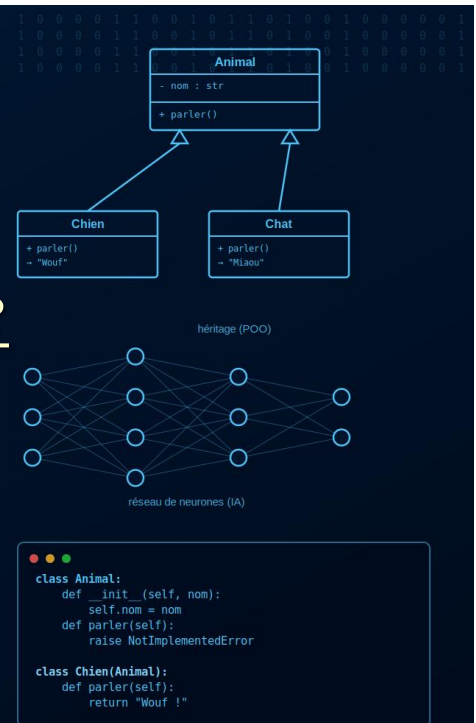


IV. Tests et qualité logicielle



M. Martin

1

1

Pourquoi tester?

- Les tests détectent les régressions, documentent le comportement attendu et sécurisent les réusines
- Ils servent de **références expérimentales** : comme en physique, un test « valide qu'un système se comporte toujours de la même façon dans les mêmes conditions »
- Tout ingénieur doit automatiser ses vérifications pour garantir la reproductibilité

Outils recommandés

- **pytest** : exécuter de tests simples et puissants
- **numpy.testing** : assertions avec tolérances numériques
- **pytest-cov** : mesure de couverture du code
- (optionnel) **pytest-mp1** : comparaison automatique de figures Matplotlib

2

2

Organisation d'un projet avec tests

```
mon_projet/  
-- src/  
| -- calculs.py  
-- data/  
-- results/  
-- env/ # environnement virtuel (non versionné)  
-- tests/  
| -- test_calculs.py  
-- requirements.txt  
-- requirements-dev.txt # dépendances de développement (tests)  
-- pyproject.toml / pytest.ini (optionnel)  
-- README.md
```

Règles

- Un fichier « test_*.py » par module testé
- Un test par cas d'usage (clair, court, isolé)
- Exécution recommandée :

```
pytest tests/ -v
```

3

3

Démarrer avec pytest

src/calculs.py

```
def aire_rectangle(L: float, l: float = 1.0) -> float:  
    """Calcule l'aire d'un rectangle."""  
    if L < 0 or l < 0:  
        raise ValueError("Dimensions négatives interdites")  
    return L * l
```

src/calculs.py

```
from src.calculs import aire_rectangle  
import pytest  
  
def test_aire_rectangle_cas_simple():  
    assert aire_rectangle(3, 2) == 6  
  
def test_aire_rectangle_defaut():  
    assert aire_rectangle(5) == 5.0  
  
def test_aire_rectangleErreur():  
    with pytest.raises(ValueError):  
        aire_rectangle(-1, 2)
```

4

4

Tests numériques et tolérances (flottants)

Ne comparez pas des float avec == : utilisez des tolérances.

```
from numpy.testing import assert_allclose

def energie_cinetique(m, v):
    return 0.5 * m * v ** 2

def test_energie_cinetique_tol():
    theorique = 12.5
    calcule = energie_cinetique(5, 2.2360679) # ~ √5
    assert_allclose(calcule, theorique, rtol=1e-6, atol=1e-12)
```

- Choisir selon les ordres de grandeur et la chaîne de calcul :
 - rtol : tolérance relative
 - atol : tolérance absolue
- assert_allclose() gère vecteurs et matrices, contrairement à math.isclose()

5

5

Fixtures : données communes, propres et réutilisables

Une **fixture** est une fonction qui prépare un contexte de test (par exemple un signal simulé, une connexion ouverte, un fichier temporaire). Elle permet d'écrire des tests clairs et sans duplication.

```
import numpy as np
import pytest

@pytest.fixture
def signal_amorti():
    t = np.linspace(0, 10, 501)
    A, alpha, omega = 5.0, 0.5, 2*np.pi
    f = A * np.exp(-alpha*t) * np.cos(omega * t)
    return t, f

def test_signal_non_nul(signal_amorti):
    t, f = signal_amorti
    assert f.size == t.size
    assert np.isfinite(f).all()
```

Autres « fixtures » : tmp_path crée un dossier temporaire pour écrire/lire des fichiers de test.
monkeypatch : modifie temporairement une fonction ou une variable d'environnement.

6

6

Paramétriser des cas

```
import pytest
from src.calculs import aire_rectangle
@pytest.mark.parametrize(
    "L, l, attendu",
    [(3, 2, 6.0), (0, 10, 0.0), (2.5, 4.0, 10.0)]
)
def test_aire_param(L, l, attendu):
    assert aire_rectangle(L, l) == attendu
```

Avantage : gain de temps, car un seul test couvre plusieurs cas

7

7

Tests avec aléatoire : reproductibilité

- Fixer la graine (*seed*) pour rendre le test déterministe
- Comme pour une expérience scientifique : « mêmes conditions $\Rightarrow$ mêmes résultats »

```
import numpy as np

def test_bruit_reproductible():
    rng = np.random.default_rng(42)
    a = rng.normal(0, 1, size=1000)
    rng = np.random.default_rng(42)
    b = rng.normal(0, 1, size=1000)
    assert np.array_equal(a, b)
```

