Professor: Ruan Chaves Machado

Data do Material Didático Criado: 16/05/2025

Data da Aula: 23/05/2025

Tópicos e Temas abordados nessa aula:

* Test-Driven Development (TDD): Conceitos e Ciclo de Desenvolvimento
* Verificações com assert: Testes Simples e Validações de Código
* Testes Estruturados com unittest: Framework Padrão do Python
* Simulando Comportamentos com unittest.mock
* Python Enhancement Proposal (PEPs) - O que São e Para que servem
* PEP8: Guia de Estilo Oficial para Código Python

**Test-Driven Development (TDD): Conceitos e Ciclo de Desenvolvimento**

TDD significa Desenvolvimento Orientado a Testes (Test-Driven Development).  
É uma metodologia de desenvolvimento de software onde o ciclo de desenvolvimento segue a seguinte ordem:

1. Escreva um teste que verifica uma funcionalidade que ainda não existe.
2. Execute o teste e veja ele falhar (isso é esperado).
3. Implemente o código mínimo necessário para fazer o teste passar.
4. Refatore o código, mantendo os testes funcionando.
5. Repita o ciclo.

**Esse ciclo é chamado de Red – Green – Refactor:**

| **Fase** | **Descrição** |
| --- | --- |
| 🔴 Red | Escreve-se o teste e ele falha. |
| ✅ Green | Escreve-se o código mínimo para passar o teste. |
| 🔧 Refactor | Melhoramos o código mantendo os testes funcionando. |

Vale ressaltar que existem diferentes tipos de testes, sendo eles:

* **Unitários**: Garantem a funcionalidade de pequenas unidades de código, como uma função;
* **Testes de Performance**: Garantem e determinam a fluidez do código;
* **Segurança**: Garantem que o código não possa ser explorado por um usuário mal-intencionado, dentre outros.

O que deve ser testado principalmente são as **regras de negócio**, ou seja, o **código de funções e classes que vão realizar algum cálculo, processar dados ou tomar decisões**. Isso inclui verificar os *inputs* que são passados para saber se os tipos de dados estão corretos, no formato adequado e no alcance (*range*) esperado.

**Exemplo simples do ciclo TDD:**

Vamos simular o ciclo TDD para uma função que retorna o dobro de um número.

**Etapa 1 – Escrevendo o teste antes do código**

**O que são Regras de Negócio?**

As regras de negócio são um conjunto de normas e diretrizes que definem como uma organização opera, orientando processos, decisões e comportamento dentro da empresa. Elas são como as "leis" internas que guiam as operações e asseguram que a empresa atinja seus objetivos.

# test\_dobro.py

**def** test\_dobro**():**

**assert** dobro**(**2**)** **==** 4 # A função ainda nem existe!

**Etapa 2 – Executar e ver o erro**

Você verá:

**NameError:** name 'dobro' **is** **not** defined

**Etapa 3 – Implementar o código mínimo**

**def** dobro**(**n**):**

**O que é esse termo REFATORAR ou REFATORAÇÃO DE CÓDIGO?**

No TDD (Test-Driven Development), a refatoração é uma etapa essencial após a implementação do código para passar nos testes e antes de se preocupar com a adição de novas funcionalidades. Trata-se do processo de reestruturar o código sem alterar seu comportamento, visando melhoria da estrutura, legibilidade e manutenção.

**return** n **\*** 2

**Etapa 4 – Rodar o teste novamente**

$ pytest test\_dobro**.***py*

# Resultado: ✅ Teste passou!

**Etapa 5 – Refatorar (se necessário)**

Se o código estiver limpo, não há refatoração por enquanto.

**Ferramentas comuns para TDD com Python:**

* **assert** (nativo, para testes simples)
* **unittest** (módulo padrão para testes estruturados)
* **pytest** (mais avançado e moderno – opcional)
* **mock** (para simular comportamentos externos)

**Exemplo de Uso: assert**

Ideal para validar comportamentos básicos em scripts pequenos.

**def** soma**(**a**,** b**):**

**return** a **+** b

# Testes usando assert

**assert** soma**(**2**,** 3**)** **==** 5

**assert** soma**(-**1**,** 1**)** **==** 0

Se algum assert falhar, o Python lança um AssertionError.

**Exemplo de Uso: unittest**

Organiza os testes em **classes e métodos**. Permite relatórios mais claros e testes mais robustos.

