Property-Based Testing

uma introdução com Hypothesis

Live de Python # 285

Roteiro

1. Um passo pra trás

O que precisamos saber antes de chegar aqui.

2. Property-bases

Para o que viemos

3. Hypothesis

Criando estratégias de randomização

4. E eu com isso?

"the life snake"

Essa live não é uma introdução a testes!

Avisos

Uma introdução aos testes: Como fazer? | Live de Python #232

9.7K views • Streamed 1 year ago

Eduardo Mendes

Nessa **live** vamos conversar sobre **testes**, sobre estrutura de **testes**. Como testar, como iniciar? ---

Pytest: Uma introdução - Live de Python #167

30K views • Streamed 3 years ago

Eduardo Mendes

Sempre quis usar o pytest mais sentiu um pouco de dific

O mínimo que você deveria saber sobre testes unitários -...

3.9K views • 3 months ago

Caso precise

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, Andre Azevedo, Andre Makoski, Andre Paula, Antonio Filho, Apc 16, Apolo Ferreira, Arthur Santiago, Aslay Clevisson, Augusto Domingos, Aurelio Costa, Belisa Arnhold, Bernardo Fontes, Bernarducs, Biancarosa, Brisa Nascimento, Bruno Batista, Bruno Bereoff, Bruno Freitas, Bruno Ramos, Bruno Russian, Brunu, Carlos Gonçalves, Celio Araujo, Christian Fischer, Claudemir Cruz, 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, Diogo Faria, Edgar, Eduardo Pizorno, Elias Soares, Emerson Rafael, Érico Andrei, Everton Silva, Fabio Barros, Fábio Belotto, Fabio Faria, Fabiokleis, 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, Harold Gautschi, Heitor Fernandes, Hellyson Ferreira, Helton, Helvio Rezende, Henri Alves, Henrique Andrade, Henrique Machado, Henriquesebastiao, Herian Cavalcante, Hiago Couto, Igor Taconi, Ivan Santiago, Janael Pinheiro, Jean Melo, Jean Victor, Jeferson Vitalino, Jefferson Antunes, Jefferson Silva, Jerry Ubiratan, Jhonata Medeiros, Jlx, Joao Rocha, Jonas Araujo, Jonatas Leon, Joney Sousa, Jorge Silva, Jose Barroso, Jose Edmario, Joseíto Júnior, Jose Mazolini, José Predo), Josir Gomes, Jplay, Jrborba, Juan Felipe, Juliana Machado, Julio Franco, Julio Silva, Kaio Engineer, Kaio Peixoto, Lara Nápoli, Leandro O., Leandro Pina, Leandro Vieira, Leonan Ferreira, Leonardo Adelmo, 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, Marcelo Araujo, Marcelo Fonseca, Marcelo Grimberg, Marcio Freitas, Marcos Almeida, Marcos Oliveira, Maria Santos, Marina Passos, Mateusamorim96, Mateus Lisboa, Matheus Vian, Mírian Batista, Mlevi Lsantos, Murilo Carvalho, Naomi Lago, Ocimar Zolin, Otávio Carneiro, Patrick Felipe, Pedro Henrique, Peterson Santos, Phmmdev, 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, Rodrigo Santana, Rodrigo Vieira, Rogério Nogueira, Rui Jr, Samanta Cicilia, Santhiago Cristiano, Sherlock Holmes, Shirakawa, Tenorio, Téo Calvo, Tharles Andrade, Thiago Araujo, 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

Calma lá....

Um passo pra trás

Um passo pra trás [exemplo_00.py]

Antes de adentrarmos de fato property-based testing [testes baseados em propriedades], seria interessante lembrar a anatomia de um teste "normal".

Por normal, dizemos que o teste é baseado em exemplos [Example-based testing].

Vamos usar um exemplo **banal**:

```python
def add(x: int, y: int) -> int:
    return x + y
```

Anatomia de um teste [exemplo_00.py]

Geralmente, quando temos um bloco de testes, temos algo como AAA:

- **Arrange**: A organização dos dados que queremos usar no SUT
- **Act**: A chamada direta ao SUT
- **Assert**: A verificação se a ação esperada aconteceu

```python
def test_add():
    # Arrange: Organização dos dados para o teste
    x, y = 1, 1
    esperado = 2
    # Chamada do SUT
    resultado = add(x, y)
    # Assert
    assert  resultado == esperado
```

