

**UNIVERSIDADE FEDERAL DE SERGIPE
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
DEPARTAMENTO DE COMPUTAÇÃO**

**GUSTAVO HENRIQUE MARQUES
HENRIQUE ROCHA VALENTIM BASTOS
MARIANA SOUZA NUNES
RAFAEL LAUTON SANTOS DE OLIVEIRA**

ATIVIDADE 3 – TESTES DE MUTAÇÃO

São Cristóvão - SE

2025

SUMÁRIO

1.Introdução.....	3
2. Desenvolvimento.....	4
2.1. Roteiro do Vídeo.....	4
2.2. Python Validators.....	7
2.3. Execução dos Testes.....	9
2.4. Análise com Mutmut.....	13
3. Conclusão.....	24
4. Links.....	25
5. Referências.....	25

1. Introdução

O desenvolvimento de software envolve a criação de sistemas cada vez mais complexos, o que torna essencial garantir a qualidade e a confiabilidade dos programas. Nesse contexto, os testes de software surgem como uma prática fundamental para identificar erros, validar funcionalidades e assegurar que o comportamento do sistema esteja de acordo com os requisitos especificados. Testes tradicionais, como testes unitários, de integração e funcionais, focam em verificar se o código executa corretamente as operações previstas, mas muitas vezes não são suficientes para detectar falhas sutis que podem comprometer o sistema.

Dentro dessa perspectiva, os testes de mutação se destacam como uma técnica avançada de avaliação da efetividade dos testes existentes. Essa abordagem consiste em introduzir pequenas alterações, chamadas de mutantes, no código-fonte do programa para simular possíveis erros. Em seguida, os testes já implementados são executados para verificar se conseguem detectar essas mudanças.

A principal vantagem dos testes de mutação é fornecer uma medida precisa da capacidade de detecção de falhas da suíte de testes, permitindo identificar pontos frágeis e aumentar a confiabilidade do software. Contudo, essa técnica também apresenta desafios, como o alto custo computacional devido à necessidade de gerar e testar múltiplos mutantes, além da complexidade de interpretar resultados em grandes bases de código. Apesar disso, os testes de mutação têm se mostrado uma ferramenta poderosa para melhorar a qualidade do software e complementar métodos de teste tradicionais.

O presente trabalho tem como objetivo aplicar testes de mutação em projetos desenvolvidos em Python, utilizando ferramentas como `pytest` para execução de testes unitários, `pytest-cov` para análise de cobertura e `mutmut` para avaliação da eficácia dos casos de teste. A atividade envolve a seleção de um projeto real, a execução do teste de mutantes, a análise dos resultados obtidos, e a proposição de melhorias nos testes existentes, com a finalidade de aumentar a detecção de mutantes e aprimorar a qualidade do software.

2. Desenvolvimento

2.1. Roteiro do Vídeo

O vídeo utilizado como base para a atividade apresentou de forma prática a configuração de um ambiente de testes em Python com o objetivo de aplicar tanto testes unitários tradicionais quanto técnicas de análise de cobertura e mutação.

Inicialmente, foi demonstrado o processo de criação de um ambiente virtual, recurso essencial para garantir o isolamento das dependências e a reprodutibilidade dos experimentos. Em seguida, foram instaladas as ferramentas fundamentais para a execução da atividade: pytest, pytest-cov e mutmut.

O pytest foi escolhido como framework de testes por sua simplicidade e ampla adoção na comunidade Python. O pytest-cov foi utilizado para mensurar a cobertura dos testes, isto é, identificar quais partes do código foram efetivamente exercitadas durante a execução. Por fim, o mutmut foi empregado para aplicar a técnica de teste de mutação, que consiste em introduzir pequenas modificações artificiais no código-fonte com o intuito de verificar a capacidade dos testes em detectar tais alterações.

O vídeo também mostrou como executar os testes de forma incremental. Primeiramente, rodaram-se os casos de teste já disponíveis no projeto com o comando pytest.

```
(venv) gs@gustavo:~/cal_python$ pytest -vv test_cal.py
===== test session starts =====
platform linux -- Python 3.12.3, pytest-8.4.1, pluggy-1.6.0 -- /home/gs/cal_python/venv/bin/python3
cachedir: .pytest_cache
rootdir: /home/gs/cal_python
plugins: cov-6.2.1
collected 21 items

test_cal.py::test_cal PASSED [ 43%]
test_cal.py::test_is_leap[4-True] PASSED [ 90%]
test_cal.py::test_is_leap[1501-False] PASSED [ 143%]
test_cal.py::test_is_leap[1900-False] PASSED [ 190%]
test_cal.py::test_is_leap[1903-False] PASSED [ 233%]
test_cal.py::test_is_leap[1904-True] PASSED [ 280%]
test_cal.py::test_is_leap[2000-True] PASSED [ 327%]
test_cal.py::test_mutation_is_leap SKIPPED (simply ignoring for now) [ 380%]
test_cal.py::test_first_of_month[input0-6] PASSED [ 427%]
test_cal.py::test_first_of_month[input1-2] PASSED [ 474%]
test_cal.py::test_first_of_month[input2-4] PASSED [ 521%]
test_cal.py::test_first_of_month[input3-2] PASSED [ 568%]
test_cal.py::test_first_of_month[input4-3] PASSED [ 615%]
test_cal.py::test_jan1[1801-4] PASSED [ 662%]
test_cal.py::test_jan1[1700-6] PASSED [ 709%]
test_cal.py::test_jan1[1500-3] PASSED [ 756%]
test_cal.py::test_number_of_days[input0-31] PASSED [ 803%]
test_cal.py::test_number_of_days[input1-29] PASSED [ 850%]
test_cal.py::test_number_of_days[input2-31] PASSED [ 897%]
test_cal.py::test_number_of_days[input3-19] PASSED [ 944%]
test_cal.py::test_number_of_days[input4-31] PASSED [ 1000%]

===== 20 passed, 1 skipped in 0.06s =====
```

