

YSINL2A1 : Initiation à la Programmation Orientée Objet (Python) – UNICAEN 2024 – 2025

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. 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/oral des fonctionnalités libres du projet fil rouge en trinôme (1/3 de la note globale).

Contenu : les 3 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 (POO).

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 des points clés de la plupart des TP sera faite à travers de nombreux exemples expliqués directement au tableau ou par projection des programmes lors du cours de la semaine correspondante.

Recommandations TP : des **fichiers .py** peuvent être fournis. Dans ce cas, ils contiennent les signatures des fonctions et les consignes en *docstring*). Il est **interdit** de modifier les signatures :

- le `__main__` sera sous la forme de variables et d'assertions de tests associées à chaque fonction. Vous ne devez pas modifier les tests mais le corps de vos fonctions afin que ceux-ci ne soient plus bloquants. Dans cette section, vous pouvez rajouter vos propres tests pour comprendre la provenance de vos erreurs.
- Les TP non terminés pendant les horaires réservés doivent être terminés sur son temps personnel avant le début du suivant (**les sections en rouge repérables comme optionnelles et/ou plus difficiles**).
- un dépôt GIT devra contenir l'ensemble des TP réalisés dans l'architecture imposée au fur et à mesure des semaines. L'idéal est de faire un *commit* après chaque fonction/méthode opérationnelle et un *push* lorsque le TP est terminé.

Recommandations projet fil rouge : Il s'agira de construire une application permettant la gestion de 3 mini-jeux basés sur une structure commune (un automate de *Conway*, un visualisateur de *Turmites* et un *Snake*)

- La structure sera **imposée** et construite au fur et à mesure des TP.
- Les derniers TP seront consacrés à la structure de base, par trinôme d'étudiants ; et votre temps personnel à l'ajout d'extensions. En amont, 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).
- La dernière semaine de TP sera consacrée à un oral pour individualiser la note finale du projet fil rouge.

Enseignants : F. Maurel (CM) - J.-M. Lecarpentier - S. Shupp - Romain Hérault - Djawab Bekkoucha - Mariana Brito Azevedo - Amal Mahboubi - Marjorie Redon - Yoann Jacquier

Programme - vacances : semaine du 17 février

Semaine 1 - 6/1/2025 : RAPPELS

- CM01 : Bases - séquences - référence/valeur, mutable/non mutable - fichiers
- TP01 : Chaînes de caractères et distance de *Hamming*
- TP02 : Échelles de mots avec une structure de liste - fichiers (mode lecture)

Semaine 2 - 13/1/2025 : RAPPELS

- CM02 : Ensembles - modules tiers (ex : `chardet`, `matplotlib`)
- TP03 : Échelles de mots avec une structure de dictionnaire - fichiers (mode écriture)
- TP04 : Encodage des caractères - histogrammes de caractères

Semaine 3 - 20/1/2025 : RAPPELS

- CM03 : Compréhension de liste (et de dictionnaire) - structure de grille - versionnage (GIT)
- TP05 : Gestion d'une liste de personnes
- TP06 : module `grid_manager.py` (gestion d'un tore 2D)

Semaine 4 - 27/1/2025

- CM04 : Classe et objet - UML - structure d'un POO - Associations
- TP07 : Impératif vs. OO : de `grid_manager` à `Grid`
- TP08 : La ferme

Semaine 5 - 3/2/2025

- CM05 : Agrégation - public/brouillé/privé, instance/classe/statique, *getter/setter*
- TP09 : Reprises et privatisations
- TP10 : Exercices supplémentaires

Semaine 6 - 10/2/2025

- CM06 : Héritage, redéfinition/surcharge
- TP11 : De `Grid` à `PlanetAlpha`
- TP12 : Agrégations et héritage par généralisation ou spécialisation

Semaine 7 - 24/2/2025

- CM07 : Spécialisation classes *built-in*, méthodes spéciales, POO et `tkinter`
- TP13 : Exercices d'application de `tkinter`
- TP14 : Affichage d'une `PlanetAlpha` d'Element

Semaine 8 - 3/3/2025

- CM08 : Multihéritage, classes abstraites, interfaces, présentation projet
- TP15 : De `PlanetAlpha` à `PlanetTk`
- TP16 : Kick-off projet fil rouge (groupes, GIT, dépôt pré-rapport)

Semaine 9 - 10/3/2025

- CM09 : `tkinter`, IHM et programmation événementielle
- TP17 et TP18 : Projet fil rouge

Semaine 10

- CM10 - 17/3/2025 : *Docstring* - *Pydoc* - Questions projet
- TP19 et TP20 - 31/3/2025 : Oral

1 CM01 (8/1/2024) - RAPPELS prog. impérative en Python 3.X

Notions vues en cours :

- 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`)
- passage des paramètres à une fonction par valeur *vs.* par référence
- les fichiers textes et leur accès en Python (`with`, `open`)

1.1 TP01 - bases - string (1h)

`S01_TP01_template.py` : fichier *template* à récupérer sur *ecampus* et à compléter. Seuls les signatures des fonctions et un programme `__main__` ne sont indiqués. Chaque signature est suivie d'une *docstring* qui rappelle ce qui est attendu. Il s'agit de remplacer l'instruction `pass` dans le corps de chaque fonction par le code Python qui permet de passer les tests qui lui sont associés.

L'interprétation du `__main__` s'arrête sur le premier test non passé. Lorsque toutes les méthodes seront écrites et les tests tous validés, le message "*Tests all OK*" s'affichera.

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. Elles sont utilisées dans le texte des questions de ce document pour les éclairer mais ne doivent pas être précisées explicitement dans votre code.

Exemple de codage de la signature `message(text: str, sender: str, times: int=1) → str` et résultats d'exécutions (instructions précédées de `>>>`) :

```
def message(text, sender, times=1):
    res = ''
    for i in range(times):
        res += f'{i+1}. {text}\n'
    return res + f'\t\t{sender}'
>>> print(message("Bonne année 2024 !", "ChatGPT"))
1. Bonne année 2024 !
    ChatGPT
>>> print(message("Je m'excuse de la confusion ! Bonne année 2025...", "ChatGréPT", 2))
1. Je m'excuse de la confusion ! Bonne année 2025...
2. Je m'excuse de la confusion ! Bonne année 2025...
    ChatGréPT
```

Remarque : dans l'exemple ci-dessus, utilisation pratique des *f-strings* possible depuis python 3.6.

Attention : respectez bien l'énoncé de la question lorsqu'il impose de réutiliser des fonctions déjà écrites dans le TP en cours voire dans un TP précédent !

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 Cytosine (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. **Il faudra utiliser la fonction `is_dna`.**

```
>>> 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', "aeeigmrrrstuwz")
"aeigmrrstuwz"
```

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 donc 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 ", " aeeigmnrstuwz ")
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 ", " aeeigmnrstuwz ")
13
```

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

resources.zip : cette archive est à décompresser dans un répertoire TP_P00. 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).

Trois autres fichiers doivent être rajoutés à la racine du répertoire TP_P00.

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_P00) 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.

Attention : Assurez-vous de respecter l'arborescence décrite ci-après 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
|    |---- OUT
|    |    |---- WORDS
|    |    |---- fr_long_dict_cleaned.txt
|---- 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

```

Les 7 fonctions suivantes ont pour finalité de construire des « échelles » entre deux mots. Il s'agit de trouver dans un fichier 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 des exemples 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_dictionary(file_path: str, length: int = None) → list[str]` | Retourne la liste des mots du fichier de nom `file_path` si `length` vaut `None`. Sinon retourne la liste des mots de longueur `length`.

```

>>> DICT_NAME = PATH_DICTIONARIES + "fr_long_dict_cleaned.txt"
>>> words6 = get_words_from_dictionary(DICT_NAME, 6)
>>> print(words6[:9])
[ 'A-T-IL' , 'ABAQUE' , 'ABATEE' , 'ABATTU' , 'ABBAYE' , 'ABCDE' , 'ABERRE' , 'ABETIE' ]

```

2. `get_words_hamming(word: str, words: list[str], hamming_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_words_hamming("ORANGE", words6, 0)
[ 'ORANGE' ]
>>> get_words_hamming("ORANGE", words6, 1)
[ 'FRANGE' , 'GRANGE' , 'ORANGS' , 'ORANTE' , 'ORONGE' ]
>>> get_words_hamming("ORANGE", words6, 2)
[ 'BRANDE' , 'BRANLE' , 'BRANTE' , 'CHANGE' , 'CRANTE' , 'GRANDE' , 'GRINGE' , 'ORACLE' , 'ORANTS' ,
'TRANSE' , 'URANIE' ]

```

3. `is_scale(scale: list[str]) → bool` | Retourne `True` si `scale` est une échelle de mot correctement construite. `False` sinon.

```

>>> is_scale(['SUD', 'SUT', 'EUT', 'EST'])
True
>>> is_scale(['FRANGE', 'GRANGE', 'ORANGS', 'ORANTE', 'ORONGE'])
False

```

4. `is_perfect_scale(scale: list[str]) → bool` | Retourne `True` si l'échelle de mots `scale` est parfaite. `False` sinon. Une échelle de mots est dite parfaite si le nombre d'étape pour passer du mot de départ au mot cible est égal à leur distance de *hamming*.

```
>>> is_perfect_scale(['SUD', 'SUT', 'EUT', 'EST'])
True
```

5. `get_removed_words(words_to_remove: list[str], all_words: list[str]) → list[str]` | Retourne une sous-liste des mots de `all_words` en retirant ceux de `words_to_remove`.

```
>>> new_word6 = get_removed_words(['A-T-IL', 'ABATTU'], word6)
>>> print(words6[:9])
['A-T-IL', 'ABAQUE', 'ABATEE', 'ABATTE', 'ABATTU', 'ABBAYE', 'ABCDE', 'ABERRE', 'ABETIE']
>>> print(new_word6[:9])
['ABAQUE', 'ABATEE', 'ABATTE', 'ABBAYE', 'ABCDE', 'ABERRE', 'ABETIE', 'ABETIR', 'ABETIS']
```

6. `get_next_scales(scale: list[str], words: str) → list[list[str]]` | retourne la liste des échelles de mots possibles constituées par l'échelle de mot `scale` et un mot de la liste `words`.

```
>>> get_next_scales(['CHANGE', 'CHANTE'], word6)
[[['CHANGE', 'CHANTE', 'CHANCE'], ['CHANGE', 'CHANTE', 'CHANTA'], ['CHANGE', 'CHANTE', 'CHANTS'],
  ['CHANGE', 'CHANTE', 'CHARTE'], ['CHANGE', 'CHANTE', 'CHASTE'], ['CHANGE', 'CHANTE', 'CHATTE'],
  ['CHANGE', 'CHANTE', 'CRANTE']]]
```

7. `get_scale(file_path: str, word1: str, word2: str) → list[str]` | Retourne une échelle de mots entre `word1` et `word2` avec les mots du dictionnaire `file_path`. Le principe de l'algorithme est d'enfiler les échelles à tester dans une liste (initialisée avec l'échelle contenant le seul mot `word1`). Tant qu'il reste des échelles à tester, la première est sélectionnée. Si elle se termine par le mot `word2` alors cette solution est retournée ; sinon toutes les échelles construites à partir de celle-ci plus un mot sont enfilées à leur tour en attendant d'être testées. Si aucune échelle n'est finalement trouvée `None` est retourné. Dans l'exemple vous remarquerez cependant grâce aux tests avec `perf_counter` (commentés dans le template) que certains temps de calcul, même sur un ordinateur puissant, ne sont pas raisonnables.

```
>>> t1 = perf_counter()
>>> print(get_scale(DICT_NAME, 'SUD', 'EST'))
['SUD', 'SUT', 'EUT', 'EST']
>>> t2 = perf_counter()
>>> print(t2 - t1)
0.41285749999951804
>>> print(get_scale(DICT_NAME, 'HOMME', 'SINGE'))
['HOMME', 'COMME', 'COMTE', 'CONTE', 'CONGE', 'SONGE', 'SINGE']
>>> t3 = perf_counter()
>>> print(t3 - t2)
32.78062040000077
>>> print(get_scale(DICT_NAME, 'EXOS', 'MATH'))
['EXOS', 'EROS', 'GROS', 'GRIS', 'GAIS', 'MAIS', 'MATS', 'MATH']
>>> t4 = perf_counter()
>>> print(t4 - t3)
212.4659079999983
>>> print(get_scale(DICT_NAME, 'TOUT', 'RIEN'))
['TOUT', 'BOUT', 'BRUT', 'BRUN', 'BREN', 'BIEN', 'RIEN']
>>> t5 = perf_counter()
>>> print(t5 - t4)
666.6731024999972
```

2 CM02 (15/1/2024) - RAPPELS prog. impérative en Python 3.6>

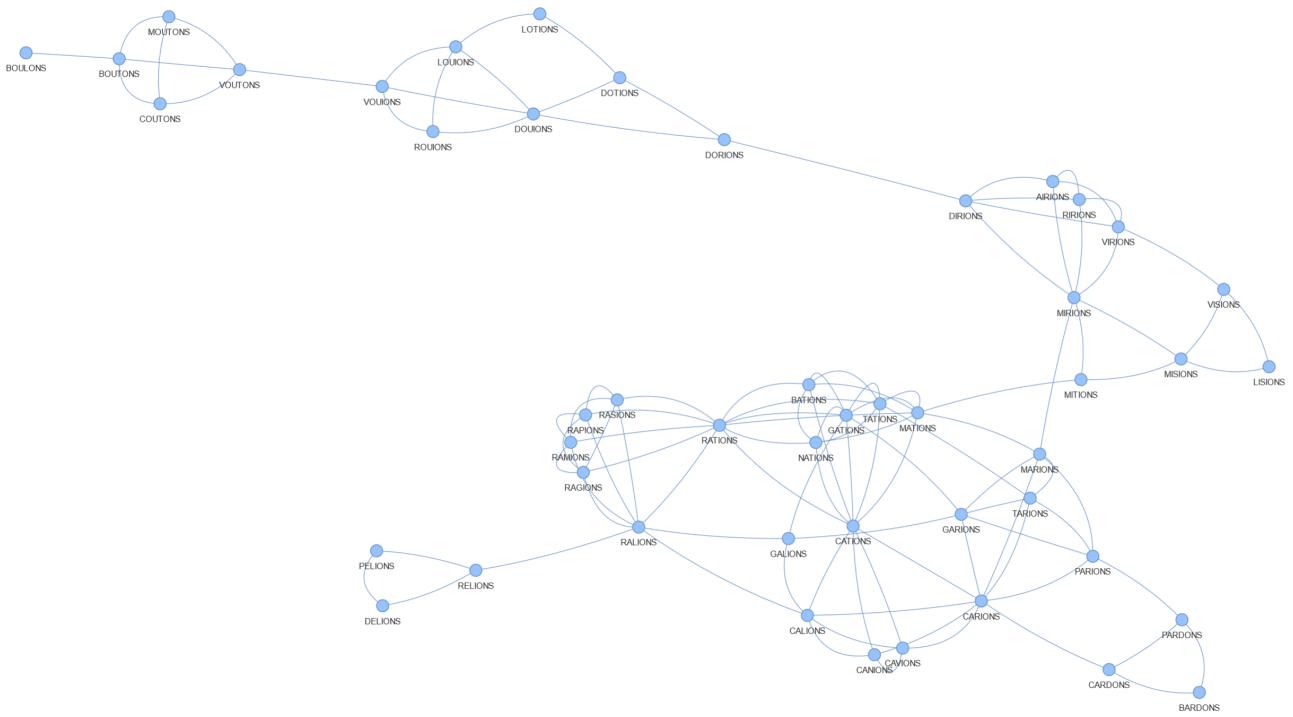
Notions vues en cours :

- 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`)
- mémoire et modifications *in-place* *vs.* *not-in-place*
- encodages des caractères, fichiers textes *vs.* binaires et leur accès en Python
- capter et lever une exception
- fonctions supplémentaires associées à des variables issues des modules `chardet` (`.detect`) et `matplotlib`

2.1 TP03 - bases - dict (1h)

