**Exercise 1: Validate Email Address**

**🧪 Scenario**

Check whether an input string is a valid email format using regex.

**utils.py**

import re

def is\_valid\_email(email: str) -> bool:

pattern = r"^[\w\.-]+@[\w\.-]+\.\w{2,}$"

return re.match(pattern, email) is not None

**🧪 test\_utils.py**

from utils import is\_valid\_email

def test\_valid\_email():

assert is\_valid\_email("test@example.com")

def test\_invalid\_email\_no\_at():

assert not is\_valid\_email("test.example.com")

def test\_invalid\_email\_no\_domain():

assert not is\_valid\_email("test@")

def test\_valid\_email\_with\_dash():

assert is\_valid\_email("user-name\_123@mail.co.uk")

**Expectations**

* Matches correct email formats
* Rejects missing @ or domain parts

**✅ Exercise 2: Password Strength Checker**

**🧪 Scenario**

Check that password meets all conditions:

* Min 8 characters
* At least 1 digit
* At least 1 uppercase
* At least 1 special character

**utils.py**

import re

def is\_strong\_password(password: str) -> bool:

if len(password) < 8:

return False

if not re.search(r"\d", password):

return False

if not re.search(r"[A-Z]", password):

return False

if not re.search(r"[!@#$%^&\*()]", password):

return False

return True

**🧪 test\_utils.py**

from utils import is\_strong\_password

def test\_strong\_password():

assert is\_strong\_password("MyPass123!")

def test\_weak\_password\_short():

assert not is\_strong\_password("M1!")

def test\_weak\_password\_no\_digit():

assert not is\_strong\_password("MyPassword!")

def test\_weak\_password\_no\_upper():

assert not is\_strong\_password("mypassword1!")

def test\_weak\_password\_no\_special():

assert not is\_strong\_password("MyPassword1")

**✅ Exercise 3: FizzBuzz Logic**

**🧪 Scenario**

Return:

* "Fizz" if divisible by 3
* "Buzz" if divisible by 5
* "FizzBuzz" if divisible by both
* The number itself otherwise

**utils.py**

def fizzbuzz(n: int) -> str:

if n % 15 == 0:

return "FizzBuzz"

elif n % 3 == 0:

return "Fizz"

elif n % 5 == 0:

return "Buzz"

return str(n)

**🧪 test\_utils.py**

from utils import fizzbuzz

def test\_fizzbuzz\_fizzbuzz():

assert fizzbuzz(15) == "FizzBuzz"

def test\_fizzbuzz\_fizz():

assert fizzbuzz(9) == "Fizz"

def test\_fizzbuzz\_buzz():

assert fizzbuzz(10) == "Buzz"

def test\_fizzbuzz\_number():

assert fizzbuzz(7) == "7"

**✅ Exercise 4: Get User by ID (Mocked API)**

**🧪 Scenario**

You have a function that fetches user data via an API. Write unit tests by mocking the requests.

**utils.py**

import requests

def get\_user(user\_id):

url = f"https://jsonplaceholder.typicode.com/users/{user\_id}"

response = requests.get(url)

if response.status\_code != 200:

return None

return response.json()

**🧪 test\_utils.py with pytest-mock**

import requests

from utils import get\_user

def test\_get\_user\_success(mocker):

mock\_response = mocker.Mock()

mock\_response.status\_code = 200

mock\_response.json.return\_value = {"id": 1, "name": "John"}

mocker.patch("requests.get", return\_value=mock\_response)

user = get\_user(1)

assert user["name"] == "John"

def test\_get\_user\_failure(mocker):

mock\_response = mocker.Mock()

mock\_response.status\_code = 404

mocker.patch("requests.get", return\_value=mock\_response)

user = get\_user(999)

assert user is None

**✅ Exercise 5: Shopping Cart Total Calculation**

**🧪 Scenario**

Calculate the total price of items in a cart with quantities and discounts.

**utils.py**

def calculate\_cart\_total(cart):

total = 0

for item in cart:

price = item["price"]

qty = item["qty"]

discount = item.get("discount", 0)

total += (price \* qty) \* (1 - discount)

return round(total, 2)

**🧪 test\_utils.py**

from utils import calculate\_cart\_total

def test\_cart\_total\_basic():

cart = [{"price": 100, "qty": 2}, {"price": 50, "qty": 1}]

assert calculate\_cart\_total(cart) == 250.0

def test\_cart\_with\_discounts():

cart = [

{"price": 100, "qty": 2, "discount": 0.1}, # 10% off

{"price": 50, "qty": 1}

]

assert calculate\_cart\_total(cart) == 230.0

**✅ Bonus: How to Run Tests with Coverage**

pip install pytest pytest-cov

pytest --cov=utils test\_utils.py

This shows you **how much of your code is covered by tests**.

