

9

# TEXTO BASE

PROGRAMAÇÃO ORIENTADA A OBJETOS

## Texto base

# 9

## Testes unitários e tratamento de erros

Prof. MSc. Rafael Maximo Carreira Ribeiro

### ***Resumo***

*Neste capítulo iremos aprender sobre dois conceitos muito importantes na programação de todo tipo de aplicação ou software: o tratamento de exceções e a criação e execução de testes unitários automatizados. Faremos também uma breve introdução a metodologia TDD, que é o Desenvolvimento Guiado por Testes, em português.*

### **9.1. Introdução**

Vimos em LP que linguagens de programação fazem parte do que chamamos de linguagens formais, pois possuem um conjunto definido de regras e não permitem nenhuma margem para ambiguidades. Portanto um computador irá executar exatamente aquilo que lhe é instruído, e pode nos dizer facilmente quando há uma instrução que não segue as regras da linguagem ou que tenta realizar uma operação não permitida, erros de sintaxe e execução, respectivamente. Já com relação a erros de lógica, somos nós os responsáveis por garantir que nosso código esteja fazendo a coisa certa.

Por exemplo, imagine que ao fazer um depósito na sua conta, o programa do banco tente adicionar um valor que não seja um número. Isso configuraria um erro de execução, pois não é possível somar um número a um não-número em Python, e o computador nos informaria desse erro.

Imagine agora outro erro, em que o programa subtraia o valor depositado do seu saldo ao invés de somá-lo. Isso caracteriza um erro de lógica, também chamado de erro de semântica, pois há uma divergência entre o que o código faz e o que esperávamos que fosse feito. Neste caso, computador não tem como saber que isso está errado, já que ele executa fielmente suas instruções, então se há um erro de lógica, somos nós os responsáveis por encontrá-lo e corrigi-lo.

## 9.2. Desenvolvimento Guiado por Testes (TDD)

O desenvolvimento guiado por testes é uma técnica para o desenvolvimento de software em que os requisitos são convertidos em testes antes de começarmos a programar o software propriamente dito. Ou seja, primeiro escrevemos os testes e só depois escrevemos o código que deverá cumprir as tarefas para as quais os testes foram escritos.

Essa metodologia foi apresentada por Kent Beck (2002) no começo dos anos 2000. Segundo o próprio autor (BECK, K. 2012), ele apenas redescobriu essa técnica, que já era usada por programadores desde a época dos cartões perfurados.

### 9.2.1. Ciclos do TDD

O TDD é constituído por ciclos de desenvolvimento e cada ciclo possui 3 fases: Red, Green, Refactor. A última fase é também conhecida como a fase azul (Blue). Na fase vermelha, escrevemos os testes para a nova funcionalidade (que não existe ainda), na fase verde escrevemos o código que implementa a funcionalidade sendo testada e na fase azul refatoramos o código, aprimorando a solução inicial da fase verde. Podemos descrever mais detalhadamente cada fase como:

- RED: Adicionamos um novo teste para a nova funcionalidade e rodamos todo o conjunto de testes. Nessa fase o teste deverá obrigatoriamente falhar, pois a funcionalidade ainda não foi implementada. Se o teste passar, há algo errado com ele e precisamos refatorá-lo até que ele falhe.
- GREEN: Implementamos a nova funcionalidade e executamos os testes até que o novo teste passe, sem quebrar nenhum outro teste. O objetivo aqui é escrever a solução mais simples possível que faça o teste passar, sem precisar se preocupar com padrões de projeto, com os princípios do SOLID, com deixar o código elegante ou com a eficiência do nosso algoritmo, tudo isso ficará para a etapa de refatoração do código.
- BLUE: Com o teste passando, sabemos duas coisas: 1) a nova funcionalidade está implementada; 2) não introduzimos nenhuma modificação que quebre outras funcionalidades. O objetivo nesta fase é melhorar desempenho, legibilidade e manutenabilidade do código, e com os testes, agora temos segurança para fazer melhorias na implementação, que podem envolver, entre outros:
  - renomear as variáveis e melhorar a legibilidade do código;
  - aplicar os princípios do SOLID que sejam relevantes;
  - reorganizar o código de acordo com a responsabilidade de cada classe;
  - aplicar um ou mais padrões de projeto que sejam pertinentes;

- remover código duplicado e valores “hard-coded”<sup>1</sup>;
- subdividir os métodos e funções se necessário;

### 9.2.2. Principais vantagens do TDD

O ciclo do TDD é então repetido para cada nova funcionalidade que precisamos implementar. No começo pode parecer contraintuitivo programar primeiro o teste, mas com o tempo irá se tornar natural e no médio e longo prazo, traz muitos benefícios, como por exemplo:

- Código mais simples e fácil de ler e dar manutenção;
- Código mais flexível e adaptável, com menos acoplamentos e interfaces mais claras e classes mais direcionadas. De certa forma, podemos dizer que a prática correta do TDD automaticamente leva à aplicação do princípio da responsabilidade única (o S do SOLID).
- Maior atendimento dos requisitos, uma vez que precisamos pensar nos testes antes de escrever o código, acabamos fazendo um exercício maior de interpretação dos requisitos e melhoramos o nosso entendimento sobre o comportamento da nova funcionalidade.
- Segurança ao desenvolver, pois sabemos que caso nossa implementação quebre alguma outra funcionalidade, seremos avisados pelos testes ainda durante o desenvolvimento e poderemos corrigir o problema antes de introduzir bugs silenciosamente na aplicação.

### 9.2.3. Limitações do TDD

Assim como qualquer outra metodologia de desenvolvimento, o TDD também apresenta algumas limitações, como por exemplo:

- O TDD, por usar principalmente testes unitários, não é capaz de testar completamente as partes da aplicação que exigem outros tipos de testes, como por exemplo conexões com um banco de dados, configurações de rede no ambiente, interfaces de usuário, etc. Nestes casos, devemos testar a parte lógica interna da nossa aplicação, usando um *mock*<sup>2</sup> para representar as partes externas às quais não temos controle.
- Para que as vantagens do TDD possam ser aproveitadas, toda a equipe envolvida no projeto precisa ter conhecimento da metodologia, caso contrário é possível que parte da equipe encare os testes como perda de tempo.