```
TP_P00
|---- TEXTS
|    |---- IN
|    |    |---- ARTICLES
|    |    |    |---- EN
|    |    |    |---- ES
|    |    |    |---- FR
|    |---- BOOKS
|    |    |---- EN
|    |    |---- FR
|    |---- OTHERS
|    |---- POLITICAL
|    |    |---- DH
|    |    |---- WISHES
|    |---- RESOURCES
|    |    |---- CHARACTERS
|    |    |---- WORDS
|    |    |    |---- fr_long_dict_cleaned.txt
|---- OUT
|    |---- WG
|    |    |---- 4.wg
|    |    |---- 5.wg
|    |    |---- 6.wg
|---- config.py
|---- S01_TP01_template.py
|---- S01_TP02_template.py
|---- S02_TP03_template.py
from S01_TP02_template import *
```

`S02_TP03_template.py` : fichier *template* à compléter pour réaliser ce TP. L'objectif est d'améliorer les performances de la recherche d'échelles de mots en utilisant une structure de données plus adaptée. Plutôt que de travailler avec la liste de mots extraits du fichier, nous allons pré-calculer les relations entre les mots qui nous intéressent dans des structures de graphe ; pour ne pas effectuer plusieurs fois ce calcul qui peut être long, les représentations résultantes seront également sauvegardées dans des fichiers textes (fichiers `n.wg` avec `n` la longueur des mots du graphe). La figure suivante illustre une petite partie du graphe des mots de 7 lettres qui connecte entre eux les mots (les noeuds du graphe) qui sont à une distance de *Haming* de 1.



Les 6 fonctions suivantes ont pour finalité de construire et sauver dans des fichiers textes les représentations de tels graphes ; puis de proposer un algorithme de recherche d'échelles de mots basé sur un parcours de graphe en largeur d'abord (*Breadth-First Search - BFS*).

1. `get_words_graph_from_dictionary(file_path: str, length: int) → dict[str: list[str]]` | Retourne un graphe de mots bidirectionnel : dictionnaire Python dont les clés sont les mots de longueur `length` du fichier `file_path` associées à la liste des mots qui sont à une distance de *Hamming* de 1.

```
>>> words_graph_5 = get_words_graph_from_dictionary(DICT_NAME, 5)
>>> print(words_graph_5['HOMME'])
['COMME', 'GOMME', 'HOMIE', 'NOMME', 'POMME', 'SOMME', 'TOMME']
```

2. `get_length_words(words_graph: dict[str: list[str]]) → int` | Retourne la longueur des mots du graphe de mots `words_graph`.

```
>>> print(get_length_words(words_graph_5))
5
```

3. `words_graph_to_file(path: str, words_graph: dict[str: list[str]])` | Sauve le graphe de mots `words_graph` dans un fichier `n.wg` du dossier `path` (`n` étant longueur des mots). Chaque ligne sera composée des mots représentant une entrée de `words_graph` et la liste des mots associés.

```
>>> words_graph_to_file(PATH_OUT + 'WG/', words_graph_5)
>>> with open(PATH_OUT + 'WG/5.wg', 'r', encoding='utf-8') as f_in:
>>>     print(f_in.readlines()[:3])
['ABACA AGACA\n', 'ABATS EBATS\n', 'ABBES ABCES ABLES AUBES\n']
```

	5.wg	222.98 Kio
1	ABACA AGACA	
2	ABATS EBATS	
3	ABBES ABCES ABLES AUBES	
4	ABCES ABGES ABLES ACCES	
5	ABETI	
6	ABIMA ABIME ANIMA	
7	ABIME ABIMA ANIME	
8	ABLES ABGES ABCES AILES ALLES	
9	ABOIE ABOIS ABOYE	
10	ABOIS ABOIE ABRIS	
11	ABOLI AIOLI	
12	ABORD	
13	ABOUT AJOUT ATOUT	
14	ABOYA ABOYE	
15	ABOYE ABOITE ABOYA	
16	ABRIS ABOIS	
17	ABUSA ABUSE AMUSA	
18	ABUSE ABUSA AMUSE	
19	ACCES ABCES ACRES ACTES	
20	ACCOT	
21	ACCRU	
22	ACERE ACORE AMERE AVERE	
23	ACHAT	
24	ACIDE AMIDE APIDE ARIDE AVIDE	
25	ACIER ANIER SCIER	
26	ACONE ACORE ATONE AXONE ICONE	
27	ACORE ACERE ACONE ADORE SCORE	
28	ACRES ACCES ACTES AERES AGRES AIRES APRES ATRES OCRES	

4. `get_words_graph_from_file(path: str, length: int) → dict[str: list[str]]` | Reconstruit et retourne le graphe de mots de longueur `length` à partir du fichier `length.wg` sauvé dans le dossier `path`.

```
>>> words_graph_5 = get_words_graph_from_file(PATH_OUT + 'WG/' , 5)
>>> print(words_graph_5['HOMME'])
[ 'COMME' , 'GOMME' , 'HOMIE' , 'NOMME' , 'POMME' , 'SOMME' , 'TOMME' ]
```

5. `insert_word(new_word: str, words_graph: dict[str: list[str]])` | Insère le nouveau mot `new_word` dans le graphe de mots `words_graph` s'il n'y est pas déjà. Une `Exception` sera levée si le nouveau mot n'a pas la longueur adéquate. Attention à bien représenter toutes les nouvelles connexions (le graphe doit rester bidirectionnel). L'exemple a été réalisé après avoir construit les 25 fichiers pour les graphes de mots de longueur 1 à 25 récupérables dans une archive sur `ecampus`.

```
>>> insert_word('ISA' , words_graph_5)
Exception: length mismatch with ISA
>>> words_graph_3 = get_words_graph_from_file(PATH_OUT + 'WG/' , 3)
>>> print(words_graph_3['ISA'])
KeyError: 'ISA'
>>> print(words_graph_3['IRA'])
[ 'ARA' , 'IRE' ]
>>> insert_word('ISA' , words_graph_3)
>>> print(words_graph_3['ISA'])
[ 'IRA' , 'OSA' , 'USA' ]
>>> print(words_graph_3['IRA'])
[ 'ARA' , 'IRE' , 'ISA' ]
```

6. `get_shortest_scale(words_graph: dict[str: list[str]], starting_word: str, target_word: str) → list[str]` | Retourne la liste des mots du plus court chemin entre les mots `starting_word` et `target_word` dans le graphe `words_graph`. Les deux mots donnés peuvent ne pas être dans le dictionnaire et doivent donc être insérer dans le graphe avant la recherche d'une solution.

Il s'agit de maintenir pendant le parcours du graphe en largeur d'abord (1) un dictionnaire Python des éléments visités associés à leur prédécesseur et initialisé à `{starting_word: None}`; (2) une liste des éléments qu'il reste à explorer initialisé à `[starting_word]`.

Selon les mots qu'il reste à explorer dans (2) :

- si (2) est vide, il n'y a pas de solutions. `None` est retourné.
- si le premier mot de (2) est `target_word` la solution est trouvée. Il suffit d'utiliser le dictionnaire (1) pour retrouver la solution en remontant de prédécesseur en prédécesseur jusqu'à `None`.
- dans les autres cas, le **premier mot** est retiré et tous ses **voisins non visités** sont ajoutés en (1) avec ce mot comme prédécesseur et également **en fin** de (2) comme nouvelles solutions à explorer.

L'exemple a été réalisé après avoir construit les 25 fichiers pour les graphes de mots de longueur 1 à 25. Vous remarquerez également la nette amélioration des performances.

```
>>> t1 = perf_counter()
>>> WORDS_GRAPHS = dict()
>>> for i in range(1, 26):
>>>     WORDS_GRAPHS[i] = get_words_graph_from_file(PATH_OUT + 'WG/' , i)
>>> t2 = perf_counter()
>>> print(t2 - t1)
0.1911443999997573
>>> print(get_shortest_scale(WORDS_GRAPHS[5], 'HOMME', 'SINGE'))
['HOMME', 'COMME', 'COMTE', 'CONTE', 'CONGE', 'SONGE', 'SINGE']
>>> t3 = perf_counter()
>>> print(t3 - t2)
0.0006738999982189853
>>> print(get_shortest_scale(WORDS_GRAPHS[5], 'AVANT', 'APRES'))
['AVANT', 'AVENT', 'AIENT', 'LIENT', 'LIENS', 'LIEES', 'LIRES', 'AIRES', 'APRES']
>>> t4 = perf_counter()
>>> print(t4 - t3)
0.0005806000008306
>>> print(get_shortest_scale(WORDS_GRAPHS[4], 'TOUT', 'RIEN'))
['TOUT', 'BOUT', 'BRUT', 'BRUN', 'BREN', 'BIEN', 'RIEN']
>>> t5 = perf_counter()
>>> print(t5 - t4)
0.0010794000008900184
>>> print(get_shortest_scale(WORDS_GRAPHS[4], 'MATH', 'PURE'))
['MATH', 'MATE', 'MARE', 'MURE', 'PURE']
>>> t6 = perf_counter()
>>> print(t6 - t5)
0.0003756000005523674
>>> print(get_shortest_scale(WORDS_GRAPHS[3], 'SUD', 'EST'))
['SUD', 'SUT', 'EUT', 'EST']
>>> t7 = perf_counter()
>>> print(t7 - t6)
0.00020149999909335747
>>> print(get_shortest_scale(WORDS_GRAPHS[3], 'ISA', 'FAB'))
['ISA', 'OSA', 'OST', 'OIT', 'FIT', 'FAT', 'FAB']
>>> t8 = perf_counter()
>>> print(t8 - t7)
0.000609300010878313
```

2.2 TP04 - bases - dict (2h)

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
|
|           +-- OTHERS
|
|           +-- POLITICAL
|               |
|               +-- DH
|                   |
|                   +-- enDH.txt
|                   |
|                   +-- frDH.txt
|                   |
|                   +-- plDH.txt
|                   |
|                   +-- ruDH.txt
|                   |
|                   +-- ruDH_source.txt
|
|               +-- WISHES
|
|           +-- RESOURCES
|               |
|               +-- CHARACTERS
|                   |
|                   +-- en_alpha.txt
|                   |
|                   +-- en_diacritics.txt
|                   |
|                   +-- fr_alpha.txt
|                   |
|                   +-- fr_diacritics.txt
|                   |
|                   +-- pl_alpha.txt
|                   |
|                   +-- pl_diacritics.txt
|
|               +-- WORDS
|
+-- config.py
+-- S01_TP01_template.py
+-- S01_TP02_template.py
+-- S02_TP03_template.py
+-- S02_TP04_template.py
    |
    +-- from config import *
    +-- import chardet
    +-- import matplotlib.pyplot as plt
```

S02_TP04_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 possible 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 `FileNotFoundException`.

```
>>> get_basic_alphabet('fr'), get_basic_alphabet('en'), get_basic_alphabet('ru')
("abcdefghijklmnopqrstuvwxyzæø", "abcdefghijklmnopqrstuvwxyzæø", "")
```

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': 'éèëë', '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')
'àâéèëëîîôöùûüÿçñ'
```

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')
'abcdefghijklmnopqrstuvwxyzàâäéèëîïööùûüçñABCDEFGHIJKLMNPQRSTUVWXYZÆÀÃÉÈÃÎÏÔÖÙÛÜÝÇÑ'
```

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`.

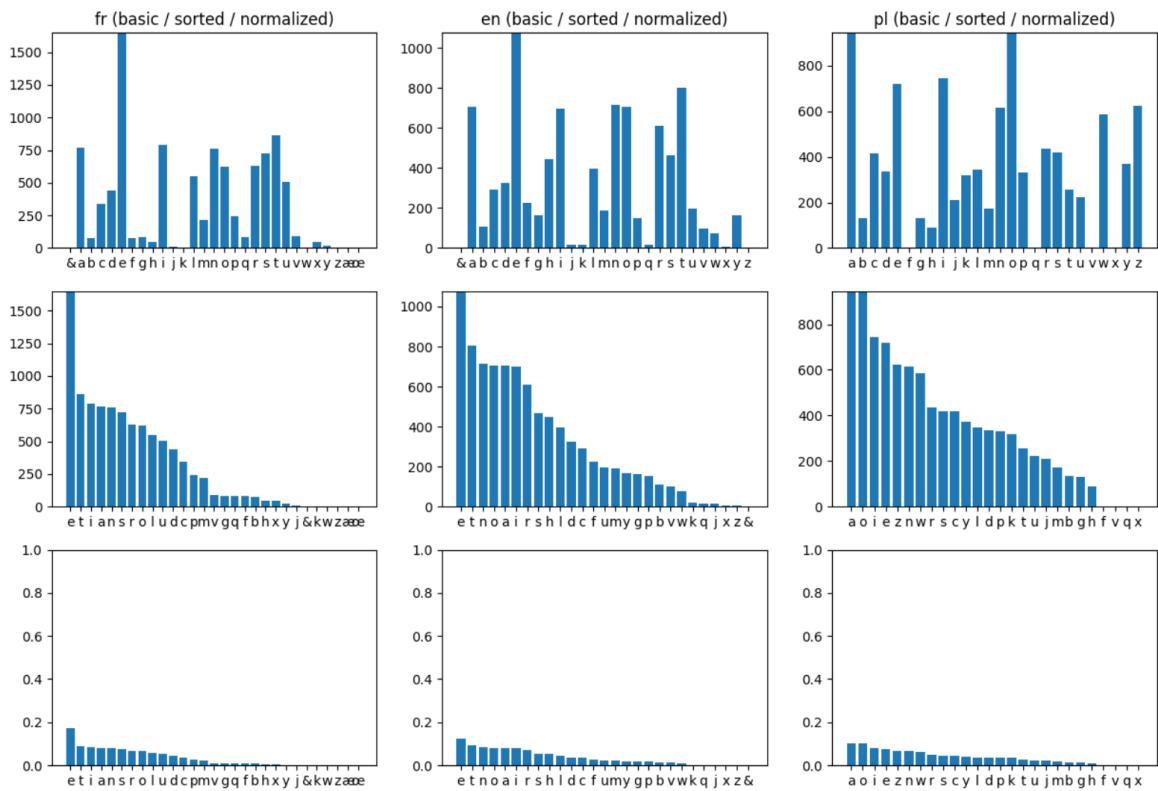
Chaque barre représentera le nombre d'occurrences (axe des *Y*) des lettres (axe des *X*). Selon la valeur booléenne de `is_sorted_by_freq` les points seront ordonnées selon les valeurs croissantes des abscisses (`False`) ou selon les valeurs décroissantes des ordonnées (`True`). Vous utiliserez les 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_normalized_histogram(ax[2][col], get_normalized_histogram(hist), True)
>>> plt.show()

```



3 CM03 (22/1/2024) - RAPPELS prog. impérative en Python 3.6>

Notions vues en cours :

- `map`, `filter` et listes/dictionnaires en compréhension
- la structure de grille

L'arborescence actuelle de vos dossiers et fichiers utiles aux TPs 5 et 6 devrait suivre l'architecture suivante :

