TP2

Installation de Pytest

Installer pytest:

pip install pytest pytest-cov ou poetry add pytest pytest-cov

Créer un dossier pour mettre les fichiers de code et tests

Pour ceux qui travaillent avec poetry ou un autre
[tool.pytest.ini_options]
    pythonpath = "src"

PYTHONATH=. pytest

Test unitaires

Exercice 1

Écrire quatre fonctions pour calculer les quatres opérations (addition, soustraction, multiplication, division) sur deux entrées a et b, créer des testes unitaire pour chaque fonction, mesurer la couverture de ces tests

Exercise 2

Écrire une fonction qui **supprime les doublons consécutifs** dans une liste, tout en conservant l'ordre initial. Et créer des tests pour la tester et mesurer la couverture.

Exemples :

- [1, 1, 2, 2, 3, 1] → [1, 2, 3, 1]
- ['a', 'a', 'b', 'a'] → ['a', 'b', 'a']

Tests d’intégration

Exercice 3

créer des tests d'intégration pour la tester et mesurer la couverture.

```
def load_numbers(path):
    with open(path, "r", encoding="utf-8") as f:
        return [int(line.strip()) for line in f if line.strip()]

def sum_numbers(nums):
    return sum(nums)
```

Exercise 4

```
import csv
def load_users(path):
    with open(path, newline="", encoding="utf-8") as f:
        reader = csv.DictReader(f)
        return [{"name": r["name"], "age": int(r["age"])} for r in reader]

def filter_adults(users):
    return [u for u in users if u["age"] >= 18]
```

Tests fonctionnelles:

Installer flask:

pip install flask OU poetry add flask

Exercise 5

Creer un fichier web_app.py:

```
from flask import Flask, jsonify, request
from uuid import uuid4

def create_app(config: dict | None = None) -> Flask:
    app = Flask(__name__)
```

```
app.config.update(TESTING=False)
if config:
    app.config.update(config)

# Stockage en mémoire réinitialisé à chaque create_app
app.state = {"users": {}}  # id -> {"id","name","age"}

@app.get("/health")
def health():
    return jsonify({"status": "ok"}), 200

@app.post("/users")
def create_user():
    if not request.is_json:
        return jsonify({"error": "JSON required"}), 400
    data = request.get_json()

    # validations simples
    name = data.get("name")
    age = data.get("age")
    if not isinstance(name, str) or not name.strip():
        return jsonify({"error": "name is required"}), 400
    if not isinstance(age, int) or age < 0:
        return jsonify({"error": "age must be a non-negative integer"}), 400

    uid = str(uuid4())
    user = {"id": uid, "name": name.strip(), "age": age}
    app.state["users"][uid] = user
    return jsonify(user), 201

@app.get("/users")
def list_users():
    return jsonify(list(app.state["users"].values())), 200

@app.get("/users/<uid>")
def get_user(uid: str):
    user = app.state["users"].get(uid)
    if not user:
        return jsonify({"error": "not found"}), 404
    return jsonify(user), 200
```

```
    @app.put("/users/<uid>")
    def update_user(uid: str):
        user = app.state["users"].get(uid)
        if not user:
            return jsonify({"error": "not found"}), 404
        if not request.is_json:
            return jsonify({"error": "JSON required"}), 400
        data = request.get_json()

        name = data.get("name", user["name"])
        age = data.get("age", user["age"])
        if not isinstance(name, str) or not name.strip():
            return jsonify({"error": "invalid name"}), 400
        if not isinstance(age, int) or age < 0:
            return jsonify({"error": "invalid age"}), 400
        user = {"id": uid, "name": name.strip(), "age": age}
        app.state["users"][uid] = user
        return jsonify(user), 200

    @app.delete("/users/<uid>")
    def delete_user(uid: str):
        if uid not in app.state["users"]:
            return jsonify({"error": "not found"}), 404
        del app.state["users"][uid]
        return "", 204

    return app
```

Créer des tests fonctionnelles pour chaque endpoint