

# Exercices de programmation

- [Retour à la page principale](#)

## Liste des exercices

- [Programmation Modulaire](#)
- [Programmation Objet](#)
- [Programmation Fonctionnelle](#)
- [Listes](#)
- [Files et piles](#)
- [Arbres binaires](#)
- [Graphes](#)

# Exercices sur la programmation modulaire

- [Retour à l'ensemble des exercices](#)
- [Accès aux solutions](#)

## Une application monolithique

Supposons partir du [code suivant](#) qui permet de manipuler des suites de nombres. Ce code est complètement monolithique, et donc difficile à faire évoluer. Pire, les fonctions à l'intérieur sont mises à plat, mélangées, et contiennent des morceaux de code en commentaire. Enfin, les standards de codage sont faibles, il manque des indications de types à plusieurs endroits.

Proposer une décomposition du code en plusieurs fichiers Python indépendants. On s'assurera des choses suivantes :

- chaque fichier est du code Python valide, ce qui signifie que l'on peut les exécuter avec `python` sans erreur (même s'ils ne font aucun calcul)
- il est possible d'exécuter les tests, ou de choisir un affichage (sans les autres), et cela de manière indépendante.

Remarque : le code utilise la bibliothèque `matplotlib` pour dessiner les suites. Si jamais cela posait difficulté d'installation, il est possible de faire l'exercice en supprimant l'ensemble des fonctions commençant par `code_` et le chargement de la bibliothèque `matplotlib`.

# Programmation modulaire

- [Retour aux exercices de programmation modulaire](#)

## Décomposition proposée

- [sequences.py](#)
- [test\\_sequences.py](#)
- [plot\\_sequences.py](#)
- sans oublier les fichiers de scripts : [script\\_plot\\_all\\_sequences.py](#), [script\\_plot\\_a\\_sequence.py](#), [script\\_plot\\_fibo\\_exponential.py](#) et [script\\_plot\\_fibo\\_linear.py](#)

# Exercices sur les objet

## [Retour à l'ensemble des exercices](#)

L'objectif est de concevoir des applications en commençant par identifier les différents objets qui la composent, puis en programmant leurs classes.

### 1 - Le blackjack

L'application que nous souhaitons développer est celle d'une table de blackjack. Nous considérons des règles très simples :

- Un donneur distribue des cartes (un paquet de 52 cartes sans les jokers) aux joueurs, à la hauteur d'une carte par joueur et par tour.
- Le donneur incarne un joueur particulier : la banque.
- A la fin d'un tour, si la somme des valeurs des cartes d'un joueur dépasse 21, le joueur est éliminé (on considère que les têtes et l'as valent 10 points, les autres valeurs correspondent à la hauteur de la carte).
- À la fin d'un tour, si un joueur n'est pas éliminé, il peut décider de ne plus recevoir de carte. Le donneur ne lui en donnera pas le tour suivant.
- Si la banque perd, tous les joueurs non éliminés gagnent (on comptera un point par gain).
- S'il ne reste plus de joueur engagé dans la partie (soit ils sont éliminés soit ils ont décidé de ne plus recevoir de carte), alors le joueur dont la somme des valeurs de ses cartes est la plus proche de 21 gagne.

On imagine, qu'il est possible de jouer à plusieurs joueurs (5 au max, mais a priori 2 pour les besoins de cet exercice).

**Vous pouvez adapter ces règles**

On ne se soucie pas des aspects d'interface graphique (affichage, interaction avec les personnes, etc.). L'objectif ici est de concevoir la partie interne du jeu (définir les objets)

Pour vous aider, voici les données que ceux-ci devront se partager :

- La liste des joueurs
- Le jeu de cartes à distribuer
- Les mains de chaque joueur
- Les scores de chaque joueur

Et voici les traitements qui devront être réalisés:

- démarrer une nouvelle partie
- mélanger le jeu de carte
- donner une carte à un joueur
- calculer la valeur d'une main
- vérifier que la valeur d'une main n'est pas au dessus de 21
- préciser qu'un joueur ne veut plus recevoir de cartes ou confirmer qu'il veut bien recevoir des cartes
- compter le nombre de joueurs engagés et, si personne n'est engagé, calculer qui est le gagnant

1. Une conception qui consisterait à ne faire qu'un seul objet ne serait pas cohérente. Est-il possible de couper en deux types d'objets : donneur et joueurs ? Précisez les données et les traitements pour chaque objet.
2. Expliquez les relations (échange de message) entre ces objets. Déroulez une partie avec deux joueurs. Vous pouvez alors montrer quels messages sont échangés et comment les objets (leurs données) évoluent.
3. Développez la classe Joueur

- l. Développez la classe `Donneur`
- i. Développez un main (script montrant le déroulement d'une application avec deux joueurs)

## 2 - Dessin Vectoriel

Une application que nous souhaitons développer est celle d'un outil de dessin vectoriel. Nous considérons qu'il est possible de créer un nouveau dessin vierge. Ensuite, il est possible d'ajouter des formes vectorielles à ce dessin: des points, des droites, des carrés, des rectangles et des cercles. Une fois les formes créées, il est possible de les modifier, de les déplacer et de les supprimer. Enfin, il est possible de sauvegarder le dessin dans un fichier au **format SVG** pour qu'il soit affichable dans un navigateur web.

- . Proposez une conception à l'aide d'objets. Vous préciserez alors quels sont les objets qui composent cette application.
- ?. Proposez un script qui construit un dessin avec deux droites et deux carrés puis qui supprime le deuxième carré.
- l. Développez la classe `Dessin`
- l. Développez plusieurs classes représentant des formes graphiques en exploitant l'héritage.

## 3 - Aller plus loin

- . Proposez un mécanisme permettant la sauvegarde d'un dessin vectoriel sous forme d'un fichier SVG.
- ?. Faites en sorte que le code de sauvegarde dépende du code métier et pas l'inverse.

# Exercices sur la programmation fonctionnelle

[Retour à l'ensemble des exercices](#)

- **Recursiveité** : des exemples d'utilisation de la récursivité.
- **Fonctions caractéristiques** : un exemple de représentation des données par les fonctions.
- **Map & Reduce** : un exemple de transformations fonctionnelles sur les listes avec possibilité de parallélisation.
- **Programmation paresseuse** : un exemple de contrôle de l'évaluation sous la forme de programmation paresseuse.

# Récurtivité

- [Retour aux exercices de programmation fonctionnelle](#)
- [Accès aux solutions](#)

Cette page contient un ensemble de fonctions qu'il est possible de programmer de manière fonctionnelle pure, en utilisant des algorithmes récurtifs. A chaque fois, on insiste sur la description formelle de la fonction à représenter, on discute les problèmes de terminaison, et de complexité.

## 1ère partie : Calcul de puissance

Les deux définitions suivantes permettent de calculer la fonction  $\text{pow}(a, n) = a^n$  :

$$\begin{cases} \text{pow}_1(a, 0) &= 1 \\ \text{pow}_1(a, n) &= a * \text{pow}_1(a, n-1) \quad \text{si } n \geq 1 \end{cases}$$

$$\begin{cases} \text{pow}_2(a, 0) &= 1 \\ \text{pow}_2(a, 1) &= a \\ \text{pow}_2(a, 2 * n) &= \text{pow}_2(a, n)^2 \quad \text{si } n \geq 1 \\ \text{pow}_2(a, 2 * n + 1) &= a * \text{pow}_2(a, n)^2 \quad \text{si } n \geq 1 \end{cases}$$

Écrire chacun de ces algorithmes en Python. Évaluer pour chacun leur complexité. Expliquer pourquoi le second algorithme nécessite plus de cas pour sa définition que le premier.