---

<sup>1</sup> Hard-coded, ou codificação rígida, é quando colocamos um valor fixo no próprio código fonte, por exemplo, quando colocamos a url para uma api como uma string no próprio código, ao invés de ler este valor de um arquivo de configurações ou variável de ambiente.

<sup>2</sup> *mock* é um objeto que simula o objeto real, por exemplo, uma conexão com um banco de dados. No objeto de mock, podemos definir o comportamento esperado daquela conexão, se vai dar erro ou não, qual o erro, qual o retorno, etc. E assim podemos testar a nossa lógica interna em cada uma dessas situações.

- Projetos com um grande número de testes podem passar uma falsa sensação de segurança, e isso pode levar a equipe a negligenciar outros tipos de testes.
- O conjunto de testes precisa de manutenção, assim como o restante do código, então testes mal escritos podem acabar aumentando o custo de manutenção da aplicação, pois será necessário mais tempo de desenvolvimento para corrigi-los. Um exemplo é o uso de testes com mensagens de erro escritas como *strings* literais no código, que poderão eventualmente falhar após uma atualização de algum módulo na qual as mensagens de erro sejam alteradas.
- O desenvolvedor irá escrever tanto os testes quanto o código, então caso haja um erro de interpretação dos requisitos do projeto, é provável que tanto o código quanto os testes compartilhem esse mesmo erro e isso passe sem ser percebido. Portanto é necessário estar atento a esse tipo de problema, sendo que ter o código/testes revisados por outros desenvolvedores ajuda a mitigá-lo.

### 9.3. Testes Unitários e Testes Automatizados

O exemplo dado na introdução é simples, e pode parecer besta, mas em situações reais, as aplicações e programas são mais complexos, com diversos módulos diferentes que interagem entre si de maneira muitas vezes não linear e interdependente. Sendo assim, nem sempre conseguimos garantir que uma alteração em um trecho de código, não irá afetar outras áreas da aplicação de maneira inesperada.

Para resolver isso, após alterarmos alguma coisa no código, podemos testar nossa aplicação para garantir que tudo esteja funcionando como antes, mas conforme a aplicação cresce, fazer tais testes manualmente demandará cada vez mais tempo e eventualmente se tornará inviável.

A resposta óbvia é, portanto, a utilização de testes automatizados. Existem diversos tipos de testes, mas agora vamos estudar os testes unitários, ou testes de unidade. Testes unitários servem para testar uma unidade isolada de código, para garantir que ela faz aquilo a que se propõe, isso pode ser o teste de uma função, de um método, da criação de um objeto com as propriedades corretas, etc.

Vamos criar uma função que recebe um valor inteiro e nos retorna esse valor mais 2. Para isso crie o arquivo “meu\_modulo.py” na pasta aula09 e digite o código da Codificação 9.1:

**Codificação 9.1: Exemplo de função que retorna número recebido mais dois**

```
def soma_2(numero):
    return numero * 2
```

**Fonte: do autor, 2021.**

Talvez você tenha percebido que há um erro na função. Esse erro é proposital e vamos supor que tenha passado despercebido por hora. Como sabemos o que a função

deve fazer, podemos pensar em alguns casos de teste para verificar seu funcionamento, como mostra a Tabela 9.1.

**Tabela 9.1: Casos de teste para a função soma\_2**

Caso de teste	Entrada	Saída esperada
#1	2	4
#2	5	7

**Fonte: do autor, 2021.**

Agora adicione o trecho de código da Codificação 9.2 ao arquivo anterior, para testar nossa função.

**Codificação 9.2: Código de teste inicial**

```
test1 = soma_2(2) == 4
test2 = soma_2(5) == 7
msg1 = 'sucesso' if test1 else 'falha'
msg2 = 'sucesso' if test2 else 'falha'
print(f'Resultado do teste 1: {msg1}')
print(f'Resultado do teste 2: {msg2}')
```

**Fonte: do autor, 2021.**

Neste código estamos guardando o resultado das verificações em uma flag booleana, uma variável que guarda um valor booleano (`True` ou `False`) referente ao status de algo em nosso código. Em seguida usamos o operador ternário<sup>3</sup> para decidir qual mensagem deve ser exibida na tela, e por fim exibimos as mensagens do resultado de cada teste. A Figura 9.1 mostra o resultado da execução deste código no terminal.

---

<sup>3</sup> Este operador foi introduzido com a PEP 0308, e funciona assim: na expressão `x if C else y`, primeiro a condição `C` é avaliada, se resultar em `True`, `x` é avaliado e o valor retornado; caso contrário, `y` será avaliado e retornado (PSF, 2021a).

**Figura 9.1: Execução da primeira versão dos testes automatizados**

```

File Edit Selection View Go Run Terminal Help meu_modulo.py - POO - Visual Studio Code
EXPLORER ... modo_debug.py dicionarios.py meu_modulo.py Settings ...
OPEN EDITORS aula09 > meu_modulo.py > ...
1 def soma_2(numero):
2     return 2 * numero
3
4
5 test1 = soma_2(2) == 4
6 test2 = soma_2(5) == 7
7 msg1 = 'sucesso' if test1 else 'falha'
8 msg2 = 'sucesso' if test2 else 'falha'
9 print(f'Resultado do teste 1: {msg1}')
10 print(f'Resultado do teste 2: {msg2}')
11
PROBLEMS OUTPUT TERMINAL ... 1: powershell +
PS E:\POO> python aula09/meu_modulo.py
Resultado do teste 1: sucesso
Resultado do teste 2: falha
PS E:\POO>

