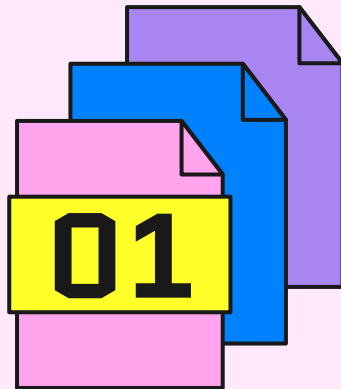
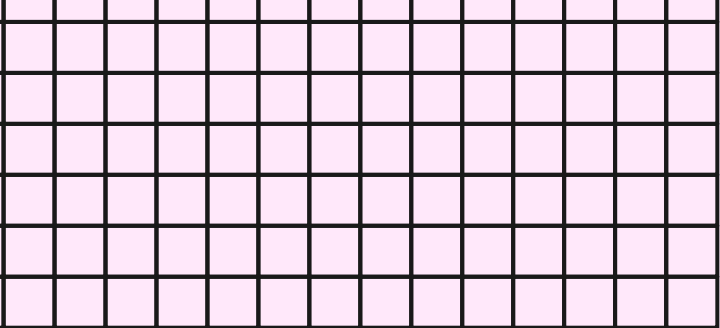


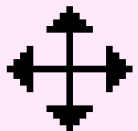


Bonnes pratiques de code et tests

Écrire du code lisible,
efficace et robuste



LISIBILITÉ



Code lisible = Code maintenable

Fondamentaux d'un code lisible



01

CONVENTION DE NOMMAGE

Noms de fonctions et
variables explicites



02

DOCUMENTATION & COMMENTAIRE

Du code
auto-documenté, des
paramètres typés



03

AUTOMATISER TOUT ÇA

Avoir du code propre
sans rien avoir à
faire

01

Convention de nommage

Les noms de variables et de fonction doivent refléter l'intention :
Ils sont comme des commentaires qui ne vieillissent pas

```
# Que fait cette fonction ?
```

```
def f(x):  
    y = x.split()  
    z = len(y)  
    return z
```

01

Convention de nommage

Les noms de variables et de fonction doivent refléter l'intention :
Ils sont comme des commentaires qui ne vieillissent pas

```
# Et maintenant ?
```

```
def count_words(text):  
    words = text.split()  
    word_count = len(words)  
    return word_count
```

PEP 8 – Nommage python



snake_case : variables, fonctions et fichiers
`ma_variable`, `ma_fonction()`, `mon_module.py`, etc.



MAJ : pour les constantes en début de fichier
`MAX_VALUE`, `PI`, etc.



CamelCase : pour les noms de classes
`class MaClasse:`



snake_case : méthodes, attributs privés
`_attribut`, `_ma_methode()`, etc.

02

Commentaires

```
# Cette fonction rend un texte utilisable
def process_texte(data):
    # Enlever les espaces au début et à la fin
    resultat = data.strip()
    # Convertir tout en minuscule
    resultat = resultat.lower()
    # Remplacer les espaces multiples par un seul
    resultat = " ".join(resultat.split())
    # Retourner le texte normalisé
    return resultat
```

Les commentaires doivent être maintenus, alors qu'un code explicite se suffit à lui-même

02

Commentaires

```
def normalize_text(raw_text):  
    trimmed_text = text.strip()  
  
    lowercase_text = trimmed_text.lower()  
  
    word_list = cleaned_text.split()  
  
    normalized_text = " ".join(word_list)  
  
    return normalized_text
```

Les commentaires doivent être maintenus, alors qu'un code explicite se suffit à lui-même

02

Docstrings

Commentaire structuré qui décrit une l'**objectif** de la fonction, ses **arguments** et **valeur retour** : utilisé pour générer de la **documentation automatique** (e.g. sphinx)

