Normes de développement Python

Ce document décrit les étapes à suivre pour installer et configurer un environnement de travail **Python** conforme aux standards internes. Il est destiné à tous les intervenants amenés à travailler sur des projets Python, afin de garantir un cadre cohérent et opérationnel.

1. Environnement de Travail

1.1 Environnements virtuels

```
python -m venv venv
source venv/bin/activate  # Linux/macOS
venv\Scripts\activate     # Windows
```

2. Structure Standard des Projets

```
/project_name/
├── src/
│   └── module/
│       └── __init__.py
├── tests/
├── requirements.txt
├── README.md
├── .env
├── .gitignore
└── pyproject.toml ou setup.py
```

- Code principal dans src/
- Tests séparés dans tests/
- README.md et .env obligatoires

3. PEP8 (Python Enhancement Proposal 8)

La PEP8 est le guide officiel de style pour le langage Python. Elle définit un ensemble de règles destinées à rendre le code Python plus lisible, cohérent et maintenable par l'ensemble de la communauté. Son respect est fortement recommandé dans tout projet collaboratif.

Parmi les recommandations principales de la PEP8 :

• *Limiter la longueur des lignes à 79 ou 88 caractères*

Cela permet de faciliter la lecture sur tous types d'écrans et d'éviter les lignes horizontales difficiles à suivre dans les revues de code ou les outils de versionnage.

Exemple correct (ligne courte) :

```
email_message = f"Bonjour {prenom}, votre commande a bien été enregistrée."
```

Exemple incorrect (ligne trop longue) :

```
email_message = f"Bonjour {prenom}, votre commande a bien été enregistrée.
Merci pour votre confiance et à bientôt sur notre plateforme."
```

• *Utiliser 4 espaces pour l'indentation (jamais de tabulations)*

Python repose sur l'indentation pour structurer les blocs de code. Utiliser 4 espaces garantit la compatibilité entre tous les éditeurs.

Correct :

```
def calcul_total(prix):
    total = prix * 1.2
    return total
```

Incorrect (tabulations ou mauvaise indentation) :

```
def calcul_total(prix):
     total = prix * 1.2
     return total
```

• *Laisser 2 lignes vides entre deux définitions de classes ou fonctions de haut niveau*

Cela améliore la lisibilité et la séparation logique entre les blocs.

Correct :

```
def fonction_a():
    pass


def fonction_b():
    pass
```

Incorrect :

```
def fonction_a():
    pass

def fonction_b():
    pass
```

• Organiser les imports en trois blocs : standard, bibliothèques tierces, modules internes

Chaque bloc doit être séparé par une ligne vide.

Exemple :

```
import os
import sys

import numpy as np
import pandas as pd

from monprojet.module import calcul
```

Ces conventions favorisent un code plus propre, plus compréhensible par tous, et plus facile à maintenir dans le temps.

3.2 Nommage

Élément	Convention	Exemple
Variable	snake_case	total_amount
Fonction	snake_case	process_file()
Classe	PascalCase	DataProcessor
Constante	UPPER_CASE	MAX_RETRIES
Fichier	snake_case	utils.py

3.3 Typage

Depuis Python 3.5, il est possible d'indiquer le type attendu **des variables, des arguments et des valeurs de retour d'une fonction,** sans que cela soit obligatoire à l'exécution**. C'est ce qu'on appelle le** typage statique ou facultatif**, ou encore les** annotations de type**.**

```
def greet(name: str) -> str:
    return f"Hello, {name}"
```

- name: str signifie que l'argument name est une **chaîne de caractères (str)** attendue.
- -> str signifie que **la fonction retourne aussi une chaîne de caractères**.

Cela n'empêche pas Python d'exécuter le code si un autre type est passé, mais cela permet :

- une **meilleure documentation automatique** (ex : avec VS Code, PyCharm, Swagger, Sphinx...),
- une **détection anticipée d'erreurs** avec des outils comme mypy.

3.5 README.md

Le fichier README.md est essentiel pour tout projet Python : il permet à toute personne accédant au dépôt de comprendre immédiatement la finalité du projet, comment l'installer, l'utiliser et le faire évoluer. Il doit être clair, structuré et à jour.

• Objectif du projet

Décrire de manière concise le but du projet, son utilité, à qui il est destiné, et éventuellement le contexte technique ou fonctionnel.