**import** unittest

**if \_\_name\_\_ == '\_\_main\_\_': — O que significa isso?**

Esse bloco é uma **estrutura de controle** em Python que garante que **certos trechos de código só sejam executados quando o arquivo for rodado diretamente** (e não importado por outro).

**Em outras palavras:**

“Se esse arquivo for executado diretamente no terminal (ou no VS Code, etc.), então execute o que está dentro desse bloco.”

\_\_name\_\_ é uma **variável especial** em Python que armazena o nome do módulo atual.

* Quando o arquivo é **executado diretamente**, \_\_name\_\_ vale '\_\_main\_\_'.
* Quando o arquivo é **importado por outro**, \_\_name\_\_ vale o nome do módulo (ex: 'testes', 'modulo\_calculadora' etc).

**def** multiplicar**(**a**,** b**):**

**return** a **\*** b

**class** **TestMultiplicacao(**unittest**.***TestCase***):**

**def** test\_multiplicar\_positivos**(**self**):**

self**.***assertEqual***(**multiplicar**(**2**,** 3**),** 6**)**

**def** test\_multiplicar\_zero**(**self**):**

self**.***assertEqual***(**multiplicar**(**0**,** 100**),** 0**)**

**if** \_\_name\_\_ **==** "\_\_main\_\_"**:**

unittest**.***main***()**

**Para podermos rodar esse comando no terminal:**

*python nome\_do\_arquivo.py*

**Exemplo de Uso: pytest**

Com pytest, você escreve testes como funções comuns, sem precisar de classes.

O pytest coleta e executa automaticamente os testes com prefixo test\_.

**def** dividir**(**a**,** b**):**

**return** a **/** b

**def** test\_divisao\_normal**():**

**assert** dividir**(**10**,** 2**)** **==** 5

**def** test\_divisao\_decimal**():**

**assert** dividir**(**7**,** 2**)** **==** 3.5

**Para podermos rodar esse comando no terminal:**

*pytest test\_divisao.py*

O resultado deverá ser um relatório de testes automático, bem legível.

**Exemplo de Uso: pytest**

Usado para simular funções, APIs ou objetos externos que não queremos (ou não podemos) executar no teste.

Portanto, suponha uma função que envia e-mails:

**def** enviar\_email**(**destinatario**):**

**print(**f"E-mail enviado para {destinatario}"**)**

**return** **True**

Agora vamos **testar** essa função **sem realmente enviar um e-mail**, usando ***unittest.mock***:

**from** unittest**.***mock* **import** patch

**import** unittest

# Função que chama o envio de e-mail

**def** registrar\_usuario**(**nome**,** email**):**

enviar\_email**(**email**)**

**return** f"Usuário {nome} registrado"

**class** **TestEmail(**unittest**.***TestCase***):**

*@patch***(**"\_\_main\_\_.enviar\_email"**)**

**def** test\_registrar\_usuario**(**self**,** mock\_enviar**):**

mock\_enviar**.***return\_value* **=** **True**

resultado **=** registrar\_usuario**(**"João"**,** "joao@email.com"**)**

self**.***assertEqual***(**resultado**,** "Usuário João registrado"**)**

mock\_enviar**.***assert\_called\_once\_with***(**"joao@email.com"**)**

**if** \_\_name\_\_ **==** "\_\_main\_\_"**:**

unittest**.***main***()**

Aqui, o mock\_enviar substitui a função enviar\_email temporariamente, evitando efeitos colaterais.

Então para resumir e finalizar, a tabela abaixo mostra a **Ferramenta de Teste, quando usar, suas Vantagens e Limitações:**

| **Ferramenta** | **Tipo** | **Quando usar** | **Vantagens** | **Limitações** |
| --- | --- | --- | --- | --- |
| **assert** | Verificação simples | Scripts pequenos ou para introdução ao TDD | Muito fácil de usar- Sem bibliotecas extras | Relatórios de erro simples- Difícil organizar muitos testes |
| **unittest** | Framework de testes | Projetos de pequeno a médio porte | Incluído no Python- Suporte a mocks- Organização por classes | Sintaxe mais verbosa- Pouco intuitivo para iniciantes |
| **pytest** | Framework de testes | Projetos modernos de qualquer porte | Sintaxe limpa e elegante- Relatórios detalhados- Plugins poderosos | Requer instalação externa (pip install pytest) |
| **mock** | Simulação de dependências | Quando você não quer executar funções reais | Testa comportamento sem efeitos colaterais- Integrado ao unittest | Curva de aprendizado no uso correto de @patch |

**Verificações com assert: Testes Simples e Validação de Código**

O assert é uma **ferramenta interna da linguagem Python** usada para criar **testes rápidos de validação** durante o desenvolvimento do código. Ele serve para garantir que **determinadas condições** sejam verdadeiras em tempo de execução.  
Caso **não sejam**, o Python **interrompe a execução** e levanta uma exceção chamada AssertionError.

