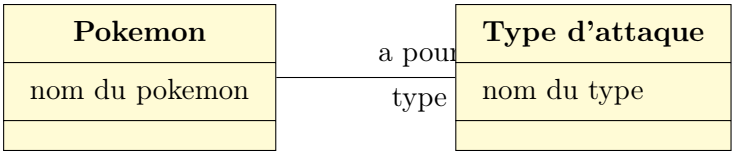
R1.01 Initiation au développement
Feuille de TP n°10
Représentation de la mémoire, dictionnaires, matrices

Objectifs de la feuille

- Consolider les acquis
- Coder une deuxième API permettant de manipuler des matrices

Exercice 1 *Choix de modélisation et complexité*

Voici le MCD d'un pokedex



Le pokedex de Anakin

Le pokedex de Romain

Anakin a trouvé trois façons différentes de modéliser ces données en python.

```python
# modélisation 1
pokedex_anakin_v1 = {
    ('Carmache', 'Dragon'), ('Carmache', 'Sol'), ('Colimucus', 'Dragon'),
    ('Palkia', 'Dragon'), ('Palkia', 'Eau')}

# modélisation 2
pokedex_anakin_v2 = {
    'Carmache': {'Dragon','Sol'}, 'Colimucus': {'Dragon'},
    'Palkia': {'Dragon', 'Eau'}}

# modélisation 3
pokedex_anakin_v3 = {
    'Dragon': {'Carmache','Colimucus', 'Palkia'},
    'Sol': {'Carmache'}, 'Eau': {'Palkia'}}
```

1.1 Préciser comment serait implémenté le pokedex de Romain dans chacune des trois versions.

1.2 En utilisant la modélisation n° 1, écrire le code des fonctions suivante et préciser leur complexité :

a) `appartient(nom_pokemon, pokedex)` qui indique si un pokemon dont on donne le nom appartient à un pokedex.

b) `toutes_les_attaques(pokedex)` qui renvoie l'ensemble des types d'attaque des pokemon du pokedex.

c) `nombre_de(type_attaque, pokedex)` qui renvoie le nombre de pokemons qui possède le type d'attaque donnée dans le pokedex passé en paramètre.

d) `attaque_preferee(pokedex)` qui renvoie le type d'attaque qui est la plus fréquente dans le pokedex passé en paramètre.

1.3 Même questions en utilisant la modélisation n° 2. N'oubliez pas de préciser à chaque fois la complexité de vos fonctions.

1.4 Même question en utilisant la modélisation n° 3.

tableau récapitulatif

Complexités	`appartient`	`toutes_les_attaques`	`nombre_de`	`attaque_preferee`
Avec la version 1				
Avec la version 2				
Avec la version 3				

1.5 Compléter les codes des deux fonctions suivantes qui permettent de transformer une version en une autre

```
def v1_to_v2(pokedex_v1):
    """ param: prend en paramètre un pokedex version 1
        renvoie le même pokedex mais en version 2 """
    ...

def v2_to_v3(pokedex_v2):
    """ param: prend en paramètre un pokedex version2
        renvoie le même pokedex mais en version3 """
    ...
```

```
assert v1_to_v2(pokedex_anakin_v1) == pokedex_anakin_v2
assert v2_to_v3(pokedex_anakin_v2) == pokedex_anakin_v3
```

Exercice 2 *Écosystème*

Un écosystème est représenté par un dictionnaire dont les clefs sont le nom des espèces présentes et les valeurs le nom de l'espèce qui constitue leur alimentation.
Par exemple :

```
ecosysteme_1 = {'Loup': 'Mouton', 'Mouton':'Herbe', 'Dragon':'Lion',
                'Lion':'Lapin', 'Herbe':None, 'Lapin':'Carotte',
                 'Requin':'Surfer'}
ecosysteme_2 = {'Renard':'Poule', 'Poule':'Ver de terre',
                'Ver de terre':'Renard', 'Ours':'Renard' }
```

Dans l'écosystème1, le lion a besoin de lapin pour survivre alors que l'herbe n'a besoin de rien.

2.1 Écrire une fonction `extinction_immediate(ecosysteme, animal)` qui prend en paramètre un écosystème et un animal et qui indique si l'animal s'éteint immédiatement faute de nourriture dans cet ecosystème.

Par exemple, dans l'écosystème1, Le lapin et le requin s'éteignent immédiatement : ils ne peuvent pas survivre car il n'y a ni carotte ni surfer à se mettre sous la dent. En revanche, dans l'écosystème2, toutes les espèces survivent : elles ont toutes de quoi se sustenter.

```
assert extinction_immediate(ecosysteme_1, 'Lapin')
assert extinction_immediate(ecosysteme_1, 'Requin')
assert not extinction_immédiate(ecosysteme_2, 'Poule')
```

2.2 Dans l'écosystème1, comme le lapin ne peut pas survivre, le lion disparaît également par manque de lapin. Et comme le lion disparaît, le dragon disparaît à son tour.

Écrire une fonction `en_voie_disparition(ecosysteme, animal)` qui prend en paramètre un écosystème et un animal et qui renvoie True si l'animal est en voie de disparition dans cet écosystème et False sinon.