• Installation et lancement

Expliquer étape par étape comment configurer l'environnement local : prérequis, création d'environnement virtuel, installation des dépendances, lancement du programme.

• Liens utiles / API

Indiquer les URL importantes, les endpoints d'API (si applicable), les documents techniques, ou autres ressources internes.

Un bon README.md facilite l'adoption du projet, accélère l'onboarding des nouveaux développeurs et permet une meilleure collaboration.

4. Tests Unitaires

Les tests unitaires permettent de vérifier que chaque fonction ou composant du code fonctionne comme prévu de manière isolée. Ils jouent un rôle clé dans la stabilité, la maintenabilité et la confiance dans le code.

4.1 Objectifs des tests

- Détecter les erreurs le plus tôt possible dans le cycle de développement
- Documenter le comportement attendu des fonctions
- Permettre des modifications du code sans crainte de régressions
- Intégrer facilement les vérifications dans une pipeline CI/CD

4.2 Organisation recommandée

- Tous les fichiers de test doivent être regroupés dans un dossier tests/
- Chaque module dans src/ devrait avoir son fichier de test dédié dans tests/
- Les fichiers de test doivent commencer par test_ : ex. test_utils.py

Exemple d'arborescence :

```
mon-projet/
├── src/
│   └── utils.py
└── tests/
    └── test_utils.py
```

4.3 Framework recommandé : pytest

pytest est un framework simple et puissant qui détecte automatiquement les fichiers de test, fournit des rapports clairs et facilite l'écriture de tests efficaces.

Installation :

```
pip install pytest
```

Exécution :

```
pytest tests/
```

4.4 Couverture de code

Utiliser pytest-cov pour mesurer la couverture du code par les tests :

```
pip install pytest-cov
pytest --cov=src/ tests/
```

Objectif minimum recommandé : **80%** de couverture.

4.5 Bonnes pratiques

- Nommer les tests de façon explicite (test_calcul_sans_remise plutôt que test1)
- Isoler chaque test (ne pas dépendre d'un état partagé)
- Tester les cas normaux, limites et erreurs
- Intégrer les tests dans la CI/CD pour automatisation
- Ajouter des tests à chaque nouvelle fonctionnalité ou bug corrigé

4.6 Types de tests

- **Unitaires** : testent une fonction isolée
- **Fonctionnels** : testent un enchaînement de fonctions ou une logique métier
- **Intégration** : testent la communication entre plusieurs composants (ex. : base de données + service)

4.7 Visualisation dans un IDE

Des IDE comme PyCharm ou VS Code (avec l'extension Python) permettent de visualiser et exécuter les tests directement depuis l'interface graphique.

5. Outils de Qualité

Les outils de qualité de code permettent de s'assurer que le code respecte les normes de style, qu'il est bien typé, bien formaté et sécurisé. Ils peuvent être utilisés en local, dans l'éditeur (ex : VS Code) ou intégrés dans des pipelines d'intégration continue.

Voici les principaux outils à utiliser dans vos projets Python :

5.1 black – formateur de code automatique

- Reformate automatiquement le code en suivant les conventions PEP8
- Pas de configuration nécessaire
- Uniformise les espaces, les sauts de ligne, les indentations

```
black src/
```

5.2 flake8 – vérification de style

- Analyse statique qui détecte les erreurs de style, de nommage ou d'espacement
- Peut être configuré via un fichier .flake8

```
flake8 src/
```

5.3 mypy – vérification des types

- Vérifie la cohérence entre les annotations de type (type hints) et le code
- Très utile avec des projets fortement typés

```
mypy src/
```

5.4 isort – tri automatique des imports

- Classe les imports en trois groupes (standard, tiers, internes)
- Corrige l'ordre et supprime les doublons

```
isort src/
```

5.5 bandit – analyse de sécurité

- Scanne le code pour détecter les vulnérabilités connues (usage de eval, accès fichiers, mots de passe hardcodés...)

```
bandit -r src/
```

Conseil : Tu peux regrouper tous ces outils dans un fichier pyproject.toml ou .pre-commit-config.yaml pour les exécuter automatiquement avant chaque commit.

Outil	Fonction	Commande
black	Formatage	black src/
flake8	Linting	flake8 src/
mypy	Typage	mypy src/
bandit	Sécurité	bandit -r src/
isort	Tri des imports	isort src/