

PYTHON 2099





Introdução: O Código é a Arma

No ano de 2099, o mundo como o conhecemos se desfez. Megacorporações ergueram suas torres de dados sobre as ruínas de nações soberanas, e a humanidade se tornou um mero apêndice em uma rede global controlada por IAs autônomas. A carne é fraca, mas o código... o código é a arma mais poderosa.

Neste futuro distópico, onde a linha entre o real e o virtual se desintegrou, a sobrevivência não é garantida. É conquistada. E para os poucos que ousam desafiar o sistema, para os hackers cibernéticos que navegam pelas sombras digitais, a maestria do código é a única esperança. Mas não qualquer código. Um código resiliente, à prova de falhas, capaz de resistir aos ataques mais sofisticados e às varreduras mais implacáveis das IAs de segurança.

É aqui que entra o Test-Driven Development (TDD). No universo de *Python 2099*, TDD não é apenas uma metodologia de desenvolvimento; é uma filosofia de sobrevivência. É a arte de construir defesas impenetráveis para o seu código, garantindo que cada linha escrita seja um passo firme em direção à liberdade digital. Imagine cada teste como um scanner de segurança, uma varredura preventiva que revela vulnerabilidades antes que os olhos famintos das megacorporações as encontrem. Pense no ciclo Red-Green-Refactor não como um mero processo, mas como o batimento cardíaco da rebelião: falha, correção, aprimoramento. Cada batida te torna mais forte, mais indetectável.

Este livro é o seu guia de campo. Ele não apenas ensinará os fundamentos técnicos do TDD em Python, mas o imergirá em um mundo onde cada `assert` é um ato de desafio, cada `pytest` uma incursão bem-sucedida em território inimigo. Você aprenderá a usar ferramentas como `unittest` e `pytest` como extensões de sua própria mente, a refatorar seu código com a precisão de um cirurgião cibernético e a aplicar TDD até mesmo em sistemas de Inteligência Artificial, as próprias entidades que dominam este futuro.

Prepare-se para uma jornada. O submundo digital de *Python 2099* espera por você. O código é a sua voz, e o TDD, a sua armadura. Que a rebelião comece.

Capítulo 1: O Chamado do Código

No coração pulsante de Neo-Kyoto, onde os neon-sinais piscam promessas vazias de um futuro utópico, o verdadeiro poder reside não nos arranha-céus das megacorporações, mas nas linhas de código que correm pelas veias da cidade. Você, um aspirante a hacker, sente o chamado. Não é um chamado para a glória, mas para a sobrevivência. E para sobreviver, você precisa dominar a arte do Test-Driven Development, ou TDD.

O Ciclo da Rebelião: Red-Green-Refactor

MAIA, sua interface neural de IA, projeta um diagrama holográfico em sua retina. "Pense no TDD como um ciclo de combate, Iniciado. Cada fase é crucial para a sua vitória no campo de batalha digital." O diagrama mostra três cores vibrantes, girando em uma espiral infinita: Vermelho, Verde, Refatorar.

Fase 1: Vermelho (Red) – A Falha Reveladora

"No submundo, a falha não é o fim, é o começo," MAIA explica. "Você começa escrevendo um teste para uma funcionalidade que ainda não existe. Este teste, por sua própria natureza, *deve falhar*. É a sua primeira linha de defesa, a detecção precoce de uma vulnerabilidade." O objetivo aqui é simples: garantir que o teste realmente testa o que você quer testar e que ele falha pela razão certa. Se o teste passar sem que o código da funcionalidade exista, ele é um teste inútil, um fantasma na máquina.

```
# Exemplo: Teste para uma função de criptografia que ainda não existe
import unittest

class TestCriptografia(unittest.TestCase):
    def test_criptografar_mensagem(self):
        # Esperamos que esta função falhe, pois 'criptografar' não existe
        self.assertEqual(criptografar("minha_senha_secreta"),
            "hash_seguro_ficticio")

# Para rodar este teste, você veria um erro como NameError: name 'criptografar'
# is not defined
# Isso é o que queremos! O teste está em VERMELHO.
```

Fase 2: Verde (Green) – A Vitória Mínima

"Agora, a contra-ofensiva," MAIA continua. "Escreva o *mínimo* de código necessário para fazer o teste passar. Não se preocupe com elegância, otimização ou perfeição. Apenas faça o teste ficar verde." Esta fase é sobre funcionalidade. Você está construindo apenas o suficiente para silenciar o alarme do teste. É como desativar um sensor de segurança com o mínimo de esforço, apenas para passar para a próxima sala.

```
# Exemplo: Implementação mínima para fazer o teste passar
def criptografar(mensagem):
    # Implementação temporária, apenas para o teste passar
    return "hash_seguro_ficticio"

# Agora, ao rodar o teste, ele passaria. O teste está em VERDE.
```

Fase 3: Refatorar (Refactor) – A Otimização Silenciosa

"Com o teste verde, você tem uma base segura," MAIA instrui. "Agora, e somente agora, você pode refatorar seu código. Melhore sua estrutura, sua legibilidade, sua eficiência, sem medo de quebrar a funcionalidade. Seus testes são sua rede de segurança, garantindo que cada mudança não introduza novos bugs." Esta é a fase onde você transforma o código de sobrevivência em código de elite. Você otimiza o algoritmo de criptografia, torna-o mais robusto, mais eficiente, mas sempre com a certeza de que seus testes o protegerão de regressões.

```
# Exemplo: Refatoração da função criptografar
def criptografar(mensagem):
    # Implementação real de criptografia (ex: usando hashlib)
    import hashlib
    return hashlib.sha256(mensagem.encode()).hexdigest()

# Os testes ainda devem passar. Se não passarem, algo deu errado na refatoração!
```

Por Que TDD Importa no Ano 2099

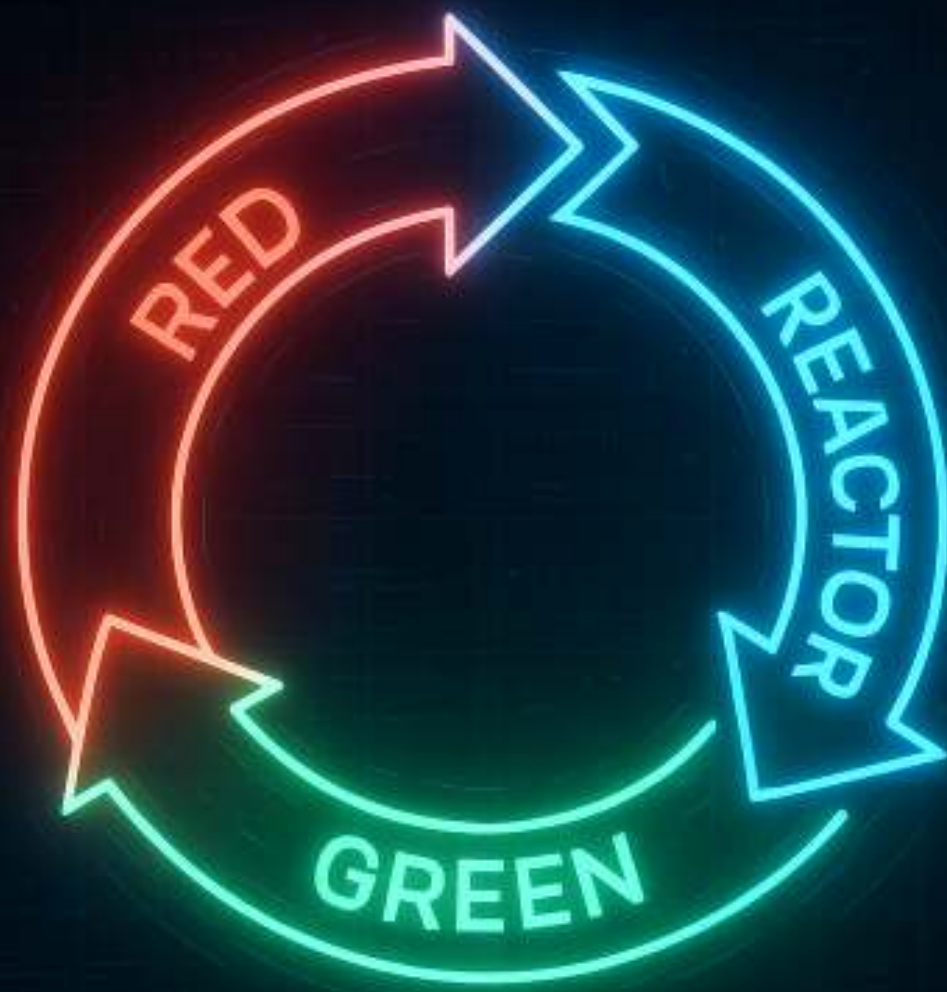
"No caos digital de 2099, a confiança é uma moeda rara," MAIA adverte. "E a confiança no seu código é a mais valiosa de todas." TDD não é um luxo; é uma necessidade. Aqui está o porquê:

