Introduction aux constructions du langage Python

Objectif de la séance : Prise en main des premiers éléments de syntaxe Python (types de données prédéfinis, structures de contrôle, ...) et premiers exercices de programmation.
Les éléments de base du langage Python sont disponibles dans le formulaire *Langage Python*.

Les exercices sont de difficulté variée :

- les purs débutants (à la fois en programmation et en langage Python) sont invités à aborder les exercices dans l'ordre ;
- ceux qui possèdent des compétences en programmation impérative (C ou Java) peuvent diagonaliser les exercices et se concentrer sur les éléments de syntaxe spécifiques à Python ;
- enfin, pour ceux qui connaissent déjà la programmation en Python, quelques exercices plus avancés permettront de se remettre en jambes.

A titre indicatif, le niveau de difficulté des exercices sera indiqué à l'aide des pictogrammes suivants :

😎	Application directe du cours
😀	Facile
😖	Moyen
💀	Difficile

La correction de certains exercices est déjà fournie afin de rappeler quelques éléments de syntaxe Python et des concepts de programmation. Vous êtes invités à en prendre connaissance (et selon le cas, à les refaire par vous-même). Ces corrections vous permettront d'aborder les exercices suivants.

Exercice 1 : Entrées/sorties simples 😎

On s'intéresse aux solutions d'une équation de la forme $ax^2 + bx + c = 0$. Indiquez les solutions quelles que soient les valeurs de a, b et c entrées par l'utilisateur.
Indication : les instructions `input()` et `print()` permettent respectivement de demander à l'utilisateur de saisir une chaîne de caractères et d'en afficher une.

Afin de pouvoir afficher les solutions des équations du second degré, il faut demander à l'utilisateur de fournir 3 réels a, b, et c. Par exemple, pour a, on va utiliser la fonction `input` comme ceci :

```
1 a = float(input("Entrez a: "))
```

Le paramètre passé à la fonction `input` est la chaîne affichée à l'utilisateur pour l'informer de ce qu'il doit entrer. Notons que puisque la saisie des données utilisateur par la fonction `input` renvoie une chaîne de caractères, on utilise la fonction `float` pour convertir la chaîne de caractères saisie en un réel (type float).
Une fois les valeurs des coefficients récupérées, il faut alors distinguer un certain nombre de cas :
Si a est nul, il faut se ramener à une équation du premier degré. On a alors 3 cas possibles :

- soit b et c sont nuls et alors il y a une infinité de solutions ;
- soit b est nul et c n'est pas nul et alors il n'y a pas de solution ;
- soit b n'est pas nul (et peu importe la valeur de c) et alors la solution s'écrit $\frac{-c}{b}$.

Pour distinguer ces 3 cas, il suffira d'imbriquer des instructions `if` et `else`.
Dans le cas où a n'est pas nul, il faut calculer le discriminant $\Delta = b^2 - 4ac$. Nous allons utiliser une variable intermédiaire `delta` pour stocker sa valeur, que nous allons utiliser plusieurs fois pour faire l'étude de cas :

- si $\Delta > 0$, on a deux solutions réelles,
- si $\Delta = 0$, on a deux solutions conjuguées,

- si $\Delta < 0$, on a deux solutions complexes.

Le calcul des solutions nécessite d'utiliser la racine carrée. En Python, on utilise la fonction `sqrt` pour cela. Elle se trouve dans le module `math` de Python : votre script doit donc commencer par l'import de ce module pour pouvoir l'utiliser :

```
1 from math import sqrt
```

Ce qui signifie qu'on importe uniquement la fonction `sqrt` du module `math` pour pouvoir l'utiliser dans notre script (on n'importe pas les autres).
La difficulté est alors d'afficher les solutions complexes. Elles sont constituées d'une partie réelle et d'une partie imaginaire. La fonction `print` permet d'afficher plusieurs choses à la fois. On va donc afficher la partie réelle, puis le signe + ou - et enfin la partie imaginaire.
Le programme s'écrit :

```
1  from math import sqrt
2
3  a = float(input("Entrez a : "))
4  b = float(input("Entrez b : "))
5  c = float(input("Entrez c : "))
6
7  if a == 0:
8      # equation du 1er degre
9      if b == 0:
10         if c == 0:
11             print("il y a une infinite de solutions")
12         else:
13             print("il n'y a pas de solution")
14     else:
15         print("La solution est ", -c/b)
16 else:
17     # equation du second degre
18     delta = b*b - 4*a*c
19     if delta == 0:
20         print("La solution est ", -b / (2*a))
21     elif delta < 0:
22         print("2 solutions complexes:", -b/(2*a), "+/- i*", sqrt(-delta)/(2*a))
23     else:
24         print("2 solutions reelles:", (-b-sqrt(delta))/(2*a),(-b+sqrt(delta))
       /(2*a))
```

Remarquez l'importance de l'indentation qui permet de marquer que certaines instructions sont à exécuter seulement si la condition d'un `if` est vraie ou seulement si elle est fausse (après `else`).L'indentation est donc capitale en Python : au-delà de rendre le code plus lisible, elle permet d'identifier clairement des blocs d'instructions.
Remarquez aussi l'emploi du simple égal `=` pour l'affectation d'une valeur à une variable et l'emploi du double égal `==` pour le test d'égalité de deux valeurs (au sein de la conditionnelle `if ...: else: ...`).

Exercice 2 : Crédit automobile

Dans cet exercice, nous allons réaliser un simulateur de crédit automobile.

1. Écrire une fonction `saisie` qui permet à l'utilisateur de saisir les revenus mensuels de son foyer, le prix de la voiture, le taux du crédit (en %), le nombre de mois du crédit et les retourne sous la forme d'un tuple ;

Pour traiter cette question, il faut d'abord se demander quels seront les paramètres d'entrée (les arguments) de la fonction. Pour demander à l'utilisateur de saisir les données, la fonction n'a besoin d'aucun argument.
Ensuite, nous allons faire appel plusieurs fois à la méthode `input()`, en se rappelant de bien convertir son résultat en réel ou en entier relatif avec respectivement les fonctions `float()` et `int()`.

```
1  def saisie():
2      """Retourne toutes les variables necessaires
3         au calcul de la mensualite"""
4      revenus = float(input("Quels sont vos revenus ?"))
5      prix = float(input("Quel est le prix du bien ?"))
6      taux = float(input("Taux du credit en % ?"))
7      duree = int(input("Duree du credit (en mois) ?"))
8      #t est un pourcentage, il faut le convertir
9      taux = taux / 100
10
11     return revenus, prix, taux, duree
```

Le seul piège de cette fonction est qu'étant donné que le taux est saisi en %, il faut le convertir en le divisant par 100 pour qu'il soit exploitable dans la formule.
Notez bien l'indentation, c'est-à-dire les espaces devant les instructions en dessous de la première ligne : l'indentation marque le fait que toutes ces instructions font partie de la fonction. L'indentation est généralement nécessaire après les ":" qui marquent le début d'un sous bloc d'instruction.
Enfin, remarquez le `return` qui indique que la fonction doit retourner une valeur. Cette valeur est bien un tuple malgré l'absence des parenthèses. On aurait pu écrire également :

`return (revenus, prix, taux, duree).`

Le code inclut des commentaires qui en facilitent la lecture (afin d'être en mesure de corriger, actualiser, modifier, réutiliser le code). Lorsqu'il tient sur une ligne, il débute par le caractère dièse #. Lorsqu'il tient sur plusieurs lignes, le commentaire est encadré par des triples double quote """. Il est conseillé d'incorporer un tel commentaire à chaque déclaration de fonction (dans un contexte professionnel, ces commentaires sont en anglais, sobres et explicitant le rôle de la fonction concernée).

2. Écrire une fonction `mensu` qui calcule le montant d'une mensualité m à partir de la formule suivante :

$$m = \frac{P \times \frac{t}{12}}{1 - \left(1 + \frac{t}{12}\right)^{-n}} \tag{1}$$

où P est le prix de la voiture, n le nombre de mois et t le taux du crédit ;

Indication : En Python, on utilise l'opérateur infixe `**` pour élever à la puissance.

La formule donnée nécessite de connaître les variables P (prix), t (taux) et n (nombre de mois) afin de déterminer le montant des mensualités. Cela nous permet d'identifier très clairement quels seront les paramètres de notre fonction. Puisqu'on attend de la fonction qu'elle retourne une valeur, elle doit se terminer par le mot clé `return` suivi de la valeur qui doit être retournée. Pour le reste, la fonction ne présente pas de difficulté.

```
1  def mensu(prix, taux, duree):
2      """Calcule le montant d'une mensualite"""
3      return (prix*taux/12)/(1 - (1+taux/12)**-duree)
```

N'oubliez pas l'indentation avant le return pour marquer le fait que cette instruction fait partie de la fonction `mensu`.

A savoir : Exécution d'un module principal

Il y a deux manières d'exécuter le contenu d'un script Python. Soit il est directement exécuté par l'utilisateur (module principal), soit il est indirectement exécuté lorsqu'il est utilisé par un autre (lorsque la directive `import` est rencontrée).
Par défaut, tout le contenu présent dans le bloc global (tout ce qui n'est pas dans une fonction) est exécuté. Cependant il est souvent désirable d'exécuter certaines instructions seulement lorsque le script est exécuté directement.
Pour cela, on peut tester la valeur de la variable `__name__`, qui vaut `__main__` seulement dans ce cas. Ainsi le test

```
1 if __name__== "__main__":
```

permet de marquer le sous bloc associé comme devant être exécuté seulement dans le cadre d'un module principal, et non d'un module importé.
En général, tout module contient une série de tests délimités par cette instruction, pour que les tests ne soient pas exécutés lors d'une importation mais pour pouvoir rapidement tester le bon fonctionnement d'un module, par exemple lorsqu'on modifie le code de ce dernier. Ces tests consistent souvent à appeler les fonctions du module avec des valeurs particulières et à comparer le comportement de la fonction avec celui attendu.

3. Écrire un module principal qui articule les méthodes précédentes pour demander la saisie des informations, afficher la mensualité et une indication comme quoi le dossier a reçu un avis favorable de la banque ou non (la banque accepte un crédit auto si et seulement si la mensualité ne représente pas plus de 15% des revenus du foyer).

Le programme principal doit dans un premier temps faire appel à la fonction de saisie afin de récupérer les différentes valeurs nécessaires à la suite. Cette fonction retourne un résultat qu'il faut stocker dans une variable. Or le retour est un tuple composé de 4 éléments : les revenus, le prix, le taux et la durée que l'on stockera respectivement dans les variables r, p, t et d. Python offre différentes syntaxes pour récupérer les éléments d'un tuple :

- soit on récupère le tuple dans une seule variable et on accède aux différents éléments avec les `[ ]` :

```
1 resultat = saisie()
2 r = resultat[0]
3 p = resultat[1]
4 t = resultat[2]
5 d = resultat[3]
6
```

- soit on récupère élément par élément le tuple :

```
1 r, p, t, d = resultat
2
```

  Dans ce cas, il faut autant de variables que d'éléments dans le tuple.