```
TP_POO
|--- TEXTS
|   |--- IN
|   |   |--- ARTICLES
|   |   |   |--- EN
|   |   |   |--- ES
|   |   |   |--- FR
|
|   |--- BOOKS
|   |   |--- EN
|   |   |--- FR
|
|   |--- OTHERS
|
|   |--- POLITICAL
|   |   |--- DH
|   |   |--- WISHES
|
|   |--- RESOURCES
|   |   |--- CHARACTERS
|   |   |--- WORDS
|--- OUT
|   |--- WG
|
|--- config.py
|--- S01_TP01_template.py
|--- S01_TP02_template.py
|--- S02_TP03_template.py
|--- S02_TP04_template.py
|--- S03_TP05_template.py
|--- S03_TP06_template.py
    import random
```

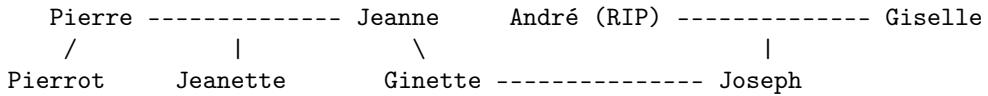
3.1 TP05 - bases - listes en compréhension (1h)

S03_TP05_template.py : fichier *template* à compléter pour réaliser ce TP. L'objectif général est de se familiariser avec la notion de liste en compréhension en Python en manipulant une structure représentant une famille 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 tests. 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 [{map} 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)
```

3.2 TP06 - bases - module grid_manager.py (2h)

S03_TP06_template.py : fichier *template* à compléter. La première ligne 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. Mettre en place à l'adresse https://forge.info.unicaen.fr/projects/poo_tps_2023 un sous-projet sur la forge de l'université. Il s'agira de synchroniser au fur et à mesure de son évolution votre dossier local `TP_00` (à renommer `TP_00_NOM_PRENOM`), avec un dépôt distant `GIT`. Vous rajouterez votre encadrant de TP et votre enseignant de CM en membre `manager` du sous-projet.
2. `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]]
```

3. `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 valeurs 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]]
```

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

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

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

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

6. `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]
```

7. `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]
```

8. `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
```

9. `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
```

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

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

11. `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]
```

12. `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)
```

13. `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
```

14. `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
```

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

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

16. `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)
```

17. `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
```

18. `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
```

19. `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
```

20. `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]
```

21. `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
```

22. `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]
```

4 CM04 (29/1/2024) - Des classes et des objets

Notions vues en cours :

- Classe/Type, Objets/Instances
- Attributs/Paramètres, Méthodes/Constructeur
- Principe d'encapsulation/Adresse `self`/Opérateur ‘.’
- Langages graphiques de modélisation/ *Unified Modeling Language - UML*
- Première expérimentation du principe d'association

L'arborescence actuelle de vos dossiers et fichiers utiles aux TPs 7 et 8 devrait suivre l'architecture suivante :

TP_POO

```
----- TEXTS
|   ----- IN
|   |   ----- ARTICLES
|   |   |   ----- EN
|   |   |   ----- ES
|   |   |   ----- FR
|
|   ----- BOOKS
|   |   ----- EN
|   |   ----- FR
|
|   ----- OTHERS
|
|   ----- POLITICAL
|   |   ----- DH
|   |   ----- WISHES
|
|   ----- RESOURCES
|   |   ----- CHARACTERS
|   |   ----- WORDS
|
|   ----- OUT
|   |   ----- WG
|
----- config.py
----- S01_TP01_template.py
----- S01_TP02_template.py
----- S02_TP03_template.py
----- S02_TP04_template.py
----- S03_TP05_template.py
----- S03_TP06_template.py
----- S04_TP07_template.py
|   import random
----- S04_TP08_template.py
import random
```

4.1 TP07 - De grid à Grid (1h)

S04_TP07_template.py : fichier *template* à compléter pour réaliser ce TP. La seule ligne d'importation sert à exploiter le module `random`.

L'objectif général de ce TP est de construire la classe `Grid` pour modéliser une grille d'entier en transposant autant que possible le travail déjà effectué en programmation impérative au TP précédent (module de gestion de grille `grid_manager.py`). Il faudra pour cela respecter le diagramme de classe et les exemples associés ci-après.

Grid

```
+ grid: list[list[int]]
+ lines_count: int
+ columns_count: int

__init__(grid_init: list[list[int]]): void
fill_random(from_values: list[int]): void
get_line(line_number: int): list[int]
get_column(column_number: int): list[int]
get_diagonal(): list[int]
get_anti_diagonal(): list[int]
get_line_str(line_number: int, separator: str='\\t'): str
get_grid_str(separator: str='\\t'): str
has_equal_values(value: int): bool
is_square(): bool
get_count(value: int): 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): int
set_cell(cell_number: int): void
get_same_value_cell_number(value: int): list[int]
get_neighbour(cell_number: int, delta: tuple(int, int), is_tore: bool): int|None
get_neighborhood(cell_number: int, deltas: list[tuple(int, int)], is_tore: bool): list[int] //OUT: sorted list
```

Le main du *template* est composé des mêmes constantes de test que pour `grid_manager.py`, reprises ci-dessous et augmentées de 2 objets de type `Grid` (`GRID_TEST` et `GRID_TEST2`) et d'une modification pour la gestion du voisinage.

```
# Constantes de test
>>> random.seed(1000) # Permet de générer toujours le 'même' hasard pour les tests
>>> NORTH, EAST, SOUTH, WEST = (-1, 0), (0, 1), (1, 0), (0, -1)
>>> NORTH_EAST, SOUTH_EAST, SOUTH_WEST, NORTH_WEST = (-1, 1), (1, 1), (1, -1), (-1, -1)
>>> CARDINAL_POINTS = (NORTH, EAST, SOUTH, WEST)
>>> WIND_ROSE = (NORTH, NORTH_EAST, EAST, SOUTH_EAST, SOUTH, SOUTH_WEST, WEST, NORTH_WEST)
>>> LINES_COUNT_TEST, COLUMNS_COUNT_TEST = 5, 7
>>> LINE_NUMBER_TEST, COLUMN_NUMBER_TEST = 1, 6
>>> CELL_NUMBER_TEST = 13
>>> VALUÉ_TEST = 0
>>> VALUES_TEST = list(range(2))
>>> IS_TORE_TEST = True
>>> DIRECTION_TEST = EAST
>>> GRID_INIT_TEST = [[VALUE_TEST] * COLUMNS_COUNT_TEST for _ in range(LINES_COUNT_TEST)]
>>> CELL_SIZE_TEST = 100
>>> MARGIN_TEST = 20
>>> SHOW_VALUES_TEST = True
>>> GRID_TEST = Grid(GRID_INIT_TEST)
>>> GRID_TEST2 = Grid([[1 for _ in range(10)] for _ in range(10)])
```

Le `main` propose sous forme d'assertions dans une syntaxe objet les mêmes tests que pour `grid_manager.py`.

```
>>> GRID_TEST.fill_random(VALUES_TEST)
>>> print(GRID_TEST.get_line(LINE_NUMBER_TEST))
[1, 0, 0, 0, 1, 1, 0]
>>> print(GRID_TEST.get_column(COLUMN_NUMBER_TEST))
[0, 0, 0, 0, 1]
>>> print(GRID_TEST.get_diagonal())
[1, 0, 1, 0, 0]
>>> print(GRID_TEST.get_anti_diagonal())
[0, 1, 0, 0, 0]
>>> print(GRID_TEST.get_line_str(2))
1 0 1 0 0 1 0 0 1 0
>>> print(GRID_TEST.get_grid_str(''))
1011010
1000110
1010010
1100100
0101001
>>> print(GRID_TEST.has_equal_values(GRID_INIT_TEST[0][0]))
False
>>> print(GRID_TEST2.has_equal_values(1))
True
>>> print(GRID_TEST.is_square())
False
>>> print(GRID_TEST2.is_square())
True
>>> print(GRID_TEST.get_count(1) == GRID_TEST.get_sum() == 16)
True
>>> print(GRID_TEST.get_coordinates_from_cell_number(13))
(1, 6)
>>> print(GRID_TEST.get_cell_number_from_coordinates(LINE_NUMBER_TEST, COLUMN_NUMBER_TEST))
13
>>> print(GRID_TEST.get_cell(9))
0
>>> GRID_TEST.set_cell(9, 1)
>>> print(GRID_TEST.get_cell(9))
1
>>> print(GRID_TEST.get_same_value_cell_numbers(1))
[0, 2, 3, 5, 7, 9, 11, 12, 14, 16, 19, 21, 22, 25, 29, 31, 34]
>>> print(GRID_TEST.get_neighbour(LINE_NUMBER_TEST, COLUMN_NUMBER_TEST, DIRECTION_TEST,
    IS_TORE_TEST))
1
>>> print(GRID_TEST.get_neighbour(LINE_NUMBER_TEST, COLUMN_NUMBER_TEST, DIRECTION_TEST,
    not IS_TORE_TEST))
False
>>> print(GRID_TEST.get_neighborhood(CELL_NUMBER_TEST, WIND_ROSE,
    IS_TORE_TEST))
[0, 5, 6, 7, 12, 14, 19, 20]
>>> print(GRID_TEST.get_neighborhood(CELL_NUMBER_TEST, WIND_ROSE,
    not IS_TORE_TEST))
[5, 6, 12, 19, 20]
```

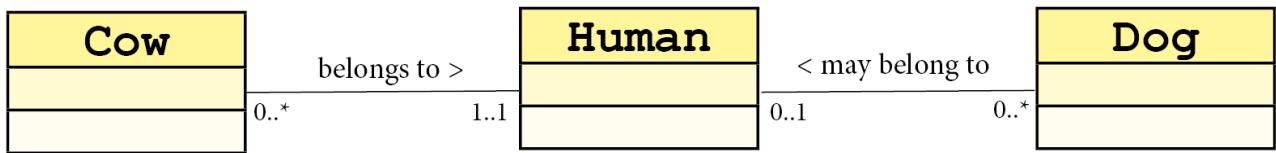
Remarque : Une fois toutes les assertions passées, n'hésitez-pas à expérimenter en créant de nouveaux objets de type/classe `Grid`, en appliquant vos propre tests, voire en définissant des méthodes supplémentaires.

Attention : Cette classe sera la classe de base du projet fil rouge à développer dans certains des TPs suivants.

4.2 TP08 - Des animaux et des hommes

S04_TP08_template.py : fichier *template* à compléter pour réaliser ce TP. La seule ligne d'importation sert à exploiter le module `random`.

L'objectif général de ce TP est de construire les 3 classes `Human`, `Cow` et `Dog` en s'appuyant sur une modélisation possible des concepts d'humain, de vache et de chien. Il faudra pour cela respecter chaque diagramme de classe proposé ainsi que les remarques et les exemples qui y sont associés. De plus les liens d'association entre les classes modélisées par la figure suivante seront intégrés comme un attribut spécifique des classes `Cow` et `Dog`.



Remarque : il est souvent 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 ce même module Python.

4.2.1 La classe Human

Réalisez la classe `Human` de manière à ce qu'elle corresponde précisément au diagramme de classe, aux précisions apportées et aux tests associés.

Human
+ <code>full_name</code> : str + <code>nationality</code> : str //length: 2 (alpha-code2 characters) + <code>greetings</code> : str + <code>age</code> : int=0 + <code>majority</code> : int
<code>__init__(first_names: list[str], last_name: str, alpha_code2: str, greetings: str, majority: int=18): void</code> <code>is_adult(): bool</code> <code>get_info(): str</code> <code>ageing(years: int=1): void</code> <code>get_shout(): str</code>

1. Initialisation des 5 attributs du constructeur :
 - `full_name` : concaténation des prénoms du paramètre `first_names` entre eux puis au nom de famille du paramètre `last_name`
 - `nationality` : code de la nationalité sur deux lettres du paramètre `alpha_code2`

- `greetings` : valeur du paramètre `greetings`
 - `age` à 0
 - `majority` : valeur du paramètre `majority`
2. `is_adult` est à `True` si la majorité est atteinte. `False` sinon.
 3. `get_info` décline les informations d'identité
 4. `ageing` fait vieillir du nombre d'années `years`
 5. la valeur de `get_shout` s'améliore avec l'âge :
 - jusqu'à 1 an : "Ouin ouin"
 - jusqu'à 2 an : "Areuh baba gaga"
 - jusqu'à 3 an : les salutations mais avec toutes les lettres mélangées
 - à partir de 3 ans : les salutations normales

```
>>> random.seed(100)
>>> farmer = Human(["Marcel", "Robert"], "Duchamps", "fr", "Bonjour", 18)
>>> farmer.ageing(35)
>>> print(farmer.get_info())
'Identité : Marcel Robert Duchamps – Nationalité : FR – Age : 35 ans (majeur)'
>>> print(farmer.get_shout())
'Bonjour'
>>> farmeress = Human(["Marcela"], "Zpola", "pl", "Dzien dobry", 18)
>>> farmeress.ageing(36)
>>> print(farmeress.get_info())
'Identité : Marcela Zpola – Nationalité : PL – Age : 36 ans (majeur)'
print(farmeress.get_shout())
'Dzien dobry'
>>> boy = Human(["Marcel", "junior"], "Duchamps Zpola", "fr", "Bonjour")
>>> print(boy.get_info())
'Identité : Marcel junior Duchamps Zpola – Nationalité : FR – Age : 0 ans (mineur)'
>>> print(boy.get_shout())
'Ouin ouin'
>>> boy.ageing()
print(boy.get_info())
'Identité : Marcel junior Duchamps Zpola – Nationalité : FR – Age : 1 ans (mineur)'
>>> print(boy.get_shout())
'Areuh baba gaga'
>>> boy.ageing()
>>> print(boy.get_info())
'Identité : Marcel junior Duchamps Zpola – Nationalité : FR – Age : 2 ans (mineur)'
>>> print(boy.get_shout())
'Bonrujo'
>>> boy.ageing()
>>> print(boy.get_info())
'Identité : Marcel junior Duchamps Zpola – Nationalité : FR – Age : 3 ans (mineur)'
>>> print(boy.get_shout())
'Bonjour'
```

4.2.2 Les classes Cow et Dog

Réalisez les classes `Cow` et `Dog` de manière à ce qu'elles correspondent aux diagrammes de classe, aux précisions apportées et aux tests associés.

1. Les initialisations des attributs de `Cow` sont en lien direct avec les paramètres de même nom

Cow	Dog
<pre>+ nickname: str + weight: float + owner: Human __init__(nickname: str, weight: float, owner: Human): void get_info(): str gain_weight(weight: int=1): void lose_weight(weight: int=1): void take_owner(owner: Human): void get_shout(): str</pre>	<pre>+ nickname: str + owner: Human + state: int //Authorized values are only 0 and 1 __init__(nickname: str, owner: Human=None, state: int=0): void get_info(): str swap_state(): void take_owner(owner: Human): void get_shout(): str</pre>

2. `get_info` décline les informations d'identité
3. `gain_weight` et `lose_weight` font grossir et maigrir d'un poids `weight`
4. `take_owner` change le propriétaire par `owner`
5. Une Cow fait "Meuh"

```
>>> milk_cow = Cow("Aglaë", 300, farmer)
>>> milk_cow.gain_weight(30)
>>> milk_cow.lose_weight(20)
>>> print(milk_cow.get_info())
'Aglaë : cow de 310 Kg. Appartient à Marcel Robert Duchamps.'
>>> milk_cow.take_owner(farmeress)
>>> print(milk_cow.get_info())
'Aglaë : cow de 310 Kg. Appartient à Marcela Zpola.'
>>> print(milk_cow.get_shout())
'Meuh'
```

Plutôt qu'un poids, un Dog gérera un état `state` qui pourra avoir la valeur 0 (signifiant "cool") ou 1 (signifiant "en colère"). La méthode `swap_state` inverse l'état courant et selon le cas le cri produit par `get_shout` sera "Ouah ouah" ou "Grrr".

