

A screenshot of a code editor with two tabs: 'car.py' and 'test\_car\_pytest.py'. The 'test\_car\_pytest.py' tab is active, showing a Python script with 9 lines of code. The code imports 'pytest' and 'Car' from 'car', defines a set 'speed\_data' with values {45, 50, 55, 100}, and uses the '@pytest.mark.parametrize' decorator to parameterize a test function 'test\_car\_brake'. The test function creates a 'Car' object with speed 50, calls 'brake()', and asserts that the speed is equal to the 'speed\_brake' parameter.

```
1 import pytest
2 from car import Car
3 speed_data = {45, 50, 55, 100}
4
5 @pytest.mark.parametrize("speed_brake", speed_data)
6 def test_car_brake(speed_brake):
7     car = Car(50)
8     car.brake()
9     assert car.speed == speed_brake
```

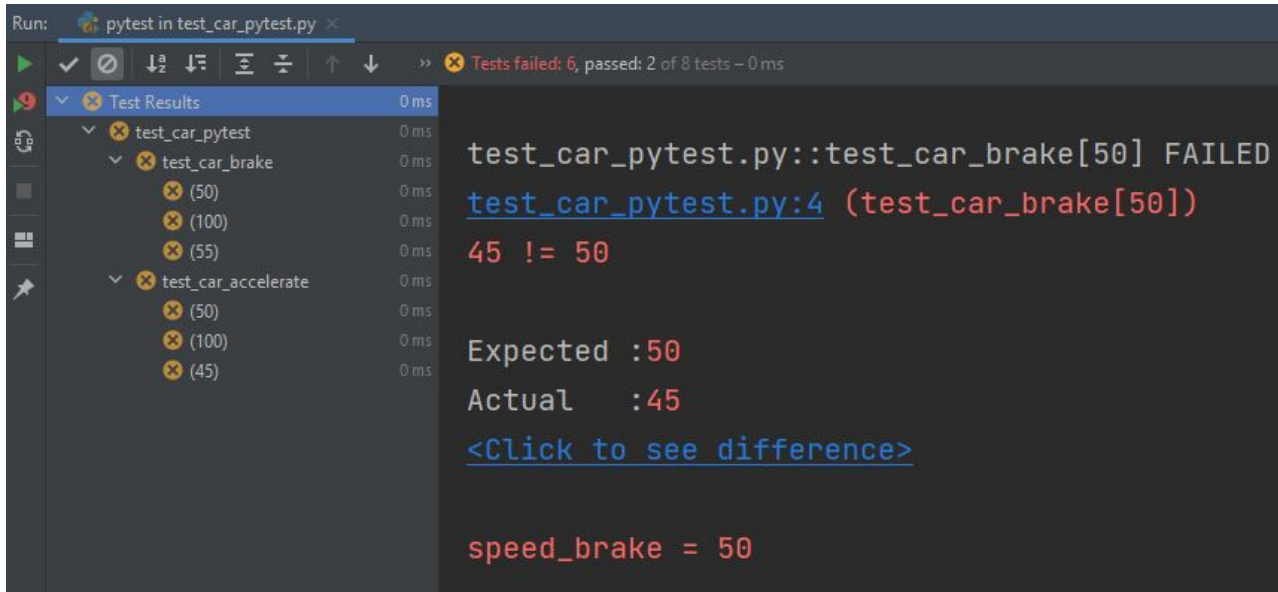
```
car.py x test_car_pytest.py x
1  import pytest
2  from car import Car
3      speed_data = {45, 50, 55, 100}
4
5      @pytest.mark.parametrize("speed_brake", speed_data)
6  ▶ def test_car_brake(speed_brake):
7      car = Car(50)
8      car.brake()
9      assert car.speed == speed_brake
10
11     @pytest.mark.parametrize("speed_accelerate", speed_data)
12  ▶ def test_car_accelerate(speed_accelerate):
13      car = Car(50)
14      car.accelerate()
15      assert car.speed == speed_accelerate
16
```



# Passo a passo

## 11) Executar o teste

Rode o teste e verifique o resultado.



Run: pytest in test\_car\_pytest.py

Test Results 0 ms

- test\_car\_pytest 0 ms
  - test\_car\_brake 0 ms
    - (50) 0 ms
    - (100) 0 ms
    - (55) 0 ms
  - test\_car\_accelerate 0 ms
    - (50) 0 ms
    - (100) 0 ms
    - (45) 0 ms

Tests failed: 6, passed: 2 of 8 tests - 0 ms

```
test_car_pytest.py::test_car_brake[50] FAILED
test_car_pytest.py:4 (test_car_brake[50])
45 != 50

Expected :50
Actual   :45
<Click to see difference>

speed_brake = 50
```



