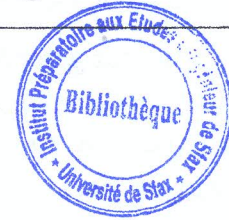


Année universitaire : 2025-2026

Examen semestre 1

Epreuve d'informatique



Classe : MP2, PC2 et PT2

Nombre de pages : 4

Date : Décembre 2025

Durée : 2h

Exercice 1

Le but de cet exercice est la manipulation de données environnementales en Python, en utilisant des dictionnaires et des listes pour analyser la conformité sur des échantillons d'eau en fonction de seuils réglementaires.

On suppose que l'on dispose :

- d'un dictionnaire `seuils` qui définit les valeurs limites acceptables pour certains paramètres (ex. : chlorures, conductivité, azote, etc.),
- d'une liste `echantillons` où chaque élément est un dictionnaire représentant un échantillon d'eau avec les valeurs mesurées.

Exemple de données de départ :

```
seuils = {  
    "Cl": 45,          # mg/L  
    "conductivite": 700, # µS/cm  
    "N": 30           # mg/L  
}  
  
echantillons = [  
    {"site": "Suisse", "Cl": 25, "conductivite": 854, "N": 20},  
    {"site": "Maroc", "Cl": 27, "conductivite": 774, "N": 22},  
    {"site": "Tunisie", "Cl": 289, "conductivite": 13200, "N": 35},  
    {"site": "Brésil", "Cl": 3976, "conductivite": 13530, "N": 143}, ]
```

1. Écrire une fonction `conformite(echantillon, seuils)` qui prend en entrée un dictionnaire représentant un échantillon et un dictionnaire `seuils` et renvoie un nouveau dictionnaire indiquant, pour chaque paramètre de l'échantillon, si la valeur est conforme (True) (c.à.d. inférieure ou égale à la valeur limite du seuil) ou non (False).

Exemple d'appel :

`conformite(echantillons[0], seuils)` # retourne : `{"Cl": True, "conductivite": False, "N": True}`

2. Écrire une fonction `sommaire(echantillons, seuils)` qui calcule, pour chaque paramètre, le pourcentage d'échantillons conformes.

Exemple d'appel :

`sommaire(echantillons, seuils)` # retourne : `{"Cl": 50.0, "conductivite": 0.0, "N": 50.0}`

3. Écrire une fonction `export_sommaire_txt(sommaire_dict, fich)` permettant d'enregistrer les résultats de `sommaire` : `sommaire_dict` dans un fichier `fich` en respectant l'exemple qui suit.

Exemple de contenu du fichier :

Paramètre : PourcentageConforme

Exercice 2

On souhaite modéliser un supermarché utilisant :

- une **pile** pour gérer le stock des articles (LIFO),
- plusieurs **files** pour gérer les clients aux différentes caisses (FIFO),
- une logique de simulation permettant de servir les clients jusqu'à épuisement des stocks ou vidage des files.

Les fonctions de base pour manipuler les files et les piles sont déjà programmées :

<i>Fonctions pour les files</i>	<i>Fonctions pour les piles</i>
<pre>def cree_file_vide(): return [] def est_vide_file(F): return F == [] def arrivee(F, c): F.append(c) def depart(F): if F == []: return None return F.pop(0) def longueur(F): n = 0 for _ in F: n += 1 return n def sommet_file(F): if F == []: return None return F[0] def afficher_file(F): for x in F: print(x, end=" ") print()</pre>	<pre>def cree_pile_vide(): return [] def empiler(P, x): P.append(x) def depiler(P): if P == []: return None return P.pop() def sommet(P): if P == []: return None return P[-1] def taille(P): n = 0 for _ in P: n += 1 return n def vider_pile(P): while P != []: P.pop()</pre>

Chaque caisse est représentée par une file. Un supermarché possède n caisses.

1. Écrire une fonction `creer_caisses(n)` qui renvoie une liste de n files vides.

Ex. pour $n = 3 \rightarrow [F0, F1, F2]$.

2. Écrire une fonction `indice_caisse_plus_courte(caisses)` qui renvoie l'indice de la caisse ayant le plus petit nombre de clients. En cas d'égalité, prendre la première. (*Utiliser uniquement longueur(F)*)

3. Écrire une fonction `ajouter_client(caisses, client)` qui ajoute un client dans la caisse la plus courte.

