5 Structures de données composites

Dans le chapitre 2, nous avons déjà manipulé des données de différents types : les nombres entiers, les nombres réels et les chaînes de caractères. Il est temps à présent d'examiner d'un peu plus près de nouvelles structures de données offertes par Python : les **conteneurs**. De façon générale, un conteneur est un objet composite destiné à contenir d'autres objets. Les conteneurs sont généralement regroupés en deux catégories : les **séquences** et les **tableaux associatifs**. Une séquence est un conteneur ordonnés d'éléments indicés par des entiers. Python dispose de trois types prédéfinis de séquences : les **chaînes de caractères,** les **listes** et les **tuples**. Un tableau associatif est, quant à lui, un type de données permettant de stocker des couples "clé-valeur" avec un accès très rapide à la valeur à partir de la clé. Python propose un type de tableau associatif : les **dictionnaires**. Ce chapitre se propose de discuter des caractéristiques principales et des contextes d'utilisation de ces différentes structures de données composites offertes par Python.

1 Les chaînes de caractères

Nous avons déjà rencontré les chaînes de caractères au chapitre 2. À la différence des données numériques, qui sont des entités singulières, les chaînes de caractères constituent **un type de donnée composite**. Nous entendons par là une entité bien définie qui est faite elle-même d'un ensemble d'entités plus petites, en l'occurrence, les caractères. Suivant les circonstances, nous serons amenés à traiter une telle donnée composite, tantôt comme un seul objet, tantôt comme une **suite ordonnée d'éléments**. Dans ce dernier cas, nous souhaiterons probablement pouvoir accéder à chacun de ces éléments à titre individuel.

En fait, les chaînes de caractères font partie d'une catégorie d'objets Python que l'on appelle des **séquences**, et dont font partie aussi les **listes** et les **tuples**. On peut effectuer sur les séquences tout un ensemble d'opérations similaires. Vous en connaissez déjà quelques unes, et nous allons en décrire quelques autres dans les paragraphes suivants.

1.1 Extraction de fragments de chaînes

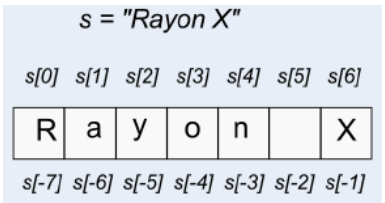
Une chaîne de caractères est une **séquence** de caractères. Chacun de ceux-ci occupe une place précise dans la séquence. Sous Python, les éléments d'une séquence sont toujours **indicés** (ou numérotés) de la même manière, c'est-à-dire **à partir de zéro**. Pour extraire un caractère d'une chaîne, il suffit d'accoler au nom de la variable qui contient cette chaîne, son **indice** entre crochets, comme l'illustre l'exemple suivant :

```
>>> s = "Rayon X"
>>> print(s[0], s[1], s[4])
R a n
```

En outre, il est souvent utile de pouvoir désigner l'emplacement d'un caractère par rapport à la fin de la chaîne. Pour ce faire, il suffit d'utiliser des indices négatifs (-1 pour le dernier caractère, -2 pour l'avant-dernier, ...) :

```
>>> print(s[-1], s[-2], s[-4], s[-7])
X   o R
```

L'indexation de la chaîne ci-dessus peut donc être résumée à l'aide de la figure suivante :



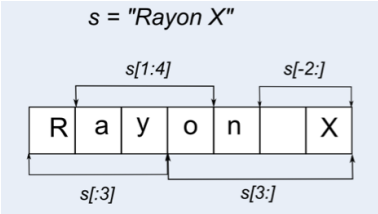
Il arrive fréquemment, lorsque l'on travaille avec des chaînes, que l'on souhaite extraire une petite chaîne d'une chaîne plus longue. Python propose pour cela une technique simple que l'on appelle **slicing** ou **découpage en tranches**. Elle consiste à indiquer entre crochets les indices correspondant au début et à la fin de la *tranche* que l'on souhaite extraire. Dans la tranche `[n:m]`, le $n^{ième}$ caractère est inclus, mais pas le $m^{ième}$:

```
>>> s = "Rayon X"
>>> print(s[1:4])
ayo
```

Les indices de découpage ont des valeurs par défaut : un premier indice non défini est considéré comme zéro, tandis que le second indice omis prend par défaut la taille de la chaîne complète :

```
>>> print(s[:3])        # les 3 premiers caractères
Ray
>>> print(s[3:])        # tout ce qui suit les 3 premiers caractères
on X
```

Le découpage en tranches de la chaîne ci-dessus peut donc être résumé à l'aide de la figure suivante :



1.2 Modification d'une chaîne

Les chaînes de caractères sont des séquences **non modifiables** en ce sens qu'il n'est pas possible de changer les caractères au sein d'une chaîne existante. En d'autres termes, vous ne pouvez pas utiliser l'opérateur `[ ]` dans la partie gauche d'une instruction d'affectation :

```
>>> salut = 'bonjour à tous'
>>> salut[0] = 'B'
Traceback (most recent call last):
TypeError: 'str' object does not support item assignment
```

Le résultat attendu par le programmeur qui a écrit ces instructions est « Bonjour à tous » (avec un B majuscule). Mais contrairement à ses attentes, ce script lève une erreur, provoquée à la deuxième ligne du script. On y essaie de remplacer une lettre par une autre dans la chaîne, mais cela n'est pas permis.