```

OUTLINE

Python 3.9.1 64-bit 0 △ 0 Ln 11, Col 1 Spaces: 4 UTF-8 CRLF Python

**Fonte:** do autor, 2021.

**Codificação 9.3: Comando utilizado na Figura 9.1 e respectiva saída no terminal**

```
> python aula04/meu_modulo.py
Resultado do teste 1: sucesso
Resultado do teste 2: falha
```

**Fonte:** do autor, 2021.

A primeira coisa que podemos fazer para melhorar nossos testes é criá-los em um arquivo separado, pois não é uma boa prática deixar nossas funções e nossos testes em um mesmo arquivo. Portanto crie na mesma pasta um arquivo chamado “test\_meu\_modulo.py” e copie para lá o código dos testes. Para executá-los, precisamos acessar a nossa função que continua no arquivo original, então para isso vamos importá-la colocando na primeira linha do arquivo a instrução da Codificação 9.4.

**Codificação 9.4: Instrução para importação da função soma\_2 no arquivo de teste**

```
from meu_modulo import soma_2
```

**Fonte:** do autor, 2021.

A criação e importação de módulos em Python é um assunto já visto em outra aula, mas da forma acima, não precisamos alterar o código do teste, pois estamos importando a partir de “meu\_modulo.py” a função `soma_2`. O resultado pode ser visto na Figura 9.2.

**Figura 9.2: Execução da segunda versão dos testes automatizados**

The screenshot shows the Visual Studio Code interface. On the left, the Explorer sidebar shows a folder structure under 'POO' containing 'aula09', which further contains 'meu\_modulo.py' and 'test\_meu\_modulo.py'. The 'meu\_modulo.py' file contains a single function definition:

```
aula09 > meu_modulo.py > ...
1 def soma_2(numero):
2     ...return 2 * numero
```

The 'test\_meu\_modulo.py' file contains test cases for the 'soma\_2' function:

```
aula09 > test_meu_modulo.py > ...
1 from meu_modulo import soma_2
2
3 test1 = soma_2(2) == 4
4 test2 = soma_2(5) == 7
5 msg1 = 'sucesso' if test1 else 'falha'
6 msg2 = 'sucesso' if test2 else 'falha'
7 print(f'Resultado do teste 1: {msg1}')
8 print(f'Resultado do teste 2: {msg2}')
```

The terminal at the bottom shows the output of running the test script:

```
PS E:\POO> python aula09/test_meu_modulo.py
Resultado do teste 1: sucesso
Resultado do teste 2: falha
PS E:\POO>
```

The status bar at the bottom indicates Python 3.9.1 64-bit, 0 errors, and 0 warnings.

**Fonte: do autor, 2021.**

Conseguimos separar as responsabilidades de cada arquivo, então agora poderíamos tentar melhorar as mensagens exibidas, incluindo informações dos valores esperados e obtidos, entre outras. No entanto, fazer isso para todos os testes nos daria muito trabalho extra com configuração e formatação, tirando o foco do desenvolvimento dos testes que verificam a lógica da aplicação em si.

Para nos ajudar nisso, vamos usar o *pytest*, uma ferramenta de testes automatizados que simplifica a construção e execução dos testes.

### 9.3.1. Instalando o Pytest

A instalação do *pytest* é feita através do gerenciador de pacotes do Python, que já usamos em aulas passadas, o PIP. Execute no terminal o comando da Codificação 9.5 referente ao sistema operacional que estiver utilizando.

#### Codificação 9.5: Comando para instalação do pytest

```
> py -m pip install -U pytest          # Windows
> python3 -m pip install -U pytest    # Linux e Mac
```

**Fonte: do autor, 2021.**

Caso ao final da instalação o pip mostre um aviso (*warning*) informando que o caminho para a pasta de pacotes do Python não está na PATH do sistema, adicione tal

pasta a variável de ambiente PATH do seu usuário<sup>4</sup>. Em seguida, verifique se a instalação foi bem sucedida conferindo a versão instalada, como na Codificação 9.6.

**Codificação 9.6: Verificação da instalação correta do módulo pytest**

```
> pytest --version
pytest 6.2.2
```

Fonte: do autor, 2021.

Caso o comando da Codificação 9.6 não funcione corretamente, tente executar o comando da codificação 9.7. Se isso tampouco funcionar, é possível que sua instalação do Python esteja com algum erro ou que não tenha sido adicionado à PATH do sistema (para Windows). Às vezes, reinstalar o Python pode resolver o problema.

**Codificação 9.7: Verificação da instalação correta do módulo pytest**

```
> py -m pytest --version          # Windows
> python3 -m pytest --version    # Linux e Mac
pytest 6.2.2
```

Fonte: do autor, 2021.

Ao executar o *pytest*, ele irá automaticamente buscar na pasta atual e em todas as subpastas por arquivos da forma “test\_\*.py” ou “\*\_test.py”, ou seja que comecem ou terminem com a palavra *test* seguida ou precedida, respectivamente, de um sublinhado<sup>5</sup>. Após encontrar os arquivos que correspondem ao padrão de nome, o *pytest* busca os testes em si, que podem ser funções ou classes. Vamos começar vendo a criação de testes usando funções.

O *pytest* irá executar todas as funções que começarem com “test\_”, caso a função execute até o final sem levantar nenhum erro, o teste passa, se ocorrer qualquer erro durante a execução da função, o teste falha, e a informação do erro é exibida junto com os resultados do teste.

### 9.3.2. Escrevendo funções de teste com o Pytest

Para escrever nossos testes, usamos o comando `assert` para verificar se a condição que queremos é atendida ou não, ele nos permite avaliar qualquer expressão lógica e levanta um erro (do tipo `AssertionError`) caso ela seja avaliada para falso. É possível gerar erros personalizados, como veremos mais adiante na seção 9.4.