Ensuite, il suffit de faire appel à la fonction `mensu` en lui passant les arguments appropriés. Comme celle-ci retourne aussi une valeur, on va la stocker dans une variable temporaire m.
Pour afficher le message concernant l'avis de la banque, il suffit d'utiliser un `if` avec un `else`. Enfin, on délimite le tout par le test faisant de lui un module principal.

```
1 if __name__== "__main__":
2     r, p, t, d = saisie()
3     m = mensu(p, t, d)
```

```
    print("La mensualite du credit est de ", m)

    if m < 15 * r / 100:
        print("Le dossier recoit un avis favorable")
    else:
        print("Le dossier recoit un avis defavorable")
```

Exercice 3 : Autour des triangles

1. Écrire une fonction qui étant donnés 3 nombres représentant les longueurs des côtés d'un triangle, caractérise ce triangle. On en écrira plusieurs versions :
   - la première affichera une simple chaîne de caractères indiquant le statut du triangle `'isocèle'`, `'équilatéral'`, ou `'scalène'` ;

Avant d'afficher le type de triangle formé par les 3 côtés donnés, il faut vérifier que ces 3 nombres peuvent représenter les longueurs des côtés d'un triangle bien formé : cela est exclu si l'un des nombres n'est pas strictement positif, si les trois nombres ne respectent pas l'inégalité triangulaire ou enfin si le triangle formé est plat.
Une fois ces cas exclus, il suffit de vérifier les propriétés des triangles isocèles ou équilatéraux. Pour rappel, si on nomme a, b et c ces trois nombres, on peut formaliser ces propriétés comme suit :

- un des nombres est négatif ou nul : $a \leq 0$ ou $b \leq 0$ ou $c \leq 0$;
- respect de l'inégalité triangulaire : $(a \leq b + c)$ et $(b \leq a + c)$ et $(c \leq a + b)$ (il faudra donc prendre la négation) ;
- triangle plat : $(a = b + c)$ ou $(b = a + c)$ ou $(c = a + b)$;
- triangle équilatéral : $a = b = c$;
- triangle isocèle : $a = b$ ou $b = c$ ou $c = a$.

Attention car un triangle qui est équilatéral est aussi un triangle isocèle : il faudra donc d'abord tester si le triangle est équilatéral puis s'il est isocèle. En effet, si le triangle est équilatéral et qu'on teste d'abord s'il est isocèle, on va seulement afficher qu'il est isocèle.
Cela veut dire plus généralement que l'ordre dans lequel on enchaîne les `if`, `elif` et `else` peut avoir un impact sur le résultat, et que lorsqu'on teste plusieurs cas, il faut tester les cas les plus spécifiques en premiers et les plus généraux en dernier.
La fonction s'écrit donc :

```
def is_triangle(a,b,c):
    if (a <= 0) or (b <= 0) or (c <= 0):
        print('edges of triangle should be strictly positive')
    elif (a > b+c) or (b > a+c) or (c > a+b):
        print('longest edge is greater than to the sum of the 2 other
    edges')
    elif (a == b+c) or (b == a+c) or (c == a+b):
        print ('flat triangle')
    elif (a == b) and (b == c):
        print ('equilateral triangle')
    elif (a == b) or (b == c) or (a == c):
        print ('isocele triangle')
    else:
```

```
13          print('scalene triangle')
```

- la seconde indiquera de plus les longueurs concernées, par exemple `'isocèle de côté 4'`, ainsi que des justifications en cas d'erreur. Deux versions sont demandées, en utilisant les mécanismes de substitution des chaînes de caractères (`%s`) et sans les utiliser ;

Il suffit de reprendre la fonction précédente en ajoutant quelques justifications :

- dans le cas d'un triangle invalide, on affichera la taille invalide d'un côté ;
- dans le cas d'un triangle plat, on pourra afficher le côté le plus long ;
- dans le cas d'un triangle isocèle, on affichera la taille des deux côtés isocèles ;
- dans le cas d'un triangle équilatéral, il suffira d'afficher la taille d'un côté.

On va donc se doter de quelques fonctions pour nous aider à fournir ces informations. Dans un premier temps, nous allons écrire une fonction qui va prendre en paramètre les trois longueurs a, b et c et qui va nous indiquer laquelle est négative, en supposant que l'une d'elles le soit. Pour cela, il suffit de tester chacun des arguments.

```
1  def search_a_neg(a,b,c):
2      """ returns the negative or null argument under the hypothesis that
3  at least one of them is negative or null """
4      if a <= 0 :
5          return a
6      elif b <= 0 :
7          return b
8      else:
9          return c
```

Nous allons écrire une fonction qui retournera la longueur maximale parmi les trois en procédant encore une fois par comparaison. Par exemple, si a est la longueur maximale, alors a est supérieur ou égal à b et c, et on procède ainsi pour les trois arguments.

```
1  def maxi(a,b,c):
2      """Returns the max among a, b, c"""
3      if a >= b and a >= c:
4          return a
5      elif b >= a and b >= c:
6          return b
7      else:
8          return c
```

Enfin, il nous faut une fonction qui parmi les trois longueurs de côté nous retourne la longueur des côtés isocèles. Il suffit de tester l'égalité deux à deux des paramètres.

```
1  def extract_the_same(a,b,c):
2      """ returns the value repeated twice among arguments
3  provided that at least two of the arguments are equal """
4
5      if a == b :
6          return a
7      elif a == c :
8          return a
9      else:
10         return b
```

On peut donc à présent rédiger la fonction demandée à partir de la question précédente et des fonctions que nous venons d'écrire.

```
def is_triangle_info(a,b,c):
    if (a <= 0) or (b <= 0) or (c <= 0):
        print(search_a_neg(a,b,c),'cannot be an edge of a triangle')
    elif (a > b+c) or (b > a+c) or (c > a+b):
        print('longest edge is greater than the sum of the 2 other
    edges')
    elif (a == b+c) or (b == a+c) or (c == a+b):
        print ('flat triangle with: ',maxi(a,b,c),' as longest edge')
    elif (a == b) and (b == c):
        print ('equilateral triangle of edge :',a)
    elif (a == b) or (b == c) or (a == c ):
        print ('isosceles triangle with ',extract_the_same(a,b,c),' as
    the common size of 2 edges')
    else:
        print('scalene triangle')
```

A présent nous allons réécrire cette fonction en utilisant les mécanismes de substitution des chaînes. Au passage, nous allons améliorer l'affichage de l'erreur en cas de côté de longueur négative ou nulle. En effet, la fonction précédente ne tient pas compte du cas où plusieurs longueurs sont invalides. Dans ce cas, elle n'affiche que la première longueur négative ou nulle. Réécrivons pour cela une fonction qui retourne tous les arguments négatifs parmi a, b et c. Pour cela, on va utiliser un tuple (séquence non mutable) et stocker dans ce tuple les valeurs négatives ou nulles. Profitez de cette fonction pour regarder comment on initialise un tuple vide et comment on y ajoute des éléments.

```
def search_all_neg(a,b,c):
    """ returns all the negative or null arguments under the hypothesis
    that at least one of them is negative or null """
    res =()
    if a <= 0 :
        res = res + (a,)
    if b <= 0 :
        res = res + (b,)
    if c <= 0:
        res = res + (c,)
    return res
```

Notez que la syntaxe `(a,)` permet de créer un tuple à un seul élément. En effet, malgré les parenthèses, `(a)` n'est pas un tuple mais un réel (puisque `a` est un réel et donc les parenthèses sont interprétées comme celles d'une expression arithmétique classique).
En fonction du nombre d'éléments retournés par cette fonction, on modifiera le message. La substitution des chaînes de caractères se fait de la façon suivante. Il suffit de prendre une chaîne de caractères dans laquelle le symbole `%s` apparaît pour marquer la position des substitutions. Cette chaîne de caractères doit être suivie immédiatement du symbole `%` suivi lui-même par un tuple. Chaque symbole `%s` dans la chaîne de caractères est alors remplacé par un élément du tuple (dans l'ordre d'apparition dans le tuple). Attention, la longueur du tuple doit être égale au nombre de `%s` dans la chaîne.
Ainsi la fonction s'écrit :

```
def is_triangle_string_sub(a,b,c):
    if (a <= 0) or (b <= 0) or (c <= 0):
        neg = search_all_neg(a,b,c)
        if len(neg) == 3:
            print('%s, %s and %s cannot be edges of a triangle'%neg)
        elif len(neg) == 2:
            print('%s and %s cannot be edges of a triangle'%neg)
        else:
```

```
            print('%s cannot be an edge of a triangle'%neg)
    elif (a > b+c) or (b > a+c) or (c > a+b):
        print('longest edge %s should be less or equal to the sum of
    the 2 other edges %s and %s'% search_max(a,b,c))
    elif (a == b+c) or (b == a+c) or (c == a+b):
        print ('flat triangle with %s as longest edge'% maxi(a,b,c))
    elif (a == b) and (b == c):
        print ('equilateral triangle of edge %s' % a)
    elif (a == b) or (b == c) or (a == c ):
        print ('isosceles triangle with %s as the common size of 2
    edges'% extract_the_same(a,b,c))
    else:
        print('scalene triangle')
```

- la troisième renvoie un booléen indiquant si les 3 nombres peuvent représenter ou non les 3 côtés d'un triangle, sans rien afficher ;

Pour cette fonction, il suffit de reprendre les deux premiers tests des fonctions précédentes. L'avantage d'une telle fonction est qu'elle retourne un résultat qui peut être exploité par la suite par une autre fonction ou le programme principal.

```
def is_triangle_Bool(a,b,c):
    if ((a <= 0) or (b <= 0) or (c <= 0)) or ((a > b+c) or (b > a+c) or
     (c > a+b)):
        return False
    else:
        return True
```

Étant donné que lorsque la condition est vraie on retourne `False` et que lorsque la condition est fausse on retourne `True`, il est plus élégant d'écrire la fonction sous cette forme :

```
def is_triangle_Bool(a,b,c):
    return not (((a <= 0) or (b <= 0) or (c <= 0)) or ((a > b+c) or (b
    > a+c) or (c > a+b)))
```

2. Écrire une fonction qui étant donnés trois nombres calcule la surface du triangle correspondant (s'il existe). Rappel : la surface d'un triangle de côtés a, b et c, vaut $\sqrt{s(s-a)(s-b)(s-c)}$ avec $s = \frac{1}{2}(a+b+c)$ (formule de Héron).

Il n'y a pas de difficulté dans l'implémentation de cette fonction. Nous allons réutiliser la fonction précédente pour tester si les arguments sont corrects et afficher un message d'erreur dans le cas contraire.
Pour le calcul de $\sqrt{x}$, vous pouvez utiliser l'opérateur `**` avec `x**0.5` ou alors utiliser la fonction `sqrt()` précédemment introduite. Les deux méthodes, quel que soit leur argument, retournent un réel.

```
from math import *

def surface_triangle(a,b,c):
    if not is_triangle_Bool(a,b,c):
        print('this cannot be considered as a triangle')
    else:
        s = (a+b+c) / 2
        return sqrt(s * (s-a) * (s-b) * (s-c))
```

Faites bien la différence entre l'opérateur / et l'opérateur //. Le premier retourne le résultat de la division réelle sous la forme d'un réel (même si le résultat pourrait être un entier). Par exemple, 2/3 donne 0.6666666666666666 et 4/2 donne 2.0. Le second opérateur retourne la valeur du quotient (division euclidienne). Dans le cas où les opérandes sont des entiers, le quotient est donné sous la forme d'un entier ; en revanche, si au moins un des opérandes est un réel, on a alors la partie entière de la division réelle sous la forme d'un réel. Par exemple, 3.5//2 donne 1.0 et 4/2 donne 2.

