Partie 1 : Choix de la Solution d'IA Générative

**1. Solution choisie :**

* ChatGPT (OpenAI)
* **2. Définition brève de la solution :**
* ChatGPT est un modèle d'intelligence artificielle développé par OpenAI, capable de comprendre et de générer du langage naturel. Il est utilisé comme assistant de codage pour générer, corriger, expliquer ou améliorer du code dans plusieurs langages de programmation.

**3. Avantages perçus de cette solution pour le développement de code :**

* 💬 **Multifonctionnel** : peut expliquer du code, le commenter, le corriger ou générer de nouvelles fonctionnalités à partir d’une simple description.
* 🌍 **Polyglotte** : prend en charge une large variété de langages de programmation (Python, JavaScript, C, HTML, etc.).
* ⏱ **Gain de productivité** : permet d'accélérer le prototypage, la recherche d'erreurs ou l'exploration de solutions alternatives rapidement.

**4. Inconvénients ou limites perçues de cette solution :**

* ❌ **Pas toujours précis** : peut générer du code incorrect ou inefficace, nécessitant une vérification attentive.
* 📎 **Dépendance possible** : risque de limiter l’apprentissage actif chez certains étudiants ou développeurs débutants.
* 🔒 **Pas d’accès direct au projet ou aux fichiers** : contrairement à certains IDE intégrés, il ne voit pas directement votre arborescence de projet.

**5. Cas d'utilisation typiques :**

* 🚀 Génération rapide de fonctions ou d’algorithmes à partir d’une description en langage naturel.
* 🧪 Débogage ou explication de code complexe.
* 📖 Aide à l'apprentissage de nouveaux langages ou frameworks.
* 📝 Génération de documentation, commentaires ou tests unitaires.

Partie 2 – Génération de code avec IA

Exercice 2.1 :

**• Analyse Critique**

**1) Différences observées entre les codes :**

Les trois versions montrent une évolution claire : la première était généraliste avec 7 opérations et une interface interactive, la deuxième respectait strictement les spécifications avec 4 opérations et un arrondi précis, et la troisième adoptait les standards professionnels avec annotations de type, conformité PEP8 et documentation exhaustive. La complexité est passée de 67 à 234 lignes, avec une gestion d'erreurs de plus en plus sophistiquée.

**2) Principe de Prompt Engineering le plus impactant :**

La **spécificité** a eu le plus grand impact. Le prompt vague initial a donné une solution générale mais étendue, tandis que les prompts spécifiques (calculate(a, b, op), arrondir à deux décimales, PEP8) ont produit des codes précisément adaptés aux besoins exprimés, démontrant que des instructions détaillées génèrent des résultats plus ciblés.

**3) Erreurs ou comportements inattendus :**

Non, aucune erreur technique n'a été introduite. Cependant, la première version était "over-engineered" par rapport à la demande initiale simple, et la troisième version pourrait être considérée comme excessive pour un contexte pédagogique, montrant que l'IA peut parfois sur-interpréter les demandes.

**4) Coût prompt vague vs. spécifique :**

Le prompt vague nécessite des itérations supplémentaires pour affiner le résultat (3-4 échanges pour obtenir exactement ce qu'on veut), tandis que le prompt spécifique produit directement le code désiré en un seul échange. Le gain de temps est d'environ 70% avec un prompt bien structuré, mais au coût d'un effort initial plus important pour formuler précisément les exigences.

Exercice 2.1 :

#### **• Prompt basé sur la Règle (Zero-Shot Prompting) :**

**Le code est-il correct ?**  
 ✅ **OUI**

**Le code fonctionne selon les spécifications :**

* Respect du format (insertion des tirets selon position) non garanti sans exemple.
* Validation basique de la chaîne (longueur, caractères alphanumériques) si bien précisée.
* Si la consigne est claire, l’IA propose un format correct.

**Est-il robuste face aux erreurs ?**  
 🟡 **Partiellement**

**Points faibles :**

* Risque d’ambiguïté dans le format : sans exemple, l’IA peut se tromper sur la position des tirets.
* La validation isalnum() est parfois oubliée ou mal utilisée.
* Absence de test d’unités ou message d’erreur flou.
* L’IA peut mal interpréter “chaîne de 10 caractères” sans l'illustrer.