Em seguida, com a inclusão da opção `--cov`, foi possível observar o percentual de cobertura do código, detalhando quais arquivos e funções foram devidamente testados.

```
(venv) gs@gustavo:~/cal_python$ pytest -vv test_cal.py --cov=cal
===== test session starts =====
platform linux -- Python 3.12.3, pytest-8.4.1, pluggy-1.6.0 -- /home/gs/cal_python/venv/bin/python3
cachedir: .pytest_cache
rootdir: /home/gs/cal_python
plugins: cov-6.2.1
collected 21 items

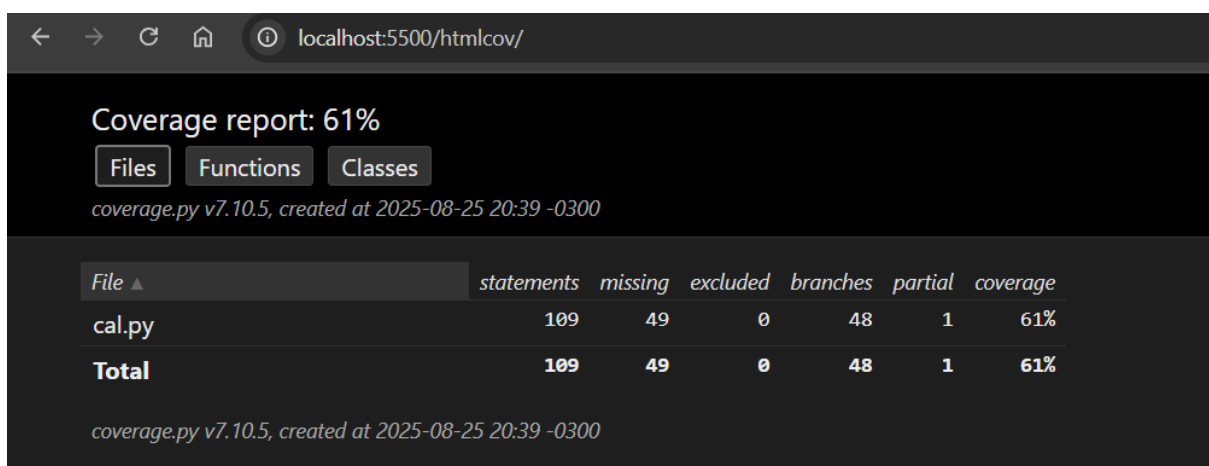
test_cal.py::test_cal PASSED [ 4%]
test_cal.py::test_is_leap[4-True] PASSED [ 9%]
test_cal.py::test_is_leap[1501-False] PASSED [ 14%]
test_cal.py::test_is_leap[1900-False] PASSED [ 19%]
test_cal.py::test_is_leap[1903-False] PASSED [ 23%]
test_cal.py::test_is_leap[1904-True] PASSED [ 28%]
test_cal.py::test_is_leap[2000-True] PASSED [ 33%]
test_cal.py::test_mutation_is_leap SKIPPED (simply ignoring for now) [ 38%]
test_cal.py::test_first_of_month[input0-6] PASSED [ 42%]
test_cal.py::test_first_of_month[input1-2] PASSED [ 47%]
test_cal.py::test_first_of_month[input2-4] PASSED [ 52%]
test_cal.py::test_first_of_month[input3-2] PASSED [ 57%]
test_cal.py::test_first_of_month[input4-3] PASSED [ 61%]
test_cal.py::test_jan1[1801-4] PASSED [ 66%]
test_cal.py::test_jan1[1700-6] PASSED [ 71%]
test_cal.py::test_jan1[1500-3] PASSED [ 76%]
test_cal.py::test_number_of_days[input0-31] PASSED [ 80%]
test_cal.py::test_number_of_days[input1-29] PASSED [ 85%]
test_cal.py::test_number_of_days[input2-31] PASSED [ 90%]
test_cal.py::test_number_of_days[input3-19] PASSED [ 95%]
test_cal.py::test_number_of_days[input4-31] PASSED [100%]

===== tests coverage =====
coverage: platform linux, python 3.12.3-final-0

Name      Stmts  Miss  Cover
-----
cal.py     109    49    55%
TOTAL     109    49    55%

===== 20 passed, 1 skipped in 0.24s =====
```

O vídeo também evidencia a geração de arquivos em HTML para documentação e análise posterior com o `pytest-cov`, mostrando a emissão do relatório HTML de cobertura, que cria a pasta `htmlcov/` com mapas de linhas executadas e não executadas, facilitando a identificação de lacunas. Esses artefatos permitem não apenas auditar a execução, mas também comunicar resultados de maneira clara e reproduzível.



← → ↺ 🏠 ⓘ localhost:5500/htmlcov/

Coverage report: 61%

[Files](#) [Functions](#) [Classes](#)

coverage.py v7.10.5, created at 2025-08-25 20:39 -0300

File ▲	statements	missing	excluded	branches	partial	coverage
cal.py	109	49	0	48	1	61%
Total	109	49	0	48	1	61%

coverage.py v7.10.5, created at 2025-08-25 20:39 -0300

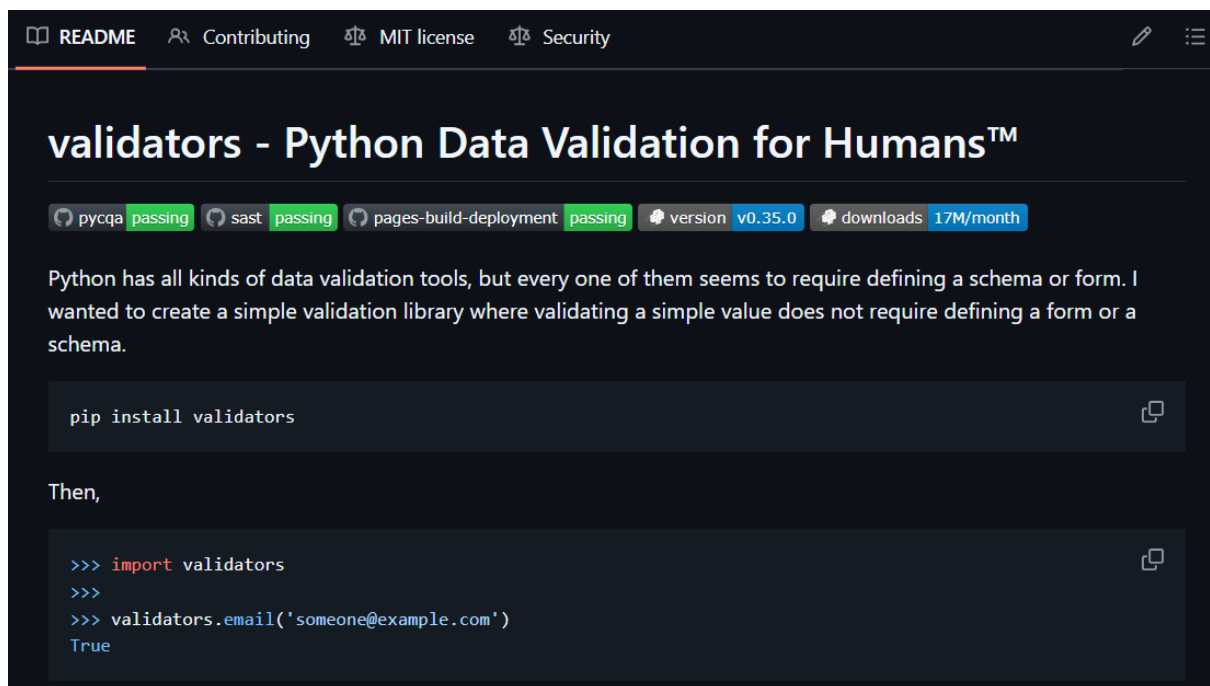
Por último, foi demonstrado o uso do mutmut, desde sua configuração até a execução completa dos testes de mutação.

```
• (venv) gs@gustavo:~/cal_python$ mutmut run
  ⚡ Generating mutants
    done in -945ms
  ⚡ Running stats
    done
  ⚡ Running clean tests
    done
  ⚡ Running forced fail test
    done
Running mutation testing
⚡ 146/146 🎉 128 🤖 0 🕒 0 🧐 0 🧐 18 🚫 0
19.65 mutations/second
```