Par contre, le script ci-dessous fonctionne parfaitement :

```
>>> salut = 'bonjour à tous'
>>> salut = 'B' + salut[1:]
```

Dans cet autre exemple, en effet, nous ne modifions pas la chaîne `salut`. Nous en re-créons une **nouvelle**, avec le même nom, à la deuxième ligne du script.

1.3 Opérations sur les chaînes

Les chaînes peuvent être **concaténées** avec l'opérateur + et **répétées** avec l'opérateur * :

```
>>> n = 'abc' + 'def'       # concaténation
>>> m = 'zut ! ' * 4        # répétition
>>> print(n, m)
abcdef zut ! zut ! zut ! zut !
```

Rappelons que les opérateurs + et * peuvent aussi être utilisés pour l'addition et la multiplication lorsqu'ils s'appliquent à des arguments numériques. Le fait que les mêmes opérateurs puissent fonctionner différemment en fonction du contexte dans lequel on les utilise est un mécanisme fort intéressant que l'on appelle **surcharge des opérateurs**. Dans d'autres langages, la surcharge des opérateurs n'est pas toujours possible.

Sous python, les chaînes de caractères sont des **objets**[1]. Il est donc possible d'agir sur une chaîne en utilisant des **fonctions** (notion procédurale) ou des **méthodes** (notion objet) :

- Pour appeler une fonction, il suffit d'utiliser son nom suivi, entre parenthèses, des arguments sur lesquels est appliquée la fonction :

```
>>> ch ="Salut"
>>> print(len(ch))
5
```

- Pour appliquer une méthode à un objet, il suffit d'utiliser la **notation pointée**[2] entre l'objet sur lequel on applique la méthode et le nom cette dernière suivi, entre parenthèses, des arguments que demande la méthode :

```
>>> ch ="Salut"
>>> print(ch.upper())
SALUT
```

Nous verrons plus loin dans ces notes qu'une méthode est en quelque sorte une fonction intégrée, ou plutôt *encapsulée* dans les objets.

A toutes fins utiles, rappelons tout d'abord que l'on peut appliquer aux chaînes de caractères un certain nombre de fonctions intégrées dans le langage :

- `len(ch)` renvoie la longueur de la chaîne `ch`, ou en d'autres termes, son nombre de caractères :

```
>>> ch ="Salut"
>>> print(len(ch))
5
```

- `float(ch)` convertit la chaîne `ch` en un nombre réel (`float`) (bien entendu, cela ne pourra fonctionner que si la chaîne représente bien un nombre, réel ou entier) :

```
>>> a = float("12.36")          # Attention : pas de virgule décimale !
>>> print(a + 5)
17.36
```

- `int(ch)` convertit la chaîne `ch` en un nombre entier (avec des restrictions similaires) :

```
>>> a = int("184")
>>> print(a + 20)
204
```

- `str(obj)` convertit (ou représente) l'objet `obj` en une chaîne de caractères. `obj` peut être une donnée d'à peu près n'importe quel type :

```
>>>> a = 17
>>> ch = str(a) +" est un nombre premier."
>>> print(ch)
17 est un nombre premier.
```

Il est possible d'effectuer de nombreux traitements sur les chaînes de caractères en utilisant des méthodes appropriées. En voici quelques-unes, choisies parmi les plus utiles :

- `find(sch)` : cherche la position d'une sous-chaîne `sch` dans la chaîne :

```
>>> ch1 = "Cette leçon vaut bien un fromage, sans doute ?"
>>> ch2 = "fromage"
>>> print(ch1.find(ch2))
25
```

1. Les concepts informatiques de **classe d'objets** et de **méthode** ne seront expliqués qu'un peu plus loin dans ces notes, mais nous devons dès à présent montrer comment les utiliser dans le cadre des structures de données composites.

2. Le point séparateur exprime une relation d'appartenance. Relier ainsi des noms à l'aide de points est une manière de désigner sans ambiguïté des éléments faisant partie d'ensembles, lesquels peuvent eux-mêmes faire partie d'ensembles plus vastes, etc. Par exemple, l'étiquette `systeme.machin.truc` désigne l'élément `truc`, qui fait partie de l'ensemble `machin`, lequel fait lui-même partie de l'ensemble `systeme`. Nous verrons de nombreux exemples de cette technique de désignation, notamment lors de notre étude des **classes** d'objets.