## 2ème partie : PGCD

Une façon de calculer le plus grand commun diviseur entre deux nombres consiste à implémenter [l'algorithme d'Euclide](#), dont on fournit une définition ici :

$$\begin{cases} \text{pgcd}(0, b) &= b \\ \text{pgcd}(b, b) &= b \\ \text{pgcd}(a, b) &= \text{pgcd}(b, a) \quad \text{si } a > b \\ \text{pgcd}(a, b) &= \text{pgcd}(b \bmod a, a) \quad \text{sinon} \end{cases}$$

Dessiner sur un quart de plan le chemin réalisé par le calcul de  $\text{pgcd}(7, 5)$ . Donner une borne supérieure de la complexité du calcul de  $\text{pgcd}(a, b)$ . Écrire cet algorithme en Python, d'abord de manière réursive, et ensuite avec une boucle `for`.

## 3ème partie : suite de Syracuse

La suite de Syracuse est une suite d'entiers naturels paramétrée par une valeur de départ donnée, qui peut être construite à l'aide de la fonction suivante :

$$\begin{cases} \text{syr}(1) &= 1 \\ \text{syr}(n) &= \text{syr}(n/2) \quad \text{si } n \text{ est pair et } \geq 1 \\ \text{syr}(n) &= \text{syr}(3 * n + 1) \quad \text{sinon} \end{cases}$$

Cette suite est connue pour la fameuse [conjecture de Syracuse](#) qui lui est associée. En 2017, il a été vérifié que pour tout entier  $n \leq 87 \times 2^{60}$ ,  $\text{syr}(n) = 1$ .

Écrire le code en Python calculant la fonction précédente. Écrire le code calculant la liste des valeurs atteintes lors du calcul de  $\text{syr}(n)$ , aussi appelé  $\text{vol}(n)$ . Par exemple, pour  $n = 12$  :

$$\text{vol}(12) = [12, 6, 3, 10, 5, 16, 8, 4, 2, 1]$$

**Plus difficile** Calculer l'entier ayant le vol le plus long compris entre  $l$  et un entier  $n$  donné.

## 4ème partie : récursivité et listes

Les listes sont des types de données que l'on peut appeler **inductifs**, signifiant ainsi qu'ils peuvent être pensés d'une manière récursive. En effet, une liste  $l$  est :

- soit la liste vide ( $[]$ ),
- soit composée d'un premier élément (la tête,  $l[0]$ ), et d'une sous-liste de taille plus petite (la queue,  $l[1:]$ ).

Écrire un code en Python recherchant de manière récursive si un élément appartient à une liste Python.

$$\begin{cases} \text{search}([], x) &= False \\ \text{search}(l, x) &= True \quad \text{si } l \text{ est non vide et } x = l[0] \\ \text{search}(l, x) &= \text{search}(l[1:], x) \quad \text{sinon} \end{cases}$$

Comparer au code réalisant la même chose avec une boucle `for`.

Adapter ce code pour compter le nombre d'occurrences d'un élément apparaissant dans une liste, cela d'abord de manière récursive, puis avec une boucle `for`. Déterminer parmi les fonctions que vous venez d'écrire celles qui sont pures et celles qui font des effets de bord.



# Récurtivité

- [Retour aux exercices de programmation fonctionnelle](#)
- [Retour aux exercices sur la récursivité](#)

Cette page contient un ensemble de fonctions qu'il est possible de programmer de manière fonctionnelle pure, en utilisant des algorithmes récursifs. A chaque fois, on insiste sur la description formelle de la fonction à représenter, on discute les problèmes de terminaison, et de complexité.

## 1ère partie : Exponentiation

- La version linéaire

```
def power_lin(a, n):  
    if (n == 0):  
        return 1  
    else:  
        return a * power_lin(a, n-1)
```

- La version binaire

```
def power_bin(a, n):  
    if (n == 0):  
        return 1  
    elif (n == 1):  
        return a  
    elif (n % 2 == 0):  
        b = power_bin(a, n // 2)  
        return b*b  
    else:  
        b = power_bin(a, n // 2)  
        return a * b * b
```

## 2ème partie : PGCD

- La version récursive

```
def pgcd_rec(a, b):  
    if a == 0:  
        return b  
    elif a == b:  
        return b  
    elif a > b:  
        return pgcd_rec(b, a)  
    else:  
        return pgcd_rec(b % a, a)
```

- La version impérative (avec boucle for)

```
def pgcd_imp(a, b):  
    if a > b:  
        a, b = b, a  
    while (a != 0) and (a != b) :
```

```

    a, b = b % a, a
    return b

```

- La page [Rosetta Code](#) contenant un ensemble d'écritures différentes pour la fonction PGCD.
- Un exemple de fonction pour tester les différentes fonctions de calcul de PGCD :

```

def test_pgcd(pgcd_fun):
    test_values = [
        ((0,0), 0),
        ((4,4), 4),
        ((4,0), 4),
        ((3,5), 1),
        ((5,3), 1),
        ((12,42), 6),
    ]
    for ((a,b), c) in test_values:
        assert(pgcd_fun(a,b) == c)

```

### 3ème partie : suite de Syracuse

```

def syra(n):
    if (n == 1):
        return 1
    elif (n % 2 == 0):
        return syra(n // 2)
    else:
        return syra(3*n + 1)

```

```

def flight(n):
    if (n == 1):
        return [1]
    elif (n % 2 == 0):
        return [n] + flight(n // 2)
    else:
        return [n] + flight(3*n + 1)

```

```

def flight_len(n):
    if (n == 1):
        return 1
    elif (n % 2 == 0):
        return 1 + flight_len(n // 2)
    else:
        return 1 + flight_len(3*n + 1)

```

```

def max_flight(flight_fun, n):
    max_s = 1
    imax_s = 1
    for i in range(2, n+1):
        l = flight_fun(i)
        if (l > max_s):
            max_s = l
            imax_s = i
    return imax_s

```

### 4ème partie : récursivité et listes

```

def search_rec(l, x):

```

```
if (l == []):  
    return False  
elif (l[0] == x):  
    return True  
else:  
    return search_rec(l[1:], x)
```

```
def search_for(l, x):  
    for y in l:  
        if (x == y):  
            return True  
    return False
```

```
def count_rec(l, x):  
    if (l == []):  
        return 0  
    elif (l[0] == x):  
        return 1 + count_rec(l[1:], x)  
    else:  
        return count_rec(l[1:], x)
```

```
def count_for(l, x):  
    res = 0  
    for y in l:  
        if (y == x):  
            res += 1  
    return res
```

# Fonctions caractéristiques

- [Retour aux exercices de programmation fonctionnelle](#)
- [Accès aux solutions](#)

Cette page contient des exercices mettant en valeur les fonctions comme capables de représenter des données. L'exemple mis en place ici est celui des **fonctions caractéristiques**, et est décliné en deux parties, la première permettant de représenter des ensembles de points du plan, la seconde portant sur une application de gestion de listes de lectures de fichiers musicaux.

## 1ère partie : les fonctions caractéristiques

Soit  $E$  un ensemble (de points du plan, de fichiers, d'espèces animales ...). Une fonction caractéristique est simplement une fonction qui prend un paramètre  $x$ , et répond `True` si  $x \in E$  et `False` sinon. Par exemple, la fonction suivante représente l'ensemble contenant tout :

```
def set_all(x):
    return True
```

Il s'agit d'un exemple simple de **représentation des données par des fonctions**. Ici, la donnée, i.e l'ensemble, est identifié à sa fonction caractéristique. Avec cet exemple, il est facile d'écrire en Python la fonction représentant l'ensemble vide. Noter que l'on peut restreindre l'usage de cette fonction à un ensemble particulier. Par exemple, si on considère que  $x$  est restreint au points du plan, alors les ensembles que l'on va représenter seront des ensembles de points du plan.