O vídeo ressalta a importância dessa ferramenta como complemento à cobertura, pois ela avalia não apenas se uma linha de código foi executada, mas também se as asserções adequadas estão presentes para detectar falhas. Dessa forma, foi possível visualizar relatórios com o número de mutantes gerados, eliminados e sobreviventes, fornecendo uma visão crítica sobre a efetividade dos testes originais.

2.2. Python Validators

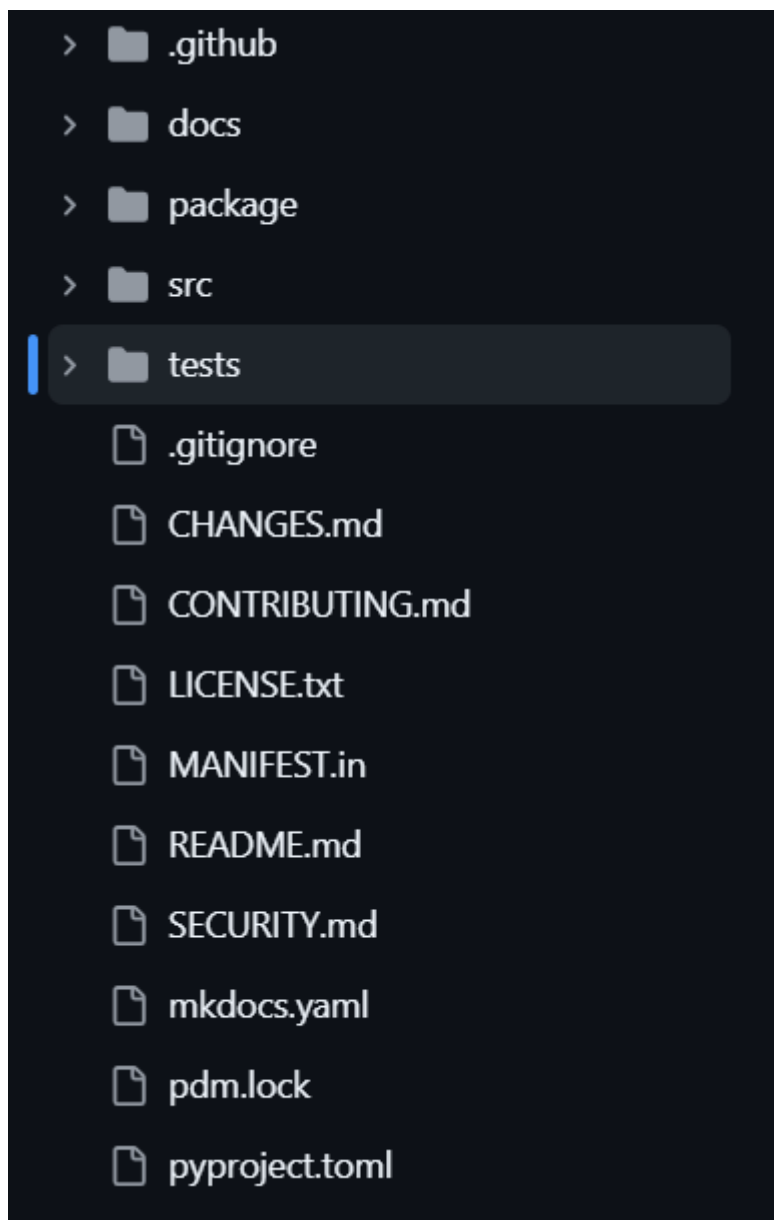
O segundo projeto selecionado para a atividade foi o Python Validators, uma biblioteca de código aberto amplamente utilizada para validação de dados em aplicações Python.



Trata-se de um pacote leve, mas bastante abrangente, que disponibiliza diversas funções auxiliares para verificar se valores seguem determinados formatos ou padrões, como endereços de e-mail, URLs, domínios, endereços IP (IPv4 e IPv6), números de cartão de crédito, entre outros. Por se tratar de um projeto mantido ativamente e com múltiplas funções independentes, ele se mostra um caso interessante para a aplicação de testes de mutação, já que permite observar como pequenas alterações no código podem comprometer a confiabilidade dos validadores.

A estrutura do projeto segue os padrões convencionais de bibliotecas Python, possuindo uma pasta `src/validators/` onde estão localizados os módulos principais e uma pasta `tests/` com a suíte de testes automatizados já disponibilizada pelos mantenedores. Essa organização facilita a integração com ferramentas como `pytest`, `coverage` e `mutmut`, permitindo a execução direta dos testes e a análise da cobertura. Por ser um projeto relativamente pequeno em termos de código-fonte, mas com uma

quantidade razoável de funções de validação, ele oferece um equilíbrio adequado entre simplicidade e diversidade de cenários de teste.



