Mathématiques spéciales le 23 Novembre 2018

TP n°4 - La Programmation Orientée Objet

1. Introduction

La Programmation Orientée Objet (POO - OOP en anglais) peut se définir comme une "philosophie" particulière de programmation. Le principe de cette philosophie est de définir des structures de données particulières regroupées en **classes** et appelées **objets** ! Un objet est un élément, une **instance** d'une classe. Ok, mais qu'est-ce que ça veut dire ?

— Une **classe** contient toutes les spécifications que partageront les objets qui lui appartiennent i.e. qui sont des instances de cette classe ;

— Un **objet** est une structure de données contenant plusieurs champs spécifiés par la classe qui le contient :

- Les **attributs** de l'objet qui permettent de définir l'objet et ce qu'il contient.
- Les **méthodes** associées à l'objet qui sont des fonctions qui vont agir sur l'objet lui-même.

Dans l'esprit de la POO, un objet représente un concept concret : un compte bancaire, un client, une voiture, ou en maths, un vecteur, un complexe (exemples qu'on va étudier dans la suite). Par exemple, on pourrait créer une classe `CompteBancaire` dont les objets serviraient à représenter les comptes bancaires des clients d'une banque :

— Les attributs spécifiés pour cette classe seraient par exemple des champs :

- `nom` qui contiendrait le nom du possesseur du compte ;
- `coordonnees` qui contiendrait les coordonnées du possesseur du compte ;
- `operation` qui contiendrait une liste des opérations sur le compte.

— Les méthodes spécifiés pour cette classe seraient par exemple des fonctions :

- `crédit` qui calculerait les entrées d'argent du compte (à partir de l'attribut `operation`) ;
- `débit` qui calculerait les sorties d'argent du compte (à partir de l'attribut `operation`) ;
- `operation` qui calculerait le solde du compte.

Il faut garder en tête que la POO n'est qu'une philosophie de programmation ; on pourrait très bien se passer du concept de classe et d'objet pour créer un programme. L'avantage de la POO est que la structure des programmes contenus dans un projet est bien mieux maîtrisée : par exemple, sans la POO, pour gérer nos comptes bancaires, on aurait simplement créé des tableaux contenant à la suite les attributs précisés plus hauts et créé des fonctions qui agissent sur ces tableaux ; cela manque de visibilité : comment se rappeler que le premier emplacement du tableau contient le nom du client ? Comment se rappeler qu'il faut appliquer telle ou telle fonction à telle emplacement du tableau.
Comme on va le voir avec Python, la manipulation des objets et la clarté qui découle de leur structure font toute la force de la POO par rapport à une programmation "à l'ancienne". D'ailleurs, en Python, on connaît déjà un exemple de classe : la classe **`list`** ! Et on a déjà joué avec les méthodes de cette classe : les méthodes `append`, `remove`, etc... En fait, il n'y a bien-sûr pas que la classe **`list`** en Python, loin de là : en Python, "tout est objet" ! C'est la philosophie du langage Python, et Python a sa propre façon de traiter les classes et les objets. Chaque langage qui utilise la POO l'utilise à sa manière. La manière dont Python intègre les objets n'est pas la même que le Java ou le C++ par exemple. Ce qui faut retenir, c'est le concept ! Et ce concept de POO, nous allons l'aborder dans ce T.P. grâce à Python.

Passons directement à la pratique :

2. Construction de la classe `Complexe`

Nous allons dans la suite construire une classe qui va nous permettre de définir et de manipuler des nombres complexes. Chaque objet créé représentera un nombre complexe.

a. Constructeur et Afficheur

Ci-dessous, voici la syntaxe de base qui permet de créer une classe, ici la classe `Complexe` :

```
1  class Complexe: #Définition de la classe Complexe
2      """ Classe définissant un nombre complexe par sa partie réelle et sa partie
           imaginaire """
3
4      # Constructeur
5      def __init__(self,x,y):
6          self.re=x #attribut partie réelle
7          self.im=y #attribut partie imaginaire
8
9      # Afficheur
10     def __repr__(self):
11         return str(self.re)+'+'+str(self.im)+'i'
```

Essayons de comprendre les différentes parties du bloc `class Complexe:` :

— Premièrement, on définit le nom de la classe juste après le mot clef **`class`**. Ce nom nous permettra de créer les objets de cette classe.