- `count(sch)` : compte le nombre de sous-chaînes `sch` dans la chaîne :

```
>>> ch1 = "Le héron au long bec emmanché d'un long cou"
>>> ch2 = 'long'
>>> print(ch1.count(ch2))
2
```

- `lower()` : convertit une chaîne en minuscules :

```
>>> ch = "CÉLIMÈNE est un prénom ancien"
>>> print(ch.lower())
célimène est un prénom ancien
```

- `upper()` : convertit une chaîne en majuscules :

```
>>> ch = "Maître Jean-Marie Hébèrt"
>>> print(ch.upper())
MAÎTRE JEAN-MARIE HÉBÈRT
```

- `title()` : convertit en majuscule l'initiale de chaque mot (suivant l'usage des titres anglais) :

```
>>> ch ="albert rené élise véronique"
>>> print(ch.title())
Albert René Élise Véronique
```

- `capitalize()` : variante de la précédente. Convertit en majuscule seulement la première lettre de la chaîne :

```
>>> b3 = "quel beau temps !"
>>> print(b3.capitalize())
"Quel beau temps !"
```

- `swapcase()` : convertit toutes les majuscules en minuscules, et vice-versa :

```
>>> ch = "Le Lièvre Et La Tortue"
>>> print(ch.swapcase())
lE lIÈVRE eT lA tORTUE
```

- `strip(car)` : enlève tous les éventuels caractères `car` situés au début et à la fin de la chaîne :

```
>>> ch = "***Monty Python*** "
>>> ch.strip("*")
'Monty Python'
```

Si aucun caractère n'est précisé, `strip` enlève les éventuels espaces entourant la chaîne :

```
>>> ch = "   Monty Python   "
>>> ch.strip()
'Monty Python'
```

- `replace(c1, c2)` : remplace tous les caractères `c1` par des caractères `c2` dans la chaîne :

```
>>> ch8 = "Si ce n'est toi c'est donc ton frère"
>>> print(ch8.replace(" ","*"))
Si*ce*n'est*toi*c'est*donc*ton*frère
```

- `index(car)` : retrouve l'indice (`index`) de la première occurrence du caractère `car` dans la chaîne :

```
>>> ch9 ="Portez ce vieux whisky au juge blond qui fume"
>>> print(ch9.index("w"))
16
```

Notons finalement que cette liste de méthodes n'est pas exhaustive. Il ne s'agit que de quelques exemples. En outre, la plupart de ces méthodes peuvent être utilisées avec différents paramètres que nous n'avons pas tous indiqués ici. Vous pouvez obtenir la liste complète de toutes les méthodes associées à un objet et leur fonctionnalité en consultant la documentation de Python à l'adresse http://docs.python.org/3/index.html.

1.4 Formatage des chaînes de caractères

Pour terminer ce tour d'horizon des fonctionnalités associées aux chaînes de caractères, il nous paraît judicieux de vous présenter encore une technique de traitement très puissante, que l'on appelle **formatage des chaînes**. Celle-ci se révèle utile dans tous les cas où vous devez construire une chaîne de caractères complexe à partir d'un certain nombre de morceaux, tels que les valeurs de variables diverses.

Considérons par exemple que vous ayez écrit un programme qui traite de la couleur et de la température d'une solution aqueuse, en chimie. La couleur est mémorisée dans une chaîne de caractères nommée `coul`, et la température dans une variable de type réel nommée `temp`. Vous souhaitez que votre programme construise la chaîne de caractères : « *La solution est devenue rouge, et sa température atteint 12,7 °C* ».

A ce dessein, vous pouvez préparer une chaîne *modèle* contenant l'essentiel du texte invariant, avec des **balises** particulières aux endroits où vous souhaitez qu'apparaissent des contenus variables. Vous appliquerez ensuite à cette chaîne la méthode `format()`, à laquelle vous fournirez comme arguments les divers objets à convertir en caractères et à insérer en remplacement des balises :

```
>>> coul ="verte"
>>> temp =1.347 + 15.9
>>> ch ="La couleur est {} et la température vaut {} °C"
>>> print(ch.format(coul, temp))
La couleur est verte et la température vaut 17.247 °C
```

Les balises à utiliser sont constituées d'accolades {}, contenant ou non des indications de formatage. La méthode `format()` doit recevoir autant d'arguments qu'il y aura de balises dans la chaîne. Si les balises sont vides, comme dans notre exemple, Python appliquera tout simplement la fonction `str()` aux arguments correspondants pour pouvoir les insérer à leur place dans la chaîne. Ces arguments peuvent être n'importe quel objet ou expression Python :

```
>>> pi =3.1416
>>> r =4.7
>>> ch ="Un disque de rayon {} a une aire égale à {}."
>>> print(ch.format(r, pi * r**2))
Un disque de rayon 4.7 a une aire égale à 69.397944.
```

La technique devient plus intéressante si vous insérez des indications de formatage dans les balises. Par exemple, vous pouvez améliorer la présentation de la chaîne, dans l'exemple précédent, en limitant la précision du résultat final, en utilisant la notation scientifique, en fixant le nombre total de caractères, etc.

```
>>> pi =3.1416
>>> r =4.7
>>> ch1 ="Un disque de rayon {} a une aire égale à {:8.2f}."
>>> ch2 ="Un disque de rayon {} a une aire égale à {:6.2e}."
>>> print(ch1.format(r, pi * r**2))
Un disque de rayon 4.7 a une aire égale à     69.40.
>>> print(ch2.format(r, pi * r**2))
Un disque de rayon 4.7 a une aire égale à 6.94e+01.
```

Dans la chaîne `ch1` essai, le résultat est formaté de manière à comporter un total de 8 caractères, dont 2 chiffres après le point décimal. Dans la chaîne `ch2`, le résultat est présenté en notation scientifique (e+01 signifie $\cdot 10^{01}$). Veuillez constater au passage que les arrondis éventuels sont effectués correctement.