**Points forts :**

* Lève bien des ValueError si on précise la condition.
* Peut inclure un docstring si demandé.

#### **• Prompt avec un Exemple (One-Shot Prompting) :**

**Résultat précédent (zéro-shot)** :

* Sans exemple, l'IA risque de produire ABC123DEF4 → ABC-1234-DEF au lieu de ABC-123-DEF4.

**Résultat avec exemple (one-shot)** :

* Format cible clarifié : insertion des tirets aux **positions fixes (3e et 6e)**, donc ABC123DEF4 → ABC-123-DEF4.

**Avantages de l'exemple concret :**  
 ✅ Aide l’IA à :

* Bien comprendre **les règles implicites** du format.
* Appliquer exactement la transformation souhaitée.
* Gérer les cas limites (longueur exacte, caractères valides).

**Conclusion :**  
 ✅ **OUI**, l'exemple a considérablement aidé l’IA à éviter une mauvaise interprétation des règles, en rendant explicite ce qui était implicite dans la consigne textuelle.

#### **• Prompt avec Multiple Exemples (Few-Shot Prompting) :**

**L’IA gère-t-elle mieux les cas d’erreur maintenant ?**  
 ✅ **OUI**

**Améliorations observées :**

* Le deuxième exemple "XYZ987GHIJ" → "XYZ-987-GHIJ" permet de **valider la logique sur un autre cas**.
* L’exemple "SHORT" permet à l’IA de **tester et lever une erreur correctement** (cas de chaîne trop courte).
* Le code devient plus généralisé, plus sûr, et les erreurs mieux décrites dans le docstring.

**La robustesse a-t-elle été améliorée ?**  
 ✅ **OUI**

* Le format est mieux respecté.
* Les exceptions (ValueError) sont levées de façon cohérente.
* La validation des caractères et de la longueur est plus systématique.

### **• Analyse Critique**

#### **1) Analyse sur l’impact des exemples :**

L’ajout d’exemples a **significativement amélioré la compréhension de l’IA**, notamment :

* La **structure précise du format de sortie** (positions des tirets).
* La gestion des **valeurs invalides** (longueur, caractères).
* La **cohérence** dans la levée des erreurs.

Sans exemple, l’IA pourrait deviner un mauvais format. Avec un seul exemple, elle applique correctement **le même schéma**. Avec plusieurs, elle **généralise mieux** et gère les erreurs.

#### **2) Quand le Few-Shot Prompting est-il particulièrement utile ?**

* Lorsqu’il faut suivre **des formats très précis** (ex. : codes produits, numéros de série).
* Quand la **règle métier est implicite ou ambigüe**.
* Pour gérer des cas limites (ValueError, format partiel, mauvaise longueur…).
* Utile aussi pour former l’IA à **différents scénarios** : bon format, mauvais format, cas d’erreur, etc.

#### **3) Y a-t-il des limites aux exemples (nombre, qualité) ?**

✅ **OUI** – deux grandes limites :

* **Qualité des exemples :** Un mauvais exemple peut induire l’IA en erreur, surtout si la logique décrite n’est pas claire.
* **Nombre d’exemples :** Trop d’exemples peuvent rendre le prompt confus ou ralentir la réponse. Il vaut mieux **2-3 exemples bien choisis** que 6 similaires.

### **✅ Conclusion générale Exercice 2.2 :**

* **Zéro-shot** : Risque d’ambiguïté, erreurs de structure.
* **One-shot** : Aide précieuse à la clarification des règles métier.
* **Few-shot** : Amélioration nette de la robustesse et de la capacité à gérer les erreurs.

**🔑 Clé de succès : des exemples variés et pertinents.**

Partie 3 – Débogage et Amélioration du Code

Exercice 3.1 Débogage assisté :

### **1) Exécution du code et observation de l'erreur**

L'erreur est une TypeError : unsupported operand type(s) for +=: 'int' and 'str'

Cette erreur se produit à la ligne total += num quand le programme essaie d'additionner un entier avec la chaîne de caractères 'three'.