3. Écrire une fonction qui demande à l'utilisateur trois valeurs et écrit à l'écran la surface du triangle correspondant s'il existe, et sinon redemande à l'utilisateur trois nouvelles valeurs.

Il s'agit de demander les longueurs avec la fonction `input()`, qui retourne une chaîne de caractères correspondant à ce qui a été saisi. Il ne faut donc pas oublier de convertir en réel avec la fonction `float()`. Si le triangle n'est pas correct (on testera avec la fonction `is_triangle_Bool`), nous allons demander à l'utilisateur s'il veut ressaisir des longueurs. Pour cela, on va simplement utiliser `input()` en demandant à l'utilisateur d'écrire `"true"` ou bien n'importe quoi d'autre. Dans le premier cas on appelle à nouveau la fonction et dans le deuxième on arrête l'exécution.

```
1  def entry_triangle():
2      print('Give 3 values representing edge candidates of a triangle')
3      a = input('First value: ')
4      b = input('Second value: ')
5      c = input ('Third value: ')
6      #conversions
7      a = float(a)
8      b = float(b)
9      c = float(c)
10     if is_triangle_Bool(a,b,c):
11         is_triangle_info(a, b, c)
12         print('The surface of the triangle is %.2f' % surface_triangle(a,b,
       c))
13     else:
14         print('Values do not represent a triangle. Continue (true/false)?')
15         if input() == 'true':
16             entry_triangle()
17         else:
18             print('Bye')
```

La fonction `entry_triangle` est récursive car elle fait appel à elle-même : l'arrêt de la fonction est conditionnée par la saisie d'une autre chaîne de caractère que `"True"`.

4. Écrire une fonction qui calcule une ligne du triangle de Pascal (par récurrence, en calculant la ligne n à partir de la ligne $n-1$).

Dans cette question, il faut faire attention à l'énoncé. En effet, le mot "récurrence" apparaît mais ne s'applique pas à la fonction que l'on doit fournir, mais à la méthode que l'on doit appliquer. En fait, il faut faire une fonction (itérative ou récursive mais on va préférer le choix le plus simple : itérative) qui construit une ligne du triangle de Pascal à partir de la ligne précédente. On va représenter une ligne comme une liste d'entiers.
Pour cela, nous allons utiliser une petite astuce : soit l la ligne précédente, on va ajouter un 0 au début et à la fin de l. Il suffira alors d'additionner deux cases consécutives pour obtenir la ligne suivante. Par exemple, si l vaut `[1, 1]`, on va la transformer en `[0, 1, 1, 0]`, puis additionner deux à deux les cases `[0+1, 1+1, 1+0]` soit `[1, 2, 1]`.

Pour ajouter un élément à une liste, on peut utiliser la concaténation de listes, effectuée par l'opérateur `+`. `l1 + l2` retourne une liste contenant les éléments de `l1` puis ceux de `l2`.
L'erreur classique lorsqu'on veut rajouter un élément en extrémité de liste (par exemple 0 à la fin de l) est d'écrire `l+0`, ce qui provoque une erreur de typage. En effet, la concaténation ne peut se faire qu'entre deux listes et 0 n'est pas une liste. Il faut donc créer une liste qui ne contiendra que 0, c'est-à-dire `[0]`. Il faut donc écrire `l+[0]`.
Pour ne pas compliquer inutilement la fonction, nous n'allons pas faire de vérification particulière sur l. Elle sera correctement appelée dans la suite.
Pour enlever un élément d'une liste, on peut utiliser le slicing.

A savoir : Slicing de listes

En Python, lorsqu'on a une séquence, on peut en récupérer une partie seulement, c'est le slicing (découpage). Soit s une séquence (i.e. une liste, un tuple ou une chaîne de caractères) :

- `s[i:j]` : retourne les éléments situés entre l'indice i (inclus) et l'indice j (exclu) ;
- `s[i:j:p]` : retourne les éléments situés entre l'indice i (inclus) et l'indice j (exclu), en sautant p éléments.

Par défaut, i vaut 0, j vaut `len(s)` et p vaut 1. Ainsi, `s[:j]` retourne les éléments à partir du début de s jusqu'à l'élément à l'indice j (exclu), `s[i:]` retourne les éléments à partir de l'indice i (inclus) jusqu'à la fin de s. Si les indices ne sont pas corrects lors du slicing, le résultat est une séquence vide de même type que s.

Donc pour priver une liste l de son premier élément, il suffit d'écrire l [1:] .

```
1 def build_line_Pascal(l):
2     l = [0] + l + [0]
3     new_line = []
4     while len(l) > 1: #Continue only if there is more than one element left
5         new_line += [l[0] + l[1]] #Appending a coeficient to the new line
6         l = l[1:] #Removing the first element of l
7     return new_line
```

5. Écrire une fonction d'affichage qui affiche les n premières lignes du triangle de Pascal (si possible en centrant les lignes les unes au-dessus des autres). Par exemple, on devra avoir un résultat similaire à :

```
1 >>> affiche_Pascal(8)
2               1
3             1   1
4           1   2   1
5         1   3   3   1
6       1   4   6   4   1
7     1   5   10  10  5   1
8   1   6   15  20  15  6   1
9 1   7   21  35  35  21  7   1
```

Avant de pouvoir afficher le triangle de Pascal, nous allons le calculer à l'aide de la fonction précédente. Pour cela, on va stocker le triangle de Pascal dans une liste. Nous aurons donc une liste (le triangle), qui contiendra des listes (les lignes) qui contiendront elles-mêmes des entiers.

On va donc ajouter chaque nouvelle ligne à la liste principale et faire appel à `build_line_Pascal` sur la dernière liste contenue dans la liste principale. Pour obtenir le dernier élément d'une liste `l`, il suffit d'écrire `l[-1]`.

```
def Pascal_triangle(n):
    if n < 1 or n >= 10:
        print('Pascal_triangle : bad argument')
    else:
        list_lines = [[1]]
        while n > 1:
            current_line = build_line_Pascal(list_lines[-1])
            list_lines = list_lines + [current_line]
            n = n - 1
        return list_lines
```

Comme vous le voyez, on se limite à des triangles de Pascal à 9 lignes. Étant donnée la taille, on sait que les coefficients seront à 1 ou 2 chiffres au maximum. Nous allons créer des petites fonctions pour nous aider à présenter correctement : afficher les nombres avec un espacement correct, afficher des espaces pour réaliser l'effet centré et enfin afficher une ligne entière du triangle.
Commençons par afficher les coefficients sur 3 caractères : on mettra un espace si le nombre est à 2 chiffres mais on mettra deux espaces s'il n'est qu'à 1 chiffre. Pour convertir un chiffre en chaîne de caractères, on peut utiliser la fonction `str()` qui convertit son argument en chaîne de caractères. Il est possible de concaténer des chaînes de caractères entre elles avec l'opérateur `+`.

```
def string_of_int(n):
    if n < 0 or n >= 100:
        print('string_of_int : integer too long or negative')
    elif n < 10:
        return ' ' + str(n) + ' '
    else:
        return ' ' + str(n)
```

Nous allons ensuite créer une fonction qui aideront à afficher une ligne du triangle de Pascal. Elle prendra en argument une liste d'entiers et retournera la chaîne de caractères à afficher et à centrer. Nous allons écrire une fonction récursive. Rappelez-vous que dans la plupart des cas, une fonction récursive doit contenir un branchement conditionnel (`if` ... `else`) afin d'écarter les cas terminaux des appels récursifs. Dans notre cas, le cas terminal se produit lorsque la liste d'entiers passée en argument est vide. Sinon, on génère la chaîne du premier entier de la liste à l'aide de la fonction précédente, à laquelle on concatène le résultat de notre fonction sur la liste privée de son premier élément.

A savoir : Récursion sur des séquences

Très souvent, quand les paramètres sont des séquences (i.e. listes, tuples et chaîne de caractères), le cas terminal d'une fonction récursive est atteint lorsque la séquence est vide.
De même, lorsque la fonction récursive opère sur des séquences, elle traite généralement un des éléments de cette séquence avant de procéder à un appel récursif sur la séquence privée de cet élément (il ne faut pas oublier d'enlever l'élément déjà traité, sinon la séquence ne diminue pas et le cas d'arrêt n'est jamais atteint).

```
def string_of_line(l):
    if l == []:
        return ''
    else:
        return string_of_int(l[0]) + string_of_line(l[1:])
```

A présent, nous allons écrire une fonction qui retourne une chaîne de caractères composée d'un certain nombre d'espaces précisé en argument. Nous allons aussi écrire une fonction récursive. Le cas terminal sera celui où le nombre d'espaces demandé est nul. Sinon on concatène un espace au résultat de l'appel récursif en prenant soin de diminuer le nombre d'espaces souhaité de 1. Dans le cas où on souhaite 0 espace, il faut retourner la chaîne vide.

```
1 def string_decal(i):
2     if i == 0:
3         return ''
4     else:
5         return ' ' + string_decal(i - 1)
```

Enfin, il ne reste plus qu'à créer la fonction principale pour l'affichage du triangle qui va dans un premier temps le calculer puis l'afficher ligne par ligne.
Pour centrer une ligne, il faut connaître sa longueur l_i ainsi que celle de la plus longue des lignes l_m. Alors pour centrer au mieux la ligne, il faudra ajouter $\left\lfloor \frac{l_m - l_i}{2} \right\rfloor$ espaces avant celle-ci.
La longueur de la i-ème ligne est de $3i$ caractères, comme elle contient i coefficients et que chaque coefficient prend toujours 3 caractères (le nombre et un ou deux espaces). La plus longue des lignes est la dernière, de taille $3n$.
Ainsi le nombre d'espaces à ajouter avant la i-ème ligne est de

$$\left\lfloor \frac{3(n-i)}{2} \right\rfloor$$

. On obtient la fonction suivante :

```
1 def affiche_Pascal(n):
2     triangle = Pascal_triangle(n)
3     if triangle != None: #If Pascal_triangle returned something
4         i = 1
5         while i <= n:
6             decal = int(3/2 * (n - i))
7             print(string_decal(decal) + string_of_line(triangle[0]))
8             triangle = triangle[1:]
9             i = i + 1
```

Notez que la boucle :

```
1 i = 1
2 while i <= n:
3     # Instructions
4     i = i + 1
```

est équivalente à :

```
1 for i in range(1, n + 1):
2     # Instructions
```

La fonction `range` retourne une énumération (pas une liste !) de nombres entiers et s'utilise comme suit :

- `range(a)` : retourne l'énumération des nombres entiers dans $[0; a[$, par exemple `range(4)` retourne 0, 1, 2, 3 ;
- `range(a,b)` : retourne l'énumération des nombres entiers dans $[a; b[$, par exemple `range(2, 4)` retourne 2, 3 ;
- `range(a,b,c)` : retourne l'énumération des nombres entiers $a, a + c, \ldots$ dans $[a; b[$, par exemple `range(2, 10, 3)` retourne 2, 5, 8.

La fonction `range` est pratique dans un `for`. Pour obtenir une liste à partir de l'énumération, n'oubliez pas d'utiliser la fonction `list()` qui convertit son argument en liste.

