

sujet 1

Question 1 Déterminer si la liste obtenue par codage RLE est forcément de longueur inférieure ou égale à la liste de départ.

Non, la liste obtenue par codage RLE peut être plus longue. Par exemple, si une image n'a aucun pixel consécutif de la même couleur (ex: [0, 255, 0, 255]), chaque pixel apparaîtra une seule fois de suite. Le codage donnera alors : [1, 0, 1, 255, 1, 0, 1, 255], ce qui double la taille de la liste de départ.

Commentaire : Le codage RLE est un algorithme de compression qui n'est efficace que si les données contiennent de longues séquences de valeurs identiques. Dans le pire des cas, il fait "gonfler" la taille des données.

Question 2 En étudiant bien la fonction `codage_rle` qui réalise le codage, écrire le corps de la fonction `decodage_rle` qui réalise le décodage d'une liste. Des tests sont fournis dans la fonction `test_codage`, on pourra les compléter.

Voici le code à écrire pour remplacer le pass de la fonction `decodage_rle` :

```
def decodage_rle(liste_rle):  
    """Renvoie la liste d'octets obtenue à partir de la liste liste_rle obtenue  
    par compression RLE"""  
    liste_octets = []  
    # On parcourt la liste de 2 en 2  
    for i in range(0, len(liste_rle), 2):  
        occ = liste_rle[i]  
        valeur = liste_rle[i+1]  
        # On ajoute 'occ' fois la 'valeur' à notre liste finale  
        for j in range(occ):  
            liste_octets.append(valeur)  
  
    return liste_octets
```

Commentaire : La liste RLE alternant systématiquement "nombre d'occurrences" et "valeur", on utilise une boucle avec un pas de 2 (`range(0, len(liste_rle), 2)`) pour récupérer ces couples facilement et reconstruire la liste d'origine. On aurait aussi pu utiliser en Python l'opération sur les listes : `liste_octets.extend([valeur] * occ)`.

Question 3 Pour tester le codage sur une image, on peut utiliser la fonction fournie `encoder_decoder_image` qui effectue le codage puis le décodage d'une image pour l'enregistrer à nouveau dans un fichier. Utiliser cette fonction sur les images `bac_nsi_32.png` et `bac_nsi_256.png` et observer la différence de comportement.

On observe que pour `bac_nsi_32.png` l'image décodée est identique à l'originale. En revanche, pour `bac_nsi_256.png`, l'image obtenue est complètement altérée. Cela s'explique car la grande image possède de larges aplats de couleurs dépassant les 255 pixels identiques consécutifs. La fonction `codage_rle` génère donc un nombre d'occurrences supérieur à 255, mais la fonction `enregistrer_octets` bride (tronque) toutes les valeurs à 255 au maximum avant de les écrire dans le fichier. L'information sur le nombre de répétitions est donc irrémédiablement perdue et faussée lors de la relecture.

Commentaire : En informatique, il est crucial de tenir compte des contraintes matérielles ou de format. Ici, un "octet" (byte) ne peut coder des entiers que de 0 à 255 (soit 28 valeurs).

Question 4 Le problème précédent est lié au fait que sur des grandes images, il est possible d'avoir plus de 255 pixels de la même couleur. Proposer une démarche de résolution de ce problème qui modifie les fonctions d'encodage et de décodage, puis l'implémenter.

Démarche de résolution : Pour conserver le format de sauvegarde basé sur des octets (limité à 255), on peut simplement imposer une limite de 255 au compteur d'occurrences lors du codage. Si une séquence contient plus de 255 pixels identiques (par exemple 300 pixels noirs), on l'encode en deux blocs distincts (255 pixels noirs, puis 45 pixels noirs). L'avantage majeur de cette démarche est que **la fonction de décodage n'a pas besoin d'être modifiée**. Elle lira simplement les blocs les uns à la suite des autres.

Implémentation : Il suffit de modifier la condition de la boucle `while` interne de la fonction `codage_rle` :

```

def codage_rle(liste_octets):
    """Renvoie une liste d'octets obtenue par compression RLE (modifiée pour max 255)"""
    liste_rle = []
    i = 0
    while i < len(liste_octets):
        valeur = liste_octets[i]
        occ = 1
        # On ajoute la condition 'occ < 255' pour forcer l'arrêt du comptage
        while i + occ < len(liste_octets) and liste_octets[i + occ] == valeur and occ <
255:
            occ += 1
        liste_rle.append(occ)
        liste_rle.append(valeur)
        i += occ
    return liste_rle

```

Commentaire : Ajouter simplement `and occ < 255` suffit à régler le problème élégamment. Cela illustre bien le principe de modularité et de simplicité : au lieu de repenser tout le système (fichiers, encodage et décodage), on adapte légèrement l'encodage pour qu'il respecte les limites du format.

sujet 2

Question 1 Écrire le code de la fonction `salaire_moyen_condition` [...] Déterminer le salaire moyen des femmes et des hommes pour le jeu de données complet.

Voici le code à écrire dans la fonction `salaire_moyen_condition` :

```
def salaire_moyen_condition(employes, champ, valeur):  
    '''Renvoie le salaire moyen des employes ayant val comme valeur associée  
    au champ donné en argument.  
    Si le nombre d'employés considéré est nul, cette fonction renvoie None'''  
    somme = 0  
    effectif = 0  
  
    for emp in employes:  
        if emp[champ] == valeur:  
            somme += emp['salaire']  
            effectif += 1  
  
    if effectif == 0:  
        return None  
  
    return float(somme / effectif)
```

Pour déterminer les salaires moyens sur le jeu complet, il faut exécuter dans la console :

```
>>> salaire_moyen_condition(donnees_completes.employes, 'sexe', 'F')  
# Renvoie le salaire moyen des femmes du jeu complet  
>>> salaire_moyen_condition(donnees_completes.employes, 'sexe', 'M')  
# Renvoie le salaire moyen des hommes du jeu complet
```

Commentaire : Ce type d'algorithme utilise un schéma classique d'accumulation : on parcourt la liste d'enregistrements (dictionnaires) avec une boucle `for`, on filtre avec un `if`, et on met à jour un compteur (pour compter) et un accumulateur (pour sommer). Le test final `if effectif == 0` : évite l'erreur fatale de la division par zéro.

Question 2 Écrire en Python une fonction nommée `effectif_par_sexe` qui prend en paramètre un tableau non vide d'employés et qui renvoie un dictionnaire ayant deux clés 'F' et 'M' associées respectivement à l'effectif des femmes et à l'effectif des hommes employés.

Voici le code de la fonction :

```
def effectif_par_sexe(employes):  
    '''Renvoie un dictionnaire ayant deux clés 'F' et 'M'  
    associée respectivement au nombre d'employées femmes et au  
    nombre d'employés hommes dans les données en arguments.'''  
    effectifs = {'F': 0, 'M': 0}  
  
    for emp in employes:  
        # On incrémente la valeur associée à la clé correspondant au sexe de l'employé  
        effectifs[emp['sexe']] += 1  
  
    return effectifs
```

Commentaire : Au lieu d'utiliser plusieurs variables simples, on exploite directement les clés du dictionnaire pour stocker nos compteurs. C'est l'un des grands intérêts des dictionnaires : associer dynamiquement une étiquette (la clé) à une donnée (la valeur).

Question 3 Expliquer pourquoi le code de cette fonction est incorrect et proposer quelques tests simples sous forme d'assertions [...]. Proposer une version corrigée de la fonction `calcul_ecart_sexe` [...]

Explication des erreurs :

1. La fonction calcule l'écart de salaire en valeur absolue (en euros) avec `moy_h - moy_f`, alors que l'énoncé demande un pourcentage calculé selon la formule : $\text{moy}_h / \text{moy}_f \times 100$.
2. Dans l'appel `salaire_moyen_condition('employes', 'sexe', 'F')`, le premier argument est la chaîne de caractères 'employes' au lieu de la variable de liste `employes`.
3. La fonction ne gère pas le cas où l'un des salaires moyens vaut `None` (ce qui provoque une erreur lors de la soustraction).

Tests sous forme d'assertions :

```
def test_calcul_ecart_sexe():
    # Vérifier que le résultat est None si un seul sexe est présent
    assert calcul_ecart_sexe(['sexe': 'M', 'salaire': 2000, 'experience': 1, 'etudes': 1]) == None

    # Vérifier que l'écart est bien entre 0 et 100
    ecart = calcul_ecart_sexe(donnees.employes)
    if ecart is not None:
        assert 0 <= ecart <= 100
```

Version corrigée :

```
def calcul_ecart_sexe(employes):
    '''Renvoie l'écart de salaire en pourcentage pour les femmes
    par rapport aux hommes'''
    moy_h = salaire_moyen_condition(employes, 'sexe', 'M')
    moy_f = salaire_moyen_condition(employes, 'sexe', 'F')

    # On vérifie si un des sexes est absent
    if moy_h is None or moy_f is None:
        return None

    return ((moy_h - moy_f) / moy_h) * 100
```

Commentaire : Il est très important de vérifier que les types des variables passées en paramètres correspondent à ce que la fonction attend. Passer "employes" (type str) au lieu de `employes` (type list) est une erreur fréquente. La gestion de la valeur `None` est aussi une bonne pratique de programmation défensive.

Question 4 Tester et comparer les salaires proposés aux deux futurs employés [...]. Identifier la source des écarts entre les deux propositions de salaire dans le programme et la corriger.

Test et observation : En exécutant dans la console `salaire_par_proximite(donnees.employes, {'experience': 3, 'etudes': 3, 'sexe': 'F'})` puis la même chose avec 'M', on obtient des propositions de salaire différentes pour un profil pourtant identique en termes de compétences (études et expérience).

Identification du problème : Le problème vient de la fonction `distance(e1, e2)` qui inclut le calcul de la différence de sexe : $s = s + (\text{sexe_vers_entier}(e1) - \text{sexe_vers_entier}(e2))^{**2}$. À cause de cela, un candidat homme sera considéré "plus proche" des autres employés hommes du jeu de données, et une femme plus proche des autres femmes. L'algorithme des k-plus proches voisins va donc reproduire les inégalités de salaires déjà existantes dans l'entreprise, en calculant la moyenne sur les voisins du même sexe !

Correction : Pour corriger ce biais algorithmique, il faut retirer le critère discriminatoire (le sexe) du calcul de la distance. Le salaire doit être proposé uniquement en fonction de l'expérience et des études.

```
def distance(e1, e2):
    '''Renvoie la mesure de distance entre deux personnes,
    sans tenir compte du sexe pour éviter tout biais de genre.'''
    s = 0
    s = s + (e1['experience'] - e2['experience'])**2
    s = s + (e1['etudes'] - e2['etudes'])**2
    return sqrt(s)
```

Commentaire : Cet exercice illustre un enjeu majeur de l'Intelligence Artificielle et de l'apprentissage automatique (Machine Learning) : si les données d'entraînement contiennent des biais (ici sexistes), l'algorithme des k-plus proches voisins va les reproduire. Retirer la variable sensible est l'une des méthodes pour construire des algorithmes plus éthiques.

sujet 3

Question 1 Écrire en Python une fonction nommée `est_bissextile` qui prend en paramètre un entier correspondant à une année et qui renvoie un booléen indiquant si elle est bissextile, en appliquant la règle donnée ci-dessus.

Voici le code attendu pour la fonction :

```
def est_bissextile(annee):  
    if annee % 400 == 0:  
        return True  
    elif annee % 100 == 0:  
        return False  
    elif annee % 4 == 0:  
        return True  
    else:  
        return False
```

Note : on peut aussi l'écrire en une seule ligne : `return (annee % 4 == 0 and annee % 100 != 0) or (annee % 400 == 0)`

Commentaire : On utilise l'opérateur modulo % qui renvoie le reste de la division euclidienne pour tester la divisibilité. L'ordre des conditions est important ici pour traiter correctement les exceptions (les multiples de 400 doivent être testés avant ceux de 100).