La description complète de toutes les possibilités de **formatage** comporterait plusieurs pages, et cela sort du cadre de ces notes. S'il vous faut un formatage très particulier, veuillez consulter la documentation en ligne de Python, ou des manuels plus spécialisés. Signalons simplement encore, que le formatage permet d'afficher très facilement divers résultats numériques en notation binaire, octale ou hexadécimale :

```
>>> n = 789
>>> txt ="Le nombre {:d} (décimal) vaut {:x} en hexadécimal et {:b} en binaire"
>>> print(txt.format(n,n,n))
Le nombre 789 (décimal) vaut 315 en hexadécimal et 1100010101 en binaire
```

2 Les listes

Comme les chaînes de caractères, les **listes** font partie d'un type général que l'on appelle **séquences**, c'est-à-dire des *collections ordonnées d'objets*. Les divers éléments qui constituent une liste sont en effet toujours disposés dans le même ordre et sont rendus accessibles par l'intermédiaire d'un **index**, indiquant l'emplacement de l'objet dans la séquence. Comme c'était déjà le cas pour les caractères dans une chaîne, il faut cependant retenir que la numérotation de ces index commence **à partir de zéro**, et non à partir de un.

Sous Python, on peut définir une liste comme une *collection d'éléments séparés par des virgules, l'ensemble étant enfermé dans des crochets :*

```
>>> nombres = [5, 38, 10, 25]
>>> mots = ["jambon", "fromage", "confiture", "chocolat"]
>>> stuff = [5000, "Brigitte", 3.1416, ["Albert", "René", 1947]]
```

Comme on peut le constater dans le dernier exemple ci-dessus, les éléments individuels qui constituent une liste peuvent être de types variés. Dans cet exemple, en effet, nous avons rassemblé un entier, une chaîne, un réel et même une liste. A cet égard, le concept de liste est donc assez différent du concept de *tableau* (*array*) l'on rencontre dans d'autres langages de programmation.

2.1 Extraction de fragments de listes

Pour accéder aux éléments d'une liste, on utilise les mêmes méthodes (index, découpage en tranches) que pour accéder aux caractères d'une chaîne :

```
>>> print(nombres[2])
10
>>> print(nombres[1:3])
[38, 10]
>>> print(nombres[2:3])
[10]
>>> print(nombres[2:])
[10, 25]
>>> print(nombres[:2])
[5, 38]
>>> print(nombres[-1])
25
>>> print(nombres[-2])
10
```

Les exemples ci-dessus devraient attirer votre attention sur le fait qu'une **tranche** découpée dans une liste est toujours elle-même une liste (même s'il s'agit d'une tranche qui ne contient qu'un seul élément, comme dans notre troisième exemple), alors qu'un élément isolé peut contenir n'importe quel type de donnée.

Si l'on souhaite accéder à un élément faisant partie d'une liste, elle-même située dans une autre liste, il suffit d'indiquer les deux index **entre crochets successifs** :

```
>>> print(stuff[3][1])
René
```

Dans l'exemple ci-dessus on affiche l'élément n° 1 d'une liste, qui est elle-même l'élément n° 3 d'une autre liste, la liste `stuff`.

2.2 Modification d'une liste

Contrairement aux chaînes de caractères, les listes sont des **séquences modifiables**. Cela nous permettra de construire plus tard des listes de grande taille, morceau par morceau, d'une manière dynamique (c'est-à-dire à l'aide d'un algorithme quelconque).

```
>>> nombres[0] = 17
>>> nombres
[17, 38, 10, 25]
```

Dans l'exemple ci-dessus, on a remplacé le premier élément de la liste `nombres`, en utilisant l'opérateur `[ ]` (opérateur d'indiçage) *à la gauche* du signe égale. Cet opérateur permet également d'ajouter ou supprimer des éléments dans une liste. L'utilisation de cet opérateur est délicate mais elle permet une grande souplesse :

```
>>> mots = ['jambon', 'fromage', 'confiture', 'chocolat']
>>> mots[2:2] =["miel"]
>>> mots
['jambon', 'fromage', 'miel', 'confiture', 'chocolat']

>>> mots[5:5] =['saucisson', 'ketchup']
>>> mots
['jambon', 'fromage', 'miel', 'confiture', 'chocolat', 'saucisson', 'ketchup']
```

Pour utiliser cette technique, vous devez prendre en compte les particularités suivantes :

- Si vous utilisez l'opérateur [ ] **à la gauche du signe égale** pour effectuer une insertion ou une suppression d'élément(s) dans une liste, vous devez obligatoirement y indiquer une *tranche* dans la liste cible (c'est-à-dire deux index réunis par le symbole `:`), et non un élément isolé dans cette liste.
- L'élément que vous fournissez à la droite du signe égale **doit lui-même être une liste**. Si vous n'insérez qu'un seul élément, il vous faut donc le présenter entre crochets pour le transformer d'abord en une liste d'un seul élément. Notez bien que l'élément `mots[1]` n'est pas une liste (c'est la chaîne 'fromage'), alors que l'élément `mots[1:3]` en est une.