Commenter intelligent

```
def total(numbers):  
    """  
    Calculate the total sum of a list of numbers.  
  
    Args:  
        numbers (list of int): The numbers to sum.  
  
    Returns:  
        int: The total sum.  
    """  
    return sum(numbers)
```

02

Typing

Le *typing* précise les types attendus pour les **arguments** et les **retours** de fonction :
il permet à l'IDE détecter des erreurs potentielles avant l'exécution

Décrire les entrées et sorties

```
def total(numbers: list[int]) -> int:
    # prend en entrée une liste d'entiers et renvoie un entier
    return sum(numbers)

def welcome(name: Optional[str]) -> None:
    # prend en entrée optionnelle une chaîne de caractère et ne renvoie rien
    print(f"Bienvenue {name or 'invité'} !")

def mean_grades(grades: dict[str, list[int, float]]) -> float:
    # In: {"geo": [12, 14.5], "math": [16]} ⇒ Out: moyenne
    all = [grade for grade in list(grades.values())]
    return sum(all) / len(all)
```

Documentation

En plus de la documentation interne au code, il est utile de décrire comment le projet fonctionne dans sa globalité (objectifs, *install*, structure, *known issues*, etc.) :

Dans le [README](#) ou la [documentation](#) du repository (onglet wiki)

03

Docstrings automatiques



PyCharm

Module intégré par défaut

Settings > Tools
> Python Integrated Tools
> [Docstring format](#)



VS Code

Plugin **autoDocstring**

[Lien d'installation](#) avec détails
de configuration

Auto-complétion après l'ouverture de
triple-quotes `"""`

03

Générer de la doc automatique

1. Installer sphinx
`pip install sphinx`
2. Créer un dossier à la racine du module `root_dir`
`mkdir docs && cd docs`
3. Initialiser un projet
`sphinx-quickstart`
4. Modifier `conf.py`
`sys.path.insert(0, os.path.abspath(' ../ ../root_dir'))`
`extensions = ['sphinx.ext.autodoc']`
5. Ajouter le module à `index.rst`
`.. automodule:: file/subdir`
`:members:`
6. Générer la doc
`make html`



03

Hooks automatiques



1. Installer black
`pip install pre-commit`
2. Définir `.pre-commit-config.yaml`

```
repos:  
- repo: https://github.com/psf/black  
  rev: stable  
  hooks:  
  - id: black
```
3. Installer pre-commit
`pre-commit install`
4. Lancer le *linting* sur les fichiers ajoutés
`pre-commit run --all-files`
`pre-commit run --files path/to/your/directory/*`

Exercice 1



01 {

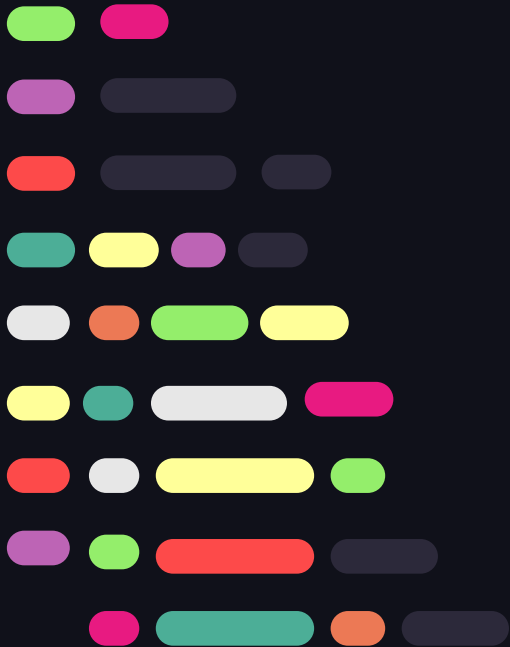
Documentation et
linting automatique



} ..