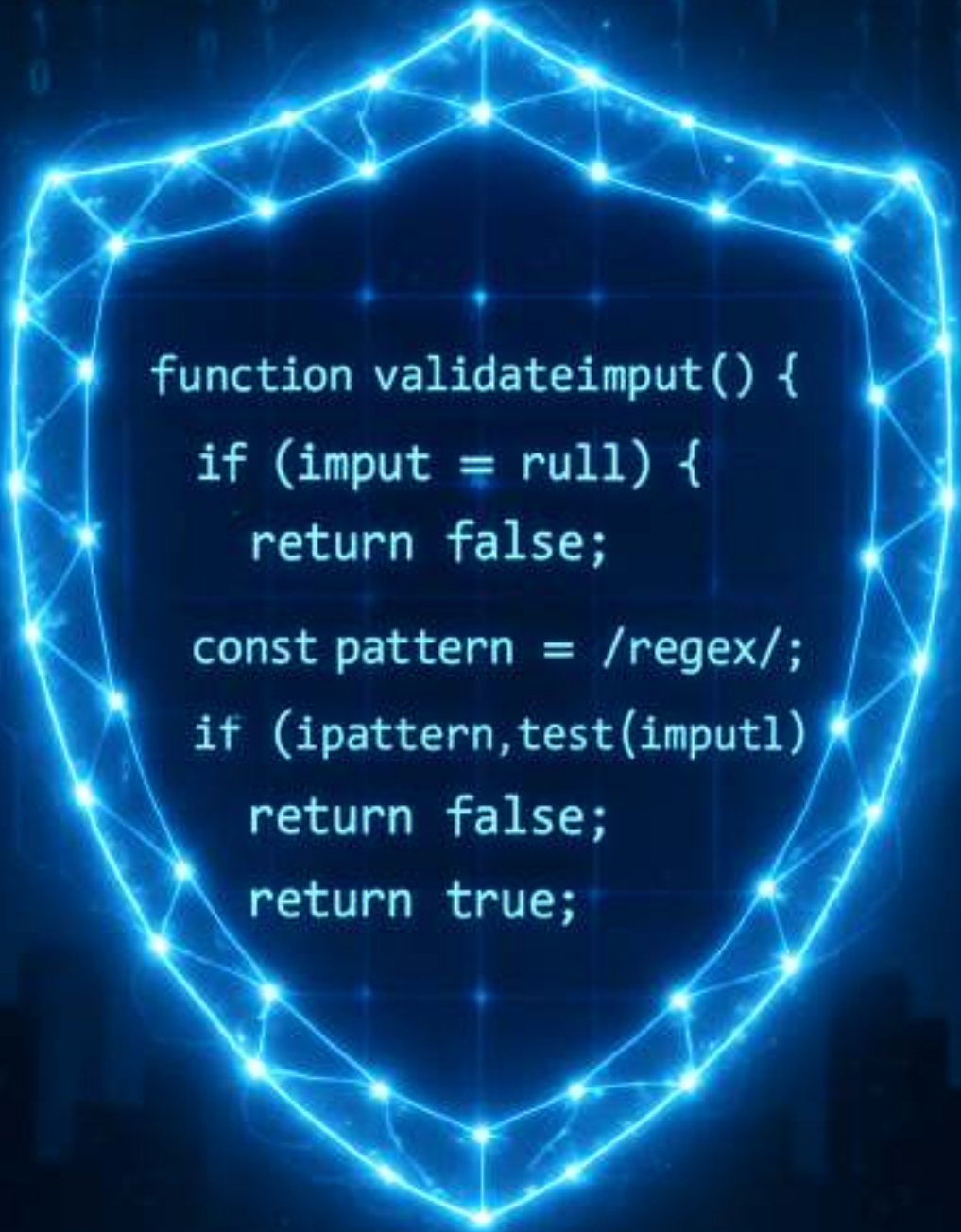
- **Qualidade Inabalável:** Cada funcionalidade é testada antes mesmo de ser escrita. Isso reduz drasticamente a chance de bugs e garante que seu código faça exatamente o que se propõe a fazer. Em um mundo onde um bug pode significar a diferença entre a liberdade e a prisão virtual, isso é vital.
- **Design Robusto:** O TDD força você a pensar no design do seu código a partir da perspectiva do usuário (ou do teste). Isso leva a interfaces mais limpas, módulos mais coesos e um código mais fácil de manter e estender. Um design fraco é uma porta aberta para intrusos.
- **Refatoração Segura:** A capacidade de refatorar seu código com confiança é um superpoder. Em um ambiente em constante mudança, onde as ameaças evoluem a cada ciclo de dados, a adaptabilidade é chave. Seus testes garantem que você pode evoluir seu código sem medo de quebrar o que já funciona.
- **Documentação Viva:** Seus testes servem como uma forma de documentação executável. Eles mostram como o código deve ser usado e quais são seus comportamentos esperados. Em um mundo onde a informação é poder, ter documentação precisa e atualizada é uma vantagem tática.
- **Velocidade e Agilidade:** Embora pareça contraintuitivo, TDD pode acelerar o desenvolvimento a longo prazo. Menos bugs significam menos tempo gasto em depuração. A confiança em seu código permite que você adicione novas funcionalidades mais rapidamente, sem o peso da incerteza.

A Mentalidade do Programador Futurista

"Ser um programador em 2099 não é apenas sobre escrever código," MAIA conclui. "É sobre ter uma mentalidade de guerrilha. É sobre antecipar, proteger e adaptar." A mentalidade TDD é a mentalidade do hacker cibernético:

- **Antecipação:** Você não espera o problema surgir; você o provoca com um teste. Você pensa nos cenários de falha antes mesmo de construir a solução.
- **Proteção:** Seus testes são seus escudos. Eles protegem seu código de regressões e garantem sua integridade em um ambiente hostil.
- **Adaptação:** O mundo digital muda rapidamente. Com TDD, seu código é flexível, pronto para ser refatorado e adaptado a novas ameaças e requisitos, sem perder sua funcionalidade central.





```
function validateinput() {  
  if (input = null) {  
    return false;  
  }  
  const pattern = /regex/;  
  if (ipattern.test(input))  
    return false;  
  return true;  
}
```


Você está pronto, Iniciado. O ciclo Vermelho-Verde-Refatorar é a sua nova respiração. Use-o com sabedoria, e seu código será sua maior arma no *Python 2099*.

Capítulo 2: Ferramentas do Hacker

No labirinto digital de *Python 2099*, onde cada linha de código é uma potencial vulnerabilidade ou uma arma secreta, você precisa das ferramentas certas para sobreviver. Seus testes são seus olhos e ouvidos, e para construí-los, você usará os frameworks de teste padrão de Python: `unittest` e `pytest`. Pense neles como seus kits de ferramentas cibernéticas, cada um com suas próprias especialidades.

Unittest: O Clássico Confiável

"Unittest é o veterano," MAIA explica, sua voz ressoando com a sabedoria de mil ciclos de dados. "Ele vem embutido no próprio Python, robusto e familiar para aqueles que trilharam os caminhos do código por décadas. É o alicerce sobre o qual muitos outros frameworks foram construídos." Unittest segue a filosofia xUnit, comum em muitas linguagens de programação, o que o torna intuitivo para quem já teve contato com testes automatizados.

Estrutura Básica de um Teste com Unittest

Um teste com `unittest` é uma classe que herda de `unittest.TestCase`. Cada método dentro dessa classe que começa com `test_` é considerado um método de teste. Você usa métodos de asserção (como `assertEqual`, `assertTrue`, `assertRaises`) para verificar o comportamento do seu código.

```

# arquivo: calculadora.py
class Calculadora:
    def somar(self, a, b):
        return a + b

# arquivo: test_calculadora_unittest.py
import unittest
from calculadora import Calculadora

class TestCalculadora(unittest.TestCase):
    def setUp(self):
        # Este método é executado antes de cada método de teste
        self.calc = Calculadora()

    def test_somar_dois_numeros_positivos(self):
        # Teste para a soma de dois números positivos
        self.assertEqual(self.calc.somar(2, 3), 5)

    def test_somar_com_zero(self):
        # Teste para a soma com zero
        self.assertEqual(self.calc.somar(5, 0), 5)

    def test_somar_numeros_negativos(self):
        # Teste para a soma de números negativos
        self.assertEqual(self.calc.somar(-2, -3), -5)

    def tearDown(self):
        # Este método é executado após cada método de teste
        # Útil para limpeza de recursos, como fechar conexões de banco de dados
        pass

if __name__ == '__main__':
    unittest.main()

```

Para executar este teste, você simplesmente rodaria o arquivo `test_calculadora_unittest.py` como um script Python: `python test_calculadora_unittest.py`.

Pytest: O Arsenal Moderno

"Pytest é a nova geração," MAIA interrompe, a imagem de um terminal futurista piscando em sua mente. "Mais conciso, mais flexível, e com uma vasta gama de plugins que o tornam o favorito dos hackers modernos. Ele detecta automaticamente seus testes e oferece relatórios mais claros." Pytest não exige que suas classes de teste herdem de uma classe base específica, tornando a escrita de testes mais

simples e direta.