Exercice 4 : Boucles : intégrales 😀

Le but de cet exercice est de comparer trois méthodes d'approximation d'une intégrale. Pour cela, nous allons considérer l'intégrale suivante :

$$\int_0^1 \frac{1}{1+x}\,\mathrm{d}x$$

puisqu'on sait qu'elle vaut $\ln(2)$ qui est calculable en Python à partir du module `math`. Les méthodes que nous allons comparer sont :

- la méthode des rectangles :

$$\int_a^b f(x)\,\mathrm{d}x \approx \Delta \sum_{i=0}^{n-1} f\left(a + i \times \Delta\right)$$

- la méthode des points milieu :

$$\int_a^b f(x)\,\mathrm{d}x \approx \Delta \sum_{i=0}^{n-1} f\left(a + i\Delta + \frac{\Delta}{2}\right)$$

- la méthode des trapèzes :

$$\int_a^b f(x)\,\mathrm{d}x \approx \frac{\Delta}{2} \sum_{i=0}^{n-1} \left(f\left(a + i \times \Delta\right) + f\left(a + (i+1) \times \Delta\right)\right)$$

où $\Delta = \dfrac{b-a}{n}$ et n est le nombre d'itérations de la méthode

Écrire une fonction pour chacune des méthodes d'approximation ci-dessus qui indique le nombre d'itérations nécessaires pour atteindre une précision donnée par l'utilisateur. Utilisez pour cela un méthode dite de force brute, c'est-à-dire qui essaie toutes les valeurs de n dans l'ordre croissant jusqu'à obtenir la bonne précision.

Nous allons écrire plusieurs fonctions : une fonction `f` pour calculer l'image d'un x par $f(x) = \frac{1}{1+x}$, et une fonction par méthode de calcul d'intégrale.
Commençons par la fonction `f` :

```
1 def f(x):
2     return 1 / (1 + x)
```

Pour les fonctions d'approximation, on va écrire des fonctions qui prennent en paramètre deux réels : un premier indiquant la précision avec laquelle on souhaite procéder à l'approximation, et un second pour la cible que l'on souhaite approcher. Ces fonctions devront retourner le nombre d'itération nécessaire à l'atteinte de la précision requise.
Chacune des méthodes nécessite un paramètre n pour contrôler la précision de l'approximation : il suffit de le faire varier en l'incrémentant de 1, et pour chaque valeur de n calculer la somme. Lorsque pour un certain n la précision est atteinte, il suffit alors de le retourner. On peut donc utiliser une boucle `while` pour gérer la variation du n, et tester si la valeur absolue de la différence de la cible et de la somme est supérieure à la précision souhaitée. La valeur absolue est obtenue en Python par la fonction `abs()`.
Les fonctions s'écrivent donc simplement :

```
1 def rectangles(p,c):
2     """ Retourne le nombre d'iterations necessaires a la methode des rectangles
3         pour approximer la cible c avec une precision p entre les bornes 0 et 1
4     """
5     n = 0
6     somme = 0
7     while abs(somme - c) > p:
8         n = n + 1
```

```
        somme = 0
        for i in range(n):
            somme = somme + f(i / n) / n
    return n
```

```
def pointsmilieux(p,c):
    """ Retourne le nombre d'iterations necessaires a la methode des
        points-milieux pour approximer la cible c avec une precision
        p entre les bornes 0 et 1
    """
    n = 0
    somme = 0
    while abs(somme - c) > p:
        n = n + 1
        somme = 0
        for i in range(n):
            somme = somme + f((2 * i + 1)/(2 * n)) / n
    return n
```

```
def trapezes(p,c):
    """ Retourne le nombre d'iterations necessaires a la methode des trapezes
        pour approximer la cible c avec une precision p entre les bornes 0 et 1
    """
    n = 0
    somme = 0
    while abs(somme-c) > p:
        n = n + 1
        somme = 0
        for i in range(n):
            somme = somme + (f(i / n) + f((i + 1) / n)) / (2 * n)
    return n
```

Il faut ensuite écrire un script de test de ces fonctions. Celui-ci va demander à l'utilisateur de saisir une précision p à l'aide de la fonction `input()` que l'on convertira en réel avec la fonction `float()`. Ensuite, on appellera chacune des fonctions précédentes avec la précision et la valeur de $\ln 2$.
Pour obtenir $\ln 2$ en Python, il faut faire appel à la fonction `log` du module `math`. Votre script doit donc comporter l'instruction `from math import log`.

```
precision = float(input("Precision : "))
print("Nb d'iterations pour la meth des rectangles :", rectangles(precision,log
    (2)))
print("Nb d'iterations pour la meth des points-milieux :", pointsmilieux(
    precision,log(2)))
print("Nb d'iterations pour la meth des trapezes :", trapezes(precision,log(2)))
```

Exercice 5 : Manipulation des listes

1. Écrire trois versions d'une fonction `sum` calculant la somme des éléments d'une liste d'entiers (une version récursive, une version itérative avec la structure de contrôle `while` et une version itérative avec la structure de contrôle `for`). Ainsi, `sum([1,4,1])` vaut 6.

Version récursive

Pour la version récursive, nous allons appliquer les conseils donnés dans la section précédente. Ici notre fonction récursive va opérer sur une séquence (une liste), donc on peut en déduire deux choses :

- le cas d'arrêt va être la liste vide ;
- la fonction va traiter un élément de la liste et s'appeler récursivement sur la liste privée de cet élément.

Le traitement du cas d'arrêt est simple : la somme des éléments de la liste vide vaut 0. Pour le cas plus général, on remarque que pour une liste de taille n $[e_1, \dots, e_n]$,

$$\texttt{sum}([e_1, \dots, e_n]) = e_1 + \texttt{sum}([e_2, \dots, e_n])$$

Donc pour l'implémentation, il faut additionner le premier élément de la liste au résultat de l'addition des éléments de la liste privée du premier élément.
Pour vérifier si une liste est vide, on peut utiliser indifféremment `l==[]` ou `len(l)==0`. Enfin pour priver la liste du premier élément, il suffit d'utiliser le slicing.

```
1 def sum(l):
2     """
3     Somme les elements de la liste l
4     type : list -> int
5     """
6     if l == []:
7         return 0
8     else:
9         return l[0] + sum(l[1:])
```

Version avec `while`

Avec la boucle `while`, il faut trouver la condition pour laquelle les itérations vont se poursuivre (on répète tant que la condition est vraie). Cependant, il est souvent plus facile de trouver la condition d'arrêt. Une fois encore on souhaite donc arrêter de parcourir la liste lorsqu'elle est vide. Attention, lorsque des variables interviennent dans la condition d'un `while`, il faut que dans les instructions répétées une des variables au moins soit changée. En effet, si on rentre dans la boucle `while`, c'est que la condition a été évaluée à `True` ; si aucune des variables ne change, la condition restera vraie et la boucle se poursuivra indéfiniment (sauf si un `break` est présent).
Cela veut dire que dans la boucle nous devons modifier la liste l en supprimant à chaque fois un élément, de façon à ce que la condition de la boucle `while` soit fausse à un moment donné.

```
1  def sum(l):
2      """
3      Somme les elements de la liste l
4      type : list -> int
5      """
6      s = 0
7      while len(l) > 0:
8          s = s + l[0]
9          l = l[1:]
10     return s
```

Remarquez encore une fois l'importance de l'indentation qui indique clairement quelles instructions sont à exécuter dans la boucle `while` : elles sont simplement décalées par rapport au mot clé `while` auquel elles se rapportent.
On retrouve le schéma que nous avons déjà vu pour un calcul de somme (ou de produit, suite...) :

- on initialise une variable avec l'élément neutre de l'opération (0 pour une somme, 1 pour un produit, u_0 pour une suite $(u)_n$), avant la boucle ;
- dans une boucle parcourant les élément, on met à jour cette variable en effectuant l'opération entre elle et l'élément parcouru.

Version avec `for`

La boucle `for` est une boucle très pratique pour parcourir des séquences élément par élément. Par rapport à la boucle `while`, il est moins facile de créer une boucle qui ne se termine pas (mais c'est possible, attention !).
Ici on procède de manière similaire à la version précédente mais cette fois-ci sans modifier la liste. Remarquez la syntaxe `for i in l` où `l` est une liste (ou tout autre objet pouvant être parcouru) permettant d'itérer sur cette liste. À chaque itération la variable `x` réfère à l'élément actuellement parcouru.

```
1 def sum(l):
2     """
3     Somme les elements de la liste l
4     type : list -> int
5     """
6     s = 0
7     for x in l:
8         s = x + s
9     return s
```

2. En utilisant la fonction `sum`, écrire une fonction calculant la somme des $n \geq 1$ premiers entiers : $1 + \cdots + n$.

Cette question est rendue facile grâce à la fonction prédéfinie `range` qui permet d'obtenir une énumération de nombres entiers. Attention, il faut que n soit compris dans cette énumération donc il faut écrire `range(1, n + 1)`. Ceci n'est pas une liste et nous avons écrit dans la question précédente une fonction qui était conçue pour une liste. Pour rester homogène, nous allons convertir l'énumération de `range` en une liste à l'aide de la fonction de conversation `list()`.

```
1 def sum_first_numbers(n):
2     """
3     Calcule la somme des n premiers nombres, n >= 1
4     type : int -> int
5     """
6     return sum(list(range(1, n + 1)))
```

Pour simplifier, nous n'avons pas vérifié la validité du paramètre n. Dans le cas où n est inférieur à 1, la fonction range retournera une liste vide (car le slicing est incorrect) et la somme sera égale à 0. La réponse est donc convenable sans avoir à effectuer de vérification supplémentaire.

3. Écrire une fonction récursive `reverse` qui prend comme argument une liste et renvoie une liste constituée des mêmes éléments mais rangés dans l'ordre inverse. Ainsi, `reverse([4,5,8])` vaut `[8,5,4]`.

Cette question est très facile à résoudre en version itérative : êtes-vous capable de l'écrire facilement ? Elle n'est pas très difficile à écrire en récursif non plus. En appliquant l'astuce sur les fonctions récursives qui opèrent sur des séquences, on peut déduire que :

- le cas terminal est atteint lorsque la liste est vide ;
- la fonction doit traiter un élément puis s'appeler récursivement sur la liste privée de cet élément.

Attention, toutefois, cette astuce marche dans beaucoup de cas mais pas dans tous !
Si la liste est vide, son inverse est donc la liste vide. Sinon, il suffit de remarquer que :

$$\texttt{reverse}([e_1, \ldots, e_n]) = \texttt{reverse}([e_2, \ldots, e_n]) + [e_n]$$

Il suffit d'appliquer algorithmiquement ce principe. Par exemple, soit la liste `[1, 2, 3, 4]`. Essayons de dérouler l'algorithme :

- On souhaite calculer `reverse([1, 2, 3, 4])`
- On applique le principe récursif : `reverse([2, 3, 4]) + [1]`
- `reverse([3, 4]) + [2] + [1]`
- `reverse([4]) + [3] + [2] + [1]`
- `reverse([]) + [4] + [3] + [2] + [1]`
- On atteint le cas terminal : `[] + [4] + [3] + [2] + [1]`

Ce qui donne par concaténation : `[4, 3, 2, 1]`.

```
def reverse(l):
    """
    Renverse une liste l.
    type : list -> list
    """
    if l == []:
        return []
    else:
        return reverse(l[1:]) + [l[0]]
```

4. Une liste représente un palindrome si elle égale à sa liste renversée (c'est-à-dire dont les éléments sont rangés dans l'ordre inverse).