Par exemple

```
assert en_voie_disparition(ecosysteme_1, 'Dragon')
assert en_voie_disparition(ecosysteme_1, 'Lapin')
assert not en_voie_disparition(ecosysteme_2, 'Poule')
```

Exercice 3 *Deuxième API pour manipuler les matrices*

Dans cet exercice, on reprend la modélisation n° 3 vue la semaine dernière (TD n° 9)

$$matrice1 = \begin{pmatrix} 10 & 11 & 12 & 13 \\ 14 & 15 & 16 & 17 \\ 18 & 19 & 20 & 21 \end{pmatrix} \qquad matrice2 = \begin{pmatrix} A & B & C \\ D & E & F \end{pmatrix} \qquad matrice3 = \begin{pmatrix} 2 & 7 & 6 \\ 9 & 5 & 1 \\ 4 & 3 & 8 \end{pmatrix}$$

```
matrice1 = [[10, 11, 12, 13], [14, 15, 16, 17], [18, 19, 20, 21]]

matrice2 = ...

matrice3 = ...
```

Vous devez écrire un fichier `API_matrice2.py` contenant le code des fonctions qui suivent. Pour les tests, utilisez le fichier `test_API_matrice.py` du TP de la semaine dernière en modifiant juste la ligne 4 :

```
import API_matrice2 as API
```

3.1 `matrice(nb_lignes, nb_colonnes, valeur_par_defaut=0)` qui crée une nouvelle matrice en mettant la valeur par défaut dans chacune de ses cases.
Il sera possible d'appeler cette fonction avec trois paramètres(`nb_lignes` , `nb_colonnes` et `taille_cellule`) ou avec deux paramètres seulement (`nb_lignes` et `nb_colonnes`). Dans ce cas, le paramètre `valeur_par_defaut` sera remplacé par la valeur `0` .

3.2 `get_nb_lignes(matrice)` permet de connaître le nombre de lignes d'une matrice

3.3 `get_nb_colonnes(matrice)` permet de connaître le nombre de colonnes d'une matrice

3.4 `get_val(matrice, ligne, colonne)` permet de connaître la valeur de l'élément de la matrice dont on connaît le numéro de ligne et le numéro de colonne.

3.5 `set_val(matrice, ligne, colonne, nouvelle_valeur)` permet de modifier la valeur de l'élément qui se trouve à la ligne et à la colonne spécifiées. Cet élément prend alors la valeur `nouvelle_valeur`)

3.6 `get_ligne(matrice, ligne)` qui renvoie sous la forme d'une liste la ligne de la matrice dont le numéro est spécifié.

3.7 `get_colonne(matrice, colonne)` qui renvoie sous la forme d'une liste la colonne de la matrice dont le numéro est spécifié.

Exercice 4 *Petites bêtes*

On donne une liste contenant des informations sur des pokemons. Ces informations sont données sous la forme d'un tuple (nom, familles, image).

Par exemple :

```
ma_liste_pokemon=[
  ('Bulbizarre', {'Plante', 'Poison'}, '001.png'),
  ('Herbizarre', {'Plante', 'Poison'}, '002.png'),
  ('Abo', {'Poison'}, '023.png'),
  ('Jungko', {'Plante'}, '254.png'), ...]
```

Ainsi, `('Bulbizarre', {'Plante', 'Poison'}, '001.png')` modélise un pokemon qui se nomme Bulbizarre. Il appartient aux familles Plante et Poison. L'image qui lui est associée est nommée `001.png` .

4.1 Écrire une fonction `pokemons_par_famille(liste_pokemon)` qui prend en paramètre une telle liste de pokémons et qui renvoie un dictionnaire dont les clefs sont les familles et chaque valeur l'ensemble des pokémons de cette famille.

```
assert pokemons_par_famille(liste_pokemon) == {
    'Plante':{'Bulbizarre', 'Herbizarre', 'Jungko', ...},
    'Poison':{'Bulbizarre', 'Herbizarre', 'Abo', ...},
    ... }
```

Exercice 5 *Quelques fonctions pour manipuler les matrices*

Compléter le fichier `utilitaires_matrice.py` en y intégrant le code des fonctions décrites dans la suite (ainsi que toutes celles qui vous sembleraient utiles).

Votre code devra rester fonctionnel si on remplace le ligne de code `import API_matrice1 as matrice` par la ligne `import API_matrice2 as matrice`

Vous penserez à écrire la documentation de chacune des fonctions et vous compléterez un fichier pour effectuer les tests.

5.1 `get_diagonale_principale(matrice)` qui renvoie sous la forme d'une liste la diagonale principale d'une matrice carrée.

5.2 `get_diagonale_secondaire(matrice)` qui renvoie sous la forme d'une liste la diagonale secondaire d'une matrice carrée.

5.3 `transpose(matrice)` qui renvoie la transposée d'une matrice.

5.4 `is_triangulaire_inf(matrice)` qui indique si une matrice est triangulaire inférieure.

5.5 `is_triangulaire_sup(matrice)` qui indique si une matrice est triangulaire supérieure.

5.6 `bloc(matrice, ligne, colonne, hauteur, largeur)` qui renvoie la sous-matrice de la matrice commençant à la ligne et colonne indiquées et dont les dimensions sont hauteur et largeur.

5.7 `somme(matrice1, matrice2)` qui renvoie la matrice résultante de la somme des deux matrices passées en paramètre.

5.8 `produit(matrice1, matrice2)` qui renvoie la matrice résultante u produit des deux matrices passées en paramètre.