Exemplos como especificação

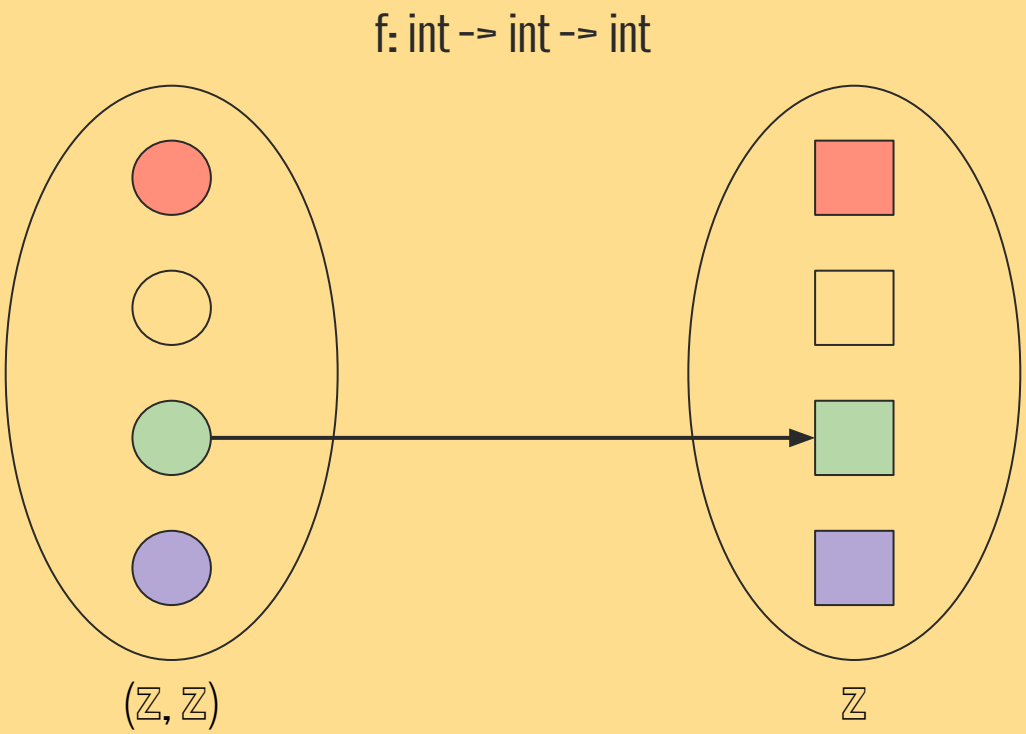
Chamamos esse caso de **example-based** pois pensamos em um "exemplo" de como executar nosso código. Trazendo para linguagem de especificação, teríamos algo como:

- Dados os números 1, 1
- Quando chamarmos a função add
- Então o resultado deve ser 2

Existe uma escolha fixa de exemplares para resolver esse teste. Escolhemos **1** para **x** e **1** para **y**, sabendo que a resposta esperada para esse caso é **2**.

