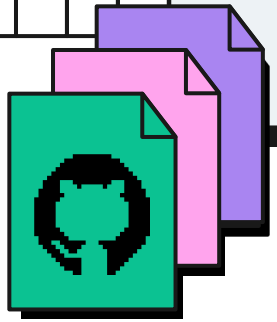
# Tests & intégration continue



Du code qui fonctionne comme prévu et reste fiable

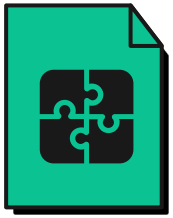
# Tests

Les tests permettent de vérifier que le **code fonctionne comme prévu** et aident à **détecter les erreurs** lors de modifications apportées au programme

# Types de tests

## TESTS UNITAIRES

Vérifient que chaque "unité" de code fonctionne correctement

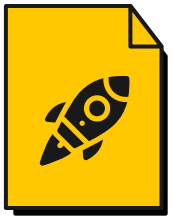


## TESTS D'INTÉGRATION

Testent l'interaction entre plusieurs composants d’un projet

## RECETTES FONCTIONNELLES

Vérification manuelle du fonctionnement conforme d’une application



## TESTS DE PERFORMANCE

Mesurent le temps d'exécution et la consommation de ressources

# Quel intérêt d'écrire des tests ?

Modification du code sans crainte de casser ce qui fonctionne

**Filet de sécurité**

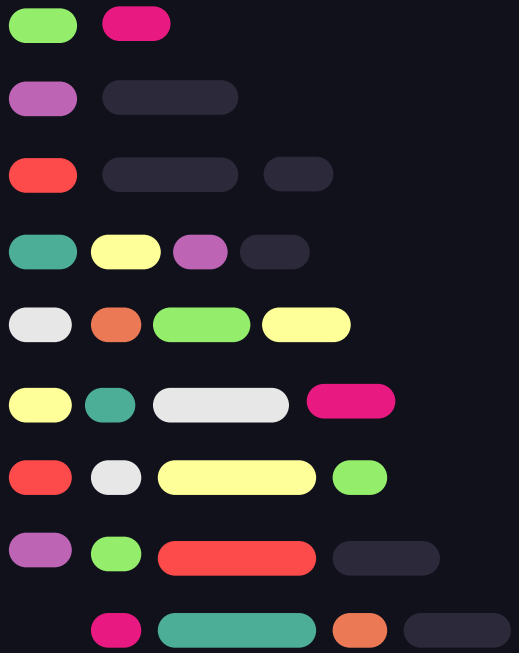
Détection des erreurs qui auraient été problématique plus tard

**Zéro bug invisible**

Comme créer un cahier des charges de tous les cas que le code doit gérer

**Clarifie l'objectif**

# Exemple de test

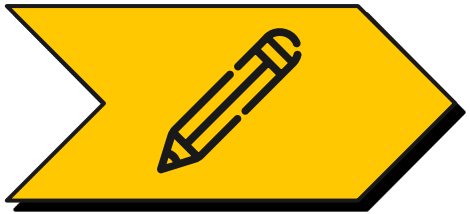


```python
# code à tester
def add(a, b):
    return a + b

# fonction de test
def test_add():
    assert add(2, 3) == 5
```

# Intégrer des tests à son projet

Identifier les points critiques

- ce qui pourrait planter
- gestion de données externes (fichier, API)
- traitements ou calculs importants



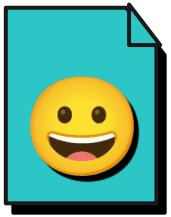
Tester les cas limites

- valeurs extrêmes (vide, très grand, négatif… )
- Formats invalides (mauvais type, données malformées)
- Situations d'erreur (fichier manquant, timeout...)