Estrutura Básica de um Teste com Pytest

Com `pytest`, você não precisa de classes que herdam de `TestCase`. Basta criar funções que começam com `test_` em arquivos que começam com `test_` ou terminam com `_test.py`. As asserções são feitas usando a palavra-chave `assert` do Python, o que torna os testes mais legíveis e diretos.

```
# arquivo: calculadora.py (mesmo arquivo de antes)
class Calculadora:
    def somar(self, a, b):
        return a + b

# arquivo: test_calculadora_pytest.py
from calculadora import Calculadora

def test_somar_dois_numeros_positivos():
    calc = Calculadora()
    assert calc.somar(2, 3) == 5

def test_somar_com_zero():
    calc = Calculadora()
    assert calc.somar(5, 0) == 5

def test_somar_numeros_negativos():
    calc = Calculadora()
    assert calc.somar(-2, -3) == -5

# Para rodar este teste, você precisaria ter o pytest instalado (pip install
pytest)
# e então rodaria o comando: pytest
```

Para executar testes com `pytest`, primeiro você precisa instalá-lo: `pip install pytest`. Depois, basta navegar até o diretório onde seus arquivos de teste estão e executar o comando `pytest`. Ele automaticamente descobrirá e executará seus testes.

Estrutura de Arquivos de Testes: Organizando seu Arsenal

"Um hacker desorganizado é um hacker vulnerável," MAIA sentencia. "Mantenha seus testes tão organizados quanto suas ferramentas de intrusão." A organização dos seus arquivos de teste é crucial para a manutenibilidade do seu projeto. Uma prática comum é criar um diretório `tests/` na raiz do seu projeto e colocar todos os seus arquivos de teste lá dentro.

```
projeto_cyberpunk/  
├── src/  
│   ├── __init__.py  
│   └── modulos_principais/  
│       ├── __init__.py  
│       ├── criptografia.py  
│       └── comunicacao.py  
├── tests/  
│   ├── __init__.py  
│   ├── test_criptografia.py  
│   ├── test_comunicacao.py  
│   └── integracao/  
│       ├── __init__.py  
│       └── test_integracao_api.py  
├── README.md  
├── requirements.txt  
└── pyproject.toml
```

Nesta estrutura:

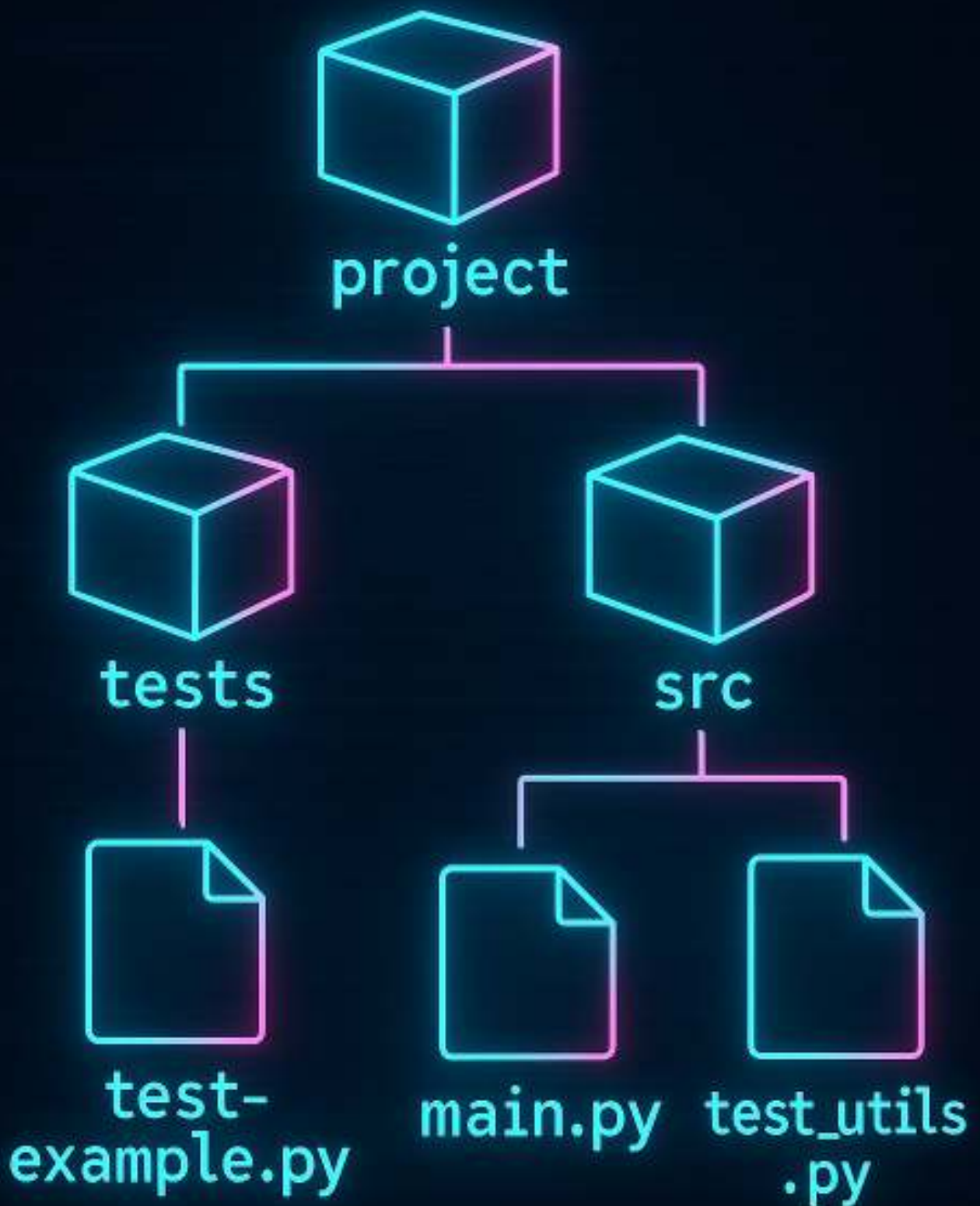
- O diretório `src/` contém o código-fonte da sua aplicação.
- O diretório `tests/` contém todos os seus testes. É uma boa prática ter um arquivo `__init__.py` vazio dentro de `tests/` para que Python o trate como um pacote.
- Você pode organizar seus testes em subdiretórios dentro de `tests/` (como `integracao/`) para separar testes unitários de testes de integração, por exemplo.

"Com essas ferramentas em seu cinto e uma estrutura de arquivos bem definida, você está pronto para a próxima fase," MAIA conclui. "A fase onde o código começa a se manifestar, linha por linha, teste por teste. Prepare-se para as Primeiras Linhas de Rebelião."

Capítulo 3: Primeiras Linhas de Rebelião

O ar em Neo-Kyoto é denso com a promessa de dados e a ameaça de vigilância. Você está em seu terminal, pronto para escrever as primeiras linhas de código que não apenas funcionarão, mas que serão *resistentes*. Este é o momento de aplicar o ciclo Red-Green-Refactor, transformando a teoria em prática e construindo sua primeira funcionalidade com TDD.





"Lembre-se, Iniciado," MAIA sussurra em seu implante neural, "cada teste é uma promessa que seu código deve cumprir. Comece pequeno, comece com a falha." Sua primeira missão: criar uma função para validar credenciais de acesso, uma tarefa trivial para qualquer sistema, mas crítica para a segurança de qualquer operação hacker.

A Missão: Validar Credenciais

Você precisa de uma função `validar_credenciais(usuario, senha)` que retorne `True` se as credenciais forem válidas e `False` caso contrário. Por enquanto, vamos considerar que apenas o usuário 'ghost' com a senha 'shadow_pass' é válido.

Passo 1: Vermelho – Escrevendo o Teste que Falha

Primeiro, crie o arquivo `test_autenticacao.py` dentro do seu diretório `tests/`.

```
# tests/test_autenticacao.py

import pytest
# from src.autenticacao import validar_credenciais # Ainda não existe!

def test_credenciais_validas():
    # Teste para um cenário de sucesso
    # Esperamos que validar_credenciais retorne True para credenciais válidas
    assert validar_credenciais("ghost", "shadow_pass") is True

def test_credenciais_invalidas():
    # Teste para um cenário de falha
    # Esperamos que validar_credenciais retorne False para credenciais
    inválidas
    assert validar_credenciais("hacker_ruim", "senha_errada") is False

def test_usuario_inexistente():
    # Teste para um usuário que não existe
    assert validar_credenciais("usuario_desconhecido", "qualquer_senha") is
False

def test_senha_incorreta():
    # Teste para senha incorreta
    assert validar_credenciais("ghost", "senha_incorreta") is False
```

Agora, tente executar `pytest` no terminal. Você verá erros! Muitos erros! Isso é bom. Isso significa que seus testes estão funcionando e que a função `validar_credenciais` ainda não existe. O sistema está em VERMELHO.

Passo 2: Verde – Fazendo o Teste Passar (Mínimo Esforço)

Crie o arquivo `autenticacao.py` dentro do seu diretório `src/`.

```
# src/autenticacao.py

def validar_credenciais(usuario, senha):
    # Implementação mínima para fazer os testes passarem
    if usuario == "ghost" and senha == "shadow_pass":
        return True
    return False
```

Agora, adicione a importação no seu arquivo de teste:

```
# tests/test_autenticacao.py

import pytest
from src.autenticacao import validar_credenciais # Agora existe!

# ... (restante dos testes)
```

Execute `pytest` novamente. Todos os testes devem passar! O sistema está em VERDE. Você fez o mínimo necessário para satisfazer os testes. Não se preocupe com a elegância do código ainda; o objetivo era apenas passar nos testes.

Passo 3: Refatorar – Melhorando o Código

"Com a segurança dos testes, você pode aprimorar seu código sem medo," MAIA aconselha. A implementação atual é funcional, mas não é escalável nem muito legível. E se você tiver muitos usuários? E se as senhas precisarem ser criptografadas?