```
>>> stray_dog = Dog("Médor")
>>> print(stray_dog.get_info())
"Médor : dog cool. N'a pas de propriétaire."
>>> print(stray_dog.get_shout())
'Ouah ouah'
>>> stray_dog.take_owner(boy)
>>> stray_dog.swap_state()
>>> print(stray_dog.get_info())
"Médor : dog en colère. Appartient à Marcel junior Duchamps Zpola."
>>> print(stray_dog.get_shout())
'Grrr'
```

5 CM05 (5/2/2024) - Protection, contrôle d'accès, agrégations

Notions vues en cours :

- Attributs privés/publics
- Attributs d'instance/de classe
- Méthodes d'instance/de classe/statiques
- Accesseurs en lecture/écriture
- Relations d'associations/d'agrégation/de composition

L'arborescence actuelle de vos dossiers et fichiers utiles aux TPs 9 et 10 devrait suivre l'architecture suivante :

```
TP_POO
|---- TEXTS
|    |---- IN
|    |    |---- ARTICLES
|    |    |    |---- EN
|    |    |    |---- ES
|    |    |    |---- FR
|
|    |---- BOOKS
|    |    |---- EN
|    |    |---- FR
|
|    |---- OTHERS
|
|    |---- POLITICAL
|    |    |---- DH
|    |    |---- WISHES
|
|    |---- RESOURCES
|    |    |---- CHARACTERS
|    |    |---- WORDS
|
|---- OUT
|    |---- WG
|
|---- config.py
|---- S01_TP01_template.py
|---- S01_TP02_template.py
|---- S02_TP03_template.py
|---- S02_TP04_template.py
|---- S03_TP05_template.py
|---- S03_TP06_template.py
|---- S04_TP07_template.py
|---- S04_TP08_template.py
|---- S05_TP09_01_template.py
|        import random
|---- S05_TP09_02_template.py
|        import random
|---- S05_TP10_template.py
|        import turtle
```

5.1 TP09 - Reprises et privatisations (1h)

Une fois les TP07 et TP08 terminés, copiez et renommez les fichiers `S04_TP07_template.py` (classe Grid) et `S04_TP08_template.py` (classes Human, Cow et Dog) respectivement en `S05_TP09_01_template.py` et `S05_TP09_02_template.py`.

L'objectif de ce TP est de modifier dans les deux fichiers les classes et les tests en vous aidant des nouveaux diagrammes de classe proposés. Il s'agira de respecter les étapes suivantes :

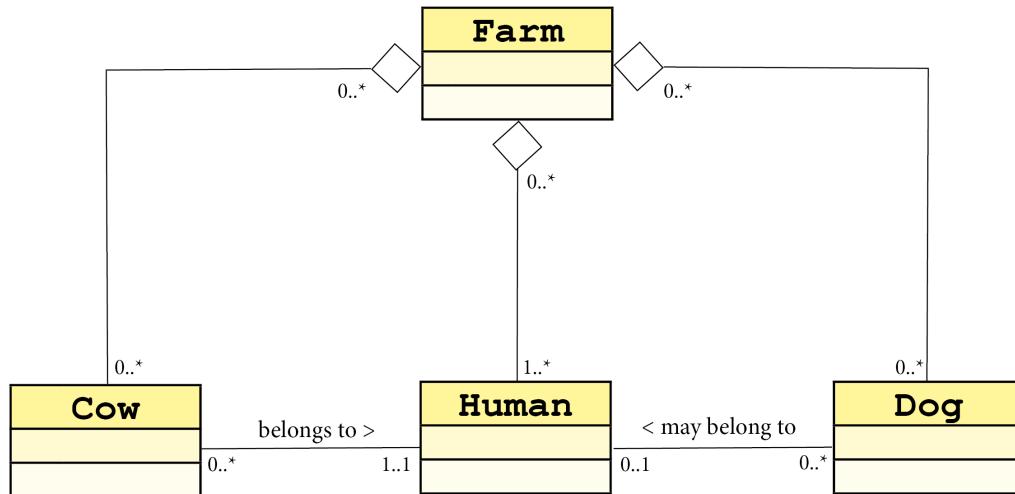
1. priver **tous** les attributs.
2. ajouter éventuellement les accesseurs en lecture et/ou en écriture (selon les contraintes en rouge)
3. tester et corriger les quelques méthodes et tests impactés
4. ajouter les 2 attributs et les 2 méthodes de classe demandés dans **Human**. Trouver la méthode à rendre statique dans **Cow**.
5. rajouter la classe **Farm**
6. ajouter vos propres tests

5.2 Modifications de la classe Grid

Grid
<pre>- grid: list[list[int]] - lines_count: int - columns_count: int __init__(grid_init: list[list[int]]): void get_grid(): list[list[int]] get_lines_count(): int get_columns_count(): int fill_random(values: list[int]): void get_line(line_number: int): list[int] get_column(column_number: int): list[int] get_diagonal(): list[int] get_anti_diagonal(): list[int] get_line_str(line_number: int, separator: str='\t'): str get_grid_str(separator: str='\t'): str has_equal_values(value: int): bool is_square(): bool get_count(value: int): 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): int set_cell(cell_number: int): void get_same_value_cell_number(value: int): list[int] get_neighbour(line_number: int, column_number: int, delta: tuple(int, int), is_tore: bool): int None get_neighborhood(line_number: int, column_number: int, deltas: list[tuple(int, int)], is_tore: bool): list[int None]</pre>

5.3 Diagramme des classes Human, Cow et Dog et Farm

Une ferme agrège au moins 1 humain et potentiellement des vaches et des chiens. Tous peuvent peupler aucune, une ou plusieurs fermes.



Human
<pre> - <u>humans_count</u>: int=0 //increased when an Human is built. - <u>nationalities_greetings</u>: dict{str: str} = {'fr': 'Bonjour', 'en': 'Hello', 'pl': 'Dzień dobry', 'default': '...'} - <u>full_name</u>: str //never changed. - <u>nationality</u>: str //length: 2 (alpha-code2 characters). Never changed. - <u>greetings</u>: str //depending on alpha-code2 characters key in nationalities_greetings dict. Never changed. - <u>age</u>: str=0 //never decreased. - <u>majority</u>: str //never changed. <u>get_humans_count</u>(): int <u>add_nationality_greetings</u>(nationality: str, greetings: str): void <u>__init__</u>(first_names: list[str], last_name: str, alpha_code2: str, majority: int=18): void <u>get_full_name</u>(): str <u>get_nationality</u>(): str <u>get_greetings</u>(): str <u>get_age</u>(): int <u>get_majority</u>(): int <u>is_adult</u>(): bool <u>get_info</u>(): str <u>ageing</u>(years: int=1): void <u>get_shout</u>(): str </pre>

Cow	Dog
<pre>- nickname: str - weight: float - owner: Human __init__(nickname: str, weight: float, owner: Human): void get_nickname(): str set_nickname(nickname: str): void get_weight(): float get_owner(): Human get_info(): str gain_weight(weight: int=1): void lose_weight(weight: int=1): void take_owner(owner: Human): void <u>get_shout()</u>: str</pre>	<pre>- nickname: str - owner: Human - state: int //Authorized values are only 0 and 1 __init__(nickname: str, owner: Human=None, state: int=0): void get_nickname(): str set_nickname(nickname: str): void get_owner(): Human get_state(): int get_info(): str swap_state(): void take_owner(owner: Human): void get_shout(): str</pre>

Farm
<pre>- name: str //never changed - inhabitants: set{Human Cow Dog} __init__(name: str, owner: Human): void get_name(): str get_inhabitants(): set{Human Cow Dog} populate(inhabitant: Human Cow Dog): void <u>get_talk()</u>: str</pre>

```
>>> farm = Farm("Fermarcel", farmer)
>>> farm.populate(farmer)
>>> farm.populate(farmeress)
>>> farm.populate(boy)
>>> farm.populate(milk_cow)
>>> farm.populate(stray_dog)
>>> print(farm.get_talk())
Les 5 habitants de la ferme Fermarcel se retrouvent :
- Marcela Zpola : Dzien dobry
- Aglaë : Meuh
- Marcel Robert Duchamps : Bonjour
- Marcel junior Duchamps Zpola : Bonjour
- Mé dor : Grrr
```

5.4 TP10 - Exercices supplémentaires (2h)

S05_TP10_template.py : fichier *template* à créer vous même pour réaliser les classes de ce TP.

1. Codez les 2 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	Point
<ul style="list-style-type: none"> - <code>lower</code>: float - <code>upper</code>: float //superior or equal to lower <code>_init_(value1: float, value2: float): void</code> <code>get_lower(): float</code> <code>get_upper(): float</code> <code>to_str(): str</code> <code>get_size(): float</code> <code>get_middle(): float</code> <code>get_union(other: Range): Range</code> <code>has_intersection(other: Range): bool</code>	<ul style="list-style-type: none"> - <code>x</code>: float - <code>y</code>: float <code>_init_(x: float, y: float): void</code> <code>get_x(): float</code> <code>get_y(): float</code> <code>to_str(): str</code> <code>translation(dx: float, dy: float): void</code> <code>get_distance(other: Point): float</code>

Attention : Respectez précisément le typage demandé

Remarque : Vous pourrez utiliser au mieux les fonctions *built-in* `min` et `max`.

```
>>> range_test1, range_test2 = Range(18.2, 5), Range(10, 20)
>>> print(range_test1.to_str())
[5,18.2]
>>> print(range_test2.to_str())
[10,20]
>>> print(range_test1.get_size(), range_test2.get_size())
13.2 10
>>> print(range_test1.get_middle())
11.6
>>> print(range_test1.get_union(range_test2).to_sttr())
[5,20]
>>> print(range_test1.has_intersection(range_test2))
True
```

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

2. Réalisez de la même manière la classe Segment

Segment
<pre> - point1: Point - point2: Point __init__(p1: Point, p2: Point): void get_point1(): Point get_point2(): Point to_str(): str translation(dx: float, dy: float): void get_length(): float get_projection_x(): Range get_projection_y(): Range get_middle(): Point </pre>

Remarque :

- translation d'un Segment doit utiliser translation d'un Point
- get_length d'un Segment doit utiliser get_distance d'un Point
- get_middle d'un Segment doit utiliser get_projection_x et get_projection_y

Remarque : get_projetction_x (resp. get_projetction_y) produit un Range correspondant à la projection du Segment sur l'axe des abscisses (resp. ordonnées).

```

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

```

3. 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.

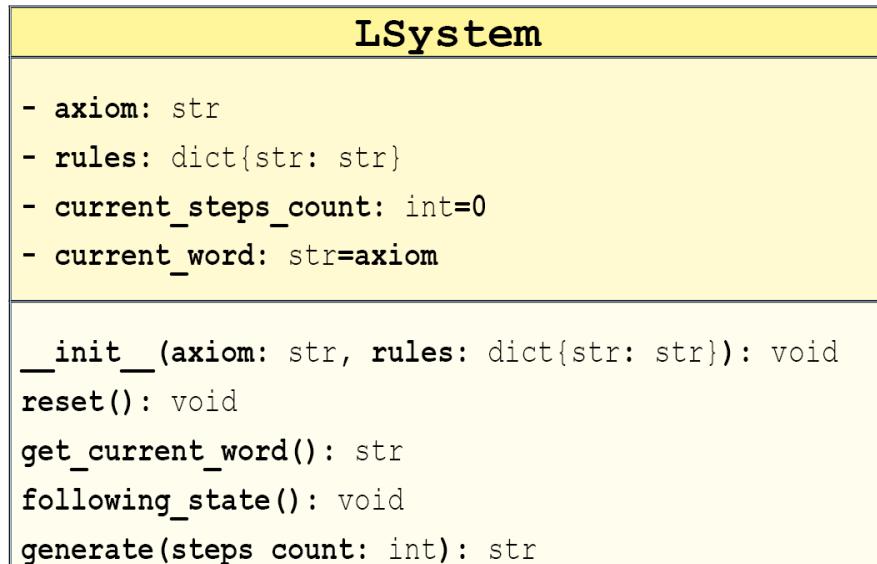
Exemple

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---
```

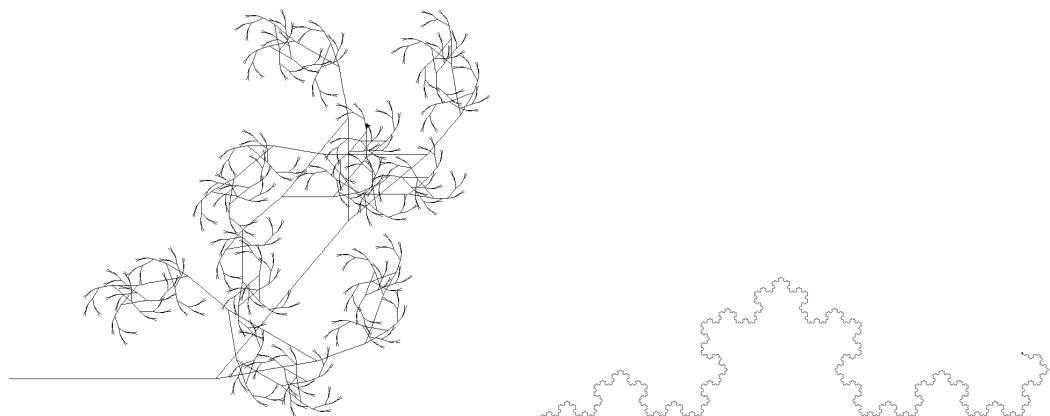
4. Nous souhaitons faire une classe **TurtleMemory** avec un attribut principal de type **Turtle** dont les mouvements seront commandés par des chaînes de caractères. Chaque caractère sera associé à une action de la tortue :

- '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/pile **stack**.
- ']' : la tortue se repositionne (sans dessiner) sur la dernière position mémorisée dans la pile **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 de la méthode **draw_l_system** sont donnés par les exemples et images qui suivent pour dessiner un **l_system** en fonction d'un point de départ **p**, d'une longueur **length** associée à 'f', d'un **angle** associé à '+' et '-' et d'un nombre d'étapes **steps_count**.

