YSINL2A1 : Initiation à la Programmation Orientée Objet (Python) – UNICAEN 2022 L1S2

Déroulement : le cours est dispensé sur 10 semaines comprenant chacune $2h$ de cours magistral, $1h$ de travail pratique d'application des notions vues en cours et $2h$ de travail pratique lié au projet fil rouge de l'année. Ce dernier sera d'abord très contraint puis deviendra plus libre à partir des dernières semaines. L'évaluation se fera sur un QCM de $1h30$ couvrant les cours magistraux, les travaux pratiques et le projet fil rouge (2/3 de la note globale) et le travail/rapport des fonctionnalités libres du projet fil rouge en trinôme (1/3 de la note globale).

Contenu : les deux premières semaines sont réservées aux rappels importants sur les principes de la programmation impérative en langage Python. Les semaines suivantes intégreront petit à petit différents concepts de la programmation orientée objet.

Recommandations cours magistraux : les cours magistraux sont indispensables à une bonne réussite à cette unité pour au moins deux raisons :

- 1/3 du QCM peut porter sur des questions abordées uniquement lors de ces cours
- les TP peuvent être volontairement longs et sans corrections documentées. En revanche une correction du début ou des points clés de la plupart des TP sera faite à travers les nombreux exemples fournis sur ordinateur ou au tableau lors du cours de la semaine correspondante (plutôt qu'avec des diapositives).

Recommandations TP :

- ils seront souvent fournis sous la forme de fichiers `.py` et des ressources éventuelles associées. Le fichier `.py` contient les signatures des fonctions et les consignes en *docstring*). Dans ce cas, il est interdit de modifier les signatures, seuls les corps des fonctions sont à écrire.
- le main des fichiers de TP seront souvent sous la forme de variables et d'assertions de tests à passer associés à chaque fonction à réaliser. Vous ne devez pas modifier les tests mais le corps de vos fonctions afin que ceux-ci ne soient plus bloquants ! Vous pouvez également observer les tests ou rajouter les vôtres pour comprendre la provenance de vos erreurs.
- Les TP non terminés pendant les horaires réservés devraient être terminés sur son temps personnel avant le début du suivant (en particulier les sections en rouge repérables comme plus difficiles).

Recommandations projet fil rouge : Il s'agira de construire une application de simulation d'un écosystème permettant de placer, gérer et visualiser des populations de proies et de prédateurs.

- La structure sera imposée et construite au fur et a mesure des TP.
- Les derniers TP ainsi que votre temps personnel seront consacrés à l'ajout de trois extensions libres, par trinôme d'étudiants. Une première version du rapport sera remise ne contenant que le cahier des charges fonctionnel, le diagramme des classes et la répartition du travail prévu par étudiant.
- Le rapport final complétera le premier en précisant le diagramme des classes de la réalisation finale, la manière dont le groupe s'est réparti réellement le travail et les difficultés rencontrées (différences entre le cahier des charges fonctionnel et les réalisations effectives). Un bon travail doit avoir produit environ 80% des fonctionnalités prévues. Moins signalerait un manque de travail, et plus un manque d'ambition ;)
- Le code sera rendu sous la forme d'un dépôt `GIT` (le nombre, la régularité et la qualité des commentaires des `push` par étudiants feront partie de la grille d'évaluation)

Enseignants : F. Maurel - J.-M. Le Bars - C. Alec - J.-M. Lecarpentier - S. Schupp - Y. Jacquier - L. Kastner

Programme prévisionnel - vacances : semaine du 13 février

Semaine du 9/1/2022 : RAPPELS
- CM01 : itératif/récursif, référence/valeur, mutable/non mutable - séquences - compréhension - fichier
- TP01 : bases - string
- TP02 : fichiers - list - tuple

Semaine du 16/1/2022 : RAPPELS
- CM02 : Ensembles - compréhension de dictionnaires - grilles
- TP03 : dict (histogrammes de caractère)
- TP04 : dict - set (génération d'un nuage de mots)

Semaine du 23/1/2022
- CM03 : Classe et objet - UML - structure d'un POO - versionnage (GIT)
- TP05 : module grid_manager.py (impératif : fonctions de gestion d'un tore)
- TP06 : Passage à l'OO : Grid - Grid[list][dict] - fichier de configuration - GIT

Semaine du 30/1/2022
- CM04 : privé/brouillé/public, instance/classe/static, UML, getter/setter
- TP07 : Exercices d'application du CM04
- TP08 : Grid[list][Animal] avec déplacements - Affichage matplotlib

Semaine du 6/2/2022
- CM05 : Agrégation, Héritage, redéfinition/surcharge - UML
- TP09 : Exercices d'application du CM05
- TP10 : Hiérarchie d'animaux - GRID[Animal] avec reproduction et morts

Semaine du 20/2/2022
- CM06 : Méthodes spéciales (built-in fonctions et opérateurs)
- TP11 : Exercices d'application du CM06
- TP12 : Intégration alphanumérique projet fil rouge écosystème (repr, add)

Semaine du 27/2/2022
- CM07 : Classes abstraites - Interfaces - UML - Multi-héritage
- TP13 : Exercices d'application du CM07
- TP14 : Extension projet fil rouge écosystème (déplacement, reproduction, mortel - Proie/Prédateur)

Semaine du 6/3/2022
- CM08 : Bibliothèques graphiques Matplotlib, vs. Tkinter vs. Pygame
- TP15 : Exercices d'application du CM08
- TP16 : Extension projet fil rouge écosystème (GUI Tkinter, animation Pygame, stats Matplotlib)

Semaine du 13/3/2022
- CM09 : Méthodes spéciales avancées (non mutables, __new__, itérateurs)
- TP17 : Exercices d'application du CM09
- TP18 : Projet fil rouge écosystème libre

Semaine du 20/3/2022
- CM10 : Évolutions projet écosystème - Docstring - Pydoc
- TP19 : Exercices d'application du CM10
- TP20 : Projet fil rouge écosystème libre

1 CM01 (9/1/2023)

Notions du cours à connaître pour les TP01 et TP02

- bonnes pratiques de nommage (convention *snake case*, constantes *vs.* variables, espaces, nomenclature)
- les différents types de séquences Python selon qu'ils sont mutables (`list`) ou non (`str`, `tuple`).
- fonctions *built-in* de base (`print`, `range`, `len`, `sum`, `min`, `max`, `all`, `any`) et de *cast* (`bool`, `int`, `str`, `list`) *vs.* fonctions associées à une variable par la notation pointée (`.upper`, `.index`, `.append`, `.extend`, `.join`)
- programmation itérative *vs.* récursive
- passage des paramètres à une fonction par valeur *vs.* par référence
- les notions de *mapping* et de filtrage appliquées à la définition en compréhension de listes Python
- les fichiers textes et leur accès en Python

1.1 TP01 - bases - string (1h)

`S01_TP01_template.py` : fichier *template* à compléter ; il s'agit de remplacer l'instruction `pass` dans le corps des fonctions sans modifications des signatures ou des tests. Les annotations de typage autorisées depuis la version 3.5 du langage Python sont informatives, optionnelles et dans tous les cas non prises en compte par l'interpréteur Python. Elles sont utilisées dans le texte des questions pour les éclairer mais ne doivent pas être précisées dans votre code. Le lancement du programme s'arrête sur le premier test non passé. Si tous les tests sont validés, le message *"Tests all OK"* s'affiche.

1.1.1 Exercice

`are_chars(chars: str, string: str)` → `bool` | Retourne `True` si tous les caractères de la chaîne `chars` apparaissent au moins une fois dans la chaîne `string`. `False` sinon.

```
>>> are_chars('test', 'est'), are_chars('tester', 'est')
(True, False)
```

1.1.2 Exercice

Les 3 fonctions suivantes simulent un brin d'ADN sous la forme d'une chaîne de caractères combinant les lettres A, T, G et C pour représenter les bases susceptibles de le composer. L'Adénine (A) est la base complémentaire de la Thymine (T) et la Guanine (G) est la complémentaire de la Cytozine (C). Les bases ont également une masse molaire :

- A pèse 135 g/mol
- T pèse 126 g/mol
- G pèse 151 g/mol
- C pèse 111 g/mol

1. `is_dna(dna: str)` → `bool` | Retourne `True` si le brin `dna` contient uniquement des bases A, T, G ou C (et au moins une). `False` sinon. Il faudra utiliser la fonction `are_chars`.

```
>>> is_dna('GTATTCTCA'), is_dna('GTAUTCTCA')
(True, False)
```

2. `get_molar_mass(dna: str)` → `int` | Retourne 0 si `dna` n'est pas un brin d'ADN. Sinon, retourne sa masse molaire. Il faudra utiliser la fonction `is_dna`.

```
>>> get_molar_mass('GTATTCTCA')
1147
```

3. `get_complementary(dna: str)` → `str` | Si `dna` est un brin, retourne son complémentaire. Sinon retourne None.

```
>>> get_complementary('GTATTCTCA'), get_complementary('GTAUTCTCA')
(CATAAGAGT, None)
```

1.1.3 Exercice

Les 4 fonctions suivantes permettent de jouer un peu avec les mots.

1. `get_first_deleted(char: char, string: str)` → `str` | Retourne la chaîne `string` amputée de la première occurrence du caractère `char`.

```
>>> get_first_deleted('r', "aeeigmnrrrstuwz")
"aeeigmnrrstuwz"
```

2. `is_scrabble(word: str, letters: str)` → `bool` | Retourne `True` si le mot `word` peut être construit comme au jeu du *Scrabble* à partir des lettres de la chaîne `letters` (les lettres répétées dans `word` seront dont également répétées au moins le même nombre de fois dans `letters`). `False` sinon. Il faudra obligatoirement utiliser `get_first_deleted`.

```
>>> is_scrabble("marguerites", "gewurztraminers"), is_scrabble("rose", "gewurztraminers")
(True, False)
```

3. `is_anagram(word1: str, word2: str)` → `bool` | Retourne `True` si `word1` et `word2` sont deux anagrammes. `False` sinon. Il faudra obligatoirement utiliser `is_scrabble`.

```
>>> is_anagram("gewurztraminers", "aeeigmnrrrstuwz")
True
```

4. `get_hamming_distance(word1: str, word2: str)` → `int` | Retourne la distance de Hamming entre `word1` et `word2` ou -1 si son calcul n'est pas possible. La distance de Hamming entre deux chaînes de même longueur correspond au nombre de positions auxquelles sont associées des caractères différents.

```
>>> get_hamming_distance("gewurztraminers", "aeeigmnrrrstuwz")
13
```

1.2 TP02 - fichiers - list - tuple (2h)

`resources.zip` : cette archive est à décompresser dans un répertoire `TP_POO`. Elle organise dans un répertoire `TEXTS` les ressources nécessaires à certains TP. En particulier le fichier `fr_long_dict_cleaned.txt` qui sera utilisé dans ce TP (il contient 242818 mots du français en majuscule et sans accents).

Plusieurs autres fichiers doivent être rajoutés à la racine du répertoire `TP_POO`.

`config.py` : un module à importer dans tout programme qui doit utiliser les ressources fournies par `resources.zip`. Il contient des variables globales pointant vers les différents répertoires (chemins relatifs au répertoire `TP_POO`) qui organisent les fichiers par thèmes (livres, dictionnaires, politiques, linguistiques...) :

```
- PATH_ALPHABET = 'TEXTS/IN/RESOURCES/CHARACTERS/'
- PATH_DICTIONARIES = 'TEXTS/IN/RESOURCES/WORDS/'
- PATH_BOOKS = 'TEXTS/IN/BOOKS/'
- PATH_ARTICLES = 'TEXTS/IN/ARTICLES/'
- PATH_DH = 'TEXTS/IN/POLITICAL/DH/'
- PATH_WISHES = 'TEXTS/IN/POLITICAL/WISHES/'
- PATH_OUT = 'TEXTS/OUT/'
```

`S01_TP01_template.py` : fichier *template* complété du TP précédent.

`S01_TP02_template.py` : fichier *template* à compléter pour réaliser ce TP. Les 3 premières lignes d'importation servent à récupérer (1) les variables globales du fichier `config.py`, (2) la fonction `get_hamming_distance` réalisée lors du TP précédent et (3) la fonction `perf_counter` du module `time` utile pour mesurer les performances de votre code. **Assurez-vous de respecter l'arborescence décrite ci-dessous et que les importations fonctionnent avant de commencer les exercices du TP**.

```
TP_POO
   |_____ TEXTS
   |          |_____ IN
   |          |       |_____ ARTICLES
   |          |       |              |_____ EN
   |          |       |              |_____ ES
   |          |       |              |_____ FR
   |          |       |_____ BOOKS
   |          |       |            |_____ EN
   |          |       |            |_____ FR
   |          |       |
   |          |       |_____ OTHERS
   |          |       |
   |          |       |_____ POLITICAL
   |          |       |               |_____ DH
   |          |       |               |_____ WISHES
   |          |       |
   |          |       |_____ RESOURCES
   |          |                       |_____ CHARACTERS
   |          |                       |_____ WORDS
   |          |                                  |_____ fr_long_dict_cleaned.txt
   |          |_____ OUT
   |
   |_____ config.py
   |_____ S01_TP01_template.py
   |_____ S01_TP02_template.py
                from config import *
                from S01_TP01_template import get_hamming_distance
                from time import perf_counter
```

1.2.1 Exercice

Les 5 fonctions suivantes ont pour finalité de construire des « échelles » entre deux mots donnés. Pour cela il s'agit de trouver dans le fichier de mots `fr_long_dict_cleaned.txt` une suite de mots ayant tous une distance de Hamming de 1 à la fois avec le précédent et avec le suivant. E.g : de `'TOUT'` à `'RIEN'` en 6 étapes : `['TOUT', 'BOUT', 'BRUT', 'BRUN', 'BREN', 'BIEN', 'RIEN']`. Le fichier de mots à utiliser contient 242818 mots du français en majuscule et sans accents (1 par ligne). Attention tous les mots du fichier finissent donc par le caractère `'\n'` (retour à la ligne).

