Randomização de dados em testes

Live de Python #281

Essa live não tem o objetivo de introduzir o assunto de testes

Aviso!

Essa live não tem o objetivo de introduzir o assunto de testes

(é uma soma de um pouco de teoria + opinião)

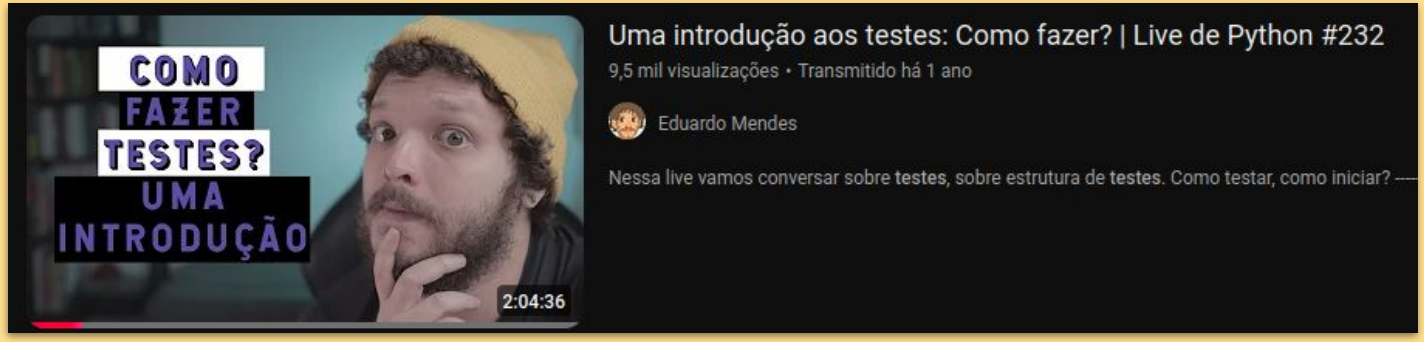
Aviso!

Essa live não tem o objetivo de introduzir o assunto de testes

(é uma soma de um pouco de teoria + opinião)

[menos código que o normal!]

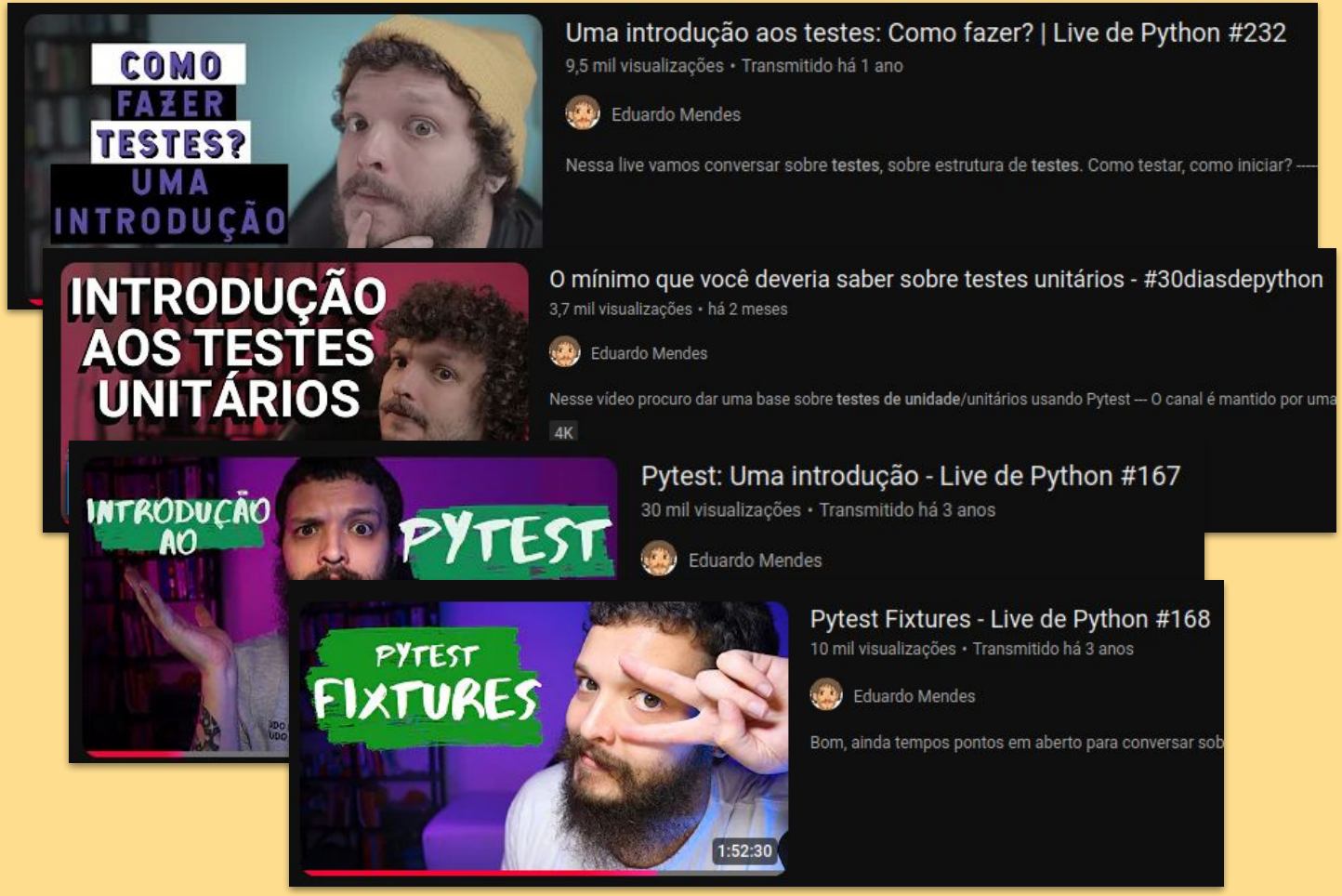
Aviso!



Temos diversos materiais sobre isso no canal

Temos diversos materiais sobre isso no canal

Temos diversos materiais sobre isso no canal



Temos diversos materiais sobre isso no canal

Roteiro

1. Estrutura dos testes

Antes de adentrar no assunto

2. Randomização de dados

Diminuindo a carga cognitiva

3. Faker

Gerando dados falsos

4. Factory-boy

Evitando fixtures...