Em outras palavras:

O assert é como um “fiscal” que você coloca no código para verificar se o que você espera realmente está acontecendo.

**Por que usar o assert?**

Imagine que você criou uma função que calcula a média de dois números. Com o assert, você pode escrever uma “regra de ouro” que diz:  
*"Essa função só está certa se a média de 2 e 4 for igual a 3."*

**def** media**(**a**,** b**):**

**return** **(**a **+** b**)** **/** 2

**assert** media**(**2**,** 4**)** **==** 3

Se a função estiver errada, esse assert vai te avisar imediatamente.

A sua sintaxe básica é a seguinte:

**assert** expressão\_booleana**,** "Mensagem de erro opcional"

Portanto:

* Se a **expressão for verdadeira**, o programa continua normalmente.
* Se for **falsa**, o programa para com um AssertionError.

**Exemplo 1 – Testando uma função de soma**

**def** soma**(**a**,** b**):**

**return** a **+** b

**assert** soma**(**2**,** 2**)** **==** 4

**assert** soma**(**1**,** **-**1**)** **==** 0

**Exemplo 2 – Teste com erro**

**def** dividir**(**a**,** b**):**

**return** a **/** b

**assert** dividir**(**10**,** 2**)** **==** 5 # Ok

**assert** dividir**(**10**,** 3**)** **==** 3.3 # AssertionError: resultado é 3.333...

**Quando usar assert**

* Para **testes rápidos durante o desenvolvimento**.
* No início do projeto como **primeiro passo no TDD**.
* Em ambientes de **teste ou prototipagem**.

⚠️ **não é recomendado** usar assert como substituto de testes formais ou em código de produção. Ele pode ser **ignorado** quando o Python roda com a flag -O (modo otimizado).

É possível também que criemos mensagens de Erro personalizadas, assim facilita o **entendimento e depuração de onde o problema/erro** está sendo gerado:

**def** dobro**(**n**):**

**return** n **\*** 2

**assert** dobro**(**4**)** **==** 8**,** "A função dobro deve retornar o dobro do número"

**Exercício rápido de reforço com assert: Validação de números pares**

Criar uma função chamada ***eh\_par(n)*** que retorne **True** se o número for par, e usar ***assert*** para verificar seu funcionamento.

**def** eh\_par**(**n**):**

"""Retorna True se o número for par, senão False."""

**return** n **%** 2 **==** 0

# Testes válidos (todos devem passar)

**assert** eh\_par**(**2**)** **==** **True**

**assert** eh\_par**(**10**)** **==** **True**

**assert** eh\_par**(**0**)** **==** **True**

# Teste propositalmente errado para gerar AssertionError

**assert** eh\_par**(**3**)** **==** **True,** "3 não é par, deve retornar False"

**Testes Estruturados com unittest: Framework Padrão do Python**

O framework de testes unitários [unittest](https://docs.python.org/pt-br/dev/library/unittest.html" \l "module-unittest" \o "unittest: Unit testing framework for Python.) foi originalmente inspirado no JUnit e tem um gostinho semelhante contendo as principais estruturas de teste de unidades existentes em outras linguagens. Ele suporta a automação de testes, compartilhamento de configuração e código de desligamento para testes, agregação de testes em coleções e independência dos testes do framework de relatórios.

O unittest é o **framework de testes nativo do Python**, baseado no padrão de testes unitários criado pela linguagem Java (JUnit). Ele permite que você:

* Escreva **testes automatizados estruturados**.
* Organize os testes em **classes e métodos**.
* Valide partes individuais do seu código (funções, classes, métodos).
* Gere **relatórios claros de sucesso ou falha**.

**Analogia Simples:**

Imagine que você é um engenheiro testando **cada botão de um painel de controle** de um avião.  
Antes de deixar o avião voar (colocar o sistema em produção), você testa **cada botão isoladamente** para ter certeza de que todos funcionam corretamente.

Assim também fazemos com o código: **antes de rodar o sistema completo, testamos função por função**.

