TP 11 : les classes, attributs et méthodes en Python

Nous avons parlé des différents types des objets Python : nous avons vu les entiers `int`, les réels `float`, les listes `list`, les n-uplets `tuple`, les objets itérables de type `range`, les vecteurs de numpy (vérifiez à l'aide de l'opérateur `type` qu'ils sont de type `numpy.ndarray`), etc.

En fait **tous les objets Python sont des classes**. Les classes sont dotées de méthodes. Définissez une liste nommée `lis` dans votre interpréteur, puis tapez la commande `dir(lis)`. Cela affiche la liste des méthodes et attributs associés à l'objet `lis`. On y trouve un certain nombre de méthodes de la forme `__methode__`, et on reconnaît aussi les méthodes que nous connaissons : `reverse`, `remove`, `count`, etc.

Nous allons coder nous-même une nouvelle classe dotée de différentes méthodes afin de mieux comprendre ce que tout cela représente. Nous allons inventer un nouveau type de données en créant une classe pour représenter les **polynômes**.

Une nouvelle classe se définit un peu comme une fonction : cela commence par le mot-clé `class` suivi du nom souhaité, suivi de deux points, qui ouvrent un bloc dans lequel on définira tout ce qui concerne la classe en question.

```
class Polynom:

    print('Chargement de la classe polynome')
    a=2
    print(a)
```

Contrairement à une fonction, ce code est concrètement exécuté : les messages s'affichent à l'exécution de ce code. Mais en plus de cet affichage, ce code définit de manière abstraite la classe `Polynom`. Pour l'utiliser, nous allons définir un nouvel objet de type `Polynom`, tout simplement en tapant la commande `P=Polynom()`, qui créée un objet dont le nom est `P` et le type `Polynom`. Dans l'interpréteur tapez les commandes suivantes les unes après les autres :

```
>>> print(a)
>>> P=Polynom()
>>> print(a)
>>> type(P)
```

```
>>> print(P)
>>> P
>>> P+2
>>> 2*P
```

Remarquez que bien que le code de la classe ait bien été exécuté, la variable a est restée locale et n'est pas accessible en dehors de notre classe. Par contre le type `Polynom` a bien été créé, et l'on peut définir un nouveau polynôme en tapant `P=Polynom()`. Mais la classe est parfaitement vide (ou presque : nous verrons par la suite que nous avons tout de même fait quelque chose) : rien n'a été codé pour ces polynômes. Si l'on tente de faire des opérations sur `P`, on obtient des erreurs, et si veut accéder au contenu de `P`, on a seulement un message rappelant que c'est un objet de type polynôme.

Lorsqu'on créée un nouvel objet d'une classe donnée, ce que l'on vient de faire avec la commande `P=Polynom()`, on dit qu'on a créé une **instance** de la classe `Polynom`. C'est une **instanciation**. Sans instanciation, la classe reste un objet abstrait, un peu comme lorsqu'on définit une fonction à l'aide de `def f(x):` par exemple.

Attributs d'une classe

On peut commencer à donner plus de contenu à notre classe `Polynom`. Une classe a des **attributs**, des caractéristiques qui lui sont propres et qui lui sont automatiquement attachées dès leur création. Voici comment cela se présente

```
class Polynom:

    a = 2
    degre = "-oo"
```

Exécutez ce code puis dans l'interpréteur exécutez les commandes suivantes

```
>>> P = Polynom()
>>> P.a
>>> P.degre
```

On voit que la présence d'affectations de variables à l'intérieur du code de notre classe créée automatiquement des **attributs** aux objets de type `Polynom`. Ces attributs sont `a` et `degre`. Finalement dans la première version de notre classe, nous avions tout de même fait quelque chose ! L'attribut `a` ne correspondant à rien du tout, nous le supprimerons dans les versions ultérieures de `Polynom`. Pour accéder à un attribut d'un objet appartenant à une classe donnée, on écrit le nom de l'objet, suivi d'un point, suivi du nom de l'attribut en question.