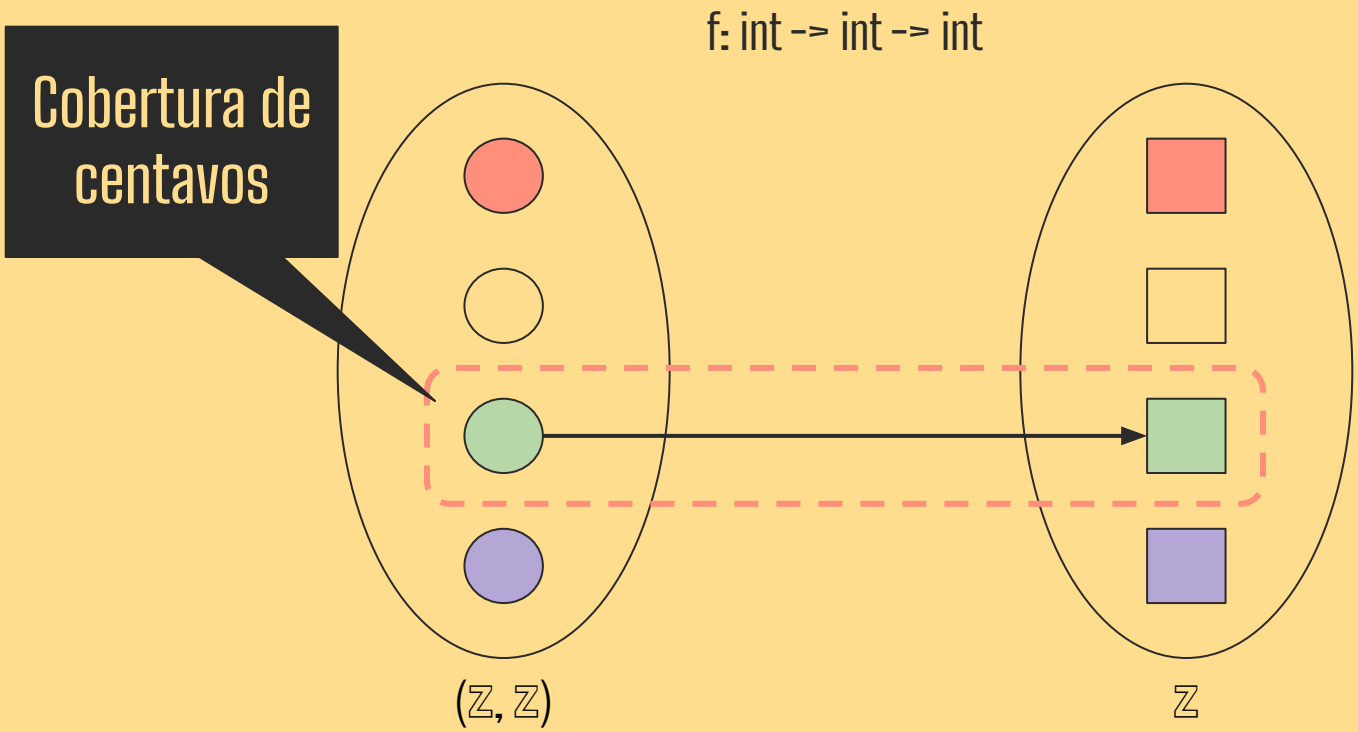
A cobertura

Se olharmos pra isso de uma forma mais "formal", teríamos algo como:



A cobertura

Se olharmos pra isso de uma forma mais "formal", teríamos algo como:



Parametrização de exemplos

Sabemos que a função de **add** trabalha com qualquer valor ***qualquer*** valor inteiro. É o esperado... Porém, ao exemplificar, não conseguimos ir muito longe, seriam muitos testes.

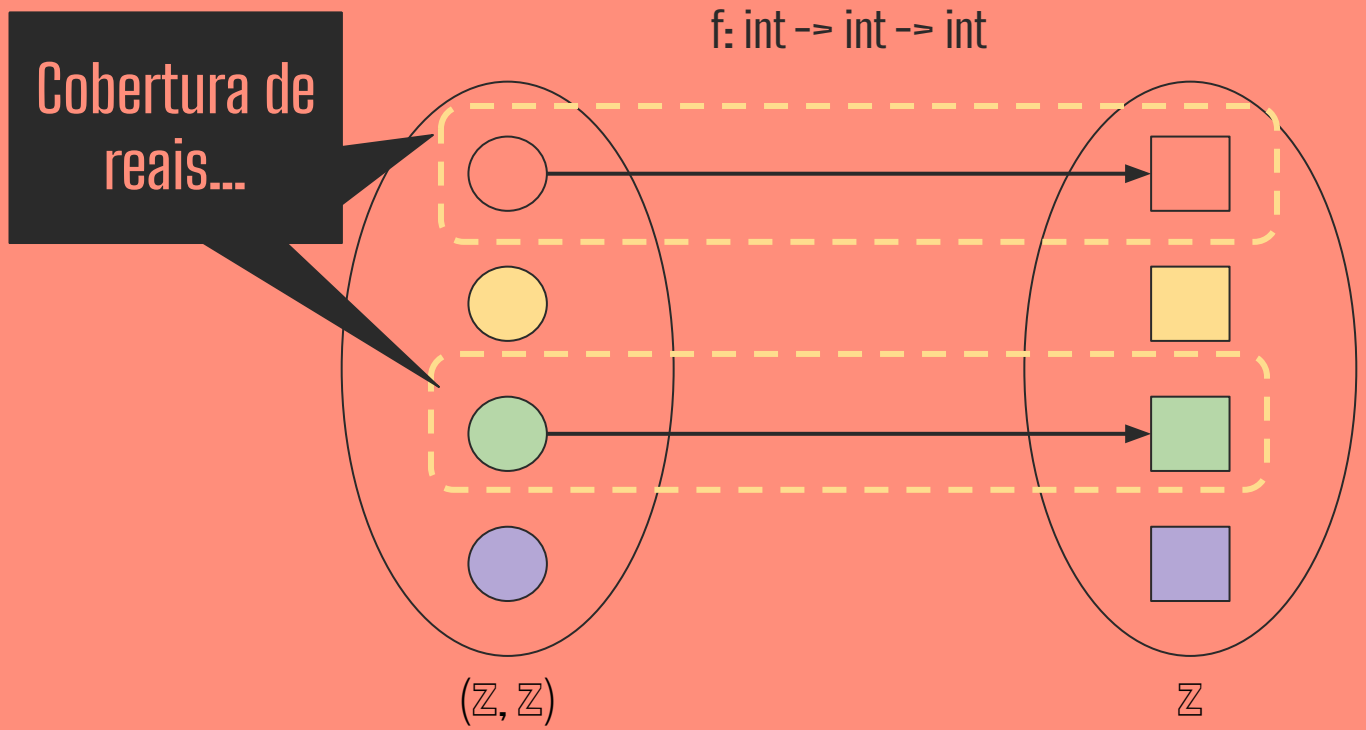
Quando as coisas chegam nesse ponto, temos um recurso nativo do pytest como o **parametrize** ou então o **subTest** do Unittest. Onde montamos uma variedade de exemplos e os chamamos todos em um único teste:

Por exemplo [exemplo_01.py]

```python
@mark.parametrize(
    'x,y,esperado', [(1, 1, 2), (2, 2, 4), (5, 5, 10)]
)
def test_add(x, y, esperado):
    # Chamada do SUT
    resultado = add(x, y)
    # Assert
    assert  resultado == esperado
```

Cobertura

Dessa forma, temos uma cobertura um pouco mais interessante:



Porém, cobrir o oceano de exemplares de inteiro seria impossível, claro, eles são infinitos.

Melhora...

O problema das bordas

Ainda assim, com mais casos você conseguiria dizer que sua cobertura de exemplares são o suficiente? Quais casos você deixou de cobrir? Se isso fosse um input de usuário, você sabe... Pessoas são bem criativas.

Para cobrir os casos que não sabemos ou podem produzir efeitos imaginários, precisamos de mais dados...

Para o que viemos....

Propert y-based

Testes baseados em propriedades

A ideia das propriedades é pensar o "contrário" dos exemplos. É descrever uma **propriedade invariante** do SUT, definir uma estratégia e deixar o teste fazer mais "com menos".

Gosto do slogan do Hypothesis:

"Teste mais rápido, corrija mais"

Propriedades

Vamos entender com um exemplo. A soma tem diversas propriedades (não precisamos de todas), como:

- **Comutativa**: x + y == y + x
- **Elemento neutro**: x + 0 == x

Ou seja, podemos testar essas propriedades sem saber quem são os exemplares.

Exemplificando [exemplo_02.py]

Independente do valor, a propriedade comutativa se mantém:

```python
def test_add_comutativo():
    # exemplares
    x, y = 3, 2
    # Chamada do sut
    assert add(x, y) == add(y, x)
```

Concorda comigo que independente de quais sejam os exemplares essa operação em teoria tem que dar certo?

Randomização [exemplo_03.py]

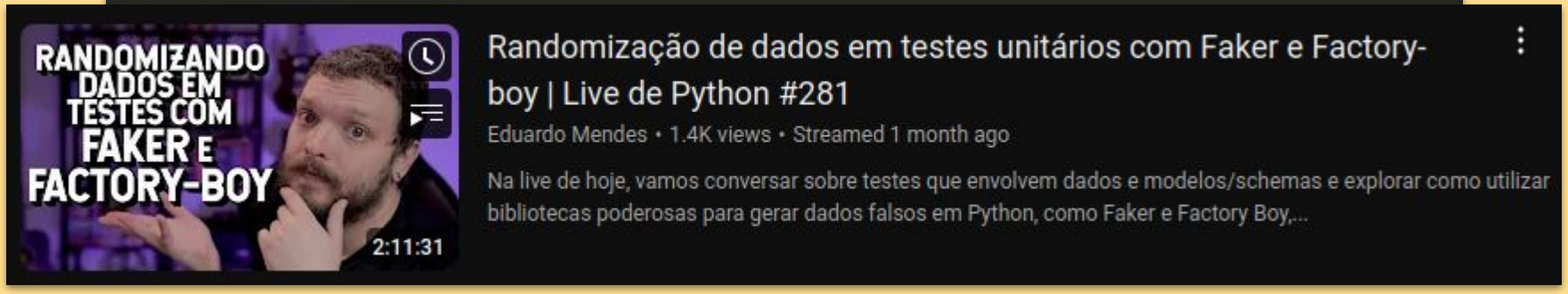
Se qualquer exemplar do tipo vale, então podemos randomizar os dados de entrada e esperar que as propriedades sejam cumpridas.

```python
def test_add_comutativo():
    # exemplares
    x, y = faker.pyint(), faker.pyint()
    # Chamada do sut
    assert add(x, y) == add(y, x)
```

Randomização

Se qualquer exemplar do tipo vale, então podemos randomizar os dados de entrada e esperar que as propriedades sejam cumpridas.

```python
def test_add_comutativo():
    # exemplares
    x, y = faker.pyint(), faker.pyint()
```



"Mas eu sou dev xpto" não escrevo funções de soma...

