TP n°19

Polynômes

1 Présentation

Dans ce TP, nous nous proposons de développer quelques fonctions permettant de manipuler des polynômes (à coefficients réels) en Python.
Nous guidons la construction des fonctions de base, permettant :

- de calculer le degré d'un polynôme ;
- de donner le coefficient dominant d'un polynôme non nul ;
- de déterminer si un polynôme est unitaire ;
- de calculer la somme de deux polynômes ;
- de calculer le résultat de la multiplication d'un polynôme par un scalaire ;
- de calculer le produit de deux polynômes ;
- de calculer une puissance de polynôme.

Nous proposons ensuite une application de quelques unes de ces « briques élémentaires » : le calcul d'une ligne du triangle de Pascal. Dans un prochain TP, nous ré-investirons le travail effectué ici, pour implémenter la division euclidienne de polynômes.

2 Représentation des polynômes en Python

La première question qu'il convient de se poser est :

Comment représenter un polynôme en Python ?

Un polynôme est, par définition, une suite de nombres réels, nulle à partir d'un certain rang. Nous choisissons d'oublier « tous les zéros qui terminent la suite » pour représenter le polynôme en Python, afin de n'avoir qu'un nombre fini de données à considérer (contrainte liée à la représentation en machine). Les coefficients restants (« ceux du début » donc) seront, quant à eux, placés dans une liste. Ainsi :

- le polynôme $1+X-X^2$ sera représenté par la liste `[1,1,-1]` ;
- le polynôme $X-3X^4+7X^5$ sera représenté par la liste `[0,1,0,0,-3,7]`.

Plus généralement le polynôme :

$$a_0+a_1X+a_2X^2+\ldots+a_nX^n$$

où $n\in\mathbb{N}$, $a_0\in\mathbb{R}$, $a_1\in\mathbb{R}$, $a_2\in\mathbb{R}$, $\ldots$, $a_n\in\mathbb{R}^*$, sera représenté par la liste

$$[a_0,a_1,a_2,\ldots,a_n]\,.$$

Le polynôme nul sera, quant à lui, représenté par la liste vide, i.e. : `[]`, conformément à notre choix de modélisation.
Nous devrons toujours garder à l'esprit :

- que le polynôme nul a une modélisation quelque peu singulière (la liste vide ne contenant aucun élément) ;
- qu'un polynôme non nul est représenté par une liste dont le dernier élément est non nul (**N.B.**) ;
- que les polynômes sont certes tous représentés par des listes, mais que les longueurs de ces listes, si elles sont finies, peuvent être quelconques.

Les trois points mentionnés ci-dessus induiront certaines « subtilités » de programmation (des distinctions de cas par exemple).

Convention : Afin d'alléger l'écriture, nous ne distinguerons pas un polynôme et la liste qui le représente en Python. Ainsi, parlerons-nous du polynôme `[1,1]` alors qu'il conviendrait, en toute rigueur, de parler de la liste `[1,1]` qui représente le polynôme $1+X$ en Python.

3 Valeurs versus adresses pour les listes

L'affectation de listes est effectuée par adresse (et non par valeur), comme l'illustre l'exemple suivant.

```
>>> A=[1,2,3]
>>> B=A # B et A pointent vers la même case mémoire
>>> A[0]=4 # on modifie la valeur de A
>>> print(B) # la valeur de B est alors également modifiée
[4, 2, 3]
```

Nous pouvons tout de même affecter des listes par valeur, en utilisant le slicing (ici la tranche prise est le tout).

```
>>> A=[1,2,3]
>>> B=A[:] # slicing
>>> A[0]=4
>>> print(B) # la valeur de B n'est pas modifiée
[1, 2, 3]
```

Rappel : Nous avions déjà rencontré ce type de phénomène avec le type `array` de la bibliothèque `numpy`. Nous avions alors utilisé la méthode `copy` pour contourner l'affectation par adresse.

4 Exercices

Exercice 1 (Degré d'un polynôme)
Écrire une fonction nommée `degre`,

- d'argument `A` un polynôme ;
- qui retourne le degré du polynôme `A` (pour le polynôme nul, la fonction retournera la chaîne de caractères `'-inf'`).