Abra uma *Shell* do Python e pratique o uso do `assert`. Siga os exemplos da Codificação 9.8.

<sup>4</sup> Caso não saiba adicionar um caminho à variável de ambiente PATH, uma rápida pesquisa no google irá te trazer diversos tutoriais sobre como editar esta variável de ambiente do seu sistema operacional.

<sup>5</sup> O asterisco indica que o resto do nome do arquivo não importa, podem ser quaisquer caracteres válidos para nomes de arquivos.

**Codificação 9.8: Exemplos de uso do comando assert do Python**

```
>>> assert True
>>> assert 3 == 2 + 1
>>> assert False
```

**Fonte:** do autor, 2021.

As duas primeiras expressões avaliam para `True`, portanto não há nenhum retorno do comando `assert`, já a última avalia para `False` e levanta o erro mostrado na Codificação 9.9.

**Codificação 9.9: Exemplo do erro levantado pelo comando assert**

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

**Fonte:** do autor, 2021.

O `pytest` captura o erro e faz uma introspecção para avaliar o motivo que o gerou, mostrando o resultado na tela para nós. Vamos então alterar nosso código criando duas funções de teste. Altere o código do arquivo “`test_meu_modulo.py`” para corresponder à Codificação 9.10.

**Codificação 9.10: Funções de teste do arquivo `test_meu_modulo.py`**

```
from meu_modulo import soma_2

def test_1_numero_2():
    assert soma_2(2) == 4

def test_2_numero_5():
    assert soma_2(5) == 7
```

**Fonte:** do autor, 2021.

A Figura 9.3 mostra ambos os arquivos no VSCode, lembrando que para o teste funcionar da forma que fizemos, eles precisam estar na mesma pasta.

```

meu_modulo.py
1 def soma_2(numero):
2     return 2 * numero
3

test_meu_modulo.py
1 from meu_modulo import soma_2
2
3
4 def test_1_numero_2():
5     assert soma_2(2) == 4
6
7
8 def test_2_numero_5():
9     assert soma_2(5) == 7
10

```

Fonte: do autor, 2021.

### 9.3.3. Executando os testes com o Pytest

O VSCode nos traz alguns atalhos e consegue gerenciar a execução dos testes, de modo que podemos escolher executar um por um ou todos de uma vez, mas é importante aprender o funcionamento usando a linha de comando, assim não ficamos dependentes de uma ferramenta específica. Para executar o *pytest* no terminal, há pelo menos três maneiras diferentes:

1) > `pytest`

Digitando apenas o comando no terminal, o *pytest* irá buscar e executar todos os testes da pasta atual e suas subpastas.

2) > `pytest caminho/para/arquivo_test.py`

Digitando o comando no terminal seguido do nome do arquivo que queremos executar, o *pytest* irá executar apenas os testes contidos neste arquivo, sem fazer a busca por arquivos de teste e, portanto, nesse caso o nome do arquivo não importa.

3) > `py -m pytest` # Windows

> `python3 -m pytest` # Linux e Mac

Precedendo qualquer uma das opções anteriores por `python -m` fará com que o *pytest* seja executado a partir do interpretador do python, como um módulo. Dependendo da forma como o *pytest* tenha sido instalado, pode ser que o seu comando, como mostrado nos itens 1 e 2, não seja reconhecido no terminal, então a forma do item 3 pode ser uma solução mais rápida que atualizar a PATH ou reinstalar o Python.

Um guia completo da utilização do *pytest* no terminal de comandos pode ser visto na documentação oficial (KREGEL, 2020).

### 9.3.4. Interpretando os resultados do Pytest

Após a execução do *pytest*, será exibido no terminal um resumo sobre os testes encontrados e executados, uma saída com detalhes do erro para cada teste que falhou e por fim uma versão resumida dos erros nos testes que falharam. Vejamos a saída para o exemplo dado acima.

#### 9.3.4.1. Resumo dos testes coletados

**Codificação 9.11: Resumo dos testes do arquivo *test\_meu\_modulo.py***

```
PS E:\POO> pytest
=====
platform win32 -- Python 3.9.1, pytest-6.2.2, py-1.10.0, pluggy-0.13.1
rootdir: E:\POO
collected 2 items

aula09\test_meu_modulo.py .F [100%]
```

**Fonte:** do autor, 2021.

Podemos ver as informações da plataforma (sistema operacional) e versões do interpretador do Python e do *pytest*, diretório raiz a partir do qual o comando de testes foi executado, a quantidade de testes encontrados e coletados pela busca automática, e por fim, na última linha temos:

- o(s) arquivo(s) de teste executado(s);
- um “ponto” para testes bem sucedidos e uma letra F para testes com falha;
- a porcentagem de testes executados, independente de terem passado ou não.

Caso sejam encontrados mais de um arquivo de testes, haverá uma linha para cada um.

#### 9.3.4.2. Saída detalhada dos testes

**Codificação 9.12: Saída detalhada dos testes do arquivo *test\_meu\_modulo.py***

```
=====
FAILURES =====
_____ test_2_numero_5 _____

def test_2_numero_5():
>     assert soma_2(5) == 7
E     assert 10 == 7
E         + where 10 = soma_2(5)

aula09\test_meu_modulo.py:9: AssertionError
```

**Fonte:** do autor, 2021.

Podemos ver aqui uma seção com o nome do teste que falhou, em seguida a linha de definição deste teste e a introspecção do assert que levantou o erro, indicando tanto o código da linha que levantou o erro quanto os valores que foram avaliados durante a execução do teste. Por fim, temos novamente o nome do arquivo, mas dessa vez seguido de um número, que representa a linha do código em que houve o erro durante o teste, e o tipo do erro.

Nesse caso, podemos concluir que a chamada à função `soma_2`, com o valor 5 retornou o valor 10, que quando comparado com o valor esperado 7, resultou em `False` e por isso o erro no `assert`.