1. `get_words_from_dictionnary(file_name: str, length: int = None) → list[str]` | Retourne la liste des mots du fichier de nom `file_name` si `length` vaut `None`. Sinon retourne la liste des mots de longueur `length`.

```
>>> DICT_NAME = PATH_DICTIONARIES + "fr_long_dict_cleaned.txt"
>>> get_words_from_dictionary(DICT_NAME, 5)[:5]
['ABACA', 'ABATS', 'ABBES', 'ABCES', 'ABETI']
```

2. `get_nearest_hamming(word: str, words: list[str], haming_distance: int) → list[str]` | Retourne une sous-liste de la liste de mots `words` qui sont à une distance de Hamming `hamming_distance` du mot `word`.

```
>>> get_nearest_hamming("ORANGE", get_words_from_dictionary(DICT_NAME), 0)
['ORANGE']
>>> get_nearest_hamming("ORANGE", get_words_from_dictionary(DICT_NAME), 1)
['FRANGE', 'GRANGE', 'ORANGS', 'ORANTE', 'ORONGE']
>>> get_nearest_hamming("ORANGE", get_words_from_dictionary(DICT_NAME), 2)
['BRANDE', 'BRANLE', 'BRANTE', 'CHANGE', 'CRANTE', 'GRANDE', 'GRINGE', 'ORACLE', 'ORANTS',
'TRANSE', 'URANIE']
```

3. `scale_to_file_annex(scale: list[str], terminal_word: str, words: list[str], max_depth: int, file_out_name: str)` | Algorithme récursif pour compléter une échelle de mots bien formée `scale` jusqu'à `terminal_word` avec des mots de la liste `words`. Une profondeur récursive supérieure à `max_depth` ne sera pas explorée. Les solutions trouvées seront sauvées dans un fichier de nom `file_out_name` (1 solution par ligne, les mots séparés par une espace).
4. `scale_to_file(word1: str, word2: str, max_depth: int)` | Utilise `scale_to_file_annex` pour sauver dans un fichier `sc_{word1}_{word2}_{max_steps}_steps.txt` placé dans le répertoire `OUT` des ressources, les échelles qui relient les deux mots de même longueur `word1` et `word2`. Les mots possibles (de la bonne longueur) seront extraits du fichier `fr_long_dict_cleaned.txt` grâce à la fonction `get_words_from_dictionary`. La profondeur maximum pour la recherche sera de `max_depth`. Un message indiquera le nombre de solutions et le nom du fichier ou bien l'absence de solution. Sera également précisé le temps mis pour trouver les solutions en secondes grâce à `perf_counter`.

```
>>> scale_to_file("TOUT", "RIEN", 5)
Je cherche une echelle de mots de TOUT à RIEN en 5 étapes maximum :
Pas de solution !
>>> scale_to_file("TOUT", "RIEN", 6)
Je cherche une echelle de mots de TOUT à RIEN en 6 étapes maximum :
1 solution(s) dans le fichier TEXTS/OUT/sc_TOUT_RIEN_6_steps.txt en 1494.56s.
>>> scale_to_file("CAENAIS", "BRETONS", 10)
Je cherche une echelle de mots de CAENAIS à BRETONS en 10 étapes maximum :
1 solution(s) dans le fichier TEXTS/OUT/sc_CAENAIS_BRETONS_10_steps.txt en 11137.22s.
```

5. `perfect_scale_to_file(word1: str, word2: str)` | Affiche toutes les échelle de mots parfaites entre `word1` et `word2`. Une échelle de mot est parfaite si le nombre d'étapes est égal au nombre de lettres mal positionnées.

```
>>> perfect_scale_to_file("CAENAIS", "CANNOIS")
Je cherche une echelle de mots de CAENAIS à CANNOIS en 2 étapes maximum
1 solution(s) dans le fichier TEXTS/OUT/sc_CAENAIS_CANNOIS_2_steps.txt en 0.11s.
```

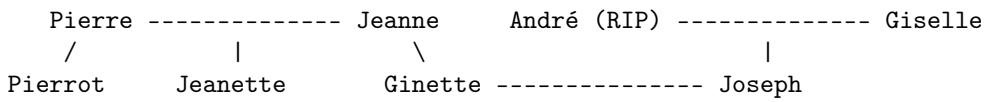
1.2.2 Exercice

Une famille est représentée sous la forme d'une liste de personnes. Chaque personne est elle même représentée par un tuple qui aura la forme :

`person : (num_id, nom, prénom, date_naissance, date_décès, num_sexe, métier, num_id_père, num_id_mère, num_id_conjoint)`

De plus les dates sont des tuples à 3 valeurs entières `(num_jour, num_mois, num_année)`. Si la personne est encore vivante, sa date de décès est un tuple vide ; `num_sexe` est de 0 pour les femmes et de 1 pour les hommes ; les 3 `num_id_XXX` sont à 0 si l'information n'est pas pertinente ou inconnue.

E.g : la variable ADAMS_FAMILY est utilisée pour les test. L'arbre généalogique exploité est le suivant :



La famille ADAMS_FAMILY, de type list[person], sera donc représentée ainsi :

```
ADAMS_FAMILY = [
        (1, "Dupond", "Pierre", (4, 6, 1949), (), 1, "physicien", 0, 0, 2),
        (2, "Dupond", "Jeanne", (7, 6, 1949), (), 0, "physicienne", 0, 0, 1),
        (3, "Dupond", "Pierrot", (7, 6, 1969), (), 1, "informaticien", 1, 2, 0),
        (4, "Dupond", "Jeannette", (5, 4, 1970), (), 0, "informaticienne", 1, 2, 0),
        (5, "Durand", "Ginette", (4, 3, 1972), (), 0, "chimiste", 1, 2, 8),
        (6, "Durand", "André", (6, 3, 1948), (7, 5, 1968), 1, "chimiste", 0, 0, 7),
        (7, "Durand", "Giselle", (7, 5, 1949), (), 0, "chimiste", 0, 0, 6),
        (8, "Durand", "Joseph", (3, 2, 1968), (), 1, "médecin", 6, 7, 5)
]
```

L'objectif des 9 fonctions suivantes est d'extraire des informations en utilisant le plus possible les listes en compréhension du type `[{mapping} for {var} in {sequence} {filter}]` plutôt que des boucles.

1. `get_living(family: list[person]) → list[person]` | Retourne la liste de toutes les personnes vivantes de `family`.

```
>>> get_living(ADAMS_FAMILY)
[(1, 'Dupond', 'Pierre', (4, 6, 1949), (), 1, 'physicien', 0, 0, 2),
(2, 'Dupond', 'Jeanne', (7, 6, 1949), (), 0, 'physicienne', 0, 0, 1),
(3, 'Dupond', 'Pierrot', (7, 6, 1969), (), 1, 'informaticien', 1, 2, 0),
(4, 'Dupond', 'Jeannette', (5, 4, 1970), (), 0, 'informaticienne', 1, 2, 0),
(5, 'Durand', 'Ginette', (4, 3, 1972), (), 0, 'chimiste', 1, 2, 8),
(7, 'Durand', 'Giselle', (7, 5, 1949), (), 0, 'chimiste', 0, 0, 6),
(8, 'Durand', 'Joseph', (3, 2, 1968), (), 1, 'médecin', 6, 7, 5)]
```

2. `get_gender_ranking(family: list[person]) → (list[person], list[person])` | Retourne le 2-uplet correspondant aux femmes (resp. aux hommes) de `family`.

```
>>> get_gender_ranking(ADAMS_FAMILY)
([(2, 'Dupond', 'Jeanne', (7, 6, 1949), (), 0, 'physicienne', 0, 0, 1),
(4, 'Dupond', 'Jeannette', (5, 4, 1970), (), 0, 'informaticienne', 1, 2, 0),
(5, 'Durand', 'Ginette', (4, 3, 1972), (), 0, 'chimiste', 1, 2, 8),
(7, 'Durand', 'Giselle', (7, 5, 1949), (), 0, 'chimiste', 0, 0, 6)],
[(1, 'Dupond', 'Pierre', (4, 6, 1949), (), 1, 'physicien', 0, 0, 2),
(3, 'Dupond', 'Pierrot', (7, 6, 1969), (), 1, 'informaticien', 1, 2, 0),
(6, 'Durand', 'André', (6, 3, 1948), (7, 5, 1968), 1, 'chimiste', 0, 0, 7),
(8, 'Durand', 'Joseph', (3, 2, 1968), (), 1, 'médecin', 6, 7, 5)])
```

3. `get_married_gender_proportion(family: list[person]) → (float, float)` | Retourne le 2-uplet correspondant à la proportion femmes mariées / femmes (resp. hommes mariés / hommes) dans `family`. Il faudra obligatoirement utiliser `get_gender_ranking`.

```
>>> get_married_gender_proportion(ADAMS_FAMILY)
(0.75, 0.75)
```

4. `get_death_age_average(family: list[person]) → float` | Retourne la moyenne d'âge des décès dans la famille `family` en ne considérant que l'année.

```
>>> get_death_age_average(ADAMS_FAMILY)
20.0
```

5. `get_age_average(family: list[person], year: int) → float` | Retourne la moyenne d'âge des personnes de `family` vivantes l'année `year` incluse. .

```
>>> get_age_average(ADAMS_FAMILY, 1967)
18.25
>>> get_age_average(ADAMS_FAMILY, 1969)
12.2
```

6. `get_deans(family: list[person]) → list[person]` | Retourne la liste des doyens de `family` en ne tenant compte que de l'année de naissance.

```
>>> get_deans(ADAMS_FAMILY)
[(1, 'Dupond', 'Pierre', (4, 6, 1949), (), 1, 'physicien', 0, 0, 2),
(2, 'Dupond', 'Jeanne', (7, 6, 1949), (), 0, 'physicienne', 0, 0, 1),
(7, 'Durand', 'Giselle', (7, 5, 1949), (), 0, 'chimiste', 0, 0, 6)]
```

7. `get_parents(ident: int, family: list[person])` → `list[person]` | Retourne la liste des parents de la personne d'identifiant `ident` dans `family`.

```
>>> get_parents(3, ADAMS_FAMILY)
[(1, 'Dupond', 'Pierre', (4, 6, 1949), (), 1, 'physicien', 0, 0, 2),
(2, 'Dupond', 'Jeanne', (7, 6, 1949), (), 0, 'physicienne', 0, 0, 1)]
>>> get_parents(8, ADAMS_FAMILY)
[(6, 'Durand', 'André', (6, 3, 1948), (7, 5, 1968), 1, 'chimiste', 0, 0, 7),
(7, 'Durand', 'Giselle', (7, 5, 1949), (), 0, 'chimiste', 0, 0, 6)]
```

8. `is_intersecting(family1: list[person], family2: list[person])` → `bool` | Retourne `True` si `family1` et `family2` ont au moins un membre en commun. `False` sinon.

```
>>> is_intersecting(living(ADAMS_FAMILY), [p for p in ADAMS_FAMILY if p[4]])
False
>>> is_intersecting(living(ADAMS_FAMILY), deans(ADAMS_FAMILY))
True
```

9. `is_sibling(id1: int, id2: int, family: list[person])` → `bool` | Retourne `True` si les personnes identifiées `id1` et `id2` ont au moins un parent en commun. `False` sinon. Il faudra obligatoirement utiliser `is_intersecting` et `get_parents`.

```
>>> is_sibling(6, 7, ADAMS_FAMILY), is_sibling(3, 4, ADAMS_FAMILY)
(False, True)
>>> is_sibling(4, 5, ADAMS_FAMILY), is_sibling(3, 6, ADAMS_FAMILY)
(True, False)
```

2 CM02 (16/1/2023)

Notions du cours à connaître pour les TP03 et TP04

- les différents types d'ensemble Python selon qu'ils sont mutables (`dict, set`) ou non (`frozenset`).
- fonctions *built-in* de base (`zip, sorted, reversed`) et de *cast* (`bytes`, `dict`, `set`, `frozenset`) *vs.* fonctions associées à une variable par la notation pointée (`.decode`,`.lower`, `.count`, `.split`, `.strip`, `.sort`, `.get`, `.keys`, `.values`, `.items`, `.update`, `.add`, `.discard`, `.intersection`, `.union`, `.difference`)
- fonctions supplémentaires associées à des variables issues du module `matplotlib` (`.subplots`, `.bar`, `.legend`, `.plot`, `.scatter`, `.show`, `.set_title`, `.set_ylim`) et du module `chardet` (`.detect`)
- les notions de *mapping* et de filtrage appliquées à la définition en compréhension de dictionnaires Python
- mémoire et modifications *in-place vs. not-in-place*
- encodages, fichiers textes *vs.* binaires et leur accès en Python
- capter et lever une exception
- notion de lambda fonction

2.1 TP03 - dict (1h)

L'arborescence actuelle de vos dossiers et fichiers utiles à ce TP devrait suivre l'architecture suivante :

```
TP_POO
   |_____ TEXTS
   |         |_____ IN
   |         |      |_____ ARTICLES
   |         |      |              |_____ EN
   |         |      |              |_____ ES
   |         |      |              |_____ FR
   |         |      |_____ BOOKS
   |         |      |           |_____ EN
   |         |      |           |_____ FR
   |         |      |                 |_____ proust_swann.txt
   |         |      |_____ OTHERS
   |         |      |_____ POLITICAL
   |         |      |               |_____ DH
   |         |      |               |        |_____ enDH.txt
   |         |      |               |        |_____ frDH.txt
   |         |      |               |        |_____ plDH.txt
   |         |      |               |        |_____ ruDH.txt
   |         |      |               |        |_____ ruDH_source.txt
   |         |      |               |_____ WISHES
   |         |      |_____ RESOURCES
   |         |                      |_____ CHARACTERS
   |         |                      |               |_____ en_alphabet.txt
   |         |                      |               |_____ en_diacriticals.txt
   |         |                      |               |_____ fr_alphabet.txt
   |         |                      |               |_____ fr_diacriticals.txt
   |         |                      |               |_____ pl_alphabet.txt
   |         |                      |               |_____ pl_diacriticals.txt
   |         |                      |
   |         |                      |_____ WORDS
   |         |_____ OUT
   |
   |_____ config.py
   |_____ S01_TP01_template.py
   |_____ S01_TP02_template.py
   |_____ S02_TP03_template.py
               from config import *
               import chardet
               import marplotlib.pyplot as plt
```

`S02_TP03_template.py` : fichier *template* à compléter pour réaliser ce TP. Les 3 premières lignes d'importation servent à récupérer (1) les variables globales du fichier `config.py` ainsi que l'accès aux fonctions des modules (2) `chardet` et (3) `matplotlib`. L'objectif général de ce TP est de travailler sur des ressources textuelles au niveau du traitement des caractères (identification des langues, gestion de la variété des caractères selon les langues en considérant les accents sur les lettres - appelés diacritiques, et la casse des caractères, construction et visualisation d'histogrammes de caractères).

Les 11 fonctions suivantes permettent de récupérer sous la forme d'une chaîne de caractère les contenus de fichiers textuels et d'en proposer un histogramme de caractères. Elles permettent de s'adapter le mieux possibles aux contraintes d'encodage des caractères dues aux différentes langues possibles de nos ressources (alphabets, gestion des diacritiques et des majuscules, fichiers dans un encodage non connu).

1. `get_text_from_file_name(file_name:str)` → `str` | retourne, sous la forme d'une chaîne de caractères, le texte du fichier de nom `file_name` et encodé en `utf8`. Par exemple l'instruction de l'exemple affichera : `"Всеобщая декларация "`.

```
>>> get_text_from_file_name(PATH_DH + "ruDh.txt")[:20]
```

2. `get_text_from_file_name_with_encoding(file_name:str)` → `(dict, str)` | Détecte l'encodage du fichier de nom `file_name` ouvert sous sa forme binaire puis retourne le dictionnaire des informations sur l'encodage détecté ainsi que le texte décodé sous la forme d'une chaîne de caractère. Vous utiliserez la fonction `detect` du module `chardet` et la fonction `.decode` associée aux chaînes binaires. Par exemple le programme de l'exemple suivant affichera :

```
Erreur d'encodage, doit s'appuyer sur les informations suivantes :
{'encoding': 'ISO-8859-5', 'confidence': 0.99, 'language': 'Russian'}
"Всеобщая декларация "
```

```
>>> try:
>>>     get_text_from_file_name(PATH_DH + "ruDH_source.txt")
>>> except UnicodeDecodeError:
>>>     "Erreur d'encodage, doit s'appuyer sur les informations suivantes :"
>>> finally:
>>>     info, text = get_text_from_file_name_with_encoding(PATH_DH + "ruDH_source.txt")
>>>     info
>>>     text[:20]
```

3. `get_basic_alphabet(alpha2_code:str)` → `str` | Retourne une chaîne constituée des caractères de la langue du pays dont le code sur 2 caractères est `alpha2_code`. Retourne `""` si le code n'existe pas. Le chemin d'accès au fichier dans nos ressources est `PATH_ALPHABET + {alpha2_code}_alphabet.txt`). Vous capterez l'exception `FileNotFoundError`.

```
>>> get_basic_alphabet('fr'), get_basic_alphabet('en'), get_basic_alphabet('ru')
("abcdefghijklmnopqrstuvwxyzœæ&", "abcdefghijklmnopqrstuvwxyz&", "")
```

4. `get_diacriticals(alpha2_code:str)` → `dict` | Retourne un dictionnaire constitué des paires associant les lettres susceptibles d'être accentuées (clés) et leur(s) homologue(s) avec l'accent (valeurs) dans la langue du pays dont le code sur 2 caractères est `alpha2_code`. Retourne `{}` si le code n'existe pas. Le chemin d'accès au fichier dans nos ressources est `PATH_ALPHABET + {alpha2_code}_diacriticals.txt`). Vous utiliserez les fonctions `str.join` et `list.split`.

```
>>> get_diacriticals('fr')
{'a': 'àâ', 'e': 'éèêë', 'i': 'îï', 'o': 'ôö', 'u': 'ùûü', 'y': 'ÿ', 'c': 'ç', 'n': 'ñ'}
```

5. `get_accented_letters(alpha2_code:str)` → `str` | Retourne la chaîne des caractères accentués dans la langue du pays dont le code sur 2 caractères est `alpha2_code`. Retourne `""` si le code n'est pas géré par nos ressources. Vous utiliserez les fonctions `str.join`, `dict.values` et `get_diacriticals`.

```
>>> get_accented_letters('fr')
'àâéèêëîïôöùûüÿçñ'
```