Nettoyer ce code



```
# 03-Testing/exercices/lisibility/original.py
def nb_w(t, words):
    nb = {}
    for w in t.split():
        if w in words:
            if w not in nb:
                nb[w] = 0
            nb[w] += 1
    return nb
```




Nommage

Renommer
variables et
fonction

Typing

Typer les
inputs/outputs de
la fonction

Docstring

Rédiger le
commentaire
explicatif

Linting

Appliquer black
pour le formatage

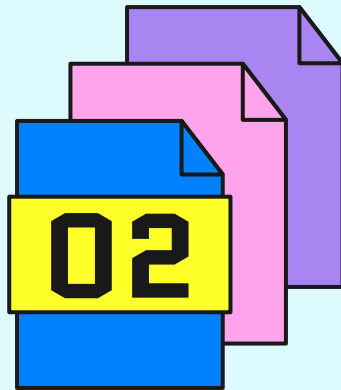
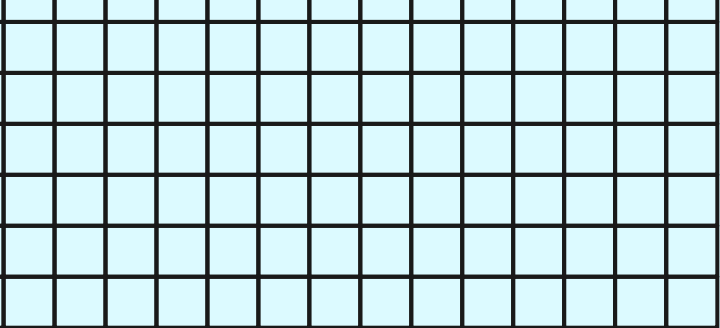
Sphinx

Générer une
documentation
avec Sphinx

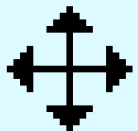
Afficher

Ouvrir [index.html](#)
dans le navigateur





EFFICACITÉ



Construire un projet de code performant

Fondamentaux d'un code efficace



01

STRUCTURE

Organisation des modules du général au particulier



02

MODULARITÉ

Fonctions courtes et réutilisables pour éviter les répétitions



03

PERFORMANCE

Code simple, mesuré et optimisé pour l'efficacité

01

Module ?

Un module python est un fichier `.py` qui peut être importé au sein d'un autre module python.

Les modules python peuvent être regroupé en package, dans un dossier contenant un `__init__.py`

01

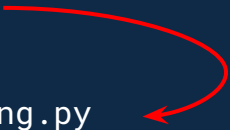
Logique des imports

Séparer les **modules fondamentaux** (code et constantes utilisés partout), des **modules spécifiques** dans des packages différents.

Pour éviter les imports circulaires, on appelle toujours le général dans le particulier

```
project/
├── core/
│   ├── __init__.py
│   ├── config.py
│   └── utils.py
├── features/
│   ├── __init__.py
│   ├── preprocessing.py
│   └── analysis.py
└── main.py
```

from core.utils import *



01

Importer un package

Le fichier `__init__.py` sert de point d'entrée aux modules du dossier

En y important le contenu des modules, on peut simplifier les imports

```
# project/core/__init__.py
from .utils import useful_function
```

```
# project/features/preprocessing.py
from core import useful_function
```

01

Import relatifs / absolus

Absolu

Utilisent le chemin complet depuis la racine du projet

Pour les gros projets car plus explicites et maintenables : fonctionnent quel que soit l'emplacement

```
from core.utils import func
```

VS

Relatif

Référencent le module depuis la position actuelle

Petits packages indépendants de la structure globale pour être facilement déplaçables

```
from ..utils import func
```



OSSEKOUR !



J'ai une erreur d'import ! 😭

ModuleNotFoundError correspond au cas spécifique où le module n'est pas trouvé du tout :

- dépendance non installée
- fichier qui n'existe pas ou mal orthographié
- fichier `__init__.py` manquant

ImportError peut être dû à :

- Module qui existe mais élément indisponible
- Import circulaire (deux fichiers ou plus qui s'appellent mutuellement)
- Import relatif à l'extérieur d'un package

Résoudre une erreur d'import

01

Module
externe ?

