ChatGPT

Roteiro de Testes em Python para Backend

1. Fundamentos de Testes de Software

Aprenda por que testar melhora a qualidade e a manutenção do código. Testes automatizados podem ser executados repetidamente, muito mais rápido que testes manuais, e verificam consistentemente funcionalidades específicas [1] [2]. Entenda os níveis de teste: **unitários** (testam pequenas funções ou métodos isoladamente), **funcionais/integrados** (testam componentes juntos, incluindo APIs e interfaces) e **end-to-end** (testam o sistema completo do ponto de vista do usuário) [2].

2. Ambiente e Ferramentas Básicas

Configure o ambiente instalando o **Python 3.8+**, criando um *venv*, e instalando frameworks de teste como **pytest** (`pip install pytest`) e usando também o builtin **unittest**. Familiarize-se com comandos de execução de testes: por exemplo `pytest` busca automaticamente arquivos `test_*.py`. Saiba que *pytest* é um pacote externo flexível e moderno, enquanto *unittest* faz parte da biblioteca padrão (tipo xUnit) [3] [4]. Em pytest, escreva testes como funções simples começando com `test_`, use `assert` normal e explore recursos como *markers* e *parametrize* para variar inputs. Em unittest, crie classes que herdam de `unittest.TestCase`, use métodos `setUp()` e `tearDown()`, e asserts específicos (`assertEqual`, etc.) [3] [5].

3. Escrevendo seus Primeiros Testes

Comece com um exemplo simples: crie uma função que retorna um valor e escreva um teste para ela. Em pytest, basta algo como:

```
def func(x): return x+1
def test_func():
    assert func(1) == 2
```

Execute com `pytest`; ele relata falhas e sucessos rapidamente. Em unittest seria:

```
import unittest
class TestFunc(unittest.TestCase):
    def test_func(self):
        self.assertEqual(func(1), 2)
```

Teste ambos para entender diferenças básicas. Use asserts para validar resultados e exceções esperadas. Leia tutoriais básicos de pytest e unittest em português ou inglês para reforçar conceitos (por exemplo, guias introdutórios [6] [7]).

4. *unittest*: Estrutura Clássica

No módulo `unittest`, cada teste é um método de uma classe `TestCase`. Implemente `setUp()` para preparar o cenário antes de cada teste e `tearDown()` para limpar após [8]. Exemplo:

```
from django.test import TestCase  # ou unittest.TestCase
class MyTests(TestCase):
    def setUp(self):
        # executa antes de cada teste
        self.obj = MyModel.objects.create(...)
    def tearDown(self):
        # executa depois de cada teste
        self.obj.delete()
    def test_algo(self):
        self.assertTrue(self.obj.field == 'valor')
```

O TestCase do Django acrescenta ferramentas para simular requisições, manipular banco de dados de teste, etc. Segundo a documentação Django, a suíte de testes estende *unittest* e oferece classes específicas (como `SimpleTestCase`, `TransactionTestCase`, `TestCase`, etc.) [9] [5]. Nelas, os testes são métodos iniciados por `test_` e usam asserts para verificar condições (por exemplo, `self.assertTrue`, `self.assertEqual`, etc.) [10].

5. *pytest*: Conceitos Essenciais

O pytest simplifica escrever testes em função. Não precisa de classes; use `assert` do Python diretamente. Ele oferece várias funcionalidades avançadas integradas: marcações (markers), parametrização de testes (`@pytest.mark.parametrize`), e plugins. Os testes se agrupam em arquivos `tests/` ou seguindo convenções de nome, e executam rapidamente com um único comando. Como nota, pytest "satisfaz aspectos chave de um bom ambiente de teste" – testes fáceis de escrever, rápidos, e execução simples [7]. Confira livros ou tutoriais (por exemplo, *Python Testing with pytest* de Brian Okken) para dominar cada recurso.

6. Fixtures (pytest) e Setup/Teardown (unittest)

Fixtures são funções que preparam estado ou contexto antes de um teste. No pytest, marque-as com `@pytest.fixture`. Por exemplo:

```
@pytest.fixture
def client():
    from myapp import app
    return app.test_client()
```

Um teste que pede `client` como parâmetro recebe o retorno da fixture. Como descrito em guias em português, "o método `smtp_connection` (fixture) será executado antes do teste `test_ehlo` e seu retorno é usado no teste" [6]. Você pode empilhar fixtures: uma fixture pode usar outra (como fonte de dados/configuração). Também pode usar `yield` na fixture para executar código de limpeza após o teste, simulando o *tearDown*. Exemplo:

```python
@pytest.fixture
def db():
    conn = setup_db()
    yield conn
    conn.close()  # cleanup
```

Isso torna possível isolar preparação e limpeza, reutilizando fixtures em vários testes [11]. No unittest, o equivalente é usar `setUp()` e `tearDown()` em cada `TestCase`.