Écrire la fonction représentant le singleton  $x$  est déjà plus compliqué, puisque l'on désire une fonction qui prend en paramètre  $x$  et renvoie une fonction caractéristique. Compléter le code suivant :

```
def set_single(x):
    return lambda y: (?? == ??)
```

Écrire une fonction qui prend un ensemble, et renvoie son complémentaire. Écrire une fonction qui prend deux ensembles, et renvoie son intersection, et une autre pour son union. Écrire une fonction qui prend un ensemble  $E$  et un paramètre  $x$ , et renvoie l'ensemble  $E \cup \{x\}$ .

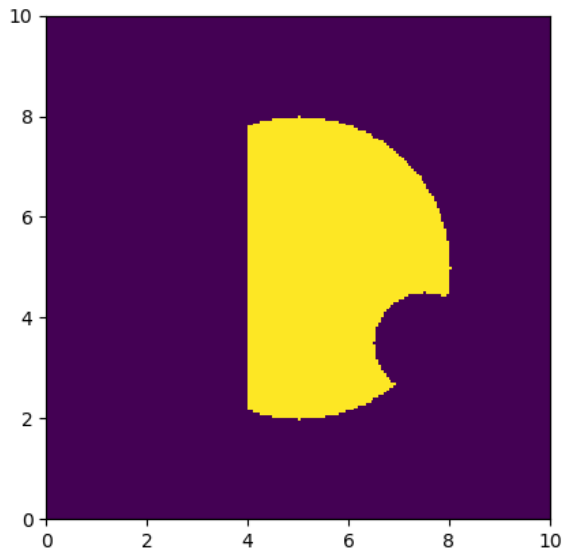
## 2ème partie : les ensembles de points du plan

Considérons la possibilité de dessiner des ensembles de points du plan, avec la bibliothèque `matplotlib`. Dans les exemples qui suivent, les points du plan sont des paires  $(x, y)$ . Par exemple, la fonction suivante permet de produire un dessin à partir d'un ensemble :

```
def set_display_2d_generic(c1, c2, n, s):
    """ Draw a set s with matplotlib inside a window
        c1 is the lower left point
        c2 is the upper right point
        n is the number of divisions """
    h1 = (c2[0] - c1[0]) / n
    h2 = (c2[1] - c1[1]) / n
    m = np.zeros((n,n))
    for i in range(n):
        for j in range(n):
            m[i,j] = 1 if s((c1[1] + j * h2,
                           c2[0] - i * h1)) else 0
    mp.imshow(m, extent=[c1[0], c2[0], c1[1], c2[1]])
    mp.show()

def set_display_2d(s):
    set_display_2d_generic((0,0), (10,10), 200, s)
```

Les ensembles dessinés à l'aide de `set_display_2d` sont affichés dans le carré d'extrémités  $(0, 0)$  et  $(10, 10)$ .



Écrire une fonction de distance euclidienne entre les points définis précédemment. Écrire la fonction caractéristique d'un disque centré sur un point  $c$  de rayon  $r$ . Écrire la fonction d'un rectangle défini par ses deux extrémités(*gauche, bas*) et (*droite, haut*). Dessiner un [Pac-Man](#).

### 3ème partie : les listes de lectures

Telles quelles, les fonctions caractéristiques gardent un parfum “mathématique”. Une application des ensembles de points consiste à manipuler des ensembles de fichiers (ici musicaux) répondant à un certain nombre de critères.

```
songs = [ \
{ "title": "Walking on Broken Glass", "artist": "Annie Lennox", "album": "Diva", "track": 2, "genre": "pop rock" },
{ "title": "Roar", "artist": "Katy Perry", "album": "PRISM", "track": 1, "genre": "power pop" },
{ "title": "Don't Wanna Fight", "artist": "Alabama Shakes", "album": "Sound & Color", "track": 2, "genre": "blues ro
{ "title": "Grace Kelly", "artist": "Mika", "album": "Life in Cartoon Motion", "track": 1, "genre": "glam rock" },
{ "title": "Don't Stop Me Now", "artist": "Queen", "album": "Jazz", "track": 6, "genre": "pop rock" },
{ "title": "Black Pearls", "artist": "Apollo Brown", "album": "Clouds", "track": 7, "genre": "hip hop underground" }
]
```

(le choix des morceaux dans cette liste a été aimablement fourni par une playlist aléatoire d'un site de musique en ligne, dans des conditions d'exercice, on pourrait demander à chaque élève de produire une musique, et les mettre ensemble)

Écrire une requête représentant l'ensemble des morceaux dont le genre contient la chaîne "rock". Écrire une requête représentant l'ensemble des morceaux dont le titre débute par "Don't".

Écrire une fonction prenant une requête, et générant une liste de lecture de morceaux répondant à cette requête.

# Fonctions caractéristiques

- [Retour aux exercices de programmation fonctionnelle](#)
- [Retour aux exercices sur les fonctions caractéristiques](#)

## 1ère partie : les fonctions caractéristiques

```
def set_all(x):  
    return True  
  
def set_empty(x):  
    return False  
  
def set_single(x):  
    return lambda y: (x == y)  
  
def belongs(s, x):  
    return s(x)  
  
def set_complement(s):  
    return lambda y: not(s(y))  
  
def set_union(s1, s2):  
    return lambda y: s1(y) or s2(y)  
  
def set_intersection(s1, s2):  
    return lambda y: s1(y) and s2(y)  
  
def set_add(s, x):  
    return lambda y: s(y) or y == x
```

## 2ème partie : les ensembles de points du plan

```
def dist(p1, p2):  
    return np.sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)  
  
def set_circle(c, r):  
    return lambda y: dist(y, c) <= r  
  
def set_square(p1, p2):  
    return lambda y: y[0] >= p1[0] and y[0] <= p2[0] and \  
        y[1] >= p1[1] and y[1] <= p2[1]  
  
c1 = set_circle((5,5), 3)  
c2 = set_complement(set_circle((7.5,3.5), 1))  
r = set_square((4,0), (10,10))  
s = set_intersection(set_intersection(c1,c2), r)  
  
set_display_2d(s)
```

### 3ème partie : les listes de lectures

```
r1 = lambda m: "rock" in m["genre"]
r2 = lambda m: m["title"].startswith("Don't")

def generate_playlist(req, n):
    res = []
    valid_songs = [ s for s in songs if req(s) ]
    if len(valid_songs) == 0:
        return res
    for i in range(n):
        res.append(random.choice(valid_songs))
    return res
```

# Map & Reduce

- [Retour aux exercices de programmation fonctionnelle](#)
- [Accès aux solutions](#)

Cette page contient un ensemble d'exemples montrant comment utiliser des fonctions d'ordre supérieur (des fonctions prenant en paramètre d'autres fonction, comme `filter`, `map`, `reduce` ...) pour manipuler des ensembles de données. La seconde partie montre comment, dans des conditions où les transformations sont pures (i.e ne font pas d'effet de bord), le calcul peut être parallélisé.