apoia.se/livedepython

pix.dunossauro@gmail.com

patreon.com/dunossauro

Ajude o projeto <3

Adriana Cavalcanti, Alan Costa, Alexandre Girardello, Alexandre Harano, Alexandre Lima, Alexandre Takahashi, Alexandre Villares, Alfredo Braga, Allan Kleitson, Alysson Oliveira, Andre Azevedo, Andre Makoski, Andre Paula, Antonio Filho, Apc 16, Arthur Santiago, Aslay Clevisson, Aurelio Costa, Belisa Arnhold, Bernarducs, Biancarosa, Brisa Nascimento, Bruno Barcellos, Bruno Batista, Bruno Freitas, Bruno Ramos, Bruno Russian, Brunu, Carlos Gonçalves, Celio Araujo, Christian Fischer, Cleiton Fonseca, Controlado, Curtos Treino, Daniel Aguiar, Daniel Bianchi, Daniel Brito, Daniel Souza, Daniel Wojcickoski, Danilo Boas, Danilo Silva, David Couto, David Kwast, Denis Bernardo, Dgeison, Diego Guimarães, Dino, Diogo Faria, Edgar, Eduardo Pizorno, Emerson Rafael, Érico Andrei, Everton Silva, Fabio Barros, Fabio Faria, Fabiokleis, Felipe Adeildo, Felipe Augusto, Felipe Corrêa, Fernanda Prado, Fernandocelmer, Fichele Marias, Francisco Aclima, Frederico Damian, Fulvio Murenu, Gabriel Lira, Gabriel Mizuno, Gabriel Paiva, Gabriel Simonetto, Geilton Cruz, Geisler Dias, Giovanna Teodoro, Giuliano Silva, Guibeira, Guilherme Felitti, Guilherme Ostrock, Guilherme Piccioni, Gustavo Suto, Haelmo Almeida, Harold Gautschi, Heitor Fernandes, Hellyson Ferreira, Helton, Helvio Rezende, Henri Alves, Henrique Andrade, Henrique Machado, Henriquesebastiao, Herian Cavalcante, Hiago Couto, Hideki, Igor Taconi, Ivan Santiago, Janael Pinheiro, Jean Melo, Jean Victor, Jeferson Vitalino, Jefferson Antunes, Jefferson Silva, Jerry Ubiratan, Jhonata Medeiros, Jlx, Joao Rocha, John Peace, Jonas Araujo, Jonatas Leon, Joney Sousa, Jorge Silva, Jose Barroso, Jose Edmario, Joseíto Júnior, Jose Mazolini, José Predo), Josir Gomes, Jrborba, Juan Felipe, Juliana Machado, Julio Franco, Julio Silva, Kaio Engineer, Kaio Peixoto, Leandro O., Leandro Pina, Leandro Vieira, Leonan Ferreira, Leonardo Mello, Leonardo Nazareth, Lisandro Pires, Lucas Carderelli, Lucas Castro, Lucas Mello, Lucas Mendes, Lucas Nascimento, Lucas Schneider, Luciano Ratamero, Luis Ottoni, Luiz Duarte, Luiz Martins, Luiz Paula, Luiz Perciliano, Mackilem Laan, Marcelo Araujo, Marcelo Fonseca, Marcelo Grimberg, Marcio Freitas, Marcos Almeida, Marcos Oliveira, Maria Santos, Marina Passos, Marlon Rocha, Mateusamorim96, Mateus Lisboa, Matheus Vian, Mírian Batista, Mlevi Lsantos, Murilo Carvalho, Ocimar Zolin, Otávio Carneiro, Patrick Felipe, Pedro Henrique, Peterson Santos, Phmmdev, Prof Santana, Pytonyc, Rafael Faccio, Rafael Ferreira, Rafael Fontenelle, Rafael Lopes, Rafael Romão, Raimundo Ramos, Ramayana Menezes, Renan, Renan Sebastião, Rene Pessoto, Renne Rocha, Ricardo Silva, Ricardo Viana, Richard Sousa, Rinaldo Magalhaes, Rodrigo Barretos, Rogério Nogueira, Rui Jr, Samanta Cicilia, Santhiago Cristiano, Sergio Nascimento, Sherlock Holmes, Shirakawa, Tenorio, Téo Calvo, Tharles Andrade, Thiago Araujo, Thiago Lucca, Thiago Paiva, Tiago, Tiago Emanuel, Tomás Tamantini, Valdir, Varlei Menconi, Vinicius Meneses, Vinicius Silva, Vinicius Souza, Vinicius Stein, Vladimir Lemos, Williamslews, Willian Lopes, Zeca Figueiredo, Zero! Studio