6. `get_all_letters(alpha2_code:str)` → `str` | Retourne une chaîne constituée de toutes les lettres autorisées par la langue du pays dont le code sur 2 caractères est `alpha2_code`. Retourne `""` si le code n'est pas géré par nos ressources. Vous utiliserez les fonctions `get_basic_alphabet`, `get_accented_letter` et `str.upper`.

```
>>> get_all_letters('fr')
'abcdefghijklmnopqrstuvwxyzœæ&àâäéèêëîïôöùûüÿçñABCDEFGHIJKLMNOPQRSTUVWXYZŒÆÀÂÄÉÈÊËÎÏÔÖÙÛÜ
ŸÇÑ'
```

7. `get_unaccented_letter(letter:char, diacriticals:dict)` → `char` | Retourne le caractère `letter` mais sans les diacritiques du dictionnaire `diacriticals` et sans changer la casse de caractère. Vous utiliserez les fonctions `dict.items`, `str.lower` et `str.upper`.

```
>>> diacriticals = get_diacriticals('fr')
>>> get_unaccented_letter("À", diacriticals), get_unaccented_letter("?", diacriticals)
('A', '?')
```

8. `get_unaccented_text(text:str, diacriticals:dict)` → `str` | Retourne le texte de la chaîne `text` mais sans les diacritiques du dictionnaire `diacriticals` et sans changer la casse de caractère. Vous utiliserez `get_unaccented_letter`.

```
>>> get_unaccented_letter("Bonne journée à tous et à toutes ! À tout à l'heure.",
diacriticals)
"Bonne journee a tous et a toutes ! A tout a l'heure."
```

9. `get_letters_histogram(text:str, alpha2_code:str)` → `dict` | Retourne le dictionnaire des occurrences des lettres de `text`. Toutes le texte est d'abord mis en bas-de-casse et tous les diacritiques remplacés par leur homologue sans accent de la langue du pays dont le code sur 2 caractères est `alpha2_code`. Vous devrez ensuite utiliser la définition en compréhension de dictionnaire et la fonction `.count`.

```
>>> get_letters_histogram('Bonne journée à tous et à toutes ! À tout à l'heure.', 'fr')
{'a': 4, 'b': 1, 'c': 0, 'd': 0, 'e': 7, 'f': 0, 'g': 0, 'h': 1, 'i': 0, 'j': 1, 'k': 0,
'l': 1, 'm': 0, 'n': 3, 'o': 5, 'p': 0, 'q': 0, 'r': 2, 's': 2, 't': 6, 'u': 5, 'v': 0,
'w': 0, 'x': 0, 'y': 0, 'z': 0, 'œ': 0, 'æ': 0, '&': 0}
```

10. `get_normalized_histogram(histogram:dict)` → `dict` | Retourne `histogram` en remplaçant pour chaque lettre le nombre d'occurrences par la valeur normalisée entre 0 et 1 (arrondi au centième).

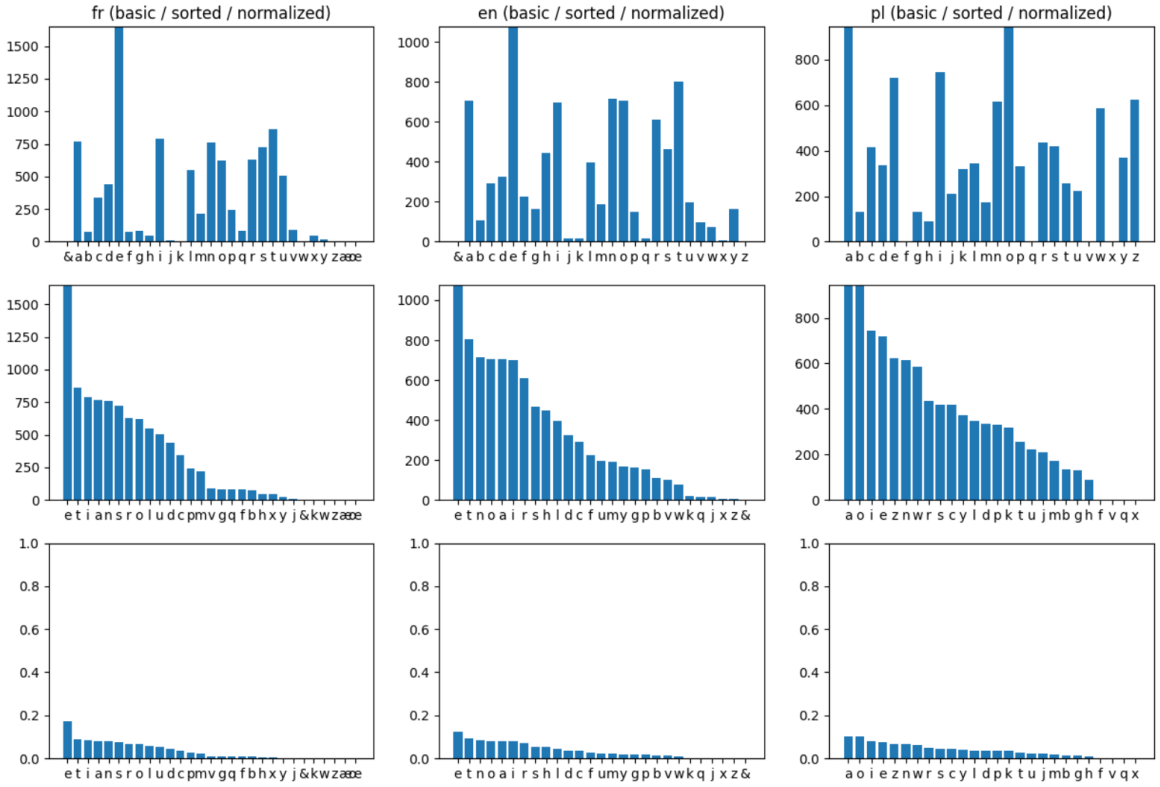
```
>>> get_normalized_histogram(get_letters_histogram('Bonne journée à tous et à toutes !
À tout à l'heure.', 'fr'))
{'a': 0.11, 'b': 0.03, 'c': 0.0, 'd': 0.0, 'e': 0.18, 'f': 0.0, 'g': 0.0, 'h': 0.03,
'i': 0.0, 'j': 0.03, 'k': 0.0, 'l': 0.03, 'm': 0.0, 'n': 0.08, 'o': 0.13, 'p': 0.0,
'q': 0.0, 'r': 0.05, 's': 0.05, 't': 0.16, 'u': 0.13, 'v': 0.0, 'w': 0.0, 'x': 0.0,
'y': 0.0, 'z': 0.0, 'œ': 0.0, 'æ': 0.0, '&': 0.0}
```

11. `add_figure_histogram(figure_axis:AxesSubplot, histogram:dict, is_sorted_by_freq:bool=False)` | Ajoute un diagramme à barre sur l'axe `figure_axis` d'une figure `matplotlib`. Ce diagramme représentera l'histogramme de caractères `histogram`.

    Chaque barre représentera le nombre d'occurrences (axe des Y) des lettres (axe des X). Selon la valeur booléenne de `is_sorted_by_freq` les points seront ordonnées selon les valeurs croissantes des abscisses (`False`) ou selon les valeurs décroissantes des ordonnées (`True`). Vous utiliserez les fonctions `dict.items` et `list.sort` ou `sorted` (en jouant avec les paramètres `key` et `reverse`).

    Le programme de l'exemple devra produire la figure suivante pour les 9 histogrammes calculés (3 diagrammes par colonne pour 3 langues).

```
>>> fig, ax = plt.subplots(3, 3, figsize=(15, 10))
>>> for col, code in zip(range(3), ['fr', 'en', 'pl']):
>>>     ax[0][col].set_title(code + " (basic / sorted / normalized)")
>>>     dh_text = get_text_from_file_name(PATH_DH + code + "DH.txt")
>>>     hist = get_letters_histogram(dh_text, code)
>>>     maxi = max(hist.values())
>>>     ax[0][col].set_ylim(0, maxi)
>>>     add_figure_histogram(ax[0][col], hist)
>>>     ax[1][col].set_ylim(0, maxi)
>>>     add_figure_histogram(ax[1][col], hist, True)
>>>     ax[2][col].set_ylim(0, 1)
>>>     add_figure_histogram(ax[2][col], get_normalized_histogram(hist), True)
>>> plt.show()
```



2.2 TP04 - dict - set (2h)

L'arborescence actuelle de vos dossiers et fichiers utiles à ce TP devrait suivre l'architecture suivante :

```
TP_POO
   |_____ TEXTS
   |         |_____ IN
   |         |      |_____ ARTICLES
   |         |      |              |_____ EN
   |         |      |              |      |_____ nytimes{1 à 9}.txt
   |         |      |              |
   |         |      |              |_____ ES
   |         |      |              |      |_____ elpais{1 à 7}.txt
   |         |      |              |
   |         |      |              |_____ FR
   |         |      |                     |_____ lemonde{1 à 11}.txt
   |         |      |
   |         |      |_____ BOOKS
   |         |      |            |_____ EN
   |         |      |            |
   |         |      |            |_____ FR
   |         |      |                   |_____ proust_swann.txt
   |         |      |                   |
   |         |      |                   |_____ verne_20000.txt
   |         |      |
   |         |      |_____ OTHERS
   |         |      |
   |         |      |_____ POLITICAL
   |         |      |               |_____ DH
   |         |      |               |
   |         |      |               |_____ WISHES
   |         |      |
   |         |      |_____ RESOURCES
   |         |                     |_____ CHARACTERS
   |         |                     |
   |         |                     |_____ WORDS
   |         |
   |         |_____ OUT
   |
   |_____ config.py
   |_____ S01_TP01_template.py
   |_____ S01_TP02_template.py
   |_____ S02_TP03_template.py
   |_____ S02_TP04_template.py
               from S02_TP03_template import *
               from collections import Counter
               import wordcloud
               import numpy as np
               from PIL import Image
```

`S02_TP04_template.py` : fichier *template* à compléter pour réaliser ce TP. La première ligne d'importation sert à récupérer tout le TP précédent (sauf le `main`). La seconde permet d'utiliser la classe `Counter` pour la construction efficace d'histogrammes. Les 3 dernières ne seront utiles que pour la dernière question (génération d'un nuage de mots).

L'objectif général de ce TP est de compléter le précédent pour la visualisation d'histogrammes de caractères avec `matplotlib` ; puis d'essayer d'exploiter les textes au niveau du mot plutôt qu'à celui du caractère.

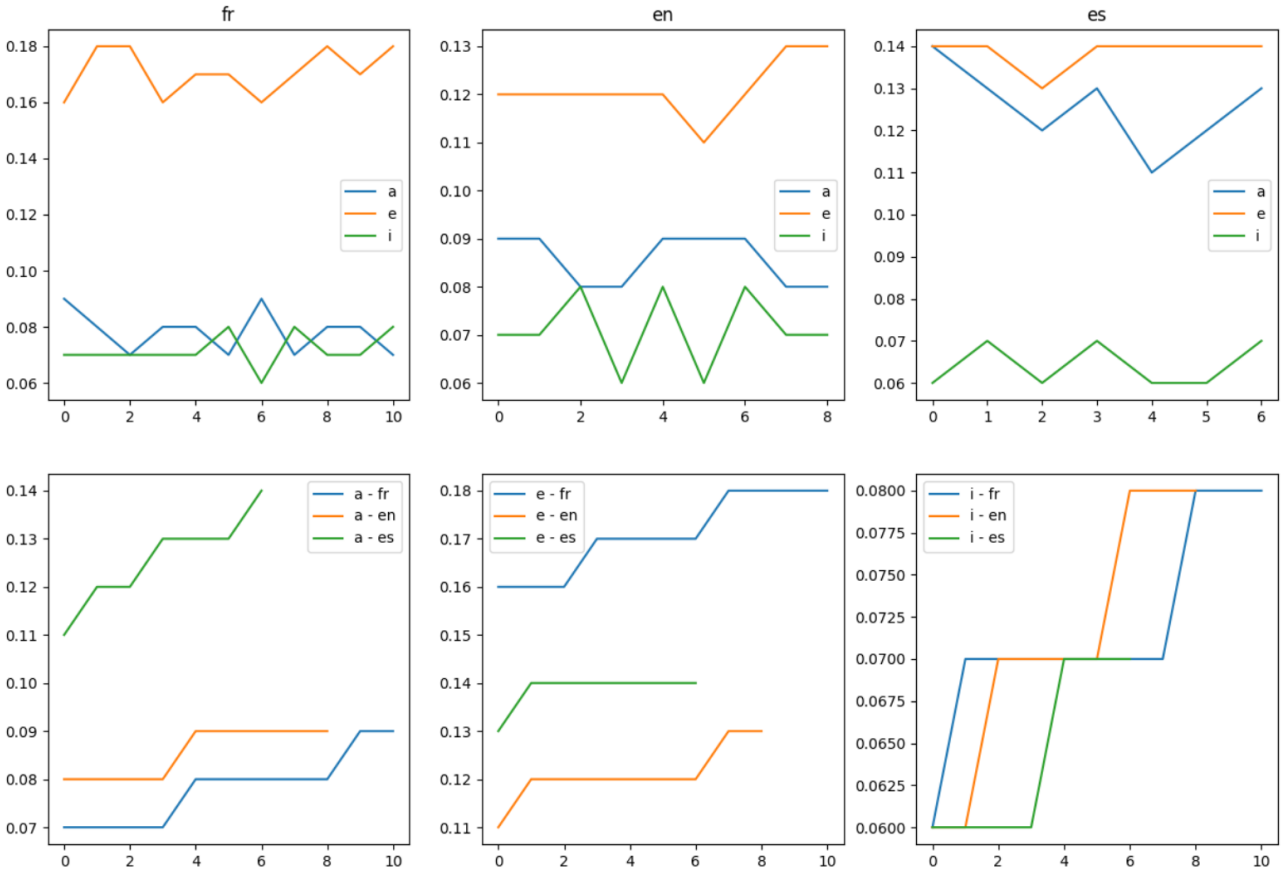
2.2.1 Exercice

Les 2 fonctions suivantes ont pour finalité la comparaison et la visualisation de valeurs extraites d'une série d'histogrammes de caractères normalisés.

1. `add_figure_histograms_1d(figure_axis:subplot, char:char, histograms:list[dict], is_sorted:bool=False, label_addon:str=None)` | Ajoute sur l'axe `figure_axis` d'une figure `matplotlib` une sous-figure montrant l'évolution des occurrences de `char` à travers la liste des `histograms` préalablement normalisés. Une absence dans l'histogramme est considéré comme une ordonnée de 0. `label_addon` permet de compléter la légende au besoin et `is_sorted` décide si les abscisses sont triées selon l'ordre décroissant des valeurs de leur image. La légende de l'image devra être également affichée.

   Le programme de l'exemple devra produire la figure suivante pour les 6 histogrammes calculés (3 diagrammes par ligne pour 2 lignes). La première ligne compare les histogrammes des lettres 'a', 'e', 'i' pour 3 langues, la seconde ligne compare les langues 'fr', 'en' et 'es' pour 3 lettres.