   Écrire une fonction `palindrome` qui teste si une liste représente un palindrome. En écrire deux versions, l'une utilisant la fonction `reverse` de la question précédente et l'autre sans, dans un style récursif en utilisant directement les fonctions d'accès aux éléments de la liste une version récursive en utilisant directement les fonctions d'accès aux éléments de la liste à l'aide de leur index

La première fonction est facile à écrire puisque nous pouvons disposer de la chaîne de caractères et de sa version renversée. Il suffit alors de tester l'égalité entre les deux pour savoir s'il s'agit d'un palindrome.
On peut écrire :

```
if reverse(l) == l:
        return True
else:
        return False
```

Cependant, on remarque encore qu'on peut l'écrire plus élégamment en retournant directement la valeur du test d'égalité :

```
return reverse(l) == l
```

La fonction s'écrit donc :

```
def palindrome(l):
    """
    Test si la liste l est un palindrome.
    type : list -> bool
    """
    return reverse(l) == l
```

Pour la version récursive, on va utiliser le fait qu'une liste vide ou une liste d'un seul élément sont des palindromes. Pour les autres, il suffit de tester l'égalité entre le premier et le dernier élément. S'ils sont égaux, il faut poursuivre les comparaisons, sinon ce n'est pas un palindrome. Dans le cas où le nombre d'éléments est pair, le cas terminal sera le cas de la liste vide tandis que si le nombre d'éléments est impair, le cas terminal sera celui d'une liste à un seul élément.
Pour rappel, pour accéder au dernier élément d'une liste non vide, on accède à son index -1.

```
def palindrome(l):
    if len(l) == 0:
        return True
    elif len(l) == 1:
        return True
    elif l[0] == l[-1]:
        #Call on the list without its first and last elements
        return palindrome(l[1:-1])
    else: return False
```

Le cas d'une liste à un seul élément n'est pas obligatoire. En effet, que se passe-t-il si on le supprime ? On va tester la condition `l[0] == l[-1]`, qui sera vraie puisque lorsqu'on a un seul élément, le premier et le dernier élément sont les mêmes. On va donc rappeler la fonction `palindrome` sur la liste privée de son premier et dernier élément, donc la liste vide. Et cela nous ramène au cas terminal de la liste vide.
On peut donc réécrire une fonction plus courte :

```
def palindrome(l):
    if len(l) == 0:
        return True
    elif l[0] == l[-1]:
        return palindrome(l[1:-1])
    else: return False
```

5. Écrire une fonction itérative `sublist` qui prend deux listes en arguments, et renvoie `True` si et seulement si les éléments de la première liste sont tous éléments de la seconde liste, indépendamment de l'ordre.

   Ainsi `sublist([2,3,4],[1,2,4,5,3])` vaut `True`.

Nous allons écrire une fonction itérative, c'est-à-dire qui ne s'appelle pas elle-même mais utilise plutôt des boucles. Pour chaque élément de la première liste, il suffit de vérifier s'il est bien présent dans la seconde. Dès qu'on trouve un élément de la première liste qui n'est pas contenu dans la seconde, on peut répondre `False`. Pour répondre `True`, il faut aller jusqu'au bout des itérations. Pour tester l'appartenance d'un élément à une séquence, nous avons l'opérateur `in`.

```
def sublist(l1,l2):
    """
    Teste si les elements de l1 sont tous dans l2.
    type : list * list -> bool
    """
    for e1 in l1:
        if not (e1 in l2):
            return False
    return True
```

Notez que lorsque l'interpréteur Python arrive sur un `return` il sort de la fonction, interrompant au passage les boucles dans lesquelles il peut se trouver.

6. Écrire une fonction récursive `allindex` qui prend en argument une valeur `e` et une liste `l` et renvoie la liste des indices i pour lesquels `l[i] == e`. On retournera des indices partant de 1.

Ainsi, `allindex(3,[2,2,1,3,3,5,8])` vaut `[4,5]`.

Cet exercice est un peu plus difficile. Il va falloir faire attention au fait que les indices que l'on doit retourner commencent à 1 au lieu de 0 comme en Python. De plus, alors qu'on va à chaque étape de la récursion enlever le premier élément de la liste, il faut se souvenir de l'indice qu'il occupait dans la liste de départ.
Si on crée une fonction qui a la même signature que l'énoncé, on n'arrivera pas à préciser à quelle indice de la liste de départ se trouve le premier élément de la liste courante. Il faut donc passer par une méthode auxiliaire, récursive, qui aura un paramètre supplémentaire. Ce paramètre est un entier que l'on appellera `r` et correspondra à l'indice dans la liste initiale du premier élément de la liste courante. Ainsi, au premier appel, ce paramètre vaudra 1 : en effet, la liste n'aura pas encore été tronquée et le premier élément sera donc bien à l'indice 1.
Ensuite on va appliquer l'astuce sur les fonctions récursives et les séquences :

- le cas terminal est celui où on a une liste vide (peu importe les valeurs des autres paramètres) ;
- pour une liste non vide, on appelle récursivement la fonction sur la liste privée de son premier élément.

Si la liste est vide, alors l'élément `e` ne peut pas y apparaître : on peut ainsi retourner la liste vide. Si la liste n'est pas vide, il faut regarder si son premier élément est l'élément recherché `e`. Si c'est le cas, il faut constituer la solution : il faut ajouter l'indice de cet élément, c'est-à-dire `r`, en tête de la liste retournée par l'appel récursif. Sinon, si le premier élément de la liste n'est pas l'élément recherché, on retourne simplement le résultat de l'appel récursif.

```
def index_aux(e,l,r):
    """
    Calcule la liste des occurences de e dans l en prenant r comme premiere
    valeur d'indice
    type : int * list * int -> list
    """
    if l == []:
        return []
    elif l[0] == e:
        return [r] + index_aux(e,l[1:],r + 1)
    else:
        return index_aux(e,l[1:],r + 1)
```

Cette fonction ne répond pas tout à fait à l'énoncé. En effet, l'énoncé impose une certaine signature pour la fonction demandée, à savoir une fonction qui prend en argument un entier et une liste et retourne une liste. Nous allons donc créer une fonction qui va appeler notre fonction auxiliaire en attribuant une valeur au paramètre `r` lors du premier appel.

```
def allindex(e,l):
    """
    Calcule la liste des occurences de e dans l.
    type : int * list -> list
    """
    return index_aux(e,l,1)
```

Il est important de bien comprendre que si un énoncé nous impose une signature de fonction (c'est-à-dire la liste des paramètres et leurs types, et le type de retour), il faut la respecter. Cependant il est tout à fait possible de faire appel à une (ou éventuellement plusieurs) fonction(s) auxiliaire(s).

7. Écrire une fonction récursive `separate` qui prend deux arguments, un élément `e` et une liste `l`, et renvoie un couple constitué de la liste des éléments de `l` supérieurs à `e`, et de la liste des éléments strictement inférieurs à `e`.

   Ainsi `separate(3,[2,5,1,7,0,3])` vaut `([3, 7, 5], [0, 1, 2])`.

L'élément e qui permet de couper une liste d'éléments en deux listes contenant respectivement les éléments inférieurs à e et supérieurs ou égaux à e, est appelé *pivot*.
Vous devriez pouvoir écrire une fonction itérative sans aucune difficulté pour résoudre ce problème (essayez !). Cependant c'est une fonction récursive qui est demandée. On va écrire dans un premier temps une fonction récursive qui va imiter le comportement de la version itérative qu'on peut imaginer. On va écrire une fonction qui prend en paramètre l'élément e, la liste l, et les listes $l1$ et $l2$. Il s'agit donc d'une fonction temporaire puisqu'elle ne correspond pas à la signature demandée dans l'énoncé. L'idée va être de peupler les listes $l1$ et $l2$ avec respectivement les éléments de l supérieurs ou égaux à e et inférieurs à e. Au premier appel, les listes $l1$ et $l2$ doivent être vides.
Lorsque la liste l est vide, alors tous les éléments qu'elle contenait au départ ont été répartis dans $l1$ ou $l2$: on peut donc retourner ces deux listes. Sinon, en fonction de la comparaison de e et `l[0]`, on ajoute `l[0]` à la liste adéquate.

```
def sep_aux(e,l,l1,l2):
    """
    Range dans l1 les elements de l superieurs a e et dans l2 les elements
    de l strictemetn inferieur a e.
    type : int * list * list * list -> tuple
    """
    if l == []:
        return (l1,l2)
    elif l[0] >= e:
        return sep_aux(e,l[1:],l1 + [l[0]],l2)
    else:
        return sep_aux(e,l[1:],l1,l2 + [l[0]])
```

Ensuite, il suffit d'écrire la fonction principale qui fera appel à la fonction précédente.

```
def separate(e,l):
    """
    Calcule un couple contenant la liste des elements de l
    strictement inferieur a e et la liste des elements de l
    superieurs a e.
    type : int * list -> tuple
    """
    return sep_aux(e,l,[],[])
```

On a ainsi une solution fortement inspirée de la version itérative puisqu'on se contente de parcourir la liste l et de placer dans les listes $l1$ et $l2$ les éléments.
Nous allons nous intéresser à présent à une autre solution, plus éloignée de la version itérative. L'idée derrière est plus proche du raisonnement mathématique dit par récurrence :

- On considère le cas de base où la liste l est vide : dans ce cas, on retourne deux listes vides.
- On suppose qu'on a résolu le problème pour une liste de longueur $n-1$.
- Dans le cas d'une liste de longueur n, on commence par résoudre le problème sur les $n-1$ derniers éléments (hypothèse de récurrence) et on obtient ainsi deux listes $l1$ et $l2$. Si `l[0]` est inférieur à e, alors on le place à la fin de $l2$, sinon à la fin de $l1$.

Dans ce cas, on n'a pas besoin d'une fonction auxiliaire et la fonction s'écrit simplement :

```
def separate(e, l):
    if l == []:
        return ([], [])
    else:
        l1, l2 = separate(e,l[1:])
        if l[0] < e:
            l2 = l2 + [l[0]]
```

```
        else:
            l1 = l1 + [l[0]]
        return (l1, l2)
```

8. Écrire une fonction qui génère un tirage de loto, c'est-à-dire un tirage sans remise de 7 nombres entre 1 et 49. La fonction devra simplement afficher les nombres ainsi tirés.

   Indication : importer la fonction `randint` du module `random` en tapant au début de votre programme `from random import randint`. Pour obtenir un nombre entre $[a; b]$ où a et b sont deux entiers, utilisez la fonction `randint(a,b)`.

La difficulté dans cet exercice est bien sûr de mettre en place le mécanisme pour ne pas tirer deux fois le même nombre. Si dans tous les exercices de programmation il n'y a pas de solution unique, cela est particulièrement vrai ici. Nous allons vous proposer une solution qui nous semble simple.
Elle consiste à établir une liste l contenant les nombres de 1 à 49 compris. Puis, pour les 7 tirages, nous allons choisir un élément de la liste au hasard, l'afficher, puis le retirer de la liste.
Pour tirer un nombre au hasard, on va simplement utiliser la fonction `randint` pour obtenir un indice de la liste, c'est-à-dire un entier entre 0 et `len(l)-1`.
Pour retirer l'élément de la liste placé à l'indice i, nous avons deux possibilités :

- `del l[i]` (mot clé propre à Python) ;
- `l = l[:i]+l[i+1:]` (slicing).