2.3. Execução dos Testes

Após a identificação do projeto *validators*, realizou-se a execução da suíte de testes já disponibilizada pelos desenvolvedores, utilizando a ferramenta Pytest. Seguindo os passos do vídeo de referência, primeiro foi criado o ambiente virtual:

```
gs@gustavo:~/validators$ python3 -m venv myenv
gs@gustavo:~/validators$ source myenv/bin/activate
(myenv) gs@gustavo:~/validators$
```

Após isso, foi preciso baixar as bibliotecas necessárias para a execução dos testes: *pytest*, *pytest-cov* e *mutmut*. Além desses, o próprio projeto conta com um arquivo *requirements.testing.txt* que contém outros módulos necessários para a execução dos testes.

```
(myenv) gs@gustavo:~/validators$ pip install -r /home/gustavo/validators/package/requirements.testing.txt
Ignoring exceptiongroup: markers 'python_version < "3.11"' don't match your environment
Ignoring tomli: markers 'python_version < "3.11"' don't match your environment
Collecting colorama==0.4.6 (from -r /home/gustavo/validators/package/requirements.testing.txt (line 4))
  Using cached colorama-0.4.6-py2.py3-none-any.whl (25 kB)
Collecting eth-hash==0.7.0 (from eth-hash[pycryptodome]==0.7.0->-r /home/gustavo/validators/package/requirements.testing.txt (line 7))
  Downloading eth_hash-0.7.0-py3-none-any.whl (8.7 kB)
Collecting iniconfig==2.0.0 (from -r /home/gustavo/validators/package/requirements.testing.txt (line 13))
  Using cached iniconfig-2.0.0-py3-none-any.whl (5.9 kB)
Collecting packaging==24.1 (from -r /home/gustavo/validators/package/requirements.testing.txt (line 16))
  Using cached packaging-24.1-py3-none-any.whl (53 kB)
Collecting pluggy==1.5.0 (from -r /home/gustavo/validators/package/requirements.testing.txt (line 19))
  Using cached pluggy-1.5.0-py3-none-any.whl (20 kB)
Collecting pycryptodome==3.20.0 (from -r /home/gustavo/validators/package/requirements.testing.txt (line 22))
  Downloading pycryptodome-3.20.0-cp35-abi3-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (2.1 MB)
  2.1/2.1 MB 4.7 MB/s eta 0:00:00
Collecting pytest==8.3.2 (from -r /home/gustavo/validators/package/requirements.testing.txt (line 44))
  Downloading pytest-8.3.2-py3-none-any.whl (341 kB)
  341.8/341.8 kB 6.3 MB/s eta 0:00:00
Installing collected packages: pycryptodome, pluggy, packaging, iniconfig, eth-hash, colorama, pytest
Successfully installed colorama-0.4.6 eth-hash-0.7.0 iniconfig-2.0.0 packaging-24.1 pluggy-1.5.0 pycryptodome-3.20.0 pytest-8.3.2
(myenv) gs@gustavo:~/validators$
```

A primeira execução dos testes foi com o comando `pytest`, que automaticamente busca por arquivos de teste e executa-os.

```
(myenv) gs@gustavo:~/validators$ pytest
===== test session starts =====
platform linux -- Python 3.12.3, pytest-8.3.2, pluggy-1.5.0
rootdir: /home/gs/validators
configfile: pyproject.toml
testpaths: tests
collected 895 items

tests/crypto_addresses/test_bsc_address.py ..... [ 2%]
tests/crypto_addresses/test_btc_address.py ..... [ 3%]
tests/crypto_addresses/test_eth_address.py ..... [ 5%]
tests/crypto_addresses/test_trx_address.py ..... [ 8%]
tests/i18n/test_es.py ..... [15%]
tests/i18n/test_fr.py ..... [18%]
tests/i18n/test_fr.py ..... [22%]
tests/i18n/test_ind.py ..... [23%]
tests/i18n/test_ru.py ..... [25%]
tests/test_extremes.py ..... [26%]
tests/test_between.py ..... [28%]
tests/test_card.py ..... [42%]
tests/test_country.py ..... [45%]
tests/test_cron.py ..... [48%]
tests/test_domain.py ..... [54%]
tests/test_email.py ..... [57%]
tests/test_encoding.py ..... [62%]
tests/test_finance.py ..... [64%]
tests/test_hashes.py ..... [69%]
tests/test_hostname.py ..... [73%]
tests/test_iban.py ..... [73%]
tests/test_ip_address.py ..... [80%]
tests/test_length.py ..... [81%]
tests/test_mac_address.py ..... [82%]
tests/test_slug.py ..... [83%]
tests/test_url.py ..... [96%]
tests/test_uuid.py ..... [96%]
tests/test_validation_failure.py .... [100%]

===== 895 passed in 0.37s =====
```

Conforme demonstrado pela saída do framework `pytest`, um total de 895 casos de teste foram coletados e executados com sucesso em um tempo de 0.37 segundos, utilizando um ambiente com Python 3.12.3 e `pytest` 8.2.2. Este resultado confirma que o código-fonte, em seu estado original, está em conformidade com as especificações e asserções definidas pela equipe de desenvolvimento.

O volume de 895 testes sinaliza um esforço significativo para a verificação do comportamento de diversas funcionalidades. Além disso, a diversidade dos módulos testados, que englobam desde validações de endereços de criptomoedas (`test_btc_address.py`), formatos financeiros (`test_iban.py`, `test_finance.py`) até padrões de rede (`test_email.py`, `test_url.py`), demonstra a ampla gama de validações cobertas. Essa extensa verificação inicial confere um grau de confiança elevado na correte funcional do sistema sob as condições já previstas pelos desenvolvedores. Em seguida, foi utilizado o comando `pytest -vv --cov=src/validators` para descobrir a porcentagem de cobertura dos testes em relação ao código fonte na pasta `src/validators`.