— Ensuite, le **constructeur** permet de définir les attributs qui caractérisent les objets de la classe. Le constructeur est toujours appelé comme une définition de fonction qui a pour nom `__init__` (Attention, il s'agit de part et d'autre de `init` d'un **double underscore**).
Passons en revue les arguments du constructeur :

- Le premier argument du constructeur désigne toujours l'objet *lui-même*. C'est un argument obligatoire, on ne l'omettra jamais : c'est la syntaxe de Python qui veut ça. Il s'agit bien d'un argument, on pourrait lui donner le nom que l'on veut pourvu qu'on utilise le même dans le corps du constructeur. **Mais**, pour éviter les confusions et pour faciliter la lecture et le relecture de la classe, on utilisera **toujours** le nom `self` pour désigner l'objet dans la définition d'une classe.
- Les arguments suivants sont facultatifs : ce sont les attributs que l'on désire attacher aux objets de la classe et qui les caractériseront. Dans notre exemple `x` est la variable contenant la partie réelle du nombre complexe, et `y` est la variable contenant la partie imaginaire du nombre complexe.

Dans le corps du constructeur, on définit chaque attribut grâce à la syntaxe `self.NomDelAttribut = AttributEnArgument` où `self` désigne l'objet.

— Pour finir, on doit définir l'affichage de notre objet grâce à la méthode spéciale (au même titre que `__init__`) appelée **afficheur** et dont la syntaxe en Python est `__repr__` - comme représentation. L'afficheur va nous permettre, comme son nom l'indique, de définir comment sera affiché l'objet quand on l'appellera. Par exemple, si on crée une liste `L=[1,2,3]` et que l'on appelle `L` dans le shell, un joli `[1,2,3]` apparaît à la ligne en dessous : c'est l'afficheur de la classe **`list`** qui produit cela. L'afficheur ne demande qu'un seul argument obligatoire, comme pour le constructeur, l'objet lui-même (donc `self`)

Ainsi, l'afficheur renvoie toujours une chaîne de caractères. Il faut donc, après le **`return`**, indiquer une chaîne de caractères qui représentera l'objet quand il sera appelé.

Essayons de comprendre ce qu'il en est dans notre exemple. Si je crée un objet de la classe `Complexe`

de partie réelle 1 et de partie imaginaire 2, j'aimerais que lors de l'appel de cet objet pour voir ce qu'il contient, s'affiche 1+2i.

Pour l'objet self, on récupère l'attribut qui contient la partie réelle grâce à l'appel self.re et de même pour la partie imaginaire avec self.im. Le souci est que ces attributs sont des nombres et pas des chaînes de caractères : il faut ainsi les convertir en chaîne de caractères grâce à la fonction **str**. Ainsi, en concaténant les chaînes - grâce à l'opérateur de concaténation des chaînes de caractères + :

```
str(self.re)    '+'    str(self.im)    'i'
```

on obtiendra bien la représentation voulue lors de l'affichage.

Copions ce code dans l'éditeur puis exécutons-le. Nous venons de créer la classe Complexe. Alors maintenant, comment créer un objet de cette classe ??? C'est très simple : on précise le même nombre d'arguments "attributs" que celui indiqué dans le constructeur (sans compter le self, bien-sûr).

```
z=Complexe(1,2) # création d'un objet de la classe Complexe de partie réelle 1 et de
    partie imaginaire 2

z # affichage de l'objet
```

Pour renvoyer un attribut d'un objet on utilise la syntaxe objet.NomDelAttribut. Par exemple :

```
z.re # renvoie l'attribut re de l'objet Complexe z (qui contient la partie réelle)

z.im # renvoie l'attribut im de l'objet Complexe z
```

Exercice 1.

Créer d'autres objets de la classe Complexe et faites quelques tests !

Exercice 2.

Améliorer l'affichage des complexes dont la partie réelle et/ou imaginaire est nulle, et également dans le cas où la partie imaginaire est négative ou égale à ± 1.
Indice : On peut bien-sûr écrire des choses entre le **def** __repr__(self): et le **return** ... !

b. Méthodes

Maintenant que l'on a vu comment définir une classe, voyons comment lui attacher des méthodes i.e. des fonctions qui vont agir sur nos objets et qui sont propres à la classe. Commençons par la syntaxe : on écrit nos méthodes comme on définit une fonction ; la différence est qu'on indique toujours en premier argument le self qui désigne notre objet. On écrit bien-sûr nos méthodes à la suite des blocs précédents dans **class** Complexe: ; on garde donc la même indentation.

```
class Complexe: #Définition de la classe Complexe
    """ Classe définissant un nombre complexe par sa partie réelle et sa partie
        imaginaire """
```

```
3
4     # Constructeur
5     ...
6
7     # Afficheur
8     ...
9
10    # Méthodes
11
12        def module(self):
13            return (self.re**2+self.im**2)**(1/2)
```

Remarque.