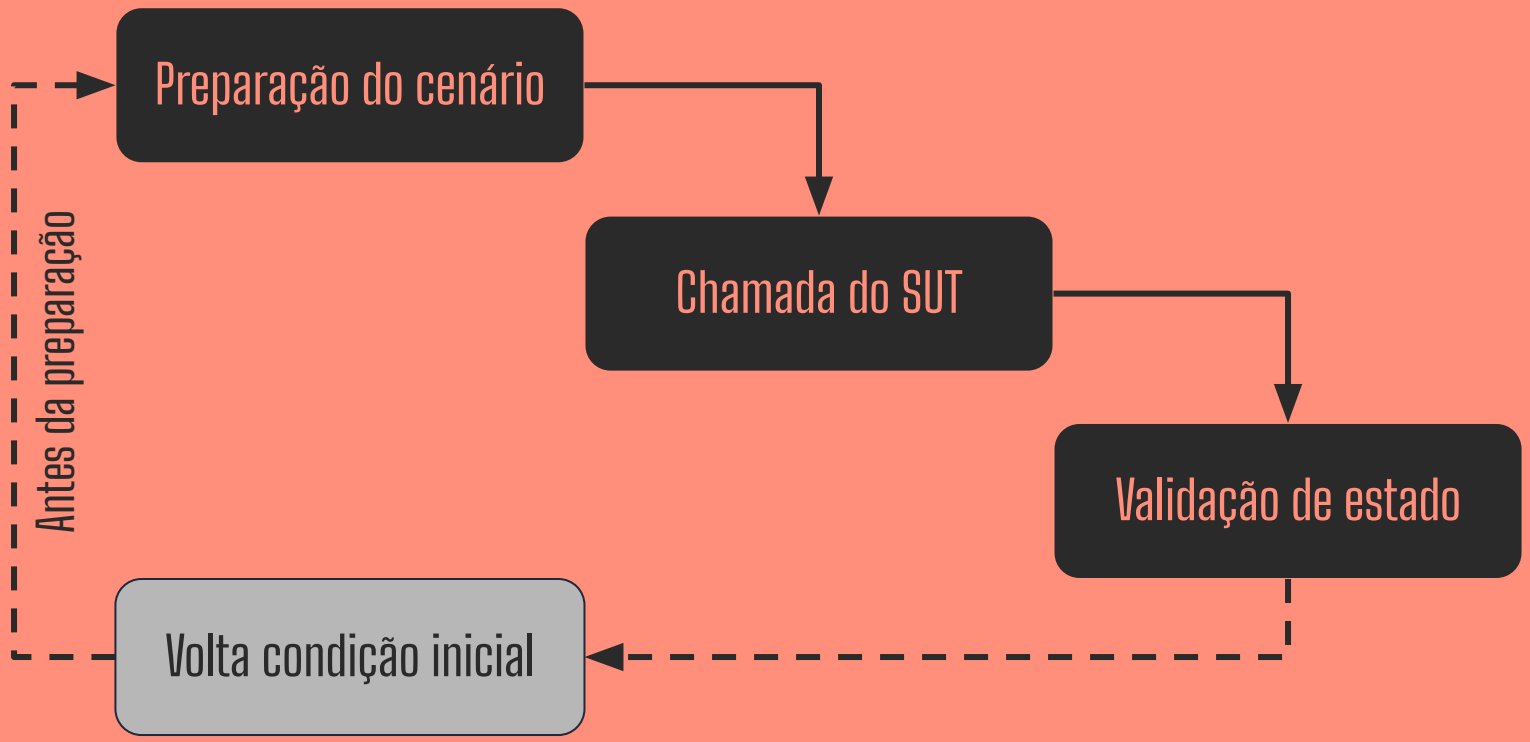
Obrigado você

Estruturas dos

Testes

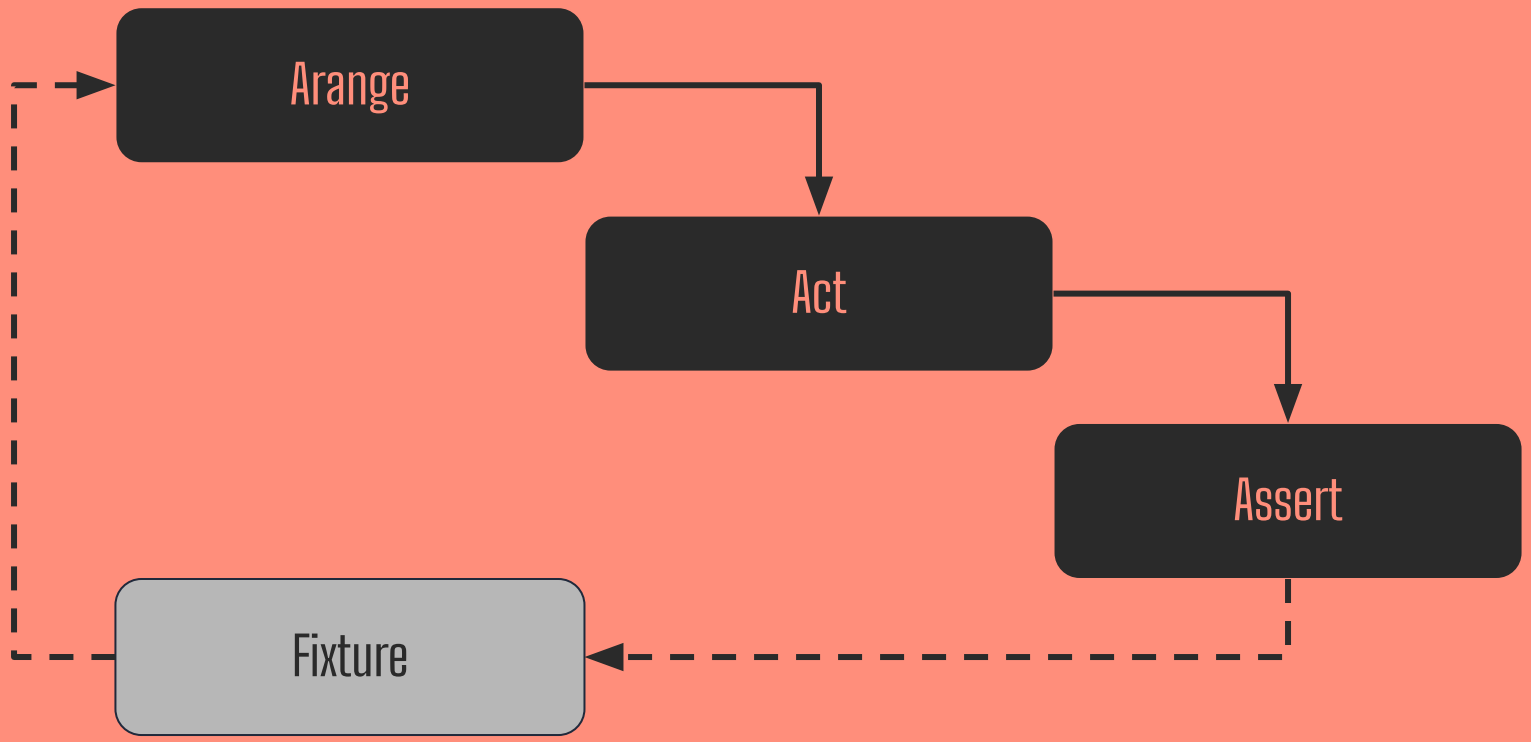
Estrutura básica de testes

Independente da bibliografia, vamos chegar sempre em algo com:



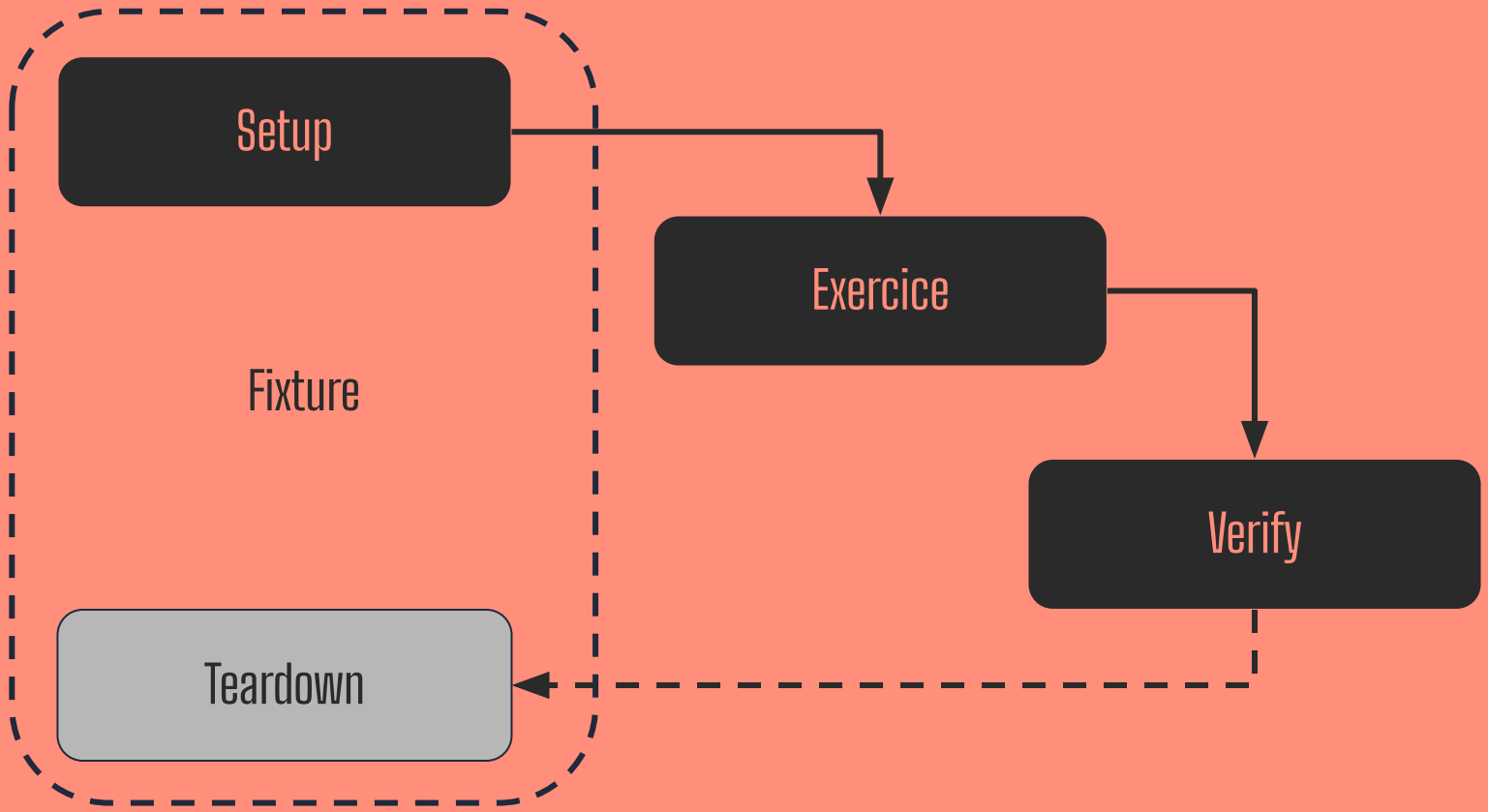
AAA [vou me basear nessa nomenclatura]

Independente da bibliografia, vamos chegar sempre em algo com:



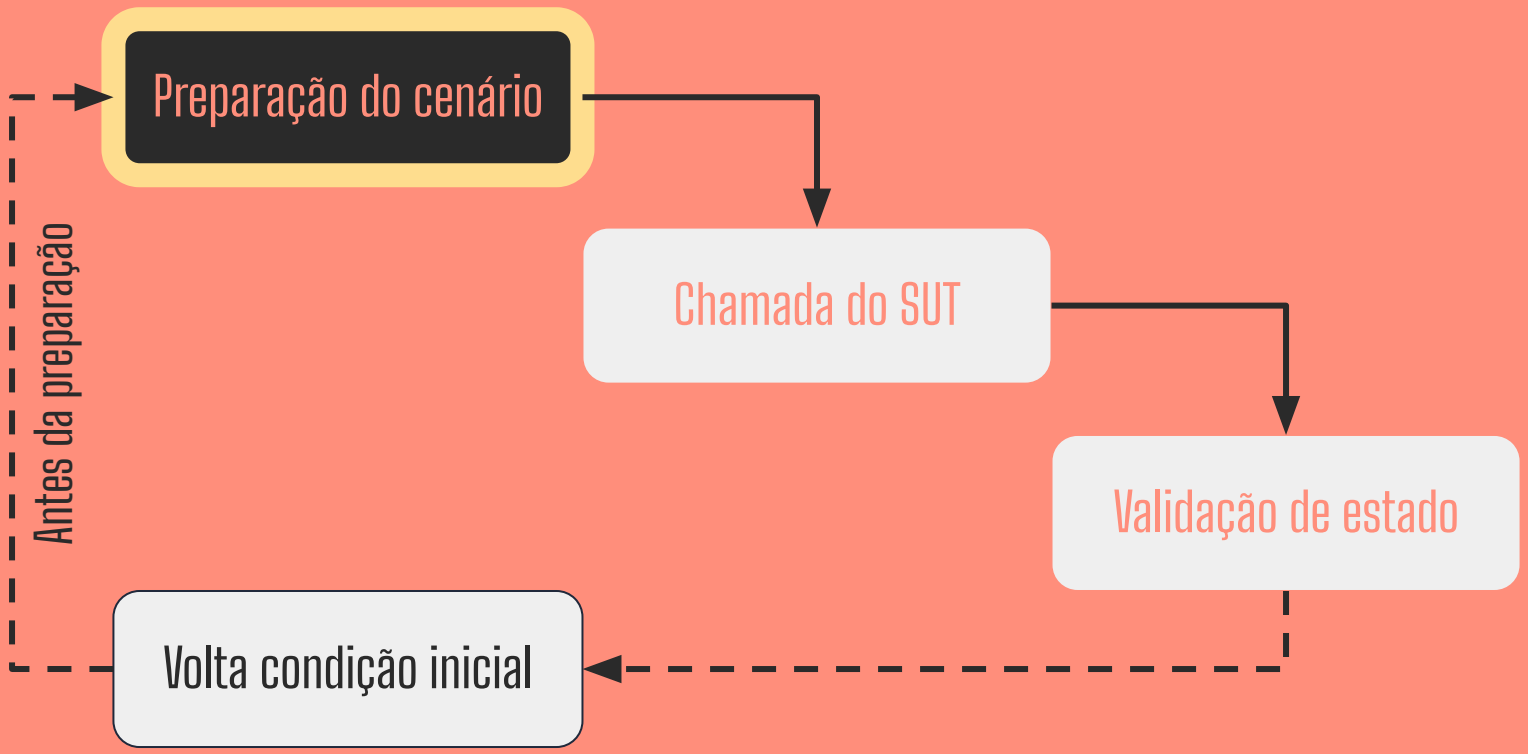
4 phases

Independente da bibliografia, vamos chegar sempre em algo com:



Onde vamos focar nesse primeiro momento!

Preparar cenários complexos geralmente dá muito trabalho



O Arange / Setup

Às vezes exercitar o SUT depende de condições bastante específicas.

Algo como:

- Um **filtro de query** no banco de dados
  - N registros em dois grupos N^1 e N^2, onde N^1 corresponde ao filtro
- **Validação/extração de dados** [ETL?]
  - Um arquivo com estrutura a estrutura (x, y, z) [csv?]
- Uma **aplicação externa** específica precisa estar de pé
  - Um banco de dados, um sistema de mensageria, API externa
- Objetos precisam de **dummies** para serem chamados
  - 5 parâmetros são exigidos, mas somente 1 é necessário para a branch
- ...

Arrange

Em algumas estruturas são exigidos dados que sigam determinado formato/tipo. Por exemplo em validações:

- **Strings com formato fixo:** cpf, email, rg, username, paths, ip, DOI, isbn, ...
- **Arquivos com formato fixo**: csv, json, xml, ...
  - Com estruturas de formato fixo
- **Localização dos formatos**: caracteres especiais, moedas, nomes, ...
- **Estruturas com formato fixo**:
  - Uma lista de tuplas com formatos fixos
  - Uma lista de inteiros maiores que zero
  - Uma tupla onde cada posição segue a um formato fixo
  - Um dicionário com questões específicas
- **Um objeto instanciado de uma forma fixa**
- ...

Anti-padrões

Geralmente nessas condições o que fazemos?

- Copiamos os **dados de produção**
  - Fere a **anonimização**
    - ex: Validar CPF com um CPF real de cliente
  - Só cobre **cenários conhecidos**
    - ex: Um caractere ainda não visto, como **Ł**, pode quebrar o encode
- Dados **incompletos** ou **artificialmente simples**
  - Não representa a diversidade dos dados reais
    - ex: test_user, test@mail.com, testpassword
- **N Fixtures** para o **mesmo teste**
  - Conftests enormes
    - ex: User1, User2, UserSemEmail

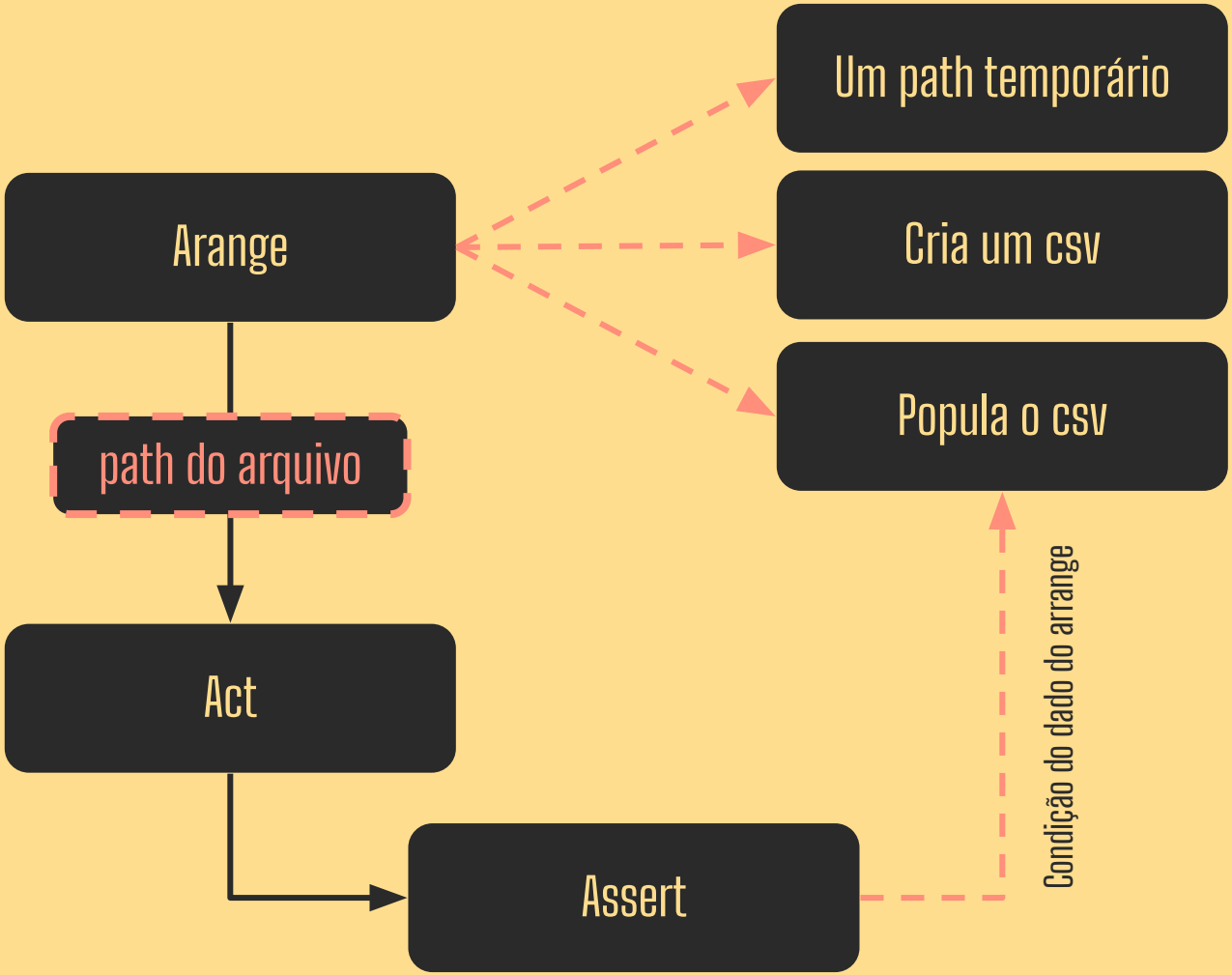
Um caso provável

Você recebe um arquivo csv com N campos e precisa tirar somente dois e remover os dados nulos:

```python
def get_cols(csv_file, *cols) -> pl.DataFrame:
    """Pega colunas de um arquivo csv e filtra os nulos"""
    df = pl.read_csv(csv_file)
    new_df = df.select(*cols).drop_nulls()

    return new_df
```

Como arranjamos isso?



