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 ;

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.

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).

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'` ;
   - 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 ;
   - la troisième renvoie un booléen indiquant si les 3 nombres peuvent représenter ou non les 3 côtés d'un triangle, sans rien afficher ;
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).
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.
4. Écrire une fonction qui calcule une ligne du triangle de Pascal (par récurrence, en calculant la ligne n à partir de la ligne $n-1$).

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.

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 :

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

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).

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 :

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

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

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.

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 $[e_1,\ldots,e_n]$,

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

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.

```
def sum(l):
    """
    Somme les elements de la liste l
    type : list -> int
    """
    s = 0
    while len(l) > 0:
        s = s + l[0]
        l = l[1:]
    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.

```
def sum(l):
    """
    Somme les elements de la liste l
    type : list -> int
    """
    s = 0
    for x in l:
        s = x + s
    return s
```

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

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]`.

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

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`.

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]`.

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])`.

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)`.

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.

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

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

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].

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

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`.

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.

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.

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.

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`.

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`.

4. Écrire une version récursive de la fonction précédente.

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.

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).

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]`.

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
```

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*.

(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€

(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.

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