#### 9.3.4.3. Saída resumida dos testes

Codificação 9.13: Saída resumida dos testes do arquivo `test_meu_modulo.py`

```
=====
short test summary info =====
FAILED aula09/test_meu_modulo.py::test_2_numero_5 - assert 10 == 7
===== 1 failed, 1 passed in 0.29s =====
```

Fonte: do autor, 2021.

Na saída resumida, vemos uma lista de testes que falharam, que inclui o nome do arquivo, o nome do teste e a comparação que falhou, e em seguida um resumo geral da execução, indicando quantos testes falharam e passaram, e o tempo de execução. A Figura 9.4 mostra a saída vista no terminal do VSCode.

Figura 9.4: Visualização da saída dos testes no terminal do VSCode

```
PS E:\POO> pytest
=====
short test summary info =====
FAILED aula09/test_meu_modulo.py::test_2_numero_5 - assert 10 == 7
===== 1 failed, 1 passed in 0.12s =====
```

Fonte: do autor, 2021.

Após analisar o resultado dos nossos testes, vemos que a função `soma_2` está incorreta, pois ao receber o valor 5, não retornou o valor esperado que era  $5 + 2 \rightarrow 7$ . Portanto, vamos corrigir o código e executar novamente os testes. Veja a Figura 9.5.

Figura 9.5: Visualização da saída dos testes no terminal do VSCode após correção do código original

```

meu_modulo.py
1 def soma_2(numero):
2     return 2 + numero
3

test_meu_modulo.py
1 from meu_modulo import soma_2
2
3
4 def test_1_numero_2():
5     assert soma_2(2) == 4
6
7
8 def test_2_numero_5():
9     assert soma_2(5) == 7
10

PROBLEMS OUTPUT TERMINAL ...
1: powershell + - ×
PS E:\P00> pytest
=====
platform win32 -- Python 3.9.1, pytest-6.2.2, py-1.10.0, pluggy-0.13.1
rootdir: E:\P00
collected 2 items

aula09\test_meu_modulo.py .. [100%]

===== 2 passed in 0.01s =====
PS E:\P00>

```

Fonte: Elaborado pelo autor

No momento de criar um teste, não nos importa a implementação da função, apenas o que ela faz, isto é, o que ela precisa receber e o que irá retornar. Com isso podemos ver um pouco da importância de se construir casos de teste adequados e de estudar o motivo que fez os testes não passarem.

Um outro ponto muito importante é que os testes são uma ferramenta para nos auxiliar, mas não são absolutos e não estão acima do nosso discernimento como programadores ao avaliar o código, pois mesmo estando incorreta, a primeira versão da função passou em um teste.

Devemos pensar nos casos de testes de maneira a cobrir o maior número de situações diferentes que seja viável. Tente pensar em outros testes para a função acima. Há outros valores que podem gerar situações semelhantes à do primeiro teste?

#### 9.4. Erros e exceções

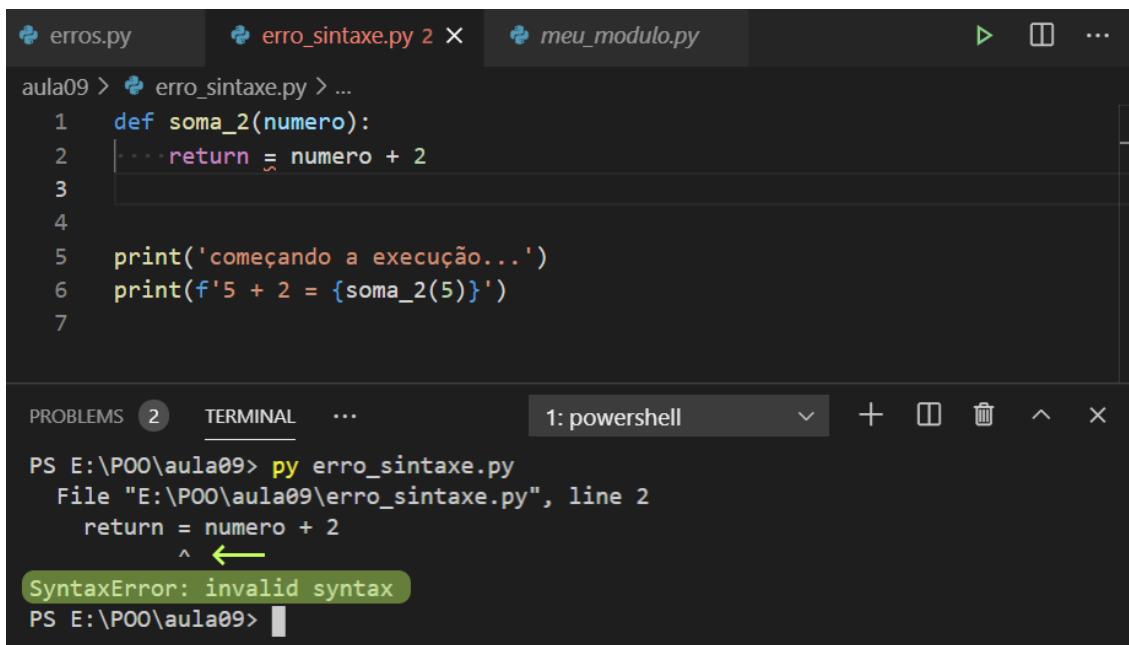
Vimos em LP que podemos agrupar os erros em programação em três categorias, de acordo com sua origem quando eles ocorrem:

- 1) Erros de sintaxe: são erros em que o interpretador não é capaz de entender as instruções pois elas não seguem as regras da linguagem;

- 2) Erros de execução: ocorrem quando as regras estão corretas, mas por algum motivo não é possível executar a ação pedida;
- 3) Erros de semântica: são erros na lógica implementada. Do ponto de vista do interpretador, nada está errado e a execução é bem sucedida, mas a ação realizada não condiz com o comportamento esperado.