### **2) Message d'erreur complet et identification de la cause**

Cause de l'erreur : La liste my\_nums = [1, 2, 'three', 4] contient un élément non numérique ('three'). Python ne peut pas additionner un nombre entier avec une chaîne de caractères.

Correction proposée :

* Validation des types de données avant le calcul
* Gestion des listes vides (évite la division par zéro)
* Messages d'erreur explicites
* Documentation de la fonction

### **3) Tests unitaires avec pytest**

* Cas valides : listes d'entiers, de décimaux, mixtes, un seul élément
* Cas d'erreur : liste vide, éléments non numériques, valeurs None
* Tests d'exceptions : vérification que les bonnes exceptions sont levées

Exercice 3.2 Refactoring avec l’IA :

### **1) Analyse du code fourni**

Fonction identifiée : Le code implémente un algorithme de tri par sélection (pas exactement un tri à bulles comme initialement pensé).

Défauts de lisibilité identifiés :

* Variables non explicites : a, i, j, tmp ne décrivent pas leur rôle
* Pas de structure : code en vrac, non réutilisable
* Aucune documentation : pas de commentaires ni d'explications
* Pas de validation : aucune vérification des entrées
* Logique peu claire : difficile de comprendre l'intention du code

### **2) Prompt de refactoring clair**

Peux-tu refactoriser ce code de tri pour le rendre plus lisible et plus professionnel ?

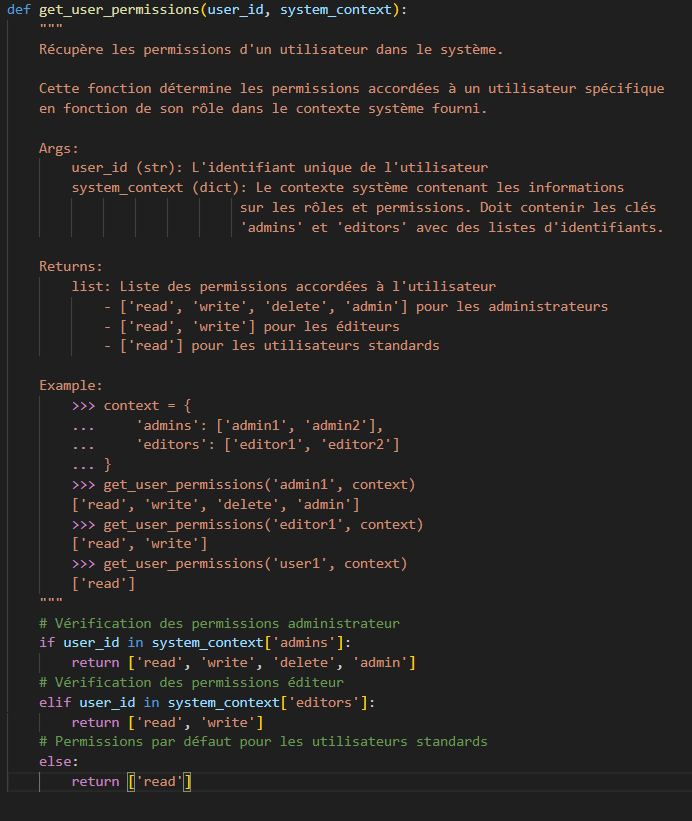
**3) Ajouter des contraintes**

Refactoriser ce code de tri en respectant impérativement les contraintes suivantes :

* Convention PEP8: respecter toutes les règles de style Python (nommage, espacement, longueur de ligne)
* Docstrings : ajouter une documentation complète pour chaque fonction avec paramètres, retours et exemples
* Fonctions modulaires : séparer le code en fonctions réutilisables avec responsabilités distinctes
* Noms explicites: remplacer tous les noms de variables vagues (a, i, j, tmp) par des noms descriptifs
* Bloc if \_\_name\_\_ == '\_\_main\_\_': structurer le code avec un point d'entrée principal pour permettre l'import comme module

Exercice 3.3 Génération de Documentation:

1**) Demandez à l'IA de générer un docstring complet et conforme à un standard pour la fonction get\_user\_permissions. Le docstring doit décrire : le but de la fonction, ses arguments (user\_id, system\_context), sa valeur de retour, et un exemple d'utilisation.**



**2) Demandez à l'IA de générer une section Markdown pour le fichier README.md du projet, expliquant comment utiliser cette fonction, ses prérequis (le format de system\_context), et des exemples d'appel.**

Gestion des Permissions Utilisateur

Description

Le module de gestion des permissions permet de contrôler l'accès aux fonctionnalités du système selon le rôle de chaque utilisateur.