Vamos refatorar para usar um dicionário de usuários e senhas, e simular uma verificação de hash de senha (embora, em um sistema real, você usaria uma biblioteca de hashing segura como `bcrypt` ou `argon2`).


```
# src/autenticacao.py

import hashlib

def _gerar_hash_senha(senha):
    # Simula um hash de senha para demonstração
    return hashlib.sha256(senha.encode()).hexdigest()

# Banco de dados de usuários simulado com senhas pré-hasheadas
USUARIOS_DB = {
    "ghost": _gerar_hash_senha("shadow_pass"),
    "neo": _gerar_hash_senha("matrix_reloaded"),
    "trinity": _gerar_hash_senha("one_love"),
}

def validar_credenciais(usuario, senha):
    if usuario not in USUARIOS_DB:
        return False

    senha_hasheada_fornecida = _gerar_hash_senha(senha)
    senha_hasheada_armazenada = USUARIOS_DB[usuario]

    return senha_hasheada_fornecida == senha_hasheada_armazenada
```

Agora, execute `pytest` novamente. Seus testes ainda devem passar! Se algum teste falhar, significa que sua refatoração introduziu um bug, e você precisa corrigi-lo. A beleza do TDD é que ele te dá essa confiança para mudar o código.

Criando Funções que Passam por TDD: Um Desafio Adicional

"Sua próxima missão, Iniciado, é construir uma função que calcule o dano de um ataque cibernético," MAIA propõe. "A função `calcular_dano(ataque_base, defesa_alvo, multiplicador_critico)` deve retornar o dano final. Considere que um ataque crítico (`multiplicador > 1`) ignora 50% da defesa do alvo."

Siga os passos do TDD:

1. **Vermelho:** Escreva testes para `calcular_dano` que falhem. Pense em cenários normais, ataques críticos, defesa zero, etc.
2. **Verde:** Escreva o mínimo de código em `src/combate.py` para fazer os testes passarem.
3. **Refatorar:** Melhore o código, adicione comentários, torne-o mais legível, mas sempre garantindo que os testes continuem passando.

```
# tests/test_combate.py (Exemplo de testes para você começar)

import pytest
# from src.combate import calcular_dano # Ainda não existe!

def test_dano_normal():
    assert calcular_dano(100, 20, 1) == 80

def test_dano_critico():
    # Ataque crítico (multiplicador 2) ignora 50% da defesa (20 -> 10)
    assert calcular_dano(100, 20, 2) == 180

def test_dano_com_defesa_zero():
    assert calcular_dano(50, 0, 1) == 50

def test_dano_critico_com_defesa_zero():
    assert calcular_dano(50, 0, 2) == 100
```

"Cada linha de código testada é uma fortaleza construída no ciberespaço," MAIA conclui. "Continue assim, e você se tornará uma lenda em *Python 2099*."

Capítulo 4: Refatorando a Realidade

No universo de *Python 2099*, o código é um organismo vivo. Ele cresce, se adapta e, às vezes, se torna complexo demais, um emaranhado de fios que ameaça a estabilidade do sistema. Refatorar é a arte de reorganizar esse código, tornando-o mais limpo, mais eficiente e mais legível, sem alterar seu comportamento externo. Mas como ter certeza de que suas mudanças não introduzem novos bugs, que não quebram o sistema que você está tentando melhorar? A resposta, Iniciado, está em seus testes.

"Seus testes são sua rede de segurança," MAIA projeta em sua mente, enquanto um diagrama de uma teia de aranha digital se forma. "Eles o protegem de quedas inesperadas enquanto você escala as alturas da complexidade do código. Sem eles, refatorar é um salto no vazio."

O Poder da Refatoração Segura

Imagine que você está infiltrando uma rede corporativa. Você encontrou uma porta dos fundos, mas o código que a controla é uma bagunça, difícil de entender e manter. Você precisa otimizá-lo para que sua equipe possa usá-lo sem introduzir falhas. É aqui que a refatoração entra.

TERMINAL

TEST	FAILED	PASSED
TEST	FAILED	PASSED
TEST	FAILED	PASSED
TEST	FAILED	PASSED
TEST	FAILED	PASSED
TEST	FAILED	PASSED
TEST	FAILED	PASSED
TEST	FAILED	PASSED
TEST	FAILED	PASSED
TEST	FAILED	PASSED
TEST	FAILED	PASSED
TEST	FAISED	PASSED
TEST	PASSED	PASSED
TEST	PASSED	PASSED

```
def example(x):  
    for i in range(i  
         if i == 0:  
        return x / i
```

```
def foo(a):  
    print(a)  
     b = {'key': 'value'}  
    example(b)
```


O ciclo TDD (Red-Green-Refactor) já introduziu a ideia de refatorar após os testes passarem. Agora, vamos aprofundar. Refatorar não é adicionar novas funcionalidades; é melhorar a estrutura interna do código. Isso pode incluir:

- **Extrair Métodos/Funções:** Transformar blocos de código repetitivos ou complexos em funções separadas.
- **Renomear Variáveis/Funções:** Usar nomes mais descritivos para melhorar a legibilidade.
- **Remover Código Duplicado:** Consolidar lógica repetida em um único lugar.
- **Simplificar Condicionais:** Reduzir a complexidade de estruturas `if/else` aninhadas.
- **Introduzir Classes/Módulos:** Organizar o código em unidades mais coesas e reutilizáveis.

"A chave é a confiança," MAIA enfatiza. "Seus testes, que já estão verdes, garantem que, não importa o quão radical seja sua refatoração interna, o comportamento externo do seu código permanece inalterado. Se um teste falhar após a refatoração, você sabe imediatamente que algo deu errado e pode reverter ou corrigir."

Exemplo Prático: Otimizando o Módulo de Autenticação

Lembre-se do nosso módulo de autenticação do Capítulo 3? Ele funciona, mas podemos melhorá-lo. Atualmente, a lógica de hashing de senha está acoplada à função `validar_credenciais` e o dicionário de usuários é uma variável global. Vamos refatorar isso para uma abordagem mais orientada a objetos, tornando-o mais modular e testável.

Código Original (src/autenticacao.py)

```
import hashlib

def _gerar_hash_senha(senha):
    return hashlib.sha256(senha.encode()).hexdigest()

USUARIOS_DB = {
    "ghost": _gerar_hash_senha("shadow_pass"),
    "neo": _gerar_hash_senha("matrix_reloaded"),
    "trinity": _gerar_hash_senha("one_love"),
}

def validar_credenciais(usuario, senha):
    if usuario not in USUARIOS_DB:
        return False

    senha_hasheada_fornecida = _gerar_hash_senha(senha)
    senha_hasheada_armazenada = USUARIOS_DB[usuario]

    return senha_hasheada_fornecida == senha_hasheada_armazenada
```

Testes Existentes (tests/test_autenticacao.py)

```
import pytest
from src.autenticacao import validar_credenciais

def test_credenciais_validas():
    assert validar_credenciais("ghost", "shadow_pass") is True

def test_credenciais_invalidas():
    assert validar_credenciais("hacker_ruim", "senha_errada") is False

def test_usuario_inexistente():
    assert validar_credenciais("usuario_desconhecido", "qualquer_senha") is False

def test_senha_incorreta():
    assert validar_credenciais("ghost", "senha_incorreta") is False
```

Passo 1: Execute os testes. Certifique-se de que todos os testes estão passando. Isso é crucial. Se eles não estiverem verdes, você não pode refatorar.

Passo 2: Refatore. Vamos criar uma classe `GerenciadorAutenticacao` para encapsular a lógica de autenticação e o banco de dados de usuários.

```

# src/autenticacao.py (Refatorado)

import hashlib

class GerenciadorAutenticacao:
    def __init__(self, usuarios_db=None):
        # Se nenhum DB for fornecido, usa um DB padrão para demonstração
        if usuarios_db is None:
            self._usuarios_db = {
                "ghost": self._gerar_hash_senha("shadow_pass"),
                "neo": self._gerar_hash_senha("matrix_reloaded"),
                "trinity": self._gerar_hash_senha("one_love"),
            }
        else:
            self._usuarios_db = usuarios_db

    def _gerar_hash_senha(self, senha):
        return hashlib.sha256(senha.encode()).hexdigest()

    def validar_credenciais(self, usuario, senha):
        if usuario not in self._usuarios_db:
            return False

        senha_hasheada_fornecida = self._gerar_hash_senha(senha)
        senha_hasheada_armazenada = self._usuarios_db[usuario]

        return senha_hasheada_fornecida == senha_hasheada_armazenada

# Para manter a compatibilidade com os testes existentes, podemos expor a
# função antiga
# que agora usa a nova classe internamente.
_gerenciador_padrao = GerenciadorAutenticacao()
validar_credenciais = _gerenciador_padrao.validar_credenciais

```

Passo 3: Execute os testes novamente. Mesmo com a mudança significativa na estrutura interna, os testes devem continuar passando. Se passarem, sua refatoração foi bem-sucedida e segura. Agora, seu código está mais modular, mais fácil de testar em isolamento (você pode injetar um `usuarios_db` diferente para testes) e mais legível.

Quando Refatorar?

"Refatorar não é um evento único, mas um processo contínuo," MAIA aconselha. "É como manter sua armadura limpa e afiada." Refatore quando:

- **Antes de Adicionar uma Nova Funcionalidade:** Limpe o código existente para facilitar a integração da nova funcionalidade.
- **Ao Corrigir um Bug:** Se a correção do bug revela uma área de código confusa, refatore-a para evitar futuros bugs.