O que nos daria algo como

```python
def test_validate_csv(tmpdir: Path):
    # Arrage (file.csv, populado, com N colunas)
    f = tmpdir / 'temp.csv'

    data = (
        "nome,email,telefone\n"
        "eduardo,duno@ssauro,929292827\n"
        "fasuto,fasuto@ssauro,92928237\n"
        "kirb,,92928237\n"
        "dejair,deja@ir.net,\n"
    )

    f.write_text(data,encoding='utf-8')

    # act
    get_cols(str(f), 'nome', 'email')
```

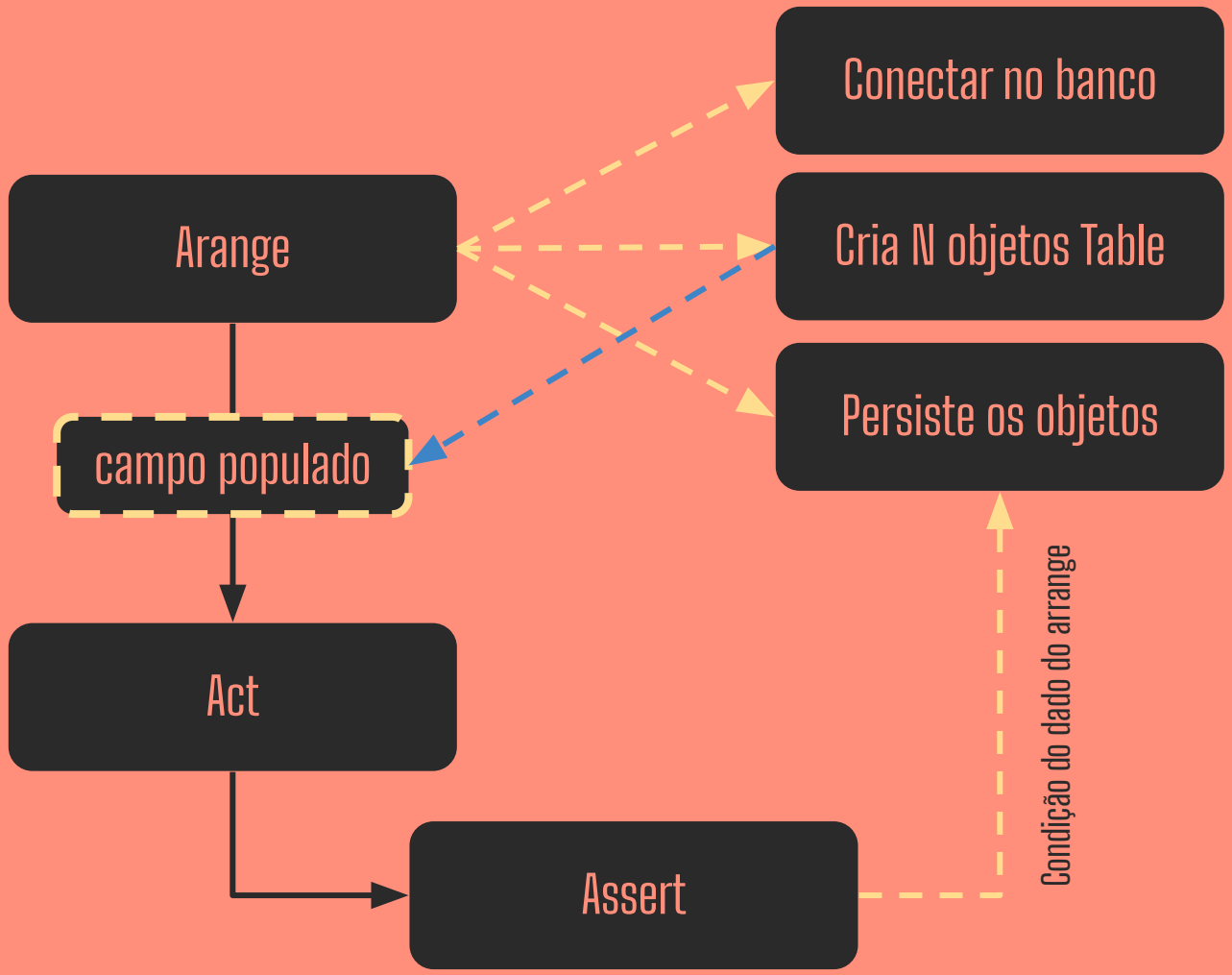
Carga cognitiva de milhões

Um outro caso comum

Buscar por objetos que deveriam estar no banco no momento do teste

```python
def list_objects(
    text: str, session: sqlalchemy.orm.Session
) -> list[Table]:
    return session.scalars(
        select(Table)
        .filter(Table.field.contains(text))
    )
```

Como arranjamos isso?



O que nos daria algo como

Dummies de db tem tipos/estruturas pré definidas (mais carga cognitiva)

```
def test_list_objects(session):
    # arrange
    session.add_all(
        [
            Table(field='texto', field_a='dummy', field_b='dummy', ...),
            Table(field='texto', field_a='dummy', field_b='dummy', ...),
            Table(field='texto', field_a='dummy', field_b='dummy', ...),
        ]
    )
    session.commit()

    #  act
    list_objects('texto', session)
```

Randomização

- Carga cognitiva
+ Testes

Randomização de dados

Em diversos cenários de testes os **dados não importam**, **somente a estrutura/forma** dos dados. Exemplo:

- Criação de registro
  - Espero que seja criado, não importam "os dados" passados
- Parâmetros que não afetam o SUT
  - Muitas vezes a branch testada não usa tudo que foi passado
- Relacionamentos em DOCs [depended-on component]
  - Uma tabela A depende de B e C, mas não usaremos B e C
- Filtros
  - Não importam os dados que não vamos filtrar, só popular a base

Exemplo [https://fastapidozero.dunossauro.com/05/]

```python
@app.post('/users/', status_code=201, response_model=UserPublic)
def create_user(user: UserSchema, session: Session = Depends(get_session)):
    db_user = session.scalar(
        select(User)
        .where((User.username == user.username) | (User.email == user.email))
    )

    if db_user:
        raise HTTPException(status_code=409, detail='User already exists')

    new_user = User(
        username=user.username, password=user.password, email=user.email
    )
    session.add(new_user)
    session.commit()
    session.refresh(new_user)
    return new_user
```

