Module & interface

Exo 1 : créer un module `vecteurs.py`

a) proposer une fonction `creation(a : float, b : float) -> tuple` qui renvoie un tuple représentant les coordonnées du vecteurs.

b) proposer une fonction `somme(v1 : tuple, v2 : tuple) -> tuple` qui renvoie le vecteur somme $v1 + v2$.

c) proposer une fonction `egaux(v1 : tuple, v2 : tuple) -> bool`.

d) proposer une fonction `colineaires(v1 : tuple, v2 : tuple) -> bool`.

Exo 2 : créer un module `dateNaissance.py` de sorte que la date soit représenter par trois entiers représentant l'année, le mois et le jour.

a) proposer une fonction `creation(a : int, m : int, j : int) -> tuple` qui renvoie un tuple représentant la date de naissance.

b) proposer une fonction `age(date : tuple) -> int` qui renvoie l'âge actuel.

c) proposer une fonction `afficher(date : tuple) -> None` qui affiche la date de naissance.

d) proposer une fonction `nextAnnif(date : tuple) -> int` qui renvoie le nombre de jours d'ici le prochain anniversaire. _on ne traitera pas le cas des années bissextiles_

e) modifier votre code afin d'obtenir une implémentation défensive.

f) proposer un fichier principal qui teste votre module.

Exo 3 : on souhaite créer un module `tableau.py` qui créé un tableau de taille fixe, ne contenant des éléments que d'un seul type (int, float ou str). Le début du module se présente ainsi :

a) compléter les fonctions ci-contre.

La fonction `_nextIndice()` renvoie l'indice de la prochaine cellule libre (*i.e.* avec `None`).

La fonction `add()` lève un `IndexError` si le tableau est plein sinon ajoute l'élément `newe` s'il est de même type que les précédents sinon lève une erreur.

b) proposer une fonction `multi(tab : list, k : int) -> None` qui multiplie tous les éléments existant par k.

c) proposer une fonction `contient(tab : list, element : object) -> bool`.

d) proposer une fonction `egaux(tab1 : list, tab2 : list) -> bool`

e) `python` permet de renvoyer des slices (des tranches) de tableaux. Ainsi si `tab=[10, 11, 12, 13]` alors `tab[1:3]` renverra un sous-tableau contenant les indices 1 à 3 exclu (*i.e.* `[11, 12]`).

Proposer une fonction `tranche(tab : list, imin : int, imax : int) -> list` qui renvoie une tranche entre `imin` et `imax` exclu.

f) proposer un fichier principal qui teste votre module.

```
def creation(tMax : int) -> list :
    return ...*[None]

def _nextIndice(tab : list) -> int :
    i = 0
    while ......... None :
        i += 1
    return ...

def add(tab:list, newe:object)->None:
    i = _nextIndice(...)
    if ... :
        tab[0] = ...
    elif i >= ... :
        raise IndexError(...)
    elif isinstance(..., type(...)) :
        tab[i] = ...
    else :
        raise ...
```

Exo 4 : créer un module `dictionnaire.py` à l'aide d'un tableau contenant des tuples `(clef, valeur)` où clef et valeurs seront des entiers ou des chaînes de caractères (*cf.* simplifie la fonction `clefsAssociees()`).

a) proposer une fonction `creation() -> list`

b) proposer une fonction `clefs(dico : list) -> list` qui renvoie la liste de toutes les clefs.

c) proposer une fonction `valeurs(dico : list) -> list` qui renvoie la liste de toutes les valeurs.

d) proposer une fonction `add(dico : list, clefVal : tuple) -> None` qui ajoute le tuple contenant la clef et la valeur au dictionnaire si la clef n'existe pas déjà. La valeur est mise à jour si la clef existe déjà.

e) proposer une fonction `valeur(dico : list, clef : [int,str]) -> [int,str]` qui renvoit la valeur associée à `clef`.

f) proposer une fonction `efface(dico : list, clefVal : tuple) -> None` qui efface le tuple (clef,valeur) si celui-ci existe.

g) proposer une fonction `clefsAssociees(dico : list, val : [int,str]) -> list` qui renvoit la liste des clefs de valeur associée égale à `val`.