Tester par ordre de priorité

- Commencer par les tests unitaires des fonctions critiques
- Ajouter des tests plus développés en fonction des problématiques rencontrées

# Quoi tester ?

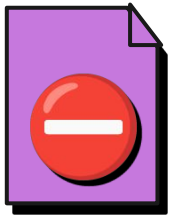


## HAPPY PATH

Vérifier que le code s'exécute et renvoie les bonnes valeurs pour une utilisation normale

## EDGE CASES

Tester que le code peut gérer lors de cas limites ou de valeurs extrêmes



## ERROR CASES

Vérifier que le code ne fait pas n'importe quoi quand les entrées sont invalides, tester ce qui peut mal tourner

## Happy path

```
def test_normal_download():
    """Test du cas d'utilisation normal."""

    url = "https://gallica.bnf.fr/iiif/image.jpg"
    result = download_image(url)

    assert result is not None
    assert len(result) > 0
```

## Edge cases

```python
def test_edge_download():
    """Test de cas limites"""

    # Very big image
    assert download_image(big_image_url, max_size=1000000)

    # Very long URL
    long_url = "https://example.com/" + "a" * 2000
    assert download_image(long_url)
```

Error cases

```python
def test_error_cases():
    """Test de cas causant des erreurs"""

    # Invalid URL
    with pytest.raises(ValueError):
        download_image("not-a-url")

    # Server that does not respond
    with pytest.raises(TimeoutError):
        download_image("http://very-slow-server.com")

    # File that is not an image
    assert download_image("https://example.com/text.txt") is None
```

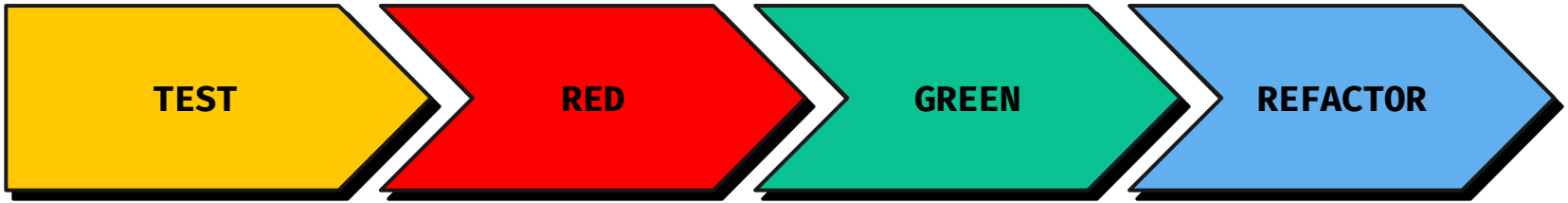
# ✅ Checklist

**INPUT**
Types attendus (str, int, dict...)
Formats (URL, JSON, nombres positifs...)
Valeurs limites (vide, très grand, très petit...)

**EXECUTION**
Problèmes réseau
Fichiers manquants
Timeout
Erreurs d'authentification

**OUTPUT**
Format du résultat
Quelle sortie en cas d'erreur
Effets de bord (fichiers créés, logs...)

# Test Driven Development




Écrire le test avant la fonction à tester

Vérifier que le test échoue lorsqu'il est lancé

Écrire la fonction minimale pour réussir le test

Améliorer le code de la fonction

...

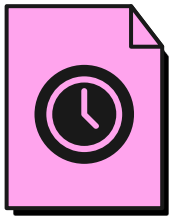
Nan mais que je soit sûr·e d'avoir bien compris …

## Ça fait pas juste 2 fois plus de code à écrire ?

Écrire les tests avant le code forcent à penser aux cas d'usages réels. On évite le code inutile vu qu'on code uniquement pour faire passer les tests. À terme c'est beaucoup de temps gagné

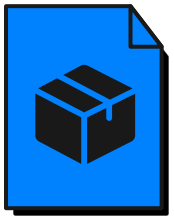
# Framework Pytest

Pour initialiser pytest sur son projet il faut :



## INSTALL

Installer le package dans son environnement de projet avec **pip install pytest**