Vous comprendrez mieux ces contraintes en analysant ce qui suit :

```
1  >>> mots[2:5] = []                    # [] désigne une liste vide
2  >>> mots
3  ['jambon','fromage','saucisson', 'ketchup']
4
5  >>> mots[1:3] = ['salade']
6  >>> mots
7  ['jambon', 'salade', 'ketchup']
8
9  >>> mots[1:] = ['mayonnaise', 'poulet', 'tomate']
10 >>> mots
11 ['jambon', 'mayonnaise', 'poulet', 'tomate']
```

Analysons quelque peu les lignes de code ci-dessus :

— À la première ligne, nous remplaçons la tranche `[2:5]` par une liste vide, ce qui correspond à un effacement.

— À la cinquième ligne, nous remplaçons une tranche par un seul élément. Notez encore une fois que cet élément doit lui-même être présenté comme une liste.

— À la 9^{e} ligne, nous remplaçons une tranche de deux éléments par une autre qui en comporte 3.

2.3 Opérations sur les listes

On peut appliquer aux listes les opérateurs de concaténation (+) et de multiplication * de la même manière qu'aux chaînes de caractères :

```
>>> fruits = ['orange','citron']
>>> legumes = ['poireau','oignon','tomate']
>>> fruits + legumes
['orange', 'citron', 'poireau', 'oignon', 'tomate']
>>> fruits * 3
['orange', 'citron', 'orange', 'citron', 'orange', 'citron']
```

L'opérateur * est particulièrement utile pour créer une liste de n éléments identiques :

```
>>> sept_zeros = [0]*7
>>> sept_zeros
[0, 0, 0, 0, 0, 0, 0]
```

Supposons par exemple que vous voulez créer une liste `B` qui contienne le même nombre d'éléments qu'une liste `A`. Vous pouvez obtenir ce résultat de différentes manières, mais l'une des plus simples consistera à effectuer :

```
>>> B = [0]*len(A)
```

Tout comme aux chaînes de caractères, il est possible d'appliquer aux listes un certain nombre de fonctions intégrées dans le langage :

- `len(lst)` renvoie la longueur de la liste `lst`, ou en d'autres termes, son nombre d'éléments.
- `list()` retourne une liste vide :

```
>>> l = list()
>>> print(l)
[]
```

- `del(lst[ ])` supprime un ou plusieurs éléments d'une liste à partir de leur(s) index :

```
>>> nombres = [17, 38, 10, 25, 72]
>>> del(nombres[2])
>>> nombres
[17, 38, 25, 72]
>>> del(nombres[1:3])
>>> nombres
[17, 72]
```

- `range(n)` renvoie une séquence d'entiers de valeurs croissantes, différents d'une unité et situés entre 0 et $n-1$:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

La fonction `range()` retourne une séquence d'entiers[3] que vous pouvez utiliser directement ou convertir en une liste avec la fonction `list()`. Notez que, si vous appelez `range()` avec un seul argument, la liste contiendra un nombre de valeurs égal à l'argument fourni, mais en commençant à partir de zéro (c'est-à-dire que `range(n)` génère les nombres de 0 à n-1). Par conséquent, l'argument fourni n'est jamais dans la liste générée. On peut aussi utiliser `range()` avec deux, ou même trois arguments séparés par des virgules, afin de générer des séquences de nombres plus spécifiques :

```
>>> list(range(5, 13))
[5, 6, 7, 8, 9, 10, 11, 12]
>>> list(range(3,16,3))
[3, 6, 9, 12, 15]
```

Si vous avez du mal à assimiler l'exemple ci-dessus, considérez que `range()` attend toujours de un à trois arguments, que l'on pourrait intituler `FROM`, `TO` et `STEP`. `FROM` est la première valeur à générer, `TO` est la dernière (ou plutôt la dernière + un), et `STEP` le *pas* à sauter pour passer d'une valeur à la suivante. S'ils ne sont pas fournis, les paramètres `FROM` et `STEP` prennent leurs valeurs par défaut, qui sont respectivement 0 et 1.