Question 2 Écrire en Python une fonction nommée `determiner_phase` qui prend en paramètre un entier, compris entre 1 et 28 inclus, qui correspond au jour d'un cycle et qui renvoie un entier correspondant au numéro de la phase associée. À l'aide d'une assertion, on garantira que l'entier donné en argument est compris entre 1 et 28 inclus.

Voici le code attendu :

```
def determiner_phase(jour):  
    assert 1 <= jour <= 28, "Le jour donné doit être compris entre 1 et 28 inclus."  
  
    if 1 <= jour <= 5:  
        return 1  
    elif 6 <= jour <= 13:  
        return 2  
    elif jour == 14:  
        return 3  
    else:  
        return 4
```

Commentaire : L'instruction `assert t` est essentielle en programmation défensive (au programme de NSI). Elle sert de précondition : si la condition est fausse, le programme s'arrête avec une erreur, ce qui évite de propager des valeurs absurdes dans la suite des calculs.

Question 3 La fonction `ajouter_jours`, dont le code est déjà fourni, prend en paramètres une date et un entier représentant un nombre de jours. Elle renvoie la nouvelle date obtenue après ajout de ces jours. Compléter la fonction `test_ajouter_jours` en ajoutant au moins trois autres tests pertinents. Pour chaque test ajouté, une brève justification doit être donnée afin d'expliquer pourquoi ce cas est important à vérifier.

Voici comment compléter la fonction de tests :

```
def test_ajouter_jours():
    assert ajouter_jours((7, 9, 2025), 3) == (10, 9, 2025)

    # Test 1 : Vérifier le passage au mois suivant
    assert ajouter_jours((25, 9, 2025), 10) == (5, 10, 2025)

    # Test 2 : Vérifier le passage à l'année suivante (changement de mois de décembre à janvier)
    assert ajouter_jours((20, 12, 2025), 20) == (9, 1, 2026)

    # Test 3 : Vérifier le traitement correct du mois de février lors d'une année bissextile
    assert ajouter_jours((25, 2, 2024), 10) == (6, 3, 2024)
```

Commentaire : Concevoir des tests (assertions) pertinents implique d'identifier les cas limites ou complexes (les "effets de bord"). Tester une simple addition de jours dans le même mois ne suffit pas à valider que l'algorithme gère correctement les changements de mois, d'années ou les années bissextiles.

Question 4 Observer avec la fonction `test_calendrier_cycles` les que le calendrier renvoyé par la fonction `calendrier_cycles` n'est pas dans un format valide. Identifier le problème dans la fonction `calendrier_cycles`, proposer une démarche de résolution et la mettre en œuvre.

Identification du problème : Le format DTSTART d'iCalendar requiert que la date fasse exactement 8 caractères (AAAAMMJJ). Or, dans le code d'origine `date = str(annee)+str(mois)+str(jour)`, si le mois est juillet (7), `str(mois)` donne "7" et non "07". Ainsi, le 3 juillet 2026 donnera "202673" au lieu de "20260703".

Démarche de résolution et mise en œuvre : Il faut formater les variables `mois` et `jour` pour qu'elles occupent toujours 2 caractères, en ajoutant un zéro devant si elles sont strictement inférieures à 10.

sujet 4

Question 1 Écrire une fonction `croissance_moyenne(plantes)` qui prend en paramètre une liste d'instances de la classe `Plante` et renvoie la moyenne des durées de croissance de l'ensemble de ces plantes (en jours). Si la liste fournie est vide, la fonction doit renvoyer `None`. Écrire au moins deux tests pour valider le bon fonctionnement de cette fonction, dont une traitant le cas d'une liste vide.

Voici le code attendu :

```
def croissance_moyenne(plantes):
    if len(plantes) == 0:
        return None

    somme = 0
    for plante in plantes:
        somme += plante.croissance

    return somme / len(plantes)
```

Tests

```
assert croissance_moyenne([]) is None
```

Test avec une liste fictive de deux plantes (en supposant la classe Plante importée)

```
p1 = Plante("A", "EspA", 10, 10, "ombre")
```

```
p2 = Plante("B", "EspB", 20, 10, "ombre")
```

```
assert croissance_moyenne([p1, p2]) == 15.0
```

Commentaire : Il est impératif de traiter le cas de la liste vide en premier avec un `if` pour éviter une erreur de division par zéro lors du calcul de la moyenne. Pour accéder à la durée de croissance, on utilise la notation pointée `plante.croissance` car on manipule des objets (instances de classe) et non des dictionnaires.

Question 2 Écrire une fonction `dictionnaire_mesure(plantes, mesures)` qui prend en paramètre la liste des plantes et la liste des mesures. Elle doit renvoyer un dictionnaire où chaque clé est le nom d'une plante (présente dans la liste `plantes`), et chaque valeur associée est la liste des mesures concernant cette plante spécifique. Si une plante de la liste ne possède aucune mesure associée, la liste correspondante dans le dictionnaire devra être vide. Concevoir une série de tests pertinente pour vérifier le bon comportement de cette fonction.

Voici le code attendu :

```
def dictionnaire_mesure(plantes, mesures):
    dico = {}

    # 1. On initialise le dictionnaire avec une liste vide pour chaque plante
    for p in plantes:
        dico[p.nom] = []

    # 2. On parcourt les mesures et on les ajoute à la bonne liste
    for m in mesures:
        nom_plante = m['plante']
        if nom_plante in dico:
            dico[nom_plante].append(m)

    return dico
```

Tests

```
p1 = Plante("Fougère", "Esp1", 10, 10, "ombre")
```

```
p2 = Plante("Rose", "Esp2", 10, 10, "soleil")
```

```
liste_mesures = [{'plante': 'Fougère', 'jour': 1, 'hauteur': 5}]
```

```
resultat = dictionnaire_mesure([p1, p2], liste_mesures)
```

```
assert resultat["Fougère"] == [{'plante': 'Fougère', 'jour': 1, 'hauteur': 5}]
```

```
assert resultat["Rose"] == [] # Vérifie qu'une plante sans mesure a bien une liste vide
```

Commentaire : L'astuce ici consiste à faire deux passages. D'abord, on prépare le dictionnaire avec toutes les clés possibles (les noms des plantes) associées à des listes vides. Ensuite, on le remplit. Cela garantit que même les plantes sans mesures figureront dans le dictionnaire final avec une liste vide, comme demandé.

Question 3 Exécuter le test de la fonction `test_purger` et analyser le code de la fonction `purger_mesures_extremes` pour identifier la source de cette erreur logique.

Identification du problème : L'erreur logique vient du fait qu'on supprime des éléments d'une liste (avec `liste_mesures.remove(mesure)`) pendant qu'on la parcourt avec une boucle `for`. Lorsqu'un élément est supprimé, tous les éléments suivants se décalent d'une case vers la gauche. Cependant, la boucle `for` avance à l'indice suivant, ce qui a pour effet de "sauter" l'élément qui vient de se décaler. Certains éléments extrêmes ne sont donc jamais testés.

Commentaire : Règle d'or en algorithmique : il ne faut jamais modifier la taille d'une structure de données (comme ajouter ou supprimer des éléments) pendant qu'on est en train de la parcourir élément par élément.

Question 4 Proposer une version corrigée de la fonction `purger_mesures_extremes` répondant parfaitement à l'objectif.

Voici deux méthodes possibles pour corriger cette fonction :

Méthode 1 : Parcours à l'envers (modification en place)

```
def purger_mesures_extremes(liste_mesures):  
    # On parcourt la liste à l'envers  
    for i in range(len(liste_mesures) - 1, -1, -1):  
        temp = liste_mesures[i]['temperature']  
        if temp < 20 or temp > 25:  
            liste_mesures.pop(i)
```

Méthode 2 : Reconstruction d'une nouvelle liste (plus "Pythonique")

```
def purger_mesures_extremes(liste_mesures):  
    # On reconstruit le contenu de la liste initiale avec uniquement les bonnes valeurs  
    mesures_valides = [m for m in liste_mesures if 20 <= m['temperature'] <= 25]  
  
    # On vide la liste d'origine et on y place les éléments valides  
    liste_mesures.clear()  
    liste_mesures.extend(mesures_valides)
```

Commentaire : Parcourir la liste à l'envers (en partant de la fin) résout le problème du décalage : supprimer un élément à la fin ne décale que les éléments situés après lui, que l'on a donc déjà visités. La méthode 2 (en compréhension) est souvent préférée en Python car elle est plus lisible, tout en respectant la contrainte de modifier la liste d'origine.

sujet 5

Question 1 Compléter le corps de la fonction `total_simple`. Ajouter un test permettant d'afficher l'empreinte carbone totale d'Ada, en utilisant les fonctions fournies pour accéder au fichier `empreinte_ada_agr.json`.

Voici le code à écrire pour la fonction et son test :

```
def total_simple(empreinte):  
    """Fonction qui renvoie l'empreinte carbone totale d'un dictionnaire associant  
    une empreinte carbone à des noms de catégories"""  
    somme = 0  
    for valeur in empreinte.values():  
        somme += valeur  
    return somme  
  
# Test pour afficher l'empreinte totale  
dictionnaire_ada_agr = chargement_json("empreinte_ada_agr.json")  
print("Empreinte totale simple :", total_simple(dictionnaire_ada_agr))
```

Commentaire : On utilise la méthode `.values()` des dictionnaires pour parcourir directement les valeurs numériques associées aux clés (les catégories), afin d'utiliser le schéma classique de cumul dans une variable `somme`.

Question 2 En utilisant la fonction `est_dictionnaire` qui teste la nature d'une valeur, écrire le corps de la fonction récursive `total_rec` qui calcule la somme des valeurs numériques présentes dans des dictionnaires imbriqués.

Voici le code attendu :

```
def total_rec(empreinte):  
    """Fonction récursive qui renvoie l'empreinte carbone totale représentée  
    par un dictionnaire dont les valeurs peuvent aussi être des dictionnaires"""  
    somme = 0  
    for valeur in empreinte.values():  
        if est_dictionnaire(valeur):  
            # Cas récursif : la valeur est un sous-dictionnaire  
            somme += total_rec(valeur)  
        else:  
            # Cas de base : la valeur est un nombre  
            somme += valeur  
    return somme
```

Commentaire : L'algorithme est récursif car on parcourt une structure de données arborescente (des dictionnaires dans des dictionnaires). Chaque fois que l'on rencontre un sous-dictionnaire, on s'appelle soi-même pour en calculer la somme, que l'on ajoute à la somme courante.

Question 3 Lorsqu'on exécute la fonction `alerte_valeur_aberrante` sur le dictionnaire complet d'Ada avec une limite fixée à 1000, elle ne détecte aucune valeur aberrante [...]. Expliquer précisément l'origine de cette erreur de conception et proposer une version corrigée de la fonction.

Origine de l'erreur : Dans le code d'origine, lorsqu'on rencontre une valeur qui est un dictionnaire, l'instruction `return` `alerte_valeur_aberrante(valeur, limite)` est exécutée de manière inconditionnelle. Le mot-clé `return` stoppe immédiatement l'exécution de la fonction en cours et interrompt la boucle `for`. Ainsi, dès le premier sous-dictionnaire rencontré, s'il ne contient pas de valeur aberrante, la fonction renvoie `False` et s'arrête sans jamais explorer les autres catégories (comme le "Logement" où se trouve le chauffage).

Version corrigée :

```
def alerte_valeur_aberrante(empreinte, limite):
    for categorie, valeur in empreinte.items():
        if est_dictionnaire(valeur):
            # Si on trouve une valeur aberrante dans le sous-dictionnaire, on renvoie
            True

            if alerte_valeur_aberrante(valeur, limite):
                return True
        else:
            # Si la valeur numérique dépasse la limite, on renvoie True
            if valeur > limite:
                return True

    # Si on a parcouru tout le dictionnaire sans rien trouver
    return False
```

Commentaire : Une règle essentielle avec le `return` : il met fin à la fonction. Dans une recherche, on ne doit renvoyer une valeur que si l'on a trouvé ce que l'on cherche (`True`), ou si l'on a fini de chercher partout sans succès (`False` à la fin, en dehors de la boucle).