===== tests coverage =====			
coverage: platform linux, python 3.12.3-final-0			
Name	Stmts	Miss	Cover
src/validators/_init_.py	22	0	100%
src/validators/_extremes.py	10	0	100%
src/validators/between.py	19	0	100%
src/validators/card.py	45	1	98%
src/validators/country.py	42	3	93%
src/validators/cron.py	44	4	91%
src/validators/crypto_addresses/_init_.py	5	0	100%
src/validators/crypto_addresses/bsc_address.py	9	1	89%
src/validators/crypto_addresses/btc_address.py	17	1	94%
src/validators/crypto_addresses/eth_address.py	23	4	83%
src/validators/crypto_addresses/trx_address.py	25	5	80%
src/validators/domain.py	36	3	92%
src/validators/email.py	15	3	80%
src/validators/encoding.py	14	0	100%
src/validators/finance.py	60	5	92%
src/validators/hashes.py	20	0	100%
src/validators/hostname.py	29	0	100%
src/validators/i18n/_init_.py	6	0	100%
src/validators/i18n/es.py	39	0	100%
src/validators/i18n/fi.py	30	0	100%
src/validators/i18n/fr.py	39	1	97%
src/validators/i18n/ind.py	8	0	100%
src/validators/i18n/ru.py	20	4	80%
src/validators/iban.py	10	0	100%
src/validators/ip_address.py	34	4	88%
src/validators/length.py	10	0	100%
src/validators/mac_address.py	5	0	100%
src/validators/slug.py	5	0	100%
src/validators/url.py	34	34	0%
src/validators/url.py	54	3	94%
src/validators/utls.py	39	7	82%
src/validators/uuid.py	14	1	93%
TOTAL	782	84	89%
===== 895 passed in 4.96s =====			

O relatório gerado pela ferramenta pytest-cov indica uma robusta cobertura geral de 89% para o projeto python-validators. Este percentual é derivado de um total de 782 declarações (statements) executáveis no código, das quais 698 foram executadas e 84 permaneceram não alcançadas pela suíte de testes. Este é um forte indicador de que a maior parte da lógica da aplicação está sendo, no mínimo, exercitada durante a fase de verificação.

Uma análise mais granular do relatório revela uma distribuição heterogênea da cobertura entre os diferentes módulos do sistema. Enquanto componentes de alta utilização, como country.py, email.py e mac_address.py, demonstram uma cobertura exemplar de 100%, outros módulos apresentam oportunidades claras para aprimoramento. Notavelmente, o módulo crypto_addresses/eth_address.py, com 81% de cobertura, e url.py, com 90%, são identificados como pontos que requerem atenção. Essa disparidade sugere que, embora a estratégia de testes seja madura, existem áreas específicas cuja lógica não está sendo completamente verificada, representando um risco potencial que poderia ser mitigado com a criação de novos casos de teste direcionados a essas 84 linhas de código não cobertas.

Para gerar o arquivo HTML auxiliar, facilitando a análise dos testes, foi utilizado a seguinte linha de comando: `pytest -vv --cov=src/validators --cov-report html`.

<div> <div>←</div> <div>→</div> <div>↺</div> <div>🏠</div> <div>🔍 localhost:5500/htmlcov/</div> </div>				
<div> <div>Coverage report: 89%</div> <div> <div>Files</div> <div>Functions</div> <div>Classes</div> </div> <div>coverage.py v7.10.5, created at 2025-08-25 21:30 -0300</div> </div>				
File ▲	statements	missing	excluded	coverage
src/validators/__init__.py	22	0	0	100%
src/validators/_extremes.py	10	0	0	100%
src/validators/between.py	19	0	0	100%
src/validators/card.py	45	1	0	98%
src/validators/country.py	42	3	0	93%
src/validators/cron.py	44	4	0	91%
src/validators/crypto_addresses/__init__.py	5	0	0	100%
src/validators/crypto_addresses/bsc_address.py	9	1	0	89%
src/validators/crypto_addresses/btc_address.py	17	1	0	94%
src/validators/crypto_addresses/eth_address.py	23	4	0	83%
src/validators/crypto_addresses/trx_address.py	25	5	0	80%

2.4. Análise com Mutmut

A etapa seguinte consistiu na aplicação de testes de mutação com a ferramenta Mutmut, cujo objetivo é avaliar a qualidade da suíte de testes já existente no projeto. Diferentemente da análise de cobertura realizada com o *pytest-cov*, que apenas indica quais linhas de código foram executadas durante os testes, o Mutmut introduz modificações artificiais (mutantes) no código e verifica se os testes são capazes de identificá-las. Quando os testes falham diante de um mutante, ele é considerado “morto”; caso contrário, o mutante “sobrevive” e indica uma possível fragilidade na suíte de testes.

No caso do projeto *validators*, o comando `mutmut run` foi executado para a geração e execução dos mutantes.

```
● (myenv) gs@gustavo:~/validators$ mutmut run
  ∷ Generating mutants
    done in 2071ms
  ∷ Running stats
    done
  ∷ Running clean tests
    done
  ∷ Running forced fail test
    done
Running mutation testing
  ∷ 689/689 🇺🇦 462 😬 12 🕒 0 🤖 0 😞 215 🚫 0
19.53 mutations/second
○ (myenv) gs@gustavo:~/validators$
```

A análise quantitativa dos resultados revela uma eficácia parcial da suíte de testes. Dos 689 mutantes gerados, 462 foram "mortos" com sucesso, indicando que os testes falharam como esperado. Adicionalmente, 12 mutantes foram eliminados por excederem o tempo de execução (timeout), o que também é considerado uma detecção. Isso resulta em um total de 474 mutantes neutralizados e uma Pontuação de Mutação (Mutation Score) de aproximadamente 69%. Contudo, o dado mais crítico da análise é a sobrevivência de 215 mutantes. Este número significa que em 31% dos cenários de falha simulada, a suíte de testes permaneceu silente, aprovando um código que continha um defeito deliberado. A ausência de mutantes em código não

coberto (uncovered mutants: 0) é um achado crucial, pois comprova que as falhas não estão em áreas ignoradas pelos testes, mas sim na qualidade e na profundidade das asserções aplicadas sobre o código que já é coberto.

