## Exercice 1. MyLittleFacebook

Nous poursuivons ici l'exemple vu au cours:

```python
# MyLittleFacebook
friendships: dict[str, set[str]] = {}

def add_friends(name1: str, name2: str) -> None:
    if name1 in friendships:
        friendships[name1].add(name2)
    else:
        friendships[name1] = { name2 }
    if name2 in friendships:
        friendships[name2].add(name1)
    else:
        friendships[name2] = { name1 }

add_friends("Alex", "Victor")
add_friends("Alex", "Emelyne")
add_friends("Alex", "Emelyne")
add_friends("Emelyne", "Rose")
```

(a) Ajoutez sous ce code une fonction **`known_people()`** qui nous donne, dans l'ordre alphabétique, tous les noms pour lesquels le dictionnaire contient des relations d'amitié.

*Indices:* Quel devra être son type de retour? Comment obtenir l'ensemble des clés connues par un dictionnaire? Vous pouvez utiliser la fonction **`sorted()`** de Python.

(b) Ajoutez une fonction **`friends_of()`**, qui prend un paramètre **`name`** de type **`str`** et qui renvoie l'ensemble des amis liés à ce nom. Prenez soin de retourner un set vide lorsque le dictionnaire n'a aucune info sur la personne demandée.

Vous pouvez tester avec ceci:

```python
print(friends_of("Emelyne")) # {'Rose', 'Alex'}
print(friends_of("Rachel"))  # set()
```

(c) Ajoutez une fonction **`are_friends()`**, qui indique si les deux noms passés comme paramètres sont déclarés comme amis. Quel est le type de retour?

Testez avec ceci:

```python
print(are_friends("Alex", "Victor")) # True
print(are_friends("Victor", "Alex")) # True
print(are_friends("Alex", "Rose"))   # False
print(are_friends("Alex", "Rachel")) # False
print(are_friends("Rachel", "Alex")) # False
```

(d) Si l'on passe par la méthode **`add_friends()`**, la modification du dictionnaire est toujours symétrique par rapport aux deux noms passés en paramètre. Mais on pourrait faire exprès de créer une incohérence en allant modifier le dictionnaire de manière asymétrique:

```python
friendships["Alex"].add("Rachel") # Modification asymétrique
```

Pour détecter cette situation problématique, ajoutez une fonction **`is_data_consistent()`**, qui dira si oui ou non l'ensemble du contenu du dictionnaire est cohérent, c'est-à-dire qui renverra **`True`** lorsque $A$ fait partie des amis de $B$ si et seulement si $B$ fait partie des amis de $A$, et **`False`** sinon.

*Indice:* vous pouvez itérer sur chaque clé **`k`** liée à sa valeur **`v`** dans un dictionnaire **`d`** comme ceci:

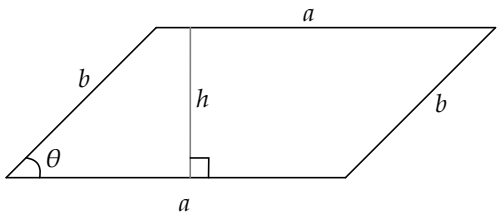
```python
for k, v in d.items():
    ...
```

Testez avec ceci:

```python
print(f"Consistent before modification? {is_data_consistent()}")  # True
friendships["Alex"].add("Rachel") # Modification asymétrique
print(f"Consistent after modification? {is_data_consistent()}")   # False
```

## Exercice 2. Une classe simple

(a) Dans un nouveau fichier, déclarez une classe **`Parallelogram`** pour modéliser un parallélogramme. Celui-ci est défini par 3 **`float`**s: les longueurs $a$ et $b$, et l'angle $\theta$ comme représenté ci-dessous.