Voici quelques exemples d'appels de la fonction `degre`.

```
>>> degre([]) # degré du polynôme nul
'-inf'
>>> degre([2,1,-3,5,2]) # degré du polynôme 2+X-3*X^2+5*X^3+2*X^4
4
```

Exercice 2 (Coefficient dominant d'un polynôme non nul)
Écrire une fonction nommée `coeff_dom`,

- d'argument `A` un polynôme ;
- qui retourne :
  - le coefficient dominant du polynôme `A`, si `A` n'est pas le polynôme nul ;
  - la chaîne de caractères `'error'`, si `A` est le polynôme nul.

Voici quelques exemples d'appels de la fonction `coeff_dom`.

```
>>> coeff_dom([])
'error'
>>> coeff_dom([2,1,-3,5,-9]) # coefficient dominant du polynôme 2+X-3*X^2+5*X^3-9*X^4
-9
```

Exercice 3 (Polynôme non nul, qui est unitaire ou non)
Écrire une fonction nommée `unitaire`,

- d'argument `A` un polynôme ;
- qui retourne :
  - `True`, si `A` n'est pas le polynôme non nul, et est unitaire ;
  - `False`, si `A` n'est pas le polynôme non nul, et n'est pas unitaire ;
  - la chaîne de caractères `'error'`, si `A` est le polynôme nul .

Voici quelques exemples d'appels de la fonction `unitaire`.

```
>>> unitaire([])
'error'
>>> unitaire([1,2,-1,3]) # le polynôme 1+2*X-X^2+3*X^3 n'est pas unitaire
False
>>> unitaire([0,2,0,3,4,1]) # le polynôme 2*X+3*X^3+4*X^4+X^5 est unitaire
True
```

Exercice 4 (Somme de deux polynômes)
On se propose dans cet exercice de définir une fonction nommée `somme`,

- d'arguments deux polynômes `A` et `B`,
- qui retourne la somme des polynômes `A` et `B`.

Nous allons guidé quelque peu la construction d'une telle fonction.

♠ Nous pourrions naïvement penser à utiliser l'opérateur + qui est pré-défini sur les listes. Regardons un exemple.

```
>>> A=[1,2,3]
>>> B=[-3,4,5,7]
>>> A+B
[1, 2, 3, -3, 4, 5, 7]
```

Nous constatons que cela ne fonctionne pas. L'opérateur + permet de concaténer des listes. Il nous sera utile, mais il ne résout pas directement le problème posé.

♠ Nous pourrions aussi remettre en cause notre choix de modéliser des polynômes par des listes. En effet, le type `array` de la bibliothèque `numpy` possède un opérateur + qui permet d'additionner des vecteurs coefficient par coefficient. Voici un exemple.

```
>>> import numpy as np
>>> A=np.array([1,2,3])
>>> B=np.array([-1,4,7])
>>> A+B
array([ 0,  6, 10])
```

Il ne s'agit pourtant pas d'une bonne idée. En effet, nous pouvons être amenés à additionner des polynômes de degrés différents. Avec le type `array` de la bibliothèque `numpy`, nous aurons alors une erreur.

```
>>> import numpy as np
>>> A=np.array([1,2,3])
>>> B=np.array([-3,4,5,7])
>>> A+B
Traceback (most recent call last):
  File "<pyshell#76>", line 1, in <module>
    A+B
ValueError: operands could not be broadcast together with shapes (3,) (4,)
```

Nous ne pourrions ainsi additionner que des polynômes de même degré (car les vecteurs doivent avoir la même longueur, pour que l'on puisse les additionner coefficient par coefficient), ce qui est par trop restrictif. Nous conservons donc le type `list` pour modéliser les polynômes en Python.

♡ Essayons de comprendre comment on peut additionner deux polynômes A et B de degrés distincts. Si l'un ou l'autre est nul, alors il n'y a aucune difficulté à calculer la somme. Nous supposons donc que les polynômes A et B sont tout deux non nuls, pour expliquer la démarche.
Introduisons les coefficients $a_0, a_1, \dots, a_n$ de A et les coefficients $b_0, b_1, \dots, b_m$ de B, où $a_0 \in \mathbb{R}$, $a_1 \in \mathbb{R}$, $\dots$, $a_n \in \mathbb{R}^*$, $b_0 \in \mathbb{R}$, $b_1 \in \mathbb{R}$, $\dots$, $b_m \in \mathbb{R}^*$.