Posteriormente, utilizou-se o comando `mutmut results` para visualizar o resumo do processo, identificando a quantidade de mutantes mortos, sobreviventes e suspeitos.

```
• (myenv) gs@gustavo:~/validators$ mutmut results
validators.uri.x__file_url__mutmut_1: no tests
validators.uri.x__file_url__mutmut_2: no tests
validators.uri.x__file_url__mutmut_3: no tests
validators.uri.x__file_url__mutmut_4: no tests
validators.uri.x__file_url__mutmut_5: no tests
validators.uri.x__file_url__mutmut_6: no tests
validators.uri.x__ipfs_url__mutmut_1: no tests
validators.uri.x__ipfs_url__mutmut_2: no tests
validators.uri.x__ipfs_url__mutmut_3: no tests
validators.uri.x__ipfs_url__mutmut_4: no tests
validators.uri.x__ipfs_url__mutmut_5: no tests
validators.uri.x__ipfs_url__mutmut_6: no tests
validators.finance.x__cusip_checksum__mutmut_17: survived
validators.finance.x__cusip_checksum__mutmut_20: survived
validators.finance.x__cusip_checksum__mutmut_21: survived
validators.finance.x__cusip_checksum__mutmut_31: survived
validators.finance.x__cusip_checksum__mutmut_32: survived
validators.finance.x__cusip_checksum__mutmut_33: survived
validators.finance.x__cusip_checksum__mutmut_34: survived
validators.finance.x__cusip_checksum__mutmut_35: survived
validators.finance.x__cusip_checksum__mutmut_46: survived
validators.finance.x__cusip_checksum__mutmut_47: survived
validators.finance.x__cusip_checksum__mutmut_48: survived
validators.finance.x__cusip_checksum__mutmut_49: survived
validators.finance.x__cusip_checksum__mutmut_50: survived
validators.finance.x__cusip_checksum__mutmut_51: survived
validators.finance.x__cusip_checksum__mutmut_52: survived
validators.finance.x__cusip_checksum__mutmut_53: survived
validators.finance.x__cusip_checksum__mutmut_54: survived
validators.finance.x__cusip_checksum__mutmut_55: survived
validators.finance.x__cusip_checksum__mutmut_56: survived
validators.finance.x__cusip_checksum__mutmut_57: survived
validators.finance.x__cusip_checksum__mutmut_60: survived
validators.finance.x__cusip_checksum__mutmut_64: survived
validators.finance.x__cusip_checksum__mutmut_65: survived
validators.finance.x__cusip_checksum__mutmut_67: survived
validators.finance.x__isin_checksum__mutmut_10: survived
validators.finance.x__isin_checksum__mutmut_11: survived
validators.finance.x__isin_checksum__mutmut_12: survived
validators.finance.x__isin_checksum__mutmut_15: survived
```

A análise inicial dos resultados do mutmut revelou que 12 mutantes sobreviveram com o status "no tests", indicando que foram gerados em trechos de código não executados por nenhum caso de teste existente. A investigação do código-

fonte apontou que esses mutantes residiam nas funções auxiliares `_file_url` e `_ipfs_url`, localizadas no módulo `uri.py`.

```
def test_returns_true_on_valid_url(value: str):
    """Test returns true on valid url."""
    assert url(value)

@pytest.mark.parametrize(
    "value, private",
    [
        ("http://username:password@10.0.10.1/", True),
        ("http://username:password@192.168.10.10:4010/", True),
        ("http://127.0.0.1", True),
    ],
)
def test_returns_true_on_valid_private_url(value: str, private: Optional[bool]):
    """Test returns true on valid private url."""
    assert url(value, private=private)
```

```
def test_returns_failed_validation_on_invalid_url(value: str):
    """Test returns failed validation on invalid url."""
    assert isinstance(url(value), ValidationError)

@pytest.mark.parametrize(
    "value, private",
    [
        ("http://username:password@192.168.10.10:4010", False),
        ("http://username:password@127.0.0.1:8080", False),
        ("http://10.0.10.1", False),
        ("http://255.255.255.255", False),
    ],
)
def test_returns_failed_validation_on_invalid_private_url(value: str, private: Optional[bool]):
    """Test returns failed validation on invalid private url."""
    assert isinstance(url(value, private=private), ValidationError)
```

Constatou-se que a suíte de testes existente focava no validador `url()`, que não invoca essas funções auxiliares. A arquitetura da biblioteca designa a função `uri()` como um despachante (dispatcher) que, ao identificar os esquemas `file:` ou `ipfs:`, aciona as respectivas funções internas. Portanto, para forçar a execução do código mutado, foram desenvolvidos novos casos de teste parametrizados. Estes novos testes chamam a função pública `uri()` com exemplos de entradas válidas e inválidas para os esquemas `file` e `ipfs`:


```
# ==> URI com esquema 'file' <== #

@pytest.mark.parametrize(
    "value",
    [
        "file:///home/user/file.txt",
        "file:///c:/Users/User/Desktop/document.pdf",
    ],
)
def test_returns_true_on_valid_uri_with_file_scheme(value: str):
    """Testa se a função uri() valida corretamente o esquema 'file'."""
    assert uri(value)

@pytest.mark.parametrize(
    "value",
    [
        "file://home/user/file.txt", # Inválido, precisa de 3 barras
        "file:/home/user/file.txt", # Inválido
    ],
)
def test_returns_failed_validation_on_invalid_uri_with_file_scheme(value: str):
    """Testa se a função uri() invalida esquemas 'file' malformados."""
    assert isinstance(uri(value), ValidationError)
```

```
# ==> URI com esquema 'ipfs' <== #

@pytest.mark.parametrize(
    "value",
    [
        "ipfs://bafybeiemxf5abjwjbikoz4mc3a3dla6ual3jsqdr4cjr3oz3evfyavhwq/wiki/",
        "ipfs://QmXoypizjW3WknFiJnKLwHCnL72vedxjQkDDP1mXwo6uco/wiki/Mars.html",
        "ipfs://",
    ],
)
def test_returns_true_on_valid_uri_with_ipfs_scheme(value: str):
    """Testa se a função uri() valida corretamente o esquema 'ipfs'."""
    assert uri(value)

@pytest.mark.parametrize(
    "value",
    [
        "ipfs:/invalid_path", # Inválido, precisa de 2 barras
    ],
)
def test_returns_failed_validation_on_invalid_uri_with_ipfs_scheme(value: str):
    """Testa se a função uri() invalida esquemas 'ipfs' malformados."""
    assert isinstance(uri(value), ValidationError)
```