Question 4 Afin de prévenir toute régression future sur la fonction `alerte_valeur_aberrante` corrigée, il convient de définir une stratégie de validation robuste. Proposer un jeu de tests pertinent pour cette fonction.

Voici une proposition de jeu de tests (utilisant `assert`) :

```
def tests_alerte():
    # Test 1 : Limite non dépassée, structure plate
    d1 = {"a": 100, "b": 200}
    assert alerte_valeur_aberrante(d1, 500) == False
    # Justification : Vérifie le comportement normal de base sans dépassement ni
    récursivité.

    # Test 2 : Limite dépassée, structure plate
    d2 = {"a": 600, "b": 200}
    assert alerte_valeur_aberrante(d2, 500) == True
    # Justification : Vérifie que la détection de base fonctionne sur les nombres
    simples.

    # Test 3 : Limite dépassée cachée dans un sous-dictionnaire profond
    d3 = {"a": 100, "b": {"c": 200, "d": {"e": 800}}}
    assert alerte_valeur_aberrante(d3, 500) == True
    # Justification : Valide la récursivité, le cas dépassement enfoui dans la structure.

    # Test 4 : Limite dépassée sur une clé située APRÈS un sous-dictionnaire ne dépassant
    pas
    d4 = {"a": {"b": 100}, "c": 600}
    assert alerte_valeur_aberrante(d4, 500) == True
    # Justification : Ce test vérifie spécifiquement que le bug de la question 3 a bien
    été corrigé et que la boucle ne s'arrête pas prématurément.
```

Commentaire : Un bon jeu de tests couvre les cas "normaux" (plate, dépassement ou non) et les cas "limites" liés à la structure de données (imbrications). Le Test 4 est fondamental car il permet de valider la correction apportée à la question 3.

sujet 6

Question 1 Écrire en Python une fonction nommée `smoothie_possible` qui prend en paramètre une chaîne de caractères représentant le nom d'un smoothie et qui renvoie un booléen qui indique s'il est possible de réaliser ce smoothie.

Voici le code attendu à insérer dans la classe `Boutique_smoothie` :

```
def smoothie_possible(self, nom_smoothie):
    if nom_smoothie not in self.db_smoothies:
        return False

    for fruit in self.db_smoothies[nom_smoothie]:
        if fruit not in self.liste_fruits_disponibles:
            return False
    return True
```

Commentaire : On parcourt les ingrédients nécessaires à la recette. Si un seul fruit manque à l'appel dans la liste des fruits disponibles, la fonction s'interrompt et renvoie `False`. Si la boucle se termine sans encombre, tous les fruits sont là, on renvoie `True`.

Question 2 Écrire en Python une fonction nommée `liste_smoothies_possibles` qui ne prend pas de paramètre et qui renvoie la liste des recettes possibles.

Voici le code attendu :

```
def liste_smoothies_possibles(self):
    possibles = []
    for nom_smoothie in self.db_smoothies:
        if self.smoothie_possible(nom_smoothie):
            possibles.append(nom_smoothie)
    return possibles
```

Commentaire : Au lieu de réécrire toute la logique, on réutilise intelligemment la méthode `smoothie_possible` développée à la question 1. C'est le principe de modularité en programmation. On itère sur les clés du dictionnaire `db_smoothies` pour construire notre liste.

Question 3 Pour tester la fonction `score_proximité`, compléter la fonction `test_score_proximité` avec des tests pertinents.

Voici le code attendu pour compléter la fonction de test :

```
def test_score_proximité():
    # L'inventaire des fruits disponibles importe peu pour ce calcul
    boutique = Boutique_smoothie([])

    # Test 1 : Deux recettes identiques (3 fruits en commun)
    assert boutique.score_proximité("Tropical", "Tropical") == 3

    # Test 2 : Deux recettes avec 1 seul fruit en commun (la Mangue)
    assert boutique.score_proximité("Tropical", "Soleil couchant") == 1

    # Test 3 : Deux recettes avec aucun fruit en commun
    assert boutique.score_proximité("Tropical", "Rouge") == 0
```

Commentaire : Un bon jeu de test doit couvrir plusieurs scénarios pour s'assurer de la robustesse d'une fonction : le cas idéal (score maximum), un cas intermédiaire et un cas extrême (score nul). L'utilisation de `assert` permet de rendre le test silencieux s'il passe, et déclenche une erreur s'il échoue.

Question 4 Observer avec la fonction `test_plus_proche_possible` que le smoothie renvoyé n'est pas le bon. Identifier le problème dans la fonction `plus_proche_possible`. Proposer une démarche de résolution et la mettre en œuvre.

Identification du problème : Dans le code d'origine de `plus_proche_possible`, la boucle `for` parcourt `self.db_smoothies`, c'est-à-dire l'ensemble des smoothies **existants** dans le dictionnaire, sans vérifier s'ils sont **réalisables** avec les fruits actuellement en stock. De plus, elle risque de proposer le smoothie de référence lui-même si on ne l'exclut pas.

Démarche de résolution et mise en œuvre : Il faut remplacer le parcours de `self.db_smoothies` par le parcours de `self.liste_smoothies_possibles()`, et s'assurer qu'on ne compare pas la recette avec elle-même.

Voici la version corrigée à remplacer dans la classe :

```
def plus_proche_possible(self, nom_smoothie_ref):
    max_communs = 0
    smoothie_proche = None

    # On itère uniquement sur les smoothies réalisables en boutique
    for nom_smoothie in self.liste_smoothies_possibles():
        if nom_smoothie != nom_smoothie_ref:
            nb_communs = self.score_proximité(nom_smoothie_ref, nom_smoothie)
            if nb_communs > max_communs:
                max_communs = nb_communs
                smoothie_proche = nom_smoothie

    return smoothie_proche
```

Commentaire : Ce bug illustre l'importance de bien lire les contraintes d'un énoncé ("parmi les smoothies possibles"). Appeler la méthode `liste_smoothies_possibles()` restreint directement l'espace de recherche aux solutions valides.

Question 5 Créer une boutique avec les ingrédients suivants : Mangue, Ananas, Banane, Fraise, Citron, Kiwi et Pomme verte et afficher les smoothies réalisables.

Voici le code à insérer à la fin du fichier Python (après les appels de tests) :

```
# Question 5
fruits_du_jour = ["Mangue", "Ananas", "Banane", "Fraise", "Citron", "Kiwi", "Pomme verte"]
ma_boutique = Boutique_smoothie(fruits_du_jour)
ma_boutique.affichage_possibles()
```

Commentaire : Il s'agit simplement ici d'instancier un objet à partir de notre classe `Boutique_smoothie` en lui donnant en argument la liste demandée pour le constructeur (`__init__`), puis d'appeler la méthode d'affichage qui fera appel à toutes les méthodes que nous avons programmées.

sujet 7

Question 1 Créer une population initiale contenant 3 coccinelles (2 femelles et 1 mâle), toutes âgées de 10 jours et ayant un niveau de nutrition de 2. Le nombre initial de pucerons est fixé à 200. Écrire une séquence d'instructions (utilisant une boucle) permettant de simuler l'évolution de ce petit écosystème sur 5 jours consécutifs, en appelant la fonction `evolution`. Afficher le nombre de coccinelles et de pucerons à la fin de chaque journée.

Voici le code attendu à la fin du fichier :

```
# Création de la population initiale
population_q1 = [
    Coccinelle("femelle", 10, 2),
    Coccinelle("femelle", 10, 2),
    Coccinelle("male", 10, 2)
]
pucerons_q1 = 200

# Boucle de simulation sur 5 jours
for jour in range(1, 6):
    population_q1, pucerons_q1 = evolution(population_q1, pucerons_q1)
    print(f"Fin du jour {jour} : {len(population_q1)} coccinelles, {pucerons_q1} pucerons.")
```

Commentaire : La fonction `evolution` renvoie deux éléments (un tuple). Il est donc indispensable d'utiliser une affectation multiple `population_q1, pucerons_q1 = ...` pour mettre à jour nos variables à chaque passage dans la boucle.

Question 2 Écrire une fonction `simulation_simple(population, nb_proies)` qui automatise ce processus sur une durée maximale de 30 jours. Cette fonction doit s'interrompre prématurément si la population de coccinelles ou de pucerons tombe à zéro. Elle doit renvoyer un triplet (tuple) contenant : le nombre final de coccinelles, le nombre final de pucerons, et le nombre de jours effectivement simulés.

Voici le code attendu :

```
def simulation_simple(population, nb_proies):
    jours_simules = 0
    # On continue tant qu'on n'a pas atteint 30 jours ET qu'aucune des populations n'est à 0
    while jours_simules < 30 and len(population) > 0 and nb_proies > 0:
        population, nb_proies = evolution(population, nb_proies)
        jours_simules += 1

    return (len(population), nb_proies, jours_simules)

# Test de la fonction
pop_init = [
    Coccinelle("femelle", 10, 2),
    Coccinelle("femelle", 10, 2),
    Coccinelle("male", 10, 2)
]
print("Résultat de la simulation :", simulation_simple(pop_init, 1000))
```

Commentaire : Une boucle `while` est ici bien plus adaptée qu'une boucle `for` car le nombre d'itérations n'est pas connu à l'avance et dépend de plusieurs conditions d'arrêt simultanées liées à l'état de l'écosystème.

Question 3 Écrire la documentation et les commentaires de la méthode `chasser`.

Voici une proposition de documentation (docstring) et de commentaires à intégrer dans la méthode de la classe `Coccinelle` :

```
def chasser(self, nb_proies, nb_coccinelles):  
    """  
    Détermine la quantité de pucerons mangés par cette coccinelle en fonction  
    de la disponibilité de la nourriture.  
    Met à jour le niveau de nutrition de la coccinelle et renvoie le nombre  
    de pucerons restants.  
  
    :param nb_proies: (int) Nombre total de pucerons disponibles.  
    :param nb_coccinelles: (int) Nombre total de coccinelles en compétition.  
    :return: (int) Nombre de pucerons restants après le repas.  
    """  
    # S'il n'y a plus de coccinelles, le nombre de proies reste intact  
    if nb_coccinelles == 0:  
        return nb_proies  
  
    # Calcul du ratio de proies disponibles par coccinelle  
    proies_par_cocci = nb_proies / nb_coccinelles  
  
    # La coccinelle mange plus ou moins selon l'abondance des proies  
    if proies_par_cocci > 20:  
        consomme = random.randint(12, 20)  
    elif proies_par_cocci > 10:  
        consomme = random.randint(8, 15)  
    else:  
        consomme = random.randint(3, 8)  
  
    # On s'assure que la coccinelle ne mange pas plus de proies qu'il n'y en a  
    consomme = min(consomme, nb_proies)  
  
    # Mise à jour du niveau de nutrition selon la quantité ingérée  
    if consomme >= 10:  
        self.niv_nutrition += 1  
    else:  
        # Le niveau de nutrition baisse, avec un minimum bloqué à 0  
        self.niv_nutrition = max(0, self.niv_nutrition - 1)  
  
    return nb_proies - consomme
```

Commentaire : Une bonne documentation explique ce *que fait* la méthode, précise le *rôle des paramètres* et ce qu'elle *renvoie*. Les commentaires dans le code aident à comprendre la logique des étapes clés.

Question 4 Modifier les méthodes *reproduction* et *a_survecu* de la classe *Coccinelle* afin d'y intégrer ces deux nouvelles règles biologiques (la maturité sexuelle à 20 jours et l'impact mortel du manque de nourriture à 33%). Tester à nouveau la simulation globale.