No primeiro tipo de erro, os de sintaxe, o interpretador indica o arquivo e a linha na qual ele encontrou o erro, e inclui também um marcador na região em que o erro foi detectado. Os erros de sintaxe devem ser corrigidos para que o código possa ser executado, e como é um erro que está escrito diretamente no código fonte, é possível garantir que todos eles sejam eliminados. Veja no exemplo da Figura 9.6 que nenhum código é executado, pois o interpretador não é capaz de começar a execução já que estamos quebrando uma das regras de sintaxe do Python, no caso, estamos tentando atribuir um valor a uma palavra reservada da linguagem, o comando `return`.

**Figura 9.6: Exemplo de erro de sintaxe ao executar um arquivo \*.py**



```

erros.py      erro_sintaxe.py 2 X  meu_modulo.py
aula09 > erro_sintaxe.py > ...
1  def soma_2(numero):
2  return = numero + 2
3
4
5  print('começando a execução...')
6  print(f'5 + 2 = {soma_2(5)}')
7

PROBLEMS 2 TERMINAL ... 1: powershell + ×
PS E:\P00\aula09> py erro_sintaxe.py
File "E:\P00\aula09\erro_sintaxe.py", line 2
    return = numero + 2
           ^
SyntaxError: invalid syntax
PS E:\P00\aula09>

```

**Fonte:** do autor, 2021.

Já com erros de execução, o segundo tipo, não é possível eliminá-los, pois muitas vezes eles ocorrem em virtude de elementos que fogem ao controle do programador, como por exemplo, uma falha na conexão com o banco de dados, uma entrada incorreta do usuário, uma falha de comunicação com uma API, etc. A estes erros, normalmente damos o nome de exceções, e para evitar que eles façam a nossa aplicação parar de funcionar, existe o tratamento de exceções, que veremos em seguida, na seção 9.4.3.

E por fim, em relação ao terceiro tipo de erro, os de semântica, a melhor forma de lidar com eles é a criação de testes automatizados, como começamos a ver neste capítulo, e que irão testar o funcionamento correto de uma aplicação ou software, com base no comportamento esperado definido no teste. Neste caso, é preciso que o programador escreva os testes manualmente, pois o interpretador não tem como saber o que está errado, dado que a execução é finalizada com sucesso, independentemente de produzir ou não o resultado esperado.

Em relação aos erros de execução (o segundo tipo), podemos distinguir duas situações: o lançamento ou levantamento de uma exceção e o seu respectivo tratamento. Ambas as partes não precisam acontecer sempre, por exemplo, em um trecho da aplicação, podemos apenas levantar exceções que serão tratadas (ou não) pelo código cliente<sup>6</sup>, e em outra podemos apenas tratar exceções que são levantadas por algum módulo que estamos usando (aqui nós seríamos o cliente). Ou podemos ter as duas coisas acontecendo na mesma aplicação: uma parte do código levanta uma exceção que será tratada em outra parte da mesma aplicação.

#### 9.4.1. Lançamento de exceções

Para lançar uma exceção em Python usamos a palavra reservada `raise`<sup>7</sup>, e podemos lançar qualquer exceção da lista de exceções integradas à linguagem, que pode ser vista na documentação (PSF, 2021b). Veja o exemplo da Codificação 9.14.

##### Codificação 9.14: Lançamento de uma exceção do tipo `TypeError`

```
def incrementa_int(n):
    if not isinstance(n, int):
        raise TypeError('n deve ser um inteiro')
    return n + 1
```

*Fonte: do autor, 2021.*

Neste exemplo, caso a função seja chamada com um valor que não seja um número inteiro, o fluxo de execução será interrompido pelo comando `raise`, que irá encerrar a função e “retornar” uma mensagem de erro. Esse retorno não é o mesmo retorno que ocorre quando usamos a palavra chave `return`, ou seja, não é possível atribuí-lo a uma variável, e ele continuará “subindo” na pilha de execução até que seja tratado ou chegue no módulo principal, momento em que o processo em execução é encerrado prematuramente, mostrando na tela a mensagem de erro.

Para entender um pouco melhor o que significa o erro “subir” ou ser “elevado”, podemos pensar que toda vez que chamamos uma função dentro de outra, a função atual fica “pausada”, esperando a função chamada encerrar a sua execução e retornar. Caso a

<sup>6</sup> O termo cliente aqui se refere a qualquer programa ou código que consuma (faça uso) do nosso código.

<sup>7</sup> `raise` pode ser traduzido para *levantar* ou *elevar*.

função retorne seu valor normalmente, o fluxo de execução continua. Caso ela retorne um erro e este erro não seja tratado, a função que recebeu como “retorno” o erro repassa-o para quem a chamou, e assim sucessivamente. Veja o exemplo da codificação 9.15, na qual a função `calcula_idade` é chamada com um número do tipo `float` como argumento.