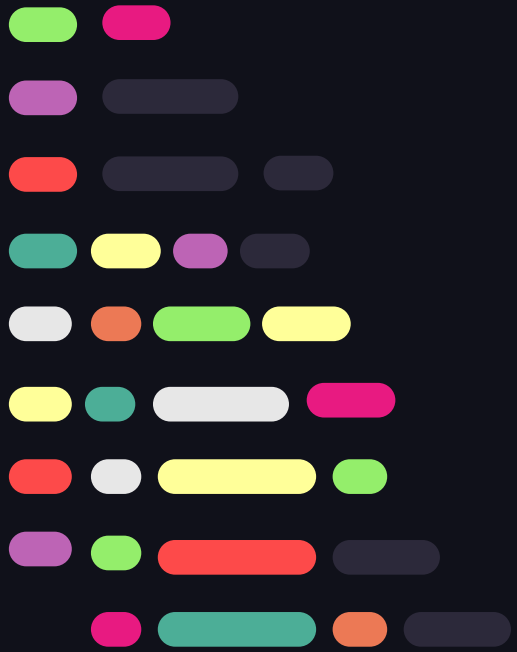
## FICHIERS

Créer un dossier **tests/** contenant des fichier nommés **test_<module>.py**

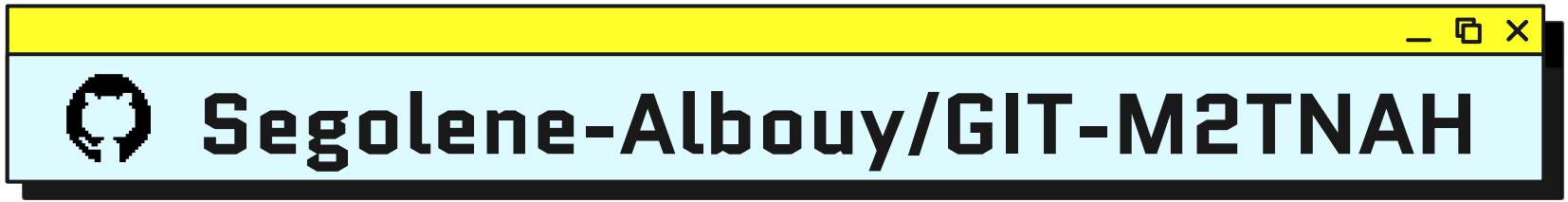
## FONCTIONS

Définir des fonctions de tests nommées **test_<fonction>()**

# Lancer les pytests



```
# Exécuter tous les fichiers test_*.py
pytest

# Tests sans masquer le stdout (e.g. print)
pytest -s

# Lister toutes les tests passés (--verbose)
pytest -v
```




tinyurl.com/py-template

## Clone

Cloner le template en local

## Inspect

Explorer les fichiers

## Repo

Comparer fichiers et *repository*

## pre-commit

Installer les pre-commit

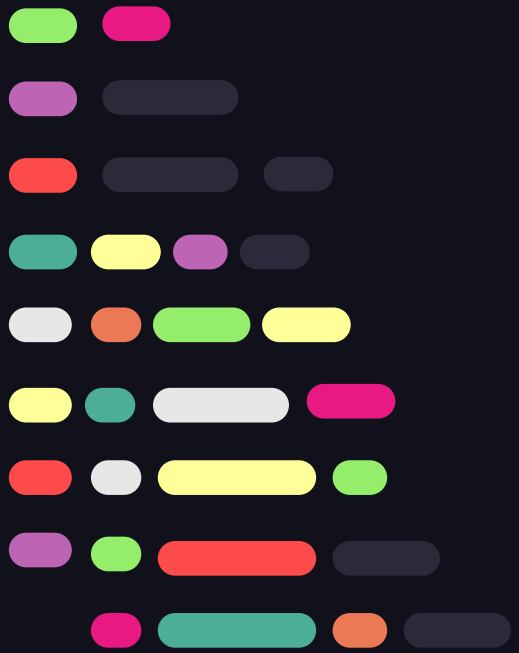
## README

Lire les instructions du README

## main

Exécuter la fonction main.py

# Exercice pratique TDD



1. Objectif : une fonction qui récupère du texte depuis internet
2. Tests : définir des tests pour cette fonction (happy, edge, error) + créer une fonction vide
3. RED : lancer les tests
4. Code : rédiger la fonction qui passe les tests