Utilisez pour cela une **`dataclass`** comme vu au cours, avec 3 champs. Vous devriez pouvoir créer un parallélogramme ainsi (où l'angle est exprimé en degrés):

```
p = Parallelogram(a=20, b=10, theta=45)
```

(b) Ajoutez les méthodes **`perimeter`**, **`height`** et **`area`** pour calculer, respectivement, le périmètre, la hauteur ($h$ dans la figure ci-dessus) et l'aire du parallélogramme. Ce code devrait fonctionner:

```
p = Parallelogram(a=20, b=10, theta=45)
print(p)             # Parallelogram(a=20, b=10, theta=45)
print(p.perimeter()) # 60
print(p.height())    # about 7.07
print(p.area())      # about 141.4
```

## Exercice 3. Compréhension de code et imports

Nous allons explorer en Python la liste des marées noires qui se sont produites depuis le début du XX$^{e}$ siècle. Pour ce faire, créez un nouveau fichier **`oilspills.py`** et collez-y le code de départ de la page Moodle.

(a) Explorez le code de **`oilspills`** tel que vous l'avez importé:

- la classe **`OilSpillEvent`** modélise les données d'une marée noire. En regardant les champs de cette classe, vous pouvez facilement repérer par quels éléments un événement est défini et quels sont les types de ces champs.
- la variable **`all_events`** est une liste où chaque élément est un objet de type **`OilSpillEvents`**.

(b) Créez un nouveau fichier **`.py`** et ajoutez-y un import approprié de manière à pouvoir y faire référence à la liste **`all_events`** du fichier **`oilspills.py`**. Affichez le nombre d'événements que vous trouvez dans cette liste selon ce format:
**`Il y a 120 événements.`**

(c) Avec une boucle **`for-in`**, affichez tous les éléments de **`all_events`** un par un.

*Le dernier exercice est sur la page suivante.*

## Exercice 4. Statistiques et filtres

(a) Dans votre fichier qui n'est pas **`oilspills.py`**, écrivez une fonction

**`def print_statistics(events: list[OilSpillEvent]) -> None`**

qui calcule la quantité totale déversée (minimum et maximum) par les événements fournis en paramètre et la période sur laquelle ça s'est passé (dates de début et de fin). À la fin, cette fonction doit afficher ces statistiques, selon les calculs faits. Par exemple, si on lui passe tous les événements connus, on devrait voir ceci:

**`Entre le 14.12.1907 et le 01.07.2011, 120 événements ont causé le déversement d'entre 6460121 et 7690591 barils de brut.`**

Votre code ne doit pas partir du principe que l'array **`events`** est trié chronologiquement.

*Indice 1:* pour comparer deux dates, vous pouvez directement utiliser les opérateurs "**`<`**" et "**`>`**".
*Indice 2:* pour formater une date selon le modèle «jour.mois.année», vous pouvez appeler la méthode **`strftime`** sur un objet de type **`datetime`** comme ceux stockés dans les champs des objets de type **`OilSpillEvent`**. Cette méthode demande un argument de type **`str`** qui dénote le format de date à utiliser pour la conversion. Ici, par exemple, on utilise **`"%d.%m.%Y"`**:

```
1  my_date = ... # de type datetime
2  my_date_as_string = my_date.strftime("%d.%m.%Y")
```

(b) Écrivez une fonction

**`def filter_by_country(country: str, events: list[OilSpillEvent]) -> list[OilSpillEvent]`**

qui retourne une nouvelle liste avec uniquement les événements de l'array **`events`** qui se sont produits dans le pays donné avec le paramètre **`country`**. Testez votre fonction avec les pays **`"United States"`** et **`"Australia"`**. Utilisez la valeur de retour de **`filter_by_country`** pour afficher des statistiques avec **`print_statistics`** sur les éléments filtrés.

Pour **`"United States"`**, vous devriez avoir:
**`Entre le 14.03.1910 et le 01.07.2011, 51 événements ont causé le déversement d'entre 2137916 et 2391355 barils de brut.`**

Pour **`"Australia"`**, vous devriez avoir:
**`Entre le 21.07.1991 et le 03.04.2010, 4 événements ont causé le déversement d'entre 21513 et 47544 barils de brut.`**