**Codificação 9.15: Código com lançamento de um erro do tipo `TypeError`**

```
def incrementa_int(n):
    if not isinstance(n, int):
        raise TypeError('n deve ser um inteiro')
    return n + 1

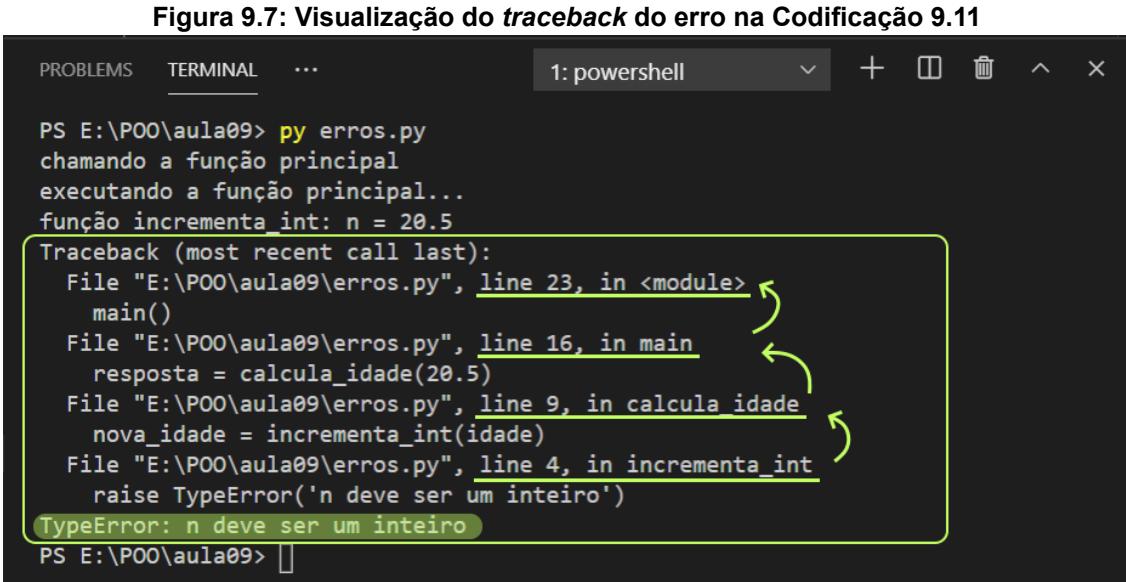
def calcula_idade(idade):
    nova_idade = incrementa_int(idade)
    print('esse código não é executado se der erro na linha acima')
    return nova_idade

def main():
    print('executando a função principal...')
    resposta = calcula_idade(20.5)
    print('esse código não será executado se der erro na linha acima')
    print('a nova idade é:', resposta)

if __name__ == '__main__':
    print('chamando a função principal')
    main()
    print('esse código não será executado se der erro na linha acima')
```

**Fonte: do autor, 2021.**

O erro que ocorreu na função `incrementa_int` foi elevado para a função `calcula_idade`, que por sua vez foi elevado para a função `main`, e por fim foi elevado para o programa principal, de onde a função `main` foi chamada. Como em nenhum momento fizemos o tratamento deste erro, a execução do programa foi interrompida e tal informação é mostrada na tela, onde conseguimos ver todo o caminho percorrido pelo erro, no que chamamos, em inglês, de *Traceback*, como mostra a figura 9.7.



```

PS E:\POO\aula09> py erros.py
chamando a função principal
executando a função principal...
função incrementa_int: n = 20.5
Traceback (most recent call last):
  File "E:\POO\aula09\erros.py", line 23, in <module>
    main()
  File "E:\POO\aula09\erros.py", line 16, in main
    resposta = calcula_idade(20.5)
  File "E:\POO\aula09\erros.py", line 9, in calcula_idade
    nova_idade = incrementa_int(idade)
  File "E:\POO\aula09\erros.py", line 4, in incrementa_int
    raise TypeError('n deve ser um inteiro')
TypeError: n deve ser um inteiro
PS E:\POO\aula09> []

```

Fonte: do autor, 2021.

No exemplo que acabamos de ver, faz sentido lançar um `TypeError`, pois estamos falando exatamente de um erro de tipo (o dado fornecido não é do tipo correto). No entanto, nem sempre o Python terá uma classe que seja adequada para representar as exceções que estamos prevendo em nossa aplicação, portanto é possível criarmos nossas próprias exceções, usando o conceito de herança que vimos na introdução a POO.

#### 9.4.2. Criação de exceções

Em Python toda exceção deve herdar da classe `Exception`, ou de outra classe que por sua vez herde de `Exception`. Isto é, toda nova exceção deve ser uma descendente de `Exception` (filha, neta, etc.), o que garante que o nosso erro ou exceção personalizado irá herdar todos os comportamentos mínimos necessários para que o Python possa identificá-la como uma exceção. É especialmente importante para que um bloco `except Exception` seja capaz de pegar todas as exceções que não sejam específicas, como veremos a seguir na seção 9.4.3.

Imagine que estamos desenvolvendo um sistema de cadastro para uma clínica veterinária, uma funcionalidade que precisaremos implementar será a de criar a ficha dos animais que serão ali tratados, o que podemos modelar em uma classe `Paciente`. Caso aconteça alguma falha durante a criação, como por exemplo se o campo “nome” não for uma `string` ou estiver vazio, podemos levantar um erro neste método. Veja na Codificação 9.16 um exemplo de implementação simplificada da classe `Paciente`, na implementação real desta classe, o método inicializador teria mais parâmetros.

#### Codificação 9.16: Criação e utilização de exceção personalizada

```
class NameIsEmptyError(Exception):
    pass

class Paciente:
    def __init__(self, nome):
        self.paciente = nome
        ... # restante do código que inicializa os dados do paciente

    @property
    def paciente(self):
        return self._paciente

    @paciente.setter
    def paciente(self, nome):
        if not isinstance(nome, str):
            raise TypeError("'nome' inválido")
        if nome == '':
            raise NameIsEmptyError("'nome' é obrigatório")
        self._paciente = nome
```

Fonte: do autor, 2021.

Na classe `Paciente`, da codificação 9.16, estamos usando uma `property/setter` para atribuir o valor do atributo não público `_paciente`. E no `setter`, fazemos uma validação do valor recebido e só permitimos a atribuição caso este valor seja uma *string* e não seja vazio. É comum fazermos a verificação logo no começo e levantarmos os erros pertinentes, interrompendo o fluxo de execução. Dessa forma garantimos que o código que irá consumir esta classe irá “falhar” o mais cedo possível caso use-a de maneira incorreta, permitindo ao desenvolvedor perceber e corrigir o erro.

