EXERCICE 3 (8 points)

Cet exercice porte sur la programmation Python (listes…), le traitement de données en table et les bases de données relationnelles.

Les deux parties sont indépendantes.

Partie A

Un **trail** est une course à pied en milieu naturel.

Une **trace** (ou *trace GPS*) est une liste d'au moins deux positions géographiques représentant un parcours.

Dans cet exercice, on souhaite réaliser un programme qui produit des statistiques intéressantes à partir d'une trace de *trail*.

La trace d'une course est stockée dans un fichier au format csv. Chaque ligne du fichier contient, dans cet ordre, la latitude en degrés Nord, la longitude en degrés Est (tous deux des nombres à virgule) puis l'altitude en mètres, et enfin une valeur d'horodatage ou *timestamp* (tous deux des entiers).

Voici un extrait typique de fichier csv :

```
46.03494, 7.09953, 1457, 1689008478
46.03522, 7.09947, 1447, 1689008481
46.03588, 7.09885, 1440, 1689008484
46.03635, 7.09819, 1439, 1689008487
46.03695, 7.09812, 1433, 1689008490
...
```

La lecture du fichier est effectuée par le programme Python suivant (il n'est pas nécessaire de comprendre ce code pour pouvoir faire le sujet) :

```
import csv
def lecture_trace_csv(filename):
    """
    Prend une chaîne de caractère (nom du fichier) en entrée
    et renvoie une liste contenant les données lues
    """
    tab_trace = []
    with open(filename, 'r') as f:
        csvreader = csv.reader(f, delimiter=',')
        for ligne in csvreader:
            tab_trace.append(ligne)
    return tab_trace
```

Ce code peut être utilisé ainsi :

```
>>> tab_trace = lecture_trace_csv('track.csv')
>>> print(tab_trace)
 [['46.03494', '7.09953', '1457', '1689008478'],
  ['46.03522', '7.09947', '1447', '1689008481'],
```

```
 ['46.03588', '7.09885', '1440', '1689008484'],
 ['46.03635', '7.09819', '1439', '1689008487'],
 ['46.03695', '7.09812', '1433', '1689008490']]
```

La variable `tab_trace` est donc une liste contenant des listes de quatre chaînes de caractères.

1. L'altitude du premier point est `'1457'`. Cette chaîne peut être affichée ainsi : `print(tab_trace[0][2])`. Donner l'instruction permettant d'afficher la longitude du dernier point uniquement.

Pour réaliser des calculs sur les données de la trace, il est nécessaire de convertir les différentes valeurs en nombres (`float` pour la latitude et la longitude et `int` pour l'altitude et le *timestamp*).

Pour cela, on écrit une fonction `creation_trace`, prenant en paramètre une liste similaire à `tab_trace` et renvoyant une liste de dictionnaires, chaque dictionnaire contenant une position horodatée (avec les quatre clés `lat`, `lon`, `alt`, `tsp`) :

```
>>> trace = creation_trace(tab_trace)
>>> for k in range(4):
        print(trace[k])
{'lat': 46.03494, 'lon': 7.09953, 'alt': 1457, 'tsp': 1689008478},
{'lat': 46.03522, 'lon': 7.09947, 'alt': 1447, 'tsp': 1689008481},
{'lat': 46.03588, 'lon': 7.09885, 'alt': 1440, 'tsp': 1689008484},
{'lat': 46.03635, 'lon': 7.09819, 'alt': 1439, 'tsp': 1689008487}
```

Noter que dans les dictionnaires, la latitude et la longitude sont des nombres à virgule flottante, alors que l'altitude et le timestamp sont des entiers.