## 1ère partie : les fonctions d'ordre supérieur

Un exemple très simple de fonction d'ordre supérieur est la fonction appelée `map`, qui prend en paramètre une liste  $l$  et une fonction  $f$ , applique la fonction à tous les éléments de  $l$  et renvoie la liste des résultats :

$$\begin{aligned} \text{map} : (A \rightarrow B, \text{List}[A]) &\rightarrow \text{List}[B] \\ \text{map}(f, [l_1, \dots, l_n]) &= [f(l_1), \dots, f(l_n)] \end{aligned}$$

Cette fonction existe en Python et possède sa propre [documentation](#), et en voici un cas d'usage :

```
>>> numbers = [1, 2, 3, 4, 5]
>>> sqrList = map(lambda x: x*x, numbers)
>>> sqrList
<map object at 0x7fd20e91ad50>
>>> list(sqrList) # must convert back to list to see contents
[1, 4, 9, 16, 25]
```

Le Python complique un peu les choses, parce qu'il permet de faire les opérations précédentes en utilisant la syntaxe des compréhensions de liste :

```
>>> numbers = [1, 2, 3, 4, 5]
>>> sqrList = [x*x for x in numbers]
>>> sqrList
[1, 4, 9, 16, 25]
```

Écrire la fonction `map` comme une fonction Python. Écrire la fonction `filter` qui prend en paramètre une fonction de filtre  $f$  et une liste  $l$ , et renvoie la liste des éléments de  $l$  pour lesquels  $f$  renvoie `True`. Écrire la fonction `sum` qui prend en paramètre une fonction  $f$  et une liste  $l$ , et qui renvoie la somme des  $f(l_i)$ .

Écrire une fonction `reduce` qui suit la spécification suivante (une écriture récursive est la manière la plus simple de faire) :

$$\begin{cases} \text{reduce} :: ((A, B) \rightarrow A), A, \text{List}[B] &\rightarrow A \\ \text{reduce}(f, acc, []) &= acc \\ \text{reduce}(f, acc, [l_1, l_2, \dots, l_n]) &= \text{reduce}(f, f(acc, l_1), [l_2, \dots, l_n]) \end{cases}$$

En pratique, cette fonction calcule la chose suivante :

$$\text{reduce}(f, acc, [l_1, l_2, \dots, l_n]) = f(\dots(f(f(acc, l_1), l_2) \dots), l_n)$$

Noter que cette fonction existe dans le module `functools` et possède une [documentation](#).

Réécrire la fonction `sum` à partir de la fonction `reduce`. Écrire à partir de la fonction `reduce` une fonction `any` qui teste si une liste de booléens contient un élément valant `True`. **Difficile** Utiliser la fonction `reduce` pour écrire une fonction `reverse` qui prend une liste et renvoie la liste symétrique.



## 2ème partie : sagesse de l'Antiquité

Les fonctions évoquées précédemment servent en quelque sorte de “couteau-suisse” pour effectuer des transformations sur des ensembles de données. Un usage courant de ces fonctions est la manipulation de données héritées d'une base de données. Pour simuler la chose ici, on propose de construire une petite base de données comme la liste suivante :

```
[
{ "name": "Thalès", "birth": -625, "death": -547 },
{ "name": "Anaximandre", "birth": -600, "death": -546 },
{ "name": "Héraclite", "birth": -544, "death": -480 },
{ "name": "Empédocle", "birth": -490, "death": -430 },
]
```

### Lien direct

Utiliser à bon escient les fonctions `map`, `filter`, `sort` et `reduce` pour effectuer les calculs suivants :

- extraire de cette base la liste des noms dans l'ordre alphabétique;
- extraire de cette base la personne née le plus tôt dans l'ordre chronologique;
- extraire de cette base la personne dont la durée de vie a été la plus longue;
- extraire de cette base la moyenne des durées de vies de ces sages.

## 3ème partie **Pour aller plus loin ...** : questions de parallélisation

Hormis la possibilité de pouvoir manipuler des données, les fonctions comme `map` permettent de mettre en place du parallélisme de manière (plus ou moins) automatique. Le module `processing` permet ainsi de distribuer des calculs sur des ensembles (pools) de processeurs. Par exemple :

```
import multiprocessing
import requests

def scrape(url):
    print("Scraping '{}'.format(url))
    res = requests.get(url)
    print("Returned {} ({}).format(res.url, res.status_code))
    return res

def parallel_scrape():
    num_workers = 5
    all_urls = [
        # insert here list of URLs
    ]

    with multiprocessing.Pool(num_workers) as pool:
        results = pool.map(scrape, all_urls)
        print(results)
```

Le choix d'une fonction comme `requests.get` permet d'assurer que les temps de retours des différents serveurs soient non triviaux et dépendant de suffisamment de conditions extérieures pour paraître aléatoires.

Tester le code précédent avec une liste d'URLs choisie avec délicatesse pour ne pas surcharger les serveurs. Considérer le fait que la parallélisation ne fonctionne bien uniquement parce que les fonctions comme `scrape` ne font pas d'effets de bords (i.e leurs appels sont indépendants).

# Map & Reduce

- [Retour aux exercices de programmation fonctionnelle](#)
- [Retour aux exercices sur map & reduce](#)

## 1ère partie : les fonctions d'ordre supérieur

```
def map_imp(f, l):
    res = []
    for x in l:
        res.append(f(x))
    return res

def map_rec(f, l):
    if (len(l) == 0):
        return []
    else:
        return [f(l[0])] + map_rec(f, l[1:])
```

```
def filter_imp(f, l):
    res = []
    for x in l:
        if (f(x)):
            res.append(x)
    return res

def filter_rec(f, l):
    if (len(l) == 0):
        return []
    elif f(l[0]):
        return [l[0]] + filter_rec(f, l[1:])
    else:
        return filter_rec(f, l[1:])
```

```
def sum_imp(f, l):
    res = 0
    for x in l:
        res += f(x)
    return res

def sum_rec(f, l):
    if (len(l) == 0):
        return 0
    else:
        return f(l[0]) + sum_rec(f, l[1:])
```

```
def reduce_imp(f, acc, l):
    res = acc
    for x in l:
```

```

        res = f(res, x)
    return res

def reduce_rec(f, acc, l):
    if (len(l) == 0):
        return acc
    else:
        return reduce_rec(f, f(acc, l[0]), l[1:])

def reverse_reduce(l):
    return reduce_rec(lambda acc, x: [x] + acc, [], l)

```

## 2ème partie : sagesse de l'Antiquité

```

age = lambda s: s["death"] - s["birth"]

names = sorted(map_rec(lambda s: s["name"], sages))
names = sorted([s["name"] for s in sages])

most_ancient = sorted(sages, key=lambda s: s["birth"])[0]

most_longlived = sorted(sages, key=age, reverse=True)[0]

mean_age = sum(map_rec(age, sages)) / len(sages)

```

## 3ème partie : questions de parallélisation

Pas de code demandé dans cette partie.

# Programmation paresseuse

- [Retour aux exercices de programmation fonctionnelle](#)
- [Accès aux solutions](#)

Cette page donne des exemples de contrôle de l'évaluation sous la forme d'une technique de programmation faisant appel à de l'**évaluation paresseuse**. Cette technique permet en particulier de restreindre les calculs pour ne réaliser que ceux qui sont nécessaires à la complétion d'un programme.

## 1ère partie : les itérateurs Python

Un premier exemple de contrôle de l'évaluation apparaît dans des objets que l'on manipule très vite en Python lorsque l'on manipule des listes. Considérons pour commencer la fonction suivante très simple qui fait un affichage et un calcul :

```
def f(x):
    print("Using {}".format(x))
    return x+1
```

L'intérêt de cette fonction, c'est d'afficher un message au moment de son exécution. Ainsi, il est possible d'appliquer cette fonction à plusieurs valeurs de la manière suivante :

```
>> [f(x) for x in range(3)]
Using 0
Using 1
Using 2
[1, 2, 3]
```

Très naturellement, le calcul de la liste d'arrivée entraîne l'application de la fonction `f` sur les valeurs de la liste. Les affichages sont réalisés pendant les calculs, et la liste obtenue est renvoyée à la fin.