Para web:

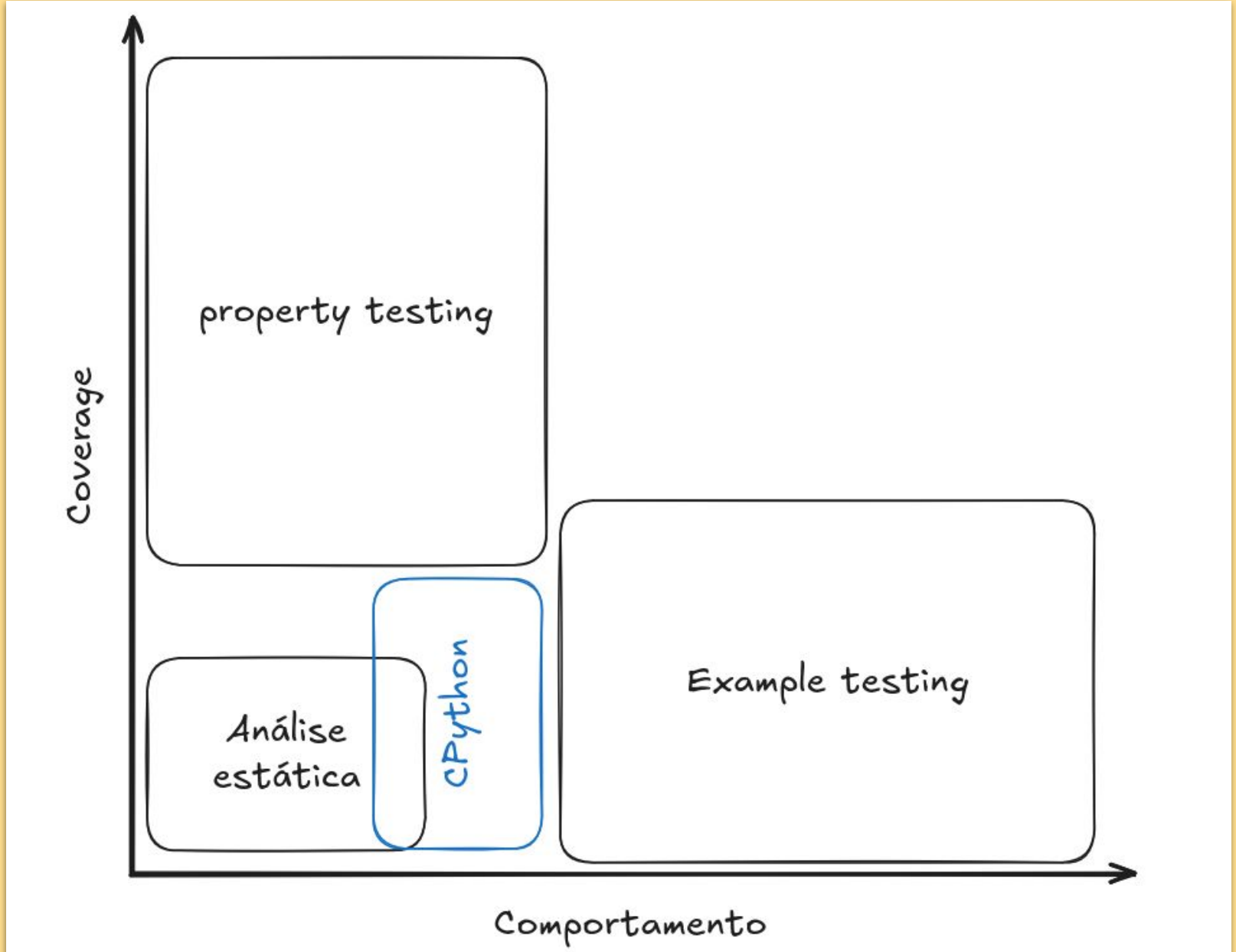
- Propriedades em testes de API: status code, body (json), headers, ...
- Schemas: Forma dos dados, tipos, restrições, ...
- Formulários, webscoket, ...