**Exercise 1: Temperature Converter using @pytest.mark.parametrize**

**🔹 Scenario:**

Test a function that converts Celsius to Fahrenheit using the formula:  
**F = C × 9/5 + 32**

**Code: converter.py**

def celsius\_to\_fahrenheit(celsius):

return round((celsius \* 9/5) + 32, 2)

**🧪 Test: test\_converter.py**

import pytest

from converter import celsius\_to\_fahrenheit

@pytest.mark.parametrize(

"celsius, expected\_fahrenheit",

[

(0, 32.00),

(100, 212.00),

(-40, -40.00),

(37, 98.6),

(25, 77.00)

]

)

def test\_celsius\_to\_fahrenheit(celsius, expected\_fahrenheit):

assert celsius\_to\_fahrenheit(celsius) == expected\_fahrenheit

**🧠 Explanation:**

* Tests the same function with **5 different inputs**.
* Confirms correct floating-point results.
* Rounds output to avoid float precision issues.

**🧪 Exercise 2: Shopping Cart Total with @pytest.fixture + @pytest.mark.parametrize**

**🔹 Scenario:**

Given a cart of items, calculate total price. Some items may have discounts.

**✅ Code: cart.py**

def calculate\_total(cart):

total = 0

for item in cart:

price = item["price"]

qty = item["qty"]

discount = item.get("discount", 0)

total += (price \* qty) \* (1 - discount)

return round(total, 2)

**🧪 Test: test\_cart.py**

import pytest

from cart import calculate\_total

@pytest.fixture

def sample\_cart():

return [

{"price": 100, "qty": 2},

{"price": 50, "qty": 1}

]

@pytest.mark.parametrize(

"cart, expected\_total",

[

(

[{"price": 100, "qty": 1}],

100.0

),

(

[{"price": 100, "qty": 2}, {"price": 50, "qty": 1}],

250.0

),

(

[{"price": 100, "qty": 2, "discount": 0.1}],

180.0

),

(

[{"price": 100, "qty": 1, "discount": 0.5}],

50.0

)

]

)

def test\_calculate\_total(cart, expected\_total):

assert calculate\_total(cart) == expected\_total

def test\_sample\_cart\_total(sample\_cart):

assert calculate\_total(sample\_cart) == 250.0

**🧠 Explanation:**

* @fixture provides a reusable sample cart.
* @parametrize tests multiple variations: no discount, with discount, quantity.
* Ensures rounding is handled correctly.

**🧪 Exercise 3: String Formatter using @pytest.mark.parametrize**

**🔹 Scenario:**

Format full name with proper capitalization and whitespace trimming.

**✅ Code: formatter.py**

def format\_name(first, last):

return f"{first.strip().title()} {last.strip().title()}"

**🧪 Test: test\_formatter.py**

import pytest

from formatter import format\_name

@pytest.mark.parametrize(

"first,last,expected",

[

("john", "doe", "John Doe"),

(" Jane", "SMITH ", "Jane Smith"),

("aLiCE", "joHnSon", "Alice Johnson"),

(" bOb", "MARLEY ", "Bob Marley")

]

)

def test\_format\_name(first, last, expected):

assert format\_name(first, last) == expected

**🧠 Explanation:**

* Tests name formatting for casing and whitespace.
* Useful in apps dealing with user input cleanup.

**🧪 Exercise 4: Bank Withdrawal using @pytest.fixture and @pytest.mark.parametrize**

**🔹 Scenario:**

Simulate withdrawal from an account, with checks for insufficient balance.

**✅ Code: bank.py**

class BankAccount:

def \_\_init\_\_(self, balance=0):

self.balance = balance

def withdraw(self, amount):

if amount <= self.balance:

self.balance -= amount

return True

return False

**🧪 Test: test\_bank.py**

import pytest

from bank import BankAccount

@pytest.fixture

def account():

return BankAccount(100)

@pytest.mark.parametrize(

"withdraw\_amount, expected\_result, expected\_balance",

[

(50, True, 50),

(100, True, 0),

(150, False, 100),

(0, True, 100)

]

)

def test\_withdraw(account, withdraw\_amount, expected\_result, expected\_balance):

result = account.withdraw(withdraw\_amount)

assert result == expected\_result

assert account.balance == expected\_balance

**🧠 Explanation:**

* Tests successful and failed withdrawal.
* Fixture sets account with balance = 100.
* Parameterization avoids code repetition.