OUI

Vérifier l'installation
`pip install <module>`

Vérifier l'environnement
`which python`
`pip list`

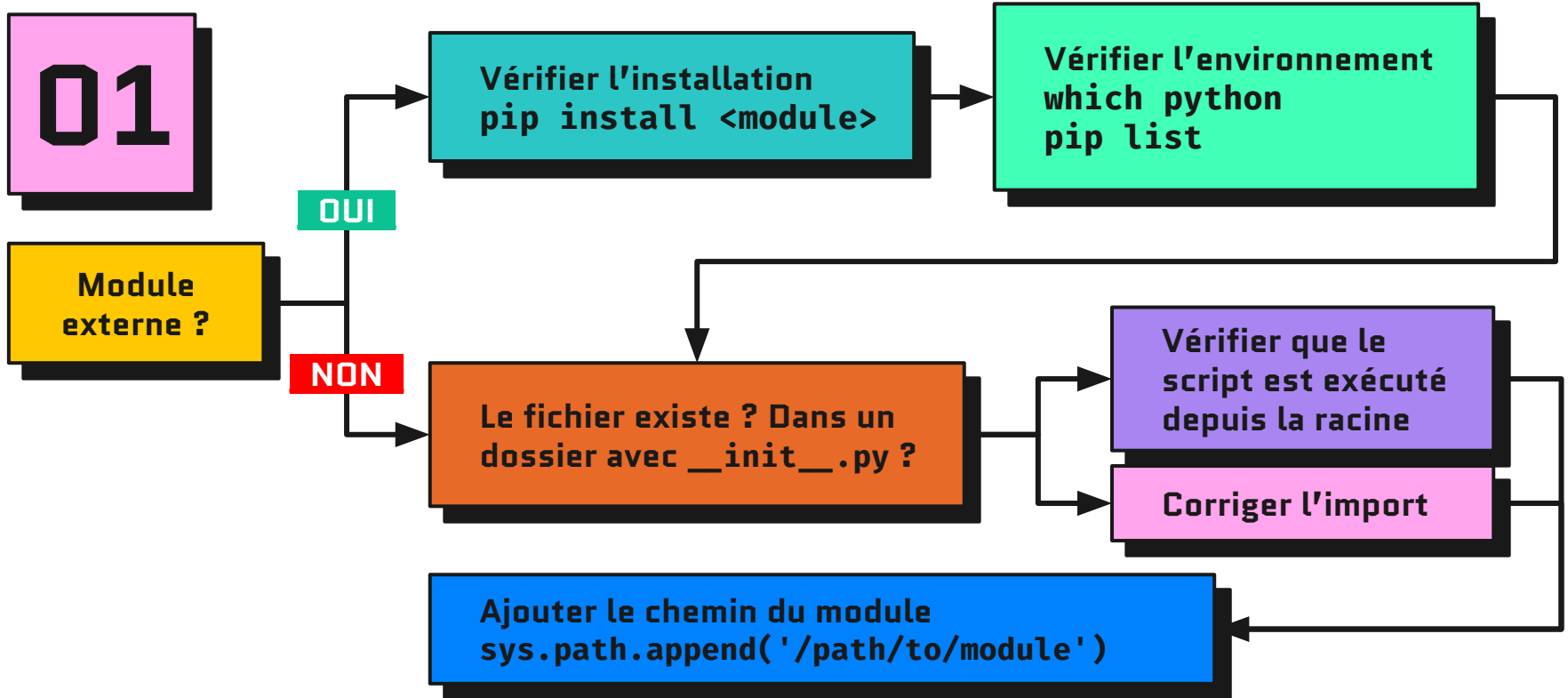
NON

Le fichier existe ? Dans un
dossier avec `__init__.py` ?