Maintenant, réalisons le même calcul à l'aide de la fonction `map`. Cette fonction ne renvoie pas une liste, mais un itérateur :

```
>> it = map(f, range(3))          # returns <map object at 0x7fd20e91aed0>
>> it.__next__()                  # prints "Using 0", returns 1
>> it.__next__()                  # prints "Using 1", returns 2
>> it.__next__()                  # prints "Using 2", returns 3
>> it.__next__()                  # raises an error : StopIteration
```

L'application de la méthode `__next__` entraîne à chaque fois une application de la fonction `f`. L'itérateur (dans ce cas le `<map object>`) permet de produire pas à pas les éléments dans la liste finale. Il s'agit d'un exemple de contrôle de l'évaluation : les calculs sur la liste sont effectués à la demande,

Construire un itérateur avec la fonction `filter`, et extraire les éléments de la liste un par un.

**Difficile.** Une fois que l'on a compris qu'on pouvait extraire les éléments d'une liste en les demandant un par un, rien n'empêche plus de construire des listes infinies (au sens où : desquelles on peut extraire autant d'éléments qu'on veut). Dans l'exemple suivant, estimer combien d'entiers on peut tirer de cet itérateur :

```
def make_f():
    x = [0]
    def f():
        x[0] = x[0]+1
        return x[0]
    return f
```

```
it = iter(make_f(), -1) # -1 acts as a sentinel indicating the end of the list
for x in it: print(x)
```

Ces exemples peuvent paraître un peu futiles, mais correspondent en fait à des fonctions apparaissant dans le module `itertools` (cf. [documentation](#)).

## 2ème partie : les expressions congelées

Une manière très simple de contrôler la réalisation d'un calcul consiste à l'encapsuler dans une fonction sans paramètre. Le calcul sera effectué uniquement lorsque la fonction sera exécutée. Par exemple, la fonction suivante transforme une valeur en une valeur encapsulée à l'intérieur d'une fonction.

```
def make_value(x):
    return lambda : x

f = make_value(666)
print(f)      # <function make_value.<locals>.<lambda> at 0x7fb1899b6e60>
print(f())    # 666
```

On dit que cette valeur est gelée (frozen en anglais). Bien sûr, cela ne semble pas avoir beaucoup d'intérêt, si on ne fait que geler des valeurs déjà calculées. L'intérêt devient plus grand lorsque l'on commence à geler des calculs :

```
def make_addition(a, b):
    return lambda : a() + b()

f = make_addition(make_value(42), make_value(58))
print(f)      # <function make_addition.<locals>.<lambda> at 0x7fb1899cf200>
print(f())    # 100
```

La finesse de ce que l'on obtient ainsi, c'est de pouvoir construire des expressions arbitrairement complexes, sans jamais évaluer les morceaux. Il devient alors envisageable de construire tout un calcul, tout en repoussant le moment où le calcul est réellement effectué. En pratique, toute fonction est une manière de congeler un calcul.

## 3ème partie : le contrôle de l'évaluation

Pour l'instant, les expressions congelées ne semblent pas particulièrement utiles. Cette partie montre un exemple dans lequel le contrôle de l'évaluation permet de décomposer un calcul en la somme de ses étapes, que l'on peut ensuite faire avancer à la demande : une forme primitive de débogueur.

Considérons la fonction `pgcd` construite dans l'exercice [sur la récursivité](#) :

```
def pgcd(a, b):
    if a == 0:
        return b
    elif a == b:
        return b
    elif a > b:
        return pgcd(b, a)
    else:
        return pgcd(b % a, a)
```

Cette fonction a le bon goût de faire une suite de calculs qui sont tous des appels à elle-même. Considérons geler tous ces calculs, en la transformant de la manière suivante :

```
def pgcd_lazy(a, b):
    if a == 0:
        return lambda: b
    elif a == b:
        return lambda: b
    elif a > b:
```

```
    return lambda: pgcd_lazy(b, a)
else:
    return lambda: pgcd_lazy(b % a, a)
```

Extraire de `pgcd_lazy(7, 5)` la valeur finale 1. Comment faire cela ? Combien d'appels de fonctions sont-ils nécessaires ?

En ajoutant les `lambda`, on a bloqué le calcul à chaque étape, mais le résultat ne donne pas beaucoup d'informations sur l'état intermédiaire du calcul. Considérons la version suivante de la même fonction :

```
def pgcd_dbg(a, b):
    dic = {"a":a, "b":b }
    if a == 0:
        return lambda: (b, dic)
    elif a == b:
        return lambda: (b, dic)
    elif a > b:
        return lambda: (pgcd_dbg(b, a), dic)
    else:
        return lambda: (pgcd_dbg(b % a, a), dic)
```

Afficher les différentes étapes du calcul de `pgcd_dbg(7,5)`. De quelles informations supplémentaires dispose-t-on ? En quoi cette manière de faire permet-elle d'implémenter à peu de frais une sorte de débogueur ?

# Programmation paresseuse

- [Retour aux exercices de programmation fonctionnelle](#)
- [Retour aux exercices sur la programmation paresseuse](#)

## 1ère partie : les itérateurs Python

```
it = map(f, range(3))
try:
    for i in range(4):
        print(it.__next__())
except StopIteration:
    print("Exception raised : StopIteration")
```

```
it = filter(lambda x: (f(x) % 2 == 0), range(4))
try:
    for i in range(4):
        print(it.__next__())
except StopIteration:
    print("Exception raised : StopIteration")
```

```
def make_function():
    x = [0]
    def f():
        x[0] = x[0]+1
        return x[0]
    return f

def make_infinite_iter():
    return iter(make_function(), -1)

for i in make_infinite_iter():
    print(i) # Infinite loop
    if i >= 10000:
        break
```

## 2ème partie : les expressions congelées

```
def make_value(x):
    return lambda : x

def make_addition(a, b):
    return lambda : a() + b()

op = make_addition(make_value(5), make_value(6))
op()
```

## 3ème partie : le contrôle de l'évaluation

- La version de base

```
def pgcd(a, b):
    if a == 0:
        return b
    elif a == b:
        return b
    elif a > b:
        return pgcd(b, a)
    else:
        return pgcd(b % a, a)

print(pgcd(7,5))
```

- La version avec contrôle des calculs

```
def pgcd_lazy(a, b):
    if a == 0:
        return lambda: b
    elif a == b:
        return lambda: b
    elif a > b:
        return lambda: pgcd_lazy(b, a)
    else:
        return lambda: pgcd_lazy(b % a, a)

f = pgcd_lazy(7,5)
print(f)
print(f())
print(f()())
print(f()()())
print(f()()()())
print(f()()()()())
```

- La version avec affichage des valeurs intermédiaires

```
def pgcd_dbg(a, b):
    dic = {"a":a, "b":b }
    if a == 0:
        return lambda: (b, dic)
    elif a == b:
        return lambda: (b, dic)
    elif a > b:
        return lambda: (pgcd_dbg(b, a), dic)
    else:
        return lambda: (pgcd_dbg(b % a, a), dic)

f = pgcd_dbg(7,5)
print(f)
print(f())
print(f()[0]())
print(f()[0]() [0]() [0]())
print(f()[0]() [0]() [0]() [0]())
```



# Exercices sur les listes

## [Retour à l'ensemble des exercices](#)

Pour travailler sur les listes, deux TDs sont disponibles, à l'aide des codes suivants :

- Classe Cellule : [code](#)

```
class Cellule:
    liste_vide = None

    def __init__(self, etiquette, liste):
        self._valeur = etiquette
        self._suivant = liste

    def valeur(self):
        return self._valeur

    def suite(self):
        return self._suivant

    @staticmethod
    def est_vide(liste):
        return liste is Cellule.liste_vide
```

La classe s'utilise ainsi :

```
import classe_cellule as cell

l1 = cell.Cellule.liste_vide # returns an empty list
l2 = cell.Cellule(1, cell.Cellule.liste_vide)
l3 = cell.Cellule(2, l2)

cell.Cellule.est_vide(l1) # -> True
cell.Cellule.est_vide(l2) # -> False

l3.suite()                # -> returns a list
l3.suite() == l2          # -> True
```

- Code liste : [code](#)

Essayez d'implémenter les méthodes demandées dans différents paradigmes pour bien intégrer le fonctionnement d'une liste.