8

8

Tester des figures Matplotlib

- Évitez les tests « au pixel près » (fragiles)
- Testez plutôt la structure et les métadonnées

```
import matplotlib.pyplot as plt
import numpy as np

def tracer(t, y, titre = "fig"):
    fig, ax = plt.subplots()
    ax.plot(t, y, label = "signal")
    ax.set_title(titre)
    ax.set_xlabel("t(s)")
    ax.set_ylabel("y (u.a.)")
    ax.legend()
    return fig, ax
```

9

9

Tester des figures Matplotlib (cont.)

```
import matplotlib.pyplot as plt
import numpy as np

def test_tracer_structure():
    t = np.linspace(0, 1, 11) ; y = np.sin(2*np.pi*t)
    fig, ax = tracer(t, y, "test") # 1 courbe
    lignes = ax.get_lines()
    assert len(lignes) == 1
    # métadonnées
    assert ax.get_title() == "test"
    assert ax.get_xlabel().startswith("t")
    assert ax.get_ylabel().startswith("y")
    plt.close(fig)
```

Astuce

Pour des comparaisons visuelles lourdes, utiliser « pytest-mp1 »

10

10

- Un **mock** est un objet simulé qui imite le comportement d'un véritable objet, mais dont on peut contrôler les retours et vérifier les appels (ex. on simule les E/S coûteuses)

```
def lire_temperature(capteur):
    # imagine une E/S matérielle lente:
    return capteur.lire()

class FakeCapteur:
    def lire(self):
        return 42.0

def test_lire_temperature_sans_io():
    c = FakeCapteur()
    assert lire_temperature(c) == 42.0

def test_lire_temperature_mock():
    with patch.object(FakeCapteur, "lire", return_value=21.5):
        assert lire_temperature(FakeCapteur()) == 21.5
```

Remarque : patch() peut aussi simuler des appels à une API, une base de données, etc

11

11

Le module hypothesis permet de générer des entrées aléatoires contraintes afin de tester des invariants

```
pip install hypothesis
```

```
from hypothesis import given, strategies as st
from numpy.testing import assert_allclose

@given(st.floats(min_value=0, max_value=1e6), st.floats(min_value=0, max_value=1e6))
def test_aire_rectangle_invariants(L, l):
    from src.calculs import aire_rectangle
    a = aire_rectangle(L, l)
    assert a >= 0
    # homogénéité : aire(kL, kl) = k^2 * aire(L, l)
    k = 3.0
    assert_allclose(aire_rectangle(k*L, k*l), k*k*a, rtol=1e-6, atol=1e-12)
```

Remarque : Utile pour valider des lois physiques (conservation, homogénéité, symétrie)

12

12

Objectifs

- Plus de 80 % des composants du coeur de l'application (calculs, traitements) doivent être couverts
- Visualiser les lignes non testées pour cibler les manques

Mesure rapide

```
pytest --cov=src --cov-report=term-missing
```

13

13

- L'intégration continue (CI) est une pratique professionnelle standard
- Chaque *commit* doit être validé automatiquement

.github/workflows/tests.yml

```
name: tests
on: [push, pull_request]
jobs:
  pytest:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: '3.12'
      - run: python -m pip install --upgrade pip
      - run: pip install -r requirements.txt
      - run: pip install pytest pytest-cov
      - run: pytest -q --cov=src --cov-report=term-missing
```

14

14

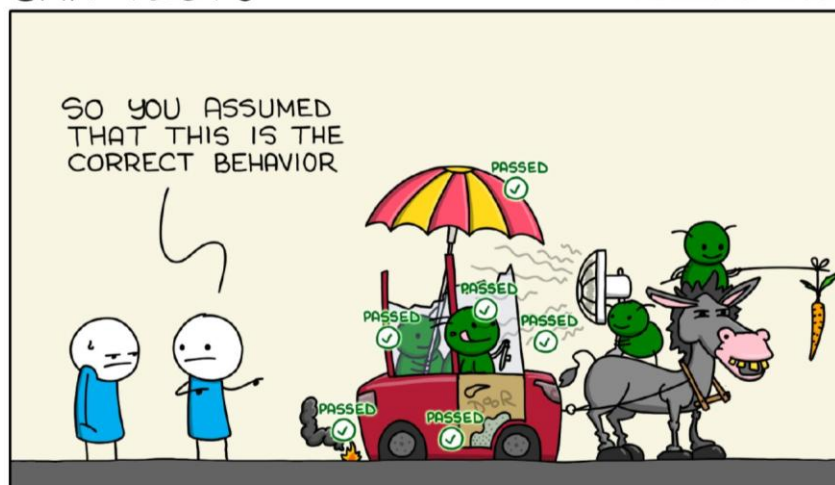
- Un test doit être **court**, **lisible**, **isolé**, et ne **vérifier qu'un seul comportement**
- Nommez **clairement** les fonctions de test : `test_fonction_condition_resultat()`
- Utilisez les **tolérances** pour les flottants, une **graine fixe** pour l'aléatoire
- Fermez les figures (`plt.close(fig)`) pour éviter les fuites
- N'utilisez pas d'E/S réelles dans les tests unitaires (utilisez **mocks/fixtures**)
- Laissez les tests d'intégration hors du noyau des tests unitaires (E/S, gros volumes)

15

15

UNIT TESTS

MONKEYUSER.COM



16

16

- **pytest** : <https://docs.pytest.org/>
- **numpy.testing** : <https://numpy.org>
- **Hypothesis** : <https://hypothesis.readthedocs.io/>
- **pytest-mpl** : <https://github.com/matplotlib/pytest-mpl>