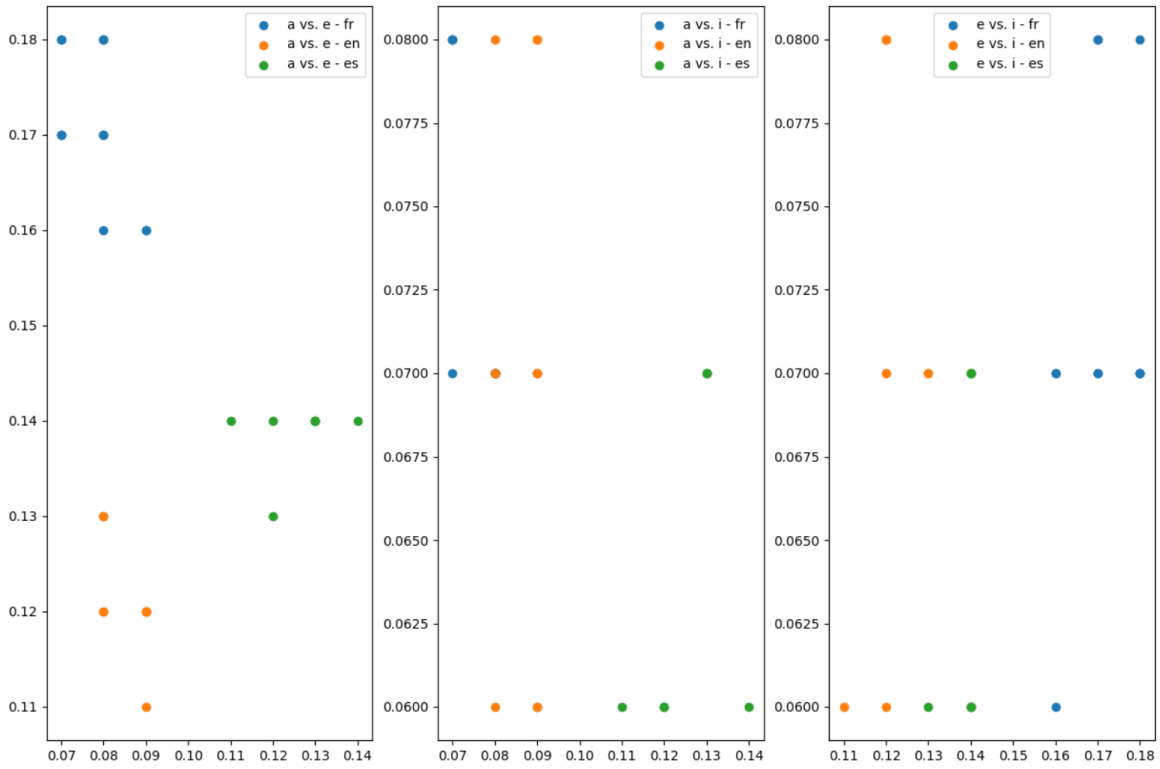
```
LE_MONDE_TEST = [PATH_ARTICLES + "FR/lemonde" + str(val) + ".txt" for val in range(1, 12)]
NY_TIMES_TEST = [PATH_ARTICLES + "EN/nytimes" + str(val) + ".txt" for val in range(1, 10)]
EL_PAIS_TEST = [PATH_ARTICLES + "ES/elpais" + str(val) + ".txt" for val in range(1, 8)]

_, ax = plt.subplots(2, 3, figsize=(15, 10))
histograms_fr = [get_letters_histogram(get_text_from_file_name(file_name), 'fr')
                 for file_name in LE_MONDE_TEST]
histograms_en = [get_letters_histogram(get_text_from_file_name(file_name), 'en')
                 for file_name in NY_TIMES_TEST]
histograms_es = [get_letters_histogram(get_text_from_file_name(file_name), 'es')
                 for file_name in EL_PAIS_TEST]

ax[0][0].set_title("fr")
add_figure_histograms_1d(ax[0][0], 'a', histograms_fr)
add_figure_histograms_1d(ax[0][0], 'e', histograms_fr)
add_figure_histograms_1d(ax[0][0], 'i', histograms_fr)

ax[0][1].set_title("en")
add_figure_histograms_1d(ax[0][1], 'a', histograms_en)
add_figure_histograms_1d(ax[0][1], 'e', histograms_en)
add_figure_histograms_1d(ax[0][1], 'i', histograms_en)

ax[0][2].set_title("es")
add_figure_histograms_1d(ax[0][2], 'a', histograms_es)
add_figure_histograms_1d(ax[0][2], 'e', histograms_es)
add_figure_histograms_1d(ax[0][2], 'i', histograms_es)

add_figure_histograms_1d(ax[1][0], 'a', histograms_fr, True, 'fr')
add_figure_histograms_1d(ax[1][0], 'a', histograms_en, True, 'en')
add_figure_histograms_1d(ax[1][0], 'a', histograms_es, True, 'es')

add_figure_histograms_1d(ax[1][1], 'e', histograms_fr, True, 'fr')
add_figure_histograms_1d(ax[1][1], 'e', histograms_en, True, 'en')
add_figure_histograms_1d(ax[1][1], 'e', histograms_es, True, 'es')

add_figure_histograms_1d(ax[1][2], 'i', histograms_fr, True, 'fr')
add_figure_histograms_1d(ax[1][2], 'i', histograms_en, True, 'en')
add_figure_histograms_1d(ax[1][2], 'i', histograms_es, True, 'es')
plt.show()
```



2. `add_figure_histograms_2d(figure_axis:subplot, char1:char, char2:char, histograms:list[dict], label_addon:str=None)` | Ajoute à l'axe `figure_axis` d'une figure `matplotlib` une sous-figure montrant le nuage de points formé par la mise en regard des occurrences des lettres `char1` et `char2` à travers la liste des `histograms` préalablement normalisés. `label_addon` permet de compléter la légende au besoin. Une absence dans l'histogramme est considéré comme une ordonnée de 0. La légende devra être indiquée.

   Le programme de l'exemple devra produire la figure suivante pour les 3 histogrammes calculés pour comparer 2 à 2 les histogrammes des lettres 'a', 'e', 'i' pour les 3 langues.

```
>>> _, ax = plt.subplots(1, 3, figsize=(15, 10))
>>> add_figure_histograms_2d(ax[0], 'a', 'e', histograms_fr, 'fr')
>>> add_figure_histograms_2d(ax[0], 'a', 'e', histograms_en, 'en')
>>> add_figure_histograms_2d(ax[0], 'a', 'e', histograms_es, 'es')
>>> add_figure_histograms_2d(ax[1], 'a', 'i', histograms_fr, 'fr')
>>> add_figure_histograms_2d(ax[1], 'a', 'i', histograms_en, 'en')
>>> add_figure_histograms_2d(ax[1], 'a', 'i', histograms_es, 'es')
>>> add_figure_histograms_2d(ax[2], 'e', 'i', histograms_fr, 'fr')
>>> add_figure_histograms_2d(ax[2], 'e', 'i', histograms_en, 'en')
>>> add_figure_histograms_2d(ax[2], 'e', 'i', histograms_es, 'es')
>>> plt.show()
```



2.2.2 Exercice

Les 5 fonctions suivantes ont pour objectif l'extraction des mots d'un texte, la sélection des plus pertinents et leur affichage sous la forme d'un nuage de mots.

1. `get_cleaned_text(text:str, alpha2_code:str)` → `str` | Retourne `text` en ne conservant que les lettres de la langue associée à `alpha_code`. Les caractères de retours à la ligne, de tabulation, apostrophes et traits d'union seront remplacés par des espaces (il s'agira des caractères : `\n`, `\t`, `'`, `’`, et `-`) .

```
>>> BOOK_TEST1 = PATH_BOOKS + "FR/proust_swann.txt"
>>> BOOK_TEST2 = PATH_BOOKS + "FR/verne_20000.txt"
>>> text1 = get_text_from_file(BOOK_TEST1)
>>> text2 = get_text_from_file(BOOK_TEST2)
>>> get_cleaned_text(text1.lower(), 'fr')[53:95]
"longtemps je me suis couché de bonne heure"
>>> get_cleaned_text(text2.lower(), 'fr')[-25:]
"le capitaine nemo et moi "
```

2. `get_stop_words_from_file(alpha2_code:str)` → `list` | Retourne la liste des *stop words* (mots fréquents que l'on veut pouvoir éliminer lors du traitement d'un texte) associée à la langue `alpha2_code`. Le fichier est dans nos ressources `'PATH_DICTIONARIES + {alpha2_code}_stopwords.txt'` et est construit comme des lignes de mots séparés par des espaces. Une liste vide sera retournée si le fichier n'existe pas (il faudra pour cela capter l'exception `FileNotFoundError`).

```
>>> get_stop_words_from_file('fr')[:9]
['a', 'abord', 'absolument', 'actuellement', 'ah', 'ai', 'aie', 'aies', 'ailleurs']
```

3. `get_words_from_text(text:str, alpha2_code:str, min_len:int=1, max_len:int=100, with_stop_words:bool=False)` → `list` | Retourne la liste des mots du `text` associée à la langue `alpha2_code` dont la longueur est comprise entre `min_len` et `max_len`. Si `with_stop_words` a pour valeur `True` les mots seront également filtrés selon ce critère. Vous pourrez utiliser les fonctions `.split` et `.strip` associées aux chaînes de caractères.

```
>>> cleaned_text1 = get_cleaned_text(text1.lower(), 'fr')
>>> cleaned_text2 = get_cleaned_text(text2.lower(), 'fr')
>>> get_words_from_text(cleaned_text1, 'fr', 8, with_stop_words=True)[:6]
['première', 'fermaient', 'chercher', 'éveillait', 'souffler', 'réflexions']
>>> get_words_from_text(cleaned_text2, 'fr', 8, with_stop_words=True)[:6]
['illustre', 'neuvilli', 'bibliotheque', 'education', 'recreation', 'matières']
```

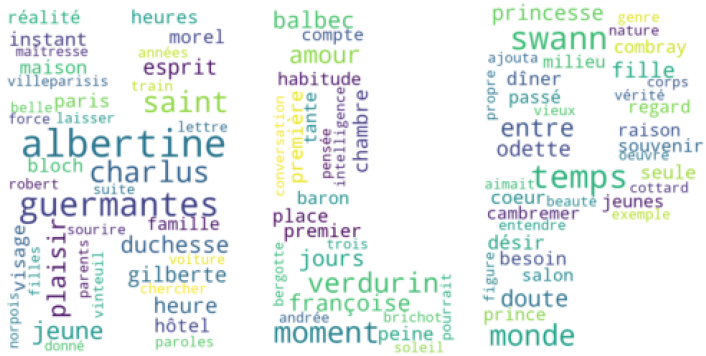
4. `get_words_histogram(words:list, max_words:int=None)` → `Counter` | Retourne un dictionnaire `Counter` qui associe à chaque mot de `words` son nombre d'occurrence. Si `max_words` est donné, il ne faut sélectionner que les `max_words` plus fréquents. Vous pourrez utiliser la classe `Counter` et sa fonction associée `.most_common`.

```
>>> get_words_histogram(words1, 4)
Counter({'albertine': 2400, 'guermantes': 1770, 'verdurin': 1184, 'françoise': 800})
```

5. `generate_wordcloud(file_name:str, alpha_code:str, max_words:int=50, min_len_words:int=1, max_len_words:int=100, img_mask:str=None)` | Génère un nuage de maximum `max_words` mots inscrits dans la forme donnée par l'image masque `img_mask` si un fichier est précisé. Le texte source `file_name` associé à la langue `alpha_code` sera découpé en ne conservant que les mots de longueur comprise entre `min_len_words` et `max_len_words`. Les *stopwords* devront être éliminés. En vous inspirant d'exemples d'utilisation du module `wordcloud` vous pourrez par exemple réaliser les quatre images suivantes.

```
>>> f1, f2 = BOOK_TEST1, BOOK_TEST2
>>> generate_wordcloud(f1, 'fr', 100, 5)
>>> generate_wordcloud(f1, 'fr', 100, 5, img_mask='S02_TP04_mask.png')
>>> generate_wordcloud(f2, 'fr', 100, 4)
>>> generate_wordcloud(f2, 'fr', 100, 4, img_mask='S02_TP04_mask.png')
```


albertine
temps
swann
guermantes
moment
charlus
saint
monde
verdurin
entre
fille
amour
jeune
heure
balbec
odette
duchesse
françoise
plaisir
doute
esprit
jours
désir
dîner
morel
seule
place
instant
princesse
chambre
peine
besoin
habitude
souvenir
bloch
paris
hôtel
cottard
salon
entendre
gilberte
coeur
premier
villeparisis
combray
ajouta


réalité
heures
balbec
princesse
swann
instant
morel
compte
maison
esprit
amour
fille
saint
habitude
dîner
passé
regard
albertine
charlus
entre
odette
raison
souvenir
temps
guermantes
baron
place
premier
jours
verdurin
françoise
désir
besoin
salon
doute
duchesse
gilberte
heure
hôtel
jeune
plaisir
moment
peine
monde


nemo
trois
cinq
land
forme
flots
conseil
eaux
canadien
vingt
cent
quatre
océan
milieu
moment
capitaine
heures
surface
bord
entre
terre
mètres
mille
nautilus
mers
professeur
aronnax
main
cents
corps
poissons
milles


vingt
terre
nemo
entre
land
professeur
eaux
canadien
capitaine
conseil
mille
océan
trois
mers
bord
nautilus
flots
cent
milieu
salon
cinq
surface
mètres
poissons
heures

3 CM03 (23/1/2023)

3.1 TP05 - module `grid manager`

`S02_TP05_template.py` : fichier *template* à compléter pour réaliser ce TP. La première ligne d'importation sert à importer le module `random`. L'objectif général de ce TP est de construire un module python composé de 21 fonctions de gestion de grille/tore.

1. `get_grid(line:int, column:int, value:Any) → list[list[Any]]` | Retourne une grille de `line` lignes et `column` colonnes initialisées à `value`.

```
>>> GRID_CONST_TEST = get_grid(5, 7, 0)
>>> print(GRID_CONST_TEST)
[[0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0]]
```

2. `get_random_grid(line:int, column:int, values:list[Any]) → list[list[Any]]` | Retourne une grille de `line` lignes et `column` colonnes initialisées aléatoirement avec des valeures de la liste `values`.

```
>>> GRID_RANDOM_TEST = get_random_grid(5, 7, range(2))
>>> print(GRID_RANDOM_TEST)
[[1, 0, 1, 1, 0, 1, 0], [1, 0, 0, 0, 1, 1, 0], [1, 0, 1, 0, 0, 1, 0],
[1, 1, 0, 0, 1, 0, 0], [0, 1, 0, 1, 0, 0, 1]]
```

3. `get_lines_count(grid:list[list[Any]]) → int` | Retourne le nombre de lignes de la grille `grid`.

```
>>> get_lines_count(GRID_RANDOM_TEST)
5
```

4. `get_columns_count(grid:list[list[Any]]) → int` | Retourne le nombre de colonnes de la grille `grid`.

```
>>> get_columns_count(GRID_RANDOM_TEST)
7
```

5. `get_line(grid:list[list[Any]], line_number:int) → list[Any]` | Retourne la ligne numéro `line_number` de `grid`.

```
>>> print(get_line(GRID_RANDOM_TEST, 1))
[1, 0, 0, 0, 1, 1, 0]
```

6. `get_column(grid:list[list[Any]], column_number:int) → list[Any]` | Retourne la colonne numéro `column_number` de `grid`.

```
>>> print(get_column(GRID_RANDOM_TEST, 6))
[0, 0, 0, 0, 1]
```

7. `get_line_str(grid:list[list[Any]], line_number:int, separator:str) → str` | Retourne la chîne de caractère correspondant à la concaténation des valeurs de la ligne numéro `line_number` de la grille `grid`. Les caractères sont séparés par la chaîne de caractère `separator`.

```
>>> get_line_str(GRID_RANDOM_TEST, 2, '\t')
1       0       1       0       0       1       0
```

8. `get_grid_str(grid:list[list[Any]], separator:str) → str` | Retourne la chaîne de caractère représentant la grille `grid`. Les caractères de chaque ligne sont séparés par la chaîne de caractère `separator`. Les lignes sont séparées par le caractère de retour à la ligne `\n`.

```
>>> get_grid_str(GRID_RANDOM_TEST, '')
1011010
1000110
1010010
1100100
0101001
```

9. `get_diagonal:list[list[Any]]) → list[Any]` | Retourne la diagonale de `grid`.

```
>>> get_diagonal(GRID_RANDOM_TEST)
[1, 0, 1, 0, 0]
```

10. `get_anti_diagonal:list[list[Any]]) → list[Any]` | Retourne l'anti-diagonale de `grid`.

```
>>> get_anti_diagonal(GRID_RANDOM_TEST)
[0, 1, 0, 0, 0]
```

11. `has_equal_values(grid:list[list[Any]], value:Any) → bool` | Teste si toutes les valeurs de `grid` sont égales à `value`.

```
>>> has_equal_values(GRID_CONST_TEST, 0), has_equal_values(GRID_RANDOM_TEST, 0)
(True, False)
```

12. `is_square(grid:list[list[Any]]) → bool` | Teste si `grid` a le même nombre de lignes et de colonnes.

```
>>> is_square(GRID_RANDOM_TEST)
False
```

13. `get_count(grid:list[list[Any]], value:Any) → int` | Retourne le nombre d'occurrences de `value` dans `grid`.

```
>>> get_count(GRID_RANDOM_TEST, 1) == 16
True
```