Sous Python, les listes sont des objets à part entière, et vous pouvez donc leur appliquer un certain nombre de **méthodes** particulièrement efficaces. En voici quelques-unes :

- `split()` : convertit une chaîne de caractères en une liste de sous-chaînes. On peut choisir le caractère séparateur en le fournissant en argument, sinon c'est un espace par défaut :

```
>>> ch1 = 'Votez pour moi'
>>> a = ch1.split()
>>> print(a)
['Votez', 'pour', 'moi']
>>> ch2 = 'Cet exemple, parmi d'autres, peut encore servir'
>>> ch2.split(",")
['Cet exemple', 'parmi d'autres', 'peut encore servir']
```

3. `range()` donne en réalité accès à un **itérateur** (un objet Python générateur de séquences), mais la description des itérateurs sort du cadre que nous nous sommes fixé. Veuillez consulter la documentation en ligne de Python, si vous souhaitez des éclaircissements.

- `join()` : rassemble une liste de chaînes de caractères en une seule. La chaîne à laquelle on applique cette méthode est celle qui servira de séparateur (un ou plusieurs caractères) ; l'argument transmis est la liste des chaînes à rassembler :

```
>>> b = ['Bête', 'à', 'manger', 'du', 'foin']
>>> print(" ".join(b))
Bête à manger du foin
>>> print("——".join(b))
Bête——à——manger——du——foin
```

- `sort()` : trie la liste sur laquelle est appliquée la méthode :

```
>>> nombres = [17, 38, 10, 25, 72]
>>> nombres.sort()
>>> nombres
[10, 17, 25, 38, 72]
```

- `append(elt)` : ajoute l'élément `elt` à la fin de la liste sur laquelle est appliquée la méthode :

```
>>> nombres.append(12)
>>> nombres
[10, 17, 25, 38, 72, 12]
```

- `reverse()` : inverse l'ordre des éléments de la liste sur laquelle est appliquée la méthode :

```
>>> nombres.reverse()
>>> nombres
[12, 72, 38, 25, 17, 10]
```

- `index(elt)` : retourne l'index de l'élément `elt` dans la liste sur laquelle est appliquée la méthode :

```
>>> nombres.index(17)
4
```

- `remove(elt)` : enlève (efface) l'élément `elt` de la liste sur laquelle est appliquée la méthode :

```
>>> nombres.remove(38)
>>> nombres
[12, 72, 25, 17, 10]
```

Notez bien la différence entre la méthode `remove()` et la fonction `del` : `del` travaille avec un **index** ou **une tranche d'index**, tandis que `remove()` recherche **une valeur** (si plusieurs éléments de la liste possèdent la même valeur, seul le premier est effacé).

2.4 Parcours d'une liste

L'instruction `for` est l'instruction idéale pour parcourir une liste :

```
>>> prov = ['La','raison','du','plus','fort','est','toujours','la','meilleure']
>>> for mot in prov:
        print(mot, end =' ')

La raison du plus fort est toujours la meilleure
```

Si vous voulez parcourir une gamme d'entiers, la fonction `range()` s'impose :

```
>>> for n in range(10, 18, 3):
        print(n, n**2, n**3)

10 100 1000
13 169 2197
16 256 4096
```

Il est très pratique de combiner les fonctions `range()` et `len()` pour obtenir automatiquement tous les indices d'une séquence (liste ou chaîne) :

```
fable = ['Maître','Corbeau','sur','un','arbre','perché']
for index in range(len(fable)):
    print(index, fable[index])
```

L'exécution de ce script donne le résultat :

```
0 Maître
1 Corbeau
2 sur
3 un
4 arbre
5 perché
```

Comme nous l'avons déjà signalé au chapitre 3, le type de la variable utilisée avec l'instruction `for` est **redéfini continuellement** au fur et à mesure du parcours : même si les éléments d'une liste sont de types différents, on peut parcourir cette liste à l'aide de `for` sans qu'il ne s'ensuive une erreur, car le type de la variable de parcours s'adapte automatiquement à celui de l'élément en cours de lecture :

```
>>> divers = [3, 17.25, [5, 'Jean'], 'Linux is not Windoze']
>>> for item in divers:
        print(item, type(item))

3 <class 'int'>
17.25 <class 'float'>
[5, 'Jean'] <class 'list'>
Linux is not Windoze <class 'str'>
```

Dans l'exemple ci-dessus, on utilise la fonction intégrée `type()` pour montrer que la variable `item` **change effectivement de type** à chaque itération (ceci est rendu possible grâce au typage dynamique des variables Python).

2.5 Copie d'une liste

Considérons que vous disposez d'une liste `fable` que vous souhaitez recopier dans une nouvelle variable que vous appellerez `phrase`. La première idée qui vous viendra à l'esprit sera certainement d'écrire une simple affectation telle que :

```
>>> phrase = fable
```

En procédant ainsi, sachez que **vous ne créez pas une véritable copie**. À la suite de cette instruction, il n'existe toujours qu'une seule liste dans la mémoire de l'ordinateur. Ce que vous avez créé est seulement **une nouvelle référence** vers cette liste. Essayez par exemple :

```
1 >>> fable = ['Je','plie','mais','ne','romps','point']
2 >>> phrase = fable
3 >>> fable[4] ='casse'
4 >>> phrase
5 ['Je', 'plie', 'mais', 'ne', 'casse', 'point']
```

Si la variable `phrase` contenait une véritable copie de la liste, cette copie serait indépendante de l'originale et ne devrait donc pas pouvoir être modifiée par une instruction telle que celle de la troisième ligne, qui s'applique à la variable `fable.` Vous pouvez encore expérimenter d'autres modifications, soit au contenu de `fable`, soit au contenu de `phrase.` Dans tous les cas, vous constaterez que les modifications de l'une sont répercutées dans l'autre, et vice-versa.