7. *Mocks*, *Stubs* e *Spies* (Test Doubles)

Em testes, *mocks*, *stubs* e *spies* são "dublês" que simulam comportamentos de componentes externos. **Mocks** são objetos falsos programados para imitar dependências (bancos, APIs, etc.) e controlar respostas. No Python, use o módulo `unittest.mock` para criar mocks e `patch` funções/clases. Por exemplo, você pode `patch('requests.get')` para que chamadas HTTP retornem respostas pré-determinadas. Como explica um tutorial em português: "Mocks são objetos simulados usados em testes para imitar o comportamento de dependências externas... permitindo isolar a unidade de código que está sendo testada" [12]. Isso ajuda a testar cenários difíceis (erros de rede, dados específicos) sem depender do recurso real [13].

Stubs são similares, mas mais simples: retornam valores fixos para chamadas específicas, sem lógica complexa. Eles "são programados para retornar respostas específicas a chamadas durante um teste" e não tentam replicar todo o comportamento original [14]. Use stubs para simular serviços externos com respostas controladas (por exemplo, uma API retorna sempre certo JSON).

Spies (espiões) observam o código real: eles deixam a função original rodar normalmente, mas registram como ela foi chamada (quantas vezes, com quais argumentos). Em Python, um "spy" pode ser criado indiretamente usando `Mock` e inspecionando seus atributos (`call_count`, `call_args`). Em geral, **mocks** permitem verificar interações e forçar comportamentos, **stubs** fornecem respostas fixas, e **spies** apenas monitoram chamadas. Conforme definido em blogs técnicos, "spies envolvem a função real e registram informações sobre o uso dela sem interromper seu funcionamento normal" [15].

Ferramentas populares: `unittest.mock.Mock`, `MagicMock`, `patch`; no pytest há o plugin **pytest-mock** (fornece fixture `mocker` simples) e a fixture built-in `monkeypatch` para sobrescrever módulos durante o teste. Use `mock.patch` para substituir uma função/classe dentro de escopos de teste, e depois fazer `mocked.assert_called_with(...)` para verificar chamadas.

8. Testando Frameworks Web Python

Para **Django**: utilize `django.test.TestCase` e o **Client** interno (para simular requisições HTTP). Os testes Django podem usar dados de teste em banco SQLite temporário. Com o plugin **pytest-django**, você pode usar pytest sintaxe e fixtures, aproveitando suas vantagens (menos boilerplate e fixtures poderosas) [16]. Por exemplo, configure `pytest.ini` para incluir o Django e escreva testes sem precisar importar `unittest`. A documentação Django ressalta que a framework de teste é baseada em `unittest` com classes específicas que permitem "simular requisições, inserir dados de teste e inspecionar a saída" [9]. Em resumo: crie classes herdando de `TestCase`, defina métodos `test_*`, use `self.client.get()` ou `self.client.post()` para testar views e assert para verificar respostas.

Para **Flask**: use `flask.testing` ou simplesmente o objeto `app.test_client()`. Em pytest, geralmente define-se uma fixture que cria o aplicativo Flask em modo de teste. Por exemplo, no TestDriven recomenda-se algo como:

```
with flask_app.test_client() as test_client:
    response = test_client.get('/')
    assert response.status_code == 200
```

Isso executa a rota inicial e verifica o status [17] [18]. O resultado (`response.data`) pode ser comparado ao esperado. Você também pode usar o plugin **pytest-flask** que disponibiliza fixtures prontas, mas não é obrigatório. Em tests práticos, geralmente separamos casos de teste funcional e de unidade, mas ambos podem usar o `test_client`.

Para **FastAPI**: graças ao Starlette, existe o `TestClient` (baseado em HTTPX) que integra facilmente com pytest. A documentação oficial mostra um exemplo típico:

```
from fastapi.testclient import TestClient
client = TestClient(app)

def test_read_main():
    response = client.get("/")
    assert response.status_code == 200
    assert response.json() == {"msg": "Hello World"}
```

Ou seja, você instancia o `TestClient(app)` e escreve funções de teste normais (não `async`) com asserts comuns [19]. A própria doc afirma que "você pode usar pytest diretamente com FastAPI" e ilustra esse padrão [20] [19].

