T.P. 17 : un T.P. qui a de la classe.... polynômes

Comme annoncé en cours, le but de ce T.P. est de se familiariser avec les classes, en écrivant une classe `Polynome` sur le modèle de la classe `Matrice` du cours (que vous pourrez aussi compléter si vous finissez le TP sur les polynômes avant).

a) Créer votre classe et écrire la méthode `__init__` qui prend comme argument une liste de coefficients. Les attributs naturels d'un polynôme sont sa liste de coefficients, éventuellement son degré. Dans la suite, on convient que le degré du polynôme nul est `-1`.

b) Le problème de stocker le degré comme attribut demande qu'il soit quand même recalculé après des opérations qui modifieraient ce polynôme. Ecrire une méthode `deg` telle que `P.deg()` renvoie le degré de `P`.

c) Bon on a quand même envie d'avoir un joli affichage pour nos polynômes. Ecrire une méthode `__repr_` de telle sorte que si on tape

```
>>>P=Polynome([1,2,3,5])
>>>P
```

on obtienne (par exemple !) :

```
1 + 2 X^1  + 3 X^2  + 5 X^3
```

d) Vous pouvez aussi écrire une méthode `__str__` qui renvoie plus ou moins la même chose, à votre guise, et qui permet d'utiliser le `print`.

e) Ecrire une méthode `coef` de telle sorte que si `P` est un polynôme, `P.coef(i)` renvoie le `i`-ième coefficient de `P`. Si vous préférez vous pouvez aussi écrire plutôt une méthode magique `__getitem__` qui donnera la même chose dès qu'on fait `P[i]`.

f) Si vous voulez modifier un coefficient via `P[i]=17`, il vous faudra écrire la méthode magique `__setitem__`. Vous pouvez gérer cela pour éviter les `out of range`.

g) Ecrire, hors de la classe, autrement dit en desindenté en dessous de la classe (et cela restera en dessous de ce que vous écrirez après dans la classe) une fonction `zero()` qui fabrique le polynôme nul, une fonction `monone(a,k)` qui fabrique le polynôme aX^k.

h) Ecrire la méthode magique `__add__` qui permet l'addition de deux polynômes. Vous pouvez si vous le souhaiter aussi y ajouter à l'intérieur l'addition d'un polynôme et d'un scalaire (avec `isinstance`) mais dans ce cas il faudrait traiter le cas où le scalaire est à droite ou à gauche (il existe une méthode magique `__radd__` pour traiter le cas où le polynôme est à droite du scalaire).

i) Ecrire la méthode magique `__mul__` qui permet la multiplication de deux polynômes ainsi que la multiplication d'un polynôme par un scalaire (eh oui il y a aussi un `__rmul__`)

j) Ecrire la méthode magique `__call__` qui permet d'évaluer votre polynôme sur un scalaire simplement avec la syntaxe `P(12)`. Si vous êtes plus ambitieux, en deuxième partie de T.P. vous pourrez imaginer comment évaluer votre polynômes sur des matrices soit de la classe `Matrix` de `numpy` soit de votre propre classe `Matrice` où même évaluer un polynôme sur un polynôme [1]. Pour l'évaluation vous pouvez aussi réviser l'algorithme de Horner.

k) Ecrire une fonction `MD` telle que `P.MD()` renvoie le monôme dominant de `P`.

l) Ecrire, hors de la classe, une fonction `divimod(A,B)` qui calcule quotient et reste de la division euclidienne de `A` par `B`. En déduire les méthodes magiques `__floordiv__` et `__mod__` qui permettent d'accéder à quotient et reste via `A//B` et `A%B`.

N.B. Le mieux est de savoir écrire l'algo de division euclidienne tout seul, mais sinon il est en pseudo-code dans le cours du K1.

m) Reprendre l'algorithme d'Euclide vu sur les entiers pour écrire une fonction `PGCD(A,B)` (hors de la classe). Mieux encore vous pouvez écrire une fonction `EuclideEtendu` que je vous laisse imaginer (cf. T.P. 9) pour avoir les coefficients de Bézout.

1. Ainsi vous pourrez méditer sur l'abîme de l'évaluation de P en X