Voici les méthodes corrigées à remplacer dans la classe *Coccinelle* :

```
def reproduction(self):
    descendants = []
    # On ajoute la condition sur l'âge (>= 20) pour la maturité sexuelle
    if self.sexe == "femelle" and self.niv_nutrition >= 2 and self.age >= 20:
        descendants.append(Coccinelle("male", 0, 0))
        descendants.append(Coccinelle("femelle", 0, 0))
        self.niv_nutrition = 0

    return descendants

def a_survecu(self):
    self.age = self.age + 1

    # Nouvelle règle de mortalité si le niveau de nutrition est à 0
    if self.niv_nutrition == 0:
        # random.random() génère un flottant entre 0.0 et 1.0.
        # Il y a 1 chance sur 3 (0.33) que ce nombre soit inférieur à 1/3
        if random.random() < (1 / 3):
            return False

    # Condition de mortalité naturelle liée à l'âge
    return self.age < self.espérance_de_vie
```

Test à rajouter en fin de fichier pour valider le changement :

```
pop_init_q4 = [
    Coccinelle("femelle", 10, 2),
    Coccinelle("femelle", 10, 2),
    Coccinelle("male", 10, 2)
]
print("Nouvelle simulation (avec limites biologiques) :", simulation_simple(pop_init_q4, 1000))
```

Commentaire : L'utilisation de `random.random() < probabilité` est la méthode standard en Python pour modéliser un événement qui a une certaine probabilité de se produire. Ajouter ces règles complexifie le modèle et le rend plus proche de la réalité biologique.

sujet 8

Question 1 Écrire, dans le fichier `addition_BCD.py`, une fonction `calcul_recettes()` qui additionne le prix de chaque menu vendu dans la journée en utilisant une boucle. Afficher le résultat de cette fonction. Sachant que la valeur théorique exacte est de 4 635 000 €, justifier le comportement observé.

Voici le code attendu à insérer au début du fichier :

```
def calcul_recettes():
    total = 0.0
    prix_menu = 2.27 + 5.19 + 1.81
    nb_menus = 1000 * 500

    for i in range(nb_menus):
        total += prix_menu

    return total
```

```
print("Recettes calculées :", calcul_recettes())
```

Justification : Le résultat affiché ne sera pas exactement 4 635 000. L'ordinateur stocke les nombres à virgule (type `float`) en base 2 (binaire). Certaines valeurs décimales ne peuvent pas être représentées de manière finie et exacte en binaire, ce qui crée une infime erreur d'arrondi. En additionnant 500 000 fois ce montant, ces minuscules erreurs s'accumulent et deviennent visibles dans le résultat final.

Commentaire : C'est une illustration classique des limites de la représentation en virgule flottante (norme IEEE 754). C'est pour cela qu'en informatique financière, on n'utilise jamais de simples flottants.

Question 2 Écrire la fonction `convertir_BCD_vers_decimal(liste_quartets)` qui prend en paramètre une liste de chaînes de caractères représentant des quartets BCD, et renvoie la valeur décimale correspondante (de type `float`). Ajouter une assertion pour vérifier que `convertir_BCD_vers_decimal(['0001', '0011', '0101', '0110'])` renvoie bien la valeur 13.56.

Voici le code attendu :

```
def convertir_BCD_vers_decimal(liste_quartets):
    chaine_chiffres = ""
    for quartet in liste_quartets:
        # int(... , 2) convertit la chaîne binaire en entier (base 10)
        chaine_chiffres += str(int(quartet, 2))

    # On applique la convention "virgule implicite deux rangs avant la fin"
    partie_entiere = chaine_chiffres[:-2]
    partie_decimale = chaine_chiffres[-2:]

    # S'il n'y a pas de partie entière, on met un "0"
    if partie_entiere == "":
        partie_entiere = "0"

    return float(partie_entiere + "." + partie_decimale)
```

Assertion demandée

```
assert convertir_BCD_vers_decimal(['0001', '0011', '0101', '0110']) == 13.56
```

Commentaire : On reconstitue la chaîne de caractères représentant le nombre décimal global, puis on insère le point décimal à la bonne place grâce aux tranches (slicing) de listes, avant de convertir le tout en flottant.

Question 3 Analyser le code fourni. Identifier l'oubli de l'étape de correction dans l'algorithme, puis insérer un appel à la fonction `corriger_BCD` au bon endroit pour résoudre ce problème.

Voici la correction à apporter dans la boucle de la fonction `additionner_nombres_format_BCD(a, b)` fournie dans le fichier :

```
# Addition binaire simple des quartets
somme, retenue = additionner_binaire_quartets(
    liste_quartets1[index], liste_quartets2[index], retenue)

# --- CORRECTION À INSÉRER ICI ---
# Si le quartet dépasse 9 ou génère une retenue, on corrige
somme, retenue = corriger_BCD(somme, retenue)
# -----

resultat.insert(0, somme)
```

Commentaire : Le BCD n'est pas une simple base 2, car un quartet ne peut pas dépasser 9. L'appel à `corriger_BCD` applique le "+6" compensatoire nécessaire pour que les opérations de retenue fonctionnent correctement en base 10 simulée.

Question 4 Tester maintenant l'addition de 23 et de 4 avec votre code et décrire ce que vous observez. Modifier la fonction `aligner_quartets(q1, q2)` pour qu'elle ajoute des quartets '0000' au début du nombre le plus court.

Observation : L'addition de '23' et '4' provoque une erreur ou un résultat incohérent car les deux listes de quartets n'ont pas la même taille. L'algorithme actuel, qui parcourt en utilisant `index`, va chercher des indices en dehors de la plus petite liste, ce qui lève une exception (comme `IndexError`).

Voici la fonction `aligner_quartets` modifiée :

```
def aligner_quartets(q1: list, q2: list) -> tuple:
    """
    Doit équilibrer les deux listes en ajoutant des '0000' à gauche
    de la liste la plus courte.
    """
    while len(q1) < len(q2):
        q1.insert(0, '0000')

    while len(q2) < len(q1):
        q2.insert(0, '0000')

    return q1, q2
```

Commentaire : Ajouter des "0" à gauche d'un nombre (ou des quartets '0000' en BCD) ne change pas sa valeur. Cela permet d'aligner correctement les puissances de 10 lors de l'addition en colonne. L'utilisation d'une boucle `while` est très adaptée pour combler l'écart de longueur.

sujet 9

Question 1 Écrire la méthode `distance` de la classe `Sommet`. Elle prend en paramètre un objet de type `Sommet`. La méthode renvoie la distance entre l'objet courant et l'objet de type `Sommet` passé en paramètre.

Voici le code attendu dans le fichier `Sommet.py` :

```
def distance(self, s):
    return math.sqrt((s.x - self.x)**2 + (s.y - self.y)**2 + (s.z - self.z)**2)
```

Commentaire : On applique directement la formule mathématique de la distance dans l'espace donnée dans l'énoncé. On utilise `math.sqrt()` (racine carrée) du module `math` importé au début du fichier. L'objet courant est représenté par `self` et le point cible par `s`.

Question 2 Écrire la méthode `sommets_adjacents` de la classe `Objet3D` qui prend en entrée deux points donnés par leurs coordonnées et renvoie `True` s'ils représentent une arête de l'objet 3D et `False` sinon.

Voici le code attendu dans le fichier `Objet3D.py` :

```
def sommets_adjacents(self, s1, s2):
    # On parcourt toutes les faces de l'objet 3D
    for face in self.faces:
        # Si les deux sommets appartiennent à la même face, ils sont adjacents
        if s1 in face.sommets and s2 in face.sommets:
            return True
    return False
```

Commentaire : La topologie d'un objet 3D (format OBJ) définit les faces via les sommets qui la composent. Deux sommets forment une arête (ou sont adjacents) s'ils appartiennent tous les deux à la liste des sommets d'au moins une même face. Le mot-clé `in` permet de vérifier facilement l'appartenance à une liste.

Question 3 Écrire une méthode `estimation_impression` pour la classe `Imprimante3D` qui prend en paramètre un `Objet3D` et renvoie une estimation de son temps d'impression en secondes.

Voici le code attendu dans le fichier `Imprimante3D.py` :

```
def estimation_impression(self, objet):
    # 1. Calcul du volume d'impression (volume réel * taux de remplissage)
    volume_estime = objet.volume_cube_englobant()
    volume_impression = volume_estime * self.remplissage

    # 2. Calcul du temps (volume d'impression / vitesse d'extrusion)
    temps_secondes = volume_impression / self.vitesse_extrusion

    return temps_secondes
```

Commentaire : L'énoncé indique clairement les deux étapes de calcul. L'objet 3D passé en paramètre dispose d'une méthode `volume_cube_englobant()` que l'on peut appeler pour récupérer son volume réel estimé. Il suffit ensuite d'appliquer les formules avec les attributs de l'imprimante.

Question 4 Utilisez cette méthode pour doubler la dimension du cube présent dans `Objet3D.py` en l'affichant avant et après l'appel. Le cube n'apparaît pas deux fois plus grand, analyser le fonctionnement de la méthode `transformer` et proposer une correction.

Test à ajouter à la fin de `Objet3D.py` :

```
cube.afficher()
cube.transformer(2)
cube.afficher()
```

Analyse du problème : La méthode `transformer` initiale crée une *nouvelle* liste contenant de *nouveaux* sommets, et remplace `self.sommets`. Cependant, les objets `Face` contenus dans `self.faces` font toujours référence aux *anciens* sommets en mémoire ! Lors de l'affichage, ce sont les faces (et donc les anciens sommets non modifiés) qui sont dessinées.

Correction (à remplacer dans `Objet3D.py`) : Pour corriger cela, il faut modifier les coordonnées des sommets *en place*, sans créer de nouveaux objets `Sommet`.

```
def transformer(self, rapport):  
    """  
    Applique une transformation d'échelle à l'objet 3D en modifiant  
    directement les coordonnées de ses sommets existants.  
    """  
    for sommet in self.sommets:  
        sommet.x = sommet.x * rapport  
        sommet.y = sommet.y * rapport  
        sommet.z = sommet.z * rapport
```

Commentaire : C'est un grand classique de la programmation orientée objet : la différence entre modifier un objet existant (mutation) et créer une nouvelle instance. En modifiant les attributs x, y et z de l'objet Sommet directement, les faces (qui pointent vers ce même objet en mémoire) "verront" automatiquement la mise à jour lors de l'affichage.

sujet 10

Question 1 Écrire une fonction `total_conso` qui prend en paramètres : `donnees` (une liste de mesures) et `jour` (une chaîne représentant le jour) et renvoie la consommation totale d'eau (somme de l'eau chaude et de l'eau froide) de toutes les mesures pour ce jour. Par convention, si aucune mesure n'existe pour ce jour, la fonction renvoie `None`.

Voici le code attendu pour compléter la fonction :

```
def total_conso(donnees, jour):
    total = 0
    mesure_trouvee = False

    for mesure in donnees:
        if mesure["jour"] == jour:
            total += mesure["chaude"] + mesure["froide"]
            mesure_trouvee = True

    if mesure_trouvee:
        return total
    else:
        return None
```

Commentaire : On utilise un booléen `mesure_trouvee` pour différencier le cas où la consommation est de 0 (des mesures existent mais aucune eau n'a été consommée) du cas où il n'y a eu aucun relevé pour cette journée (renvoi de `None`).

Question 2 Une fuite est suspectée lorsqu'il y a au moins 3 mesures consécutives entre 00:00 et 05:00 inclus où la consommation totale est toujours non nulle. Écrire une fonction `fuite_possible` qui renvoie `True` si une fuite est possible ce jour-là, `False` sinon.

Voici le code attendu pour compléter la fonction :

```
def fuite_possible(donnees, jour):
    consecutifs = 0

    for mesure in donnees:
        if mesure["jour"] == jour:
            # On vérifie que l'heure est dans la bonne plage
            if "00:00" <= mesure["heure"] <= "05:00":
                conso = mesure["chaude"] + mesure["froide"]
                if conso > 0:
                    consecutifs += 1
                    if consecutifs >= 3:
                        return True
            else:
                consecutifs = 0 # Remise à zéro si la consommation est nulle

    return False
```

Commentaire : Les chaînes de caractères au format "HH:MM" peuvent être comparées directement avec les opérateurs `<=`. Le compteur `consecutifs` doit impérativement être réinitialisé à zéro dès qu'une mesure indique une consommation nulle.

Question 3 Expliquer pourquoi la fonction `lissage_conso`, testée avec la liste `[10, 20, 30, 40, 50]`, présente un résultat incorrect, et proposer une correction.

Explication : L'erreur se situe dans la branche `else` (pour les éléments intermédiaires). Le code fait la somme de trois valeurs (`valeurs[i-1] + valeurs[i] + valeurs[i+1]`) mais divise par 2 au lieu de 3 pour calculer la moyenne.