Attention ! Quand vous modifiez votre classe, il faut la recharger entièrement (avec CTRL+E par exemple) et pas seulement la dernière partie modifiée ! Il faut également recharger les objets précédemment créés, sinon ils ne tiendront pas compte des méthodes nouvellement définies.

Testons cette méthode sur un objet :

```
1 z=Complexe(4,3) # création du complexe z=4+3i
2
3 z.module() # calcul du module de z
```

On peut également ajouter des arguments à nos méthodes :

```
1 #Toujours à la suite de la classe Complexe
2
3 def addition(self, z): #addition de l'objet lui-même et d'un autre objet de la classe
      Complexe
4     if type(z)==Complexe: #on vérifie que l'argument z est bien dans la classe Complexe
          pour éviter les erreurs.
5         x=self.re+z.re #addition des parties réelles
6         y=self.im+z.im #addition des parties imaginaires
7         return Complexe(x,y)
```

Testons cette méthode sur un objet :

```
1 z=Complexe(4,3) # création du complexe z=4+3i
2 t=Complexe(1,1) # création du complexe z=1+i
3
4 z.addition(t) # renvoie le complexe z+t=5+4i
```

Bon ce n'est pas très pratique et on aimerait simplement écrire `z+t` pour obtenir le résultat... mais on verra ça dans la partie suivante ! Tout d'abord, quelques exercices !

Exercice 3.

Écrire les méthodes suivantes pour la classe `Complexe` :

1. `argument()` qui renvoie l'argument du complexe (dans $[0, 2\pi[$). Attention au cas de 0 !
2. `oppose()` qui renvoie le complexe opposé du complexe lui-même ;
3. `conjugue()` qui renvoie le complexe conjugué du complexe lui-même ;
4. `inverse()` qui renvoie le complexe inverse du complexe lui-même ;
5. `produit(z)` qui renvoie le complexe produit du complexe lui-même et de `z` ;
6. `difference(z)` qui renvoie le complexe différence du complexe lui-même et de `z`
7. `quotient(z)` qui renvoie le complexe quotient du complexe lui-même et de `z` ;
8. `trigo()` qui affiche la forme trigonométrique du nombre complexe i.e. $r(sin(\theta) + cos(\theta))$
9. `expo()` qui affiche la forme exponentielle du nombre complexe i.e. $re^{i\theta}$.

Exercice 4.

Améliorer l'affichage afin que lorsque la partie réelle ou imaginaire du complexe est un entier, elle s'affiche comme un entier i.e. on veut `1+2i` et pas `1.0+2.0i` (ce qui peut se produire après un calcul).

On pourra se servir des fonctions suivantes :

```
def isclose(a, b):
    return abs(a-b) <= 10**(-9)*max(abs(a), abs(b))

def intifint(x):
    if isclose(x,round(x)):
        return round(x)
    else:
        return x
```

D'ailleurs, que font-elles ?

3. Construction de la classe `Vecteur`

On cherche à créer une classe `Vecteur` pour représenter les vecteurs de $\mathbb{R}^3$ avec des méthodes permettant de faire des opérations usuelles sur ces ceux-ci. On va voir dans cette partie comment **surcharger des opérateurs** c'est-à-dire comment définir un comportement pour des opérateurs "simples" de Python, comme `+` ou `*` afin d'alléger la syntaxe des opérations entre objets d'une classe.

a. Création de la classe et de ses méthodes

Exercice 5.

1. Créer une nouvelle classe `Vecteur` dont les attributs s'appellent `x`, `y` et `z` et qui contiendront chacune des coordonnées représentant un vecteur de $\mathbb{R}^3$.
   L'afficheur doit permettre de renvoyer un vecteur sous la forme `(1,2,3)`
2. Écrire les méthodes suivantes :
   — `norme()` qui renvoie la norme du vecteur lui-même ;
   — `unitaire()` qui renvoie le vecteur unitaire de même sens que le vecteur lui-même ;
   — `det(v,w)` qui renvoie le déterminant unitaire de la famille formée du vecteur lui-même, du vecteur `v` et du vecteur `w` ;
   — `decomposition(u,v,w)` qui affiche la décomposition du vecteur lui-même dans la base de vecteurs (u, v, w) (faire une vérification préalable sur le fait que (u, v, w) est bien une base !) .