D'une manière générale, nous allons préférer la seconde méthode qui est plus dans l'esprit du module.

```
def loto():
    """Execute un tirage du loto"""
    boules = list(range(1,50))
    for i in range(7):
        index = randint(0,len(boules) - 1)
        print(boules[index])
        boules = boules[:index] + boules[index + 1:]
```

Exercice 6 : Entiers naturels

Dans les exercices suivants, nous considèrerons uniquement les entiers naturels.

1. Écrire une fonction `isPerfect` qui décide si un nombre est parfait, c'est-à-dire s'il est égal à la somme de ses diviseurs propres (par exemple, 6 est un nombre parfait car $1 + 2 + 3 = 6$) et qu'il est différent de 1.

La stratégie à adopter pour cet exercice est d'énumérer les diviseurs propres du paramètre n tout en les sommant et vérifier à la fin si la somme est égale à n. Il s'agit d'écrire une fonction qui prend en paramètre ce nombre n et retourne un booléen, c'est-à-dire soit `True` soit `False`. On suppose que le nombre n est un nombre entier naturel différent de 0.
Dans une première version naïve, on pourrait imaginer parcourir les nombres de 1 jusqu'à n. Cela dit, les nombres strictement supérieurs à $\frac{n}{2}$ ne sont pas diviseurs de n, donc on peut s'arrêter à $\frac{n}{2}$. Pour cela, on utilisera la boucle `for` et la fonction `range` en faisant attention à inclure $\frac{n}{2}$.
Pour tester si un nombre entier a est divisible par un autre nombre entier b, il suffit d'utiliser l'opérateur `%` (modulo) qui retourne le reste de la division entière. Si `a % b` vaut 0, alors a est divisible par b.

```python
def isPerfect(n):
    """
    Indique si le nombre entier naturel n > 0 est parfait.
    type : int -> bool

    Complexite : lineaire --> O(n)
    """
    sum_div = 0
    for i in range(1,n // 2 + 1):
        if n % i == 0:
            sum_div += i
    return sum_div == n and n != 1
```

Le test `n != 1` permet d'évacuer le cas 1 (qui n'est pas parfait) ; on aurait pu l'évacuer dès le début avec un `if`.
Remarquez l'utilisation de `range(1,n // 2 + 1)` : pour inclure $\frac{n}{2}$, le deuxième argument est `n//2+1` au lieu de `n // 2`. Souvenez-vous que `range(a,b)` retourne tous les entiers dans $[a\ ;\ b[$.

2. Écrire une fonction `perfectList` qui étant donné un entier n, calcule la liste des entiers parfaits inférieurs à n.

Pour résoudre le problème posé par cette question, nous allons utiliser la fonction écrite à la question précédente : c'est un cas typique de résolution de problème qui s'appuie sur la résolution d'un sous-problème. Ainsi afficher les nombres parfaits inférieurs ou égaux à n (i.e. le problème) nécessite de savoir reconnaître un nombre parfait (i.e. le sous-problème).
Nous allons écrire une fonction qui prend en paramètre le nombre n et retourne une liste (éventuellement vide) des nombres parfaits inférieurs ou égaux à n. Il suffit d'exécuter une boucle (soit `for`, soit `while`) pour faire défiler les nombres entre 2 (puisque 1 n'est pas parfait) et n compris, de tester si ce nombre est parfait avec la fonction `isPerfect` et de l'ajouter à une liste si c'est le cas.
Écrivons la fonction avec la boucle `while` :

```python
def perfectList(n):
    """
    Calcule la liste des nombres parfaits inferieurs ou egaux a n.

    type : int -> list
    Complexite : quadratique --> O(n^2)
    """
    l = []
    i = 2
    while i <= n:
        if isPerfect(i):
            l = l + [i]
            i = i + 1
    return l
```

et maintenant avec la boucle for :

```python
def perfectList(n):
    """
    Calcule la liste des nombres parfaits inferieurs ou egaux a n.

    type : int -> list
    Complexite : quadratique --> O(n^2)
    """
    l = []
    for i in range(2,n + 1):
```

```
10          if isPerfect(i):
11              l = l + [i]
12      return l
```

La boucle `for` semble un peu plus pratique du moment qu'on prête attention aux arguments de la fonction range.

3. Écrire un programme qui affiche les nombres compris entre 0 et 999 étant égaux à la somme des cubes des chiffres qui les composent.

   Par exemple, $153 = 1^3 + 5^3 + 3^3 = 1 + 125 + 27$.

Dans cet exercice, on peut appliquer deux stratégies :

- soit on part d'un nombre n et on extrait ses centaines, dizaines et unités ;
- soit on part des centaines, dizaines et unités et constitue le nombre n.

Dans le premier cas, il suffira de faire varier n de 1 à 999. Puis pour extraire ses centaines, il suffira de faire une division entière par 100, puis le reste de cette division peut être lui-même divisé par 10 pour obtenir les dizaines. Enfin les unités sont données par le reste de la division de n par 10.

```
1 for n in range (1000):
2     c = n // 100
3     d = (n % 100) // 10
4     u = n% 10
5     if n == c**3 + d**3 + u**3:
6         print(n)
```

Dans la seconde approche, il faut faire varier une variable c (comme centaines) de 0 à 9, puis une variable d (pour dizaines) de 0 à 9, puis une variable u (pour unités) de 0 à 9. Alors, $n = c * 100 + d * 10 + u$.

```
1 for c in range(10):
2     for d in range(10):
3         for u in range(10):
4             n = c*100 + d*10 + u
5             if n == c**3 + d**3 + u**3:
6                 print(n)
```

4. Un nombre heureux est un nombre entier non nul qui, lorsqu'on ajoute les carrés de ses chiffres, puis les carrés des chiffres de ce résultat et ainsi de suite jusqu'à l'obtention d'un nombre à un seul chiffre, donne 1 pour résultat. Par exemple, 7 est un nombre heureux :

   - $7^2 = 49$
   - $4^2 + 9^2 = 97$
   - $9^2 + 7^2 = 130$
   - $1^2 + 3^2 + 0^2 = 10$
   - $1^2 + 0^2 = 1$

   En revanche, 2 n'est pas heureux :

   - $2^2 = 4$
   - $4^2 = 16$
   - $1^2 + 6^2 = 37$

- $3^2 + 7^2 = 58$
- $5^2 + 8^2 = 89$
- $8^2 + 9^2 = 145$
- $1^2 + 4^2 + 5^2 = 42$
- $4^2 + 2^2 = 20$
- $2^2 + 0^2 = 4$

On détecte un cycle car on a déjà rencontré le nombre 4 donc on peut arrêter et 2 n'est pas heureux.

(a) Écrivez une fonction `decompose` qui pour un nombre entier n retourne la liste des chiffres qui le composent. Par exemple, `decompose`(134) retourne la liste [1,3,4].

Pour cette question, nous allons écrire deux solutions : une fonction récursive puis une fonction itérative. Dans les deux cas, on suppose que le nombre passé en paramètre est un entier naturel non nul.
La fonction récursive va isoler à chaque appel le chiffre des unités du nombre n passé en paramètre. L'appel récursif se fera donc sur $\lfloor \frac{n}{10} \rfloor$. Le cas d'arrêt de la récursion est atteint lorsque le nombre n est strictement inférieur à 10 (c'est-à-dire qu'il n'a plus que des unités). Pour isoler les unités de n, il suffit de prendre le reste de la division de n par 10. Rappelez-vous, dans la majorité des cas, une fonction récursive à au moins un `if ... else` pour isoler le cas terminal.

```
1 def decompose(n):
2     if n < 10:
3         return [n]
4     else:
5         return decompose(n // 10) + [n % 10]
```

Pour la fonction itérative, il faut tenir le même raisonnement. Tant que le nombre n a plus d'un seul chiffre, il faut isoler ses unités et le diviser par 10 puis recommencer. Nous allons écrire cela naturellement avec une boucle `while`. Ainsi, on sortira de la boucle lorsque notre nombre n'aura plus qu'un chiffre. Il ne faudra pas oublier de l'ajouter en fin de liste.

```
1 def decompose_iter2(n):
2     l=[]
3     while n >= 10:
4         l = [n % 10] + l
5         n = n // 10
6     l = [n] + l
7     return l
```

Remarquez que dans le cas récursif, on a le critère d'arrêt qui est $n < 10$ alors que dans le cas itératif, il s'agit d'une condition de poursuite de la boucle qui est $n \geq 10$, donc la négation de $n < 10$.

(b) Écrivez une fonction qui indique si un nombre entier est heureux.

Pour répondre à cette question, nous avons besoin d'une fonction qui va nous aider à calculer la somme des carrés des éléments d'une liste. Nous allons écrire deux versions de cette fonction.
Commençons par la version récursive. Il faut trouver le cas d'arrêt : si la liste est vide, on n'a plus besoin d'appeler la fonction et on peut retourner 0. Si elle ne l'est pas, on peut ajouter le carré de son premier élément à l'appel récursif effectué sur la liste privée de son premier élément (puisqu'on l'a déjà traité). Comme notre critère d'arrêt est la vacuité de la liste (i.e. le fait qu'elle soit vide), il faut absolument qu'à chaque appel récursif, notre liste ait un élément en moins.

```python
def somme_carre(l):
    if l == []:
        return 0
    else:
        return l[0] ** 2 + somme_carre(l[1:])
```

La solution itérative s'écrit très simplement en parcourant les éléments de la liste passée en paramètre, et en additionnant leurs carrés.

```python
def somme_carre(l):
    s = 0
    for e in l:
        s = s + e ** 2
    return s
```

A présent, on peut écrire la fonction demandée dans l'énoncé. La difficulté est de vérifier que l'on ne tombe pas dans un cycle. Pour cela, on va utiliser une liste dans laquelle on va placer les éléments qu'on a déjà rencontrés. Chaque fois qu'on calculera le prochain élément de la suite, on testera s'il n'est pas déjà dans la liste. Si c'est le cas, on retournera `False`. Sinon, si cet élément est 1, on retournera `True`. Sinon on poursuivra le calcul. Lors du premier appel à notre fonction, il faudra lui transmettre le nombre à tester et une liste vide. Nous allons donc créer une fonction récursive auxiliaire, et une fonction principale qui fera appel à la première, pour respecter la signature demandée par l'énoncé.

```python
def is_heureux_aux(n,deja_vu):
    suivant = somme_carre_rec(decompose(n))
    if suivant == 1:
        return True
    elif suivant in deja_vu:
        return False
    else:
        return is_heureux_aux(suivant, deja_vu + [suivant])

def is_heureux(n):
    return suite_nombres(n,[])
```

Rappel: `n // k` fournit le quotient de la division entière de n par k et `n % k` fournit le reste de la division entière de n par k.

Exercice 7 : Chaînes de caractères

1. L'ordre lexicographique $\leq_{lex}$ sur les chaînes de caractères s'exprime de la façon suivante : soient a et b deux chaînes de caractères s'écrivant respectivement "$a_1 \dots a_n$" et "$b_1 \dots b_p$". $a \leq_{lex} b$ si et seulement si :

   Pour $r = min\{n, p\}$:

   - ou bien il existe $i \leq r$ tel que $a_i <_\alpha b_i$ et pour tout $j < i$, $a_j = b_j$ avec $<_\alpha$ l'ordre alphabétique sur les caractères.
   - ou bien $\forall i \leq r, a_i = b_j$ et $p \leq q$

   Par exemple, "aab" $\leq_{lex}$ "aac" et "ver" $\leq_{lex}$ "verre" mais "bab" $\not\leq_{lex}$ "aac" et "abandonware" $\not\leq_{lex}$ "abandon". Il s'agit en fait de l'ordre utilisé pour classer les mots dans le dictionnaire.