Como resultado, a execução do código vulnerável foi garantida, e todos os 12 mutantes que antes sobreviviam por falta de cobertura foram detectados e eliminados ("mortos"), validando a eficácia da melhoria implementada na suíte de testes.

```
● (myenv) gs@gustavo:~/validators$ mutmut run
  :: Generating mutants
    done in 1230ms
  :: Listing all tests
  :: Running clean tests
    done
  :: Running forced fail test
    done
Running mutation testing
  :: 689/689 🚩 474 😬 0 🕒 0 😬 0 😬 215 🚫 0
36.05 mutations/second
○ (myenv) gs@gustavo:~/validators$
```

```
● (myenv) gs@gustavo:~/validators$ mutmut results
validators.finance.x__cusip_checksum__mutmut_17: survived
validators.finance.x__cusip_checksum__mutmut_20: survived
validators.finance.x__cusip_checksum__mutmut_21: survived
validators.finance.x__cusip_checksum__mutmut_31: survived
validators.finance.x__cusip_checksum__mutmut_32: survived
validators.finance.x__cusip_checksum__mutmut_33: survived
validators.finance.x__cusip_checksum__mutmut_34: survived
validators.finance.x__cusip_checksum__mutmut_35: survived
validators.finance.x__cusip_checksum__mutmut_46: survived
validators.finance.x__cusip_checksum__mutmut_47: survived
validators.finance.x__cusip_checksum__mutmut_48: survived
validators.finance.x__cusip_checksum__mutmut_49: survived
validators.finance.x__cusip_checksum__mutmut_50: survived
validators.finance.x__cusip_checksum__mutmut_51: survived
validators.finance.x__cusip_checksum__mutmut_52: survived
validators.finance.x__cusip_checksum__mutmut_53: survived
validators.finance.x__cusip_checksum__mutmut_54: survived
validators.finance.x__cusip_checksum__mutmut_55: survived
validators.finance.x__cusip_checksum__mutmut_56: survived
validators.finance.x__cusip_checksum__mutmut_57: survived
validators.finance.x__cusip_checksum__mutmut_60: survived
validators.finance.x__cusip_checksum__mutmut_64: survived
validators.finance.x__cusip_checksum__mutmut_65: survived
validators.finance.x__cusip_checksum__mutmut_67: survived
```

A execução inicial da ferramenta mutmut identificou algumas vulnerabilidade na suíte de testes, com uma concentração de 24 mutantes sobreviventes na função auxiliar `_cusip_checksum`, localizada no módulo `finance.py`. Tal concentração indicou que, apesar de o código possuir cobertura de testes, a sua complexidade algorítmica não estava sendo adequadamente verificada.

A partir dessa análise, constatou-se que o conjunto de CUSIPs válidos nos testes estava restrito a combinações com dígitos, letras maiúsculas e caracteres especiais. Em termos práticos, isso significava que o critério de cobertura estrutural não estava assegurado: uma parte relevante do domínio de entradas possíveis para a função não estava devidamente representada nos testes automatizados.

O CUSIP (*Committee on Uniform Securities Identification Procedures*) é um identificador padronizado utilizado no mercado financeiro norte-americano para individualizar títulos mobiliários, como ações, títulos públicos e títulos corporativos. Um código CUSIP é composto por nove caracteres alfanuméricos. Os seis primeiros representam o emissor do título, os dois seguintes identificam o tipo específico de título emitido, e o último caractere é um dígito verificador (checksum), que confere integridade ao código. Esse dígito de verificação é calculado a partir de um algoritmo específico que converte letras em números, aplica multiplicadores alternados e soma os valores resultantes. O cálculo é finalizado pela aplicação de uma operação de módulo 10, cujo resultado determina o valor do checksum.

```
# ==> CUSIP <== #

@pytest.mark.parametrize("value", ["912796X38", "912796X20", "912796x20"])
def test_returns_true_on_valid_cusip(value: str):
    """Test returns true on valid cusip."""
    assert cusip(value)

@pytest.mark.parametrize("value", ["912796T67", "912796T68", "XCVF", "00^^^1234"])
def test_returns_failed_validation_on_invalid_cusip(value: str):
    """Test returns failed validation on invalid cusip."""
    assert isinstance(cusip(value), ValidationError)
```

A fase de implementação consistiu na expansão da suíte de testes com casos parametrizados (@pytest.mark.parametrize), direcionados às lacunas identificadas pelos mutantes sobreviventes. Foram elaborados CUSIPs válidos para valores de fronteira e caracteres especiais, com o auxílio de LLMs, como ChatGPT e Gemini, utilizados como suporte para a geração e verificação dos exemplos.

```

@pytest.mark.parametrize(
    "value",
    [
        # Casos de teste originais
        "912796X38",
        "912796X20",
        "912796x20",

        # --- NOVOS TESTES DIRECIONADOS (COM CHECKSUMS CORRIGIDOS) ---

        # 1. Testando as fronteiras de letras e números
        "A12345673", # Contém 'A' (fronteira inferior de letras)
        "Z12345676", # Contém 'Z' (fronteira superior de letras)
        "123456782", # Usa o dígito '8'
        "123456790", # Usa o dígito '9' (checksum é 0)

        # 2. Testando os caracteres especiais
        "*12345675", # Contém '*' (asterisco)
        "@12345674", # Contém '@' (arroba)
        "#12345673", # Contém '#' (cerquilha)

        # 3. Teste para estressar a lógica de soma (um exemplo da Wikipedia)
        "037833100",

        "a12345673", # Contém 'a' (fronteira inferior de minúsculas)
        "z12345676", # Contém 'z' (fronteira superior de minúsculas)
    ],
)
def test_returns_true_on_valid_cusip(value: str):
    """Test returns true on valid cusip."""
    assert cusip(value)

```

```

@pytest.mark.parametrize(
    "value",
    [
        # Casos inválidos originais
        "912796T67",
        "912796T68",
        "XCVF",
        "00^^^1234",

        # Comprimento inválido
        "12345678", # 8 caracteres
        "12345678901", # 11 caracteres

        # Checksum incorreto
        "037833101", # válido seria ...100
        "912796X39", # válido seria ...38

        # Caracteres ilegais
        "12345$678", # símbolo no meio
        "abcdefghi", # letras minúsculas
    ],
)
def test_returns_failed_validation_on_invalid_cusip(value: str):
    """Test returns failed validation on invalid cusip."""
    assert isinstance(cusip(value), ValidationError)

```

Com essas alterações, foi feita uma nova rodada com o mutmut run, que mostrou um aumento de 22 mutantes eliminados com relação à última rodada:

```

• (myenv) gs@gustavo:~/validators$ mutmut run
  :: Generating mutants
    done in 1052ms
  :: Listing all tests
  :: Running clean tests
    done
  :: Running forced fail test
    done
Running mutation testing
  :: 689/689 🎯 496 🤖 0 🧑 0 🧐 0 🧐 193 🚫 0
24.89 mutations/second