14. `get_sum(grid:list[list[Any]]) → Any` | Retourne la somme de tous les éléments de `grid`.

```
>>> get_sum(GRID_RANDOM_TEST)
True
```

15. `get_coordinates_from_cell_number(grid:list[list[Any]], cell_number:int) → tuple[int, int]]` | Retourne le résultat de la conversion du numéro de case `cell_number` de `grid` vers les coordonnées (ligne, colonne) correspondants.

```
>>> get_coordinates_from_cell_number(GRID_RANDOM_TEST, 13)
(1, 6)
```

16. `get_cell_number_from_coordinates(grid:list[list[Any]], line_number:int, column_number:int)` → `int` | Retourne le résultat de la conversion des coordonnées (`line_number`, `column_number`) de `grid` vers le numéro de case correspondant.

```
>>> get_cell_number_from_coordinates(GRID_RANDOM_TEST, 1, 6)
13
```

17. `get_cell(grid:list[list[Any]], cell_number:int)` → `int` | Retourne la valeur de la cellule numéro `cell_number` de la grille `grid`.

```
>>> get_cell(GRID_RANDOM_TEST, 9)
0
```

18. `set_cell(grid:list[list[Any]], cell_number:int, value:Any)` | Positionne à la valeur `value` la case numéro `cell_number` de la grille `grid`.

```
>>> set_cell(GRID_RANDOM_TEST, 9, 1)
>>> get_cell(GRID_RANDOM_TEST, 9)
1
```

19. `get_same_value_cell_numbers(grid:list[list[Any]], value:Any)` → `list[int]` | Retourne la liste des numéros des cases à valeur égale à `value` dans la grille `grid`.

```
>>> get_same_value_cell_numbers(GRID_RANDOM_TEST, 1)
[0, 2, 3, 5, 7, 9, 11, 12, 14, 16, 19, 21, 22, 25, 29, 31, 34]
```

20. `get_neighbour(grid:list[list[Any]], line_number:int, column_number:int, delta:[int, int], is_tore:bool)` → `Any` | Retourne le voisin de la cellule `grid[line_number][column_number]`. La définition de voisin correspond à la distance positionnelle indiquée par le 2-uplet `delta = (delta_line, delta_column)`. La case voisine est alors `grid[line_number + delta_line][column_number + delta_column]`. Si `is_tore` est à `True` le voisin existe toujours en considérant `grid` comme un tore. Si `is_tore` est à `False` retourne `None` lorsque le voisin est hors de la grille `grid`.

```
>>> get_neighbour(GRID_RANDOM_TEST, 1, 6, (0, 1), True)
1
>>> get_neighbour(GRID_RANDOM_TEST, 1, 6, (0, 1), False)
None
```

21. `get_neighborhood(grid:list[list[Any]], line_number:int, column_number:int, deltas:list[tuple[int, int]], is_tore:bool)` → `list[Any]` | Retourne pour la grille `grid` la liste des N voisins de `grid[line_number][column_number]` correspondant aux N *2-uplet* `(delta_line, delta_column)` fournis par la liste `deltas`. Si `is_tore` est à `True` le voisin existe toujours en considérant `grid` comme un tore. Si `is_tore` est à `False` un voisin hors de la grille `grid` n'est pas considéré.

```
>>> WIND_ROSE = ((-1, 0), (-1, 1), (0, 1), (1, 1), (1, 0), (1, -1), (0, -1), (-1, -1))
>>> get_neighborhood(GRID_RANDOM_TEST, 1, 6, WIND_ROSE, True)
[0, 1, 1, 1, 0, 1, 1, 1]
>>> get_neighborhood(GRID_RANDOM_TEST, 1, 6, WIND_ROSE, False)
[0, None, None, None, 0, 1, 1, 1]
```

3.2 TP06 - classe `Grid`

`S02_TP06_template.py` : fichier *template* à compléter pour réaliser ce TP. Seules deux lignes d'importation sont indiquées. La première sert à importer le module `random` et la seconde le module `turtle` nécessaire uniquement pour la dernière question (rouge).

1. L'objectif général de ce TP est de construire la classe `Grid` en s'inspirant du module de gestion de grille précédent et du diagramme de classe ci-après.

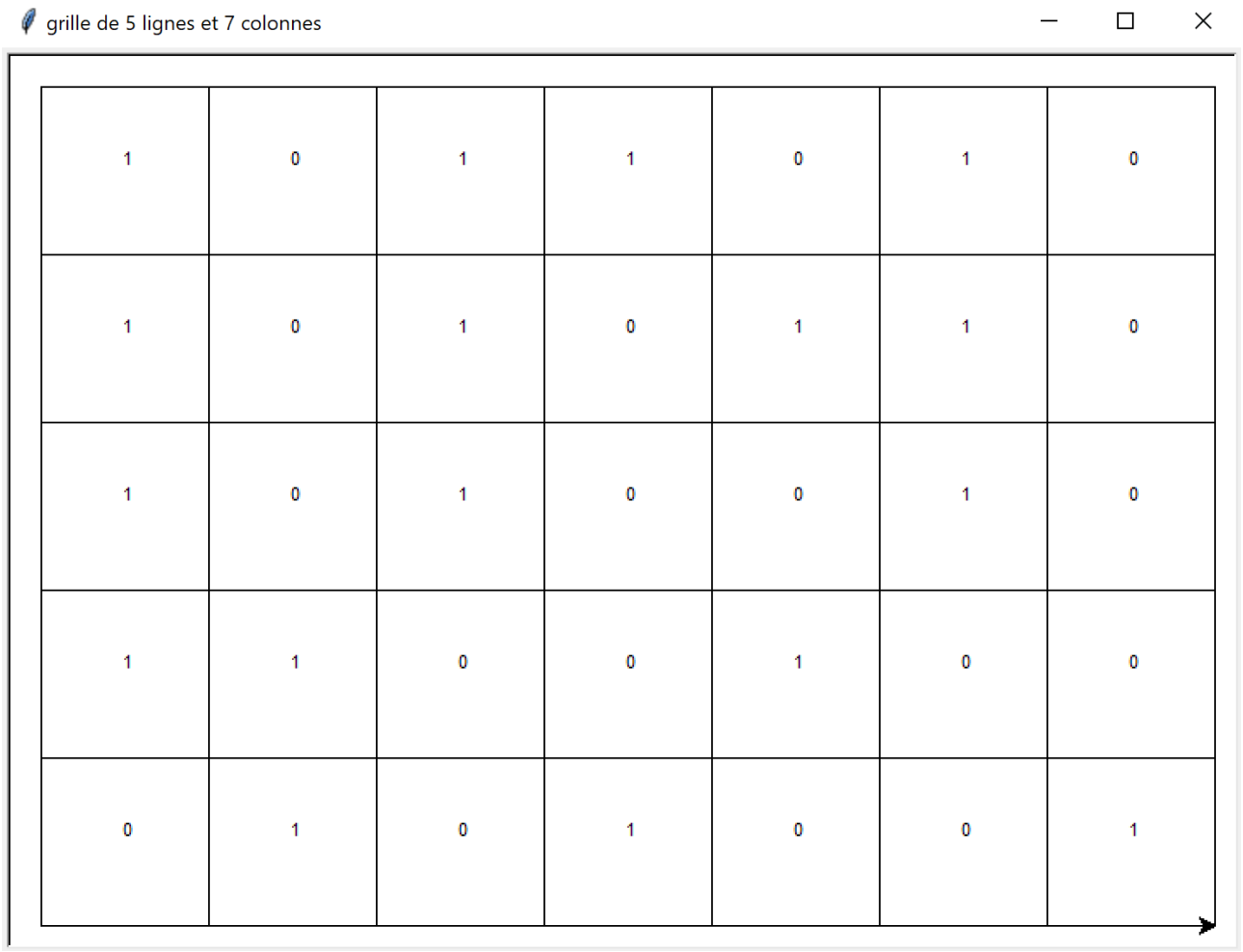
Grid
grid:list[list[Any]] lines_count:int columns_count:int
__init__(grid_init:list[list[Any]]) fill_random(values:list[Any]):void get_line(line_number:int):list[Any] get_column(column_number:int):list[Any] get_line_str(line_number:int, separator:str):str get_grid_str(separator:str):str get_diagonal():list[Any] get_anti_diagonal():list[Any] has_equal_values():bool is_square():bool get_count(value:Any):int get_sum():int get_coordinates_from_cell_number(cell_number:int):tuple[int, int] get_cell_number_from_coordinates(line_number:int, column_number:int):int get_cell(cell_number:int):Any set_cell(value:Any):void get_same_value_cell_numbers(value:Any):list[int] get_neighbour(line_number:int, column_number:int, delta:tuple[int, int], is_tore:bool):Any|None get_neighborhood(line_number:int, column_number:int, deltas:list[tuple[int, int]], is_tore:bool):list[Any|None]

2. Faites un `main` dans le fichier qui permettra de tester votre classe. En continuant l'exemple, y écrire au fur et a mesure vos propres tests de la classe Grid.

```
>>> random.seed(1000)
>>> LINES_COUNT_TEST, COLUMNS_COUNT_TEST = 5, 7
>>> GRID_INIT_TEST = [[0] * COLUMNS_COUNT_TEST
                      for _ in range(LINES_COUNT_TEST)]
>>> grid_random = Grid(GRID_INIT_TEST)
>>> grid_random.fill_random(list(range(2)))
>>> grid_random.grid
[[1, 0, 1, 1, 0, 1, 0], [1, 0, 0, 0, 1, 1, 0], [1, 0, 1, 0, 0, 1, 0],
[1, 1, 0, 0, 1, 0, 0], [0, 1, 0, 1, 0, 0, 1]]
```

3. Rajouter à la classe `Grid` une méthode `draw_with_turtle(cell_size:int, margin:int, show_values:bool)`. Son appel générera le dessin de la grille avec le module `turtle` comme l'instruction de l'exemple produit la figure ci-après : la grille est centrée avec `margin` pixels de marge. Les cases ont une taille de `cell_size` pixels. Les valeurs de la grille sont affichées au centre de chaque case uniquement si `show_values` a pour valeur `True`.

```
>>> grid_random.draw_with_turtle(100, 20, True)
```



4 CM04 (30/1/2023)

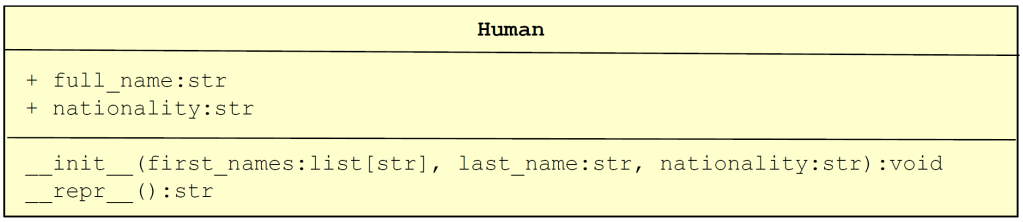
4.1 TP07 - Attributs et méthodes

Il s'agit dans ce TP d'expérimenter les notions suivantes :
- attributs publics / privés
- accesseurs / mutateurs
- attributs d'instance / de classe
- méthodes d'instance/de classe

Il est d'usage de mettre une seule classe par fichier. Au contraire, dans le cadre de cet exercice, vous coderez et ferez évoluer les 3 classes `Human`, `Cow` et `Dog` dans un même fichier. Vous testerez dans le `main` au fur et a mesure chaque nouvelle méthode en vous inspirant des exemples.

4.1.1 Attributs publics *vs* privés, accesseurs, mutateurs

1. Réalisez la classe `Human` de manière à ce qu'elle corresponde précisément au diagramme de classe et aux tests ci-après.

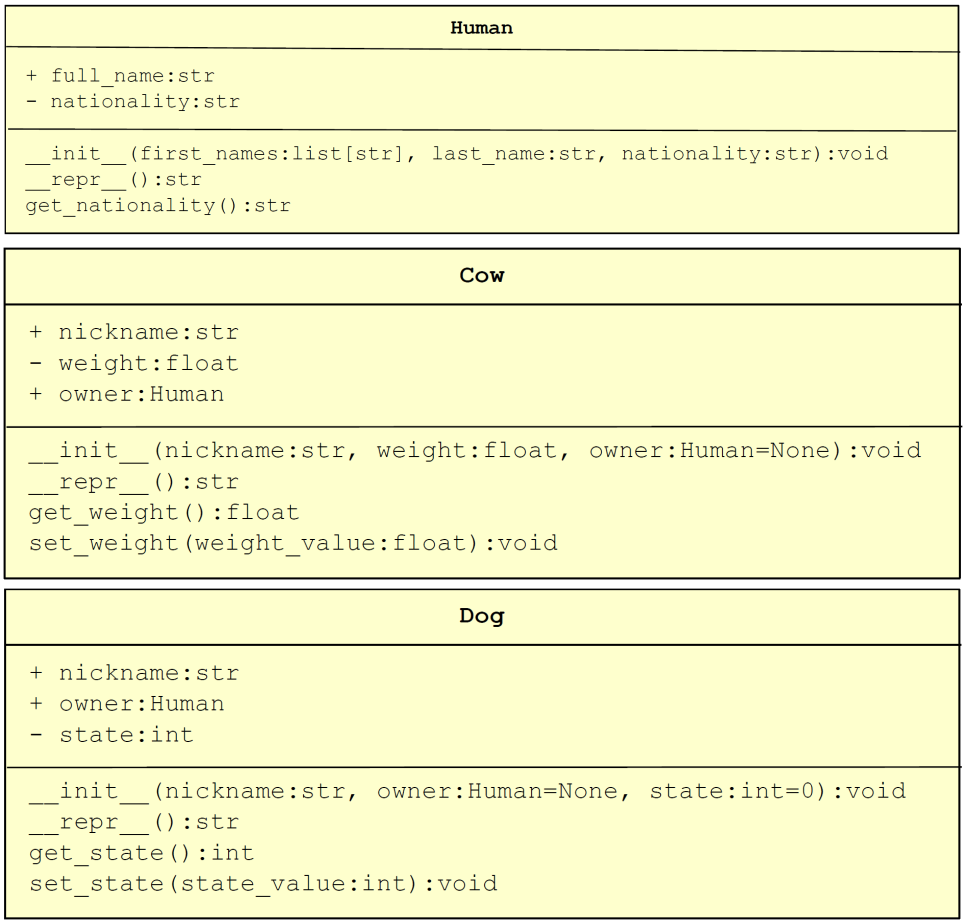



```
>>> farmer = Human(["Marcel", "Robert"], "Duchamps", "française")
>>> print(farmer.full_name)
"Marcel Robert Duchamps"
>>> print(farmer.nationality)
"française"
>>> farmer.nationality = "anglaise"
>>> print(farmer)
"Je m'appelle Marcel Robert Duchamps et je suis de nationalité anglaise."
```

2. Nous souhaitons rendre impossible le changement de nationalité de l'exemple. Il faut pour cela rendre cet attribut « invisible » à l'extérieur de la classe, autrement dit le rendre **privé**. Modifier la classe en conséquence.
3. Il n'est maintenant plus possible de modifier directement la nationalité. Mais il est du coup impossible également de l'afficher ! Modifier la classe en ajoutant une méthode **accesseur** `get_nationality` qui permettra d'afficher la valeur de l'attribut `nationality` depuis le programme principal. Nous ne rajouterons **pas de méthode mutateur** pour que la nationalité reste non modifiable.