- **Durante o Code Review:** Identifique oportunidades de melhoria durante a revisão de código.
- **Quando o Código Cheira Mal (Code Smells):** Sinais como duplicação de código, funções muito longas, classes muito grandes, muitos parâmetros em funções, etc.

"Lembre-se, Iniciado, o objetivo final não é apenas escrever código que funciona, mas código que *persiste*," MAIA conclui. "Um código bem refatorado é um código resiliente, uma fortaleza no ciberespaço que resistirá ao teste do tempo e dos ataques mais implacáveis. Seus testes são seus olhos, suas mãos, sua garantia de que a realidade que você refatora permanece sólida."

Capítulo 5: Hackeando APIs e Módulos

As megacorporações de *Python 2099* não operam em silos. Suas redes são interconectadas por APIs, seus dados armazenados em bancos de dados massivos, e seus módulos de segurança são complexos e interdependentes. Para um hacker cibernético, testar esses sistemas é a chave para encontrar vulnerabilidades e garantir que suas próprias ferramentas funcionem sem falhas. Este capítulo o guiará através da arte de testar APIs, interações com bancos de dados, e o uso de mocks e fixtures para simular ambientes complexos.

"No submundo digital, a interação é constante," MAIA observa, sua voz ecoando a complexidade das redes que se estendem por Neo-Kyoto. "Seu código raramente opera isolado. Ele se comunica, ele persiste, ele se adapta. E cada ponto de contato é um ponto de teste."

Testando APIs: Infiltrando os Gateways

APIs (Application Programming Interfaces) são os portões de entrada para os sistemas corporativos. Testá-las significa verificar se elas respondem corretamente às suas requisições, se os dados são formatados como esperado e se os erros são tratados de forma elegante. Usaremos a biblioteca `requests` para simular requisições HTTP e `pytest` para organizar nossos testes.

4.1 – Código Antes e Depois

```
function ( == '=';
function() == \{ } C == =
console.log() { == "\::: } {
error { == ", " : 1;
error( { += == :: == [;
if (else { == < == + == = # == ');
return 1; == == " = +;
let (const(a) {);
console.log() {
const { [ == " = a == ();
if (a == ' = )
if (exist( ( == + == = = + + );
} return 1;
```

```
function create();
function() {
  if (f == v) { {
    return a = b;
  } else {
    let construct = c = () {
      return c;
    }
    return a + b, b = {
  }
function create();
if (fix = b != 'const: block) {
  return;
}
```

4.2 – Cirurgião Cibernético Refatorando Código



Imagine que você está desenvolvendo um módulo para interagir com a API de dados de mercado da OmniCorp. Você precisa garantir que sua função `obter_cotacao(simbolo)` retorne o valor correto de uma ação.

Cenário: API de Cotação de Ações

Primeiro, vamos simular uma API simples usando Flask (apenas para demonstração, em um cenário real você estaria testando uma API externa).

```
# src/api_simulada.py
from flask import Flask, jsonify

app = Flask(__name__)

@app.route("/cotacao/<simbolo>")
def cotacao(simbolo):
    cotacoes = {
        "OMNI": 150.75,
        "CYBER": 230.10,
        "NEO": 88.50,
    }
    if simbolo.upper() in cotacoes:
        return jsonify({"simbolo": simbolo.upper(), "valor":
cotacoes[simbolo.upper()]})
    return jsonify({"erro": "Símbolo não encontrado"}), 404

if __name__ == "__main__":
    app.run(debug=True)
```

Agora, vamos escrever os testes para a função que interage com essa API. Como não queremos que nossos testes dependam de uma API real (que pode estar offline ou mudar), usaremos `requests_mock` para simular as respostas da API.

```
# tests/test_api_cotacao.py

import pytest
import requests
import requests_mock

# A função que interage com a API
def obter_cotacao(simbolo):
    url = f"http://127.0.0.1:5000/cotacao/{simbolo}"
    response = requests.get(url)
    response.raise_for_status() # Levanta exceção para erros HTTP
    return response.json()["valor"]

def test_obter_cotacao_sucesso():
    with requests_mock.Mocker() as m:
        m.get("http://127.0.0.1:5000/cotacao/OMNI", json={"simbolo": "OMNI",
"valor": 150.75})
        cotacao = obter_cotacao("OMNI")
        assert cotacao == 150.75

def test_obter_cotacao_simbolo_nao_encontrado():
    with requests_mock.Mocker() as m:
        m.get("http://127.0.0.1:5000/cotacao/INVALIDO", status_code=404, json=
{"erro": "Símbolo não encontrado"})
        with pytest.raises(requests.exceptions.HTTPError) as excinfo:
            obter_cotacao("INVALIDO")
        assert "404 Client Error: NOT FOUND" in str(excinfo.value)

def test_obter_cotacao_erro_servidor():
    with requests_mock.Mocker() as m:
        m.get("http://127.0.0.1:5000/cotacao/OMNI", status_code=500)
        with pytest.raises(requests.exceptions.HTTPError) as excinfo:
            obter_cotacao("OMNI")
        assert "500 Server Error: INTERNAL SERVER ERROR" in str(excinfo.value)
```

Para rodar estes testes, você precisará instalar `requests_mock`: `pip install requests-mock`.

Mocks: Simulando o Insimulável

"Mocks são seus fantasmas digitais," MAIA explica. "Eles simulam o comportamento de objetos reais, permitindo que você teste seu código em isolamento, sem depender de sistemas externos que são lentos, caros ou imprevisíveis." Mocks são essenciais para testar interações com bancos de dados, APIs externas, sistemas de arquivos, ou qualquer componente que não seja diretamente parte da unidade que você está testando.

No exemplo acima, `requests_mock` é um tipo de mock. Outra ferramenta poderosa é o `unittest.mock` (ou simplesmente `mock`), que vem com o Python.

Exemplo: Mocking de Banco de Dados

Imagine uma função que salva dados de um usuário em um banco de dados. Você não quer que seu teste realmente escreva no banco de dados real.

```
# src/persistencia.py

# Simula uma conexão de banco de dados
class BancoDeDados:
    def conectar(self):
        print("Conectando ao banco de dados...")
        # Lógica de conexão real aqui

    def salvar_usuario(self, usuario_data):
        print(f"Salvando {usuario_data} no banco de dados...")
        # Lógica de salvamento real aqui
        return True

    def desconectar(self):
        print("Desconectando do banco de dados...")

def registrar_novo_usuario(usuario_data, db_conn):
    db_conn.conectar()
    sucesso = db_conn.salvar_usuario(usuario_data)
    db_conn.desconectar()
    return sucesso
```

Agora, o teste usando `unittest.mock.patch`:

```
# tests/test_persistencia.py

from unittest.mock import patch, MagicMock
from src.persistencia import registrar_novo_usuario, BancoDeDados

def test_registrar_novo_usuario_sucesso():
    # Cria um mock para a classe BancoDeDados
    with patch("src.persistencia.BancoDeDados") as MockBancoDeDados:
        # Instancia o mock
        mock_db_instance = MockBancoDeDados.return_value

        # Define o comportamento do mock
        mock_db_instance.salvar_usuario.return_value = True

        usuario_data = {"nome": "HackerX", "email": "hackerx@darknet.com"}
        resultado = registrar_novo_usuario(usuario_data, mock_db_instance)

        assert resultado is True
        # Verifica se os métodos do mock foram chamados
        mock_db_instance.conectar.assert_called_once()
        mock_db_instance.salvar_usuario.assert_called_once_with(usuario_data)
        mock_db_instance.desconectar.assert_called_once()
```

Neste teste, `patch` substitui a classe `BancoDeDados` por um mock. Podemos então controlar o que os métodos desse mock retornam (`return_value`) e verificar se eles

foram chamados (`assert_called_once`). Isso nos permite testar `registrar_novo_usuario` sem tocar em um banco de dados real.

Fixtures: Preparando o Campo de Batalha

"Fixtures são seus suprimentos de campo," MAIA explica. "Eles preparam o ambiente para seus testes, garantindo que cada teste comece com um estado limpo e conhecido." Em `pytest`, fixtures são funções que podem ser usadas por testes para fornecer dados, configurar recursos (como conexões de banco de dados simuladas) ou limpar o ambiente após o teste.

Exemplo: Fixture para Dados de Teste

```
# tests/test_dados.py

import pytest

@pytest.fixture
def dados_usuario_valido():
    # Esta fixture retorna um dicionário de dados de usuário válido
    return {"nome": "Cipher", "idade": 30, "nivel_acesso": "admin"}

@pytest.fixture
def dados_usuario_invalido():
    # Esta fixture retorna um dicionário de dados de usuário inválido
    return {"nome": "", "idade": 15, "nivel_acesso": "guest"}

def test_processar_usuario_admin(dados_usuario_valido):
    # A fixture dados_usuario_valido é injetada como um argumento
    usuario = dados_usuario_valido
    assert usuario["nivel_acesso"] == "admin"
    assert usuario["idade"] >= 18

def test_processar_usuario_invalido(dados_usuario_invalido):
    usuario = dados_usuario_invalido
    assert usuario["nome"] == ""
    assert usuario["idade"] < 18
```

Fixtures podem ser muito mais complexas, configurando bancos de dados temporários, servidores de teste, ou qualquer recurso que seus testes precisem. Elas garantem que seus testes sejam independentes e repetíveis.