TurtleMemory
<pre>- turtle: Turtle=Turtle() - stack: list[Point]=[]</pre>
<pre>__init__(): void draw_l_system(l_system: LSystem, steps_count: int, starting_point: Point, length: int, angle: float): void</pre>

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



6 CM06 (12/2/2024) - Héritage

Notions vues en cours :

- agrégation forte (X est composé de Y)
- Héritage simple (X est une sorte de Y)
- Classe mère/fille
- Surcharges/redéfinitions
- Spécialisation d'attributs/de méthodes

L'arborescence actuelle de vos dossiers et fichiers utiles aux TPs 11 et 12 devrait suivre l'architecture suivante :

```
TP_POO
|---- TEXTS
|    |---- IN
|    |    |---- ARTICLES
|    |    |---- BOOKS
|    |    |---- OTHERS
|    |    |---- POLITICAL
|    |    |---- RESOURCES
|    |
|    |---- OUT
|
|---- config.py
|---- S01_TP01_template.py
|---- S01_TP02_template.py
|---- S02_TP03_template.py
|---- S02_TP04_template.py
|---- S03_TP05_template.py
|---- S03_TP06_template.py
|---- S04_TP07_template.py
|---- S04_TP08_template.py
|---- S05_TP09_01_template.py
|---- S05_TP09_02_template.py
|---- S05_TP10_template.py
|---- S06_TP11.py
|    |    import random
|---- S06_TP12.py
|    |    import random
```

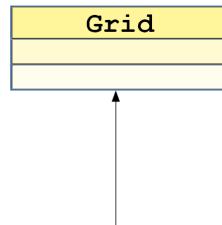
6.1 TP11 - héritage simple

Une solution du TP `S05_TP09_01_template.py` vous est proposée sur *ecampus*. Vous noterez l'ajout de deux méthodes proches de celles utiles pour la récupération d'un voisin ou du voisinage d'une cellule. Les méthodes `get_cell_neighbour_number` et `get_cell_neighborhood_numbers` sont différentes à deux niveaux :

- un paramètre d'entrée est le numéro de la cellule plutôt que les numéros de ligne et de colonne.
- la sortie fournit la liste des numéros de cellule triés par ordre croissant, plutôt que leur contenu (car il est toujours possible de retrouver le contenu à partir du numéro mais pas forcément le contraire!)

L'objectif de ce TP est d'implémenter la classe `PlanetAlpha` pour respecter le diagramme suivant. Vous construirez vous-même un fichier `S06_TP11.py` ne contenant que la classe `PlanetAlpha`.

Remarque : 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.



PlanetAlpha
+ <u>NORTH</u> , <u>EAST</u> , <u>SOUTH</u> , <u>WEST</u> , <u>NORTH EAST</u> , <u>SOUTH EAST</u> , <u>SOUTH WEST</u> , <u>NORTH WEST</u> : tuple[int, int] //type defined as <<t2>> + <u>CARDINAL_POINTS</u> : [4]<<t2>>=(NORTH, EAST, SOUTH, WEST) + <u>WIND ROSE</u> : [8]<<t2>>=(NORTH, NORTH_EAST, EAST, SOUTH_EAST, SOUTH, SOUTH_WEST, WEST, NORTH_WEST) - <u>name</u> : str - <u>ground</u> : char __init__(name: str, latitude_cells_count: int, longitude_cells_count: int, ground: char): void get_name(): str get_ground(): char get_random_free_place(): int //a random free cell number or -1 if no free place born(cell_number: int, element: char): int //1 if success. Else 0. die(cell_number: int): int //1 if success. Else 0. repr(): str

La classe `PlanetAlpha` est une spécialisation de `Grid` qui est construite à partir du nombre de latitudes (lignes) et du nombre de longitudes (colonnes). Chaque cellule représente une place sur la planète qui peut être soit vide, soit occupée. Chaque case contient donc un caractère spécifique. Celui par défaut qui correspond au sol (cellule libre) est fourni à la création de la planète et n'est pas modifiable.

2 accesseurs en lecture des 2 attributs d'instance permettent de récupérer le nom de la planète ainsi que le caractère utilisé pour représenter le sol.

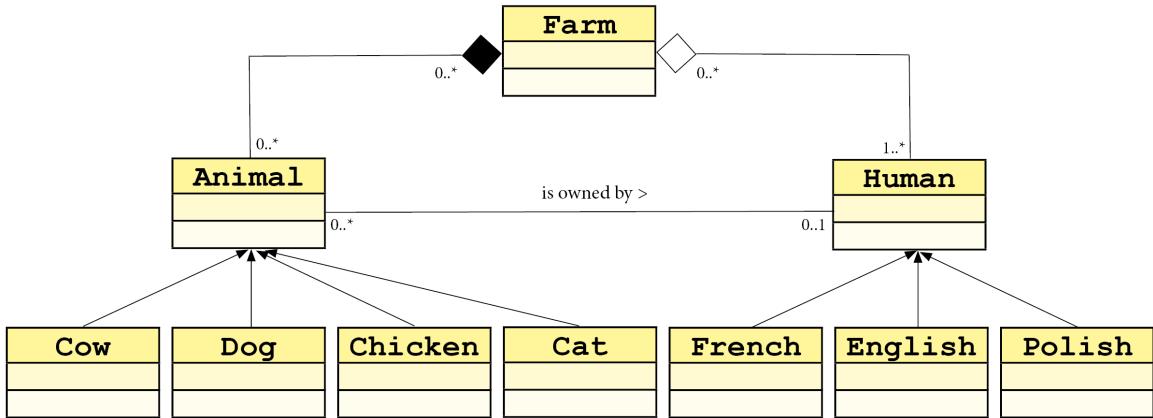
La méthode `get_random_place` permet de trouver une place non occupée, si elle existe, sur la planète. Si il n'y a pas de place disponible, `-1` est retourné.

2 accesseurs en écriture sont également à définir pour ajouter (sur une cellule libre donnée), ou retirer (d'une cellule occupée donnée) un élément. Dans les deux cas si l'opération a pu se faire alors 1 est retourné. 0 Sinon.

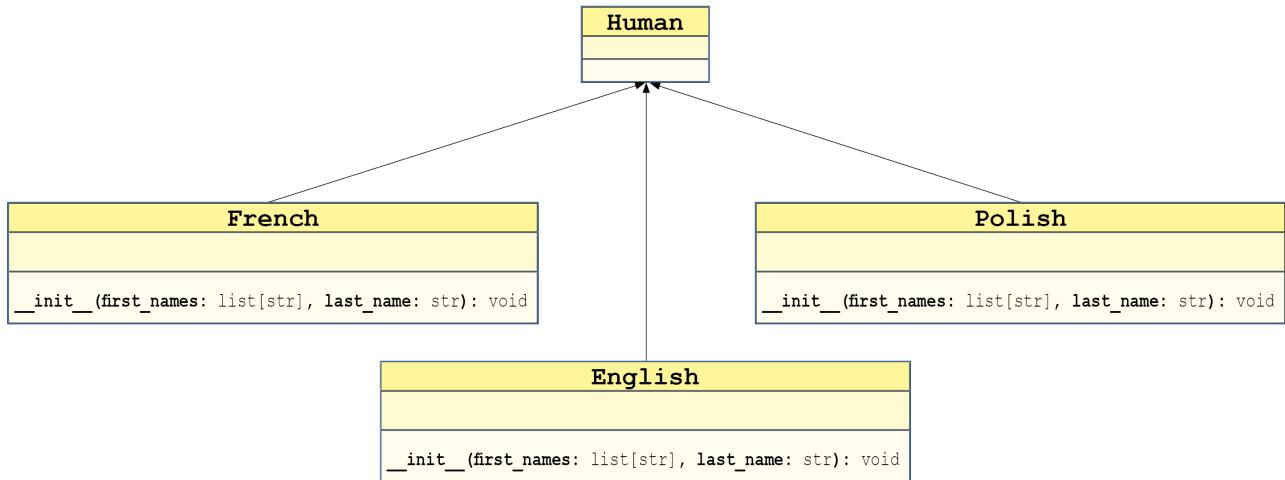
Enfin une méthode spéciale sera proposée pour l'affichage de la planète comme proposé dans la série de tests ci-dessous.

6.2 TP12 - Agrégations et héritage par généralisation ou spécialisation

Faites une copie du fichier `S05_TP09_02_template.py` que vous renomerez `S06_TP12.py`. L'objectif de ce TP est d'ajouter et/ou modifier les classes nécessaires pour respecter le diagramme suivant.



1. Les classes French, English et Polish spécialisent Human en fixant les paramètres de nationalité et de majorité.

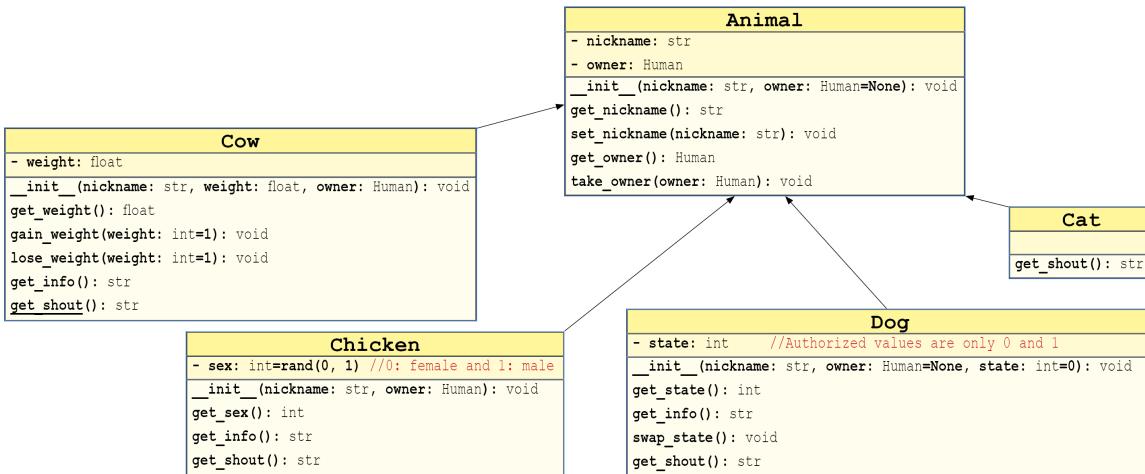


```

>>> 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. La classe Animal généralise Cow et Dog en regroupant les éléments communs. Ces deux dernières classes devront être modifiées **au minimum** pour être construites par héritage de Animal. Les nouvelles classes Chicken et Cat peuvent alors être construites par spécialisation de Animal.



```

>>> 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.

```

3. La classe **Chicken** spécialise **Animal** en ajoutant la gestion d'un attribut pour le sexe (initialisé aléatoirement à 0 pour une femelle et à 1 pour un mâle). Un tel animal fait en effet *Cocorico* quand c'est un mâle mais *cot cot cot* si c'est une femelle. Rajoutez également la classe **Cat** qui ne spécialise la classe **Animal** que par son cri (*Ronron* si il a un propriétaire. *Miaou* sinon).

```

>>> 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.get_talk())
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 !
- grrrr !
- Meuuuuuuuuuuuuuh !
- cot cot cot codec !
- cocorico !

```

4. Contrôler une agrégation forte (ou composition) en Python n'est pas aisément possible en raison de la nature même du langage. Comment rendre au moins les tests des exemples proposés cohérents de ce point de vue avec le diagramme de classe ?

Remarque : Il s'agit de respecter la relation de composition demandée entre les classes **Farm**, **Animal** et **Human**. Cette question est liée au cycle de vie des objets : théoriquement, si un objet de la classe **Farm** est détruit alors les animaux qui le composent devraient l'être également mais pas les humains.

7 CM07 (4/3/2024) - Spécialisations classes *built-in*, méthodes spéciales, POO et tkinter

Notions vues en cours :

- Bases du module tkinter : Tk, Button, Frame, Canvas, Label, Listbox
- Spécialisation d'une fenêtre Tk
- Méthodes spéciales `__repr__`, `__str__`, `__eq__`

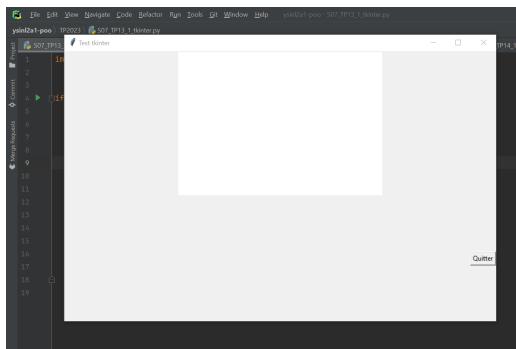
L'arborescence actuelle de vos dossiers et fichiers utiles aux TPs 13 et 14 devrait suivre l'architecture suivante :