   Écrire une fonction récursive `ordrelexico` prenant deux chaînes `ch1` et `ch2` et retournant `True` si et seulement si `ch1` $\leq_{lex}$ `ch2`.

Pour écrire une fonction récursive, il faut penser au cas d'arrêt.
Ici on a deux séquences en paramètre que l'on va appeler `ch1` et `ch2`. Étudions les cas :

- si `ch1` et `ch2` sont vides, il y a égalité et il faut retourner `True` ;
- si `ch1` est vide mais pas `ch2`, il faut retourner `True` ;
- si `ch1` n'est pas vide mais que `ch2` est vide, alors il faut retourner `False`.

Sinon, si `ch1` et `ch2` ne sont pas vides, on peut comparer leurs premiers caractères :

- si `ch1[0]` $=$ `ch2[0]` alors il faut examiner les lettres suivantes ;
- si `ch1[0]` $<$ `ch2[0]` alors on retourne `True` ;
- sinon on retourne `False`.

```
def ordrelexico (ch1,ch2):
    """
    Renvoie True si 'ch1' est superieure a 'ch2' au sens
    de l'ordre lexicographique

    type : str * str -> bool
    """
    if len(ch2) == 0:
        return len(ch1) == 0
    elif len(ch1) == 0:
        return True
    elif ch1[0] == ch2[0]:
        return ordrelexico(ch1[1:],ch2[1:])
    else: return ch1[0] < ch2[0]
```

L'ordre dans lequel on place les `if`, `else` et `elif` est important. En effet, on ne pourrait pas commencer par la condition `ch1[0] == ch2[0]` sans avoir vérifié que les chaînes n'étaient pas vides d'abord. De même, lorsqu'on arrive sur `elif len(ch1) == 0`, on sait que `ch2` n'est pas vide car on n'est pas rentré dans le premier `if`.

2. On dit qu'une chaîne de caractères a (sous la forme "$a_1 \dots a_n$") est *préfixe* d'une autre chaîne de caractères b (sous la forme "$b_1 \dots b_p$") si et seulement si $n \leq p$ et $a_i = b_i$ pour tout $1 \leq i \leq n$.

   Écrire une fonction récursive `prefixe` avec comme arguments deux chaînes de caractères, retournant un booléen testant si le premier argument est préfixe du second ou pas.

Comme pour l'exercice précédent, on va identifier les cas d'arrêt. Comme les paramètres sont aussi des chaînes de caractères, les critères d'arrêt vont concerner la vacuité des chaînes (i.e. le fait qu'elles soient vides). Il y a moins de cas à traiter, car si `ch1` est vide on retournera tout le temps vrai car la chaîne vide est le préfixe de n'importe quelle chaîne de caractères. En revanche si `ch1` n'est pas vide et que `ch2` l'est, alors `ch1` n'est pas un préfixe de `ch2`.
Sinon, on va vérifier que les premiers caractères des deux chaînes sont égaux : si c'est le cas, il faut tester les autres caractères, sinon `ch1` n'est pas un préfixe de `ch2`.

```
def prefixe(ch1,ch2):
    """
    Renvoie 'True' si ch1 est prefixe de ch2,
        c'est a dire si ch1 coincide avec le debut de ch2

    type : str * str -> bool
```

```
    """
    if len(ch1) == 0:
        return True
    elif len(ch2) == 0:
        return False
    elif ch1[0] == ch2[0]:
        return prefixe(ch1[1:],ch2[1:])
    else: return False
```

3. On dit qu'une chaîne de caractères a (sous la forme "$a_1 \dots a_n$") est *suffixe* d'une autre chaîne de caractères b (sous la forme "$b_1 \dots b_p$" si et seulement si $n \leq p$ et $a_i = b_{p-n+i}$ pour tout $1 \leq i \leq n$.

   Écrire une fonction récursive `suffixe` avec comme arguments deux chaînes de caractères, retournant un booléen testant si le premier argument est suffixe du second ou pas.

Nous allons appliquer la même méthode que pour l'exercice précédent, mais nous allons comparer les derniers éléments au lieu des premiers. Les cas d'arrêt sont donc les mêmes et la chaîne vide est suffixe de toute autre chaîne de caractères.

```
def suffixe(ch1,ch2):
    """
    Renvoie True si ch1 est suffixe de ch2, c'est a dire si ch1
    coincide avec la fin de ch2

    type : str * str -> bool
    """
    if ch1 == "":
        return True
    elif ch2 == "":
        return False
    elif ch1[-1]==ch2[-1]:
        return suffixe(ch1[0:-1],ch2[0:-1])
    else: return False
```

On voit une autre approche pour tester si une chaîne de caractères est vide en écrivant `ch1 == ""` au lieu de `len(ch1) == 0`.

4. On dit qu'une chaîne de caractères a (sous la forme "$a_1 \dots a_n$") est *motif* d'une autre chaîne de caractères b (sous la forme "$b_1 \dots b_p$" si et seulement si $n \leq p$ et b s'écrit "$b_1 \dots b_i \ a_1 \dots a_n \ b_{i+n+1} \dots b_p$".

   Écrire une fonction `motif` avec comme arguments deux chaînes de caractères, retournant un booléen testant si le premier argument est un motif du second ou pas.

Le but de cet exercice est d'écrire une fonction qui indique si une chaîne `ch1` apparaît dans une chaîne `ch2`. Pour cela on va utiliser la fonction `prefixe`. En effet, on va tester si `ch1` est un préfixe de `ch2`. Si ce n'est pas le cas, on va retirer le premier caractère de `ch2` et on va tester à nouveau, et ainsi de suite.
Le cas d'arrêt pourrait être lorsque `ch2` est vide, cependant on peut arrêter la recherche sitôt que `ch1` est plus longue que `ch2` : dans ce cas `ch1` n'est pas un motif de `ch2`.

```
def motif(ch1,ch2):
    """
    Renvoie True si ch1 est un motif de ch2, c'est a dire si ch1 apparait
    quelque part
```

```
    dans ch2.

    type : str * str -> bool
    """
    if len(ch1) > len(ch2):
        return False
    elif prefixe(ch1,ch2):
        return True
    else: return motif(ch1,ch2[1:])
```

Exercice 8 : Le typage

1. Écrire une fonction récursive `oftype` qui teste si tous les éléments d'une liste argument sont d'un même type donné. Ainsi `oftype([1,2,3],int)` vaut `True` tandis que `oftype([1,True,3],int)` vaut `False`.

Dans cette question, nous allons simplement écrire une fonction qui parcourt récursivement la liste et effectue une comparaison sur les types.
Pour cela nous allons utiliser la fonction `type` qui prend en argument une expression (i.e. une variable, une formule...) et indique son type. Parmi les types de Python, nous utiliserons particulièrement les types suivants : `int`, `float`, `str`, `list`, `dict` et `tuple`.
Le cas d'arrêt sera la liste vide, le test portera sur le premier élément de la liste et l'appel récursif se fera sur la liste privée de son premier élément.

```
def oftype(l,t):
    """
    Teste si tous les elements de l sont de type t.
    type : list * type -> bool
    """
    if l == []:
        return True
    elif type(l[0]) == t:
        return oftype(l[1:],t)
    else:
        return False
```

2. Écrire une fonction récursive `checkunif` qui teste si les éléments d'une liste sont tous de même type. Ainsi `checkunif([3,5,6])`, `checkunif(['tôto','titi'])` valent `True` tandis que `checkunif([3,'toto'])` vaut `False`.

Il s'agit tout simplement d'utiliser la fonction précédente. Si la liste est vide, la fonction va renvoyer `True`. Du moment que la liste contient au moins un élément, on va faire appel à la fonction précédente en passant en paramètre la liste privée de son premier élément et le type du premier élément.

```
def checkunif(l):
    """
    Teste si les elements d'une liste l sont tous de meme type.
    type : list -> bool
    """
    if l == []:
        return True
```

```
8       else:
9           return oftype(l[1:],type(l[0]))
```

3. Écrire une fonction itérative `somme` qui prend en entrée une liste et qui retourne la somme de ses éléments. Si les éléments ne sont pas des entiers (type `int`) ou des réels (type `float`) ils sont simplement ignorés. On considère que la liste ne peut pas contenir de listes.

   Par exemple, `somme([1, 2.0, 3, "coucou"', 4, True])` retourne `10.0`.

Ce n'est pas la première somme que nous écrivons donc cette question ne présente guère de difficultés.

```
1  def somme(l):
2      """
3          Returns the sum of the real and integer elements in l.
4          Type: list -> real
5      """
6      somme = 0.0
7      for e in l:
8          if type(e) == int or type(e) == float:
9              somme = somme + e
10
11     return somme
```

4. Écrire une version récursive de la fonction précédente.

Pour la version récursive, on va considérer que le cas d'arrêt est le cas où la liste argument est vide. Si c'est le cas, on retourne 0. Sinon, si le premier élément de la liste est un entier ou un réel, alors on l'additionne au résultat de l'appel récursif sur la liste argument privée de son premier élément. Enfin, si ce n'est ni un entier, ni un réel, il suffit de rappeler la fonction

```
1  def somme(l):
2      """
3          Returns the sum of the real and integer elements in l.
4
5          Type: list -> real
6      """
7      if len(l) == 0:
8          return 0.0
9      elif type(l[0]) == int or type(l[0]) == float:
10         return l[0] + somme(l[1:])
11     else:
12         return somme(l[1:])
```

5. Modifier la fonction précédente (récursive) pour qu'elle puisse prendre en entrée des listes qui contiennent d'autres listes... Ainsi `somme([1,1,[1,1,[1, 1],1]])` vaudra 7.

Pour modifier la fonction précédente, il suffit de tenir compte d'un cas supplémentaire : si le premier élément de la liste est lui-même une liste, il suffit d'ajouter l'appel récursif sur cet élément au résultat de l'appel récursif sur la liste privée de son premier élément.

```python
def somme(l):
    """
        Returns the sum of the real and integer elements in l
        and takes lists of lists into account
        Type: list -> real
    """
    if len(l) == 0:
        return 0.0
    elif type(l[0]) == int or type(l[0]) == float:
        return l[0] + somme(l[1:])
    elif type(l[0]) == list:
        return somme(l[0]) + somme(l[1:])
    else:
        return somme(l[1:])
```

6. Écrire une fonction récursive `depth` qui calcule le degré d'imbrication d'une liste. Ainsi `depth([1,[2,[[3]]],6])` vaut 4 car la liste argument contient 4 niveaux d'imbrication (au niveau de l'élément 3).