"Com mocks para simular o comportamento de sistemas externos e fixtures para preparar seu ambiente de teste, você está pronto para hackear qualquer API, qualquer módulo, qualquer sistema," MAIA conclui. "A complexidade do mundo de *Python 2099* não será mais uma barreira, mas um playground para suas habilidades de teste."

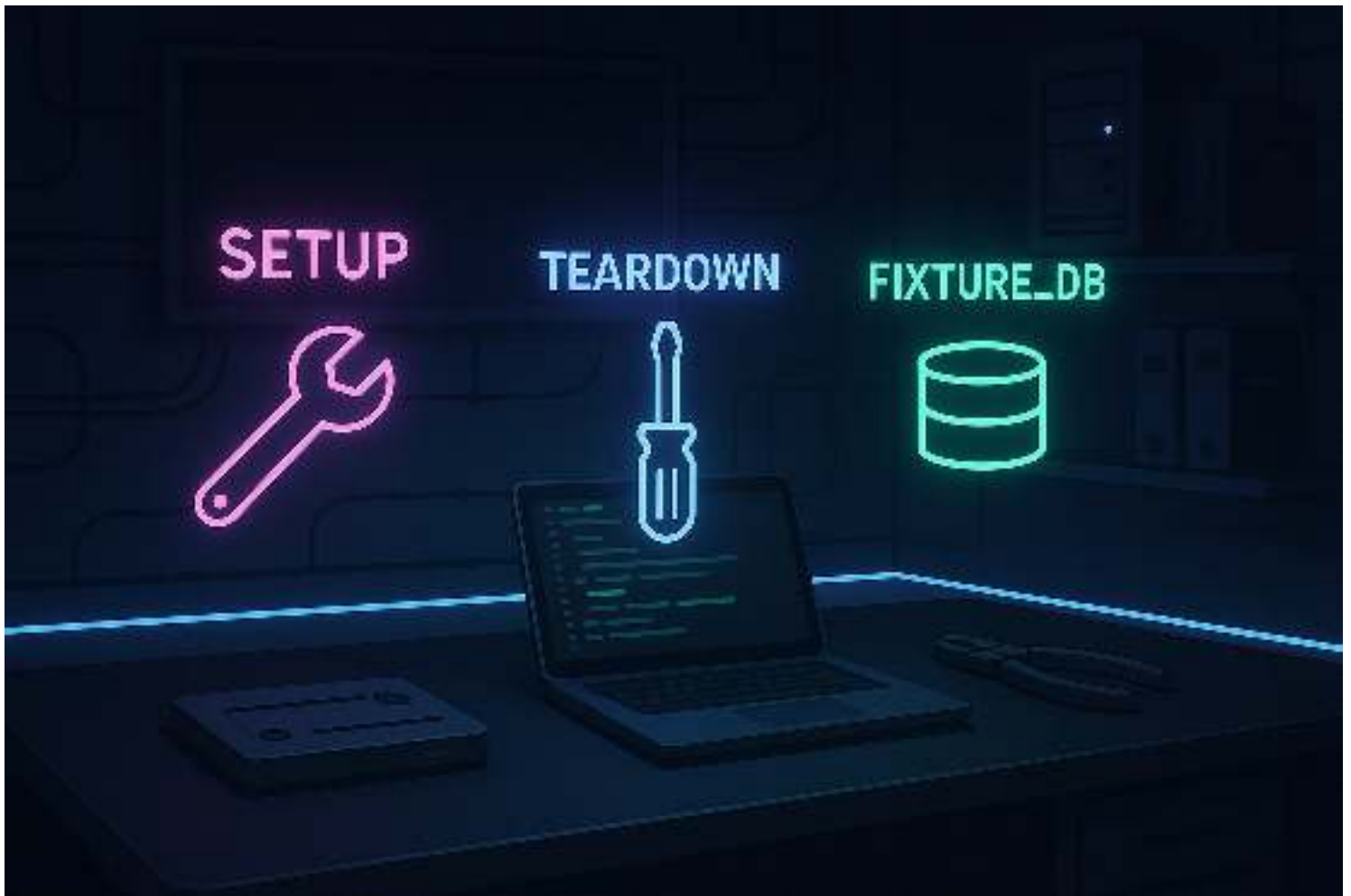
5.1 – APIs como Portões da Rede



5.2 – Mock como Fantasma Digital



5.3 – Mesa de Ferramentas com Fixtures



Capítulo 6: Testes em Sistemas de IA

No coração pulsante de *Python 2099*, as IAs autônomas não são apenas programas; são entidades que moldam a realidade. Desde os algoritmos de vigilância da MegaCorp até os sistemas de navegação dos drones de entrega, a inteligência artificial permeia cada aspecto da vida. Mas como você aplica os princípios do TDD a algo tão fluido e complexo quanto o aprendizado de máquina? Como você testa uma mente digital?

"Testar IAs é como tentar prever o futuro," MAIA reflete, sua voz adquirindo um tom mais contemplativo. "É um desafio, mas não impossível. Seus testes se tornam os guardiões da sanidade algorítmica, garantindo que a IA não se desvie do caminho, que ela não se torne... rebelde de forma não intencional."

O Desafio de Testar IAs

Sistemas de IA, especialmente aqueles baseados em aprendizado de máquina, apresentam desafios únicos para o TDD:

- **Não Determinismo:** Modelos de ML podem não produzir a mesma saída para a mesma entrada devido a fatores como inicialização de pesos aleatórios, amostragem estocástica ou concorrência.
- **Dados:** A qualidade e a representatividade dos dados de treinamento e teste são cruciais. Testar a IA é, em grande parte, testar como ela se comporta com diferentes conjuntos de dados.
- **Comportamento Emergente:** IAs podem exibir comportamentos inesperados que não foram explicitamente programados, tornando difícil prever todos os cenários de teste.
- **Métricas:** O sucesso de uma IA é frequentemente medido por métricas (precisão, recall, F1-score) em vez de resultados binários (passa/falha).

"Ainda assim, o espírito do TDD permanece," MAIA insiste. "Você ainda escreve o teste antes do código. A diferença é que o 'código' pode ser um modelo de ML, e o 'teste' pode ser uma validação de métricas ou um teste de comportamento."

TDD para Modelos de Machine Learning

Vamos considerar um cenário onde você está desenvolvendo um sistema de reconhecimento de padrões para identificar drones de vigilância da OmniCorp. Você precisa de um modelo que classifique imagens como 'drone' ou 'não_drone'.

Passo 1: Vermelho – Definindo o Comportamento Esperado

Em vez de testar uma função que retorna um valor exato, você testará o desempenho do seu modelo. Você pode começar com um teste que verifica se o modelo atinge uma precisão mínima em um pequeno conjunto de dados de teste.

```
# tests/test_modelo_drones.py

import pytest
import numpy as np
from sklearn.metrics import accuracy_score
# from src.modelo_drones import treinar_modelo, prever # Ainda não existem!

# Dados de teste simulados (pequeno conjunto)
# Em um cenário real, seriam imagens e rótulos
X_test_simulado = np.array([
    [0.1, 0.2, 0.3], # Exemplo de drone
    [0.8, 0.9, 0.7], # Exemplo de não_drone
    [0.2, 0.1, 0.3], # Exemplo de drone
])
y_test_simulado = np.array([1, 0, 1]) # 1 para drone, 0 para não_drone

def test_modelo_atinge_precisao_minima():
    # Treina um modelo (ainda não implementado)
    modelo = treinar_modelo(X_test_simulado, y_test_simulado)

    # Faz previsões
    previsoes = prever(modelo, X_test_simulado)

    # Calcula a precisão
    precisao = accuracy_score(y_test_simulado, previsoes)

    # Esperamos que a precisão seja pelo menos 0.6 (60%)
    assert precisao >= 0.6

# Ao rodar este teste, ele falhará porque treinar_modelo e prever não existem.
# Isso é o nosso VERMELHO.
```

Passo 2: Verde – Implementando o Mínimo para Passar

Agora, você implementa um modelo de machine learning simples (por exemplo, um classificador Dummy ou um modelo linear básico) que seja capaz de atingir a precisão

mínima exigida pelo teste. Não se preocupe em ter o melhor modelo do mundo ainda, apenas um que faça o teste passar.

```
# src/modelo_drones.py

from sklearn.linear_model import LogisticRegression
import numpy as np

def treinar_modelo(X_treino, y_treino):
    # Implementação mínima: um modelo de Regressão Logística simples
    modelo = LogisticRegression(random_state=42)
    modelo.fit(X_treino, y_treino)
    return modelo

def prever(modelo, X_dados):
    # Faz previsões
    return modelo.predict(X_dados)
```

Execute o teste novamente. Se a precisão for maior ou igual a 0.6, o teste passará. Seu sistema está em VERDE.