b. Surcharge des méthodes

Comme on l'a dit dans l'introduction de cette partie, on aimerait utiliser l'opérateur `+` pour ajouter des objets de la classe `Vecteur` pour alléger la syntaxe de l'addition. Et bien, rien de plus simple : en Python, l'opérateur `+` est désigné par la méthode spéciale `__add__`. Il suffit donc de redéfinir (surcharger) cette méthode dans notre classe `Vecteur` pour pouvoir lui donner le comportement que l'on veut ! Voici comment faire pour configurer l'opérateur `+` comme étant l'addition de deux objets de la classe `Vecteur`
On écrit bien-sûr à la suite des blocs de la classe :

```
class Vecteur:

    # Constructeur
    ...

    # Afficheur
    ...

    # Méthodes
    ...

    #Surcharge des méthodes

    def __add__(self,v): #addition
        if type(v)==Vecteur:
            return Vecteur(self.x+v.x,self.y+v.y,self.z+v.z) #on additionne coordonnée
                par coordonnée.
```

On exécute le tout et on teste le résultat !

```
v=Vecteur(1,2,3)
w=Vecteur(3,4,5)
v+w
```

Donc, si on veut surcharger une méthode spéciale, il faut connaître sa désignation en Python ! En voici une liste non exhaustive :

Opérateurs mathématiques :

- `__add__` représente l'opérateur `+`
- `__mul__` représente l'opérateur `*`
- `__sub__` représente l'opérateur `-`
- `__truediv__` représente l'opérateur `/`
- `__floordiv__` représente l'opérateur `//`
- `__mod__` représente l'opérateur `%`
- `__pow__` représente l'opérateur `**`

Opérateurs de comparaison : Attention, ils doivent renvoyer un booléen !

- `__eq__` représente l'opérateur `==`
- `__ne__` représente l'opérateur =!
- `__gt__` représente l'opérateur `>`
- `__ge__` représente l'opérateur `>=`
- `__lt__` représente l'opérateur `<`
- `__le__` représente l'opérateur `<=`

Il existe bien d'autres méthodes spéciales que nous n'aborderons pas ici.

Exercice 6.

1. Surcharger l'opérateur `-` pour la différence de vecteurs ;
2. Surcharger l'opérateur `*` pour le produit vectoriel de vecteurs ;
3. Surcharger les opérateurs `==` et =! pour tester l'égalité ou la non égalité de vecteurs ;
4. Surcharger les opérateurs d'ordre pour tester l'ordre lexicographique de vecteurs ;

Exercice 7.

1. Surcharger les méthodes adéquates pour simplifier la syntaxe de l'addition, différence, multiplication et division de la classe `Complexe`.
2. Surcharger l'opérateur `**` pour calculer des puissances entières d'objets de la classe `Complexe`.

4. Construction de la classe `Polynome`

On cherche ici à créer une classe `Polynome` pour manipuler et afficher des polynômes à coefficients réels.

a. Création de la classe et de ses méthodes

Exercice 8.

1. Créer une nouvelle classe `Polynome` avec un seul attribut `coefficients` qui contient la liste `L` des coefficients du polynôme (dans l'ordre croissant des degrés des monômes).
   L'afficheur doit permettre de renvoyer un polynôme `P=Polynome([1,2,3,0,5])` sous la

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

2. Écrire les méthodes suivantes :
   — `deg()` qui renvoie le degré du polynôme. On conviendra que le degré de 0 est $-\infty$: on peut importer la "constante" `inf` du module `math` qui représente $+\infty$ - et dans ce cas, `-inf` représente $-\infty$.
   Attention s'il y a des 0 *à la fin de la liste !*
   — `suppr_zero()` qui débarrasse la liste des coefficients d'un polynôme des 0 inutiles en fin de liste.
   — `derivee(k)` qui renvoie le polynôme k-ième dérivé du polynôme lui-même. Quel sera alors le résultat renvoyé par l'instruction `P.derivee(P.deg())` (avec $P \neq 0$) ?

Exercice 9.