En fait, les noms `fable` et `phrase` désignent tous deux **un seul et même objet** en mémoire. Pour décrire cette situation, les informaticiens diront que le nom `phrase` est un **alias** du nom `fable`. Avec les notions vues précédemment, vous devriez pouvoir trouver une technique permettant d'effectuer une véritable copie d'une liste (cf. exercices).

2.6 Histogramme des fréquences d'apparition de nombres aléatoires

Bien que les listes puissent rassembler des données de types différents, les programmeurs les utilisent habituellement pour stocker des données similaires. Ainsi, par exemple, une liste pourra contenir une suite d'entiers représentant les résultats d'un test ou une suite de chaînes de caractères correspondant aux noms des employés d'une entreprise. En général, un programme utilise une liste pour stocker des données dans le but de pouvoir ensuite leur appliquer une même opération. Cependant, la longueur de liste n'est souvent pas prédéterminée et peut même varier tout au long du programme.

Afin d'illustrer l'utilisation des listes dans une application concrète, nous vous proposons d'étudier un programme permettant de créer un histogramme[4] des fréquences d'apparition d'entiers situés entre 1 et 10 à partir d'une liste de données générées aléatoirement par l'ordinateur.

La plupart des programmes d'ordinateur font exactement la même chose chaque fois qu'on les exécute. De tels programmes sont dits **déterministes**. Le déterminisme est certainement une bonne chose : nous voulons évidemment qu'une même série de calculs appliquée aux mêmes données initiales aboutisse toujours au même résultat. Pour certaines applications, cependant, nous pouvons souhaiter que l'ordinateur soit imprévisible. Le cas des jeux de hasard constitue un exemple évident, mais il en existe bien d'autres.

Contrairement aux apparences, il n'est pas facile du tout d'écrire un algorithme qui soit réellement non-déterministe (c'est-à-dire qui produise un résultat totalement **imprévisible**). Il existe cependant des techniques mathématiques permettant de simuler plus ou moins bien l'effet du hasard. Des livres entiers ont été écrits sur les moyens de produire ainsi un hasard *de bonne qualité*. Nous n'allons évidemment pas développer ici une telle question.

Dans son module `random`, Python propose toute une série de fonctions permettant de générer des nombres aléatoires qui suivent différentes distributions mathématiques. Nous n'examinerons ici que la fonction `randrange` qui peut être importée à l'aide de la commande suivante :

```
>>> from random import randrange
```

Cette fonction peut être utilisée avec 1, 2 ou 3 arguments. Avec un seul argument, `randrange` renvoie un entier compris entre zéro et la valeur de l'argument diminué d'une unité. Par exemple, `randrange(6)` renvoie un entier compris entre 0 et 5. Avec deux arguments, le nombre renvoyé est compris entre la valeur du premier argument et la valeur du second argument diminué d'une unité. Ainsi, `randrange(2, 8)` retourne un entier compris entre 2 et 7. Finalement, si l'on rajoute un troisième argument, celui-ci indique que le nombre tiré au hasard doit faire partie d'une série limitée d'entiers, séparés les uns des autres par un certain intervalle, défini lui-même par ce troisième argument. Par exemple, `randrange(3, 13, 3)` renverra un des entiers de la série 3, 6, 9, 12 :

```
>>> from random import randrange
>>> for i in range(15):
        print(randrange(3, 13, 3), end =' ')

12 6 12 3 3 12 12 12 9 3 9 3 9 3 12
```

Les nombres ainsi générés sont-ils vraiment aléatoires ? C'est difficile à dire. Si nous ne tirons qu'un petit nombre de valeurs, nous ne pouvons rien vérifier. Par contre, si nous utilisons un grand nombre de fois la fonction `randrange(10)`, nous nous attendons à ce qu'un dixième des valeurs produites soient égales à chacun des 10 entiers situés entre 0 et 9. Le programme suivant permet de vérifier le fonctionnement du générateur de nombres aléatoires de Python en appliquant ce raisonnement.

