Bonnes pratiques en Python

Pierre Jaumier

Bonnes pratiques en Python

Comprendre `if __name__ == "__main__"`

Le guide de style PEP 8

Documentation: Docstrings au format Google

name == “main”

```python
if __name__ == "__main__":
    main()
```

Une astuce essentielle pour structurer vos scripts Python

Le rôle des modules en Python

En Python : - Chaque fichier `.py` est un **module** - On peut **importer** des modules dans d’autres fichiers - Les modules peuvent contenir : - Des fonctions - Des classes - Des variables - Du code exécutable

La variable spéciale `__name__`

Tous les modules ont une variable interne appelée `__name__`.

Son contenu dépend de la façon dont on utilise le module :

Utilisation	Valeur de `__name__`
Fichier exécuté directement	`"__main__"`

Utilisation	Valeur de `__name__`
Fichier importé comme module	`"nom_du_module"`

Exemple concret

Fichier : `mon_module.py`

```
print("Module chargé")

def dire_bonjour():
    print("Bonjour !")

if __name__ == "__main__":
    print("Exécution directe")
    dire_bonjour()
```

Cas 1 – Exécution directe

```
print("Module chargé")

def dire_bonjour():
    print("Bonjour !")

if __name__ == "__main__":
    print("Exécution directe")
    dire_bonjour()
```

Commande :

```
python mon_module.py
```

Résultat :

```
Module chargé
Exécution directe
Bonjour !
```

Le bloc `if __name__ == "__main__":` est exécuté.

Cas 2 – Import du module

Fichier : `autre_fichier.py`

```
import mon_module

mon_module.dire_bonjour()
```

Commande :

```
python autre_fichier.py
```

Résultat :

```
Module chargé
Bonjour !
```

Le bloc `if __name__ == "__main__":` n'est **pas exécuté**.

À quoi ça sert ?

Utile pour : - Tester facilement un module sans tout exécuter - Créer des scripts réutilisables - Structurer clairement ton code - Définir un point d'entrée principal (comme dans d'autres langages)

Bonne pratique – Structure recommandée

```
def main():
    print("Programme principal")

if __name__ == "__main__":
    main()
```

Facilite les tests et la maintenance
Clairement identifiable comme point d'entrée

PEP 8 – Le guide de style Python

Écrire du code Python propre, lisible et professionnel

Qu'est-ce que PEP 8 ?

PEP 8 = *Python Enhancement Proposal* n°8
C'est le **guide officiel de style de codage** en Python.

Objectif :
> Rendre le code Python **cohérent**, **lisible** et **facile à maintenir** par tous les développeurs.

Site officiel : https://peps.python.org/pep-0008/

Pourquoi est-ce important ?

Code lu bien plus souvent qu'écrit.

Avantages : - Facilite la collaboration - Réduit les erreurs - Améliore la maintenance - Uniformise le style dans une équipe

En entreprise ou en open-source, respecter PEP 8 est essentiel !

Principales règles de style

Voici quelques grandes lignes directrices :

- **Indentation** : 4 espaces par niveau
- **Longueur des lignes** : maximum de caractères
- **Noms clairs et en minuscules** : `ma_variable`, `ma_fonction`
- **Classes** : `PascalCase` : `MajusculeCamelCase`
- **Constantes** : `MA_CONSTANTE`
- **Espaces autour des opérateurs** : `x = 1 + 2`
- **Un seul import par ligne** ou un nombre raisonnable

Exemple avant / après PEP 8

Avant (non conforme) :

```
def calcul(x):return x+1
```

Après (conforme) :

```
def calcul(x):
    return x + 1
```

Plus lisible, non ?

Bonnes pratiques supplémentaires

Quelques autres conseils : - Pas d'espace final en fin de ligne - Lignes vides pour séparer les fonctions et classes - Commentaires complets mais concis - Docstrings pour chaque module, classe et fonction publique - Pas de formatage du genre

```
a    = 1
long = 10
```

- Exemples de fonctions

```
def afficher_profil(nom, prenom, age=0, ville="inconnue"):
    ...

def envoyer_message(destinataire, message_texte):
    ...
```

Outils pour vérifier PEP 8

Automatise la vérification avec ces outils :

- `pylint` – analyse complète
- `flake8` – rapide et efficace
- `black` – reformateur automatique
- `isort` – trie les imports
- Extensions VSCode / PyCharm intégrant PEP 8

Gain de temps et qualité assurée !

Documenter ses fonctions et classes en Python

Google Style Docstrings

Un **docstring** est une chaîne de documentation intégrée dans le code :

```
def ma_fonction(param):
    """Ceci est un docstring - il explique ce que fait la fonction."""
    return param + 1
```

Il sert à : - Décrire ce que fait une fonction/méthode/classe - Expliquer les paramètres et valeurs retournées - Donner des exemples d'utilisation - Être lu par les développeurs et les outils (IDE, générateurs de docs…)

Pourquoi utiliser un format standard ?

Plusieurs formats existent : - **Google Style** - **NumPyDoc** - **Epytext** - **reST / Sphinx**

Le **format Google** est simple, clair et bien structuré. C'est l'un des plus répandus.

Structure du Google Style Docstring

Exemple :

```python
def diviser(a, b):
    """
    Divise deux nombres.

    Args:
        a (float): Numérateur
        b (float): Dénominateur

    Returns:
        float: Résultat de la division

    Raises:
        ZeroDivisionError: Si b est égal à zéro
    """
    if b == 0:
        raise ZeroDivisionError("division par zéro")
    return a / b
```

Sections principales

Les sections courantes :

Section	Description
`Args`	Liste des paramètres
`Returns`	Ce que retourne la fonction
`Raises`	Exceptions levées
`Example`	Exemple d'utilisation
`Note`	Remarque importante
`Warning`	Avertissement

Bien organisé = facile à lire

Avantages du Google Style

Pourquoi l'adopter ? - Lisibilité optimale - Supporté par les IDEs (VSCode, PyCharm…) - Compatible avec Sphinx via `sphinx.ext.napoleon` - Bonne transition vers une documentation officielle

Bonnes pratiques

Conseils : - Écris les docstrings dès la création de la fonction - Sois concis mais précis - Mets des exemples quand c'est utile - Utilise des types explicites (`str`, `int`, `list[str]`, etc.) - Met à jour le docstring si tu modifies la fonction