1. Surcharger les méthodes adéquates pour multiplier, additionner, soustraire, comparer deux polynômes (on définira un ordre partiel grâce au degré). Pour l'addition et la soustraction, ne pas oublier la suppression des 0 inutiles !
2. Améliorer l'addition et la multiplication afin de pouvoir ajouter et multiplier un polynôme et un scalaire.
   Remarque : On remarquera que les définitions de `__mul__` et `__add__` ne sont pas commutatives : il faut surcharger les méthodes spéciales `__rmul__` et `__radd__` pour pouvoir gérer l'ajout et le produit d'un scalaire et d'un polynôme.
   Autre précision, pour pouvoir utiliser les opérateurs du type `+=` ou `*=`, on surchargera la méthode spéciale correspondant à l'opération précédée par un `i` ; par exemple, `__iadd__` pour l'opérateur `+=`.
3. Surcharger les opérateurs `//` et `%` pour renvoyer le quotient et le reste (respectivement) de la division euclidienne de deux polynômes.
4. Surcharger l'opérateur `^` (méthode spéciale `__xor__`) pour renvoyer le PGCD (unitaire) de deux polynômes - grâce à l'algorithme d'Euclide.

b. Notion d'héritage

On aimerait désormais créer une classe dédiée aux polynômes du second degré afin d'implémenter le calcul du discriminant par exemple, tout en gardant les méthodes que l'on a déjà créées pour la classe `Polynome`. Il serait dommage de devoir copier/coller la première classe et de rajouter nos méthodes supplémentaires pour arriver à nos fins ! C'est ici qu'intervient la notion **d'héritage**. De quoi s'agit-il ? Et bien, étant donnée une classe Python, on peut créer une *sous-classe* qui :

— possède les mêmes méthodes que la "sur"-classe ;
— bénéficie d'un constructeur allégé et de méthodes spécifiques aux objets de cette sous-classe.

La syntaxe à adopter pour une sous-classe est la suivante : `class SousClasse(SurClasse)`. Puis on surcharge le constructeur afin d'obtenir un format spécifique pour les objets de la sous-classe. On peut bien-sûr surcharger l'afficheur ou tout autre méthode déjà définie dans la sur-classe si la sous-classe nécessite une adaptation. Voyons cela sur notre exemple concret de la sous-classe `Polydeg2` de la classe `Polynome` :

```python
class Polynome:
    #constructeur
    #afficheur
    #méthodes
    #surcharge de méthodes

class Polydeg2(Polynome):

    #surcharge du constructeur
    def __init__(self,c,b,a):
        self.coefficients = [c,b,a]

    #surcharge de l'afficheur si nécessaire (ici, non)

    #méthodes spécifiques
```

Ensuite, on définit les méthodes spécifiques à la sous-classe de la même façon que pour une classe quelconque.

Exercice 10.

Créer les méthodes spécifiques suivantes pour la sous-classe `Polydeg2` :

1. la méthode `discriminant()` qui renvoie le discriminant du polynôme lui-même ;
2. la méthode `racines()` qui renvoie une liste `[r1,r2]` contenant les racines quand elles existent (et deux fois la même s'il n'y en a qu'une)
3. la méthode `extremum()` qui renvoie une liste `[x,y]` où `y` est l'extremum global fini du polynôme lui-même (seulement dans le cas où il est de degré exactement 2) et `x` est l'abscisse de cet extremum.
4. la méthode `factorisation()` qui affiche le polynôme lui-même sous forme d'un produit d'un scalaire et de facteurs irréductibles unitaires.

5. Construction de la classe `Pile`

Dans cette partie, nous allons créer une classe `Pile` afin d'implémenter en Python une structure de pile. Nous nous serviront des listes Python comme contenant et nous partirons du principe que lors de la création d'un objet de la classe `Pile`, celui-ci est vide (plus précisément une liste vide) et qu'on pourra le modifier seulement grâce à des méthodes basées sur l'empilage et le dépilage.
On rappelle que la seule façon de "voir" une élément de la pile est de le placer en dernière position et de le dépiler (le supprimer).

Voici donc le code de base de la classe `Pile` :

```python
class Pile:

    ##constructeur
    def __init__(self):
        self.assiettes=[] #la pile est initialisée à la liste vide

```

```
7     ##afficheur
8
9     ##méthodes
```

a. Création de la classe et de ses méthodes

Exercice 11. *Création des méthodes avancées de la classe Pile*

1. Écrire la méthode `vide()` qui retourne `True` si la pile elle-même est vide (i.e. égale à la liste vide), `False` sinon.
2. Écrire la méthode `empile(x)` qui ajoute l'élément `x` en haut de la pile (grâce à la méthode `append(x)`)
3. Écrire la méthode `depile()` qui retourne et supprime l'élément en haut de la pile (grâce à la méthode `pop()`). On ne peut pas dépiler une piste vide. Afin d'afficher une erreur du même type que celles que renvoie Python, on utilisera la syntaxe suivante :

```
1 raise ValueError("cause de l'erreur")
```