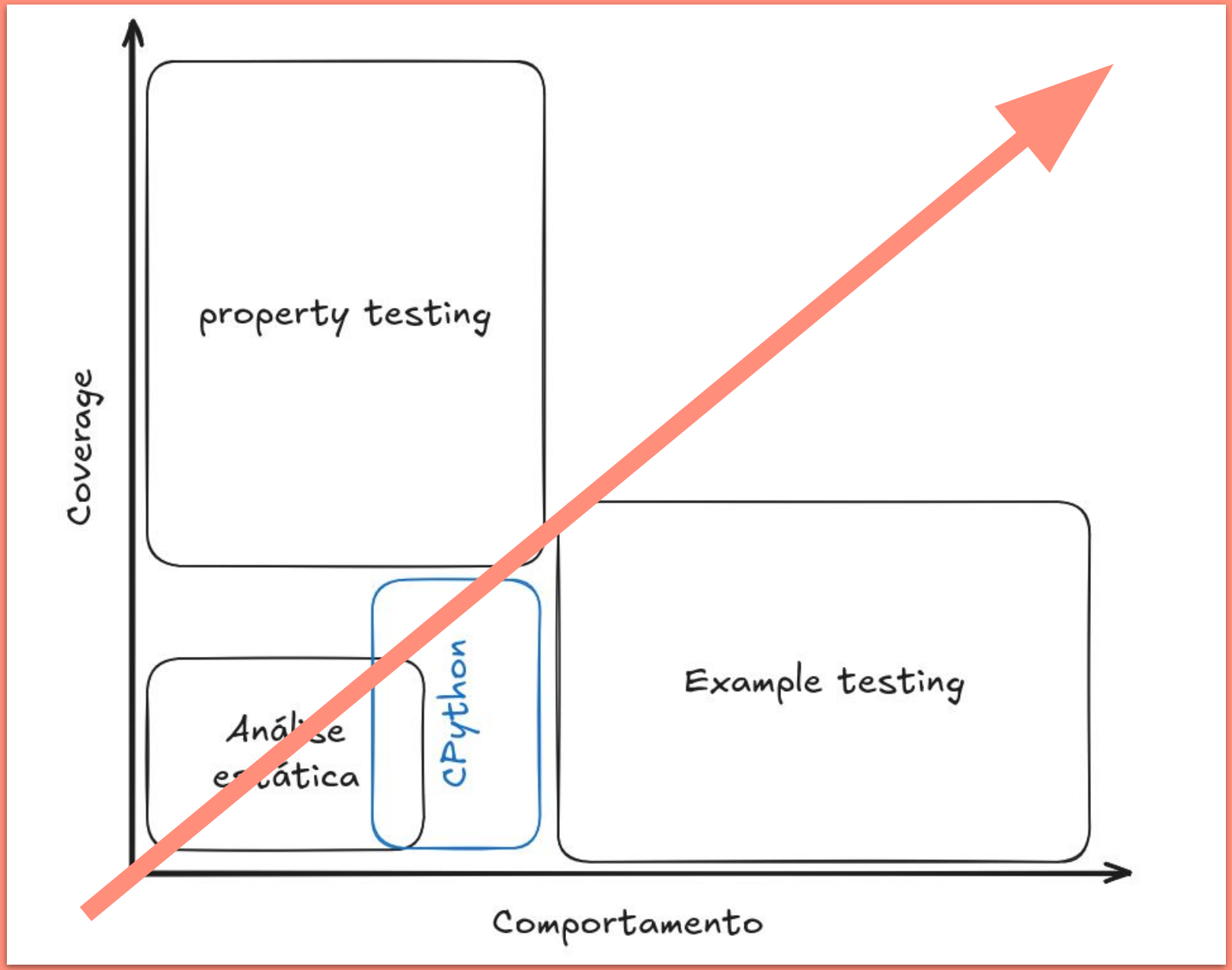
Para DS:

- Validadores
  - Condicionais, limpeza de dados, tipos de campos, grandezas de dados, ...
- Splits: reino e validação

Ambos:

- Modelos de dados (tabelas de banco): Persistência


property testing
Coverage
Análise estática
cPython
Example testing
Comportamento


property testing
Coverage
Análise estática
cPython
Example testing
Comportamento

Estratégias de randomização

Hypothesis

Hypothesis

É uma biblioteca criada para fazer PBT em testes de unidade usando **estratégias** de inputs randomizados.

- Criada por: David MacIver e Zac Hatfield-Dodds
- Licença: **MPL**
- Primeira release: **2013**
- Release atual: **6.130.5** (março/2025)
- Duas forma de funcionamento:
  - **CLI** - Interface de linha de comando
  - **Biblioteca**

```
pip install hypothesis
```

Os componentes básicos

O básico trivial do Hypothesis é o uso de decoradores para funções/métodos de testes. Como:

- **@given**: Passa argumentos randomizados para as funções de teste
- **@example**: Fornece um exemplo do estilo **EBT**
- **@settings**: Configura diversas coisas no teste, controles, casos, verbosidade, ...

@given [exemplo_04.py]

Podemos randomizar os dados de um teste usando somente @given e anotando os parâmetros do teste. Algo como:

```python
from hypothesis import given

@given(...)
def test_add_comutativo(x: int, y: int):
    assert add(x, y) == add(y, x)
```

Isso executará o teste 100 vezes (o padrão) com valores randomizados para os parâmetros da função respeitando o tipo anotado.