Nous avons déjà fait cela plusieurs fois : par exemple avec une matrice `numpy`, nous avons utilisé plusieurs fois l'attribut `shape` de notre matrice ! Pour rappel voici un petit code qui créée une matrice A à l'aide de numpy, puis qui montre le contenu de l'attribut `shape` de la matrice A.

```
>>> import numpy as np
>>> A = np.array([[1, 2], [-4, 0.1], [-3.1, 2]])
>>> A.shape
```

Exercice 1 (Mon premier ajout d'attribut à une classe)**.**

Ajoutez à la classe `Polynom` l'attribut `coeffs` qui sera une liste vide.

Cette liste représentera dans toute la suite les coefficients du polynôme.
Si P est un objet de type `Polynom`, alors `P.coeffs` affichera la liste des coefficients du polynôme.

Méthodes standard des classes

Une **méthode** est tout simplement une **fonction Python définie à l'intérieur d'une classe**. Les méthodes dont le nom est de la forme `__nom__` sont des méthodes transversales qui sont partagées par plusieurs classes en même temps.

Deux méthodes sont quasi indispensables : la méthode `__str__` qui indiquera à Python le comportement qu'il devra adopter lors d'un appel de la fonction `print` appliquée à un objet de notre classe, et la méthode `__init__` qui donne une valeur par défaut aux nouveaux objets de notre classe créés. Voici comment elles fonctionnent :

```
class Polynom:

    degre = "-oo"
    coeffs = [ ]

    def __init__(self):
        self.coeffs = [0]
        self.degre = -1

    def __str__(self):
        message = "C'est un polynome"
        return message
```

Exécutez ce code puis dans l'interpréteur, définissez à nouveau un polynôme P à l'aide de `P=Polynom()`. Exécutez maintenant `print(P)` puis tout simplement la commande `P`. La première affiche le message "C'est un polynôme". On pourra la modifier plus tard lorsque nous aurons un peu complexifié notre classe. Lorsque l'on **instancie** notre classe, le contenu de la fonction `__init__` (qui vient bien sûr du mot « initialisation ») est exécuté. Ce qui signifie que tout nouvel objet de type `Polynom` sera automatiquement affecté d'une liste de coefficient ne contenant qu'un seul zéro, et d'un degré qui est l'entier 0. Vérifiez cela en exécutant `P.degre` et `P.coeffs` dans l'interpréteur.

Cela rend inutile les deux premières lignes de notre code : sans instanciation de la classe (création d'un objet `Polynom`) la classe ne sera pas utilisée, alors que si instanciation il y a, le degré sera automatiquement modifié, passant de `"-oo"` à -1, et la liste des coefficients passera tout de suite de `[ ]` à `[0]`. Nous supprimerons donc ces deux premières lignes dans la suite.

Définir de nouvelles méthodes

Dans toutes les méthodes, apparaît toujours un premier argument `self` : il fait référence à l'objet de type `Polynom` que nous sommes en train de manipuler. Voici un exemple de deux nouvelles méthode, une ayant un argument supplémentaire, l'autre non :

```
class Polynom:

    def __init__(self):
        self.coeffs = [0]
        self.degre = -1

    def __str__(self):
        message="C'est un polynome"
        return message

    def coeff_dom(self):
        return self.coeffs[-1]

    def coeff(self,k):
        return self.coeffs[k]
```

Comme vous pouvez vous en douter, la première méthode renvoie le coefficient dominant, la deuxième permet d'accéder directement à l'élément numéro k de la liste des coefficients.

Testez les en exécutant dans l'interpréteur (après avoir créé une instance nommée `P` de la classe `Polynom` bien sûr) la commande `P.coeff_dom()`, puis la commande `P.coeff(0)`.