· 1er cas : $n = \deg(A) < \deg(B) = m$

La liste B se découpe alors naturellement en deux parties.

$$[a_0, a_1, \dots, a_n]$$
$$[\underbrace{b_0, b_1, \dots, b_n}_{\text{1ère partie}}, \underbrace{b_{n+1}, \dots, b_m}_{\text{2ème partie}}]$$

Nous souhaitons construire la liste :

$$[\underbrace{a_0 + b_0, a_1 + b_1, \dots, a_n + b_n}_{\text{1ère partie}}, \underbrace{b_{n+1}, \dots, b_m}_{\text{2ème partie}}].$$

Nous la construisons de proche en proche dans une variable notée `res`.
— Au début, nous initialisons `res` à la liste vide.
— Ensuite, nous parcourons chaque indice k de la première partie et nous ajoutons, à la liste `res` déjà construite, le terme $a_k + b_k$, au moyen de la méthode `append`.
— Après cette suite d'instructions, la valeur de `res` est :

$$[\underbrace{a_0 + b_0, a_1 + b_1, \dots, a_n + b_n}_{\text{1ère partie}}].$$

Il nous reste à lui concaténer la deuxième partie de B, i.e. la liste

$$[\underbrace{b_{n+1}, \dots, b_m}_{\text{2ème partie}}]$$

au moyen du slicing et de l'opérateur + sur les listes, pour avoir la liste souhaitée (i.e. la somme de A et B).

N.B. : Dans le cas considéré ici ($n = \deg(A) < \deg(B) = m$), le dernier terme de la liste `res` est b_m. Puisque b_m est non nul, cette liste représente bien un polynôme.

· $2^{\text{ème}}$ cas : $n = \deg(A) \geq \deg(B) = m$

Nous procédons comme dans le 1^{er} cas, en échangeant les rôles joués par A et B, pour construire la liste `res`.
N.B. : Dans le cas où $\deg(A) = \deg(B)$, un problème apparaît. Il se peut que la liste `res` se termine par des zéros (et donc ne représente pas un polynôme). Pensons, par exemple, au cas extrême ou A et B sont des polynômes non nuls, opposés. Dans ce cas, `res` ne contient que des 0 ! Il va donc nous falloir corriger cet éventuel défaut.

Après ces remarques liminaires, passons à l'implémentation.

1. Télécharger le fichier support du TP disponible à l'adresse suivante.

   `http://www.dblottiere.org/index.php?rep=ptsi/ipt`

2. Compléter les commentaires de la fonction `erase_zeros_end` (cf. balises `---COMPLETER---`).
   Indication : Lire l'intégralité du code et faire des tests, pour bien comprendre cette fonction.
3. Compléter la fonction `somme` en ajoutant un bloc d'instructions à l'endroit indiqué (en fin de code).
   Indication : Nous avons commencé à traduire, en langage Python, l'algorithme exposé précédemment (cf. paragraphe débutant par ♡).

Voici quelques exemples d'appels de la fonction `somme`.

```
>>> somme([],[1,2,3,4])
[1, 2, 3, 4]
>>> somme([1,2,3,4],[])
[1, 2, 3, 4]
>>> somme([1,3,4,5],[2,3,4])
[3, 6, 8, 5]
>>> somme([2,3,4],[1,3,4,5])
[3, 6, 8, 5]
>>> somme([1,1,1,1],[1,0,-1,-1])
[2, 1]
>>> somme([1,1,1],[-1,-1,-1])
[]
```

Exercice 5 (Multiplication d'un polynôme par un scalaire)
Écrire une fonction nommée `mult_scal`,

- d'arguments `k` un flottant et `A` un polynôme ;
- qui retourne le polynome obtenu en multipliant le polynôme `A` par le scalaire `k`.

