

Activité 05 - Polynômes

Equipe Pédagogique LU1IN0*1

Consignes : Cette activité se compose d'une première partie guidée, suivie de suggestions. Il est conseillé de traiter en entier la partie guidée avant de choisir une ou plusieurs suggestions à explorer.

L'objectif de cette activité est l'étude de la représentation des polynômes dans le langage du cours, et leur manipulation.

Rappel. Un polynôme en l'indéterminée X sur l'anneau unitaire \mathcal{A} est une expression de la forme :

$$\sum_{i=0}^n a_i \cdot X^i$$

où n est un entier naturel, et où les a_i sont des éléments de \mathcal{A} . On dit que a_i est *le coefficient de degré i* du polynôme (par extension, le coefficient de degré i pour $i > n$ est 0).

1 Partie Guidée : Polynômes

Dans un premier temps, on décide de représenter un polynôme en l'indéterminée X sur l'anneau unitaire \mathbb{Z} par **la suite $(a_i)_{0 \leq i \leq n}$ de ses coefficients, rangés dans l'ordre de degré croissant**. Ainsi un polynôme sera, en Python, une liste d'entiers.

On définit un alias de type et des exemples de polynômes :

```
Polyn = List[int]
ex1 : Polyn = [3, 0, 2]
ex2 : Polyn = [1, -1, 1, -1, 0]
ex3 : Polyn = [27]
ex4 : Polyn = []
```

Ainsi les constantes `ex1`, `ex2`, `ex3`, `ex4` représentent, respectivement, les polynômes X^2+3 , $-X^3+X^2-X+1$, 27 et 0, le polynôme nul.

Question 1. Le *degré*, d'un polynôme $\sum_{i=0}^n a_i \cdot X^i$ est le plus grand entier $i \in \llbracket 0; n \rrbracket$ tel que a_i est non-nul. Ecrire une fonction `degre`, qui renvoie le degré d'un polynôme.

```
assert degre(ex1) == 2
assert degre(ex2) == 3
assert degre(ex3) == 0
assert degre(ex4) == 0
assert degre([0,0,0,0,0]) == 0
```

Question 2. La somme de deux polynômes se fait terme-à-terme. Ecrire une fonction `somme` qui renvoie la somme de deux polynômes passés en argument.

```
assert somme(ex1, ex1) == [6, 0, 4]
assert somme(ex1, ex4) == ex1
assert somme(ex1, ex2) == [4, -1, 3, -1, 0]
```

Question 3. L'égalité entre deux polynômes ne peut pas être testée directement selon notre représentation. En effet, les listes `[3, 0, 2]` et `[3, 0, 2, 0]` sont différentes, mais représentent le même polynôme. La *forme normale* d'un polynôme $\sum_{i=0}^n a_i.X^i$ est $\sum_{i=0}^d a_i.X^i$ où d est le degré du polynôme.

Ecrire une fonction `normalise` qui prend en entrée un polynôme et renvoie sa forme normale.

```
assert normalise(ex1) == ex1
assert normalise(ex2) == [1, -1, 1, -1]
assert normalise([0,0,0,0,0]) == []
assert normalise([]) == []
```

Ainsi, pour tester l'égalité de deux polynômes, on peut maintenant tester l'égalité de leur forme normale.

Question 4. Le produit de deux polynômes $\sum_{i=0}^n a_i.X^i$ et $\sum_{j=0}^m b_j.X^j$ est obtenu de manière standard, en développant le terme $(\sum_{i=0}^n a_i.X^i).(\sum_{j=0}^m b_j.X^j)$

Ecrire une fonction `produit` qui prend en entrée deux polynômes et renvoie un polynôme correspondant à leur produit.

```
assert normalise(produit(ex1, ex4)) == []
assert normalise(produit(ex1, ex1)) == [9, 0, 12, 0, 4]
assert normalise(produit(ex1, ex2)) == [3, -3, 5, -5, 2, -2]
assert normalise(produit(ex1, ex3)) == [27 * 3, 0, 27 * 2]
assert normalise(produit([1, 1], [1, 0, 1])) == [1, 1, 1, 1]
```