L'utilisation des méthodes se fait presque comme celle des attributs, sauf qu'on utilise des parenthèses : elles se comportent comme des fonctions. Par contre l'argument `self`, qui représente le nom de l'instance de classe manipulée, n'est pas donné en argument lorsqu'on appelle la méthode : il est déjà présent **avant le `"."`**, par exemple dans `P.coeff(0)`, l'argument `self` de notre code est remplacé par `P` et l'argument `k` est remplacé par `1`.

Exercice 2.

1. Créez une instance Q de la classe `Polynom`.
   Vérifiez que la commande `Q.coeff(1)` renvoie une erreur ? Pourquoi ?
2. Modifiez votre code pour éviter cette erreur.
3. Modifiez la méthode `__str__` pour qu'elle affiche la liste des coefficients du polynôme au lieu d'un message.

Amélioration de la méthode `__init__`

On va permettre à l'utilisateur de définir des polynômes de son choix. Pour cela, il faut que l'utilisateur communique, au moment de l'utilisation de `Polynom()`, la liste des coefficients qu'il souhaite. Nous sommes libres de le faire de la manière que nous souhaitons. Nous allons utiliser pour cela une liste Python, que l'on ajoute dans la liste des arguments de la méthode `__init__`. Cette méthode devient donc (le reste de la classe est inchangé)

```
def __init__(self, liste_coeffs):
    self.coeffs = liste_coeffs
    self.degre = 0
```

Dorénavant, pour créer un nouveau polynôme, il faudra utiliser `Polynom(uneliste)`, où `uneliste` doit être une liste Python. Par exemple on peut définir deux nouveaux polynômes avec `P=Polynom([1,-2,-5])` ou bien en exécutant `li=[0,1,2]` puis `Q=Polynom(li)`. Testez ces commandes. Quel sont alors les valeurs des attributs `degre` des deux polynômes P et Q ? P.degre = Q.degre = 0

Exercice 3.

Modifiez `__init__` afin que le degré corresponde bien à la liste des coefficients donnée par l'utilisateur.

On prendra garde au cas où l'utilisateur aurait donné une liste contenant des zéros à la fin de la liste. Pour cela : parcourir la liste en partant de la fin jusqu'à rencontrer un coefficient non nul. On affectera le degré -1 au polynôme nul, et on lui affectera une liste de coefficients vide.

Surcharge d'opérateurs

On aimerait maintenant pouvoir additionner deux polynômes. Pour cela on fait appel à la méthode standard `__add__`, qui permet une fois qu'on l'a correctement définie dans notre classe, d'utiliser l'opération `+` entre deux polynômes !

Voici un exemple de méthode d'addition, adaptée seulement au cas où les deux polynômes ont exactement le même degré. L'argument `other` désigne un deuxième objet de type `Polynom` :

```
def __add__(self, other):
    a = self.coeffs                          # les coeffs du premier polynome
    b = other.coeffs                         # ceux du deuxieme
    n = self.degre                           # le degre commun
    c = [a[k] + b[k] for k in range(n)]       # liste contenant les sommes
    resultat = Polynom(c)                    # le polynome que l'on va renvoyer
    return resultat
```

Nous pouvons utiliser cette nouvelle méthode pour additionner deux polynômes en utilisant `+` **comme d'habitude** : c'est un exemple de **surcharge d'opérateurs**. L'addition ne fonctionnait pas pour deux polynômes avant cela, elle fonctionne désormais.

Exercice 4 (Amélioration de `__add__`)**.**

1. Notre fonction `__add__` contient un bug. Détectez le en faisant des tests et corrigez-le.
2. Modifiez la méthode d'addition pour que les polynômes de degrés différents soient additionnés correctement : il s'agit d'ajouter des zéros au bout de la liste de coefficients la plus courte.

Exercice 5.

1. Créez une méthode `evaluation` qui évalue le polynôme en un réel x passé en argument.
2. En dehors de la classe `Polynom`, tracez la courbe représentative de $1-3X+X^3+5X^4$ sur l'intervalle $[-1;1]$ à l'aide de `matplotlib`.