# Utilisation de assert

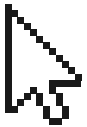
Pour afficher un message d'erreur personnalisé
**assert age > 0, "L'âge doit être positif"**

Fonction à tester

```python
# src/maths.py

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

Test de la fonction

```python
from src.maths import divide

def test_divide():
    assert divide(8, 4) == 2
    assert divide(24, 3) == 8
    assert divide(-5, 2) == -2.5
```



# Parametrization

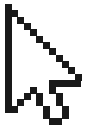
Ne stoppe pas l'exécution des tests si l'un des paramètres ne réussit pas

Fonction à tester

```python
# src/maths.py

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

Test de la fonction

```python
from src.maths import divide

@pytest.mark.parametrize("a,b,out", [
  (8, 4, 2),
  (24, 3, 8),
  (-5, 2, -2.5)
])
def test_divide(a, b, out):
    assert divide(a, b) == out
```



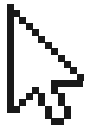
# Vérification d'erreurs

Fonction à tester

```python
# src/maths.py

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

Test de la fonction

```python
from src.maths import divide

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



# Fixtures

Une fixture est une fonction qui **prépare l'environnement** de test, **génère des données** (objets, valeurs, connexion) et **les nettoie** après exécution.

# Fixtures

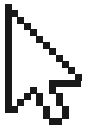
Par défaut les fixtures contenues dans **test/conftest.py** sont accessibles par toutes les fonctions de test des fichiers dans **test/**

Fonction à tester

```python
# src/maths.py

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

Test de la fonction

```python
@pytest.fixture
def test_cases():
    return [
        (8, 4, 2),
        (24, 3, 8),
        (-5, 2, -2.5)
    ]

def test_divide(test_cases):
  for a, b, out in test_cases:
      assert divide(a, b) == out
```



Par défaut les fixtures contenues dans **test/conftest.py** sont accessibles par toutes les fonctions de test contenues dans **test/**

conftest.py

```
test/
  ├── test_models/
  │   ├── test_utils/
  │   │   ├── conftest.py     <-- Fixtures locales
  │   │   └── test_utils.py   <-- ✅ conftest
  │   ├── test_models.py      <-- 🚫 conftest
  └── conftest.py             <-- Fixtures globales
```

# Exercice

01 { ..

Rédiger des tests pour un modèle

} ..

## Person

Inspecter la classe Person

## Conftest

Compléter les fixtures

## Tests

Rédiger les tests dans test_person.py

## Add/commit

Sauver les modifications effectuées

## Push

Publier le commit sur le repo distant

## Pipeline

Observer le *repository* GitHub

pre-commit.yaml

```
# Pour updater les hook
pre-commit autoupdate

# Pour désinstaller un hook
pre-commit uninstall && pre-commit install

# Pour commit sans utiliser pre-commit
git commit --no-verify
```

# Mocking

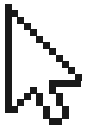
Le mocking permet de **simuler le comportement** de composants externes (API, BDD, etc.) pour tester le code de manière contrôlée.

Fonction à tester

```python
# src/get.py

def get_txt(url):
    res = requests.get(url)
    return res.text
```

Test de la fonction

```python
def test_get_txt():
    with patch('requests.get') as mock_get:
        # Définir une valeur de retour
        mock_get.return_value.text = “hello”

        text = get_txt(“url”)