Vérifier que le
script est exécuté
depuis la racine

Corriger l'import

Ajouter le chemin du module
`sys.path.append('/path/to/module')`



01

Import circulaire

Identifier dans l'erreur les noms de fichiers qui font une boucle

Erreur classique

```
Traceback (most recent call last):
  File "module_a.py", line 1, in <module>
    from module_b import func_b
  File "module_b.py", line 1, in <module>
    from module_a import func_a

ImportError: cannot import name 'func_a' from partially
initialized module 'module_a' (most likely due to a circular import)
(/path/to/module_a.py)
```

Pour résoudre le problème : soit déplacer l'**import à l'intérieur de la fonction** ou **extraire la/les fonction(s)** fautive(s) dans un fichier partagé.

Exercice 2



02 {

Résoudre un import
circulaire



} ..

02

Le code modulaire



Lisible

Plus facile
d'identifier
l'objectif du code



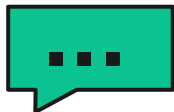
Réutilisable

Écrire une seule
fois le code pour
l'intégralité du
projet



Maintenable

Plus facile de
corriger un bug ou
modifier une fonction
bien isolée



02

Modularité

```
def user_welcome_message(user):  
    username = user.strip().capitalize()  
    return f"Hello, {username}! Welcome back."  
  
def user_goodbye_message(user):  
    username = user.capitalize()  
    return f"Bye, {username}! See you soon."
```

Lorsqu'une fonctionnalité, même simple est utilisée à plusieurs endroits du code, il est utile de la transformer en fonction indépendante

02

Modularité

```
def get_username(user):  
    return user.strip().capitalize()  
  
def user_welcome_message(user):  
    username = get_username(user)  
    return f"Hello, {username}! Welcome back."  
  
def user_goodbye_message(user):  
    username = get_username(user)  
    return f"Bye, {username}! See you soon."
```

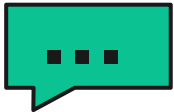
Lorsqu'une fonctionnalité, même simple est utilisée à plusieurs endroits du code, il est utile de la transformer en fonction indépendante.

Si on a des besoins au cas par cas, on peut ajouter cela dans les arguments

02

Une fonction modulaire

Une = Une
fonction action



02

Programmation fonctionnelle ?

Paradigme de programmation pour
éviter les effets de bord :

- les **arguments contiennent tout**
ce dont la fonction a besoin
- La fonction **renvoie toute**
valeur modifiée par elle

...

02

Repérer le code répété



PyCharm

Module intégré par défaut

Settings > Editor
> Inspections > General
> [Duplicated code fragments](#)



VS Code

Plugin **Duplicated code**

[Lien d'installation](#)
[Code refactoring](#) in VScode



À retenir

Si vous utilisez copier/coller en codant c'est probablement que vous pouvez créer une fonction

Exercice 3



03 {

Factoriser du
code dupliqué



} ..

03

Performance

Des choses simples permettent de veiller à la performance de votre code.

Qu'est-ce qui ralentit généralement un code ?
boucles inutiles,
calculs répétitifs,
structures de données inadéquates, etc.

```
# Voyez vous le souci ?
```

```
data = [1, 2, 3, 4, 5]
```

```
for i in range(len(data)):  
    result = (len(data) ** 2) + data[i]  
    print(result)
```

03

Performance

Des choses simples
permettent de veiller à
la performance de votre
code

Par exemple en
effectuant les calculs à
l'extérieur des boucles
quand cela est possible

```
data = [1, 2, 3, 4, 5]

# Calculé une seule fois
constant = len(data) ** 2

for i in range(len(data)):
    result = constant + data[i]
    print(result)
```

03

Temps d'exécution

Pour repérer les fonctions qui prennent le plus de temps à s'exécuter, on peut utiliser un wrapper



Définir un wrapper de fonction

```
def timer(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        execution_time = end_time - start_time
        print(f"\n[{func.__name__}]: {execution_time:.3f} secondes")
        return result
```

return wrapper

Utiliser en précédant sa fonction avec le décorateur @timer

03

Structure de données

```
data = [x for x in range(10**6)]  
  
if 999999 in data:  
    print("Found!")
```

Du code simple, avec un objectif bien défini.