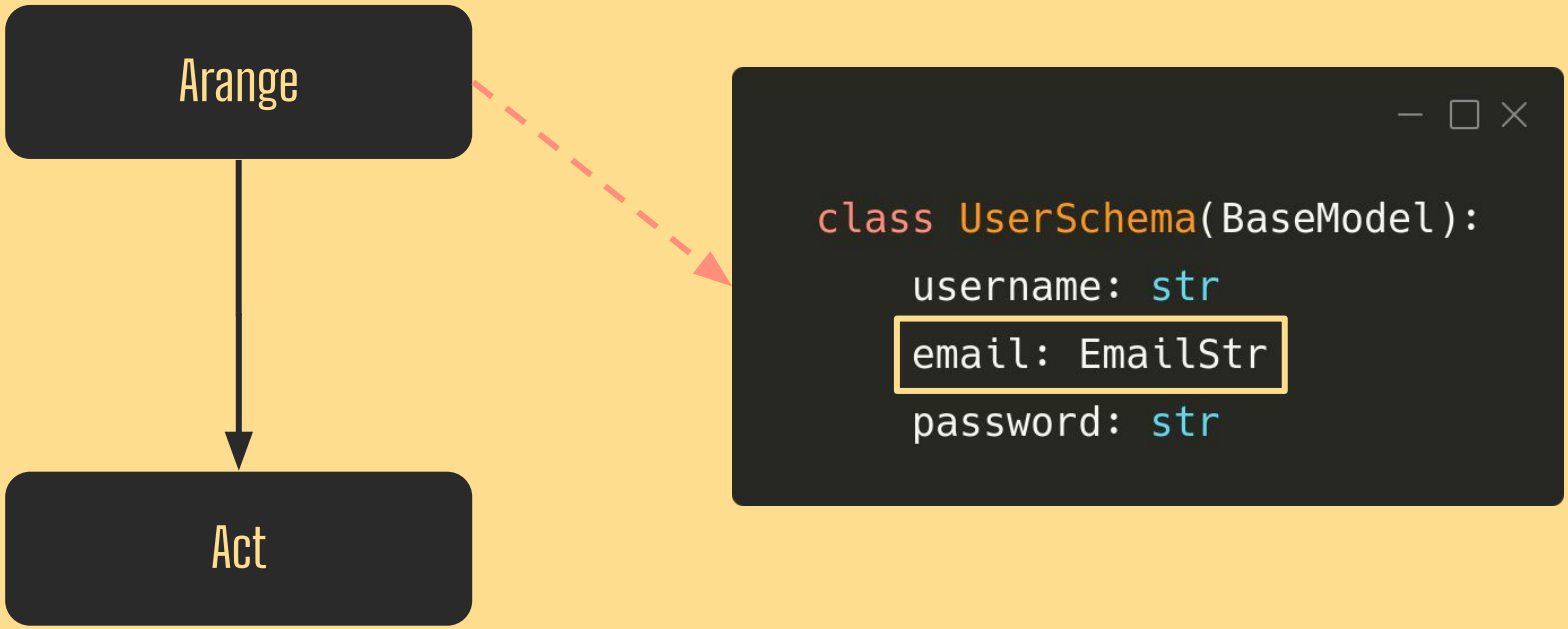
Exemplo [https://fastapidozero.dunossauro.com/05/]

```python
@app.post('/users/', status_code=201, response_model=UserPublic)
def create_user(user: UserSchema, session: Session = Depends(get_session)):
    db_user = session.scalar(
        select(User)
        .where((User.username == user.username) | (User.email == user.email))
    )

    if db_user:
        raise HTTPException(status_code=409, detail='User already exists')

    new_user = User(
        username=user.username, password=user.password, email=user.email
    )
    session.add(new_user)
    session.commit()
    session.refresh(new_user)
    return new_user
```

Ou seja, somente a estrutura importa



Se, em alguns casos, somente a estrutura importa

Podemos diminuir a carga cognitiva usando bibliotecas de randomização que fornecem "fabricas" de valores randômicos estruturados.

Por exemplo:

- Faker
- Mimesis
- Factory-boy
- Model-backery
- FuxFactory
- fake2DB
- ...

E podemos ir disso

```python
def test_create_user(client):
    response = client.post(
        '/users',
        json={
            'username': 'alice',
            'email': 'alice@example.com',
            'password': 'secret',
        },
    )
    assert response.status_code == HTTPStatus.CREATED
    assert response.json() == {
        'username': 'alice',
        'email': 'alice@example.com',
        'id': 1,
    }
```

E podemos ir disso, para isso

```python
def test_create_user(client):
    response = client.post(
        '/users',
        json={
            'username': 'alice'
            'email': 'alice@exa
            'password': 'secret
        },
    )
    assert response.status_code
    assert response.json() == {
        'username': 'alice',
        'email': 'alice@example
        'id': 1,
    }
```

```python
from mimesis import Person
from mimesis.locales import Locale

provider = Person(locale=Locale.PT_BR)


def test_create_user(client):
    # arrange
    dummy_data = {
        'username': provider.username(),
        'email': provider.email(),
        'password': provider.password()
    }

    # act
    client.post('/users', json=dummy_data)
```

O outro caso [https://fastapidozero.dunossauro.com/05/]

```python
@app.post('/users/', status_code=201, response_model=UserPublic)
def create_user(user: UserSchema, session: Session = Depends(get_session)):
    db_user = session.scalar(
        select(User)
        .where((User.username == user.username) | (User.email == user.email))
    )

    if db_user:
        raise HTTPException(status_code=409, detail='User already exists')

    new_user = User(
        username=user.username, password=user.password, email=user.email
    )
    session.add(new_user)
    session.commit()
    session.refresh(new_user)
    return new_user
```

O outro caso [https://fastapidozero.dunossauro.com/05/]

Nesse caso o dado é importante, mas os dados não importantes ainda podem ser dummies randomizados

```python
@app.post('/users/', status_code=201, resp
def create_user(user: UserSchema  session:
    db_user = session.scalar(
        select(User)
        .where((User.username == user.username) | (User.email == user.email))
    )

    if db_user:
        raise HTTPException(status_code=409, detail='User already exists')

    new_user = User(
        username=user.username, password=user.password, email=user.email
    )
    session.add(new_user)
    session.commit()
    session.refresh(new_user)
    return new_user
```

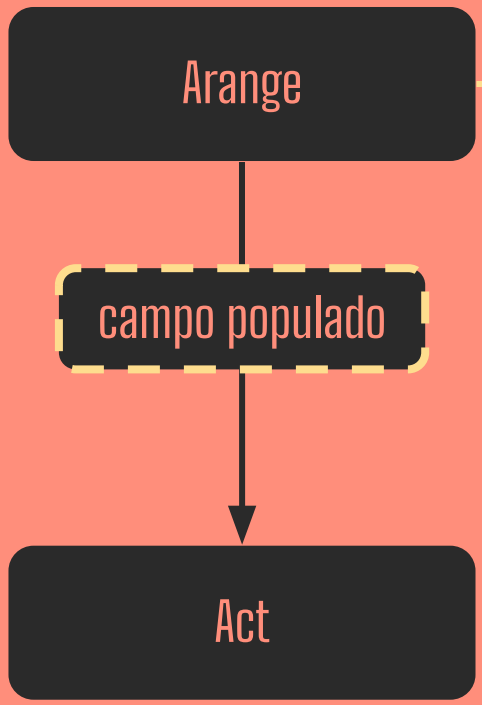
Onde isso brilha?

Com fábricas de objetos grandes e inter-relacionados. Como:

- Classes de ORM
- Schemas
- Data seed
- Anonimização
- Testes de carga
- ...

```python
@dataclass
class Usuario:
    id: int
    nome: str
    sobrenome: str
    email: str
    senha: str
    telefone: str
    data_nascimento: date
    cpf: str
    rg: str
    endereco: Endereco
    data_registro: datetime
    ultimo_acesso: datetime
    ativo: bool
    nivel_acesso: str
    foto_perfil: str | None = None
```

```python
@dataclass
class Endereco:
    rua: str
    numero: str
    bairro: str
    cidade: str
    estado: str
    cep: str
    pais: str
```




```python
usuario_exemplo = Usuario(
    id=1,
    nome="João",
    sobrenome="Silva",
    email="joao.silva@email.com",
    senha="senha_segura_123",
    telefone="(11) 98765-4321",
    data_nascimento=date(1990, 5, 23),
    cpf="123.456.789-00",
    rg="12.345.678-9",
    endereco=Endereco(
        rua="Rua Fictícia",
        numero="123",
        bairro="Centro",
        cidade="Cidade Exemplo",
        estado="EX",
        cep="12345-678",
        pais="País Exemplo"
    ),
    data_registro=datetime.now(),
    ultimo_acesso=datetime.now(),
    ativo=True,
    nivel_acesso="usuario",
    foto_perfil="http://exemplo.com/foto.jpg"
)
```

Randomizando

```python
class UserFactory(factory.Factory):
    class Meta:
        model = Usuario

    nome = factory.Faker('name')
    sobrenome = factory.Faker('last_name')
    email = factory.Faker('email')
    senha = factory.Faker('password')
    telefone = factory.Faker('phone_number')
    data_nascimento = factory.Faker('date')
    data_registro = factory.Faker('date')
    ultimo_acesso = factory.Faker('date')
    rg = factory.Faker('rg', locale='pt_BR')
    cpf = factory.Faker('cpf', locale='pt_BR')
    ativo = factory.Faker('pybool')
    id = factory.Sequence(int)
    endereco = factory.SubFactory(EnderecoFactory)
```

```
>>> pprint(UserFactory.create())
Usuario(id=0,
        nome='John Maddox',
        sobrenome='Williams',
        email='daniel83@example.net',
        senha='0#v9teAmVf',
        telefone='(381)940-1562x559',
        data_nascimento='2011-01-01',
        cpf='497.365.812-09',
        rg='605128479',
        endereco=Endereco(rua='Alexander Bridge',
                          numero='1759',
                          bairro='Delta',
                          cidade='Port Mary',
                          estado='MP',
                          cep='37539',
                          pais='Bhutan'),
        data_registro='1989-10-21',
        ultimo_acesso='2024-04-11',
        ativo=False,
        nivel_acesso='usuario',
        foto_perfil=None)
```

Dá pra fazer mais...