2. Compléter les lignes 6 à 8 de la fonction `creation_trace` ci-après.

```
 1 def creation_trace(tab_trace):
 2     trace = []
 3     for enregistrement in tab_trace:
 4         pt = {}
 5         pt['lat'] = float(enregistrement[0])
 6         pt['lon'] = ...
 7         ...
 8         ...
 9         trace.append(pt)
10     return trace
```

Une nouvelle fonction doit prendre en paramètre une liste renvoyée par `creation_trace` et renvoyer une liste contenant les altitudes successives. Elle devra satisfaire l'assertion suivante (sur les données d'exemple) :

```
# Calcul des altitudes pour les 4 premiers éléments de trace
seulement
>>> alti = valeurs_altitude(trace)
>>> assert alti[0] == 1457
>>> assert alti[1] == 1447
```

```
>>> assert alti[2] == 1440
>>> assert alti[3] == 1439
```

3. Compléter la fonction suivante, qui prend en paramètre une trace et renvoie la liste des altitudes successives :

```
1 def valeurs_altitude(trace):
2     # Compléter avec autant de lignes que nécessaire
3     ...
4     ...
```

On appelle **segment** d'une trace *t* tout couple *(p1, p2)* où *p1* et *p2* sont 2 points consécutifs de *t*. Ainsi, les *n* points d'une trace déterminent *n - 1* segments.

Le **dénivelé d'un segment** est la différence d'altitude entre le point final d'un segment et le point initial.

On appelle **dénivelé cumulé positif** d'une trace des dénivelés pour les segments dont le dénivelé est positif et **dénivelé cumulé négatif** la somme des valeurs absolues des segments dont le dénivelé est négatif (voir Figure 1).

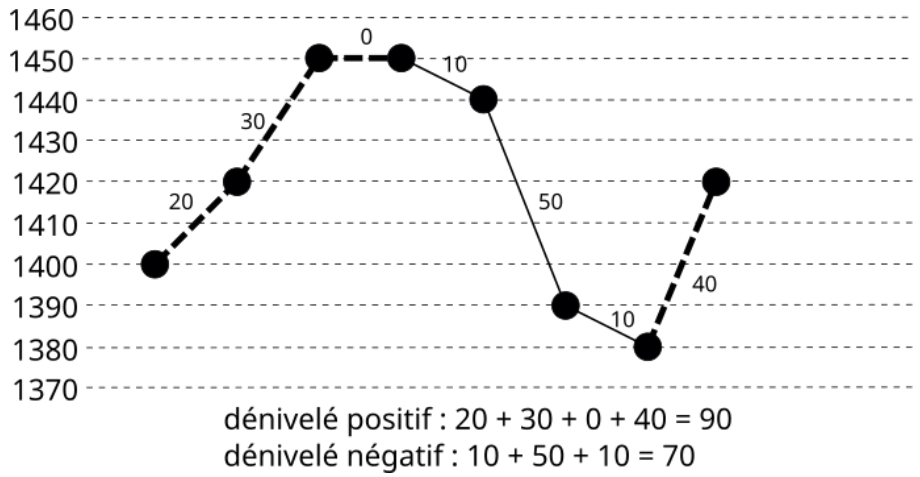

Figure 1. Calcul des dénivelés cumulés positif (pointillés épais) et négatif (traits pleins fins).

4. Donner le code de la fonction `denivele` qui prend en paramètre une liste d'altitudes et renvoie un tuple contenant le dénivelé cumulé positif et le dénivelé cumulé négatif, qui sont tous deux des nombres positifs :

```
>>> denivele([1400, 1420, 1450, 1450, 1440, 1390, 1380, 1420])
 (90, 70)
```

Les valeurs des horodatages (*timestamps*) sont des valeurs en secondes, exprimées à partir d'une date référence (1er janvier 1970). La différence entre deux horodatages donne donc le temps écoulé entre les deux mesures en secondes.

Afin de refléter la difficulté d'un parcours, on souhaite disposer d'une nouvelle fonction qui prendra en paramètre une trace (liste renvoyée par `creation_trace`) et renverra

un tuple de trois valeurs entières contenant : le nombre de secondes d'ascension, le nombre de secondes de descente, et le nombre de secondes sur plat, dans cet ordre.

