Dans le fichier python fourni avec ce TD, on utilise des *annotations de type* : pour chaque argument de chaque fonction, on indique en plus le type attendu (après le symbole :). Le type de la valeur de retour est également donné (après la flèche)

Par exemple, tiré du TD précédent :

```
def polygone(n : int) -> None:
    """
    Trace un polygone régulier à n côtés inscrit dans le cercle
    trigonométrique si n >= 3. Ne fait rien si n < 3

    Pas de valeur de retour.
    """
```

qui indique que la fonction polygone attend un entier comme argument et ne retourne rien (`None`)

I Rendu de monnaie

1.1 Présentation du problème

On souhaite obtenir une somme S donnée (la somme à rendre) en utilisant seulement des pièces d'une certaine valeur (les pièces disponibles dans la caisse, et on suppose que pour chaque valeur de pièce la quantité n'est pas limitée).
Par exemple, pour la somme S $= 15$ à rendre en disposant de pièces de $10, 2, 1$ on peut rendre $7 \times 2 + 1 \times 1$ (8 pièces) ou $1 \times 10 + 2 \times 2 + 1 \times 1$ (4 pièces).
On se fixe comme but de rendre le moins de pièces possible.

1.2 Numérisation des données

On considère que `S` est un nombre entier, ie de type `int` (quitte à compter la valeur des pièces en centimes) et on dispose d'une liste `L` contenant les valeurs des pièces disponibles.

```
S = 15
L = [10, 2, 1]
```

dans l'exemple précédent.
Hypothèse : on suppose que la liste `L` contient toujours la valeur 1 ce qui permet d'assurer que le rendu est toujours possible.
La valeur à calculer sera une liste contenant des éléments de la forme `[n, v]` où `n` est un entier et `v` la valeur d'une pièce disponible. Ainsi les deux exemples de rendus précédents seront représentés par

```
[[7, 2], [1, 1]] # premier exemple
[[1, 10], [2, 2], [1, 1]]
```

Exercice 1
Compléter la fonction `nombre_pieces` du fichier **td5.py** : cette fonction prend comme argument un rendu de monnaie au format précédent et retourne le nombre total de pièces correspondant.
Tester avec les deux listes précédentes.

1.3 Algorithme glouton

La procédure choisie est la suivante :

- On choisi la pièce de plus grande valeur possible (inférieure à la somme à rendre) et on compte le nombre de pièces de cette valeur que l'on peut rendre sans dépasser la somme.
- On recommence avec la somme restant à rendre.

Cet algorithme sera implémenté dans l'exercice 4

Exercice 2
Pour une somme S et une pièce de valeur $v <$ S, exprimer sous forme d'une instruction python le plus grand entier k tel que $k \times v <=$ S.

```
k =
```

A faire sur feuille.

Exercice 3
Compléter la fonction `plus_grande_piece(S, L)` qui prend comme argument une somme `S` > 0 entière et une liste `L` (contenant la valeur 1) triée par ordre décroissant, et retournant la plus grande valeur de `L` qui soit inférieure ou égale à `S`.

Exercice 4
Implémenter l'algorithme précédent dans la fonction `rendu_glouton(S, L)` (même signification des arguments)
Pour l'exemple donné de `S` et `L` on doit obtenir le deuxième rendu proposé. Il s'agit d'ailleurs d'un rendu optimal.

Exercice 5
Dans le fichier **td5.py**, une fonction `tous_les_rendus(S, L)` est fournie. Elle retourne, pour une somme S et une liste de valeurs L triée et contenant 1, la liste de tous les rendus possibles (un rendu est défini plus haut, c'est une liste de liste). Ainsi la valeur de retour est une liste de liste de liste.
En utilisant cette fonction et la fonction `nombre_pieces`, compléter la fonction `rendu_optimal`.
Pour comparer les performances, on pourra taper dans la console

```
%timeit rendu_optimal(48, [10, 5, 2, 1])
%timeit rendu_glouton(48, [10, 5, 2, 1])
```

1.4 Algorithme non-optimal

Exercice 6
Trouver un exemple de valeur d'une liste `L` telle que l'algorithme glouton ne renvoie pas le meilleur rendu possible (c'est à dire qu'on peut trouver un rendu utilisant strictement moins de pièces). On utilisera la somme S = 15.

II Attribution de salles de cours

Le problème est le suivant : on dispose d'une liste de cours définis par leurs heures de début et de fin (une liste de taille 2, contenant des nombres) et on se demande combien de salles au minimum il faut utiliser pour assurer tous ces cours.
On représente une salle par la liste des cours qui lui sont déjà attribués (ie une liste d'éléments de la forme `[d, f]`, début, fin) et on crée la liste des salles nécessaires pas à pas.

Exercice 7
Compléter la fonction `salle_libre` prenant comme argument un cours (de la forme `[d, f]` où `d, f` sont des entiers) et la liste des salles déjà créées (et contenant les cours déjà attribués) et qui retourne soit i, l'indice minimal d'une salle libre pour accueillir ce cours, soit `None` si aucune salle ne convient.
Il s'agit d'un algorithme glouton, car on choisit la première salle libre sans chercher à optimiser le choix d'une salle possible.

Exercice 8
Compléter la fonction `attribue_cours(liste_cours)` qui prend une liste de cours comme argument et retourne une liste de salle attribuées.

Cette fois, on peut prouver que si la liste des cours est triée par ordre croissant de l'heure de fin des cours, alors l'algorithme glouton proposé est optimal