

# Enoncé de Projet Python — Gestion de Produits d'Épargne et Personnes

## Gestion de Produits d'Épargne et Personnes

Gestion de Produits d'Épargne et Personnes

Contexte

Objectifs

Spécifications Techniques

Architecture des dossiers

Classes à développer (dans `models/`)

Plan d'implémentation par étapes pour les fonctionnalités à coder

1. Création de l'architecture projet
2. Implémentation des classes `Personne` et `Epargne`
3. Création de la fonction de calcul des intérêts composés
4. Nettoyage des données
5. Fonctions d'import/export des données
6. Première simulation et suggestion d'épargne
7. Classe `ResultatEpargne`
8. Tests unitaires
9. Décorateurs et générateurs

## Contexte

Vous êtes chargé·e de développer une application Python modulaire qui permet de gérer des profils de personnes, différents produits d'épargne, ainsi que des simulations de placement en fonction des capacités d'épargne individuelles et des contraintes des produits.

Le projet doit être organisé de manière professionnelle en suivant une architecture modulaire claire.

---

# Objectifs

## 1. Modéliser les entités principales :

- Personne : caractéristiques financières et personnelles (revenu, dépenses, capacité d'épargne, etc.).
- Produit d'Épargne : nom, taux d'intérêt, fiscalité, durée minimale, plafond de versement.
- Résultats de simulation : proposition de plans d'épargne combinés répondant à un objectif financier.

## 2. Importer et exporter les données depuis/vers des fichiers CSV, TXT (séparateur tabulation) et Excel (.xlsx) pour les personnes et les produits d'épargne.

## 3. Simuler des plans d'épargne adaptés aux profils des personnes, en prenant en compte :

- Plusieurs scénarios de versements mensuels,
- Le respect des durées minimales des produits,
- Le respect des plafonds de versements,
- La possibilité de répartir les versements sur plusieurs produits.

## 4. Organiser le projet selon une structure modulaire professionnelle avec notamment :

- Un dossier `models/` contenant les classes métier,
- Des modules dédiés à la logique métier, aux utilitaires, et aux opérations d'import/export,
- Un dossier `tests/` pour les tests unitaires,
- Un fichier principal `main.py` pour exécuter l'application. /div

---

# Spécifications Techniques

## Architecture des dossiers

```

mon_projet/
|
|— pyproject.toml
|— README.md
|— src/
|   |— mon_module/
|       |— __init__.py
|       |— core.py
|       |— utils.py
|       |— models/
|           |— __init__.py
|           |— personne.py
|           |— epargne.py
|           |— resultat.py
|       |— data/          # (optionnel) fichiers exemples CSV/TXT/XLSX
|— main.py
|— personnes.csv
|— epargnes.csv
|— tests/
    |— test_core.py
    |— test_personne.py
    |— test_epargne.py
    |— test_resultat.py

```

## Classes à développer (dans `models/` )

- `Personne` : attributs, calcul capacité d'épargne.
- `Epargne` : attributs, méthodes d'affichage.
- `ResultatEpargne` (ou `Resultats` ) : structuration des résultats de simulation avec méthodes d'affichage et d'export (DataFrame).

## Plan d'implémentation par étapes pour les fonctionnalités à coder

## 1. Création de l'architecture projet

- Création des dossiers et fichiers suivants :

```
mon_projet/
├── src/
│   ├── mon_module/
│   │   ├── __init__.py
│   │   ├── core.py
│   │   ├── utils.py
│   │   └── models/
│   │       ├── __init__.py
│   │       ├── personne.py
│   │       └── epargne.py
│   ├── main.py
│   ├── README.md
│   └── tests/
```

- Initialisation des fichiers `__init__.py` pour faciliter les imports.

Remarque : les différences entre cette structure et la structure présentée en amont s'expliquent par les futurs développements à créer.

## 2. Implémentation des classes `Personne` et `Epargne`

### Objectifs :

- Modéliser précisément les entités `Personne` et `Epargne` avec leurs attributs et méthodes principales.
- Assurer une bonne encapsulation et gestion des types.

### Tâches :

- Dans `models/personne.py` :
  - Définir la classe `Personne` avec attributs : `nom`, `age`, `revenu_annuel`, `loyer`, `depenses_mensuelles`, `objectif`, `duree_epargne`, `versement_mensuel_utilisateur` (optionnel).

- Ajouter une méthode privée `_calcul_capacite_epargne()` calculant la capacité d'épargne mensuelle

$$capacité_{mensuelle} = (revenu_{annuel}/12) - loyer - dépenses_{mensuelle}$$

- Ajouter méthode `__str__` pour affichage humain.