```

Em outras palavras, o conjunto de testes tornou-se mais robusto e alinhado ao comportamento esperado da função, evidenciando a utilidade prática dos testes de mutação como técnica de avaliação e melhoria da qualidade dos testes. Contudo, a execução subsequente do mutmut results indicou que um subconjunto de mutantes persistiu. Uma análise detalhada dos sobreviventes, exemplificados por mutantes como _33 e _34, revelou que estes se concentravam na lógica de tratamento de letras minúsculas na função _cusip_checksum.

```

• (myenv) gs@gustavo:~/validators$ mutmut show validators.finance.x_cusip_checksum__mutmut_33
# validators.finance.x_cusip_checksum__mutmut_33: survived
--- src/validators/finance.py
+++ src/validators/finance.py
@@ -7,7 +7,7 @@
     val = ord(c) - ord("0")
     elif c >= "A" and c <= "Z":
         val = 10 + ord(c) - ord("A")
-     elif c >= "a" and c <= "z":
+     elif c >= "XXaXX" and c <= "z":
         val = 10 + ord(c) - ord("a")
     elif c == "*":
         val = 36

• (myenv) gs@gustavo:~/validators$ mutmut show validators.finance.x_cusip_checksum__mutmut_34
# validators.finance.x_cusip_checksum__mutmut_34: survived
--- src/validators/finance.py
+++ src/validators/finance.py
@@ -7,7 +7,7 @@
     val = ord(c) - ord("0")
     elif c >= "A" and c <= "Z":
         val = 10 + ord(c) - ord("A")
-     elif c >= "a" and c <= "z":
+     elif c >= "A" and c <= "z":
         val = 10 + ord(c) - ord("a")
     elif c == "*":
         val = 36

```

No primeiro caso, o intervalo de comparação da letra “a” foi substituído por um valor inválido (“XXaXX”), o que impossibilitaria o reconhecimento de caracteres

minúsculos. No segundo, a mutação ampliou a faixa de verificação, permitindo que caracteres entre “A” e “z” fossem interpretados como válidos, criando uma sobreposição entre maiúsculas e minúsculas.

O fato de esses mutantes terem sobrevivido indica que a suíte de testes, ainda que tenha sido expandida com novos casos direcionados, não foi suficiente para explorar todos os cenários necessários. A inserção de valores de fronteira como “a” e “z” foi uma estratégia importante, pois buscou contemplar os extremos do intervalo de caracteres aceitos. No entanto, a ausência de testes intermediários, como aqueles que envolvem letras centrais da faixa alfabética (“b”, “m” ou “y”), deixou lacunas que permitiram a permanência dos mutantes sem detecção. Assim, observa-se que a cobertura estrutural, embora aumentada, não atingiu a robustez necessária para eliminar as mutações propostas pela ferramenta. Esse resultado evidencia uma limitação comum em suítes de testes que se apoiam predominantemente em valores de fronteira. Embora tais valores sejam essenciais, eles não garantem sozinhos a identificação de todas as falhas potenciais.

3. Conclusão

O presente trabalho demonstrou a aplicação prática e a eficácia do teste de mutação como uma ferramenta de avaliação da qualidade de suítes de teste em software. O objetivo principal foi utilizar a ferramenta Mutmut para analisar os casos de teste do projeto Python Validators, identificar fragilidades e propor melhorias direcionadas para aumentar a capacidade de detecção de falhas. A metodologia envolveu uma análise sequencial, partindo da execução dos testes unitários com Pytest, passando pela medição da cobertura de código com Pytest-cov e culminando na análise aprofundada dos mutantes sobreviventes.

A investigação revelou duas categorias principais de deficiência na suíte de testes original. A primeira foi a existência de 12 mutantes com o status "no tests", localizados nas funções auxiliares `_file_url` e `_ipfs_url`. Constatou-se que, embora a cobertura de código geral fosse alta (89%), estes trechos específicos de código nunca eram executados. A solução envolveu a criação de novos testes que chamavam a função pública `uri()` com os esquemas `file:` e `ipfs:`, garantindo a execução do código vulnerável e eliminando com sucesso esses 12 mutantes.

A segunda e mais crítica deficiência foi a identificação de pontos críticos, como a função `_cusip_checksum`, que concentrou mutantes sobreviventes devido a lacunas nos casos de teste originais. A expansão da suíte de testes, com o uso de entradas direcionadas a fronteiras e caracteres especiais, resultou em uma melhora significativa, eliminando parte dos mutantes sobreviventes e evidenciando a eficácia da abordagem incremental. Contudo, mutantes mais complexos, ligados ao tratamento de letras minúsculas e condições intermediárias, permaneceram sem detecção, sugerindo a necessidade de um refinamento ainda maior no design dos testes.

Quantitativamente, o impacto das melhorias foi significativo. A execução inicial do Mutmut resultou em 215 mutantes sobreviventes e uma pontuação de mutação de aproximadamente 69%. Após a implementação dos novos casos de teste, o número de mutantes sobreviventes foi reduzido para 193, elevando o total de mutantes mortos de 474 para 496 e, conseqüentemente, aumentando a pontuação de mutação para 72%. Este aumento de 22 mutantes eliminados demonstra que melhorias direcionadas, guiadas pela análise de mutação, são mais eficazes do que a simples adição de testes sem um critério definido.

Tal análise evidenciou a importância dos testes de mutação como um complemento à cobertura de código tradicional. Enquanto ferramentas como o pytest-cov mostram apenas quais linhas foram executadas, os testes de mutação permitem avaliar a efetividade dos testes em detectar comportamentos incorretos, fornecendo uma visão mais profunda sobre possíveis falhas não capturadas. Essa observação possibilita identificar pontos de melhoria nos testes, garantindo maior confiabilidade e robustez do software.

4. Links

[Link do Repositório no GitHub](#)

[Link do Vídeo](#)

5. Referências

PYTHON-VALIDATORS. *Python data validation for humans™*. GitHub, 2025. Disponível em: <https://github.com/python-validators/validators>. Acesso em: 28 ago. 2025.

WIKIPEDIA. *CUSIP*. In: *Wikipedia: the free encyclopedia*. [S. l.], 19 ago. 2025. Disponível em: <https://en.wikipedia.org/w/index.php?title=CUSIP&oldid=1306711795>. Acesso em: 28 ago. 2025.

ANDRADE, Stevao. *Introdução ao teste de mutação em Python com a ferramenta mutmut*. YouTube, 2025. Disponível em: <https://www.youtube.com/watch?v=FbMpoVOorFI>. Acesso em: 28 ago. 2025.

GMARQUES79.

Teste_Software_2025_Gustavo_Marques_Henrique_Bastos_Mariana_Nunes_Rafael_Oliveira. GitHub, 2025. Disponível em:

https://github.com/gmarques79/Teste_Software_2025_Gustavo_Marques_Henrique_Bastos_Mariana_Nunes_Rafael_Oliveira. Acesso em: 28 ago. 2025.