Cette fonction pourra être utilisée ainsi (elle indique ici 190 secondes de montée, 233 secondes de descente et 22 secondes de plat) :

```
>>> asc_desc(trace)
(190, 233, 22)
```

5. Donner le code de la fonction `asc_desc`.

Soient deux points A et B définis par leur latitude et leur longitude. La distance qui les sépare à la surface du globe (sans tenir compte de l'altitude) est donnée par la formule :

$$d = R \times \arccos\big(\sin(lat_A) \times \sin(lat_B) + \cos(lat_A) \times \cos(lat_B) \times \cos(long_A - long_B)\big)$$

avec R le rayon de la Terre en mètres (on pourra prendre 6371000 m), tous les angles exprimés en radians et $\arccos$ la fonction *arccosinus* (fonction réciproque de *cosinus*).

6. Écrire une fonction nommée `distance(p1: dict, p2: dict)` qui prend en paramètre deux *points* (deux dictionnaires donc) et renvoie la distance qui les sépare en mètres.

   Les fonctions suivantes, du module `math` peuvent être utiles :

   – `math.cos(a)` : renvoie le cosinus de l'angle `a` exprimé en radians
   – `math.sin(a)` : renvoie le cosinus de l'angle `a` exprimé en radians
   – `math.acos(v)` : renvoie l'arc cosinus de `v`, en radians
   – `math.radians(a)` : renvoie la valeur en radians d'un angle `a` donné en degrés

7. En utilisant le théorème de Pythagore, proposer le code d'une fonction `distance_precise(p1: dict, p2: dict)` qui tient en plus compte de l'altitude des deux points pour le calcul de la distance (voir Figure 2).

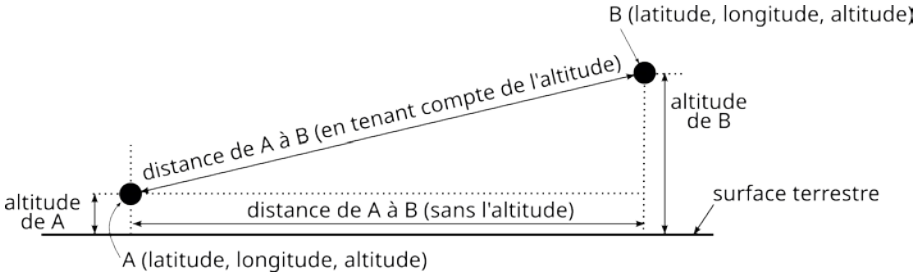

Figure 2. Calcul de la distance en tenant compte de l'altitude.

Partie B

Des fichiers `csv` et les différentes fonctions écrites précédemment sont utilisés pour alimenter une base de données relationnelle. Cette base ne contient qu'une relation, nommée `trails`, dont voici un échantillon (les dénivelés sont en mètres, la distance est en kilomètres, et le temps est en heures) :

Relation `trails`					
`id`	`lieu`	`dist`	`temps`	`denivele_p`	`denivele_n`
1	Mont Blanc	21.3	3	1919	1898
5	Saint Pardoux	24	2	40	40
2	Mont Blanc	171	30	10000	10000
6	Ile de Ré	92	7.5	320	320
…	…	…	…	…	…

8. Donner le résultat de la requête suivante dans le cas où la relation ne contient que les quatre enregistrements ci-dessus :

```
SELECT lieu, dist FROM trails WHERE denivele_p > 1000;
```

9. Donner la requête SQL permettant d'obtenir uniquement les `id` et les lieux (colonne `lieu`) des *trails* dont la distance dépasse 50 km.

10. On suppose que si les dénivelés positifs et négatifs sont égaux, alors la course est une **balade parfaite**. En utilisant cette idée, donner une requête SQL permettant d'afficher le lieu et la distance, et uniquement cela, de toutes les courses **qui ne sont pas** des balades parfaites.

11. Donner la requête SQL affichant la distance cumulée de tous les *trails* dont le dénivelé cumulé positif est supérieur à 1000 m (sur l'échantillon de tableau ci-dessus, la réponse serait 192.3).

N.B. : on rappelle que si `c` est une colonne d'une relation contenant des données numériques, la requête suivante : `SELECT sum(c) FROM rel` renvoie la somme des valeurs figurant dans la colonne `c` de la relation `rel`.