@example [exemplo_05.py]

O decorado **@example** pode ser usado para os exemplos que normalmente seriam usados em um EBT, sem ter que escrever um novo teste

```python
from hypothesis import given, example

@given(...)
@example(x=1, y=1)
@example(x=-1000, y=17)
def test_add_comutativo(x: int, y: int):
    assert add(x, y) == add(y, x)
```

@settings [exemplo_06.py]

O decorador **@settings** faz muitas coisas, talvez as mais simples, pra gente começar a brincar sejam a quantidade de casos e a verbosidade:

```python
from hypothesis import Verbosity, example, given, settings

@given(...)
@example(x=-1000, y=17)
@settings(max_examples=50, verbosity=Verbosity.verbose)
def test_add_comutativo(x: int, y: int):
    assert add(x, y) == add(y, x)
```

Estratégias

Parte fundamental de lidar com o hypothesis e descrever a ele como os dados randomizados serão, quais seus tipos, suas limitações, seu tamanho...

Chamamos essas características de estratégias. Temos alguns grupos, como:

- **Primitivas**: none, nothing, just, boolean
- **Numéricas**: int, float, complex, decimal, fraction
- **Strings**: text, chars, emails, domains, urls, ...
- **Coleções**: list, tuple, sets, dicts, iterables, ...
- **Datetimes**: date, time, datetime, timezone, ...
- ...

Testando a nossa propriedade [exemplo_07.py]

Aplicando a estratégia de inteiros, teríamos um teste como esse:

```python
from hypothesis import given
from hypothesis.strategies import integers

@given(integers(), integers())
def test_add_comutativo(x, y):
    assert add(x, y) == add(y, x)
```

Parâmetros [exemplo_08.py]

Tava mais fácil usando type hints...

Uma das coisas interessantes sobre a estratégias é que elas podem ser parametrizadas.

```python
@given(integers(-100, 0), integers(0, 100))
def test_add_comutativo(x, y):
    assert add(x, y) == add(y, x)
```

Cada estratégia tem seus própios parâmetros: https://hypothesis.readthedocs.io/en/latest/reference/strategies.html

Modificadores [exemplos_09.py]

Tava mais fácil usando type hints...

Bom, as estratégias podem usar modificadores, como **filter**, **map** e **flatmap**

```
@given(
    integers(-100, 0).filter(lambda x: x % 2 == 0), # pares
    integers(0, 100).map(lambda x: x ** 3)  # cúbicos
)
def test_add_comutativo(x, y):
    assert add(x, y) == add(y, x)
```

Compondo estratégias [exemplo_10.py]

```python
from hypothesis import given
from hypothesis.strategies import lists, text

def concat(x: list[str], y: list[str]) -> list[str]:
    return x + y

@given(lists(text()), lists(text()))
def test_concat_size(x, y):
    result = concat(x, y)
    assert len(result) == len(x) + len(y)
```

E eu com isso?

Usando isso de "verdade"...

Vamos lá...

Estratégias para quem tem pressa:

- **composite**: Cria uma estratégia se baseando em um conjunto das mesmas
- **from_type**: Cria um exemplo de qualquer coisa que tenha o tipo anotado
- **builds**: Cria estratégias com tipos, mas permite parametrização

Um caso de validação [exemplo_11.py]

Vamos supor que você tenha validar um dicionário qualquer, que tenha alguns dados:

```python
def validation(data: dict) -> bool:
    return all([
        # Nenhum campo é vazio
        all([value for value in data.values()]),
        # Idade maior que 18
        data['idade'] >= 18,
        # cpf é formado por números
        data['cpf'].isnumeric(),
        # cpf tem o tamanho certo
        len(data['cpf']) == 11
    ])
```

@composite [exemplo_11.py]

A ideia do composite é usar várias estratégias e desenhar a sua própria. Algo como:

```python
from hypothesis import strategies as st

@st.composite
def custom_dict(draw):
    return {
        'idade': draw(st.integers(min_value=18)),
        'nome': draw(st.text(min_size=3)),
        'cpf': draw(st.from_regex(r'\d{11}', fullmatch=True))
    }
```

```python
from hypothesis import strategies as st

@st.composite
def custom_dict(draw):
    return {
        'idade': draw(st.integers(min_value=18)),
        'nome': draw(st.text(min_size=3)),
        'cpf': draw(st.from_regex(r'\d{11}', fullmatch=True))
    }
```

```python
@given(custom_dict())
def test_validation(data):
    assert validation(data)
```