Par exemple, dans notre cas, on utilisera :

```
1     #dépilage
2     def depile(self):
3         if self.vide():
4             raise ValueError("Erreur : impossible de dépiler, cette pile
                  est vide") #renvoie une erreur si la pile est vide
5     #A FINIR
```

Dans l'exercice suivant, on utilisera seulement les méthodes `empile`, `depile` et les méthodes créées dans l'exercice lui-même :

Exercice 12. *Création des méthodes de base de la classe Pile*

1. Écrire la méthode `peek()` qui retourne l'élément en haut de la pile sans modifier la pile elle-même.
2. Écrire la méthode `empiliste(L)` qui empile la liste sur la pile elle-même (en commençant par l'élément d'indice 0 puis 1 etc... le dernier élément de `L` se retrouvant en haut de la pile).
3. Écrire la méthode `flip()` inverse les éléments de la pile elle-même (haut en bas et bas en haut) sans modifier la pile elle-même. Quelle est la complexité de cette méthode (en terme d'opérations élémentaires d'empilation et dépilation) ?
4. Écrire la méthode `copie()` qui retourne une copie de la pile elle-même, sans la modifier. Quelle est la complexité de cette méthode ?

5. Définir l'afficheur afin qu'il permette d'afficher, sans la modifier, une pile contenant, par exemple,

   (de haut en bas) 5, 4, 3, 2, 1, 0

   de la façon suivante :

```
>>>P=Pile()
>>>for i in range(6):
...   P.empile(i)
...
...
>>>P
5||--4||--3||--2||--1||--0||>
```

b. Les méthodes de conteneur

Une pile (i.e. un objet de la classe `Pile` précédente) étant donné, on souhaiterait pouvoir accéder à l'élément d'indice i - l'élément d'indice 0 étant étant le plus bas dans la pile - en utilisant la même syntaxe simple que pour la classe **`list`** i.e. pour `P=empiliste(['a','b','c','d'])`, on voudrait pouvoir accéder à l'élément `'c'` de la pile avec la syntaxe `P[2]`. De la même façon, on souhaiterait pouvoir modifier ou supprimer des éléments grâce aux syntaxes `P[i]=x` et **`del`** `P[i]`.

Pour cela, on surcharge les **méthodes de conteneur** `__getitem__` pour l'accès ; `__setitem__` pour la modification et `__delitem__` pour la suppression. Voyons la syntaxe à utiliser dans un cas général :

```
1  class MaClasse:
2      #constructeur, afficheur, méthodes
3
4      #surcharge des méthodes
5
6      def __getitem__(self, i):
7          """ Permet de définir l'action souhaitée pour l'instruction self[i] """
8
9      def __setitem__(self, i, x):
10         """ Permet de définir l'action souhaitée pour l'instruction self[i]=x """
11
12     def __delitem__(self, i):
13         """ Permet de définir l'action souhaitée pour l'instruction del self[i] """
```

Exercice 13.

Surcharger les méthodes de conteneur pour la classe `Pile` afin qu'on puisse accéder, modifier et supprimer les éléments d'une pile grâce à leur indice. On n'utilisera bien-sûr que les méthodes `empile` et `depile`.
On pourra se servir de la gestion des erreurs **`raise`** `ValueError('erreur')` en cas de problème d'indice par rapport à la taille de la pile.

Quelle est la complexité de l'accès `P[n]` en terme d'empilages et dépilages ?

Exercice 14.

On souhaiterait ajouter à la classe `Polynome` une méthode d'évaluation d'un polynôme P en un nombre t avec la syntaxe simple `P[t]` où `P` est un objet de la classe `Polynome` et `t` est un nombre entier ou flottant.

1. Surcharger la méthode `__getitem__` pour obtenir l'évaluation $P(t) = \sum_{k=0}^{\deg(P)} a_k t^k$.
2. Surcharger l'opérateur ** pour obtenir le produit scalaire suivant, pour $P, Q \in \mathbb{R}[X]$:

$$(P|Q) = \sum_{k=0}^{\infty} P^{(k)}(k) Q^{(k)}(k)$$

D'ailleurs, est-ce bien un produit scalaire sur $\mathbb{R}[X]$?