#### 9.4.3. Tratamento de exceções

O tratamento de exceções é extremamente importante para garantir que a aplicação ou programa continue funcionando ao encontrar algo inesperado em relação ao seu funcionamento normal. Ele captura o erro, impedindo que ele continue subindo na pilha de execução, e permite desvios no fluxo para que sejam tomadas as medidas necessárias em cada caso, o que pode incluir enviar mensagens aos usuários, escrever mensagens de log, reagendar a tarefa que falhou, fechar um arquivo aberto ou encerrar a conexão com o banco de dados, entre outros.

Observe que essa situação pode ser inesperada do ponto de vista do funcionamento padrão, mas do ponto de vista de quem está programando a aplicação, as

fallas são sempre esperadas, pois em uma situação real, não é possível controlar todo o ambiente e garantir que o usuário não irá digitar um valor inválido, que o servidor do banco de dados não irá passar por uma instabilidade ou que a conexão não será interrompida porque uma API estava sobrecarregada e demorou demais para responder. Portanto, faz parte do nosso trabalho identificar os pontos suscetíveis a falha e implementar o tratamento adequado.

Em Python, esse tratamento é feito com o bloco try-except, como mostra a Codificação 9.17.

**Codificação 9.17: Sintaxe do bloco try-except em Python**

```
try:  
    # código suscetível a falha  
except:  
    # código executado após ocorrer um erro  
else:  
    # código executado apenas se nenhum erro ocorrer  
finally:  
    # código executado sempre
```

**Fonte:** do autor, 2021.

Durante o tratamento de uma exceção, o único trecho de código seguro é aquele contido no bloco do try, ou seja, qualquer erro que ocorrer ali será capturado e o fluxo redirecionado para os blocos except. No entanto, se ocorrer um erro nos demais blocos, esse erro irá seguir o fluxo padrão de erros e será elevado na pilha de execução. É possível aninhar blocos try-except, mas em geral isso é considerado uma má prática, pois piora a legibilidade do código.

Veja na Codificação 9.18 um exemplo de código que utiliza a classe Paciente, da Codificação 9.16.

**Codificação 9.18: Exemplo de uso do bloco try-except em Python**

```
from paciente import Paciente, NameIsEmptyError  
  
try:  
    nome = input('Digite o nome do paciente: ')  
    p = Paciente(nome)  
except TypeError:  
    print('O nome deve ser uma string')  
except NameIsEmptyError:  
    print('O nome não pode ser uma string vazia')  
except Exception as e:  
    print('Ocorreu um erro inesperado ao criar o objeto')  
    print('informações do erro:', e)
```

**Fonte:** do autor, 2021.

Aqui podemos observar que há um encadeamento dos tipos de exceções que podem ser capturados, seguindo uma lógica parecida com a dos blocos `elif` no Python, isto é, será executado o bloco `except` da primeira exceção que corresponder, sendo os blocos subsequentes ignorados, pulando direto pro bloco do `finally`, se houver.

Devido a natureza da função `input`, que sempre retorna uma string, não é possível termos um erro de tipo, e devido a simplicidade deste exemplo, dificilmente teremos um erro genérico, portanto neste exemplo simplificado, poderíamos capturar apenas a exceção `NameIsEmptyError`.

Caso queira testar a captura dos demais erros, podemos introduzir erros propositais no código. Faça o teste substituindo, no bloco do `try`, a instrução do `input` por `nome = 23`, ou forçando, também no bloco do `try`, o levantamento de um erro genérico com `raise Exception('erro genérico')`<sup>8</sup>.

Há ainda três pontos importantes sobre o tratamento de exceções no Python:

- 1) Como no exemplo dado, é possível capturar o objeto de exceção levantado pelo Python em uma variável, usando o comando `as` para fazer a atribuição, como no exemplo da codificação 9.18.
- 2) É possível agrupar mais de um tipo de exceção no mesmo bloco `except`, colocando-as em uma tupla:

**Codificação 9.19: Agrupamento de mais de um tipo de exceção**

```
try:  
    ...  
except (TypeError, ValueError, ZeroDivisionError):  
    ...
```

**Fonte: do autor, 2021.**

- 3) Podemos também apenas interceptar a exceção, fazer algum tipo de tratamento, como por exemplo salvar em um registro de log a ocorrência, e deixá-la seguir o seu fluxo natural para ser tratada em outra parte da aplicação. Para isso, basta usar o comando `raise` sozinho, dentro de um bloco `except`:

**Codificação 9.20: Interceptando a exceção**

```
except:  
    logger.exception('salvando log da exceção')  
    raise
```

**Fonte: do autor ,2021.**

Com isso, a mesma exceção que ocorreu originalmente será relançada com o comando `raise`, após a execução do tratamento parcial.

<sup>8</sup> Observe que levantar um erro dentro de um bloco `try` fará com que esse erro seja capturado e tratado no próprio bloco, portanto isso não é uma prática utilizada normalmente, servindo aqui apenas para ilustrar o exemplo dado, já que no código do exemplo, não há margem para um erro genérico. Alternativamente, poderíamos ter editado a nossa classe para, ao receber um nome específico, levantar um erro genérico, este exemplo seria mais perto de um exemplo de utilização real do tratamento de exceções.

## Bibliografia

- BECK, K. **Test-driven development by example.** Boston: Addison-Wesley Professional. 2002.
- BECK, K. **Quora:** why does kent beck refer to the "rediscovery" of test-driven development? 2012. Disponível em: <<https://qr.ae/pGIVYU>>. Acesso em: 18 abr. 2021.
- KREGEL, H. **Pytest:** usage and invocations. 2020. Disponível em: <<https://docs.pytest.org/en/stable/usage.html>>. Acesso em: 10 fev. 2021.
- PSF. **Expressions:** conditional expressions. 2021a. Disponível em: <<https://docs.python.org/3/reference/expressions.html#conditional-expressions>>. Acesso em: 07 fev. 2021.
- PSF. **Built-in exceptions:** exception hierarchy. 2021b. Disponível em: <<https://docs.python.org/3/library/exceptions.html#exception-hierarchy>>. Acesso em: 18 abr. 2021.