2 Suggestion : Autres opérations

On pourra implémenter d'autres opérations sur les polynômes, par exemple :

- la multiplication d'un polynôme par un entier,
- la puissance entière d'un polynôme,
- la dérivation et l'intégration formelle d'un polynôme,
- la division euclidienne de deux polynômes,
- la recherche des racines d'un polynômes,
- la réalisation d'un "tableau d'étude de fonction" (cf. partie suivante),
- ...

```
assert normalise(derivee(ex1)) == [0, 4]
assert normalise(derivee(ex4)) == []
assert normalise(derivee(ex2)) == [-1, 2, -3]
```

3 Suggestion : Fonction associée

Au polynôme $\sum_{i=0}^n a_i.X^i$, on peut associer la fonction de $\mathbb{R} \rightarrow \mathbb{R}$:

$$x \mapsto \sum_{i=0}^n a_i.x^i$$

Ecrire une fonction `valeur` qui prend en entrée un polynôme et un flottant x et qui renvoie la valeur de la fonction associée au polynôme calculée en x .

```
assert valeur(ex1, 0) == 3
assert valeur(ex1, -1) == 5
assert valeur(ex4, 2.5) == 0
```

Puis écrire une fonction `courbe.poly` qui prend un polynôme P , une borne nx pour les abscisses, une borne ny pour les ordonnées et qui renvoie l'image de la courbe de la fonction associée à P dans la partie du plan comprise entre $(-nx, -ny)$ et (nx, ny) .

Image de la courbe de la fonction associée à $-X^3 + X^2 - X + 1$, entre $(-10, -100)$ et $(10, 100)$:



4 Suggestion : Présentation

La présentation sous forme de liste de coefficients croissants n'est pas particulièrement lisible, et on lui préfère une écriture plus "mathématique" qui fait apparaître les degrés et l'indéterminée X .

Ecrire une fonction `cdp` ("chaîne de polynôme") qui prend en entrée un polynôme et renvoie une chaîne de caractères correspondant à une écriture lisible de ce polynôme.

Par exemple (on pourra se restreindre à des présentations moins élégantes) :

```
assert cdp(ex1) == "2.X^2 + 3"
assert cdp(ex4) == "0"
assert cdp(ex2) == "-X^3 + X^2 - X + 1"
assert cdp(ex3) == "27"
assert cdp(produit(ex1, ex2)) == "-2.X^5 + 2.X^4 - 5.X^3 + 5.X^2 - 3.X + 3"
```

Puis écrire la fonction inverse, `pd` ("polynôme de chaîne").

Par exemple :

```
assert pd(cdp(ex1)) == normalise(ex1)
assert pd(cdp(ex2)) == normalise(ex2)
assert pd(cdp(ex3)) == normalise(ex3)
assert pd(cdp(ex4)) == normalise(ex4)
assert pd("-X^2 + 3.X") == [0, 3, -1]
assert pd("X^4 - 3.X") == [0, -3, 0, 0, 1]
assert pd("0.X^2 + 3.X") == [0, 3]
```

On pourra ensuite écrire des fonctions qui lisent des polynômes dans un fichier `.txt` et qui écrivent le résultat de la somme de ces polynômes (ou d'une autre opération) dans un autre fichier (ou dans le même fichier, à la fin).

5 Suggestion : Interpolation

Si $\{(x_i, y_i)\}_{1 \leq i \leq n}$ sont les coordonnées de n points du plan réel d'abscisses toutes différentes, on peut montrer qu'il existe un unique polynôme à coefficients dans \mathbb{R} de degré $n - 1$ interpolant ces points (c'est-à-dire que la fonction associée au polynôme prend la valeur y_i en x_i pour chaque i).