Voici quelques exemples d'appels de la fonction `mult_scal`.

```
>>> mult_scal(-2,[1,2,0,13,1]);
[-2, -4, 0, -26, -2]
>>> mult_scal(0,[1,2,0,13,1]);
[]
>>> mult_scal(1,[]);
[]
>>> mult_scal(0,[]);
[]
```

Exercice 6 (Coefficient d'un produit de deux polynômes)
Écrire une fonction nommée `coeff_prod`,

- d'arguments deux polynômes non nuls `A` et `B`, et un entier `k` compris entre 0 et $\deg(A) + \deg(B)$ au sens large ;
- qui retourne le coefficient d'indice `k` du produit des deux polynômes `A` et `B`.

Voici quelques exemples d'appels de la fonction `coeff_prod`.

```
>>> coeff_prod([1,1],[1,1,1],3)
1
>>> coeff_prod([1,2,3],[1,-1,1,5],3)
4
>>> coeff_prod([1,2,3],[1,-1,1,5],5)
15
>>> coeff_prod([1,2,3],[1,-1,1,5],2)
2
```

Exercice 7 (Produit de deux polynômes)
Écrire une fonction nommée `produit`,

- d'arguments deux polynômes `A` et `B`,
- qui retourne le produit des deux polynômes `A` et `B`.

Indication : Dans le corps de cette fonction, la fonction `coeff_prod` *pourra être appelée, mais cette manière de procéder n'est pas imposée (être plus astucieux ?).*
Voici quelques exemples d'appels de la fonction `produit`.

```
>>> produit([],[1,1,1])
[]
>>> produit([1,1,1],[])
[]
>>> produit([1,1],[1,1])
[1, 2, 1]
>>> produit([1,2,0,1],[0,3,0,1,0,0,0,5])
[0, 3, 6, 1, 5, 0, 1, 5, 10, 0, 5]
```

Exercice 8 (Puissance d'un polynôme)
Écrire une fonction nommée `puissance`,

- d'arguments un polynôme `A` et un entier naturel `n`,
- qui retourne la puissance `n`−ième du polynôme `A`.

Indication : Dans le corps de cette fonction, la fonction `produit` *pourra être appelée.*
Voici quelques exemples d'appels de la fonction `puissance`.

```
>>> puissance([],0)
[1]
>>> puissance([],10)
[]
>>> puissance([1,1],5)
[1, 5, 10, 10, 5, 1]
>>> puissance([1,2,4,5],4)
[1, 8, 40, 148, 424, 992, 1910, 3032, 3976, 4180, 3400, 2000, 625]
```

Exercice 9 (Ligne du triangle de Pascal)
Écrire une fonction nommée `ligne_triangle_Pascal`,

- d'argument un entier naturel non nul `n`,
- qui retourne la $(n+1)$−ième ligne du triangle de Pascal, i.e. :

$$\left[\binom{n}{0},\binom{n}{1},\binom{n}{2},\ldots,\binom{n}{n}\right].$$

Indication : Il pourra être pertinent de, tout d'abord, transformer l'écriture du polynôme :

$$\binom{n}{0}+\binom{n}{1}X+\binom{n}{2}X^2+\ldots+\binom{n}{n}X^n$$

et la fonction `puissance` *pourra être appelée dans le corps de la fonction demandée.*
Voici quelques exemples d'appels de la fonction `ligne_triangle_Pascal`.

```
>>> ligne_triangle_Pascal(1)
[1, 1]
>>> ligne_triangle_Pascal(2)
[1, 2, 1]
>>> ligne_triangle_Pascal(3)
[1, 3, 3, 1]
>>> ligne_triangle_Pascal(6)
[1, 6, 15, 20, 15, 6, 1]
>>> ligne_triangle_Pascal(7)
[1, 7, 21, 35, 35, 21, 7, 1]
>>> ligne_triangle_Pascal(12)
[1, 12, 66, 220, 495, 792, 924, 792, 495, 220, 66, 12, 1]
>>> ligne_triangle_Pascal(13)
[1, 13, 78, 286, 715, 1287, 1716, 1716, 1287, 715, 286, 78, 13, 1]
```