- [Opérations sur les listes](#)
- [Représentation polynomiale](#)

# Opérations

- [Retour aux exercices sur les listes](#)

L'objectif de ce TD est de proposer de nouvelles méthodes pour le type `Liste`.

Pour commencer, ajoutez le prédicat suivant :

```
taille : Liste -> entier
# à partir d'une liste L, renvoi le nombre d'éléments qu'elle contient
```

Nous pourrions ensuite ajouter deux autres prédicats :

```
max: Liste -> entier
# à partir d'une liste L, renvoi la valeur maximum

min: Liste -> entier
# à partir d'une liste L, renvoi la valeur minimum
```

Enfin, pour pousser encore plus loin, essayez d'implémenter le transformateur suivant :

```
trier: Liste -> Liste
# à partir d'une liste L, renvoi cette liste avec les valeurs triées (tri au choix)
```

# Polynômes avec des listes

- [Retour aux exercices sur les listes](#)

L'objectif de ce TD est de proposer une gestion de polynômes à l'aide de listes.

Les cellules, triées par ordre décroissant, représenteront chaque degré du polynôme et la valeur de ces cellules sera le multiplicateur. Par exemple, le polynôme  $3x^3 + x + 6$  sera représenté par la liste suivante :

```
[3]-[0]-[1]-[6]
```

Proposez une méthode qui calcule pour un  $x$  donné en paramètre, la valeur du polynôme :

```
calcul: (Valeur * Liste) -> Valeur
# Pour une valeur et un polynome donné, renvoi la valeur calculée du polynôme.
```

Proposez ensuite plusieurs méthodes, d'abord d'addition et de soustraction :

```
addition: (Liste * Liste) -> Liste
# Pour deux polynômes donnés, renvoi la somme des deux

soustraction: (Liste * Liste) -> Liste
# Pour deux polynômes donnés, renvoi la soustraction du premier par le deuxième
```

Enfin, proposez une méthode pour multiplier deux polynômes entre eux :

```
multiplier: (Liste * Liste) -> Liste
# Pour deux polynômes donnés, renvoi le polynôme résultant de la multiplication des ces derniers
```

Pour aller encore plus loin :

```
diviser: (Liste * Liste) -> Liste
# Renvoie la division des deux polynômes en paramètres
```

# Exercices sur les files et piles

[Retour à l'ensemble des exercices](#)

- [Files](#)
- [Piles](#)

# Files

- [Retour aux exercices sur les files et piles](#)

L'objectif de ce TD est de proposer de nouvelles méthodes pour le type `File` à l'aide des codes suivants :

- Classe `File` : [code](#)
- Code `File` : [code](#)

Pour commencer, ajoutez le prédicat suivant :

```
taille : File -> entier
# à partir d'une file F, renvoi le nombre d'éléments qu'elle contient
```

Nous pourrions ensuite ajouter deux autres prédicats :

```
max: File -> entier
# à partir d'une file F, renvoi la valeur maximum

min: File -> entier
# à partir d'une file F, renvoi la valeur minimum
```

Ensuite, proposez une implémentation pour la méthode suivante :

```
inverse: File -> File
# Renvoie une copie de la file en paramètre avec l'ordre des éléments inversé
```

Nous allons maintenant nous intéresser à des opérations un peu plus compliquées :

```
trier: File -> File
# Tri une liste donnée en paramètre

fusionne: (File * File) -> File
# Fusionne deux files triées en une seule (bien triée)
```

Enfin, pour pousser un peu plus loin, proposez une implémentation pour la méthode suivante :

```
popValeur: (Valeur * File) -> File
# Enlève d'une file tout les éléments dont la valeur est passée en paramètre
```

# Piles

- [Retour aux exercices sur les files et piles](#)

L'objectif de ce TD est de proposer de nouvelles méthodes pour le type `Pile` à l'aide des codes suivants:

- Classe `Pile` : [code](#)
- Code `Pile` : [code](#)

Pour commencer, ajoutez le prédicat suivant :

```
taille : Pile -> entier
# à partir d'une pile P, renvoi le nombre d'éléments qu'elle contient
```

Nous pourrions ensuite ajouter deux autres prédicats :

```
max: Pile -> entier
# à partir d'une pile P, renvoi la valeur maximum

min: Pile -> entier
# à partir d'une pile P, renvoi la valeur minimum
```

Ensuite, proposez une implémentation pour la méthode suivante :

```
copie: Pile -> Pile
# Renvoie une copie de la pile en paramètre dans une nouvelle pile
```

Nous allons maintenant nous intéresser à une opération un peu plus compliquée : à partir d'une file contenant des parenthèses, crochets et accolades, vérifier si le parenthésage est bon. Vous aurez besoin d'une pile. Exemple :

```
["(", "[", "]", ")"] -> True
["(", "[", "]", "[", "]"] -> False
```

Enfin, pour pousser un peu plus loin, proposez une implémentation de la méthode suivante :

```
switch: Pile -> Pile
# Inverse les éléments du sommet et du bas de la pile uniquement
```

# Exercices sur les arbres binaires

[Retour à l'ensemble des exercices](#)

- [Opérations sur les arbres](#)
- [Représentation visuelle](#)
- [Expressions arithmétiques](#)

# Opérations

- [Retour aux exercices sur les arbres binaires](#)
- [Accès aux solutions](#)

L'objectif de cet exercice est d'étendre le TADArbre binaire avec de nouvelles opérations dont vous proposerez une implémentation pour la mise en oeuvre à base de listes de listes ([code](#)) et avec la classeNoeud ([code](#)).

Pour commencer, on souhaite étendre le TAD avec le prédicat suivant :

```
est_feuille : Arbre binaire -> bool
# à partir d'un arbre binaire A, produit un booléen indiquant si A est une feuille
```

En utilisant ce prédicat, on veut ensuite ajouter l'opération suivante :

```
compte_feuilles : Arbre binaire -> int
# à partir d'un arbre binaire A, produit un entier indiquant le nombre de feuilles de A
```

Avec une simple modification, vous pourrez calculer la taille d'un arbre, c'est-à-dire son nombre de nœuds :

```
taille : Arbre binaire -> int
# à partir d'un arbre binaire A, produit un entier indiquant la taille de A
```

On souhaite enfin calculer la hauteur d'un arbre, c'est-à-dire sa profondeur maximale :

```
hauteur : Arbre binaire -> int
# à partir d'un arbre binaire A, produit un entier indiquant la hauteur de A
```



# Opérations - solutions

- [Retour aux exercices sur les arbres binaires](#)
- [Retour à l'exercice](#)

## Solutions pour la mise en oeuvre avec des listes de listes :

```
def est_feuille(arbre):
    return est_vide(gauche(arbre)) and est_vide(droit(arbre))

def compte_feuille (arbre):
    if est_vide(arbre):
        return 0
    if est_feuille(arbre):
        return 1
    return compte_feuille(gauche(arbre)) + compte_feuille(droit(arbre))

def taille (arbre):
    if est_vide(arbre):
        return 0
    return 1 + taille(gauche(arbre)) + taille(droit(arbre))

def hauteur (arbre):
    if est_vide(arbre):
        return -1
    h1 = 1 + hauteur(gauche(arbre))
    h2 = 1 + hauteur(droit(arbre))
    return max(h1,h2)
```