```
TP_POO
|   |
|   ----- TEXTS
|   |   |
|   |   ----- IN
|   |   |   |
|   |   |   ----- ARTICLES
|   |   |   |
|   |   |   ----- BOOKS
|   |   |   |
|   |   |   ----- OTHERS
|   |   |   |
|   |   |   ----- POLITICAL
|   |   |   |
|   |   |   ----- RESOURCES
|   |   |
|   |   ----- OUT
|
|   ----- config.py
|   ----- S01_TP01_template.py
|   ----- S01_TP02_template.py
|   ----- S02_TP03_template.py
|   ----- S02_TP04_template.py
|   ----- S03_TP05_template.py
|   ----- S03_TP06_template.py
|   ----- S04_TP07_template.py
|   ----- S04_TP08_template.py
|   ----- S05_TP09_01_template.py
|   ----- S05_TP09_02_template.py
|   ----- S05_TP10_template.py
|   ----- S06_TP11.py
|   ----- S06_TP12.py
|   ----- S07_TP13_01.py
|       import tkinter as tk
|   ----- S07_TP13_02.py
|       import tkinter as tk
|   ----- S07_TP13_03.py
|       import tkinter as tk
|   ----- S07_TP14_01.py
|       import random
|   ----- S07_TP14_02.py
|       import random
|   ----- S07_TP14_03.py
|       import tkinter as tk
|       from S06_TP11 import PlanetAlpha
|       from S07_TP14_1 import Ground, Water, Herb, Cow, Lion
|   ----- S07_TP14_04.py
|       import random
|       from S06_TP11 import PlanetAlpha
|       from S07_TP14_1 import Ground, Water, Herb, Cow, Lion, Dragon
```

7.1 TP13 - Exercices tkinter

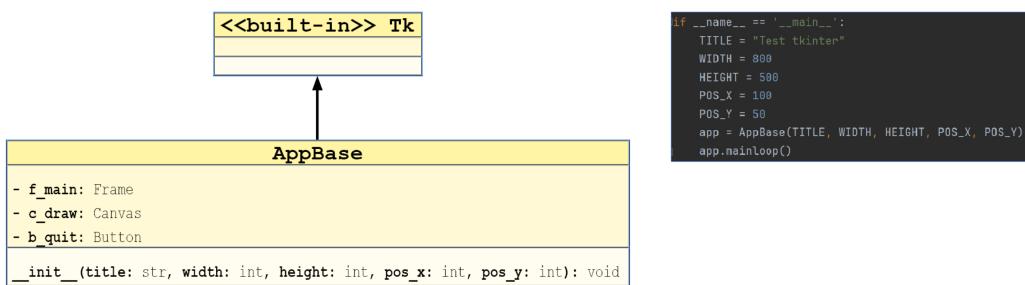
7.1.1 Création d'une fenêtre : version impérative

1. Créez le fichier **S07_TP13_01.py**. Importez-y le module classique **tkinter** en le renommant **tk**. Dans une partie `if __name__ == "__main__"`, créez les 5 constantes **TITLE**, **WIDTH**, **HEIGHT**, **POS_X**, **POS_Y** avec par exemple les valeurs respectivement « test tkinter », 800, 500, 100 et 50. Faire à la suite un programme qui crée et affiche une fenêtre dont le titre est **TITLE**, la largeur est de **WIDTH** pixels, la hauteur est de **HEIGHT** pixels et le coin haut/gauche est aux coordonnées (**POS_X** pixels, **POS_Y** pixels).
2. Placez à droite de l'interface un **Button** qui permet de quitter et fermer la fenêtre.
3. Placez dans l'interface une sous-fenêtre (**Frame**) positionnée avant le bouton (celle-ci sera vide donc invisible). Un attribut **f_main** y fera référence.
4. Placez dans la sous-fenêtre une surface de dessin (**Canvas**) blanche. Le résultat final doit correspondre à la figure ci-dessous.



7.1.2 Création d'une fenêtre : version objet

Modifiez le code précédent dans un fichier **S07_TP13_02.py** pour le réorganiser avec la classe **AppBase** qui spécialise la classe **Tk** du module **tkinter** comme dans le diagramme ci-dessous.



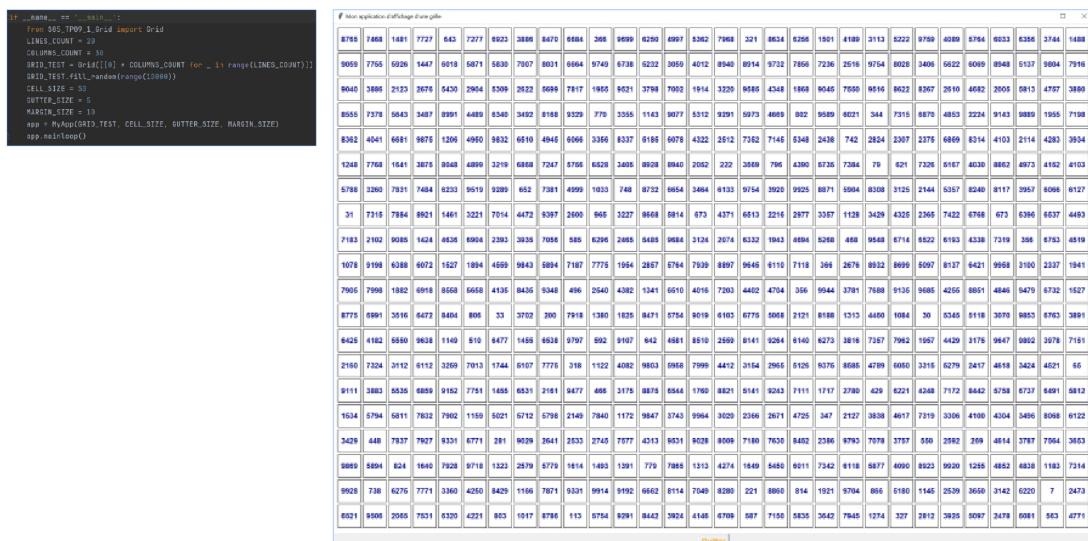
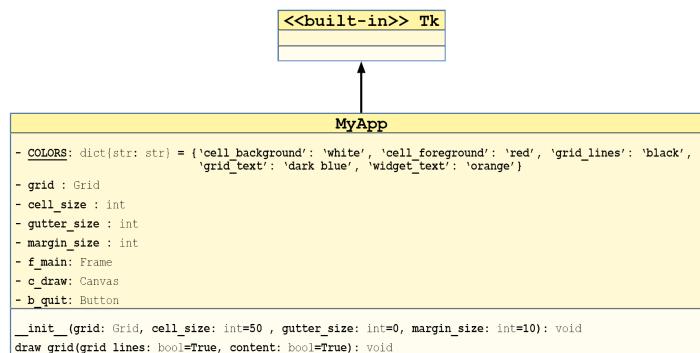
Attention : Le résultat de l'exécution du programme indiqué à côté du diagramme doit être strictement identique à la question précédente.

7.1.3 Crédit d'une fenêtre plus complexe

L'objectif est de faire dans un fichier **S07_TP13_03.py** une nouvelle classe **MyApp** spécialisant une fenêtre Tk du module **tkinter**. Une instance de cette application graphique s'inspirera de la précédente pour dessiner sur une surface de dessin la grille de type **Grid** passée en paramètre.

Le constructeur préparera le **Canvas** à la bonne taille en fonction des autres paramètres données : tailles en pixel des cases (formes carrées), des gouttières (distance entre les cases) et de la marge (distance avec la bordure de la surface de dessin). Le titre de la fenêtre sera « Mon application d'affichage d'une grille ». Le constructeur termine par l'appel de la méthode **draw_grid** qui supprime tout ce qu'il y a sur le **Canvas** et y dessine la grille.

La méthode **draw_grid** dessinera potentiellement les bordures de la grille (paramètre **grid_lines** à **True**) et son contenu (paramètre **content** à **True**). Les couleurs de l'interface (case vide, case remplie, lignes de la grille, texte des cases, textes des **widgets**) seront gérées par un dictionnaire **COLORS** en constante de classe. La taille de la police de caractère du texte centré dans les cases dépendra de la taille de la case (le quart). Le diagramme de classes et un exemple d'exécution sont donnés ci-dessous.

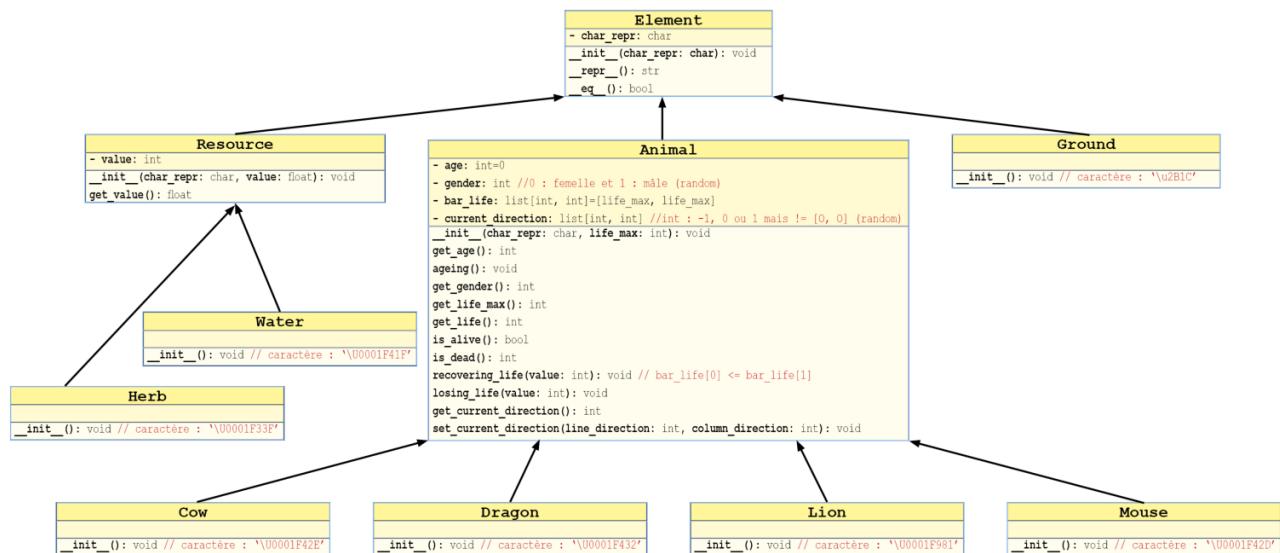


7.2 TP14 - Affichage d'une PlanetAlpha d'Element

7.2.1 La hiérarchie de classe Element

En vous inspirant des TP précédents, coder dans le fichier `S07_TP14_01.py` le diagramme de classes suivant.

Remarque : Est ajoutée à `Element` la méthode spéciale adéquate pour que la comparaison de deux objets avec le signe « `==` » s'appuie plutôt sur leur classe et leur attribut `char_repr` que sur leur identité.



Les tests ci-dessous doivent produire l'affichage indiqué par la figure qui suit.

```

>>> print(Ground(), str(Ground()))
>>> print(Ground() == str(Ground()))
>>> print(Ground() == Ground())
>>> print(Ground() is Ground())
>>> TYPES_COUNT = {Herb: 2, Water: 3, Cow: 2, Dragon: 1, Lion: 5, Mouse: 10}
>>> ELEMENTS_BY_TYPE = {element_type: [element_type() for _ in range(element_count)] for element_type, element_count in TYPES_COUNT.items()}
>>> for element_type, elements in ELEMENTS_BY_TYPE.items():
>>>     print(element_type.__name__, elements)
  
```

```

False
True
False
Herb [leaf, leaf]
Water [water droplet, water droplet, water droplet]
Cow [cow, cow]
Dragon [apple]
Lion [lion, lion, lion, lion, lion]
Mouse [mouse, mouse, mouse, mouse, mouse, mouse, mouse, mouse]
  
```

7.2.2 Application d'affichage alphanumérique d'une PlanetAlpha d'Element

En vous inspirant des tests du TP précédent, créez le fichier **S07_TP14_02.py** pour y coder la classe Main suivante. La constante de classe AUTHORIZED_TYPES permet de contrôler les classes d'Element qui peuvent peupler la planète.

```
Main
- AUTHORIZED_TYPES: set{type}
- planet_alpha: PlanetAlpha
__init__(planet_name: str, latitude_cells_count: int, longitude_cells_count: int): void
get_planet(): PlanetAlpha
add_element(cell_number: int, element: Element): void
add_element_randomly(element: Element): void
populate(types_count: dict(type:int)): void
__repr__(): str
```

Le code ci-dessous doit produire les affichages qui suivent.

```
>>> random.seed(10)
>>> app = Main("Terre", 5, 10)
>>> app.populate({Lion: 7, Cow: 3})
>>> app.populate({Water: 10, Herb: 100})
>>> print(app)
>>> nw_neighborhood = app.get_planet().get_cell_neighborhood_numbers(0, PlanetAlpha.WIND_ROSE)
>>> app.get_planet().die(0)
>>> for cell in nw_neighborhood:
>>>     app.get_planet().die(cell)
>>> print(app)
```

```
***** Planet with 50 places (■) available *****
Terre (50 habitants)

***** Planet with 50 places (■) available *****
Terre (41 habitants)

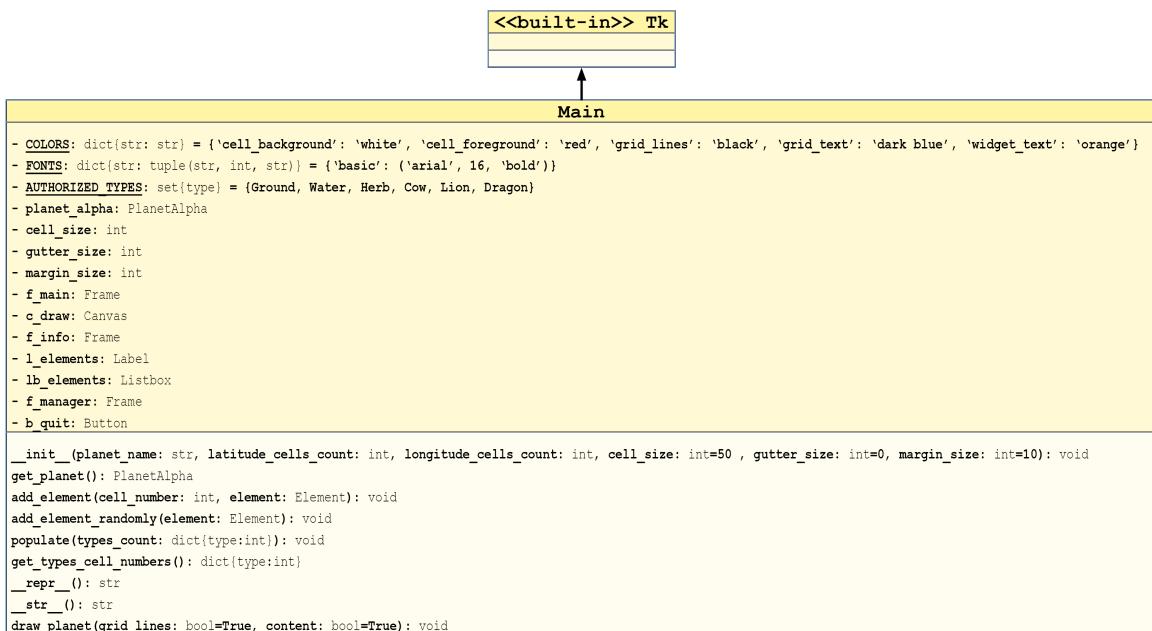
```

7.2.3 Application d'affichage tkinter d'une PlanetAlpha d'Element

En utilisant et en vous inspirant des classes de cette semaine, réaliser dans un fichier **S07_TP14_03.py** la classe suivante qui permet de construire une application qui spécialise une fenêtre Tk du module `tkinter` pour afficher dans un `Canvas` la simulation d'une `PlanetAlpha` composée de classes « feuilles » de la hiérarchie d'`Element`.

L'interface aura également une `Frame` contenant des informations générales (un `Label`) et détaillées (`Listbox`). Le bouton « Quitter » sera également placé dans une `Frame` de gestion de l'application.

La méthode `get_types_cell_numbers` retourne un dictionnaire dont les clés sont les classes autorisées et les valeurs leur nombre de représentant. Elle sera utilisée pour les informations détaillées dans la `Listbox`.

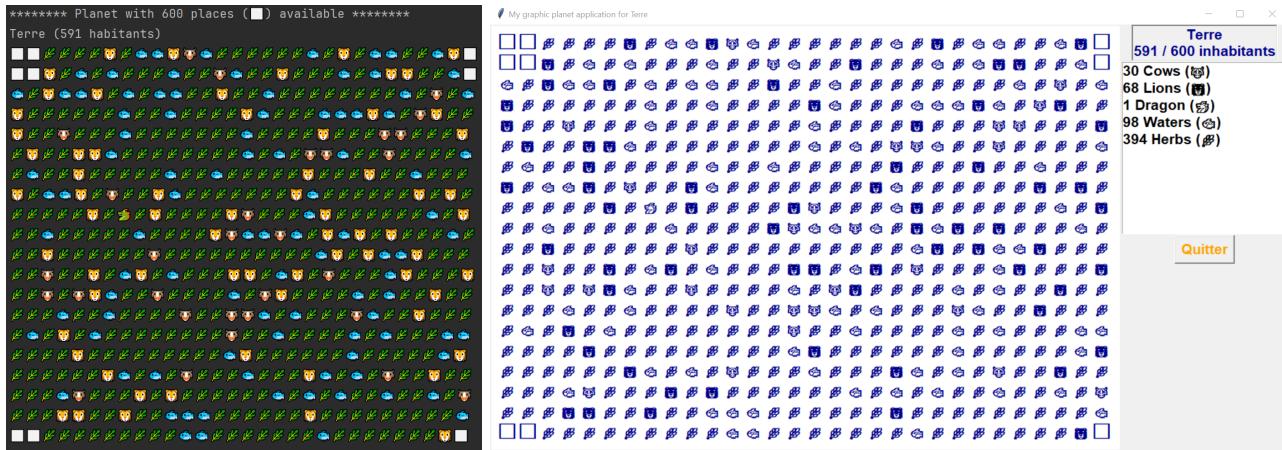