**Exemplo simples com unittest:**

**import** unittest

**def** eh\_par**(**n**):**

**return** n **%** 2 **==** 0

**class** **TestParidade(**unittest**.***TestCase***):**

**def** test\_par**(**self**):**

self**.***assertTrue***(**eh\_par**(**2**))** # Espera que retorne True

**def** test\_impar**(**self**):**

self**.***assertFalse***(**eh\_par**(**3**))** # Espera que retorne False

**if** \_\_name\_\_ **==** '\_\_main\_\_'**:**

unittest**.***main***()**

**Explicação por partes:**

* **import** unittest **– importa o módulo de testes.**
* **Class** **TestParidade(**unittest**.***TestCase***): – define uma classe de teste,** sempre herdando de unittest**.***TestCase*
* **def** test\_par**(**self**): – métodos de teste** devem começar com test\_**, para o framework reconhecê-los como testes.**
* self**.***assertTrue***(**...**),** self**.***assertFalse***(**...**)– são** métodos de asserção**, usados para verificar se a função retorna o esperado.**

**A saída esperada é:**

**..**

**----------------------------------------------------------------------**

Ran 2 tests **in** 0.001s

OK

**Principais métodos de asserção em unittest**

| **Método** | **Verifica se...** |
| --- | --- |
| assertEqual(a, b) | a == b |
| assertNotEqual(a, b) | a != b |
| assertTrue(x) | bool(x) is True |
| assertFalse(x) | bool(x) is False |
| assertIs(a, b) | a is b (mesmo objeto) |
| assertIn(a, b) | a in b |
| assertRaises(Error) | Uma exceção específica é lançada |

**Simulando Comportamentos com unittest.mock**

O[*unittest.mock*](https://docs-python-org.translate.goog/3/library/unittest.mock.html?_x_tr_sl=en&_x_tr_tl=pt&_x_tr_hl=pt&_x_tr_pto=tc#module-unittest.mock) é uma biblioteca para testes em Python. Ela permite substituir partes do seu sistema em teste por objetos simulados e fazer afirmações sobre como eles foram utilizados.

[unittest.mock](https://docs-python-org.translate.goog/3/library/unittest.mock.html?_x_tr_sl=en&_x_tr_tl=pt&_x_tr_hl=pt&_x_tr_pto=tc#module-unittest.mock)fornece uma classe central chamada [Mock](https://docs-python-org.translate.goog/3/library/unittest.mock.html?_x_tr_sl=en&_x_tr_tl=pt&_x_tr_hl=pt&_x_tr_pto=tc#unittest.mock.Mock), eliminando a necessidade de criar uma série de stubs (partes de código que não está implementada) em todo o seu conjunto de testes. Após executar uma ação, você pode fazer afirmações sobre quais métodos/atributos foram usados ​​e os argumentos com os quais foram chamados. Você também pode especificar valores de retorno e definir os atributos necessários normalmente.

Ou seja: O mock é uma ferramenta de testes que permite simular o comportamento de objetos, funções ou métodos.

Isso é extremamente útil em situações onde você:

* **Não quer executar a função real (porque ela acessa banco de dados, internet, arquivos, etc.).**
* **Quer testar o código isoladamente, sem depender de serviços externos.**
* **Quer simular um resultado específico ou um erro controlado.**

**Analogia Simples:**

Imagine que você tem uma função que envia um e-mail para um cliente após ele se cadastrar.

Mas no momento dos testes, **você não quer enviar e-mails de verdade**, só quer **simular** que o envio foi feito com sucesso.

O mock é como um “ator substituto” no seu código: ele **finge ser uma função real**, mas não faz nada de verdade — só **atua**.

**Exemplo simples: simulando envio de e-mail**

(Explicar para eles rapidamente sobre módulos (arquivo) para poder usar esse exemplo)

# Arquivo: cadastro.py

**def** enviar\_email**(**destinatario**,** assunto**,** corpo**):**

**print(**f"E-mail enviado para {destinatario}"**)** # Em produção: código real de envio

**def** cadastrar\_usuario**(**nome**,** email**):**

# lógica de cadastro aqui

enviar\_email**(**email**,** "Bem-vindo"**,** "Obrigado por se cadastrar!"**)**

**return** **True**

# Arquivo: test\_cadastro.py

**import** unittest

**from** unittest**.***mock* **import** patch

**from** cadastro **import** cadastrar\_usuario

**class** **TestCadastroUsuario(**unittest**.***TestCase***):**

*@patch***(**'cadastro.enviar\_email'**)** # Substitui a função real por um mock

**def** test\_cadastro\_dispara\_email**(**self**,** mock\_enviar**):**

resultado **=** cadastrar\_usuario**(**"João"**,** "joao@email.com"**)**

mock\_enviar**.***assert\_called\_once\_with***(**

"joao@email.com"**,** "Bem-vindo"**,** "Obrigado por se cadastrar!"