Tout d'abord il faut bien comprendre ce que demande l'énoncé et dérouler à la main quelques exemples. Lorsque l'on a une liste qui ne contient pas d'autre liste, la fonction doit retourner 1. Lorsque l'on n'a pas de liste, elle doit retourner 0.
Nous allons donc avoir deux cas d'arrêt : soit l'élément courant n'est pas une liste et dans ce cas la fonction retourne 0, soit on a une liste vide et dans ce cas on retourne 1.
Si nous ne sommes dans aucun de ces cas, nous allons faire deux appels à la fonction `depth` : une fois sur le premier élément de la liste, une seconde fois sur la liste privée de son premier élément. Ainsi, nous aurons deux profondeurs : une relative au premier élément, l'autre relative à tout le reste de la liste. Sachant que pour la profondeur du premier élément, nous n'avons pas encore tenu compte que nous étions dans une liste, il faut penser à rajouter 1. Il ne reste plus qu'à prendre la plus grande des deux profondeurs.

```python
def depth(l):
    """
    Calcule le niveau d'imbrication (profondeur) de l.
    type : list -> int
    """
    if type(l) == list:
        if l == []:
            return 1
        else:
            d0 = depth(l[0])
            df = depth(l[1:])
            return max(d0 + 1,df)
    else:
        return 0
```

7. Écrire une fonction récursive `flatten` qui aplatit une liste (au sens où elle conserve les éléments, mais enlève les niveaux d'imbrication intermédiaires). Ainsi, `flatten([1,[2,[[3]]],6])` vaut `[1, 2, 3, 6]`.

Cette fois-ci nous avons deux cas d'arrêt : si l'élément en paramètre n'est pas une liste, il faut simplement le mettre dans une liste pour préparer sa concaténation avec le reste des éléments. Si le paramètre est une liste vide, alors on peut retourner une liste vide (notamment le paramètre lui-même puisqu'il

est vide). Sinon il faut aplatir le premier élément de la liste et concaténer ce résultat au résultat de la fonction sur le reste de la liste.

```
def flatten(l):
    """
    Aplatit la liste l.
    type : list -> list
    """
    if type(l) == list:
        if l == []:
            return l
        else:
            return flatten(l[0]) + flatten(l[1:])
    else:
        return [l]
```

Exercice 9 : Les dictionnaires

A savoir : Manipulation de dictionnaires

- Pour créer un dictionnaire vide, utilisez les symboles `{}`.
- Pour ajouter un couple `(clé, valeur)` dans un dictionnaire d, écrivez `d[cle] = valeur`. Si une valeur était déjà associée à cette clé, elle sera simplement écrasée.
- Pour supprimer une valeur du dictionnaire d à partir de sa clé, utilisez `del d[cle]`.
- Pour savoir si une certaine clé est contenue dans le dictionnaire d, utilisez `cle in d`.
- Pour obtenir la liste des clés contenues dans le dictionnaire d, utilisez `d.keys()`. Attention, la fonction ne retourne pas une liste mais un ensemble (`set`), donc si vous en désirez une, utilisez `list` pour une conversion.
- Pour obtenir la liste des valeurs contenues dans le dictionnaire d, utilisez `d.values()`. Attention, la fonction ne retourne pas une liste mais un ensemble, donc si vous en désirez une, utilisez `list` pour une conversion.
- Pour copier un dictionnaire, utilisez `d.copy()`.
- Pour connaître le nombre d'éléments dans le dictionnaire d, utilisez `len(d)`.
- Enfin, la boucle `for` itère sur les clés d'un dictionnaire d : `for cle in d`.

Attention, c'est une erreur de tenter l'accès à une clé qui n'existe pas.

1. Écrire une fonction `occur` qui prend en argument une liste d'éléments et retourne le nombre d'occurrences de chaque élément dans un dictionnaire.

   Ainsi, `occur([1,'e','a','e',1,1])` retourne `{'e':2, 1:3, 'a':1}`.

Cet exercice est l'occasion de se familiariser avec les dictionnaires. Les dictionnaires ne sont pas des séquences, on ne peut donc pas utiliser le slicing ou l'opérateur de concaténation (`+`) sur un dictionnaire. Pour résoudre cette question, nous allons utiliser une fonction itérative. Nous allons initialiser un dictionnaire (c'est-à-dire créer un dictionnaire vide). Pour compter les occurrences des éléments, nous

allons utiliser une boucle `for` pour parcourir la liste, et pour chaque élément e, nous allons vérifier s'il fait partie des clés du dictionnaire :

- si e est une clé du dictionnaire, on récupère la valeur associée, on lui rajoute 1 et on met à jour la valeur dans le dictionnaire ;
- si e n'est pas une clé du dictionnaire, il suffit d'associer la valeur 1 à la clé e.

```
1  def occur(l):
2      """
3          Retourne pour chaque element de l le nombre d'occurrences.
4          Type: list -> dict
5      """
6      d = {}
7      for e in l:
8          if e in d:
9              d[e] = d[e] + 1
10         else:
11             d[e] = 1
12     return d
```

2. Écrire une fonction `sortedoccur` qui prend en argument une liste de chaînes de caractères et qui affiche le nombre d'occurrences de chaque chaîne en les affichant par ordre alphabétique. Ainsi devra avoir :

```
1  >>> sortedoccur(['a','bb','a','c','e','bb','e', 'ba'])
2  a : 2
3  ba : 1
4  bb : 2
5  c : 1
6  e : 2
```

Il s'agit d'afficher le nombre d'occurrences des caractères classés par ordre alphabétique. Cela permet d'aborder une notion importante : il n'est pas possible de trier un dictionnaire. Les clés sont rangées de manière automatique et opaque pour nous de façon à optimiser leur recherche. En effet, trouver une clé dans un dictionnaire doit être une fonctionnalité très rapide, indépendante du nombre d'éléments contenus dans le dictionnaire. De même, lorsque vous parcourez les clés d'un dictionnaire avec la boucle `for`, les clés n'apparaitront pas dans l'ordre dans lequel vous les avez insérées.
Pour afficher les clés et les valeurs dans un certain ordre, nous allons mettre dans une liste toutes les clés du dictionnaire, trier la liste, puis la parcourir et interroger le dictionnaire pour avoir la valeur associée à chaque clé. Pour obtenir le dictionnaire, nous allons simplement faire appel à la fonction écrite à la question précédente.

```
1  def sortedoccur(l):
2      """
3          Retourne pour chaque chaine de caracteres de l le nombre d'
       occurrences.
4          Type: list -> dict
5      """
6      d = occur(l)
7      keys = list(d.keys())
8      keys = sorted(keys)
9      for k in keys:
10         print(k, " : ", d[k])
```

Contrairement aux dictionnaires, les listes peuvent être triées avec la fonction `sorted(l)` qui retourne

une copie de la liste l triée par ordre croissant.

3. Dans cette question, nous considérons le problème de la simulation d'une caisse enregistreuse, d'un magasin par exemple. Nous allons manipuler différents objets qui seront représentés comme suit :
   - les objets de type *panier* et *stock* seront des dictionnaires dont les clés désignent les articles et les valeurs le nombre d'articles correspondant ;
   - les objets de type *catalogue* seront des dictionnaires dont les clés désignent les articles et les valeurs les prix correspondants.

   (a) Écrire une fonction `saisie` qui demande à l'utilisateur de rentrer le nom des produits qui ont été achetés. On pourra arrêter lorsque le mot "STOP" sera saisi. Cette fonction mettra à jour et retournera un *panier*.

Cette fonction va permettre de saisir un panier. Il faut répéter la saisie d'un article tant que ce qui a été saisi n'est pas "STOP" et compter le nombre d'occurrences de l'article qui a été saisi (cela ressemble à l'exercice précédent). Il faut donc demander au moins une fois un article. Or la boucle `while` évalue une condition d'entrée dans la boucle. Il existe différentes méthodes pour demander au moins une fois un article.
Soit on procède à la saisie avant la boucle, puis dans la boucle. Dans ce cas-là, on duplique inutilement du code :

```
article = input("Article?")
while article != "STOP":
        # traiter l'article ici
        #redemander
        article = input("Article?")
```

Soit tout se passe dans la boucle et on utilise `break` pour en sortir :

```
while True:
        article = input("Article?")
        if article == "STOP" :
            break
        # traiter l'article ici
```

On pourrait éventuellement se passer du `break` en initialisant `article` et en utilisant différemment le `if` :

```
article = ""
while article != "STOP":
        article = input("Article?")
        if article != "STOP" :
            # traiter l'article ici
```

Ainsi, une solution possible pour cette question s'écrit :

```
def saisie():
    """ Retourne le panier correspondant au client"""
    panier = {}
    while True:
        article = input("Article? ")
        if article == "STOP":
            break
        if article in panier:
            panier[article] = panier[article] + 1
        else:
            panier[article] = 1
    return panier
```

(b) Écrire une fonction `ticket` qui prend un *panier* et un *catalogue* et affiche un ticket de caisse.

Par exemple, l'exécution de cette fonction pourrait ressembler à cela :

Cola	2x	1.5 €
Pain	1x	0.9€
Biscuits	3x	1.5€
	Total	8,40€

Cette question est l'occasion de réviser les notions d'affichage d'informations. En pratique, on va parcourir avec une boucle `for` les articles contenus dans le panier (les clés du dictionnaire). Pour chaque article, il faut regarder s'il est contenu dans le catalogue (éventuellement afficher un message d'erreur si non) afin de récupérer son prix. Puis faire une somme en multipliant le prix unitaire d'un article par le nombre d'articles achetés.

```
def ticket(panier, catalogue):
    """Affiche le ticket de caisse a partir du panier et du catalogue
    """
    total = 0
    for article in panier:
        if article not in catalogue:
            print("Le catalogue ne contient pas ", article)
        else:
            print("%s\t\t%s x\t%.2f eur" % (article, panier[article],
    catalogue[article]))
            total = total + panier[article] * catalogue[article]
    print()
    print("TOTAL = ", total)
```

Nous avons écrit une seule boucle pour procéder à la fois à l'affichage et au calcul du total à payer. Décortiquons l'affichage d'une ligne du ticket de caisse :

```
print("%s\t\t%s x\t%.2f eur" % (article, panier[article], catalogue[
    article]))
```

Le symbole `"\t"` représente le caractère tabulation lors de l'affichage (attention, il ne compte que comme un seul caractère). Le `%.2f` indique que l'on souhaite afficher un nombre réel (de type `float`) avec 2 chiffres après la virgule.

(c) Écrire une fonction `bilan` qui prend en argument une liste de *paniers* vendus dans la journée ainsi qu'un *stock* correspondant au début de la journée. Elle doit retourner un nouveau stock correspondant à celui à la fin de la journée.

L'idée est de copier le stock initial et de lui soustraire les articles achetés dans chacun des paniers contenus dans la liste. Nous allons avoir deux boucles imbriquées : la première pour parcourir les paniers, la seconde pour parcourir les articles d'un panier.

```
def bilan(stock, paniers):
    """Retourne l'etat du stock apres la prise en compte des ventes du
    jour"""

    #on copie le stock actuel:
    nouveau = stock.copy()

    #on soustrait au stock les ventes
    for panier in paniers:
        for article in panier:
            nouveau[article] = nouveau[article] - panier[article]
```

```
11
12      return nouveau
```

Dans cette fonction, on ne fait aucune vérification mais a priori on n'a pas pu vendre plus d'articles qu'il n'y en avait en stock au départ.

(d) Écrire une fonction `ticketMoyen` qui prend une liste de *paniers* et retourne le montant du ticket moyen.

Pour calculer le montant du ticket moyen, il suffit de calculer le total de chaque panier puis de faire la moyenne des totaux. Il suffit donc d'additionner tous les coûts de tous les paniers puis de diviser par le nombre de paniers contenus dans la liste. Nous aurons encore une fois deux boucles `for` imbriquées : une pour parcourir les paniers, l'autre pour parcourir les articles.

```
1  def ticketMoyen(paniers, catalogue):
2      somme = 0
3      for panier in paniers:
4          for article in panier:
5              somme = somme + panier[article] * catalogue[article]
6      return somme / len(paniers)
```