Estratégia from_type [exemplo_12.py]

```python
from dataclasses import dataclass, asdict
from hypothesis import given
from hypothesis import strategies as st

@dataclass
class Data:
    nome: str
    idade: int
    cpf: str

@given(st.from_type(Data))
def test_validation(data):
    assert validation(asdict(data))
```

Estratégia builds [exemplo_13.py]

builds podem executar tipos, assim como **from_type**, mas com alterações em parâmetros de estratégia

```python
from pydantic import BaseModel, Field

class Data(BaseModel):
    nome: str = Field(min_length=3)
    idade: int = Field(ge=18)
    cpf: str = Field(min_length=11, max_length=11, pattern=r'\d{11}')


@given(st.builds(Data, cpf=st.from_regex(r'\d{11}', fullmatch=True)))
def test_validation(data):
    assert validation(data.model_dump())
```

Um exemplo com FastAPI [api.py e exemplo_14.py]

```python
@app.post('/create', response_model=TodoOut, status_code=201)
def create_todo(
    todo: TodoIn, session: Annotated[Session, Depends(get_session)]
    # pydantic  , SQlalchemy.Session
):
    try:
        todo_db = Todo(**todo.model_dump())
        session.add(todo_db)
        session.commit()
        session.refresh(todo_db)
        return todo_db
    except IntegrityError:
        session.rollback()
        raise HTTPException(status_code=400)
```

Um exemplo com FastAPI [api.py e exemplo_14.py]

```python
@app.post('/create', response_model=TodoOut, status_code=201)
def create_todo(
    todo: TodoIn, session: Annotated[Session, Depends(get_session)]
    # pydantic  , SQlalchemy.Session
):
    try:
        todo_db = Todo(**todo.model_dump())
        session.add(todo_db)
        session.commit()
        session.refresh(todo_db)
        return todo_db
    except IntegrityError:
        session.rollback()
        raise HTTPException(status_code=400
```

```python
class TodoIn(BaseModel):
    name: str = Field(min_length=4)
    description: str = Field(default='')


class TodoOut(TodoIn):
    id: int
```

Um exemplo com FastAPI [api.py e exemplo_14.py]

```python
@app.post('/create', response_model=TodoOut, status_code=201)
def create_todo(
    todo: TodoIn, session: Annotated[Session, Depends(get_session)]
    # pydantic  , SQlalchemy.Session
):
    try:
        todo_db = Todo(**todo.model_dump())
        session.add(todo_db)
        session.commit()
        session.refresh(todo_db)
```

```python
class TodoIn(BaseModel):
    name: str = Field(min_length=4)
                d(default='')
```

```python
@reg.mapped_as_dataclass
class Todo:
    __tablename__ = 'todos'

    id: Mapped[int] = mapped_column(primary_key=True, init=False)
    name: Mapped[str] = mapped_column(nullable=False)
    description: Mapped[str] = mapped_column(nullable=True)
```

Um pequeno problema com fixtures antes...

Quando um teste com fixture é executado, a fixture não é executada todas as vezes...

```python
@given(st.builds(TodoIn))
@settings(
    suppress_health_check=[HealthCheck.function_scoped_fixture],
)
def test_create_todo_hypo(client: TestClient, todo):
    response = client.post('/create', json=todo.model_dump())
    assert response.status_code == HTTPStatus.CREATED
    assert TodoOut(**response.json())
```

Bom...

Várias coisas não foram mostradas aqui e que são incríveis:

- Estratégias de escopo compartilhado
- Máquinas de estado
- HypoFuzz
- Schemathesis
- ...

Como diz o título... Uma introdução. Mais lives sobre o assunto podem ser feitas <3

apoia.se/livedepython

pix.dunossauro@gmail.com

patreon.com/dunossauro

Ajude o projeto <3

Referências

- https://dl.acm.org/doi/10.1145/3597503.3639581
- https://hypothesis.readthedocs.io/en/latest/
- https://hypothesis.works/
- https://hypofuzz.com/
- https://hypofuzz.com/docs/literature.html
- https://schemathesis.readthedocs.io/en/stable/index.html

—

- https://excalidraw.com/#json=8XYVQ8zai5e2Klm0zHQet,Fugfd3Fr_GG7G42XK-QDCA

Vídeos citados

- Uma introdução aos testes: Como fazer?: https://youtu.be/-8H2Pyxnoek
- O mínimo que você deveria saber sobre testes unitários: https://youtu.be/pZvhZ-Lr-PE
- Pytest: Uma introdução: https://youtu.be/MjQCvJmc31A
- Randomização de dados em testes unitários com Faker e Factory-boy: https://youtu.be/q_P-2h5L1cE