Correction (à remplacer dans la fonction `lissage_conso`) :

```
else:
    # Remplacement de la division par 2 par une division par 3
    m = (valeurs[i-1] + valeurs[i] + valeurs[i+1]) / 3
```

Commentaire : La moyenne de `N` éléments s'obtient toujours en divisant la somme totale par `N`. Il faut donc bien diviser par 3 pour le lissage central.

Question 4 La fonction `lissage_conso` prend en compte les cas limites dans lesquels la liste fournie contient seulement deux valeurs. Identifier un autre cas limite qui n'est pas pris en compte par la fonction, et proposer une solution pour y remédier.

Identification du cas limite : Si la liste fournie est vide `[]` ou ne contient qu'une seule valeur `[10]`. Avec une seule valeur, la boucle s'exécute pour `i = 0`. Le code tente d'accéder à `valeurs[i+1]` (soit `valeurs[1]`), ce qui provoque une erreur `IndexError` car l'indice n'existe pas.

Solution proposée (à insérer au tout début de `lissage_conso`) :

```
def lissage_conso(valeurs):  
    # --- AJOUT POUR GÉRER LES LISTES DE TAILLE 0 OU 1 ---  
    if len(valeurs) == 0:  
        return []  
    if len(valeurs) == 1:  
        return [valeurs[0]]  
    # -----  
  
    lisse = []  
    for i in range(len(valeurs)):  
        # ... suite du code
```

Commentaire : Il est très courant en algorithmique de devoir "protéger" l'entrée d'une fonction en traitant manuellement les cas triviaux (listes vides, listes d'un seul élément) avant d'appliquer une logique qui nécessite plusieurs éléments.

sujet 11

Question 1 Écrire le code de la fonction *distance* qui prend en paramètres deux habitats sous la forme de dictionnaires contenant au moins les clés 'vegetation', 'proximite_eau', 'densite_urbaine', 'disponibilite_proies' et qui renvoie la distance entre ces deux habitats selon la formule présentée au-dessus.

Voici le code attendu à insérer pour la fonction *distance* :

```
def distance(habitat_1, habitat_2):
    v = (habitat_1['vegetation'] - habitat_2['vegetation']) ** 2
    p = (habitat_1['proximite_eau'] - habitat_2['proximite_eau']) ** 2
    u = (habitat_1['densite_urbaine'] - habitat_2['densite_urbaine']) ** 2
    d = (habitat_1['disponibilite_proies'] - habitat_2['disponibilite_proies']) ** 2

    return sqrt(v + p + u + d)
```

Commentaire : On accède aux valeurs des dictionnaires en utilisant leurs clés. On applique ensuite la formule de la distance euclidienne fournie dans l'énoncé en utilisant `sqrt` (racine carrée) importée depuis le module `math`.

Question 2 Écrire le code de la fonction *distance_d_un_habitat* qui prend en paramètres un habitat sous la forme de dictionnaire et une liste d'habitats sous la forme de liste de dictionnaires. La fonction doit renvoyer une liste de tuples où chaque tuple contient : la distance entre l'habitat fourni et un habitat de la liste donnée ; le dictionnaire représentant l'habitat.

Voici le code attendu pour la fonction *distance_d_un_habitat* :

```
def distance_d_un_habitat(habitat, habitats):
    resultats = []
    for h in habitats:
        dist = distance(habitat, h)
        resultats.append((dist, h))
    return resultats
```

Commentaire : On crée une liste vide, puis on parcourt chaque habitat de la liste connue avec une boucle `for`. Pour chaque élément, on calcule sa distance avec notre habitat cible, et on ajoute un n-uplet (tuple) contenant la distance et le dictionnaire à notre liste de résultats.

Question 3 Tester la fonction *distance_d_un_habitat* avec l'habitat nouveau et la liste d'habitats fournis, en affichant les 3 premiers tuples de la liste.

Voici le code à rajouter à la fin de votre fichier pour faire ce test :

```
distances_calculees = distance_d_un_habitat(nouveau, zones_connues)
```

```
for i in range(3):
    print(distances_calculees[i])
```

Commentaire : On stocke le résultat renvoyé par la fonction dans une variable. Ensuite, on utilise une boucle (ou on fait 3 `print` successifs sur les indices 0, 1 et 2) pour afficher les trois premiers éléments de la liste.

Question 4 La fonction *presence_renard* contient une erreur de traitement des tuples. Corriger la fonction *presence_renard*.

Correction (à modifier dans *presence_renard*) : L'erreur se situe à l'intérieur de la boucle `for`. Le code essaie de vérifier la présence d'un renard dans la variable *distance* (qui est un nombre) au lieu de chercher dans *caracteristiques* (qui est le dictionnaire).

```
for h in habitats:
    distance = h[0]
    caracteristiques = h[1]

    # --- CORRECTION ICI ---
    # On vérifie la clé 'presence_renard' dans le dictionnaire, pas dans la distance
    if caracteristiques['presence_renard']:
        # -----
        n_renards += 1
```

Commentaire : Il est important de bien suivre le type de vos variables. `h[0]` est un flottant (la distance), tandis que `h[1]` est un dictionnaire. C'est bien le dictionnaire qui possède la clé 'presence_renard'.

Question 5 *L'habitat nouveau proposé est-il susceptible ou non de contenir une population de renards ? Expliquer en utilisant la fonction précédente avec plusieurs valeurs pour k.*

Test à ajouter à la fin du fichier :

```
print("Pour k=3 :", presence_renard(3, nouveau, zones_connues))
print("Pour k=5 :", presence_renard(5, nouveau, zones_connues))
print("Pour k=7 :", presence_renard(7, nouveau, zones_connues))
```

Explication : On exécute l'algorithme des *k plus proches voisins* (KNN) pour différentes valeurs impaires de k (pour éviter les égalités). Si la fonction renvoie True en majorité pour ces différentes valeurs de k, on peut en conclure que cet habitat est très susceptible d'accueillir des renards.

Commentaire : L'algorithme des k-plus proches voisins est un algorithme d'apprentissage automatique (machine learning) très classique. Tester plusieurs valeurs de k permet de vérifier la robustesse de la prédiction et d'éviter qu'elle ne soit faussée par une seule donnée atypique ("bruit").

sujet 12

Question 1 Écrire le code du constructeur `__init__` de la classe `Renard`.

Voici le code attendu pour remplacer le pass dans la méthode `__init__` de la classe `Renard` :

```
def __init__(self, identifiant, nom, poids, date_arrivee):
    self.identifiant = identifiant
    self.nom = nom
    self.poids = poids
    self.date_arrivee = date_arrivee
```

Commentaire : Le constructeur initialise les attributs de l'objet lors de sa création. L'utilisation du mot-clé `self` permet d'associer les valeurs passées en paramètres à l'instance courante de la classe.

Question 2 Écrire le code de la méthode `__str__` de la classe `Renard` qui renvoie une chaîne de caractères qui présente l'animal sous le format précis : "Renard ID [id] - [Nom] (Arrivé le [date_arrivee])". Tester ensuite cette classe en instanciant un renard dans une variable `renard1` [...] et afficher.

Voici le code pour la méthode `__str__` :

```
def __str__(self):
    return f"Renard ID {self.identifiant} - {self.nom} (Arrivé le {self.date_arrivee})"
```

Voici le test à ajouter tout en bas du fichier `gestion_refuge.py` :

```
renard1 = Renard(200, "Oscar", 5.1, "2026-01-01")
print(renard1)
```

Commentaire : Les f-strings (chaînes formatées débutant par `f`) permettent d'insérer facilement les attributs de l'objet. La méthode spéciale `__str__` est automatiquement appelée lorsque l'on utilise la fonction `print()` sur un objet.

Question 3 L'exécution de la méthode `importer_donnees` provoque une erreur logique lors de l'utilisation ultérieure des données, notamment lors de la manipulation du poids et de l'identifiant des renards. Identifier la source de cette erreur dans la lecture des données brutes, proposer une correction du code de la méthode, puis tester cette correction...

Identification de l'erreur : Le module `csv` importe toutes les données sous forme de chaînes de caractères (`str`). Ainsi, le poids devient par exemple "5.8" et non le nombre 5.8. Cela bloque les comparaisons numériques ultérieures (comme `poids < 6.0`).

Correction (à modifier dans `importer_donnees` de la classe `Refuge`) :

```
for ligne in lignes:
    # On convertit explicitement l'id en entier et le poids en flottant
    renard = Renard(int(ligne['id']), ligne['nom'],
                    float(ligne['poids']), ligne['date_arrivee'])
    self.recueillir(renard)
```

Test à ajouter en bas du fichier :

```
sos_goupil = Refuge("SOS Goupil", "12 rue de la Forêt")
sos_goupil.importer_donnees("donnees_renards.csv")
```

Commentaire : Il faut toujours vérifier le type des données provenant d'un fichier externe. On réalise ici un "cast" (conversion de type) avec `int()` pour l'entier et `float()` pour le nombre à virgule.

Question 4 Exécuter les deux méthodes d'analyse de la corpulence sur l'instance de votre refuge. Justifier le pourcentage obtenu en isolant et en affichant le nombre de renards peu corpulents par rapport au nombre total de renards hébergés.

Code à ajouter en bas du fichier (à la suite du test précédent) :

```
# Exécution des méthodes
renards_maigres = sos_goupil.lister_peu_corpulents()
pourcentage = sos_goupil.pourcentage_peu_corpulents()

# Justification et affichage
nb_maigres = len(renards_maigres)
nb_total = len(sos_goupil.liste_renards)

print(f"Nombre de renards peu corpulents : {nb_maigres}")
print(f"Nombre total de renards : {nb_total}")
print(f"Pourcentage calculé : {pourcentage} %")
```

Commentaire : La fonction `len()` permet d'obtenir la taille d'une liste. En affichant le numérateur (nombre de renards de moins de 6.0kg) et le dénominateur (nombre total), on peut vérifier par un simple calcul mental que le pourcentage retourné par la classe est correct.

sujet 13

Question 1 En utilisant cette fonction `recupere_donnees_fichier_csv`, écrire la ou les lignes de code qui récupèrent les données relevées par le ballon sonde dans 4 listes différentes (altitudes, températures, longitudes et latitudes).

Voici le code à ajouter :

```
altitudes, temperatures, longitudes, latitudes =  
recupere_donnees_fichier_csv("releves_ballon_sonde.csv")
```

Commentaire : La fonction renvoie un tuple de quatre listes. En Python, on peut "déballer" ce résultat directement dans quatre variables distinctes en une seule ligne.

Question 2 Écrire en Python une fonction nommée `conversion_K_en_C` qui prend en paramètre une liste de températures en kelvins (K) et retourne cette même liste de températures, mais converties en degrés Celsius (°C). Les valeurs de températures doivent être arrondies à 1 chiffre après la virgule. Écrire une ligne de code permettant de tester la fonction proposée.

Voici le code de la fonction et son test :

```
def conversion_K_en_C(liste_temperatures):  
    temp_Celsius = []  
    for t in liste_temperatures:  
        # Formule : Celsius = Kelvin - 273.15  
        valeur_convertie = round(t - 273.15, 1)  
        temp_Celsius.append(valeur_convertie)  
    return temp_Celsius
```

Test de la fonction

```
print(conversion_K_en_C([273.15, 288.15, 300.0]))
```

Commentaire : On utilise une boucle pour parcourir la liste d'origine, on applique la transformation mathématique à chaque élément, puis on ajoute le résultat arrondi dans une nouvelle liste qu'on renvoie à la fin.

Question 3 Proposer une écriture de la fonction `altitude_la_plus_froide` qui prend en paramètres une liste d'altitudes, ainsi qu'une liste des températures en degrés Celsius. Cette fonction doit renvoyer la température la plus froide, ainsi qu'une liste contenant la ou les altitudes correspondantes.