Existem tipos de testes randomizados diferentes, como:

- Fuzzy testing; e
- Property based testing

Não vou me aprofundar isso, será nosso assunto na futura live 284

Faker

Gerando dados falsos

Faker

Faker é uma biblioteca usada pra geração de dados falsos. Com ideia de inicializar arquivos de banco de dados, anonimização, testes de carga, ...

- Criada por **Daniele Faraglia**
- Licença: **MIT**
- Primeira release: 2010
- Release atual: 36.1.1 (13/02/2025)

Faker é baseado em bibliotecas de dados falsos já consagradas como PHP Faker, Perl Faker, and by Ruby Faker...

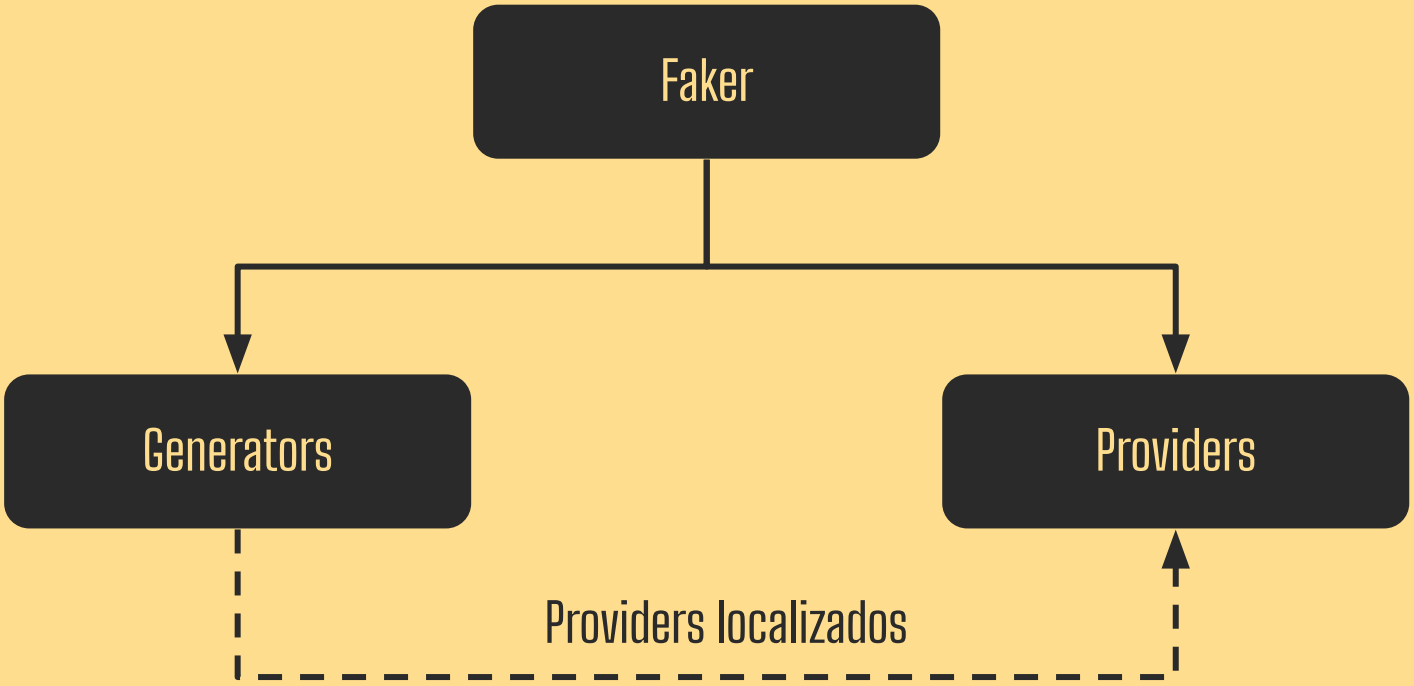
```
pip install faker
```

Uso básico

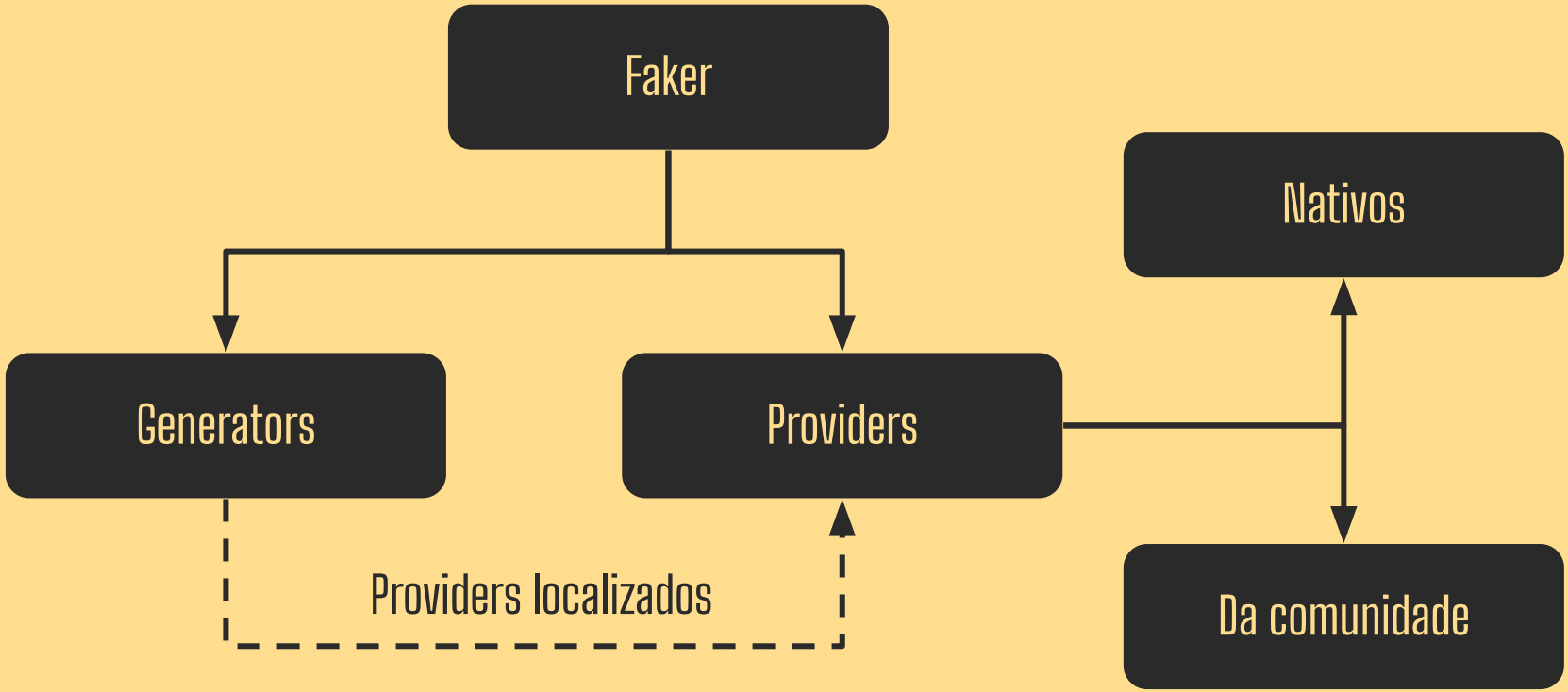
```python
from faker import Faker
fake = Faker()

fake.name()        # nome falso
fake.user_name()   # username falso
fake.email()       # email falso
fake.
```

A estrutura do Faker



A estrutura do Faker



Localização [https://faker.readthedocs.io/en/master/locales.html]

Quando usamos o faker, por padrão, ele vem com o idioma em inglês.