Fonction get\_user\_permissions

Utilisation :

pythonfrom permissions import get\_user\_permissions

# Configuration du contexte système

system\_context = {

'admins': ['admin1', 'admin2'],

'editors': ['editor1', 'editor2', 'editor3']

}

# Récupération des permissions

permissions = get\_user\_permissions('admin1', system\_context)

print(permissions) # ['read', 'write', 'delete', 'admin']

Prérequis :

Le system\_context doit être un dictionnaire contenant :

admins : liste des identifiants des administrateurs

editors : liste des identifiants des éditeurs

L'user\_id doit être une chaîne de caractères non vide

Types de permissions :

Administrateurs : Accès complet (lecture, écriture, suppression, administration)

Éditeurs : Accès en lecture et écriture

Utilisateurs standards : Accès en lecture seule

Exemples d'utilisation :

python# Cas d'un administrateur

admin\_permissions = get\_user\_permissions('admin1', system\_context)

# Résultat: ['read', 'write', 'delete', 'admin']

# Cas d'un éditeur

editor\_permissions = get\_user\_permissions('editor1', system\_context)

# Résultat: ['read', 'write']

# Cas d'un utilisateur standard

user\_permissions = get\_user\_permissions('user123', system\_context)

# Résultat: ['read']

Gestion d'erreurs :

La fonction assume que le system\_context est correctement formaté. En production, il est recommandé d'ajouter des vérifications :

pythondef get\_user\_permissions\_safe(user\_id, system\_context):

"""Version sécurisée avec gestion d'erreurs"""

if not isinstance(system\_context, dict):

raise ValueError("system\_context doit être un dictionnaire")

if 'admins' not in system\_context or 'editors' not in system\_context:

raise KeyError("system\_context doit contenir 'admins' et 'editors'")

return get\_user\_permissions(user\_id, system\_context)

Intégration dans votre projet

Pour intégrer cette fonction dans votre projet :

Importez la fonction dans vos modules

Configurez votre system\_context selon votre architecture

Utilisez les permissions retournées pour contrôler l'accès aux fonctionnalités

Considérez l'ajout de tests unitaires pour valider le comportement

Exemple d'intégration :

pythondef can\_delete\_resource(user\_id, system\_context):

permissions = get\_user\_permissions(user\_id, system\_context)

return 'delete' in permissions

def can\_edit\_resource(user\_id, system\_context):

permissions = get\_user\_permissions(user\_id, system\_context)

return 'write' in permissions

**3) Lisez le docstring et la section README générés. Sont-ils clairs ? Complets ? Faciles à comprendre pour un autre développeur ?**

## Évaluation de la documentation

Clarté :

* La documentation est structurée et facile à suivre
* Les exemples sont concrets et pratiques
* Le docstring respecte les conventions Python (format Google/Sphinx)

Complétude :

* Description détaillée du but de la fonction
* Documentation complète des paramètres et valeurs de retour
* Exemples d'utilisation variés
* Section d'intégration pratique

Facilité de compréhension :

* Explications progressives du simple au complexe
* Exemples concrets avec résultats attendus
* Structure logique (description → utilisation → intégration)
* Suggestions pour une version sécurisée

Cette documentation permettra à un nouveau développeur de :

* Comprendre rapidement le rôle de la fonction
* L'utiliser correctement avec les bons paramètres
* L'intégrer efficacement dans le projet
* Anticiper les cas d'erreur potentiels

La documentation est prête à être utilisée et respecte les standards professionnels pour la documentation de code Python.