## Solutions pour la mise en oeuvre avec la classe Noeud :

```
def est_feuille(self):
    return Noeud.est_vide(self.gauche()) and Noeud.est_vide(self.droit())

def compte_feuille (self):
    if Noeud.est_vide(self):
        return 0
    if self.est_feuille():
        return 1
    return self.gauche().compte_feuille() + self.droit().compte_feuille()

def taille (self):
    if Noeud.est_vide(self):
        return 0
    if self.est_feuille():
        return 1
    return 1 + self.gauche().taille() + self.droit().taille()
```

```
def hauteur (self):  
    if Noeud.est_vide(self):  
        return -1  
    if self.est_feuille():  
        return 0  
    h1 = 1 + self.gauche().hauteur()  
    h2 = 1 + self.droit().hauteur()  
    return max(h1,h2)
```

# Représentation visuelle

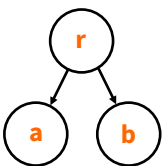
- [Retour aux exercices sur les arbres binaires](#)
- [Accès aux solutions](#)

Dans cet exercice nous aimerions produire une représentation visuelle lors de l’affichage de l’arbre sur le terminal.

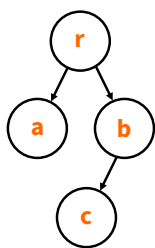
Par exemple, l’arbre  sera affiché par :

```
r
- *
- *
```

où le symbole – indique la profondeur et \* désigne l’arbre vide.

L’arbre  sera visualisé ainsi :

```
r
-a
-- *
-- *
-b
-- *
-- *
```

et l’arbre  ainsi :

```
r
-a
-- *
-- *
-b
--c
--- *
--- *
-- *
```

Écrire une fonction `represente` permettant cette représentation textuelle. Dans le cas de la classe `Noeud`, cette fonction sera une méthode statique. Vous réalisez ainsi une **parcours préfixe** de l’arbre.

*Remarque* : le comportement de la fonction Python `print` en fin de ligne peut être modifié à l’aide du paramètre `end` (par défaut égal à `'\n'`).

# Représentation visuelle - solutions

- [Retour aux exercices sur les arbres binaires](#)
- [Retour à l'exercice](#)

## Solution pour la mise en oeuvre avec des listes de listes :

```
def represente(arbre, p = 0) :  
    if est_vide(arbre):  
        print('*')  
    else:  
        print(etiquette(arbre))  
        p += 1  
        print('-' * p, end='')  
        represente(gauche(arbre), p)  
        print('-' * p, end='')  
        represente(droit(arbre), p)
```

## Solution pour la mise en oeuvre avec la classe Noeud :

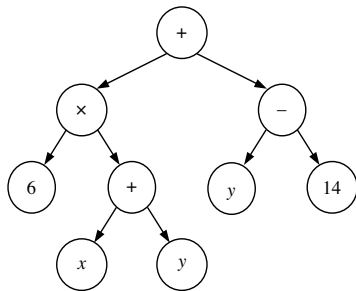
```
@staticmethod  
def represente (arbre, p=0):  
    if Noeud.est_vide(arbre):  
        print('*')  
    else :  
        print(arbre.etiquette())  
        p += 1  
        print('-' * p, end='')  
        Noeud.represente(arbre.gauche())  
        print('-' * p, end='')  
        Noeud.represente(arbre.droit())
```

# Expressions arithmétiques

- [Retour aux exercices sur les arbres binaires](#)
- [Accès aux solutions](#)

Une expression arithmétique construite avec des opérateurs binaires (c'est-à-dire à deux opérandes telles l'addition, la soustraction, la multiplication, la division) peut être représentée par un arbre binaire dont les nœuds internes portent les opérateurs et les feuilles des symboles de variables ( $x, y, z$ , etc.) ou des constantes (6, 14, etc.)

Ainsi, l'arbre suivant représente l'expression  $6(x + y) + (y - 14)$ .



En utilisant l'implémentation à base de listes, cet arbre peut être construits avec l'instruction suivante :

```

expression = noeud('+',
    noeud('*',
        noeud(6, arbre_vide(), arbre_vide()),
        noeud('+',
            noeud('x', arbre_vide(), arbre_vide()),
            noeud('y', arbre_vide(), arbre_vide()))),
    noeud('-',
        noeud('y', arbre_vide(), arbre_vide()),
        noeud(14, arbre_vide(), arbre_vide())))
  
```

- Proposez une fonction `affiche` qui prend en paramètre un arbre représentant une expression arithmétique et affiche cette expression sur le terminal. Vous réalisez ainsi une **parcours infixe** de l'arbre.
- Proposez une fonction `evalue` qui prend en paramètre un arbre représentant une expression arithmétique et renvoie la valeur de cette expression.  
Il est nécessaire de fournir à cette fonction les valeurs associées à chacune des variables présentes dans l'expression. Quelle structure de données proposez-vous d'utiliser pour mémoriser ces associations ?
- Proposez une fonction `affichePolonaise` qui prend en paramètre un arbre représentant une expression arithmétique et affiche cette expression en utilisant la **notation polonaise inverse**. Vous réalisez ainsi une **parcours postfixe** de l'arbre. Remarquez que cette expression non-ambiguë ne nécessite pas de parenthèses.

# Expressions arithmétiques - solutions

- [Retour aux exercices sur les arbres binaires](#)
- [Retour à l'exercice](#)

## Question 1

```
def affiche(expression):  
    if est_feuille(expression):  
        print(etiquette(expression), end='')  
    else:  
        print('(', end='')  
        affiche(gauche(expression))  
        print(etiquette(expression), end='')  
        affiche(droit(expression))  
        print(')', end='')
```

## Question 2

```
def evalue(expression, valeurs):  
    if est_vide(expression):  
        return 0  
    if est_feuille(expression):  
        v = etiquette(expression)  
        if v in valeurs:  
            return valeurs[v]  
        return v  
    a=evalue(gauche(expression), valeurs)  
    b=evalue(droit(expression), valeurs)  
    o=etiquette(expression)  
    if o=='+':  
        return a+b  
    if o=='*':  
        return a*b  
    if o=='-':  
        return a-b  
    if o=='/':  
        return a/b
```

## Question 3

```
def affichePolonaise(expression):  
    if est_feuille(expression):  
        print(etiquette(expression), end=' ')  
    else:  
        affichePolonaise(gauche(expression))  
        affichePolonaise(droit(expression))  
        print(etiquette(expression), end=' ')
```

# Exercices sur les graphes

[Retour à l'ensemble des exercices](#)

- [Manipulation des représentations](#)
- [Opérations sur les graphes](#)
- [Passage d'une représentation à l'autre](#)

# Manipulation des représentations

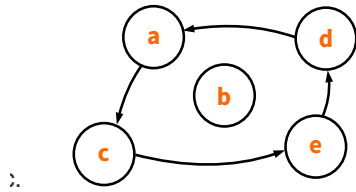
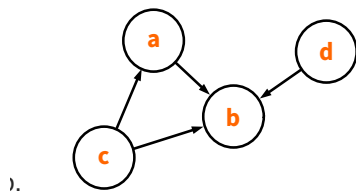
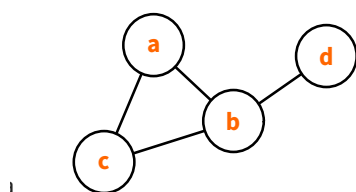
• [Retour aux exercices sur les graphes](#)

• [Accès aux solutions](#)

L'objectif des exercices de cette page est de manipuler sur papier les deux représentations des graphes vues en cours : les matrices d'adjacence et les listes de successeurs.