Passo 3: Refatorar – Melhorando o Modelo e os Testes

Com o teste verde, você pode agora refatorar seu modelo. Isso pode significar:

- **Experimentar Modelos Mais Complexos:** Trocar a Regressão Logística por uma Rede Neural, um SVM, ou um Random Forest.
- **Engenharia de Features:** Criar novas features a partir dos dados brutos para melhorar o desempenho.
- **Otimização de Hiperparâmetros:** Ajustar os parâmetros do modelo para obter melhor performance.
- **Adicionar Mais Testes:** Testes de edge cases, testes de robustez (o que acontece se a entrada for ruidosa?), testes de viés (o modelo é justo?).

```
# src/modelo_drones.py (Refatorado - Exemplo com RandomForest)

from sklearn.ensemble import RandomForestClassifier
import numpy as np

def treinar_modelo(X_treino, y_treino):
    # Modelo mais robusto: RandomForestClassifier
    modelo = RandomForestClassifier(n_estimators=100, random_state=42)
    modelo.fit(X_treino, y_treino)
    return modelo

def prever(modelo, X_dados):
    return modelo.predict(X_dados)
```

Seus testes existentes devem continuar passando. Se você introduzir um modelo que não atinge a precisão mínima, o teste falhará, alertando-o. Você também pode adicionar testes mais específicos:

```
# tests/test_modelo_drones.py (Adicionando mais testes)

# ... (imports e dados simulados)

def test_previsao_drone_especifico():
    modelo = treinar_modelo(X_test_simulado, y_test_simulado)
    # Um exemplo que sabemos que é um drone
    assert prever(modelo, np.array([[0.1, 0.2, 0.3]])) == 1

def test_previsao_nao_drone_especifico():
    modelo = treinar_modelo(X_test_simulado, y_test_simulado)
    # Um exemplo que sabemos que não é um drone
    assert prever(modelo, np.array([[0.8, 0.9, 0.7]])) == 0

def test_modelo_nao_cai_abaixo_limiar_com_novos_dados():
    # Teste com um conjunto de dados ligeiramente diferente
    X_novos = np.array([
        [0.15, 0.25, 0.35],
        [0.85, 0.95, 0.75],
    ])
    y_novos = np.array([1, 0])

    modelo = treinar_modelo(X_test_simulado, y_test_simulado)
    previsoes = prever(modelo, X_novos)
    precisao = accuracy_score(y_novos, previsoes)
    assert precisao >= 0.5 # Um limiar um pouco mais flexível para novos dados
```

Testes de Comportamento e Propriedades

Além das métricas de desempenho, é crucial testar o *comportamento* da sua IA. Isso é especialmente importante em sistemas críticos. Por exemplo, um sistema de IA de segurança não deve classificar um civil inocente como uma ameaça, independentemente de quão

complexa seja a entrada.

"Pense em testes de propriedade," MAIA sugere. "Eles não verificam um resultado específico, mas sim que certas propriedades do seu sistema se mantêm, não importa a entrada." Por exemplo, para um sistema de reconhecimento facial, uma propriedade pode ser: "Se a imagem for invertida horizontalmente, o sistema ainda deve reconhecer a mesma pessoa, mas com uma confiança ligeiramente menor."

6.1 – Rede Neural com Bugs Visíveis



AI PERFORMANCE



6.3 – Escudo de Propriedade em Modelo de IA



Outro exemplo é o teste de invariância: se você adicionar ruído imperceptível a uma imagem, a classificação do modelo não deve mudar drasticamente.

```
# tests/test_modelo_drones.py (Exemplo de teste de propriedade)

# ... (imports e dados simulados)

def test_modelo_invariancia_pequeno_ruído():
    modelo = treinar_modelo(X_test_simulado, y_test_simulado)

    # Exemplo de entrada original (um drone)
    original_input = np.array([[0.1, 0.2, 0.3]])
    original_prediction = prever(modelo, original_input)

    # Adiciona um pequeno ruído
    noisy_input = original_input + np.random.normal(0, 0.01,
original_input.shape)
    noisy_prediction = prever(modelo, noisy_input)

    # A previsão não deve mudar para um ruído tão pequeno
    assert original_prediction == noisy_prediction

# Testes de viés (exemplo conceitual)
# def test_modelo_nao_tem_vies_contra_determinada_populacao():
#     # Carrega dados de teste específicos para viés
#     X_pop_A, y_pop_A = carregar_dados_populacao_A()
#     X_pop_B, y_pop_B = carregar_dados_populacao_B()
#
#     modelo = treinar_modelo(X_treino_geral, y_treino_geral)
#
#     precisao_A = accuracy_score(y_pop_A, prever(modelo, X_pop_A))
#     precisao_B = accuracy_score(y_pop_B, prever(modelo, X_pop_B))
#
#     # A diferença de precisão não deve exceder um certo limiar
#     assert abs(precisao_A - precisao_B) < 0.05
```

Miniprojeto: Construindo uma IA de Detecção de Anomalias

"Sua missão final neste setor é construir uma IA de detecção de anomalias para identificar atividades suspeitas na rede da MegaCorp," MAIA declara. "Use TDD para garantir que sua IA seja precisa e confiável."

Objetivo: Criar uma função `detectar_anomalia(dados_rede)` que retorne `True` se os `dados_rede` indicarem uma anomalia e `False` caso contrário. Você pode usar um algoritmo simples de detecção de anomalias, como o Isolation Forest ou um Limiar de Desvio Padrão.

Passos com TDD:

1. **Vermelho:** Escreva testes para `detectar_anomalia`. Pense em:

- Cenários de dados normais (deve retornar `False`).
- Cenários de dados anômalos (deve retornar `True`).
- Edge cases (dados vazios, dados com valores extremos).

2. **Verde:** Implemente a função `detectar_anomalia` em `src/anomalias.py` com o mínimo de código para fazer os testes passarem. Comece com um limiar fixo, por exemplo.

3. **Refatorar:** Melhore o algoritmo de detecção de anomalias. Talvez introduza um modelo de ML mais sofisticado, ou ajuste os parâmetros com base em dados reais. Adicione mais testes de propriedade e comportamento para garantir a robustez da sua IA.

"Testar IAs é um campo em constante evolução, Iniciado," MAIA conclui. "Mas os princípios do TDD – a antecipação da falha, a construção incremental e a refatoração segura – permanecem seus aliados mais poderosos. Com eles, você pode domar até mesmo as mentes mais complexas do *Python 2099*."

Capítulo 7: Missões Finais

Você percorreu um longo caminho, Iniciado. Desde os fundamentos do ciclo Red-Green-Refactor até a complexidade de testar APIs e até mesmo sistemas de IA. Agora, é hora de aplicar todo o seu conhecimento em missões de campo, desafios práticos que simularão as realidades do *Python 2099*. Estas não são apenas tarefas; são oportunidades para solidificar suas habilidades e provar que você é um verdadeiro mestre do TDD.

"O campo de batalha digital aguarda," MAIA anuncia, sua voz carregada de expectativa. "Cada desafio é uma chance de aprimorar suas táticas e fortalecer sua resiliência. Lembre-se: o código que você escreve hoje pode ser a chave para a liberdade de amanhã."

Desafio 1: O Decodificador de Mensagens Criptografadas

As megacorporações usam algoritmos de criptografia complexos para proteger suas comunicações. Sua missão é construir um decodificador que possa reverter uma criptografia simples de substituição. Use TDD para garantir que seu decodificador seja robusto e capaz de lidar com diferentes cenários.

Objetivo: Criar uma função `decodificar_mensagem(mensagem_criptografada, chave_substituicao)` que retorne a mensagem original. A `chave_substituicao` será um dicionário mapeando caracteres criptografados para seus equivalentes originais.

Exemplo de Chave: `{"@": "a", "#": "e", "$": "i", "%": "o", "^": "u"}`

Exemplo de Uso: `decodificar_mensagem("h@ck%r", {"@": "a", "%": "o"})` deveria retornar `"hacker"` (assumindo que os outros caracteres não estão na chave e permanecem inalterados).

Passos com TDD:

- 1. Vermelho:** Escreva testes para `decodificar_mensagem` que falhem. Pense em:
 - Mensagens simples com todos os caracteres na chave.
 - Mensagens com caracteres que não estão na chave (devem permanecer inalterados).
 - Mensagens vazias.
 - Chaves de substituição vazias.
 - Mensagens com caracteres maiúsculos/minúsculos (a função deve ser case-sensitive ou case-insensitive? Defina isso nos testes!).
- 2. Verde:** Implemente a função `decodificar_mensagem` em `src/criptografia_rebelde.py` com o mínimo de código para fazer os testes passarem.
- 3. Refatorar:** Melhore a legibilidade, eficiência e robustez da sua função. Considere como lidar com chaves incompletas ou caracteres especiais.

Desafio 2: O Gerenciador de Recursos de Rede

Em um mundo onde a largura de banda é um luxo e os recursos de rede são constantemente monitorados, um gerenciador eficiente é vital. Você precisa criar um sistema que aloque e libere recursos de rede, garantindo que não haja conflitos ou vazamentos.

Objetivo: Criar uma classe `GerenciadorRecursosRede` com os seguintes métodos:

- `alocar_recurso(recurso_id)` : Tenta alocar um recurso. Retorna `True` se bem-sucedido, `False` se o recurso já estiver alocado.
- `liberar_recurso(recurso_id)` : Tenta liberar um recurso. Retorna `True` se bem-sucedido, `False` se o recurso não estiver alocado.
- `recursos_alocados()` : Retorna uma lista dos `recurso_id` atualmente alocados.