9. Projetos Práticos e Estudos de Caso

Aplique conhecimento criando projetos de exemplo e escrevendo testes completos:

- **Aplicação CRUD simples (FastAPI ou Flask)** – monte uma API de gerenciamento (por exemplo, usuários ou tarefas) com banco de dados (SQLite ou outro). Escreva testes unitários para funções de negócio e testes de integração para as rotas REST. Use pytest, fixtures para setar ambiente, e mocks para simular chamadas externas ou o próprio banco (por exemplo, usando um banco em memória). [Exemplo de referência: tutorial de FastAPI com pytest mostra testes de endpoints CRUD usando fixtures e setup/teardown [21].]

- **Site básico em Django** – crie um pequeno blog ou lista de itens. Utilize os testes do Django para cobrir modelos (models), views, e formulários. Experimente tanto o framework de testes padrão (`TestCase`) quanto o pytest-django. Certifique-se de configurar dados de teste e usar as clients para requisições em views. Consulte guias como o tutorial da Mozilla sobre testes em Django para orientações passo a passo.

- **Automação de testes em pipeline de dados** – seguindo um exemplo no contexto de dados, pratique mocks e fixtures simulando conexões com banco, APIs externas, etc. Um artigo em português demonstra como usar mocks (`unittest.mock.patch`) para testar funções que

fazem chamadas HTTP, garantindo que você possa verificar se a função trata respostas de forma correta [12].

Em cada projeto, avance dos testes unitários (pequenas funções) para testes de integração (rotas, interações com DB, autenticação, etc.), adicionando cobertura de caso de erro e uso de fixtures para simplificar repetição de pré-condições [18] [11]. Integre ferramentas como **coverage.py** para medir cobertura de testes e **CI/CD** (GitHub Actions, GitLab CI) para rodar testes automaticamente a cada alteração.

10. Tópicos Avançados e Recursos Extras

Aprofunde-se em técnicas avançadas: por exemplo, **pytest parametrization** para gerar muitos casos de teste com facilidade, o plugin **pytest-xdist** para testes paralelos, e **hypothesis** para *property-based testing*. Estude **Test-Driven Development (TDD)** escrevendo testes antes do código, e **Behavior-Driven Development (BDD)** usando frameworks como *behave* (se desejar). Aplique **mocks mais sofisticados** (fixtures do pytest-mock, context managers do MonkeyPatch, etc.), e teste **serviços externos** com *responses* ou *aioresponses* para APIs assíncronas.

Para cada conceito acima, há tutoriais e documentações em português e inglês. Por exemplo, leia posts recentes que abordam pytest, unittest, fixtures e mocks (como tutoriais Dev.to e Medium em português [12] [6] e blogs em inglês como o TestDriven.io e Pytest-with-Eric). As referências usadas neste roteiro incluem artigos atualizados e oficiais (pytest docs, Django docs, FastAPI docs) que ajudarão no aprendizado aprofundado [7] [20] [9]. Explore estes materiais para cada tópico citado, garantindo uma formação completa do básico ao avançado.

Referências: Documentações oficiais (pytest, Django, FastAPI), posts técnicos e tutoriais em português e inglês [3] [20] [12] [6] [11]. Cada conceito apresentado acima pode ser consultado mais detalhadamente nessas fontes.

---

1 5 8 9 10 Django Tutorial Part 10: Testing a Django web application - Learn web development | MDN
https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Server-side/Django/Testing

2 3 7 17 18 Testing Flask Applications with Pytest | TestDriven.io
https://testdriven.io/blog/flask-pytest/

4 16 pytest-django·PyPI
https://pypi.org/project/pytest-django/

6 Tutorial de pytest para iniciantes | by Leonardo Galani | assert(QA) | Medium
https://medium.com/assertqualityassurance/tutorial-de-pytest-para-iniciantes-cbdd81c6d761

11 [PT-BR] Utilizando fixtures do pytest para melhorar seus testes - DEV Community
https://dev.to/mvtenorio/pt-br-utilizando-fixtures-do-pytest-para-melhorar-seus-testes-4fp

12 Pytest Mocks, o que são? - DEV Community
https://dev.to/mchdax/mocks-o-que-sao-40id

13 14 15 Mocks, Spies, and Stubs: How to Use? - testRigor AI-Based Automated Testing Tool
https://testrigor.com/blog/mocks-spies-and-stubs/

19 20 Testing - FastAPI
https://fastapi.tiangolo.com/tutorial/testing/

21 Building And Testing FastAPI CRUD APIs With Pytest (Hands-On Tutorial) | Pytest with Eric
https://pytest-with-eric.com/pytest-advanced/pytest-fastapi-testing/