        # Vérifier que la fonction renvoie la valeur
        assert text == “hello”
        mock_get.assert_called_once_with(“url”)
```



# Mocking

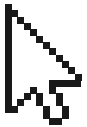
Le mocking permet de **simuler le comportement** de composants externes (API, BDD, etc.) pour tester le code de manière contrôlée.

Fonction à tester

```python
# src/json.py

def get_json(url):
    res = requests.get(url)
    return res.json()
```

Test de la fonction

```python
def test_get_json():
    with patch('requests.get') as mock_get:
        # Définir une valeur de retour
        mock_get.return_value.json.return_value = 12

        text = get_json("url")

        # Vérifier que la fonction renvoie la valeur
        assert text == 12
        mock_get.assert_called_once_with("url")
```



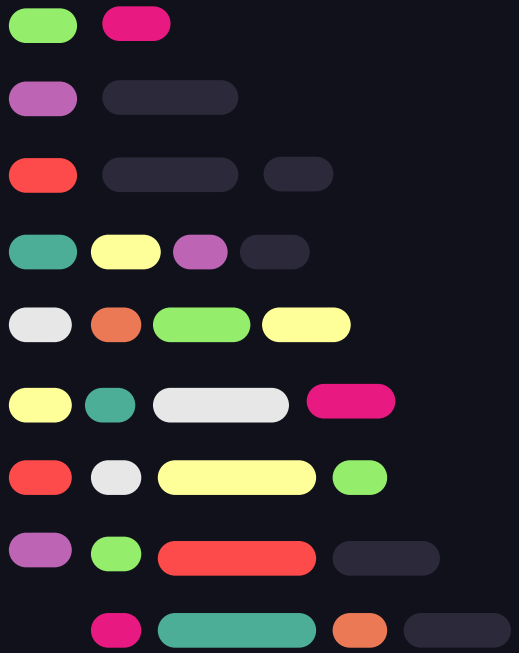
# Setup / teardown


**Setup** prépare l'environnement de test avant chaque test (par exemple : créer des fichiers temporaires, initialiser une base de données de test).

Exécution de la fonction de test qui prend en entrée la valeur “*yieldée*” par la fixture

**Teardown** nettoie après chaque test pour éviter les effets de bord (par exemple : supprimer les fichiers temporaires, vider la base de données).

# Set up / Teardown



```python
@pytest.fixture
def temp_file():
    # Setup: création d'un fichier avant le test
    path = "temp.txt"
    with open(path, "w") as f:
        f.write("test")

    # Le test reçoit le path
    yield path

    # Teardown: le fichier est supprimé
    os.remove(path)
```

# CI/CD

CI (Intégration Continue)/ CD (Déploiement Continu) est un paradigme de développement qui vise à automatiser la mise en production de projets

# GitHub actions

Outils d'automatisation intégrés à GitHub permettant de créer des pipelines

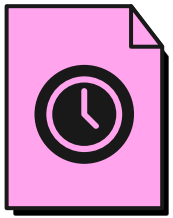
**Déclencheurs** : push, pull request, planification
**Jobs** : tâches indépendantes (tests, build, deploy)
**Steps** : actions pré-définies ou commandes shell

[Marketplace d'actions réutilisables]

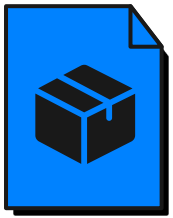
.github/workflow/<action>.yml

```yaml
name: Tests Python
on: [push]  # Déclenché sur chaque push

jobs:
  test:
      runs-on: ubuntu-latest
      steps:
         - uses: actions/checkout@v2
         - name: Set up Python
           uses: actions/setup-python@v2
         - name: Run tests
           run: |
               pip install pytest
               pytest
```

# Exemples de pipelines



TEST



DEPLOY

DOCS

# GitHub pages

1. Créer un *repository* nommé exactement comme votre username
2. Ajouter un fichier index.html
3. (ajouter des fichiers html, css et javascript)
4. Settings > Pages > Select branch

C'est en ligne !
→ https://username.github.io