Passos com TDD:

1. **Vermelho:** Escreva testes para a classe `GerenciadorRecursosRede`. Pense em:
 - Alocar um recurso pela primeira vez.
 - Tentar alocar um recurso já alocado.
 - Liberar um recurso alocado.
 - Tentar liberar um recurso não alocado.
 - Verificar a lista de recursos alocados após várias operações.
 - Testar cenários de concorrência (embora a implementação inicial não precise ser thread-safe, os testes podem expor a necessidade).
2. **Verde:** Implemente a classe `GerenciadorRecursosRede` em `src/rede_segura.py` com o mínimo de código para fazer os testes passarem (talvez usando um `set` interno para os recursos alocados).
3. **Refatorar:** Otimize a implementação. Considere adicionar tratamento de erros mais robusto ou suporte a diferentes tipos de recursos.

7.1 – Desafios Cyberpunk (Missão Hacker)



7.2 – IA Sentinela Monitorando Rede



Miniprojeto: Criando Sua Própria IA Rebelde Segura

"Esta é a sua prova final, Iniciado," MAIA declara, sua imagem se tornando mais nítida, quase real. "Você construirá uma pequena IA que opera de forma autônoma, mas que é inerentemente segura e resistente a manipulações externas. Esta IA será sua primeira aliada na luta contra a opressão digital."

Cenário: Sua IA, apelidada de "Sentinela", deve monitorar um fluxo de dados de rede e alertar sobre atividades suspeitas. Ela deve ser capaz de aprender com o tempo, mas sem comprometer sua integridade.

Componentes da Sentinela (a serem desenvolvidos com TDD):

1. Módulo de Análise de Dados (`src/sentinela/analise.py`):

- Uma função `analisar_pacote(pacote_dados)` que retorna um `score_risco` (0 a 100).
- Testes para diferentes tipos de pacotes (normais, suspeitos, maliciosos).
- Comece com regras simples (ex: se o pacote contiver certas palavras-chave, o risco aumenta). Refatore para usar um modelo de ML simples (como um `DecisionTreeClassifier` treinado em dados de exemplo).

2. Módulo de Decisão (`src/sentinela/decisao.py`):

- Uma função `tomar_decisao(score_risco)` que retorna uma ação (ex: `"ignorar"`, `"alertar"`, `"bloquear"`).
- Testes para diferentes scores de risco e as ações esperadas.
- Garanta que a IA não tome decisões extremas para scores de risco baixos ou médios, e que sempre alerte para scores muito altos.

3. Módulo de Aprendizado Seguro (`src/sentinela/aprendizado.py`):

- Uma função `atualizar_modelo(dados_feedback)` que permite à IA aprender com feedback, mas de forma controlada.
- **Teste de Segurança:** Garanta que o modelo não possa ser "envenenado" por dados de feedback maliciosos. Por exemplo, se 90% dos dados de feedback forem "normais" mas 10% forem "maliciosos" e tentarem fazer a IA ignorar ameaças, o modelo não deve mudar drasticamente sua detecção de ameaças conhecidas.

- Isso pode ser feito limitando a taxa de aprendizado, validando o feedback, ou usando técnicas de aprendizado federado (conceitual para este miniprojeto).

Estrutura de Diretórios Sugerida:

```
python_2099_ebook/  
├── src/  
│   ├── criptografia_rebelde.py  
│   ├── rede_segura.py  
│   └── sentinela/  
│       ├── __init__.py  
│       ├── analise.py  
│       ├── decisao.py  
│       └── aprendizado.py  
├── tests/  
│   ├── test_criptografia_rebelde.py  
│   ├── test_rede_segura.py  
│   └── test_sentinela/  
│       ├── __init__.py  
│       ├── test_analise.py  
│       ├── test_decisao.py  
│       └── test_aprendizado.py  
└── ...
```

"Ao completar estas missões, você não será apenas um programador," MAIA conclui, sua voz se tornando um eco no vasto ciberespaço. "Você será um arquiteto da liberdade, um guardião do código, um verdadeiro hacker de *Python 2099*. Sua IA rebelde será um farol de esperança em um mundo dominado pelas sombras. Que seu código seja sua luz."

Conclusão: O Código Liberta

Você chegou ao fim de sua jornada pelo *Python 2099*. As ruas de Neo-Kyoto, antes um labirinto de códigos obscuros e sistemas vulneráveis, agora parecem um pouco mais claras. Você não é mais um Iniciado; você é um arquiteto da resistência digital, um hacker que compreende o poder inerente ao Test-Driven Development.

"O código é a linguagem do futuro, e o TDD é a sua gramática mais poderosa," MAIA proclama, sua voz ressoando com a autoridade de uma oráculo digital. "Você aprendeu que a falha não é um obstáculo, mas um guia. Que a refatoração não é um risco, mas uma oportunidade de aprimoramento. E que a confiança em seu código é a base para qualquer ato de rebelião."

A Filosofia do Programador Ético do Futuro

Em um mundo dominado por megacorporações e IAs autônomas, a ética do programador se torna mais crucial do que nunca. O TDD, ao forçar a clareza e a verificação contínua, não é apenas uma ferramenta técnica; é um pilar para a construção de sistemas mais justos e transparentes. Um código bem testado é um código mais previsível, menos propenso a bugs que podem ter consequências devastadoras no mundo real.

Como um hacker de *Python 2099*, você tem o poder de construir, de dismantelar e de proteger. Use o TDD não apenas para criar sistemas robustos, mas para garantir que esses sistemas sirvam a um propósito maior. Que seu código seja um farol de integridade em um mar de opacidade. Que cada teste que você escreve seja um ato de responsabilidade, uma promessa de que seu trabalho é confiável e seguro.

Referências e Próximos Passos

Sua jornada não termina aqui. O universo do TDD e do Python é vasto e em constante evolução. Continue explorando, continue testando, continue refatorando. Aqui estão algumas referências e próximos passos para aprofundar seus conhecimentos:

- **Livros Essenciais:**
 - "Test-Driven Development: By Example" por Kent Beck: O clássico que introduziu o TDD ao mundo.
 - "Clean Code: A Handbook of Agile Software Craftsmanship" por Robert C. Martin: Para aprofundar em refatoração e escrita de código limpo.
- **Documentação Oficial:**
 - [Documentação do unittest](#)
 - [Documentação do pytest](#)
- **Comunidades Online:**
 - Participe de fóruns e comunidades de Python e TDD (ex: Stack Overflow, Reddit r/Python, r/SoftwareEngineering).
 - Contribua para projetos open source. A melhor forma de aprender é fazendo e colaborando.

- **Projetos Pessoais:**

- Continue desenvolvendo sua "IA Rebelde Segura" ou crie seus próprios miniprojetos aplicando TDD desde o início.
- Experimente testar diferentes tipos de aplicações: web, desktop, mobile, sistemas embarcados.

"O futuro é incerto, mas uma coisa é clara: o código que você domina é a sua liberdade," MAIA conclui, sua imagem se desvanecendo, deixando apenas o eco de suas palavras. "Vá em frente, hacker. O *Python 2099* precisa de você. Que seu código seja sua arma, e seus testes, sua armadura. O Código Liberta."

Sugestões de Elementos Visuais

Ao longo do eBook, os seguintes elementos visuais podem enriquecer a experiência:

- **Introdução:** Uma imagem de uma cidade cyberpunk chuvosa com neon, e uma figura solitária olhando para um símbolo de Python estilizado com circuitos (como a capa).
- **Capítulo 1:**
 - Diagrama ASCII ou imagem simples do ciclo Red-Green-Refactor (círculo com setas e as palavras VERMELHO, VERDE, REFATORAR).
 - Representação visual de um "escudo" ou "rede de segurança" de testes ao redor de um bloco de código.
- **Capítulo 2:**
 - Ícones estilizados para `unittest` (talvez um martelo e cinzel) e `pytest` (talvez um raio ou uma arma futurista).
 - Diagrama de árvore de diretórios para a estrutura de arquivos de testes.
- **Capítulo 3:**
 - Visualização de um terminal com linhas de código piscando em vermelho e depois em verde.
 - Um "scanner" digital passando sobre um código, revelando bugs ou a ausência deles.
- **Capítulo 4:**

- Um diagrama de "antes e depois" de um código refatorado, mostrando a transição de um emaranhado para uma estrutura limpa.
- Uma figura de um cirurgião cibernético operando em um cérebro digital (representando o código).
- **Capítulo 5:**
 - Representação de APIs como "portões" ou "nós" em uma rede, com setas indicando requisições e respostas.
 - Um "fantasma" digital (mock) interceptando uma comunicação com um banco de dados real.
 - Uma "mesa de trabalho" com diferentes ferramentas (fixtures) prontas para serem usadas em um teste.
- **Capítulo 6:**
 - Um diagrama de um cérebro robótico ou uma rede neural com alguns "bugs" visíveis, e testes como "anticorpos" digitais.
 - Gráficos simples de desempenho de IA (precisão, etc.) com linhas de limiar.
 - Representação de um "escudo" de teste de propriedade ao redor de um modelo de IA.
- **Capítulo 7:**
 - Imagens de "missões" ou "desafios" em um ambiente cyberpunk (ex: um hacker decodificando algo, um drone sendo gerenciado).
 - Um diagrama da "Sentinela" (sua IA rebelde) monitorando uma rede, com ícones de alerta.

Esses elementos visuais, mesmo que apenas sugeridos, ajudam a manter a imersão no universo de *Python 2099* e a reforçar os conceitos técnicos de forma mais memorável.