Ecrire une fonction `interpolation` qui prend en entrée une liste d'abscisses x_i et une liste d'ordonnées y_i et qui renvoie, quand c'est possible, le polynôme à coefficients dans \mathbb{D} (les flottants de Python) qui interpole les points $\{x_i, y_i\}$.

On pourra implémenter la résolution d'un système de n équations à n inconnues selon, par exemple, la méthode du *pivot de Gauss* ou utiliser la formule du *polynôme d'interpolation de Lagrange*.

```
assert cdp(int_poly(interpolation([1, 0], [1, 0]))) == "X"
assert cdp(int_poly(interpolation([1, -1, 0], [0, 0, -1]))) == "X^2 - 1"
assert normalise(int_poly(interpolation([i for i in range(4, -1, -1)],
                                         [valeur(ex2, i) for i in range(4, -1, -1)])))
                                         == normalise(ex2)
```

(ici `int_poly` est une fonction qui convertit un polynôme à coefficients dans \mathbb{D} en un polynôme à coefficients dans \mathbb{Z})

6 Suggestion : Polynômes creux

(Remarque. Cette partie utilise des **listes de paires**. (Cours 06))

Représenter des polynômes par des listes de coefficients successifs est peu pratique lorsque que le polynôme est "creux", c'est-à-dire lorsqu'il possède de nombreux coefficients nuls, comme $X^{1000} - 1$ (qui est représenté par une liste de taille 1001).

Une autre manière de représenter les polynômes est l'utilisation d'une liste de couples $\{(c_i, d_i)\}_{0 \leq i \leq n}$ pour le polynôme $\sum_{i=0}^n c_i X^{d_i}$

Par exemple $[(3, 0), (2, 2)]$ représente $3 + 2X^2$ et $[(1, 1000), (-1, 0)]$ représente $X^{1000} - 1$.

On notera que les d_i peuvent ne pas être tous égaux, par exemple $[(1, 1), (2, 2), (-1, 1)]$ représente le polynôme $2X^2$

En Python on aura :

```
CPolyn = List[Tuple[int, int]]
# le membre droit de chaque element de la liste est positif
cex1 : CPolyn = [(3, 0), (2, 2)]
cex2 : CPolyn = [(-1, 1), (1, 0), (0, 4), (1, 2), (-1, 3)]
cex3 : CPolyn = [(27, 0)]
cex4 : CPolyn = []
cex5 : CPolyn = [(1, 1), (2, 2), (-1, 1)]
```

Ecrire de nouvelles versions des fonctions de la partie 1 pour cette représentation. Par exemple :

```
assert cdegre(cex1) == 2
assert cdegre(cex2) == 3
assert cdegre(cex3) == 0
assert cdegre(cex4) == 0
assert cdegre(cex5) == 2

assert cnormalise(cex1) == [(2, 2), (3, 0)]
assert cnormalise(cex2) == [(-1, 3), (1, 2), (-1, 1), (1, 0)]
assert cnormalise(cex3) == [(27, 0)]
assert cnormalise(cex4) == []
assert cnormalise(cex5) == [(2, 2)]
```

Ecrire des fonctions `standard_de_creux` et `creux_de_standard` qui permettent de passer d'une représentation à l'autre :

```
assert standard_de_creux(cex1) == normalise(ex1)
assert standard_de_creux(cex2) == normalise(ex2)
assert standard_de_creux(cex3) == normalise(ex3)
assert standard_de_creux(cex4) == normalise(ex4)
assert standard_de_creux(cex5) == [0, 0, 2]

assert creux_de_standard(ex1) == cnormalise(cex1)
assert creux_de_standard(ex2) == cnormalise(cex2)
assert creux_de_standard(ex3) == cnormalise(cex3)
assert creux_de_standard(ex4) == cnormalise(cex4)
assert creux_de_standard([0, 0, 2]) == cnormalise(cex5)
```
