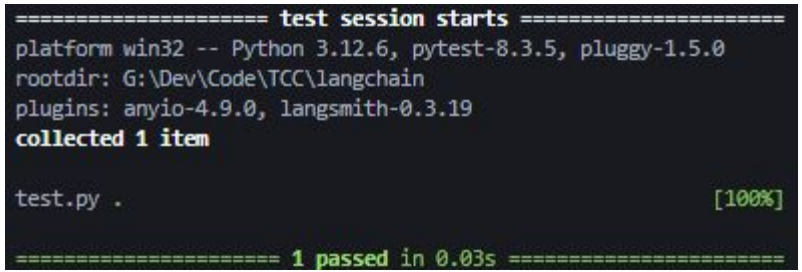
pytest

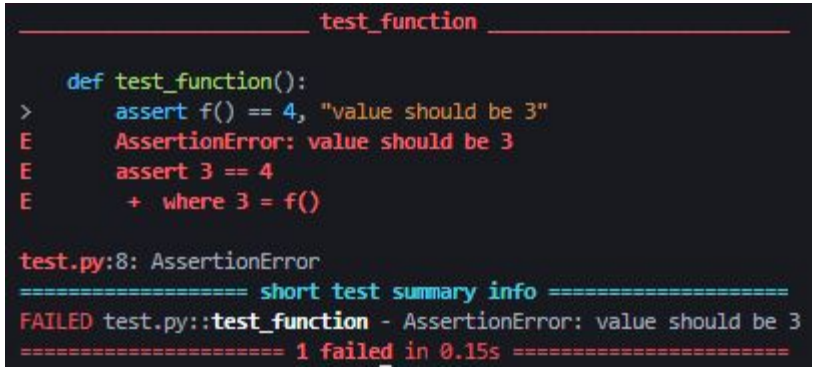
assert

```python
# test.py
def f():
    return 3


def test_function():
    assert f() == 3, "value should be 3"
```



assert

```python
# test.py
def f():
    return 3


def test_function():
    assert f() == 4, "value should be 3"
```



Exceptions

```python
import pytest

def divide(a, b):
      return a / b

def test_divide_by_zero():
      with pytest.raises(ZeroDivisionError):
      divide(10, 0)
```

Se **`divide(10, 0)`** der **`ZeroDivisionError`**, o teste passa

Exceptions

```
import pytest

def square_root(number):
      if number < 0:
      raise ValueError("Cannot calculate square root of a negative number")
      return number ** 0.5

def test_square_root_negative_number():
      with pytest.raises(ValueError) as exc_info:
      square_root(-9)

      assert "Cannot calculate square root" in str(exc_info.value)
```

Não é possível calcular raiz quadrada de número negativo - o teste passa

@pytest.fixture

```python
import pytest

class Student:
      def __init__(self, name, year):
      self.name = name
      self.year = year

      def is_final_year(self):
      return self.year == 4

@pytest.fixture
def student():
      return Student(name="Alice", year=4)


def test_student_properties(student):
      assert student.name == "Alice"
      assert student.year == 4
      assert student.is_final_year() is True
```

- fixture: objetos, dados ou configurações que o teste precisa para rodar

- no pytest, usa-se o decorator `@pytest.fixture` antes do método. Fixtures dão o contexto para o teste funcionar.

- fixtures podem ser usadas várias vezes em testes diferentes

- para usar uma fixture em um teste, ele deve ser passado como argumento da função de teste.

@pytest.fixture

```python
import pytest

class Fruit:
    def __init__(self, name):
        self.name = name

    def __eq__(self, other):
        return self.name == other.name

@pytest.fixture
def my_fruit():
    return Fruit("apple")

@pytest.fixture
def fruit_basket(my_fruit):
    return [Fruit("banana"), my_fruit]


def test_my_fruit_in_basket(my_fruit, fruit_basket):
    assert my_fruit in fruit_basket
```

```
[Fruit("banana"), Fruit("apple")]
```

@pytest.mark.parametrize

```python
def test_is_palindrome_empty_string():
    assert is_palindrome("")

def test_is_palindrome_single_character():
    assert is_palindrome("a")

def test_is_palindrome_mixed_casing():
    assert is_palindrome("Bob")

def test_is_palindrome_with_spaces():
    assert is_palindrome("Never odd or even")

def test_is_palindrome_with_punctuation():
    assert is_palindrome("Do geese see God?")

def test_is_palindrome_not_palindrome():
    assert not is_palindrome("abc")

def test_is_palindrome_not_quite():
    assert not is_palindrome("abab")
```

```python
@pytest.mark.parametrize("maybe_palindrome, expected_result", [
    ("", True),
    ("a", True),
    ("Bob", True),
    ("Never odd or even", True),
    ("Do geese see God?", True),
    ("abc", False),
    ("abab", False),
])
def test_is_palindrome(maybe_palindrome, expected_result):
    assert is_palindrome(maybe_palindrome) == expected_result
```

Em vez de fazer várias funções de testes no mesmo formato, usamos `@pytest.mark.parametrize` para condensar os testes em uma função.

TDD

- Objetivo: Escrever uma classe que valida senhas.

```
def test_valid_password():
      validator = PasswordValidator()
      assert validator.is_valid("password123") is True
```

Classe `PasswordValidator()` ainda não existe!

TDD

```
class PasswordValidator:
      def is_valid(self, password):
          return True
```

Classe minimamente implementada

```
def test_valid_password():
      validator = PasswordValidator()
      assert validator.is_valid("password123") is True
```

Teste passa!

TDD

Adicionamos mais um teste

```
def test_reject_short_password():
      validator = PasswordValidator()
      assert validator.is_valid("abc") is False
```

Teste não funciona mais com classe implementada antes!

TDD

Adicionamos mais um teste

```
def test_reject_short_password():
      validator = PasswordValidator()
      assert validator.is_valid("abc") is False
```

Teste não funciona mais com classe implementada antes!

Atualizamos a implementação minimamente para que o teste passe

```
class PasswordValidator:
    def is_valid(self, password):
        return len(password) >= 8
```

TDD

```
def test_valid_password():
      validator = PasswordValidator()
      assert validator.is_valid("password123") is True
```

```
def test_reject_short_password():
      validator = PasswordValidator()
      assert validator.is_valid("abc") is False
```

```
class PasswordValidator:
    def is_valid(self, password):
        return len(password) >= 8
```