. Pour chaque graphe ci-dessous, donnez :

- sa matrices d'adjacence,
- ses listes de successeurs,
- ses listes de prédécesseurs.



4. Pour chaque tableau ci-dessous, tracez le graphe correspondant et donnez sa matrice d'adjacence.

1.

sommets	a	b	c	d	e
successeurs	b	c,d	d	e	

2.

sommets	a	b	c	d	e
prédécesseurs	b	c,d	d	e	

3.

sommet	a	b	c	d
prédécesseurs	b,c,d	a,c	b,a,d	a,c

4. Pour chaque matrice d'adjacence ci-dessous, tracez le graphe associé et donnez ses listes de successeurs et de prédécesseurs.



1.  $\begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}$  associée à l'ensemble de sommets  $S = \{a, b, c\}$ .

2.  $\begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix}$  associée à l'ensemble de sommets  $S = \{a, b, c, d\}$ .

# Manipulation des représentations - solutions

- [Retour aux exercices sur les graphes](#)
- [Retour à l'exercice](#)

## Question 1

- Matrices d'adjacence

1. 
$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

2. 
$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

3. 
$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

- Listes de successeurs :

1.

sommet	a	b	c	d
successeurs	b,c	a,c,d	a,b	b

2.

sommet	a	b	c	d
successeurs	b		a,b	b

3.

sommet	a	b	c	d	e
successeurs	c		e	a	d

- Listes de prédécesseurs :

1.

sommet	a	b	c	d
prédécesseurs	b,c	a,c,d	a,b	b

).

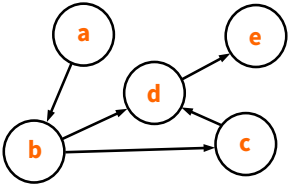
sommet	a	b	c	d
prédécesseurs	c	a,c,d		

).

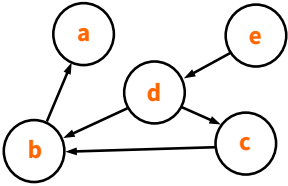
sommet	a	b	c	d	e
prédécesseurs	d		a	e	c

Question 2

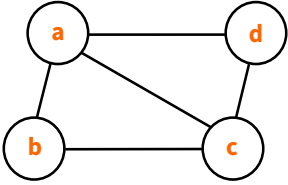
- Graphes :



l.



).



).

- Matrices d'adjacence :

l.

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

).

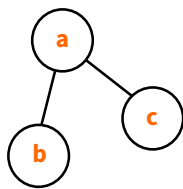
$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

).

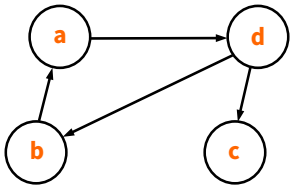
$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

Question 3

- Graphes :



l.



).

- Listes de successeurs :

l.

sommet	a	b	c
successeurs	b,c	a	a

).

sommet	a	b	c	d
successeurs	d	a	a	

- Listes de prédécesseurs :

l.

sommet	a	b	c
prédécesseurs	b,c	a	a

).

sommet	a	b	c	d
prédécesseurs	b	d	d	a

# Opérations sur les graphes

- [Retour aux exercices sur les graphes](#)
- [Accès aux solutions](#)

L'objectif de cet exercice est d'étendre le `TADGraphe` simple avec de nouvelles opérations dont vous proposerez une implémentation pour les deux représentations vues en cours : les matrices d'adjacence ([code](#)) et les listes de successeurs ([code](#)). Pour plus de simplicité, on considérera des graphes **non-orientés**.

Tout d'abord, on souhaite pouvoir calculer le degré d'un sommet du graphe :

```
degre : Graphe * Sommet -> int
# à partir d'un graphe simple G et d'un sommet s, produit un entier indiquant le degré de s
```

À partir de cette opération, on veut ajouter le prédicat :

```
est_isole : Graphe * Sommet -> bool
# à partir d'un graphe simple G et d'un sommet s, produit un booléen indiquant si s est isolé
```

On souhaite maintenant calculer le degré moyen des sommets du graphe :

```
degre_moyen : Graphe -> int
# à partir d'un graphe simple G, produit un entier indiquant le degré moyen des sommets de G
```

Enfin, on veut ajouter le prédicat :

```
sont_voisins : Graphe * Sommet * Sommet -> bool
# à partir d'un graphe simple G et de deux sommets s1 et s2,
# produit un booléen indiquant si s1 est voisin de s2
```

# Opérations sur les graphes - solutions

- [Retour aux exercices sur les graphes](#)
- [Retour à l'exercice](#)

## Solutions avec les matrices d'adjacence

```
def degre(graphe, sommet):  
    i = sommets(graphe).index(sommet)  
    c = 0  
    for j in range(len(sommets(graphe))):  
        c += graphe[1][i][j]  
    return len(voisins(graphe, sommet))  
  
def est_isole(graphe, sommet):  
    return degre(graphe, sommet) == 0  
  
def degre_moyen(graphe):  
    c = 0  
    for s in sommets(graphe):  
        c += degre(graphe, s)  
    return c/len(sommets(graphe))  
  
def sont_voisins(graphe, s1, s2):  
    i = sommets(graphe).index(s1)  
    j = sommets(graphe).index(s2)  
    return graphe[1][i][j] == 1
```

## Solutions avec les listes de successeurs

```
import functools  
import operator  
  
def degre(graphe, sommet):  
    return len(voisins(graphe, sommet))  
  
def est_isole(graphe, sommet):  
    return degre(graphe, sommet) == 0  
  
def degre_moyen(graphe):  
    S = sommets(graphe)  
    # c = 0  
    # for s in S:  
    #     c += degre(graphe, s)  
    # return c / len(S)  
    return functools.reduce(operator.add, [ degre(graphe,s) for s in S ]) / len(S)  
  
def sont_voisins(graphe, s1, s2):
```

```
return s2 in voisins(graphe, s1)
```

# Passage d'une représentation à l'autre

- [Retour aux exercices sur les graphes](#)
- [Accès aux solutions](#)

Avec le type abstrait `Graphe simple` défini en cours, passer d'une représentation à l'autre consiste à énumérer les sommets et les voisins depuis une représentation tout en construisant l'autre représentation.

Écrivez deux fonctions Python permettant respectivement de passer des listes de successeurs à une matrice d'adjacence et réciproquement.

*Remarque* : vous pouvez utiliser la programmation modulaire pour encapsuler les deux représentations puis créer un module effectuant ses conversions.



# Passage d'une représentation à l'autre - solutions

- [Retour aux exercices sur les graphes](#)
- [Retour à l'exercice](#)

Les deux représentations étant implémentées dans leur module respectif :

```
import graphe_listes as gl
import graphe_matrice as gm
```

- Passage des listes de successeurs à une matrice d'adjacence

```
def gl2gm(graphe_listes):
    graphe_matrice = gm.creer_graphe(gl.sommets(graphe_listes))
    for s1 in gl.sommets(graphe_listes):
        for s2 in gl.voisins(graphe_listes, s1):
            gm.ajouter_arete(graphe_matrice, s1, s2)
    return graphe_matrice
```

- Passage d'une matrice d'adjacence à des listes de successeurs

```
def gm2gl(graphe_matrice):
    graphe_listes = gl.creer_graphe(gm.sommets(graphe_matrice))
    for s1 in gm.sommets(graphe_matrice):
        print(gm.voisins(graphe_matrice, s1))
        for s2 in gm.voisins(graphe_matrice, s1):
            gl.ajouter_arete(graphe_listes, s1, s2)
    return graphe_listes
```