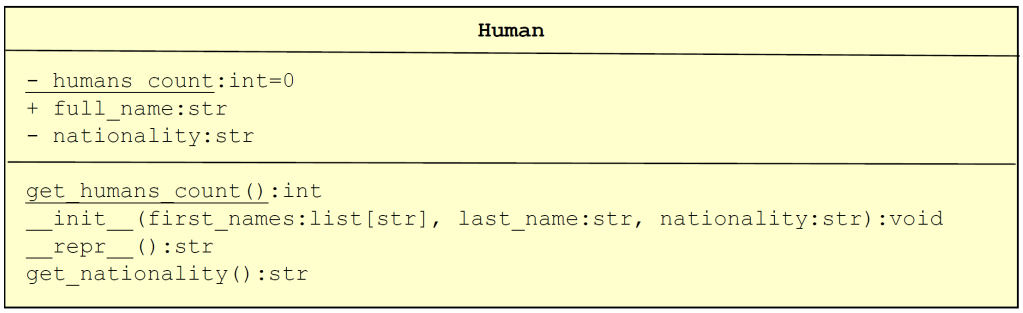
```
>>> farmer = Human(["Marcel", "Robert"], "Duchamps", "française")
>>> print(farmer.__nationality)
AttributeError: 'Human' object has no attribute '__nationality'.
>>> print(farmer.get_nationality())
française
```

4. Sur le modèle de la classe modifiée `Human`, réaliser les classes `Dog` et `Cow` en respectant scrupuleusement les diagrammes et les tests suivants. Vous profiterez en particulier des mutateurs des attributs privés `weight` et `state` pour contrôler que leur modification n'entraîne pas d'incohérence : un poids ne peut être négatif et l'état d'un chien ne peut prendre que les valeurs 0 (calme) ou 1 (énervé).




```
>>> farmer = Human(["Marcel", "Robert"], "Duchamps", "française")
>>> print(farmer)
Je m'appelle Marcel Robert Duchamps et je suis de nationalité française.
>>> milk_cow = Cow("Aglaë", 300, farmer)
>>> milk_cow.set_weight(302.3)
>>> print(milk_cow)
Aglaë : cow de 302.3 Kg. Appartient à Marcel Robert Duchamps.
>>> stray_dog = Dog("Médor", state=0)
>>> print(stray_dog)
Médor : dog cool. N'a pas de propriétaire.
>>> stray_dog.set_state(1)
>>> print(stray_dog)
Médor : dog en colère. N'a pas de propriétaire.
```

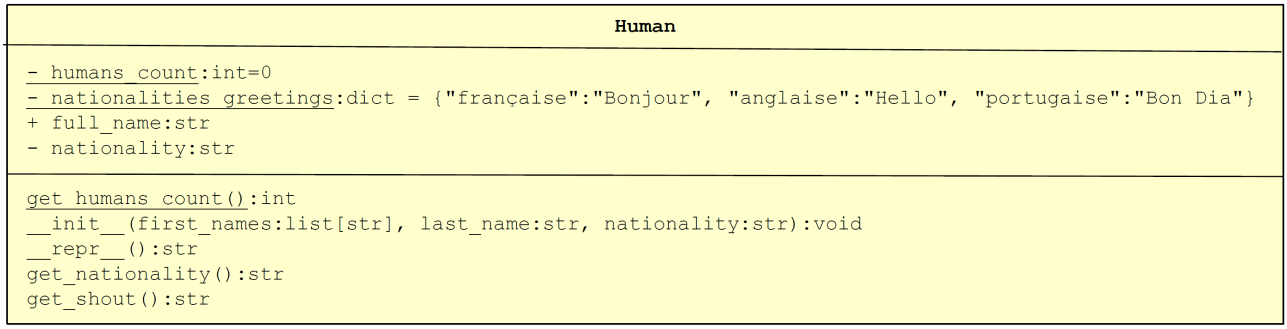
4.1.2 Attributs et méthodes d'instance *vs* de classe

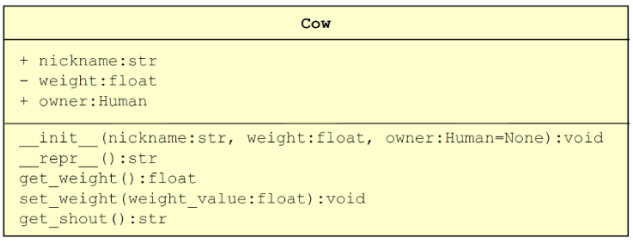


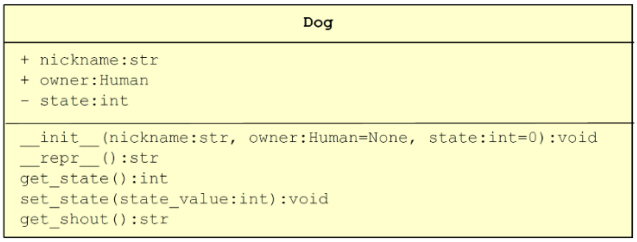
1. Nous souhaitons stocker dans un attribut le nombre d'humains créés avec la classe `Human`. Cet attribut ne sera donc pas « encapsulé » avec les autres attributs (dits d'instance) afin qu'il puisse être lié directement et une seule fois à la classe elle même plutôt que répété dans chaque objet. Autrement dit, cet attribut (dit de classe) existera même si aucun objet n'existe encore (sa valeur est dans ce cas de 0) ! Intégrez l'attribut `humans_count` à la classe `Human` de la manière imposée par la section dédiée aux attributs du nouveau diagramme.
2. Modifiez le constructeur de la classe `Human` afin qu'il incrémente de 1 l'attribut `humans_count` à chaque création d'un nouvel objet.
3. Faire la méthode accesseur `get_humans_count` de l'attribut de classe `humans_count`. Vous observerez dans la première instruction de l'exemple que celle-ci n'a pas besoin d'un objet pour fonctionner (aucune utilisation de `self` dans le corps de la méthode). Il s'agit donc d'une méthode dite de classe et non d'instance car elle ne manipule que des attributs de classe et aucun attribut d'instance. Utilisez la syntaxe associée à `@classmethod` pour intégrer cette remarque à votre code. Il n'y aura pas de méthode mutateur supplémentaire pour qu'il ne soit pas possible de modifier le compte des humains autrement qu'en créant un nouvel objet.

```
>>> print(Human.get_humans_count())
0
>>> farmer = Human(["Marcel", "Robert"], "Duchamps", "française")
>>> print(Human.get_humans_count())
1
>>> farmeress = Human(["Marcela"], "Delcampos", "portugaise")
>>> print(Human.get_humans_count())
2
```

4. Sur les mêmes principes, compléter les classes `Human`, `Cow` et `Dog` pour qu'elles correspondent aux 3 classes suivantes. L'objectif est de créer un dictionnaire en attribut de classe privé pour `Human` (sans accesseur, ni mutateur) qui aura pour vocation d'associer un terme de salutation à une nationalité. Dans notre exemple, le dictionnaire associera la valeur `"Bonjour"` à la clé `"française"`, la valeur `"Hello"` à la clé `"anglaise"` et la valeur `"Bon dia"` à la clé `"portugaise"`. Chaque classe se verra également ajouter une méthode `get_shout` qui produira une chaîne de caractère comme indiquée par l'exemple final.



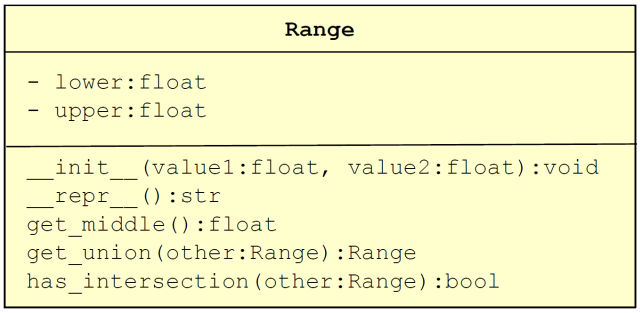




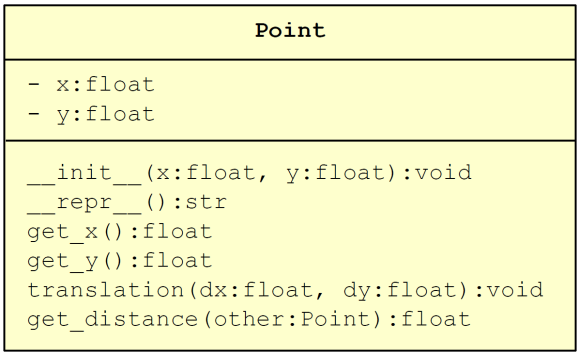

```
>>> farmer = Human(["Marcel"], "Duchamps", "française")
>>> print(farmer)
Je m'appelle Marcel Duchamps et je suis de nationalité française.
>>> print(farmer.get_shout())
– Je m'appelle Marcel Duchamps et je suis de nationalité française. Bonjour !
>>> farmeress = Human(["Marcela"], "Delcampos", "portugaise")
>>> print(farmeress)
Je m'appelle Marcela Delcampos et je suis de nationalité portugaise.
>>> print(farmeress.get_shout())
– Je m'appelle Marcela Delcampos et je suis de nationalité portugaise. Bon Dia !
>>> milk_cow = Cow("Aglaë", 300, farmer)
>>> print(milk_cow)
Aglaë : cow de 300 Kg. Appartient à Marcel Duchamps.
>>> print(milk_cow.get_shout())
– Meuuuuuuuuuuuuh !
>>> stray_dog = Dog("Médor")
>>> print(stray_dog)
Médor : dog cool. N'a pas de propriétaire.
>>> print(stray_dog.get_shout())
– ouah ouah !
>>> stray_dog.set_state(1)
>>> print(stray_dog)
Médor : dog en colère. N'a pas de propriétaire.
>>> print(stray_dog.get_shout())
– grrrrr !
```

4.2 TP08 - Des classes et encore des classes

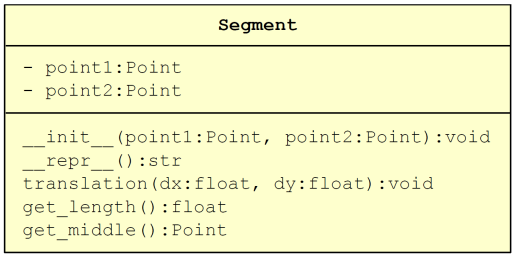
1. Codez les 3 classes suivantes en interprétant les noms des attributs et des méthodes, en respectant les diagrammes, en étudiant les exemples donnés et en réalisant vos propres tests.




```
>>> range_test1, range_test2 = Range(18.2, 5), Range(10, 20)
>>> print(range_test1)
[5,18.2]
>>> print(range_test2)
[10,20]
>>> print(range_test1.get_middle())
11.6
>>> print(range_test1.get_union(range_test2))
[5,20]
>>> print(range_test1.has_intersection(range_test2))
True
```




```
>>> point_test1, point_test2 = Point(1, 1), Point(-1, 1)
>>> print(point_test1)
(1,1)
>>> print(point_test2)
(-1,1)
>>> point_test1.translation(-1, 1)
>>> print(point_test1)
(0,2)
>>> print(point_test1.get_distance(point_test2) == 2 ** 0.5)
True
```




```
>>> segment_test = Segment(point_test1, point_test2)
>>> print(segment_test)
[(0,2);(-1,1)]
>>> segment_test.translation(2, 1)
>>> print(segment_test)
[(2,3);(1,2)]
>>> print(segment_test.get_length() == 2 ** 0.5)
True
>>> print(segment_test.get_middle())
(1.5, 2.5)
```

2. Un système de *Lindenmayer* est un système de réécriture constitué d'un axiome (*i.e.* un mot initial) et d'un ensemble de règles qui spécifie pour certains caractères quels mots (éventuellement vides) vont les remplacer. A chaque étape, le mot courant est réécrit suivant ces règles.

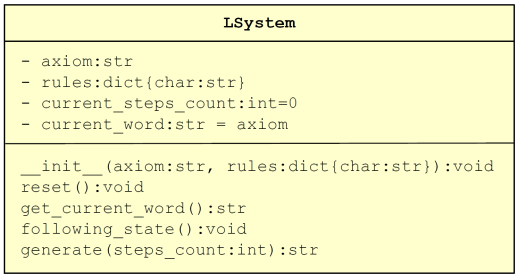
   Exemple

   Soit l'axiome : `'fx'` et l'ensemble de règles : `{'f':'', 'x':'-fx++fy-', 'y':'+fx--fy+'}`.

   A chaque étape, chaque occurrence de `'f'` est remplacée par la chaîne vide (règle `'f':''`), chaque occurrence de `'x'` est remplacée par `'-fx++fy-'` (règle `'x':'-fx++fy-'`) et chaque occurrence de `'y'` est remplacée par `'+fx--fy+'` (règle `'y':'+fx--fy+'`). Aucun changement pour les autres caractères.

   — A l'étape 0, le mot initial est l'axiome `'fx'`.
   — A l'étape 1, le mot courant est `'-fx++fy-'`.
   — A l'étape 2, le mot courant est `'--fx++fy-+++fx--fy+-'`
   — ...

   Codez la classe `LSystem` en respectant le diagramme ci-après.



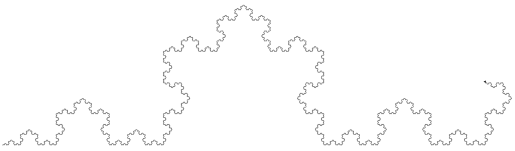

```
>>> AXIOM_TEST = 'fx', RULES_TEST = {'f': '', 'x': '-fx++fy-', 'y': '+fx--fy+'}
>>> LSystem(AXIOM_TEST, RULES_TEST).generate(3)
--fx++fy-+++fx--fy+-+++-fx++fy---+fx--fy++-
```

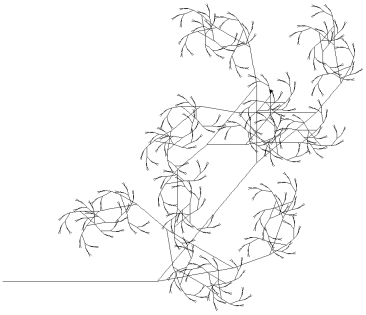
3. Nous souhaitons faire une classe `MyTurtleMemory` avec un attribut principal de type `Turtle` dont les mouvements seront commandés par des chaînes de caractères. Chaque caractère sera associé à une action de la tortue :
   — `'f'` : la tortue avance d'une longueur `length` en pixel.
   — `'+'` : la tortue tourne à gauche d'un angle `angle` en degré.
   — `'-'` : la tortue tourne à droite d'un angle `angle` en degré.
   — `'['` : la tortue ajoute sa position courante à la fin d'une liste `stack`.
   — `']'` : la tortue se repositionne (sans dessiner) sur la dernière position mémorisée dans la liste `stack` et la supprime de la liste. Il faut obligatoirement un nombre identique de caractères `'['` et `']'`.

   Codez la classe décrite par le diagramme ci-après. Quelques exemples sont donnés par les images qui suivent.