Un code frugal fait exactement ce qu'on lui demande, ni plus ni moins

Emploi de structures de données adaptées

03

Structure de données

```
# Utilisation d'un set : valeurs uniques  
data = {x for x in range(10**6)}  
  
if 999999 in data:  
    print("Found!")
```

Les structures comme set ou dict sont plus performantes pour des recherches fréquentes, mais utilisent plus de mémoire.

03

Allocation de mémoire

```
def double_numbers(numbers):  
    result = []  
    for n in numbers:  
        # Nouvelle liste à chaque itération  
        result = result + [n * 2]  
    return result  
  
def build_sentence(words):  
    sentence = ""  
    for word in words:  
        # Variable créée à chaque itération  
        sentence = sentence + word + " "  
    return sentence.strip()
```

Pour assurer une bonne allocation de mémoire, utilisez des structures adaptées aux besoins : évitez de stocker des données inutiles en mémoire, et limitez la création d'objets temporaires dans des boucles.

03

Allocation de mémoire

```
def double_numbers(numbers):  
    result = []  
    for n in numbers:  
        # Pas de nouvelle allocation de mémoire  
        result.append(n * 2)  
    return result  
# Encore plus efficace : list-comprehension  
return [n * 2 for n in numbers]  
  
def build_sentence(words):  
    # Une seule opération de concaténation  
    return " ".join(words)
```

Pour assurer une bonne allocation de mémoire, utilisez des structures adaptées aux besoins : évitez de stocker des données inutiles en mémoire, et limitez la création d'objets temporaires dans des boucles.

Exercice 4

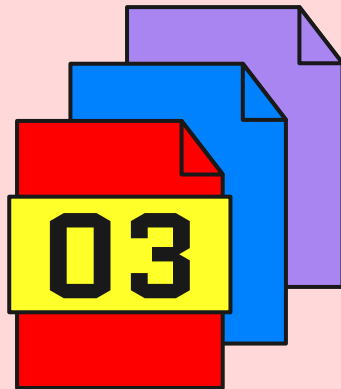
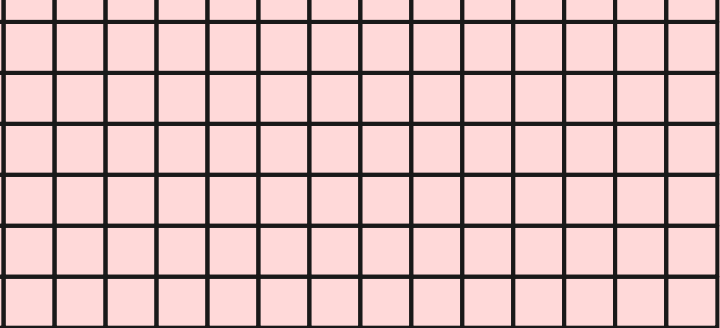


03 {

Améliorer la
performance



} ..



ROBUSTESSE



Être certain d'éviter les bugs



Tests

Il permettent de vérifier que le code fonctionne comme prévu et détectent les erreurs dès que des modifications sont apportées au programme.

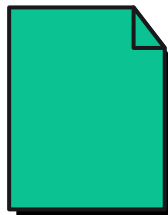
Les tests aident à éviter les bugs et à maintenir un code de meilleure qualité sur le long terme.