Programmation Orientée Objets

Exo 5 : reprendre l'exercice 1 en créant une classe Vecteur.

Exo 6 : on reprend la classe Vecteur de l'exercice précédent

a) renommer la fonction `somme` en `__add__` et tester $v1 + v2$

b) proposer une fonction `__sub__(self, v2 : 'Vecteur') -> 'Vecteur'` qui définit la soustraction

c) proposer une fonction `__mul__(self, k : float) -> 'Vecteur'` qui définit la multiplication

Exo 7 : reprendre l'exercice 2 en créant une classe DateNaissance.

Exo 8 : on souhaite une classe Fraction qui définira les opérations $+$, $-$, $<$, $==$, $\times$ et $\div$.

a) définir le constructeur `__init__(self, a : int, b : int) -> 'Fraction'` (si $b = 0$, une erreur est levée).

b) définir les fonctions `__repr__` et `__str__`

c) définir les fonctions `__add__`, `__sub__`, `__lt__` et `__eq__`.

d) définir les fonctions `__mul__` et `__truediv__`.

e) définir la fonction privée `_simplifier(self) -> None` qui rend la fraction irréductible et modifier les fonctions $+$, $-$, $\times$ et $\div$ afin qu'elles renvoyent une fraction irréductible.

Exo 9 : un tableau `python` est dans, les faits, dynamique. Agrandir l'allocation mémoire du tableau et on copie tout le tableau dans le nouveau bloc mémoire en ajoutant le nouvel élément, cela demande trop d'accès mémoire et est inefficace. On choisit donc de prévoir une taille standard de tableau (*e.g.* 4 cases ici) et l'on doublera la taille maximale du tableau à chaque dépassement.

```
class Tableau:
    """
    Tableau() créé un objet Tableau avec self.tab=[None,None,None,None]
    Tableau(1,2,3,4,5,6) créé un objet Tableau avec self.tab=[1,2,3,4,5,6,None,None]
    self.tab est par défaut de taille 4 et a pour prochain indice libre i = 0
    Si la place manque, sa taille est doublée et jamais ne diminue sauf avec freeSpace()
    """
    def __init__(self, *args : tuple) -> 'Tableau' :
        self.tab = 4*[...]
        self.i = 0
        if len(args) != 0 :
            for e in args :
                self.add( ... )

    def add(self, e : object) -> None :
        if self.i == len(self.tab) :
            n = len(self.tab)
            self.tab = self.tab + ...
        self.tab[...] = ...
        self.i += ...
```

a) compléter la classe ci-dessus

b) proposer une fonction `__len__(self)`.

c) proposer une fonction `__str__(self)` qui renvoie la chaîne "[1,2]" si `self.tab=[1,2]`

d) proposer une fonction `__repr__(self)` qui renvoie la chaîne "Tableau(1,2)" si `self.tab=[1,2]`

e) proposer une fonction `__getitem__(self, j : int) -> object`.

f) proposer une fonction `__setitem__(self, j = int, e : object) -> None`.

g) proposer une fonction `__contains__(self, e : object) -> bool`

h) proposer une fonction `__eq__(self, tab2 : 'Tableau') -> bool`

i) Approf : proposer une fonction `efface(self, e : object) -> None` qui efface la 1ère occurence de l'élément `e` sans laisser aucun trou dans `self.tab` (*e.g.* `[1, None, 2, None]` est incorrect)

j) Approf : proposer une fonction `freeSpace(self) -> None` qui retaille `self.tab` afin que sa taille soit la plus petite puissance de 2 contenant tous ces éléments ($\neq$ *None*).