```

>>> LINES_COUNT = 20
>>> COLUMNS_COUNT = 30
>>> CELL_SIZE = 50
>>> GUTTER_SIZE = 0
>>> MARGIN_SIZE = 10
>>> app = Main("Terre", LINES_COUNT, COLUMNS_COUNT, CELL_SIZE, GUTTER_SIZE, MARGIN_SIZE)
>>> INHABITANTS_TEST = {Lion: 70, Cow: 30}
>>> RESOURCES_TEST = {Water: 100, Herb: 1000}
>>> app.populate(INHABITANTS_TEST)
>>> app.populate(RESOURCES_TEST)
>>> nw_neighborhood = app.get_planet().get_cell_neighborhood_numbers(0, PlanetAlpha.WIND_ROSE)
>>> app.get_planet().die(0)
>>> for cell in nw_neighborhood:
>>>     app.get_planet().die(cell)
>>> app.draw_planet(False)
>>> print(app)
>>> app.mainloop()

```

Les deux dernières instructions du code ci-avant produisent respectivement les deux sorties alphanumériques et graphiques suivantes.



8 CM08 (11/3/2024) - Multihéritage, classes abstraites, interfaces, présentation projet

Notions vues en cours :

- Spécialisation d'un **Canvas**
- Sélection et modification d'un **item** dans un **Canvas** par l'ajout de **tags**
- Multi-héritage **Canvas** et **PlanetAlpha**
- Automates cellulaires de *Conway* et *Turning Machine*
- Jeu du *Snake*
- rédiger un cahier des charges fonctionnel (pré-rapport)

8.1 TP15 - De PlanetAlpha à PlanetTk

Il s'agira dans le fichier **S08_TP15.py** de proposer la classe **PlanetTk** à la base du projet fil rouge. La classe doit être à la fois une **PlanetAlpha** et un **Canvas**. La planète ainsi conçue permettra de produire dans la console une planète alphanumérique d'**Element** mais également d'en afficher une grille comme dans le programme du TP précédent.

Une différence toutefois concerne le choix graphique pour éviter de détruire et créer trop souvent dans le **Canvas** des **items** pour dessiner les représentants textuels des **Element** lorsqu'ils changent. De plus, par nature en **tkinter**, les éléments dessinés dans le **Canvas** sont des objets mais non directement récupérables car les méthodes de création d'un dessin sur un **Canvas** ne retournent qu'un identifiant de type **int**. Cet identifiant peut être utilisé pour rechercher ou modifier les attributs d'un dessin à travers de nombreuses méthodes des **Canvas** mais nous allons plutôt utiliser une alternative avec l'utilisation de **tags**.

L'implantation devra respecter les indications suivantes :

- le constructeur aura un paramètre indiquant l'ensemble des classes d'`Element` autorisés sur la `PlanetTk` (l'`Element Ground` est systématiquement ajouté aux `Element` autorisés).
- chaque modification d'une case de la grille entraînera le changement approprié de la position, du texte ou de la couleur du dessin correspondant au nouvel `Element` contenu. Il s'agira de jouer avec les coordonnées de l'item pour un déplacement ; avec le représentant textuel de l'`Element` pour la forme du dessin ; avec les couleurs de fond (`background_color`) et de premier plan (`foreground_color`) du `Canvas` pour le rendre visible ou invisible.
- chaque dessin se verra attribué un `tag` qui pourra se substituer à un identifiant. L'intérêt est de pouvoir utiliser les méthodes sur tous les dessins qu'un `tag` donné sélectionne (voir exemple).

Dans l'exemple du code ci-après, le rectangle de 30 pixels de côté et positionné selon les coordonnées diagonaux 10, 10, 40, 40 correspond dans la grille à la cellule numéro 3 qui contient un `Dragon`. Son dessin sur le `Canvas c_grid` est paramétré pour lui attribuer un `tag` '`c_3`' qui permettra de le rechercher uniquement avec le numéro de cellule qui lui est associé. De même le texte (code utf-8 du `Dragon`) positionné en son centre sera paramétré pour lui attribuer le `tags` '`t_3`'. Ceci permet par exemple de positionner cet élément au dessus de la pile des dessins du `Canvas` avec la méthode `lift`. Sans cette dernière instruction le symbole du dragon n'apparaîtrait pas car le rectangle a été (fort mal à propos) construit par dessus. Les couleurs du texte et du fond sont ensuite changées également grâce aux `tags`.