MyTurtleMemory
+ turtle:Turtle - stack:list
__init__():void reset():void draw_word(x:int, y:int, word:str, length:int, angle:float):void draw_star(x:int, y:int, length:int, branches_count:int) :void draw_l_system(x:int, y:int, l_system:LSystem, depth:int, length:int, angle:float):void

```
>>> MyTurtleMemory().draw_star(0, 0, 100, 12)
>>> AXIOM_TEST = 'f—f—f', RULES_TEST = {'f': 'f+f—f+f'}
>>> l_system_test = LSystem(AXIOM_TEST, RULES_TEST)
>>> MyTurtleMemory().draw_l_system(-500, 100, l_system_test, 3, 5, 60)
>>> AXIOM_TEST = 'x', RULES_TEST = {'x':'f[+x]f[-x]+x', 'f':'ff'}
>>> l_system_test = LSystem(AXIOM_TEST, RULES_TEST)
>>> MyTurtleMemory().draw_l_system(-500, 100, l_system_test, 7, 6, 10)
```





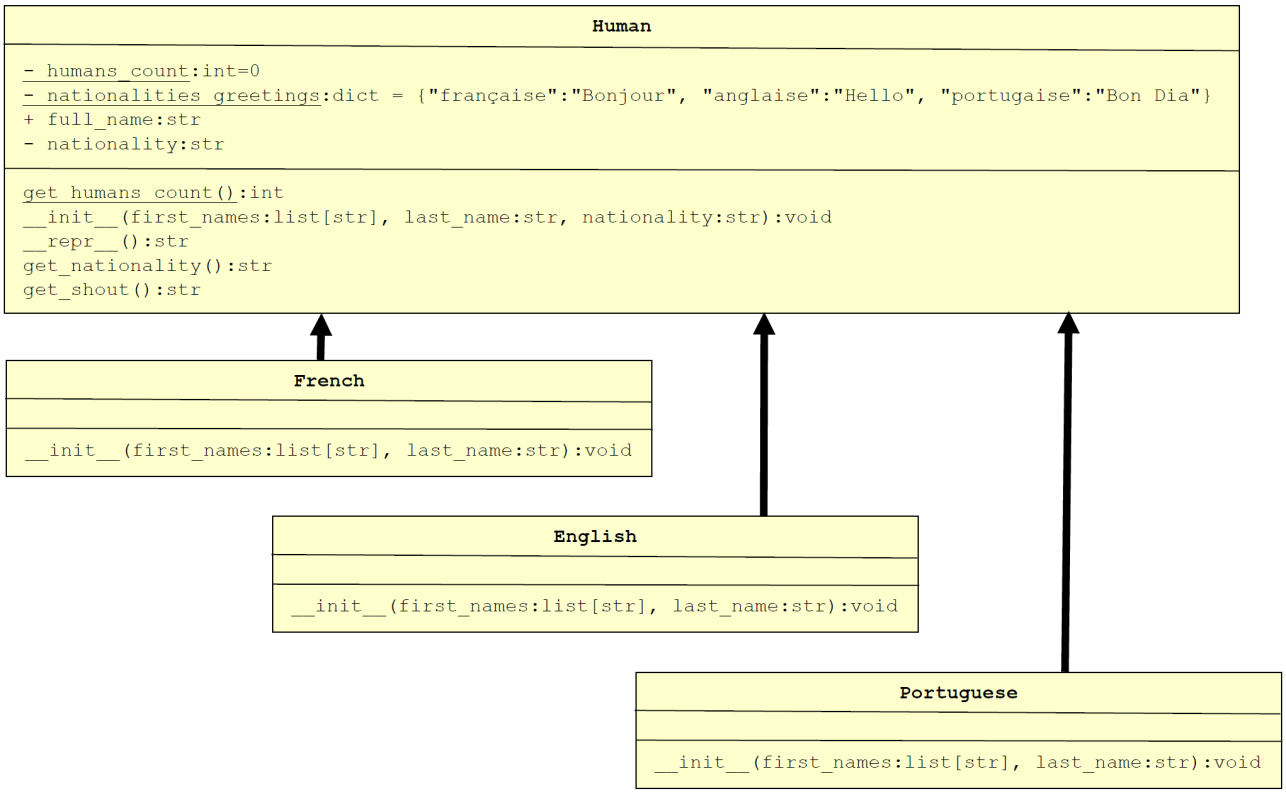
5 CM05 (6/2/2023)

5.1 TP09 - Agrégation et héritage simple

Dans les diagrammes de classe à venir, pour ne pas surcharger les schémas, les attributs et méthodes ne seront pas toujours précisés si la classe a déjà été faite.

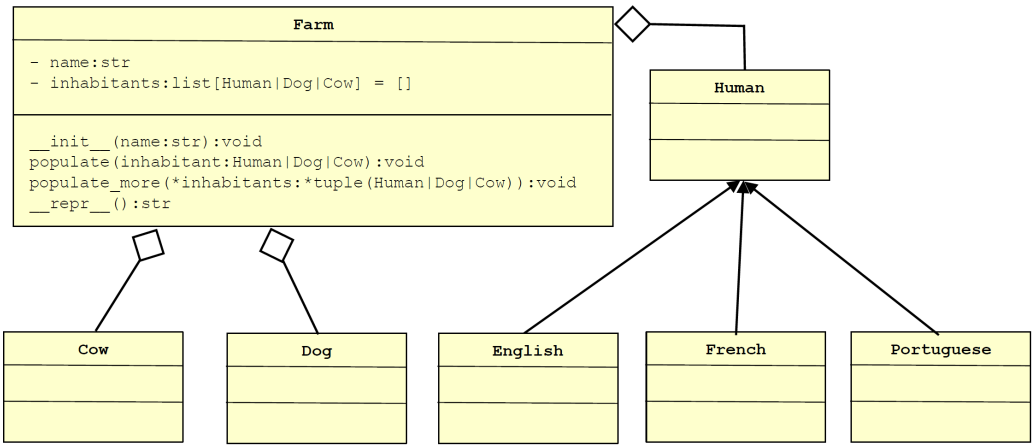
5.1.1 La ferme !

1. Dans un nouveau fichier `S05_TP09_farm.py`, importer les classes `Human`, `Dog` et `Cow` du TP07. Utilisez les pour réaliser les classes `French`, `English` et `Portuguese` du diagramme.



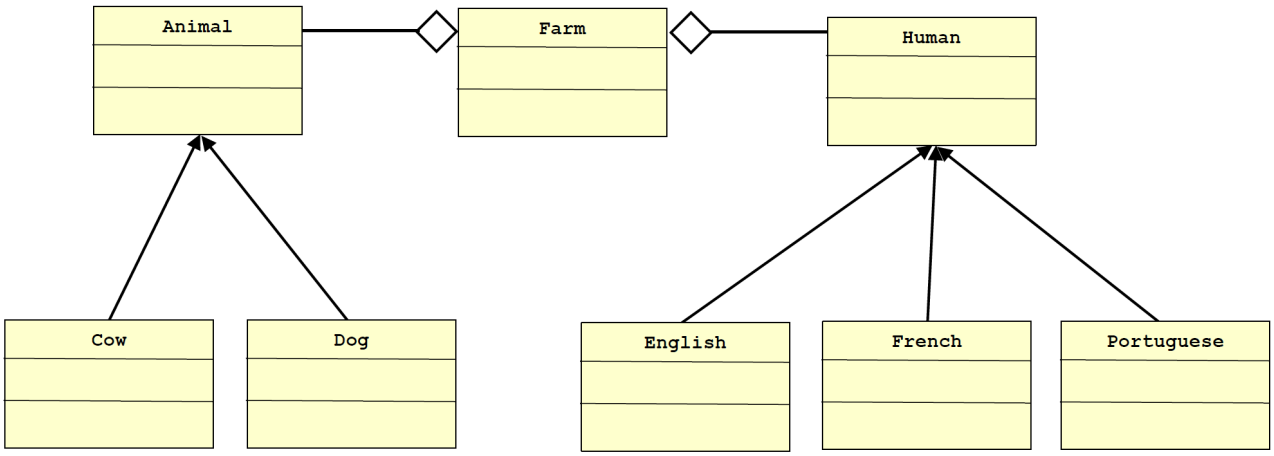

```
>>> farming_couple = (French(["Marcel", "Robert"], "Duchamps"), Portuguese(["Marcela"],
    "Delcampos"))
>>> english_tenant_farmer = English(["Singlet"], "Fromfield")
>>> print(farming_couple[0].get_shout())
- Je m'appelle Marcel Robert Duchamps et j'ai la nationalité française. Bonjour !
>>> print(farming_couple[1].get_shout())
- Je m'appelle Marcela Delcampos et j'ai la nationalité portugaise. Bon Dia !
>>> print(english_tenant_farmer.get_shout())
- Je m'appelle Singlet Fromfield et j'ai la nationalité anglaise. Hello !
```

2. Faire la classe `Farm` pour qu'elle respecte le diagramme de classe et l'exemple final.




```
>>> farm = Farm("Fermarcel")
>>> farm.populate(english_tenant_farmer)
>>> farm.populate_more(*farming_couple, Cow("Aglaë", 300, farming_couple[0]), Dog("Médor",
    english_tenant_farmer))
>>> print(farm)
Les habitants de la ferme Fermarcel se retrouvent :
- Je m'appelle Singlet Fromfield et j'ai la nationalité anglaise. Hello !
- Je m'appelle Marcel Robert Duchamps et j'ai la nationalité française. Bonjour !
- Je m'appelle Marcela Delcampos et j'ai la nationalité portugaise. Bon Dia !
- Meuuuuuuuuuuuuh !
- ouah ouah !
```

3. En vous inspirant de `Human` et ses sous-classes, revenez dans le fichier du TP07 pour faire une classe Animal qui regroupe les éléments communs à `Cow` et à `Dog` (les attributs `nickname` et `owner` ainsi qu'une partie de la méthode `__repr__`). Ces deux dernières classes devront être modifiées **un minimum** pour hériter de `Animal` et passer les tests de l'exemple à écrire dans le TP09.



```
>>> stray_dog = Dog("Médor", state=1)
>>> milk_cow = Cow("Aglaë", 300, english_tenant_farmer)
>>> print(stray_dog)
Médor n'a pas de propriétaire. C'est un chien en colère.
>>> print(milk_cow)
Aglaë appartient à Singlet Fromfield. C'est une vache de 300 Kg.
```

4. Construisez la sous-classe `Chicken` qui spécialise `Animal`. Un tel animal fait *Cocorico* quand c'est un mâle mais *cot cot cot codec* si c'est une femelle.

```
>>> pullet = Chicken("Cocotte", 0, farming_couple[0])
>>> cockerel = Chicken("Roadkill", 1, farming_couple[1])
>>> print(pullet)
Cocotte appartient à Marcel Robert Duchamps. C'est une poulette.
>>> print(cockerel)
Roadkill appartient à Marcela Delcampos. C'est un coquelet.
>>> farm.populate_more(pullet, cockerel)
>>> print(farm)
Les habitants de la ferme Fermarcel se retrouvent :
- Je m'appelle Singlet Fromfield et j'ai la nationalité anglaise. Hello !
- Je m'appelle Marcel Robert Duchamps et j'ai la nationalité française. Bonjour !
- Je m'appelle Marcela Delcampos et j'ai la nationalité portugaise. Bon Dia !
- grrrrr !
- Meuuuuuuuuuuuuh !
- cot cot cot codec !
- cocorico !
```

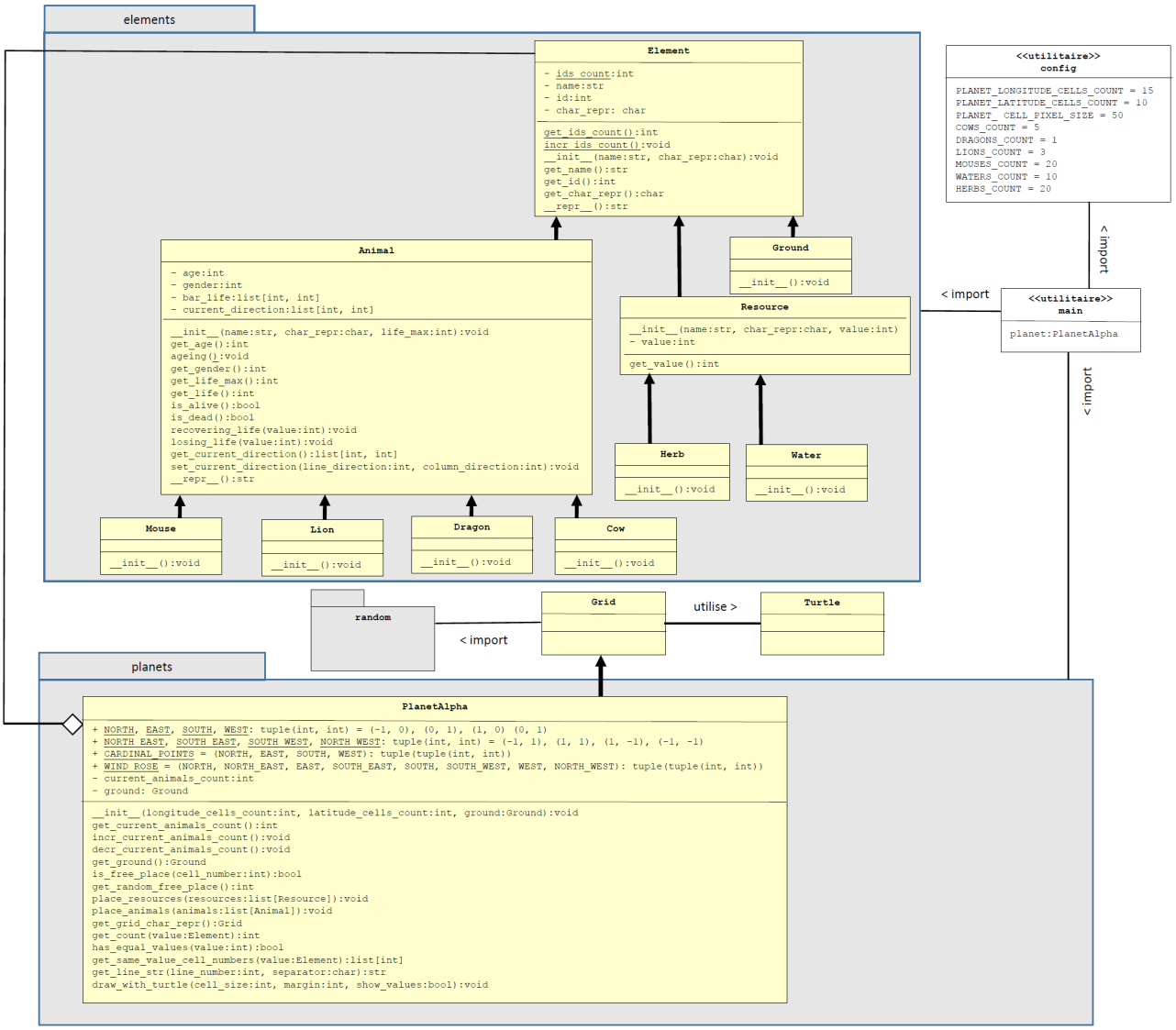
5.2 TP10 - planète peuplée d'animaux et de ressources

5.2.1 Mis en place des fichiers

Dans votre architecture, vous créerez un répertoire `projet` dans lequel vous déposerez 5 fichiers :

- Un fichier `config.py` qui contiendra les variables globales utiles pour tester différentes configurations du logiciel.
- un ficher `elements.py` qui contiendra la hiérarchie des classes qui permet de spécialiser progressivement les classes correspondant aux éléments qui vont peupler la planète (animaux, ressources, terrains...) : `Element`, `Ground`, `Resource`, `Animal`, `Herb`, `Water`, `Mouse`, `Lion`, `Dragon` et `Cow`.
- le fichier `grid.py` contenant la classe `Grid` dont le corrigé est sur *ecampus*.
- un fichier `planets.py` qui contiendra la hiérarchie de classe basée sur `Grid` correspondant aux planètes (pour l'instant uniquement la classe `PlanetAlpha`).
- un fichier `main.py` qui importera les fichiers `config.py`, `elements.py` et `planets.py` pour tester la création et l'affichage d'une `PlanetAlpha` peuplée d'animaux et de ressources.

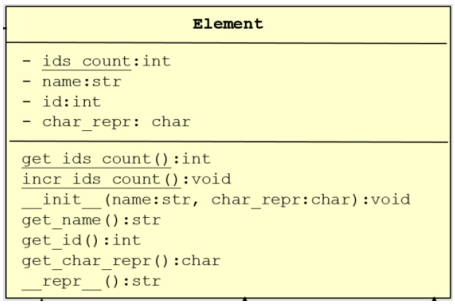
Vous commencerez par organiser l'architecture complète, puis écrirez dans les fichiers `elements.py` et `planets.py` les entêtes de toutes les classes (contenant uniquement l'instruction `pass`). Complétez ensuite le fichier `config.py` et préparez le fichier `main.py`. Pour l'ensemble de ces actions, vous pourrez vous aider du diagramme complet ci-après.



5.2.2 Le fichier `elements.py`

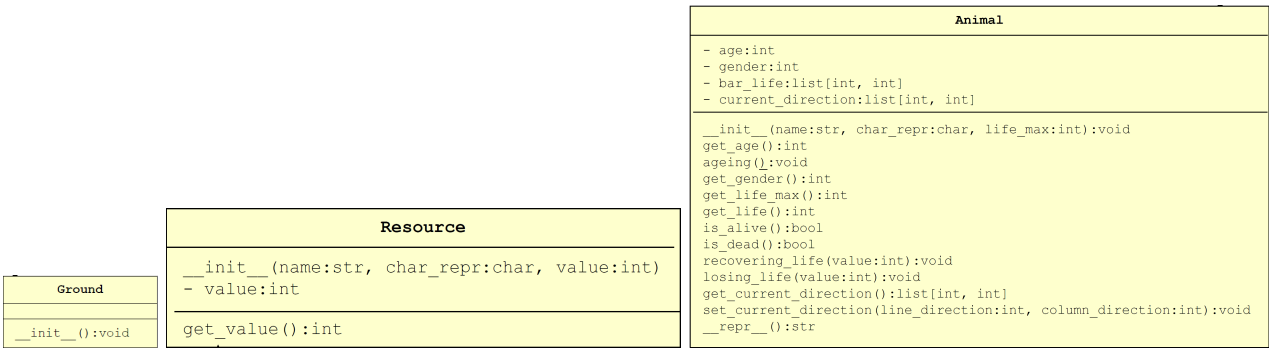
Un `Element` factorise tout ce qui sera commun aux objets qui peupleront la planète. Tous auront un identifiant unique numérique (`id` affecté à partir de l'attribut de classe `ids_count`). Ils ont également un `name` et un `char_repr` qui servira pour représenter l'élément sous la forme d'un caractère (de nombreuses ressources sont à votre disposition sur le Web pour trouver des points *unicode* 32 bits sous leur forme hexadécimale – par exemple en python le caractère représentant un dragon est représenté par la chaîne `"\U0001F432"`).