Voici le code de la fonction :

```
def altitude_la_plus_froide(liste_altitudes, liste_temperatures):  
    temp_min = min(liste_temperatures)  
    altitudes_froides = []  
  
    for i in range(len(liste_temperatures)):  
        if liste_temperatures[i] == temp_min:  
            altitudes_froides.append(liste_altitudes[i])  
  
    return temp_min, altitudes_froides
```

Commentaire : On identifie d'abord la valeur minimale grâce à `min()`. Ensuite, on parcourt les indices des listes pour retrouver toutes les altitudes qui partagent cette température minimale (cas où le ballon stagne ou rencontre plusieurs fois la même température).

Question 4 Observer le corps de la fonction `genere_kml`, puis, à l'aide d'une assertion, insérer une ligne de code qui garantit que les deux listes `liste_longitudes` et `liste_latitudes` sont de même longueur.

Voici la ligne à insérer au début de la fonction `genere_kml` :

```
def genere_kml(liste_longitudes, liste_latitudes):  
    assert len(liste_longitudes) == len(liste_latitudes), "Erreur : les listes de  
coordonnées n'ont pas la même taille"  
    # ... reste du code
```

Commentaire : L'instruction `assert` vérifie une condition. Si celle-ci est fausse, le programme s'arrête immédiatement avec un message d'erreur, évitant ainsi un plantage plus loin dans la boucle `for`.

Question 5 Écrire une ligne de code qui appelle la fonction `genere_kml` avec les deux listes de longitudes et latitudes obtenues à la question 1.

Voici l'appel à ajouter :

```
genere_kml(longitudes, latitudes)
```

Commentaire : On utilise simplement les noms des variables créées lors de l'étape de récupération des données.

Question 6 Proposer une amélioration du code visant à répondre à cette difficulté (la balise `<kml>` n'a pas été fermée en toute fin de fichier).

Il faut modifier la variable `bas_fichier` à la fin de la fonction `genere_kml` pour inclure la balise de fermeture racine du format KML.

Correction à apporter :

```
# Remplacer la ligne existante par :
bas_fichier = '</Document>\n</kml>\n'

fichier_kml.write(bas_fichier)
fichier_kml.close()
```

Commentaire : Un fichier KML est un fichier de type XML. Chaque balise ouverte (ici `<kml>` à la ligne 41) doit impérativement être fermée (`</kml>`) pour que le fichier soit jugé "bien formé" et lisible par les logiciels comme Google Earth.

sujet 14

Question 1 Écrire le corps de la méthode `nb_occupants_restants` de la classe `Piece`. Comme son nom l'indique, cette méthode doit renvoyer le nombre d'occupants restants dans la pièce.

Voici le code à insérer dans la classe `Piece` :

```
def nb_occupants_restants(self):
    total = 0
    for ligne in self.grille:
        for case in ligne:
            total += case
    return total
```

Commentaire : `self.grille` étant une liste de listes (tableau à deux dimensions), on utilise deux boucles imbriquées pour parcourir chaque case et cumuler le nombre de personnes présentes dans une variable `total`.

Question 2 Écrire le corps de la fonction `evacuation` afin qu'elle simule l'évacuation complète de la pièce et renvoie le nombre de tours nécessaire. On pourra, dans cette fonction, faire appel à la méthode `alerter` qui simule un tour [...]. En complément de la pièce à évacuer, la fonction `evacuation` a un paramètre `silencieux` dont la valeur par défaut est `True`. Si ce paramètre vaut `False`, l'état de la pièce doit être affiché à chaque tour dans la console.

Voici le code pour la fonction `evacuation` :

```
def evacuation(p, silencieux=True):
    tours = 0
    while p.nb_occupants_restants() > 0:
        tours += 1
        p.alerter()
        if not silencieux:
            print(p)
    return tours
```

Commentaire : On utilise une boucle `while` car on ne connaît pas à l'avance le nombre de tours nécessaires. La boucle s'arrête dès que la pièce est vide. L'affichage via `print(p)` utilise la méthode spéciale `__str__` déjà définie dans la classe.

Question 3 Modifier la méthode `ajouter_sortie(self, direction, position)` afin qu'il soit aussi possible d'ajouter une sortie dans les directions qui ne sont pour l'instant pas prises en compte : "S" (pour le sud) et "E" (pour l'est).

Voici la version complétée de la méthode :

```
def ajouter_sortie(self, direction, position):
    if direction == "N":
        self.sorties.append((0, position))
    elif direction == "O":
        self.sorties.append((position, 0))
    # --- AJOUTS ---
    elif direction == "S":
        self.sorties.append((self.i_max, position))
    elif direction == "E":
        self.sorties.append((position, self.j_max))
```

Commentaire : Pour le Sud, la ligne est fixe (la dernière, soit `i_max`) et la colonne varie selon `position`. Pour l'Est, c'est la colonne qui est fixe (la dernière, soit `j_max`) et la ligne qui varie.

Question 4 Identifier l'erreur logique et la variable non définie dans le code de cette méthode (`choix_sortie`), puis effectuer les corrections nécessaires afin qu'elle renvoie la sortie la plus proche.

Identification des erreurs :

1. La variable `d2` est utilisée mais jamais calculée.
2. La condition `if k < 0` est absurde car l'indice `k` de la boucle commence à 1.
3. On ne mettait pas à jour la distance de référence pour les comparaisons suivantes.

Correction de la méthode choix_sortie :

```
def choix_sortie(self, i, j):
    assert len(self.sorties) > 0, "Aucune sortie"
    choix = self.sorties[0]
    # Calcul de la distance de Manhattan pour la première sortie
    distance_min = abs(i - choix[0]) + abs(j - choix[1])

    for k in range(1, len(self.sorties)):
        autre_sortie = self.sorties[k]
        # Calcul de la distance pour la sortie k
        d2 = abs(i - autre_sortie[0]) + abs(j - autre_sortie[1])
        # Si cette sortie est plus proche, on la choisit
        if d2 < distance_min:
            choix = autre_sortie
            distance_min = d2
    return choix
```

Commentaire : C'est un algorithme classique de recherche de minimum. On initialise le "champion" avec la première sortie, puis on compare avec toutes les autres. On utilise la distance de Manhattan (somme des écarts absolus en abscisse et ordonnée) adaptée aux déplacements sur grille.

sujet 16

Question 1 Écrire la fonction `ecart_temperature(datas, annee)` qui prend en paramètres la liste des données et une année cible. Elle doit renvoyer l'écart de température correspondant à cette année, ou `None` si l'année n'est pas présente dans les données. Rédiger ensuite un jeu de tests (via des assertions) permettant de vérifier le bon fonctionnement de votre fonction, y compris pour une année hors du jeu de données.

Voici le code de la fonction et les tests associés :

```
def ecart_temperature(datas, annee):
    for dico in datas:
        if dico["année"] == annee:
            return dico["écart"]
    return None

# Jeu de tests
# On suppose que datas_temperature est déjà chargé
assert ecart_temperature(datas_temperature, 2020) == 1.01
assert ecart_temperature(datas_temperature, 1948) == 0.0
assert ecart_temperature(datas_temperature, 2050) is None
```

Commentaire : On parcourt la liste de dictionnaires. Dès que l'année correspond à la clé "année", on renvoie la valeur associée à la clé "écart". Si la boucle se termine sans avoir trouvé l'année, on renvoie `None`.

Question 2 En utilisant la fonction `derniere_annee_ecart_negatif` (déjà fournie et qui utilise votre fonction), déterminer quelle a été la dernière année où la température mondiale était inférieure à la moyenne de référence. Pour obtenir ce résultat, il faut exécuter la ligne suivante dans la console ou à la fin du script :

```
print(derniere_annee_ecart_negatif(datas_temperature))
```

Commentaire : La fonction fournie part de l'année la plus récente et "remonte le temps" à l'aide d'une boucle `while` le tant que l'écart est positif ou nul. Elle s'arrête dès qu'elle croise le premier écart négatif.

Question 3 En exécutant `prevision(datas_temperature, 2040, 20)`, le résultat obtenu semble absurde compte tenu du réchauffement actuel. Cette absurdité provient d'une erreur de logique qui s'est glissée dans la fonction annexe `moyenne_ecarts`. Analyser le code de cette fonction, identifier l'erreur, et la corriger. Que devient alors la prévision pour 2040 ?

L'erreur se situe à la ligne du calcul de la somme : le script effectue une soustraction au lieu d'une addition.

Correction de la fonction `moyenne_ecarts` :

```
def moyenne_ecarts(annee_debut, annee_fin, datas):
    # ... (début identique)
    for dico in datas:
        if annee_debut <= dico["année"] and dico["année"] <= annee_fin:
            somme = somme + dico["écart"] # Correction : + au lieu de -
            compteur += 1
    return somme / compteur
```

Après correction, l'appel `prevision(datas_temperature, 2040, 20)` donne un résultat cohérent avec la tendance (environ **1.45 °C** selon les séries de données).

Commentaire : Une moyenne est le rapport entre la **somme** des valeurs et l'effectif total. Soustraire les valeurs faussait totalement le calcul de la droite de régression.

Question 4 Compléter le code de la fonction `[graphique(datas)]` en générant les listes `annees` (pour l'axe des abscisses) et `ordonnees`. Indication : Pour obtenir des bandes colorées de hauteur uniforme couvrant tout le graphique, la liste `ordonnees` devra contenir la valeur 1 pour chaque année traitée.

Voici les lignes à ajouter dans la fonction `graphique` :

```
# Création des listes annees et ordonnees
# pour les abscisses et ordonnées
annees = [dico["année"] for dico in datas]
ordonnees = [1 for dico in datas]
```

Puis on appelle la fonction pour afficher le résultat :
`graphique(datas_temperature)`

Commentaire : On utilise ici des listes par compréhension. La liste `ordonnees` crée une série de "1" de la même longueur que la liste des données, ce qui permet à la fonction `ax.bar` de dessiner des barres de hauteur identique pour chaque année. Seule la couleur change grâce au paramètre `color`.

sujet 17

Question 1 Écrire la fonction `total_par_type(mouvements, type_mouvement)` qui prend en paramètres une liste de mouvements et une chaîne de caractères (parmi 'dépense' ou 'recette'). Cette fonction doit renvoyer la somme totale des montants correspondant à ce type précis. La fonction renverra 0 si aucun mouvement ne correspond. Créer ensuite une fonction `test_total()` contenant au moins deux assertions pour valider le bon fonctionnement de votre code sur le jeu de données `mouvements_test`.

Voici le code de la fonction et de son test :

```
def total_par_type(mouvements, type_mouvement):
    total = 0.0
    for m in mouvements:
        if m['type'] == type_mouvement:
            total += m['montant']
    return total

def test_total():
    # Sur le jeu de test : 1200.0 + 300.0 + 800.0 = 2300.0
    assert total_par_type(mouvements_test, 'recette') == 2300.0
    # Sur le jeu de test : 450.0 + 200.0 + 1500.0 = 2150.0
    assert total_par_type(mouvements_test, 'dépense') == 2150.0
```

Commentaire : On utilise un accumulateur (`total`) initialisé à 0. On parcourt la liste de dictionnaires et, pour chaque dictionnaire, on vérifie si la valeur associée à la clé 'type' correspond au paramètre attendu pour ajouter le 'montant' à la somme.

Question 2 Calculer manuellement le solde annuel attendu pour la liste `mouvements_test`. Écrire ensuite une fonction `test_solde_annuel()` contenant une assertion qui vérifie que la fonction `solde_annuel(mouvements_test)` renvoie bien ce résultat théorique. Exécuter ce test.

Calcul manuel :

- Total Recettes = 1200.0 (mois 1) + 300.0 (mois 6) + 800.0 (mois 12) = 2300.0 €
- Total Dépenses = 450.0 (mois 6) + 200.0 (mois 12) + 1500.0 (mois 12) = 2150.0 €
- Solde = Recettes - Dépenses = 2300.0 - 2150.0 = **150.0 €**

Voici la fonction de test :

```
def test_solde_annuel():
    resultat = solde_annuel(mouvements_test)
    assert resultat == 150.0, f"Erreur : attendu 150.0, obtenu {resultat}"
```

Commentaire : Le solde annuel est la différence entre la somme de toutes les recettes et la somme de toutes les dépenses sur l'ensemble de l'année.