4. Écrire une fonction `servir_client_caisse(caisse, Stock)` qui :
 - retire un client de la caisse (si elle n'est pas vide), sinon affiche "caisse vide : aucun client"
 - dépile un article du Stock (si possible), sinon affiche "stock vide : impossible de servir"
 - affiche le service réalisé : "Client X servi avec article Y".
5. Écrire une fonction `tour_complet(caisses, Stock)` qui sert un client par caisse, dans l'ordre des caisses.

Exercice 3

Une entreprise de transport souhaite développer une application Python pour gérer son parc automobile (voitures, camions, motos...).

Chaque véhicule possède des caractéristiques communes (marque, modèle, année, kilométrage), mais certains types de véhicules ont aussi des propriétés spécifiques.

L'objectif est de concevoir un programme orienté objet permettant de :

- Représenter les véhicules sous forme d'objets.
- Calculer le coût d'entretien annuel selon le type et l'usage.
- Afficher les informations complètes du parc automobile.

Classe Vehicule

1. Créer la classe `Vehicule` avec un attribut de classe `compteur`, initialisé à 0, qui compte le nombre total d'objets créés (tous types confondus).

Implémenter les méthodes suivantes :

2. `__init__()` : constructeur initialisant les attributs suivants : `marque`, `modele`, `annee`, `__kilometrage` (attribut privé) et incrémentant l'attribut de classe `compteur`.
3. Créer une méthode getter `get_kilometrage()` qui retourne le kilométrage du véhicule.
4. Créer une méthode setter `set_kilometrage(val)` qui met à jour le kilométrage du véhicule avec la valeur `val`. Cette méthode doit refuser les valeurs négatives et afficher un message d'erreur via `try / except` sinon mettre à jour la valeur.

Exemple attendu pour une valeur négative :

```
v1.set_kilometrage(-100)
```

```
# Affiche "Erreur : le kilométrage ne peut pas être négatif."
```

5. Créer une méthode `afficher()` qui affiche les informations de base du véhicule.

Exemple attendu :

```
Toyota Yaris (2018) - 42000 km
```

6. Créer une méthode `nb_vehicules()` qui affiche le total.

Exemple attendu :

```
Vehicule.nb_vehicules() # "Nombre total de véhicules : 4"
```

Classes dérivées

Créer les classes dérivées suivantes :

8. Classe Voiture qui hérite de la classe Vehicule et qui possède un attribut supplémentaire :
nbportes.
9. Redéfinir la méthode cout_entretien() qui dépend du kilométrage :
Si kilometrage < 50000 → coût = 200 €
Sinon → coût = 350 €
10. Redéfinir la méthode afficher() pour inclure nbportes.
11. Classe Camion qui hérite de la classe Vehicule et qui possède un attribut supplémentaire :
capacite_tonnes.
12. Redéfinir la méthode cout_entretien() qui va dépendre de la capacité :
Si capacite_tonnes < 10 → coût = 500 €
Sinon → coût = 800 €
13. Redéfinir la méthode afficher() pour inclure capacite_tonnes.
14. Classe Moto héritant de Vehicule, avec un attribut cylindree.
15. Redéfinir la méthode cout_entretien() qui va dépendre cylindrée de la moto :
Coût d'entretien = 150 € si cylindrée < 600 cm³, sinon 250 €.
16. Redéfinir la méthode afficher() pour inclure cylindree.

Gestion du parc

17. Écrire une fonction afficher_parc(parc), où parc est une liste d'objets (de type Vehicule, Voiture, Camion ou Moto). Cette fonction doit parcourir la liste et appeler pour chaque objet : sa méthode afficher(), sa méthode cout_entretien().
18. Écrire une fonction sauvegarder(parc, fichier) qui permet de sauvegarder la liste des véhicules parc dans le fichier fichier.
Utiliser uniquement les outils standard (open, write, read).
Chaque ligne du fichier doit contenir :
type; marque; modele; annee; kilometrage; info_supplementaire
19. Créer une méthode charger(fichier) qui lit le fichier fichier et affiche les véhicules sauvegardés.