**)**

self**.***assertTrue***(**resultado**)**

**if** \_\_name\_\_ **==** '\_\_main\_\_'**:**

unittest**.***main***()**

**Explicação do Código acima:**

| **Código** | **Significado** |
| --- | --- |
| @patch('cadastro.enviar\_email') | Diz para substituir a função enviar\_email do módulo cadastro |
| mock\_enviar | É o objeto simulado, passado como parâmetro automático no teste |
| mock\_enviar.assert\_called\_once\_with(...) | Verifica se a função foi chamada **exatamente uma vez**, com esses argumentos |
| self.assertTrue(resultado) | Confirma que o retorno da função de cadastro foi True |

**Métodos úteis com *Mock*:**

| Método | Verifica se... |
| --- | --- |
| mock.called | O mock foi chamado (True/False) |
| mock.call\_count | Quantas vezes foi chamado |
| mock.assert\_called() | Foi chamado pelo menos uma vez |
| mock.assert\_called\_once() | Foi chamado exatamente uma vez |
| mock.assert\_called\_with(...) | Foi chamado com os argumentos esperados |
| mock.return\_value = ... | Define o que o mock deve retornar |

**⚠️ Dica de ouro:**

**Quando usar patch, sempre coloque o** caminho completo **da função no módulo onde ela é usada,** não onde ela foi definida**.**

**Python Enhancement Proposal (PEPs) - O que São e Para que servem**

**PEP** significa **Python Enhancement Proposal** (Proposta de Aprimoramento do Python).  
É um documento oficial usado para **propor melhorias** na linguagem Python, seja na sintaxe, boas práticas ou funcionamento interno.

A **PEP 8** é a PEP que define o **guia de estilo** do Python.  
Ou seja: **como escrever código limpo, legível, consistente e Pythonico**.

**Analogia Simples:**

É como o “manual de redação” para desenvolvedores Python.  
Ajuda a manter o código padronizado, o que facilita o trabalho em equipe e a manutenção futura.

**Por que seguir a PEP 8?**

* **Facilita a leitura e compreensão do código.**
* **Facilita o trabalho em equipe (todos escrevem da mesma forma).**
* **Evita bugs causados por nomenclaturas confusas ou más práticas.**
* **Melhora a reputação profissional do desenvolvedor.**
* **É um sinal de maturidade técnica.**

**Principais Regras da PEP8:**

**1 - Nomenclatura de variáveis e funções**

| **Tipo** | **Exemplo** | **Regra** |
| --- | --- | --- |
| Variáveis / funções | minha\_variavel | snake\_case |
| Classes | MinhaClasse | PascalCase |
| Constantes | VALOR\_MAXIMO | UPPER\_CASE |

**2 – Identação**

* Use 4 espaços para cada nível de identação
* Nunca use tab

**def** minha\_funcao**():**

**if** **True:**

**print(**"Indentado corretamente"**)**

**3. Comprimento de linhas**

* Máximo recomendado: 79 caracteres por linha.
* Linhas muito longas dificultam a leitura em IDEs e revisões de código.

**4. Espaços**

Evite usar espaços desnecessários!

✅ Correto:

x **=** 1

y **=** x **+** 2

**print(**x**,** y**)**

❌ Errado:

x**=**1

y **=** x**+**2

**5.** Imports organizados

* **1º: Imports da biblioteca padrão**
* **2º: Imports de bibliotecas de terceiros**
* **3º: Imports locais (do próprio projeto)**

Com uma linha em branco separando:

**import** os

**import** sys

**import** requests

**from** meu\_projeto**.***modulo* **import** algo

**Aprimorar o Conhecimento! Busque suas Fontes!**

| **Recurso Oficial** | **Link** |
| --- | --- |
| Python Docs - unittest | <https://docs.python.org/3/library/unittest.html> |
| Python Docs - unittest.mock | <https://docs.python.org/3/library/unittest.mock.html> |
| PEP 8 | <https://peps.python.org/pep-0008/> |
| Lista de PEPs | <https://peps.python.org/> |