Les méthodes autres que le constructeur sont les accesseurs et les mutateurs utiles et la méthode `__repr__` pour afficher un `Element` comme dans l'exemple.




```
>>> print(Element('Elem', 'X'))
X : Elem 1
>>> print(Element('Elem', 'Y'))
Y : Elem 2
```

Les classes `Ground`, `Resource` et `Animal` sont des `Element` spécialisés respectivement pour représenter les terrains inoccupés, les ressources et les les animaux. En cas de doutes sur un attribut référez vous à l'exemple. La plupart des méthodes sont des accesseurs ou des mutateurs.



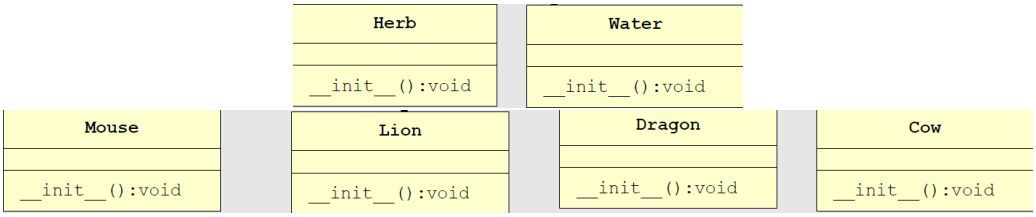
Quelques précisions pour la classe `Animal` :

- l'âge est toujours initialisé à 0
- un animal vieillit obligatoirement de 1 a chaque appel de `ageing`
- le genre est choisi aléatoirement avec les valeurs 0 pour une femelle et 1 pour un mâle
- `bar_life` est une liste de 2 valeurs entières dont la première représente les points de vie courants (retournés par la méthode `get_life` et la seconde les points de vie initiaux (qui ne peuvent en aucun cas être dépassés et qui sont retournés par la méthode `get_life_max`)
- un animal meurt dès que ses points de vie courants tombent à 0 ou une valeur inférieure
- les méthodes `losing_life` et `recovering_life` sont les mutateurs de l'attribut `bar_life` en modifiant à la hausse ou à la baisse la première valeur correspondant aux points de vie courrants.
- `current_direction` est une liste de deux valeurs entières dont la première est la direction en terme de ligne et la seconde celle en terme de colonne (chaque valeur entière sera donc de -1, 0 ou 1 mais les deux ne peuvent pas être simultanément à 0). L'initialisation est aléatoire.

Les affichages devront se faire pour respecter l'exemple suivant.

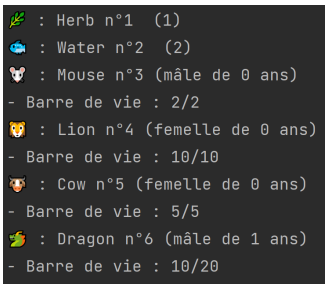
```
>>> print(Ground())
. : Ground 1
>>> print(Resource('Water', 'W', 10))
W : Water 2  (10)
>>> a = Animal('Dragon', 'D', 30)
>>> a.ageing()
>>> a.losing_life(10)
>>> a.recovering_life(5)
>>> print(a)
D : Dragon 3 (femelle de 1 an(s))
- Barre de vie : 25/30
```

Les 6 classes `Herb`, `Water`, `Mouse`, `Lion`, `Cow` et `Dragon` sont les classes finales de cette hiérarchie. Ce sont les `Element` qui peupleront concrètement leur planète d'accueil. Aucune n'a besoin de paramètres fournis au constructeur. Ce dernier se contentera d'appeler le constructeur de sa classe mère `Resource` ou `Animal` en fixant les paramètres à la valeur désirée pour l'attribut correspondant.



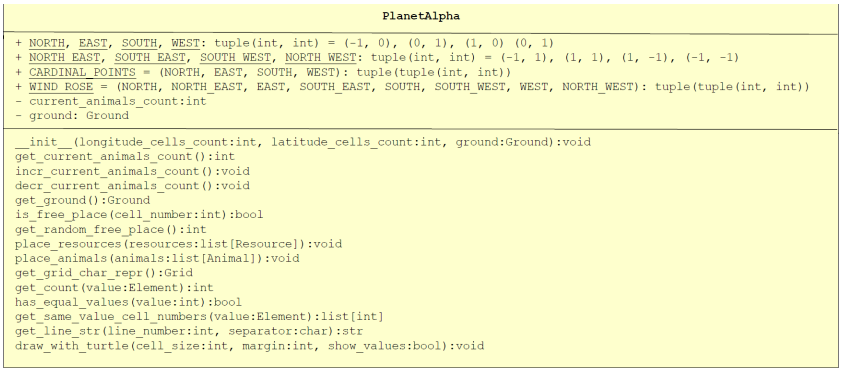
La série d'instruction de l'exemple produiront l'affichage de l'image qui suit. Les caractères utilisés pour la représentation des 6 éléments sont respectivement : `"\U0001F33F"`, `"\U0001F41F"`, `"\U0001F42D"`, `"\U0001F981"`, `"\U0001F42E"`, `"\U0001F432"`.

```
>>> print(Herb())
>>> print(Water())
>>> print(Mouse())
>>> print(Lion())
>>> print(Cow())
>>> a = Dragon()
>>> a.ageing()
>>> a.losing_life(10)
>>> print(a)
```

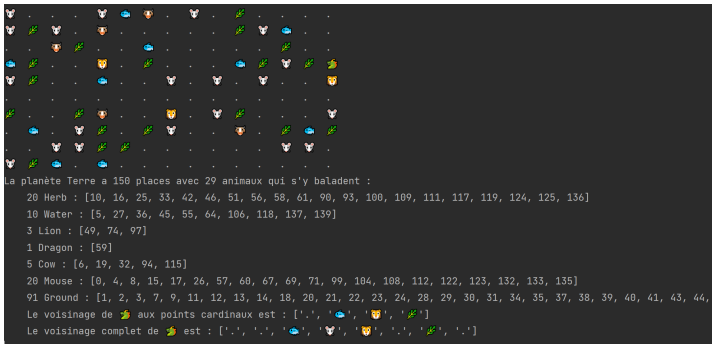


5.2.3 Le fichier `planets.py`

La classe `PlanetAlpha` ci-après devra respecter les conditions exprimées par le programme du fichier `main.py` et les résultats de son exécution (code et image qui suit le diagramme de classe).




```python
import random
from config import *
from elements import Ground, Herb, Water, Cow, Mouse, Lion, Dragon
from planets import PlanetAlpha


if __name__ == "__main__":
    random.seed(1000)

    planet = PlanetAlpha('Terre', PLANET_LONGITUDE_CELLS_COUNT, PLANET_LATITUDE_CELLS_COUNT, Ground())
    planet.place_resources([Herb() for _ in range(HERBS_COUNT)])
    planet.place_resources([Water() for _ in range(WATERS_COUNT)])
    planet.place_animals([Lion() for _ in range(LIONS_COUNT)])
    planet.place_animals([Dragon() for _ in range(DRAGONS_COUNT)])
    planet.place_animals([Cow() for _ in range(COWS_COUNT)])
    planet.place_animals([Mouse() for _ in range(MOUSES_COUNT)])

    print(planet.get_grid_str())
    h, w, l, d, c, m, g = Herb(), Water(), Lion(), Dragon(), Cow(), Mouse(), Ground()
    print(f"La planète {planet.get_name()} a {planet.get_lines_count() * planet.get_column_count()} places", end=' ')
    print(f"avec {planet.get_current_animals_count()} animaux qui s'y baladent :")
    for element in (h, w, l, d, c, m, g):
        print(f"\t{planet.get_count(element)} {element.get_name()} : {planet.get_same_value_cell_numbers(element)}")
    dragon_cell_number = planet.get_same_value_cell_numbers(d)[0]
    line_dragon, column_dragon = planet.get_coordinates_from_cell_number(dragon_cell_number)
    print(f"\tLe voisinage de {d.get_char_repr()} aux points cardinaux est :", end=' ')
    print([element.get_char_repr()
           for element in planet.get_neighborhood(line_dragon, column_dragon, planet.CARDINAL_POINTS, True)])
    print(f"\tLe voisinage complet de {d.get_char_repr()} est :", end=' ')
    print([element.get_char_repr()
           for element in planet.get_neighborhood(line_dragon, column_dragon, planet.WIND_ROSE, True)])
    planet.draw_with_turtle(PLANET_CELL_PIXEL_SIZE)
```



Quelques précisions pour la classe `PlanetAlpha` :

- Les paires d'entier gérant les orientations sont définis en attributs de classe publics
- les deux seuls attributs d'instance permettent de connaître le nombre d'animaux en vie sur `PlanetAlpha` et l'`Element` représentant un bout de terre libre (*i.e* : cellule de la grille qui n'est occupé par aucune `Resource` ou `Animal`
- les méthodes `is_free_place` et `get_random_free_place` permettent respectivement de savoir si une place est disponible et de trouver une place libre aléatoirement parmi celles qui sont disponibles (-1 sera retourné s'il n'y en a aucune).
- `place_resources` et `place_animals` placent une liste de `Resource` et d'`Animal` aléatoirement dans les cellules disponibles de la grille `PlanetAlpha`

De plus certaines méthodes de `Grid` ne fonctionnent plus puisqu'elles n'étaient pas fondées sur des valeurs de type `Element`. Il faut donc les redéfinir. Une première approche serait de toutes les réécrire complètement. C'est ce que vous devez faire pour redéfinir `get_line_str` pour conserver un affichage correct. Une autre solution est d'écrire tout d'abord une méthode qui retourne un objet de type `Grid` mais dont les valeurs sont les caractères représentant les éléments plutôt que les éléments eux-mêmes. C'est le rôle de `get_grid_char_repr`. Il suffira ensuite de redéfinir les 4 autres méthodes problématiques (`get_count`, `has_equal_values`, `get_same_value_cell_numbers` et `draw_with_turtle`) en utilisant les méthodes de même nom sur la grille retournée par `get_grid_char_repr`.

6 CM05 (20/2/2023)

6.1 TP11 - Méthodes spéciales

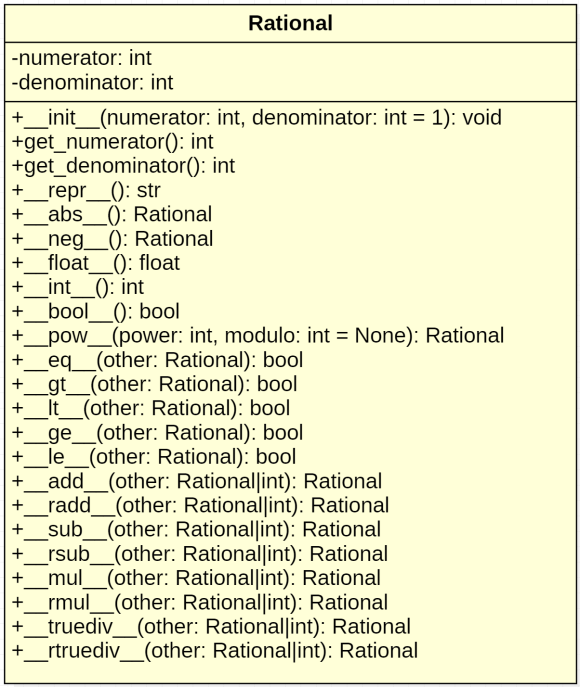
Ce TP sera l'occasion de réaliser une classe `Rational` pour modéliser la gestion des nombres rationnels sous la forme de deux entiers `numerator` et `denominator` qui représentent la fraction $\frac{numerator}{denominator}$. Il s'agira de faire en sorte que toutes les opérations classiques sur les `Rational` fonctionnent également. Pour cela nous allons exploiter toutes les méthodes spéciales appelées lors de l'utilisation des fonctions *built-in* ou des opérateurs énumérés ci-après.

- unaires : `print`, `abs`, `-`, `float`, `int`, `bool`, `pow`
- binaires : `==`, `!=`, `<`, `>`, `<=`, `>=`, `+`, `-`

Pour rappel, si a et b sont deux `entiers relatifs` alors :

1. soit $p = pgcd(a, b)$, $\frac{a}{b}$ se réduit à $\frac{\frac{a}{p}}{\frac{b}{p}}$. Le `pgcd` se calcule avec l'*algorithme d'Euclide* basé sur le `modulo`.
2. $\frac{a}{b} \implies b \neq 0$
3. $\forall a, \frac{0}{a}$ s'écrit `0`
4. $\frac{a}{b}$ s'écrit $\frac{|a|}{|b|}$ `si` $\frac{a}{b} > 0$ sinon $-\frac{|a|}{|b|}$
5. $\frac{a}{1}$ s'écrit `a`

Appuyez vous sur ces règles, le diagramme de classe et les exemples pour construire la classe `Rational`.




```
>>> r1,r2, r3, r4 = Rational(0, 1), Rational(32, -24), Rational(-1, 5), Rational(10, -2)
>>> print(r1, r2, r3, r4)
0 -4/3 -1/5 -5
>>> print(abs(r2), -r2)
4/3 4/3
>>> print(float(r2))
-1.3333333333333333
>>> print(int(r2))
-1
>>> print(bool(r1), bool(r2))
False True
>>> print(r3 ** 2, r3 ** 3)
1/25 -1/125
>>> print(r3 ** 0 == r4 ** 0 == 1)
True
>>> print(r4 == Rational(-25, 5), r4 != Rational(-25, 5))
True False
>>> print(r4 > -4.5, r4 >= -5, r4 < -5, r4 <= -5)
False True False True
>>> print(r3 + r4)
-26/5
>>> print(r3 + 1, 1 + r3)
4/5 4/5
>>> print(r3 - r4, r4 - r3, r3 - 1, 1 - r3)
24/5 -24/5 -6/5 6/5
>>> print(r3 * r2)
4/15
>>> print(r3 * 2, 2 * r3)
-2/5 -2/5
>>> print(r2 / r3, r3 / r2, r3 / 2, 2 / r3)
20/3 3/20 -1/10 -10
```

6.2 TP12, 13 et 14 -Amélioration `ELement`, `PlanetAlpha` – groupes, pré-rapport et GIT

En vous appuyant sur le dernier pdf de cours (exemples, utilisation de GIT, rédaction du pré-rapport) :

1. vous commencerez par modifier la classe `Element` pour que la comparaison de deux objets de cette classe avec les signes == ou != teste l'identité de classe plutôt que l'identité des objets. Vous testerez avec des classes filles de `Element` comme dans l'exemple.

```
>>> print(Mouse().get_id() == Mouse().get_id())
False
>>> print(Mouse() == Mouse())
True
>>> print(Mouse() == Lion())
False
```

2. Modifiez la classe `PlanetAlpha` en adaptant les tests de 5.2.3 en vérifiant que :
   — la méthode `get_grid_char_repr` peut être supprimée
   — les 3 méthodes `get_count`, `has_equal_values` et `get_same_value_cell_numbers` ne nécessitent plus d'être redéfinies
   — les 2 méthodes `get_line_str` et `draw_with_turtle` peuvent être réécrites plus simplement
3. vous commencerez à organiser vos groupes (obligatoirement dans le même créneau horaire), votre dépôt GIT et réfléchir aux fonctionnalités de votre programme, aux diagrammes de classe associés et à la répartition du travail dans le groupe (plusieurs animaux par case, déplacement, naissance et mort des animaux, stratégies de gestion physiologiques (faim, soif, fatigue) et de déplacement intelligent (fuite, combat, recherche partenaire, ennemi ou ressource...), interface graphique, génération de statistiques, ... Le pré-rapport sera récupéré sur le dépôt GIT à la date du 12 mars.