Question 3 Analyser le code de la fonction `solde_annuel` pour identifier la source de cette erreur logique, expliquer pourquoi certains mouvements ont été ignorés, puis proposer une correction. Appliquer ensuite votre fonction corrigée sur le fichier `budget_complet.csv` pour annoncer le véritable solde du club.

Analyse de l'erreur : L'erreur classique dans ce type d'exercice (souvent présente dans le code non fourni ici mais suggérée par l'énoncé) est que la fonction ne traite qu'un seul mouvement par mois ou écrase le solde mensuel au lieu de cumuler tous les mouvements. Si plusieurs lignes concernent le même mois, seules les dernières sont prises en compte.

Correction de la fonction `solde_annuel` : La méthode la plus simple et la plus fiable consiste à réutiliser la fonction `total_par_type` écrite à la question 1.

```
def solde_annuel(mouvements):  
    recettes = total_par_type(mouvements, 'recette')  
    depenses = total_par_type(mouvements, 'dépense')  
    return recettes - depenses
```

Application au fichier complet : En décommentant les lignes du programme principal et en utilisant la fonction corrigée, on obtient le solde réel.

```
mouvements_complets = lire_mouvements_depuis_csv("budget_complet.csv")  
print("Le solde annuel sur le fichier complet est de :",  
solde_annuel(mouvements_complets))
```

Commentaire : En utilisant `total_par_type`, on s'assure de parcourir l'intégralité de la liste `mouvements` sans se soucier du mois, ce qui garantit qu'aucune donnée n'est oubliée, même s'il y a des centaines de lignes pour le même mois.

sujet 18

Question 1 Écrire une fonction `temperature_moyenne(zone, donnees)` qui prend en paramètres une chaîne `zone` (nom d'un archipel) et un tableau `donnees` de dictionnaires représentant les relevés. Elle renvoie la température moyenne (float) pour cette zone ou `None` si la zone n'a aucun relevé.

```
def temperature_moyenne(zone, donnees):
    somme = 0
    compteur = 0
    for releve in donnees:
        if releve['zone'] == zone:
            somme += releve['temperature']
            compteur += 1

    if compteur == 0:
        return None
    return somme / compteur
```

Commentaire : On parcourt la liste des relevés et on filtre ceux qui correspondent à la zone demandée. On cumule les températures et on compte le nombre de relevés pour calculer la moyenne à la fin.

Question 2 Écrire une fonction `detecter_anomalies(zone, seuil, donnees)` qui calcule la température moyenne de la zone et renvoie la liste des dates où la température s'écarte de plus de `seuil` degrés (en valeur absolue) de cette moyenne. Elle renvoie une liste vide si la zone n'existe pas.

```
def detecter_anomalies(zone, seuil, donnees):
    moyenne = temperature_moyenne(zone, donnees)
    if moyenne is None:
        return []

    dates_anomalies = []
    for releve in donnees:
        if releve['zone'] == zone:
            ecart = abs(releve['temperature'] - moyenne)
            if ecart > seuil:
                dates_anomalies.append(releve['date'])
    return dates_anomalies
```

Commentaire : On réutilise la fonction de la question 1 pour obtenir la référence. On utilise `abs()` pour calculer l'écart sans se soucier de savoir si la température est au-dessus ou en dessous de la moyenne.

Question 3 Compléter les trois fonctions de test pour la fonction `evolution_par_decennie` en utilisant le jeu de données fourni. Vos tests doivent permettre d'identifier le bug présent dans le code.

```
def test_zone_inexistante():
    resultat = evolution_par_decennie('Inconnue', donnees_test)
    assert resultat == {}

def test_une_seule_decennie():
    # Les Marquises n'ont que des relevés entre 2020 et 2022
    resultat = evolution_par_decennie('Marquises', donnees_test)
    # Le bug fera que la clé sera 202 au lieu de 2020
    assert 2020 in resultat
    assert len(resultat) == 1

def test_plusieurs_decennies():
    # Societe a des relevés en 2010, 2011 (décennie 2010) et 2020, 2021 (décennie 2020)
    resultat = evolution_par_decennie('Societe', donnees_test)
    assert 2010 in resultat
    assert 2020 in resultat
```

Commentaire : En exécutant ces tests, on remarque que `2010 in resultat` échoue. En inspectant le dictionnaire, on verrait des clés comme 201 ou 202. Le bug vient du calcul de la décennie.

Question 4 Après avoir identifié le bug grâce à vos tests, corriger la fonction `evolution_par_decennie`.

Voici la correction de la ligne fautive dans la fonction `evolution_par_decennie` :

```
# Ligne originale : decennie = (annee // 10)
```

```
# Ligne corrigée :
```

```
    decennie = (annee // 10) * 10
```

Réécriture de la partie concernée dans la boucle :

```
    for releve in releves_zone:
```

```
        annee = int(releve['date'].split('-')[0])
```

```
        # On multiplie par 10 après la division entière pour retrouver l'année ronde
```

```
        decennie = (annee // 10) * 10
```

```
    if decennie not in temperatures_par_decennie:
```

```
        temperatures_par_decennie[decennie] = []
```

```
    # ... la suite reste identique
```

Commentaire : L'opérateur `//` est la division entière. Pour l'année 2015, `2015 // 10` donne 201. Pour obtenir la décennie 2010, il faut multiplier ce résultat par 10.

sujet 19

Question 1 *Un réservoir est considéré comme en pénurie si son taux de remplissage est strictement inférieur à 20 %. Écrire en Python une fonction nommée `est_en_penurie` qui prend en paramètres une liste de réservoirs et le nom d'un réservoir. La fonction renverra un booléen indiquant si ce réservoir est en pénurie.*

```
def est_en_penurie(reservoirs, nom_reservoir):
    for r in reservoirs:
        if r["nom"] == nom_reservoir:
            taux = r["volume"] / r["capacite"]
            return taux < 0.20
    return False
```

Commentaire : On parcourt la liste des dictionnaires pour trouver celui dont la clé "nom" correspond à la demande. Le taux de remplissage se calcule en divisant le volume actuel par la capacité totale.

Question 2 *Écrire en Python une fonction nommée `volume_par_district` qui prend en paramètres une liste de réservoirs et qui renvoie un dictionnaire ayant pour clés chaque nom différent de district associés au volume total d'eau, en litres, disponible dans ce district.*

```
def volume_par_district(reservoirs):
    bilan = {}
    for r in reservoirs:
        district = r["district"]
        if district in bilan:
            bilan[district] += r["volume"]
        else:
            bilan[district] = r["volume"]
    return bilan
```

Commentaire : C'est un algorithme classique d'agrégation. On crée un dictionnaire vide, puis pour chaque réservoir, on ajoute son volume à l'entrée correspondante du district. Si le district n'existe pas encore dans les clés, on l'initialise.

Question 3 La fonction `volume_moyen` est censée renvoyer le volume moyen d'eau disponible. Le code est incorrect. Proposer des tests (assertions) mettant en évidence les problèmes et proposer une version corrigée.

Tests (assertions) :

```
# Test liste non vide
assert len(reservoirs) > 0, "Erreur : la liste est vide"

# Test cohérence (la moyenne ne peut pas être supérieure au volume max présent)
vol_max = max(r["volume"] for r in reservoirs)
assert volume_moyen(reservoirs) <= vol_max, "Erreur : moyenne incohérente"

# Test cas simple (deux réservoirs identiques)
test_simples = [{"volume": 100}, {"volume": 100}]
assert volume_moyen(test_simples) == 100, "Erreur de calcul sur la moyenne"
```

Correction de la fonction :

```
def volume_moyen(reservoirs):
    if len(reservoirs) == 0:
        return 0
    somme_totale = 0
    for r in reservoirs:
        somme_totale += r["volume"]
    # Correction : on divise par N et non par N-1
    moyenne = somme_totale / len(reservoirs)
    return moyenne
```

Commentaire : L'erreur principale résidait dans le calcul de la moyenne (division par `len(reservoirs)-1` au lieu de `len(reservoirs)`). Il faut aussi gérer le cas d'une liste vide pour éviter une division par zéro.

Question 4 À l'aide des fonctions précédentes, écrire une fonction `districts_vulnérables` permettant d'identifier les districts dont le volume moyen est inférieur à 80 % du volume moyen global.

```
def districts_vulnérables(reservoirs):
    seuil_critique = 0.8 * volume_moyen(reservoirs)
    dict_rpd = reservoirs_par_district(reservoirs)
    vulnérables = []

    for district, liste_r in dict_rpd.items():
        # On réutilise volume_moyen pour calculer la moyenne du district
        if volume_moyen(liste_r) < seuil_critique:
            vulnérables.append(district)

    return vulnérables
```

Commentaire : On calcule d'abord le seuil de référence sur toute la ville. Ensuite, on utilise le dictionnaire regroupant les réservoirs par district (fourni dans l'énoncé) pour calculer la moyenne locale de chaque zone et la comparer au seuil.

sujet 20

Question 1 Dans le fichier `code_empreinte.py`, écrire en Python le code de la fonction `calculer_empreinte(utilisateur)` qui prend en paramètre un dictionnaire représentant les usages d'un utilisateur (au format décrit précédemment) et qui renvoie l'empreinte carbone totale mensuelle en gCO2e qui est de type entier.

```
def calculer_empreinte(utilisateur):  
    total = 0  
    for activite, quantite in utilisateur.items():  
        if activite in EMISSIONS:  
            total += quantite * EMISSIONS[activite]  
    return int(total)
```

Commentaire : On parcourt les éléments du dictionnaire `utilisateur`. Pour chaque activité, si elle est présente dans notre dictionnaire de référence `EMISSIONS`, on multiplie la quantité consommée par le facteur d'émission correspondant et on l'ajoute au total.

Question 2 Écrire en Python le code de la fonction `classer_par_impact(utilisateur)` qui prend en paramètre un dictionnaire représentant les usages d'un utilisateur et qui renvoie un dictionnaire contenant trois clés 'fort', 'moyen' et 'faible', chacune associée à une liste des noms des activités correspondantes.

```
def classer_par_impact(utilisateur):  
    classement = {'fort': [], 'moyen': [], 'faible': []}  
    for activite, quantite in utilisateur.items():  
        if activite in EMISSIONS:  
            impact = quantite * EMISSIONS[activite]  
            if impact >= 1000:  
                classement['fort'].append(activite)  
            elif impact >= 200:  
                classement['moyen'].append(activite)  
            else:  
                classement['faible'].append(activite)  
    return classement
```

Commentaire : On initialise un dictionnaire avec des listes vides pour chaque catégorie. On calcule l'impact de chaque activité de l'utilisateur, puis on utilise une structure conditionnelle (`if/elif/else`) pour ajouter le nom de l'activité dans la bonne liste selon les seuils demandés.

Question 3 Compléter le code de la fonction `test_comparer` en ajoutant au moins deux tests pertinents qui permettent de vérifier différents aspects du comportement de la fonction. Pour chaque test ajouté, une brève justification doit être donnée.

```
def test_comparer():  
    diff = comparer(utilisateur4, utilisateur5)  
    assert diff['emails_simples'] == -200 # (50-100) * 4  
    assert diff['recherches'] == 350 # (100-50) * 7  
  
    # Test 1 : Comparaison avec un profil vide (utilisateur6)  
    diff_vide = comparer(utilisateur6, utilisateur4)  
    assert diff_vide['emails_simples'] == 400 # (100 - 0) * 4  
    # Justification : Vérifie que la fonction gère correctement l'absence d'une activité  
    # chez l'un des deux utilisateurs (valeur par défaut à 0).  
  
    # Test 2 : Comparaison d'un utilisateur avec lui-même  
    diff_identique = comparer(utilisateur1, utilisateur1)  
    assert diff_identique['streaming_hd'] == 0  
    # Justification : Vérifie que la différence est bien nulle lorsque les comportements  
    # sont strictement identiques.
```

Commentaire : Les tests doivent couvrir les cas limites, comme l'absence de données (dictionnaire vide) ou l'égalité parfaite, pour s'assurer que l'algorithme est robuste.