Types de tests



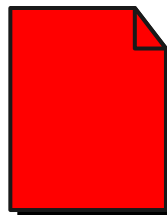
TESTS UNITAIRES

Vérifient que
chaque "unité" de
code fonctionne
correctement



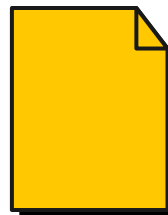
TESTS D'INTÉGRATION

Testent l'interaction
entre plusieurs
composants



RECETTES FONCTIONNELLES

Vérification
manuelle du
fonctionnement
conforme d'une
application



TESTS DE PERFORMANCE

Mesurent le temps
d'exécution et la
consommation de
ressources

Test Driven Development

TEST

Écrire le test
avant la
fonction à
tester

RED

Vérifier que le
test échoue
lorsqu'il est
lancé

GREEN

Écrire la
fonction
minimale pour
réussir le test

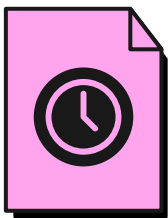
REFACTOR

Améliorer le
code de la
fonction

...

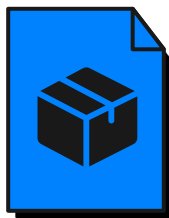
Pytest

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 un
fichier
test_<module>.py pour
chaque module à tester



FONCTIONS

Définir des fonctions
de tests nommées
test_<truc>() pour
vont vérifier que le
code s'exécute
correctement

Lancer les tests avec la commande **pytest**

Utilisation de assert

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

Fonction à tester

```
# src/maths.py  
  
def divide(a, b):  
    return a / b
```

Test de la fonction

```
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

```
# src/maths.py

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

Test de la fonction

```
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

```
# src/maths.py

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

Test de la fonction

```
from src.maths import divide

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

Fixtures

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

Fonction à tester

```
# src/maths.py

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

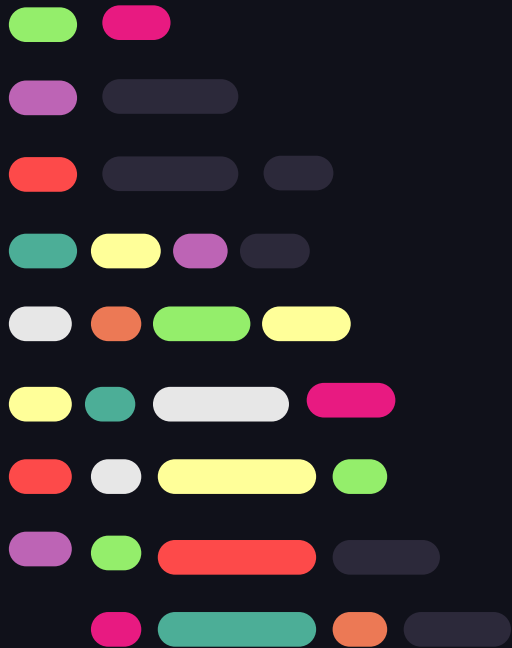
Test de la fonction

```
@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
```

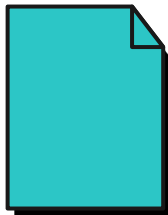


Contourner une Exception



```
try:
    response = requests.get(url, timeout=5)
    response.raise_for_status()
    return response.content
except Exception as e:
    print(f"[{e.__class__.__name__}] : {e}")
    return None
```

Quoi tester ?



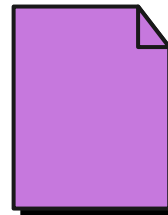
HAPPY PATH

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

```
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

```
def test_error_cases():  
    """Test de cas causant des erreur"""  
  
    # 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

Quelles sont les entrées possibles ?

Types attendus (str, int, dict...)

Formats (URL, JSON, nombres positifs...)

Valeurs limites (vide, très grand, très petit...)

Que peut-il se passer pendant l'exécution ?

Problèmes réseau

Fichiers manquants

Timeout

Erreurs d'authentification

Quelles sont les sorties attendues ?

Format du résultat

En cas d'erreur

Effets de bord (fichiers créés, logs...)