Exemple :

```
class Personne:
    def __init__(self, #TODO):
        #TODO
        pass

    def _calcul_capacite_epargne(self) → float:
        #TODO
        pass

    def __str__(self):
        #TODO
        pass
```

- Dans `models/epargne.py` :
  - Définir la classe `Epargne` avec attributs : `nom`, `taux_interet`, `fiscalite`, `duree_min`, `versement_max` (optionnel).
  - Ajouter méthodes `__str__` et `__repr__`.
- Penser à gérer correctement les valeurs optionnelles (e.g. `versement_max` peut être `None`).

```
class Epargne:
    def __init__(self, #TODO):
        #TODO
        pass

    def __repr__(self):
```

```
#TODO
pass

def __str__(self):
    #TODO
    pass
```

### 3. Création de la fonction de calcul des intérêts composés

#### Objectif :

- Fournir une fonction réutilisable pour calculer le montant accumulé par un placement à intérêts composés.

#### Tâches :

- Dans `utils.py`, écrire une fonction `calcul_interets_composes(versement_annuel: float, taux_annuel: float, duree_annees: int) → float`.

$$montant_{n+1} = (montant_n + versement_{annuel}) * (1 + taux)$$

- Tester manuellement la fonction avec des exemples simples.

### 4. Nettoyage des données

#### Objectif :

- Assurer la robustesse des imports de données en traitant les valeurs manquantes ou mal formatées.

#### Tâches :

- Dans `core.py` (ou `utils.py`), prévoir les fonctions ou les blocs pour :
  - Traiter les valeurs NaN, None, et chaînes `"None"` dans les fichiers importés.
  - Convertir correctement les colonnes `taux_interet`, `fiscalite`, `versement_max` en float.
  - Convertir `duree_min` et `age` en int.
- Utiliser `pandas` pour faciliter ce nettoyage.

- Prévoir des exceptions claires en cas d'erreurs de format
- 

## 5. Fonctions d'import/export des données

### Objectifs :

- Implémenter l'import/export de fichiers multi-formats (CSV, TXT, XLSX) pour personnes et produits d'épargne.
- Intégrer le nettoyage de données lors de l'import.

### Tâches :

- Fonctions `import_personnes(fichier: str) → List[Personne]` et `import_epargnes(fichier: str) → List[Epargne]` dans `core.py`.
  - Fonctions `save_personnes(personnes: List[Personne], fichier: str)` et `save_epargnes(epargnes: List[Epargne], fichier: str)` dans `core.py`.
  - Gérer les différents formats de fichier (CSV, TXT, XLSX).
  - Valider que les données sont bieninstanciées en objets métiers.
  - Assurer un logging ou print en cas de succès ou d'erreur.
- 

## 6. Première simulation et suggestion d'épargne

### Objectifs :

- Proposer des plans d'épargne adaptés à une personne selon plusieurs scénarios de versement.
- Gérer les contraintes produits (durée minimale, plafond de versement).

### Tâche :

- Implémenter la fonction `suggestion_epargne(personne: Personne, epargnes: List[Epargne], objectif: float, duree: int) → List[ResultatEpargne]` dans `core.py` qui répondra à ces différentes demandes :

Pour chaque épargne, nous envisagerons 5 scénarios d'effort :

- L'effort saisi par l'utilisateur s'il l'a indiqué
- La 25% de la capacité d'épargne calculée au préalable

- 50% de cette capacité
- 75 % de la capacité
- 100% de la capacité

Nous ignorerons les produits inaccessibles selon la durée d'investissement, calculerons le capital brut avec intérêts (grâce à `calcul_interets_composes`) et nous n'oublierons pas d'appliquer l'imposition grâce à la fiscalite de l'épargne.

- Si le versement total respecte le plafond éventuel du produit (`versement_max`), on ajoute le scénario aux résultats.

---

## 7. Classe `ResultatEpargne`

### Objectifs :

- Structurer les résultats de simulation dans une classe dédiée.

### Tâches :

- Créer `resultat.py` dans `models/`.
- Définir la classe avec au moins : nom du produit, effort mensuel, montant net final, booléen atteinte objectif.
- Ajouter méthodes :
  - `afficher()` pour sortie console formatée.
  - `to_dataframe()` pour exporter vers DataFrame pandas.
- Penser à l'extensibilité (ex: ajout futur d'autres indicateurs).

---

## 8. Tests unitaires

### Objectifs :

- Assurer la fiabilité du code par des tests unitaires systématiques.

### Tâches :

- Créer les fichiers `test_personne.py`, `test_epargne.py`, `test_resultat.py`, `test_core.py` dans `tests/`.
- Coder des tests sur :



- Initialisation des classes,
- Calcul capacité d'épargne,
- Calcul intérêts composés,
- Import/export avec nettoyage,
- Simulation d'épargne.
- Utiliser un framework (ex: pytest).
- Documenter la procédure pour lancer les tests.

## 9. Décorateurs et générateurs

Objectifs :

- Clarté du processus avec un affichage plus compréhensible grâce aux décorateurs

Tâches :

- Créer un décorateur pour la fonction de suggestion d'épargne :
  - En entrée de fct : datetime + " Nous allons faire une comparaison de X placements selon la situation de YYYY"