```
>>> root = Tk()
>>> c_grid = Canvas(root, bg="white", width=80, height=50)
>>> c_grid.pack()
>>> x, y, cell_number, cell_size, name = 10, 10, 3, 30, Dragon
>>> c_grid.create_text(x + cell_size // 2, y + cell_size // 2, text=str(name()), tags=(f't_{cell_number}',))
>>> c_grid.create_rectangle(x, y, x + cell_size, y + cell_size, tags=(f'c_{cell_number}',))
>>> c_grid.lift('t_3')
>>> c_grid.itemconfigure('t_3', fill="dark blue")
>>> c_grid.itemconfigure('c_3', fill="yellow")
>>> root.mainloop()
```



FIGURE 1 – Affichage du rectangle et du texte sans ou avec l'instruction `lift`

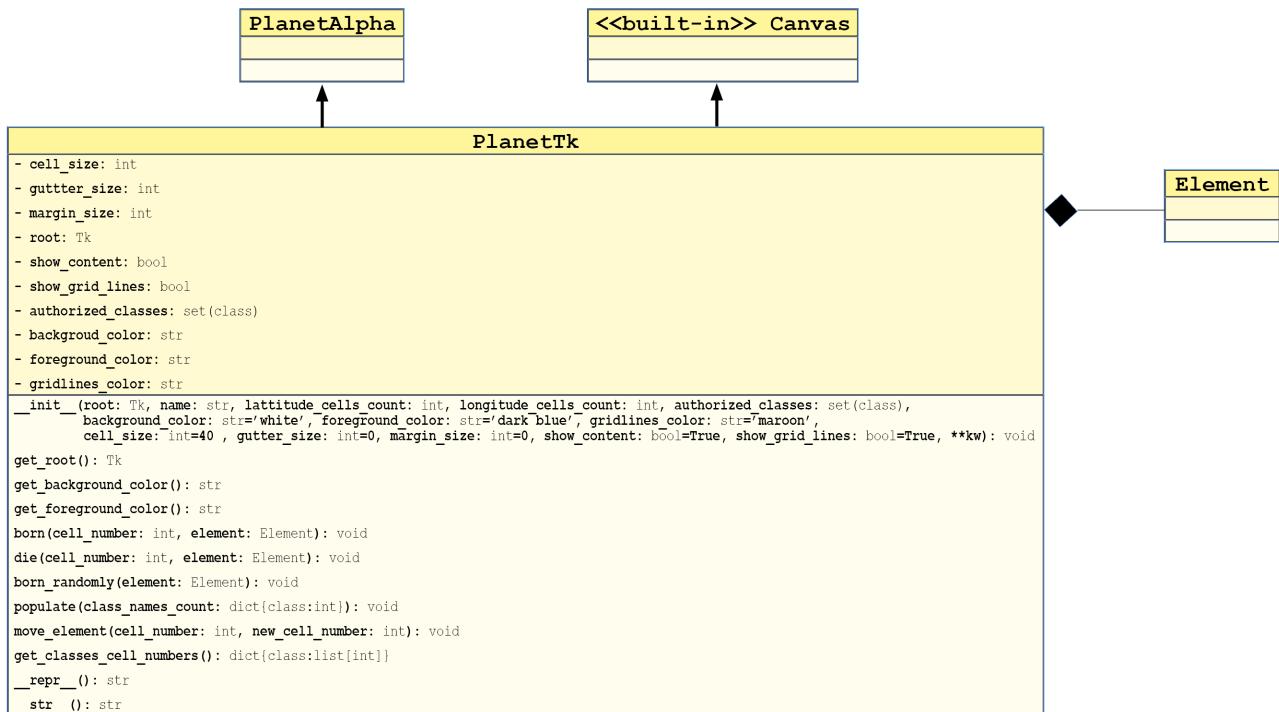
Attention : le programme devra gérer toutes les modifications d'affichage des `Element` en maintenant ce système de `tags`.

Remarque : les méthodes de modification ou d'obtention d'un attribut de `widget` sont `cget` et `configure`. Pour modifier ou obtenir l'attribut d'un dessin il faut à la place utiliser les méthodes de `Canvas` `itemcget` et `itemconfigure`. D'autres méthodes comme `gettags` peuvent également être utiles.

L'arborescence actuelle de vos dossiers et fichiers utiles au TP 15 devrait suivre l'architecture suivante :

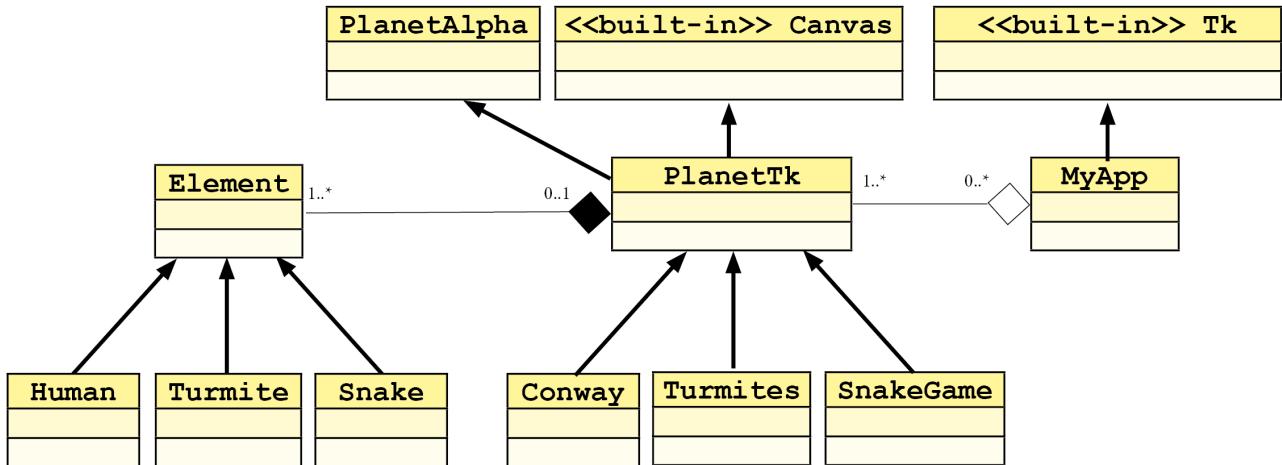
```
TP_POO
|____ TEXTS
|____ config.py
|____ S01_TP01_template.py
|____ S01_TP02_template.py
|____ S02_TP03_template.py
|____ S02_TP04_template.py
|____ S03_TP05_template.py
|____ S03_TP06_template.py
|____ S04_TP07_template.py
|____ S04_TP08_template.py
|____ S05_TP09_01_template.py
|____ S05_TP09_02_template.py
|____ S05_TP10_template.py
|____ S06_TP11.py
|____ S06_TP12.py
|____ S07_TP13_01.py
|____ S07_TP13_02.py
|____ S07_TP13_03.py
|____ S07_TP14_01.py
|____ S07_TP14_02.py
|____ S07_TP14_03.py
|____ S08_TP15.py
    import tkinter as tk
    from S06_TP11_planetalpha import PlanetAlpha
    from S07_TP14_1_element import *
```

Le diagramme des classes final est décrit ci-après.



8.2 Présentation du projet

Il s'agira de construire 3 classes spécialisant `PlanetTk` et de les intégrer dans une classe `MyApp` qui permette de tester graphiquement chacune d'elle selon le diagramme des classes suivant.



Les supports décrivant les règles de construction des 3 classes `Conway`, `Turmites` et `SnakeGame` sont fonctionnellement décrites dans les 3 sous-sections suivantes. A vous de les traduire en python orienté objet en respectant/exploitant/augmentant au mieux l'architecture imposée.

8.2.1 Automate cellulaire / Jeu de la vie / jeu de *Conway*

Selon [wikipedia](#) un automate cellulaire consiste en une grille régulière de « cellules » contenant chacune un « état » choisi parmi un ensemble fini et qui peut évoluer au cours du temps. L'état d'une cellule au temps $t+1$ est fonction de l'état au temps t d'un nombre fini de cellules appelé son « voisinage ». Un des automates cellulaires le plus célèbre est celui d'un jeu « à zéro joueur » : le jeu de la vie, parfois nommé du nom de son inventeur, le jeu de *Conway* (https://fr.wikipedia.org/wiki/Jeu_de_la_vie).

Le jeu de la vie sera simulé par une `PlanetTk` qui ne peut avoir qu'un unique type d'habitant : les `Human`. Ceux-ci n'ont aucune capacité de déplacement mais peuvent naître ou mourir selon certaines conditions de leur voisinage (les 8 cellules adjacentes horizontalement, verticalement et diagonalement).

Les règles d'évolution **SIMULTANÉE** dans toutes les cases sont les suivantes :

- une cellule libre possédant exactement 3 voisins de type `Human` entraîne une naissance sur la cellule libre, sinon elle n'évolue pas ;
- une cellule occupée par un `Human` et possédant 2 ou 3 voisins de type `Human` n'évolue pas
- une cellule occupée par un `Human` et possédant plus de 3 voisins de type `Human` meurt par étouffement
- une cellule occupée par un `Human` et possédant moins de 2 voisins de type `Human` meurt par isolement

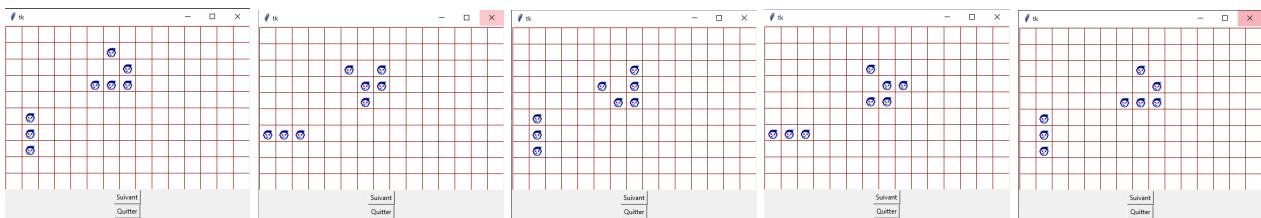
Attention : la simultanéité de l'évolution implique qu'à l'instant t , l'évolution d'une case ne doit pas influer sur le voisinage d'une autre (et donc sur son évolution). Il faudra donc évaluer le voisinage sur une copie de la grille pour faire les modifications éventuelles dans la grille originale.

Remarque : l'existence d'un **Human** dans la cellule courante à $t+1$ dépend de la véracité à t du booléen :

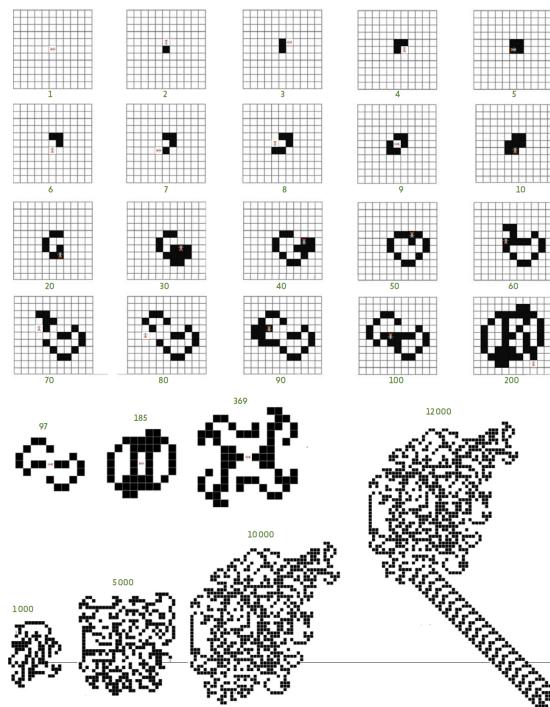
$$S == 3 \text{ or } E == 1 \text{ and } S == 2$$

avec **S** le nombre de **Human** dans le voisinage (de 0 à 8) et **E** le nombre de **Human** sur la cellule courante (0 ou 1)

Voici ci dessous un exemple d'évolution d'une planète avec des configurations de départ particulières qui semblent faire émerger des animations se reproduisant indéfiniment à l'identique : la première sur place en 2 étapes (un « clignotant » de la catégorie des « oscillateurs ») et la seconde avec un mouvement diagonal en 4 étapes (un « planeur » de la catégorie des « vaisseaux »)



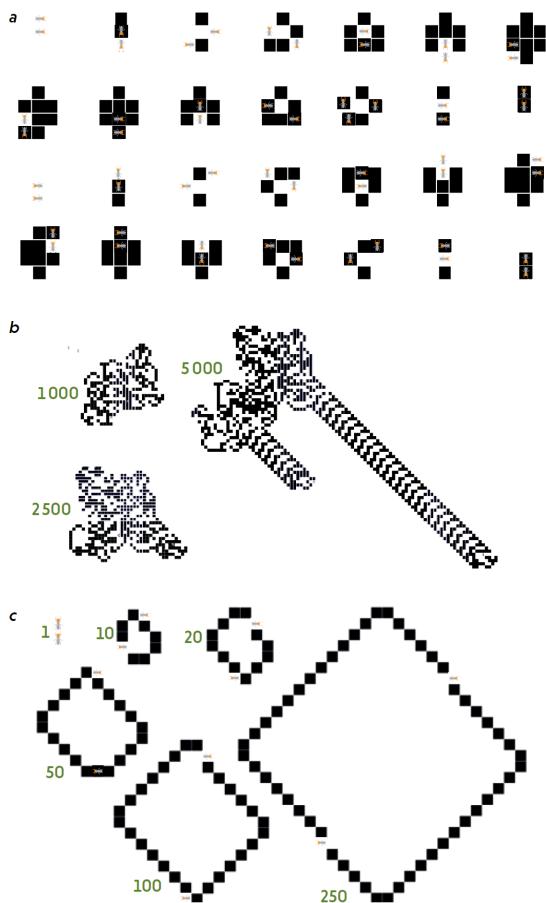
8.2.2 Turning Machine / Fourmi de Langton / Turmites



L'automate cellulaire de la fourmi de *Langton* sera représenté par un drôle d'animal, une **Turmite**, qui se déplace sur une grille en laissant ou effaçant une trace colorée dans les cases visitées. Quand elle est sur une case blanche, elle tourne de 90 degré vers la droite et avance d'une case en laissant une trace colorée sur la cellule quittée. Quand elle est sur une case colorée, elle tourne de 90 degré vers la gauche et avance d'une case en retirant la trace colorée de la cellule quittée.

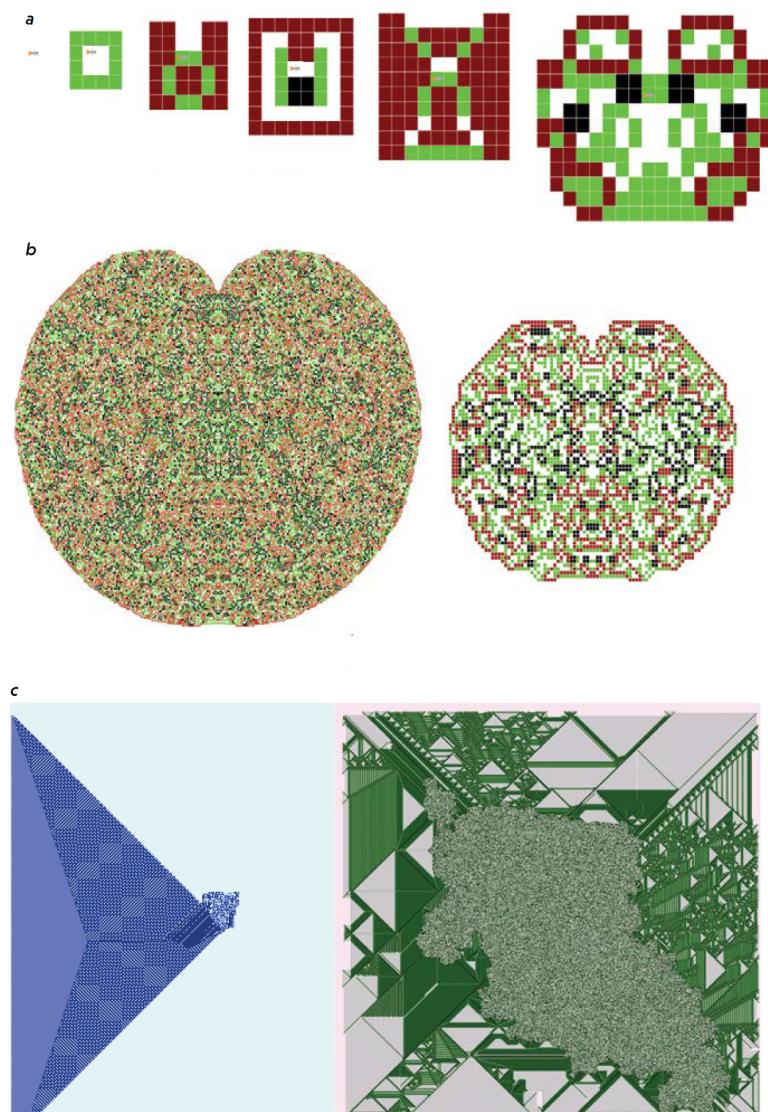
La figure précédente présente le cheminement d'une seule **Turmite** laissant une trace noire et démarrant sur un plan tout blanc. Aux étapes 97, 185 et 360, les dessins des cases noires ont un centre de symétrie. Après un comportement complexe pendant 10102 étapes, elle avance en diagonale de façon répétitive par cycles de 104 étapes : c'est « l'autoroute ».

Tout se complique lorsqu'il y a plus d'une fourmi. Il apparaît notamment des cycles. En (a) de la figure ci-dessous sont indiquées les 28 étapes qu'empruntent cycliquement deux fourmis placées initialement côté à côté sur un plan blanc. Parfois, chaque fourmi construit son autoroute. Ainsi, en (b), deux fourmis, après une période confuse, construisent chacune leur autoroute. On est alors tenté de généraliser et de conjecturer que plusieurs fourmis donnent soit un cycle, soit plusieurs autoroutes. Cette conjecture est hélas fausse : des croissances infinies autres que l'autoroute apparaissent. La création d'un carré qui grandit indéfiniment en laissant son intérieur vide est l'un de ces comportements infinis non répétitifs (c).



Une Turmite est en fait une fourmi sur un plan où, au lieu d'une seule trace colorée, il y a n couleurs possibles (par exemple quatre), auxquelles sont associées n rotations (par exemple +90, +90, -90, -90, auquel cas la Turmite est notée 1100). Les Turmite dont le code des rotations ne comporte que des doubles 0 et des doubles 1 (par exemple 1100, 111100, 00110011, etc.) ont l'extraordinaire propriété de produire régulièrement des dessins symétriques.

La figure ci-dessous montre quelques-unes de ces étapes symétriques dont la Turmite 1100 (a). Les Turmite donnent naissance à des dynamiques très différentes et produisent des dessins étonnantes que l'on peut voir comme un art automatique imprévisible. Les images proposées ont été réalisées avec un programme de Dean Tersigni (voir www.thealmightyguru.com/Wiki/index.php?title=Langton%27s_ant).



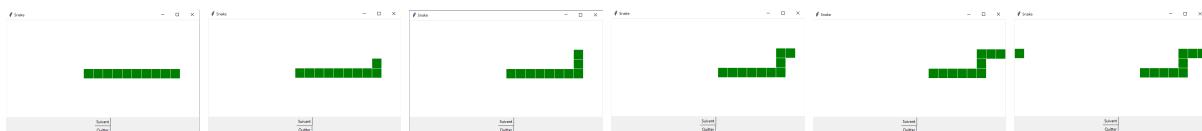
8.2.3 Jeu du *Snake*

Le jeu du *snake*, de l'anglais signifiant « serpent », est un genre de jeu vidéo dans lequel le joueur dirige un serpent qui grandit et constitue ainsi lui-même un obstacle. Le joueur contrôle une longue et fine ligne semblable à un serpent, qui doit potentiellement slalomer entre des obstacles qui parsèment le niveau. Pour gagner chacun des niveaux, le joueur doit faire manger à son serpent un certain nombre de pastilles similaire à de la nourriture, allongeant à chaque fois la taille du serpent. Alors que le serpent avance inexorablement, le joueur ne peut que lui indiquer une direction à suivre (tourner à gauche ou à droite par rapport à sa direction courante) afin d'éviter que la tête du serpent ne touche les murs ou son propre corps, auquel cas il risque de mourir. Certaines variantes proposent des niveaux de difficulté dans lesquels varient l'aspect du niveau (simple ou labyrinthique), le nombre de pastilles à manger, l'allongement du serpent ou encore sa vitesse.

Le jeu sera ici aussi construit à partir de `PlanetTk` et un `Element Snake` dont les paramètres supplémentaires sont la taille, la vitesse et la direction de départ. Une liste de cases contiguës permettra de conserver l'information de la position courante de chaque morceau du serpent ; la valeur en position 0 correspondant au numéro de cellule de la tête.

Pour faire avancer le serpent il faudra développer une programmation événementielle en assignant :

- la fonction de démarrage/arrêt du serpent à la barre espace du clavier
- la fonction de tourner de 90 degré vers la gauche au bouton gauche de la souris
- la fonction de tourner de 90 degré vers la droite au bouton droit de la souris



Remarque : Pour faire avancer le serpent dans la direction courante, il ne faut pas déplacer toutes les cases mais 1 seule ! Pour y parvenir, il suffit, à chaque évolution de la grille, de déplacer la case de queue du serpent sur la cellule vers laquelle doit se diriger la tête du serpent.

8.3 Démarrage projet

Une fois par groupe de 2 ou 3 il s'agit de cloner un dépôt GIT sur la forge de l'université AU BON ENDROIT et de rajouter en membre propriétaire du projet votre encadrant de TP et moi-même. Il faut ensuite réfléchir aux différentes fonctionnalités et amélioration à rajouter aux versions de base de la classe `Element` et des spécialisations de `PlanetTk` et se répartir le travail sous la forme d'un cahier des charges fonctionnels.

8.3.1 Rédiger un cahier des charges fonctionnel

Tous les membres d'un groupe doivent obligatoirement s'appuyer sur les mêmes classes de base du dépôt. Pour leur réalisation individuelle ils devront ajouter des classes et des méthodes au bons endroits pour maximiser la cohérence et factoriser au mieux le code.

En plus des bases à réaliser, plusieurs parties du programme final peuvent être décrites sous la forme de fonctionnalités supplémentaires. Les exemples ci-dessous ne sont ni obligatoires, ni exhaustifs.

- jeu de **Conway** : affichage d'informations sur l'état du plateau, programmation événementielle pour construire une configuration de départ à la souris, permettre aux **Human** de mourir de vieillesse, ...
- **Turmites** : affichage d'informations, fonctionnalités décrites dans la sous-section pour améliorer la fourmi de **Langton** jusqu'aux vraies **Turmite** en passant par la possibilité de plusieurs fourmis...
- **GameSnake** : gestion de la mort, des obstacles, des niveaux, du score, de la nourriture qui fait grandir, du poison qui fait rétrécir voire mourir, de l'accélération...
- interface **MyApp** : toute la partie de gestion du jeu, des planètes, des éléments...
- nouvelles règles pour une autre planète...

Les fonctionnalités de base incluent les 3 spécialisations quelque soit le nombre de personnes dans le groupe. Le fait d'être 2 ou 3 pourra en revanche avoir un impact que sur les fonctionnalités supplémentaires que vous proposez dans votre pré-rapport.

Attention : Toutes les classes et méthodes devront être correctement associées à des *docstrings* et une documentation minimale automatique sera générée avec un module tel que **pydoc** présenté lors du CM10

Remarque : Chaque *docstring* devrait contenir le ou les auteurs de la classe/méthode commentée afin que l'information apparaisse directement dans la documentation générée

8.3.2 Évaluation

Date limite de remise du pré-rapport (cahier des charges fonctionnel) : date du dernier TP de la semaine du 11 mars + 6 jours

Date limite de remise du rapport (explication des différences entre les prévisions du pré-rapport et le travail accompli) : vendredi de la semaine précédent celle des oraux (environ 10mn maximum par étudiant présentation, questions et discussions comprises).

Attention : Barème uniquement indicatif

(se référer en priorité aux indications données par votre encadrant de TP sur le processus à suivre pour l'évaluation orale) :

- 8 points sur les fonctionnalités de base montrées en cours pour exploiter obligatoirement les classes finales des TP (**PlanetTk** et **Element**) dans les trois spécialisations demandées.
- 6 à 8 points sur la qualité du pré-rapport, du rapport final et des fonctionnalités supplémentaires. Cette note peut être individualisée si la répartition du travail est très déséquilibrée sans justifications pertinentes.
- 4 à 6 points pour l'oral sur la qualité de votre investissement individuel (utilisation du GIT, capacité à expliquer votre code et l'architecture globale, justification des différences éventuelles décrites dans le rapport final entre ce qui été prévu dans le pré-rapport et ce qui a été réellement fait).

Exemple pour le projet fil rouge de cette année :

Contrôle Continu 2023/2024 - L1 INFO - Projet Fil Rouge en binôme ou trinôme															Oral				
Pré-rapport (cahier des charges fonctionnel et réparti)				Rapport (Analyse comparative et documentation)				Travail							Oral				
Présentation	Fonctionnalités	Diagrammes	Répartition	Présentation	Fonctionnalités	Diagrammes	Documentation	Grid	PlanetAlpha	PlanetTk	Element	Conway	Turmites	Snake	GIT	Sup	Présentation	Répartition	Questions
0,5	1	0,5	1	0,5	0,5	0,5	0,5	0,5	0,5	1	1	1	1	1	1	3	0,5	0,5	4