Question 4 Identifier dans quels cas l'erreur se produit dans `comparer_v2` et proposer une correction.

Analyse de l'erreur : L'erreur est une `ZeroDivisionError`. Elle se produit lorsque `emission1` vaut 0 (c'est-à-dire quand l'utilisateur 1 n'a pas pratiqué l'activité en question). La division par zéro est impossible en informatique et fait planter le programme.

Correction proposée : Il faut ajouter une condition pour vérifier que `emission1` est différent de zéro avant d'effectuer le calcul du pourcentage.

```
def comparer_v2(u1, u2):
    ecarts = {}
    for activite in EMISSIONS:
        quantite1 = 0
        quantite2 = 0
        if activite in u1:
            quantite1 = u1[activite]
        if activite in u2:
            quantite2 = u2[activite]
        emission1 = quantite1 * EMISSIONS[activite]
        emission2 = quantite2 * EMISSIONS[activite]

        if emission1 == 0:
            # On peut décider de mettre None ou 0, ou de ne pas ajouter la clé
            ecarts[activite] = None
        else:
            ecarts[activite] = (emission2 - emission1)/emission1 * 100
    return ecarts
```

Commentaire : La division par la valeur de référence (`emission1`) est nécessaire pour obtenir un pourcentage d'évolution. Si cette référence est nulle, le concept de "pourcentage d'augmentation" n'est mathématiquement pas défini. On utilise donc un test `if` pour protéger le calcul.

sujet 21

Question 1 Écrire la méthode `traiter_reponse(self, succes)` qui prend en paramètre un booléen `succes` (True si l'utilisateur a bien répondu, False sinon). Cette méthode doit mettre à jour l'attribut `self.niveau` de la carte selon les règles de Leitner énoncées, puis calculer et mettre à jour l'attribut `self.date_prochaine` en utilisant la liste globale `DELAIS`.

```
def traiter_reponse(self, succes):
    if succes:
        if self.niveau < 4:
            self.niveau += 1
        else:
            self.niveau = 0

    delai = DELAIS[self.niveau]
    self.date_prochaine = date_futur(delai)
```

Commentaire : On utilise une structure conditionnelle pour faire évoluer le niveau. En cas de succès, on vérifie que l'on ne dépasse pas l'indice maximum du tableau `DELAIS` (qui est 4). On met ensuite à jour la date de révision en appelant la fonction d'aide fournie.

Question 2 Écrire une fonction `extraire_cartes_du_jour(paquet, date_jour)` qui prend en paramètres une liste de cartes `paquet` et une date de référence `date_jour` et qui renvoie une nouvelle liste contenant uniquement les cartes dont la `date_prochaine` est inférieure ou égale à `date_jour`.

```
def extraire_cartes_du_jour(paquet, date_jour):
    selection = []
    for carte in paquet:
        if carte.date_prochaine <= date_jour:
            selection.append(carte)
    return selection
```

Commentaire : C'est un algorithme classique de filtrage. On parcourt la liste des objets `Carte` et on teste l'attribut de date par rapport au paramètre. Si la condition est vraie, on ajoute l'objet à notre liste de résultats.

Question 3 Exécuter la fonction `test_reforcement()` fournie. Observer le résultat affiché dans la console et constater l'incohérence. Analyser le code de la fonction `extraire_cartes_a_renforcer(paquet)`, identifier la source de cette erreur logique, puis corriger le code.

Analyse du bug : L'incohérence vient du fait que lorsque la fonction trouve une carte avec un niveau strictement inférieur au minimum actuel (`carte.niveau < niveau_min`), elle change bien la valeur de `niveau_min` mais elle ajoute simplement la carte à la liste `a_renforcer`. Or, cette liste contient encore les cartes précédemment trouvées qui avaient l'ancien niveau minimum (plus élevé).

Correction de la fonction :

```
def extraire_cartes_a_renforcer(paquet):
    if len(paquet) == 0:
        return []

    niveau_min = paquet[0].niveau
    a_renforcer = []

    for carte in paquet:
        if carte.niveau < niveau_min:
            niveau_min = carte.niveau
            # CORRECTION : On réinitialise la liste car on a trouvé un nouveau minimum
            a_renforcer = [carte]
        elif carte.niveau == niveau_min:
            a_renforcer.append(carte)

    return a_renforcer
```

Commentaire : Pour corriger l'erreur, il faut vider la liste et n'y mettre que la nouvelle carte dès qu'un niveau plus faible est détecté. L'instruction `a_renforcer = [carte]` remplace la liste existante par une nouvelle liste contenant uniquement l'élément courant.

sujet 22

Question 1 Écrire une fonction en Python nommée `bin2dec` qui prend en paramètre un tuple représentant un nombre binaire et qui renvoie l'entier naturel en base 10 correspondant. À l'aide des informations ci-dessus, déterminer la chaîne de caractères contenue dans le QR code de la figure 1 pour découvrir le nom de l'inventeur de ce système de codage.

```
def bin2dec(tuple_binaire):
    decimal = 0
    puissance = 0
    # On parcourt le tuple de droite à gauche
    for i in range(len(tuple_binaire) - 1, -1, -1):
        if tuple_binaire[i] == 1:
            decimal += 2**puissance
        puissance += 1
    return decimal
```

Commentaire : Pour convertir du binaire au décimal, on additionne les puissances de 2 correspondant aux positions où se trouve un bit "1". On commence par 20 pour le bit le plus à droite.

Question 2 Écrire une fonction en Python nommée `qrcode2dec` qui prend en paramètre une liste de tuples représentant un QR code et qui renvoie une liste d'entiers décimaux correspondant à chacune des lignes du QR code. Proposer un test de `qrcode2dec` qui utilisera la représentation du QR code de la Figure 1 fourni dans le module `ascii.py`.

```
def qrcode2dec(qrcode):
    liste_decimaux = []
    for ligne in qrcode:
        liste_decimaux.append(bin2dec(ligne))
    return liste_decimaux
```

```
# Test de la fonction
print(qrcode2dec(ascii.figure1))
```

Commentaire : On parcourt la liste de tuples (le QR code) et pour chaque tuple, on appelle la fonction `bin2dec` précédemment créée pour remplir une nouvelle liste de nombres entiers.

Question 3 Exécuter la fonction fournie `test_dec2str` et observer les résultats affichés. Identifier le problème et proposer une modification de la fonction `dec2str` pour l'éviter.

Identification du problème : Le `test3` contient la valeur 233. Or, le dictionnaire `dict_ascii` s'arrête à l'indice 127. L'appel `table_ascii[233]` provoque donc une erreur de type `KeyError` car la clé n'existe pas.

Correction de la fonction :

```
def dec2str(liste_dec):
    table_ascii = ascii.dict_ascii
    chaine = ""
    for entier in liste_dec:
        if entier in table_ascii:
            chaine += table_ascii[entier]
        else:
            chaine += "?" # Caractère de remplacement si le code n'est pas reconnu
    return chaine
```

Commentaire : Avant d'accéder à une clé dans un dictionnaire, il est prudent de vérifier si elle existe avec le mot-clé `in` pour éviter que le programme ne s'arrête brutalement sur une erreur.

Question 4 Analyser le code de la fonction `str2qrcode`. Identifier la source de ce problème, puis proposer une modification du code afin de garantir l'obtention d'un QR code simplifié valide.

Identification du problème : La fonction `bin(entier)[2:]` génère une chaîne binaire de longueur variable (par exemple "1" pour l'entier 1). Or, le format du QR code simplifié exige des lignes de **8 bits** fixes. De plus, la fonction actuelle ne remplit pas le tuple avec des zéros à gauche pour atteindre cette longueur.

Correction de la fonction :

```
def str2qrcode(message):
    qrcode = []
    table_inverse = {valeur: cle for cle, valeur in ascii.dict_ascii.items()}

    for caractere in message:
        entier = table_inverse.get(caractere, 63)
        # On convertit en binaire et on complète avec des 0 pour avoir 8 caractères
        binaire_str = "0"*(8-len(bin(entier)[2:])) + bin(entier)[2:]
        # On crée un tuple de 8 entiers (0 ou 1)
        ligne = tuple(int(bit) for bit in binaire_str)
        qrcode.append(ligne)

    return qrcode
```

sujet 23

Question 1 Implémenter les méthodes `decoder_temperature` et `decoder_humidite`. Écrire des tests pour vérifier que ces méthodes décodent bien 26.4 et 62 avec la trame d'exemple.

Dans la classe Transmission :

```
def decoder_temperature(self):
    valeur_binaire = self._trame[16:28]
    valeur_decimale = int(valeur_binaire, 2)
    self._temperature = round((valeur_decimale - 900) / 10, 1)

def decoder_humidite(self):
    valeur_binaire = self._trame[28:36]
    if valeur_binaire == "10100000":
        self._humidite = 100
    else:
        # Décodage BCD : 4 bits pour les dizaines, 4 bits pour les unités
        dizaines = int(valeur_binaire[0:4], 2)
        unites = int(valeur_binaire[4:8], 2)
        self._humidite = dizaines * 10 + unites
```

Tests

```
t_ex = Transmission("001010101100100001001000110001100011000101101")
assert t_ex.get_temperature() == 26.4
assert t_ex.get_humidity() == 62
```

Commentaire : Pour la température, on convertit la tranche correspondante en entier puis on applique la formule fournie. Pour l'humidité (BCD), on traite séparément les deux groupes de 4 bits pour reconstituer le nombre décimal chiffre par chiffre.

Question 2 Écrire la méthode `est_valide(self)` qui renvoie `True` si les 4 bits de contrôle correspondent bien à la parité des 4 blocs de données, et `False` sinon.

```
def est_valide(self):
    # On isole les 4 blocs et les 4 bits de contrôle
    blocs = [self._trame[0:8], self._trame[8:16], self._trame[16:28], self._trame[28:36]]
    bits_controle = self._trame[36:40]

    for i in range(4):
        # Compte le nombre de '1' dans le bloc
        nb_uns = blocs[i].count('1')
        # La parité attendue est 0 si pair, 1 si impair (nb_uns % 2)
        if str(nb_uns % 2) != bits_controle[i]:
            return False
    return True
```

Commentaire : La méthode `count('1')` permet de compter facilement les occurrences. On compare le reste de la division par 2 (le modulo) avec le bit de contrôle correspondant. Si une seule parité ne correspond pas, la trame est invalide.

Question 3 Exécuter le fichier `analyse.py`, identifier les erreurs qui pourraient survenir.

Lors de l'exécution, deux problèmes majeurs peuvent survenir :

1. **Erreur de formatage (IndexError) :** Dans le fichier `data.txt`, la ligne 122 possède 41 bits au lieu de 40. Cela risque de décaler les découpages (slices) ou de provoquer une erreur si le code s'attend à une longueur fixe.
2. **Erreurs de mesure :** Certaines températures décodées peuvent être inférieures à -10°C, ce qui est considéré comme une erreur selon l'énoncé, ou certains codes BCD pourraient être invalides (ex: 1111 en binaire vaut 15, ce qui n'est pas un chiffre décimal entre 0 et 9).

Commentaire : Un programme de traitement de données doit toujours anticiper que le fichier source peut contenir des "bruits" ou des erreurs de saisie/transmission.

Question 4 Proposer des corrections de la classe *Transmission* permettant de la rendre plus robuste afin de pouvoir effectuer l'analyse et commenter l'affichage obtenu.

Modifications à apporter dans Transmission

```
def decoder(self):
    # On vérifie d'abord la longueur de la trame
    if len(self._trame) != 40:
        return
    self.decoder_id()
    self.decoder_temperature()
    self.decoder_humidite()

def est_valide(self):
    if len(self._trame) != 40:
        return False

    # Vérification de la parité (code de la Q2)
    # ... (voir Q2) ...

    # Ajout d'un contrôle sur la cohérence des données
    if self._temperature is not None and self._temperature < -10:
        return False

    return True
```

Commentaire : La robustesse consiste ici à ignorer les trames malformées (trop courtes/longues) ou physiquement impossibles (température trop basse) pour ne pas fausser les statistiques de l'analyse finale.