Dessa forma podemos gerar os dados em inglês do Provider **en**:

https://faker.readthedocs.io/en/master/locales/en.html

Por padrão (o EN), pode gerar endereços e pessoas.

Localização

Existe, porém, suporte para diversos idiomas, como:

- Alemão
- Armênio
- Árabe
- Bengali
- Bósnio
- Búlgaro
- Checo
- Coreano
- Croata
- Dinamarquês
- Espanhol
- Eslovaco
- Francês
- Georgiano
- Grego
- Gujarate
- Hebraico
- Hindi
- Holandês
- Húngaro
- Indonésio
- Irlandês
- Italiano
- Japonês
- Latim
- Letão
- Lituano
- Luxemburguês
- Maltês
- Nepali
- Norueguês
- Oriá
- Polonês
- Português
- Romeno
- ...

https://faker.readthedocs.io/en/master/locales.html

Como usar a localização?

```python
from faker import Faker
fake = Faker('pt_BR')

fake.cpf()          # 163.275.948-91
fake.rg()           # 074368527
fake.cnpj()         # 42.107.936/0001-66'
fake.bairro()       # Vila dos anjos
fake.color_name()   # Amarelo brasilis
```

Providers

O faker conta com diversos providers nativos.

Eles fornecem dados para escopos específicos, como:

- Endereços
- Códigos bancários
- Internet
- Moedas
- Datas
- Telefones
- Objetos python
- Arquivos
- ...

https://faker.readthedocs.io/en/stable/providers.html

Como adicionar os providers?

```python
from faker import Faker
from faker.providers import internet, python

fake = Faker()
fake.add_provider(internet)
fake.ipv4()          # 121.125.181.170
fake.ipv6()          # 56bb:c213:5438:6469:61cf:8dc:1a99:6ac
fake.image_url()     # https://dummyimage.com/68x236
fake.mac_address()   # 76:53:5f:20:18:d8
```

Como adicionar os providers?

```
fake.add_provider(python)

fake.pylist(2, value_types=(int, float))  # [863.69, 2089]
fake.pyint()                              # 1537
fake.pyset(1)                             # {'saya32ygb3'}
fake.pyset(3, value_types=int)            # {6912, 8337, 5148}
fake.pybool()                             # False
fake.pydecimal()                          # Decimal('-76542.231')
```

Providers da comunidade

Existem diversos:

https://faker.readthedocs.io/en/stable/communityproviders.html

```
# pip install faker_music
from faker_music import MusicProvider
fake.add_provider(MusicProvider)

fake.music_genre()       # K-pop
fake.music_subgenre()    # Anti-Folk
fake.music_instrument()  # Drum machine'
fake.music_instrument_category()  # percussion
```

Integração com pytest

Quando instalamos o faker, ele provê uma fixture para o pytest, dessa forma a integração entre os testes e o objeto fake podem acontecer naturalmente:

```
def test_com_faker(faker):
    fake_data = {
        'name': faker.name()
    }
    # ...
```

Caso precise de um locale padrão

```python
import pytest

@pytest.fixture(scope='session', autouse=True)
def faker_session_locale():
    return ['pt_BR']
```

Factory Boy

Falseando objetos completos

Factory-boy

Factory-boy é uma biblioteca para substituir o uso de fixtures (que dão trabalho de manter) por fábricas de objetos.

- Criada por **Mark Sandstrom**
- Licença: MIT
- Primeira release: 2010
- Release atual: **3.3.3** (03/02/2025)
- Suporte a ORMs (django, mono, mongoengine e sqlalchemy)

Criada com inspiração no factory_bot, um gem do ruby.

```
pip install factory_boy
```

A ideia principal

É criar fábricas (o padrão de projeto) de classes com valores randômicos **e/ou** pré-customizados:

```python
from dataclasses import dataclass
from datetime import date

@dataclass
class Pessoa:
    nome: str
    sobrenome: str
    data_de_nascimento: date
```

```python
from factory import Factory, Faker

class PessoaFactory(Factory):
    class Meta:
        model = Pessoa

    nome = Faker('first_name')
    sobrenome = Faker('last_name')
    data_de_nascimento = Faker('date_object')
```

A ideia principal

É criar fábricas (o padrão de projeto) de classes com valores randômicos e/ou pré-customizados:

```python
from dataclasses import dataclass
from datetime import date

@dataclass
class Pessoa:
    nome: str
    sobrenome: str
    data_de_nascimento: date
```

```python
from factory import Factory, Faker

class PessoaFactory(Factory):
    class Meta:
        model = Pessoa

    nome = Faker('first_name')
    sobrenome = Faker('last_name')
    data_de_nascimento = Faker('date_object')
```

Factory

Dado que você tenha uma fábrica, você poderia criar objetos de Meta.model:

```
>>> PessoaFactory.build()
Pessoa(
  nome='Larry', sobrenome='Dorsey', data_de_nascimento=datetime.date(1986, 8, 27)
)

>>> PessoaFactory.build_batch(2)
[
  Pessoa(
    nome='Molly', sobrenome='Newton', data_de_nascimento=datetime.date(2023, 12, 18)
  ),
  Pessoa(
    nome='Stephanie', sobrenome='Lane', data_de_nascimento=datetime.date(1993, 7, 26)
  )
]
```

Dados fixos, não randomizados

Você pode customizar qualquer parâmetro da Factory para adaptar ao seu cenário de teste.

Vamos supor que você precise testar com uma pessoa nascida hoje:

```
>>> PessoaFactory.build(
  data_de_nascimento=date.today()
)

Pessoa(
  nome='Douglas',
  sobrenome='Harris',
  data_de_nascimento=datetime.date(2025, 2, 24)
)
```

Mais recursos necessários

Às vezes temos interdependências entre campos, precisamos que não se repitam e diversas outras situações.

- **Sequence**
  - Altera o valor a instância de Meta.model criada
- **LazyFunction**
  - Executa uma função e passa aoa tributo o resultado
- **LazyAttribute**
  - Usa valores pré-definidos em outros atributos com parte

```python
@dataclass
class Pessoa:
    id: str
    username: str
    email: str
    criado_em: date
```

```python
import factory

class PessoaFactory(factory.Factory):
    class Meta:
        model = Pessoa

    id = factory.Sequence(int)
    username = factory.Faker('user_name')
    email = factory.LazyAttribute(lambda obj: f'{obj.username}@test.com')
    criado_em = factory.LazyFunction(date.today)
```

```
>>> PessoaFactory.build_batch(2)
Pessoa(
  id=0, username='ybrown', email='ybrown@test.com', criado_em=datetime.date(2025, 2, 24)
)
Pessoa(
  id=1, username='ion', email='ion@test.com', criado_em=datetime.date(2025, 2, 24)
)
```

```python
import factory

class PessoaFactory(factory.Factory):
    class Meta:
        model = Pessoa

    id = factory.Sequence(int)
    username = factory.Faker('user_name')
    email = factory.LazyAttribute(lambda obj: f'{obj.username}@test.com')
    criado_em = factory.LazyFunction(date.today)
```

O que mais?

Existem coisas que não vamos cobrir aqui, por uma questão de tempo. Como:

- Herança
- SubFactories
- Traits
- FuzzyAttributes
- Integração com ORMs
- Integração com mimesis

https://youtu.be/gK-tuF_OPJQ

Eu acho que o Factory-boy merecia uma live "técnica" somente dele em diversos cenários. E vocês? (talvez o faker também...)

apoia.se/livedepython

pix.dunossauro@gmail.com

patreon.com/dunossauro

Ajude o projeto <3

Referências

- **xUnit Patterns**: http://xunitpatterns.com/index.html
- **Faker**: https://faker.readthedocs.io/en/master/
- **Mimesis**: https://mimesis.name/master/index.html
- **Factory-boy**: https://factoryboy.readthedocs.io/en/stable/index.html