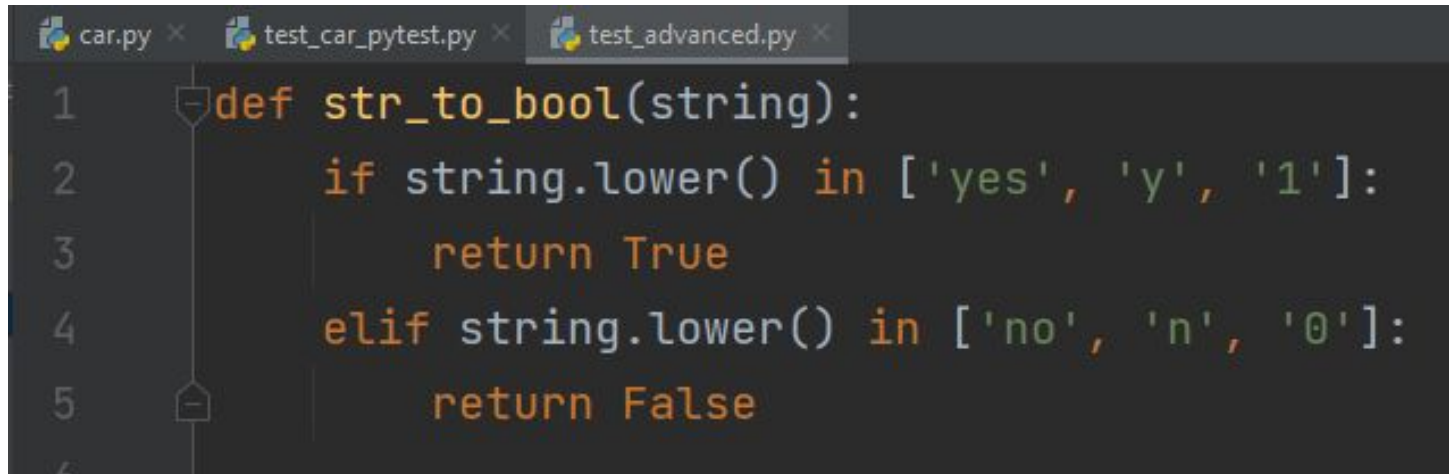
## Exemplo 2

# Conversão de string para TRUE ou FALSE

# Passo a passo

## 1) Criar uma classe

Crie o arquivo **test\_advanced.py** com um método que retorna True ou False, dependendo dos dados de entrada.



```
car.py × test_car_pytest.py × test_advanced.py ×  
1 def str_to_bool(string):  
2     if string.lower() in ['yes', 'y', '1']:  
3         return True  
4     elif string.lower() in ['no', 'n', '0']:  
5         return False  
6
```

# Passo a passo

## 2) Utilizar a anotação `@pytest.mark.parametrize`

Crie dois métodos para testar um conjunto de valores de strings, que retornem True ou False.

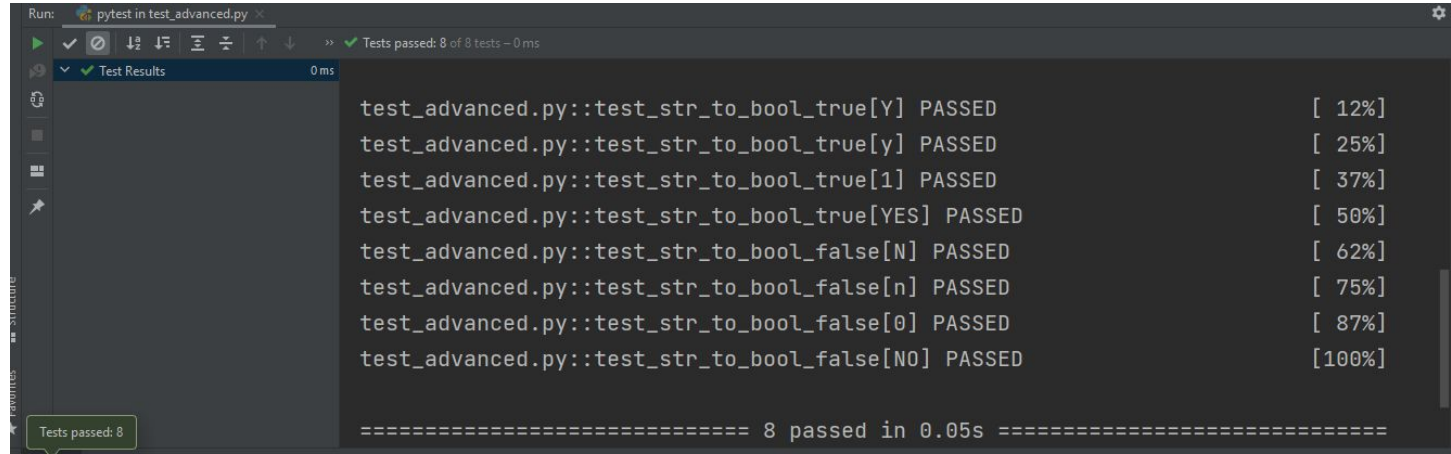
```
8  import pytest
9
10 @pytest.mark.parametrize("string", ['Y', 'y', '1', 'YES'])
11 def test_str_to_bool_true(string):
12     assert str_to_bool(string) is True
13
14 @pytest.mark.parametrize("string", ['N', 'n', '0', 'NO'])
15 def test_str_to_bool_false(string):
16     assert str_to_bool(string) is False
17
```

# Passo a passo

## 3) Executar o teste

Rode o teste e verifique o resultado.

Embora você tenha escrito apenas duas funções de teste, o Pytest foi capaz de criar oito testes no total graças à função `parametrize()`.



```
Run: pytest in test_advanced.py x
>> Tests passed: 8 of 8 tests - 0 ms

Test Results 0 ms

test_advanced.py::test_str_to_bool_true[Y] PASSED [ 12%]
test_advanced.py::test_str_to_bool_true[y] PASSED [ 25%]
test_advanced.py::test_str_to_bool_true[1] PASSED [ 37%]
test_advanced.py::test_str_to_bool_true[YES] PASSED [ 50%]
test_advanced.py::test_str_to_bool_false[N] PASSED [ 62%]
test_advanced.py::test_str_to_bool_false[n] PASSED [ 75%]
test_advanced.py::test_str_to_bool_false[0] PASSED [ 87%]
test_advanced.py::test_str_to_bool_false[NO] PASSED [100%]

===== 8 passed in 0.05s =====
```

# Dúvidas?

Profa. Thaiana Pereira dos Anjos Reis, Dra. Eng.  
[thaiana.anjos@ifsc.edu.br](mailto:thaiana.anjos@ifsc.edu.br)