Le corps principal du programme demande à l'utilisateur le nombre `n` d'entiers à tirer au hasard. Ce nombre est transmis à la fonction `list_alea` qui crée une liste de `n` entiers aléatoires situés entre 0 et 9. Une fois les `n` entiers mémorisés dans cette liste `listVal`, le programme transmet cette dernière à la fonction `list_effectifs` qui la parcourt afin de dénombrer le nombre d'occurrences de chaque valeur. Pour chacune des valeurs, `list_effectifs` crée un compteur, initialisé à zéro, capable de dénombrer sa fréquence d'apparition. Ces 10 compteurs sont finalement mémorisés dans une liste qui est retournée au corps principal du programme par la fonction `list_effectifs`. Celui-ci n'a dès lors plus qu'à afficher l'état de chaque compteur sous la forme d'un diagramme en bâtons formés d'une succession de `*` :

4. Dans une étude statistique dont le caractère étudié possède un nombre fini de modalités, un **histogramme** présente l'effectif de chaque modalité sous la forme d'un diagramme en bâtons.

```
from random import randrange

# Définition de fonctions nécessaires au corps du programme:
def list_alea(n):
    "list_alea(int n) --> list. Retourne une liste de <n> entiers tirés au hasard \
entre 0 et 9."

    valeurs = []                        # listes des entiers à tirer au hasard

    for i in range(n):
        nouv_valeur = randrange(10)     # tirer un entier au hasard entre 0 et 9
        valeurs.append(nouv_valeur)     # insérer le nouvel entier dans la liste

    return valeurs

def list_effectifs(list_valeurs):
    "list_effectifs(list list_valeurs) --> list. Retourne la liste des effectifs \
de chaque entier située dans la liste <list_valeurs>. Cette dernière est composée \
d'entiers situés entre 0 et 9."

    listCompt = [0]*10                  # création d'une liste de 10 compteurs à zéro

    for i in range(len(list_valeurs)):
        valeur = list_valeurs[i]        # prochaine valeur à traiter
        listCompt[valeur] += 1          # incrémentation du compteur correspondant

    return listCompt

# Corps principal du programme:
n = int(input("Nombre d'entiers à tirer au hasard entre 0 et 9: "))
print("Tirage au sort des", n, "valeurs...\n")
listVal = list_alea(n)
print("Histogramme des fréquences d'apparition des", n, "entiers tirés au hasard: \n")

frequences = list_effectifs(listVal)

print("Entier \t Fréquence \t Histogramme")
for i in range(10):
    freq = frequences[i]
    print(i, "\t", freq, "\t \t", "*" * freq)
```

Dans la définition de la fonction `list_alea`, vous constaterez que nous avons pris le parti de construire petit à petit la liste des valeurs tirées au hasard en utilisant la méthode `append` à partir d'une liste vide. Dans la définition de la fonction `list_effectifs`, nous avons d'abord construit une liste de 10 compteurs initialisés à zéro que nous avons ensuite incrémentés en fonction du nombre d'occurrences de chaque valeur de la liste d'entiers aléatoires transmise. L'exécution du programme avec 500 appels à la fonction `randrange` donne une répartition presque uniforme des entiers entre 0 et 9 puisque chacun d'eux apparaît en moyenne 50 fois :

```
Nombre d'entiers à tirer au hasard entre 0 et 9: 500
Tirage au sort des 500 valeurs...

Histogramme des fréquences d'apparition des 500 entiers tirés au hasard:

Entier   Fréquence   Histogramme
0        48          ************************************************
1        50          **************************************************
2        49          *************************************************
3        48          ************************************************
4        48          ************************************************
5        55          *******************************************************
6        45          *********************************************
7        55          *******************************************************
8        53          *****************************************************
9        49          *************************************************
```

3 Les tuples

Python propose un type de données appelé **tuple** qui est assez similaire à une liste dans la mesure où il s'agit d'une séquence de valeurs de types différents indexées par des entiers mais qui, comme les chaînes de caractères, **n'est pas modifiable**.

3.1 Opérations sur les tuples

Du point de vue syntaxique, un tuple est une collection d'éléments séparés par des virgules :

```
>>> tup1 = 'a', 'b', 'c', 'd', 'e'
>>> print(tup1)
('a', 'b', 'c', 'd', 'e')
```

Il est vivement conseillé de mettre le tuple en évidence en l'enfermant dans une paire de parenthèses, comme la fonction `print` de Python le fait elle-même. Il s'agit simplement d'améliorer la lisibilité du code :

```
>>> tup1 = ('a', 'b', 'c', 'd', 'e')
```

Pour créer un tuple contenant un seul élément, il est nécessaire de lui inclure une virgule finale :

```
>>> tup2 = ('André',)
>>> type(tup2)
<class 'tuple'>
```

Sans virgule, Python interprète `('André')` comme une chaîne de caractères entre parenthèses :

```
>>> tup2 = ('André')
>>> type(tup2)
<class 'str'>
```

Les opérations que l'on peut faire sur les tuples sont syntaxiquement similaires à celles que l'on effectue sur les listes, si ce n'est que les tuples ne sont pas modifiables :

```
>>> tup = ('a', 'b', 'c', 'd', 'e')
>>> print(tup[0])
'a'
>>> print(tup[2:4])
('c', 'd')
>>> tup[0] = 'André'                    ⟹ ***** erreur ! *****
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> tup = ('André',) + tup[1:]
>>> print(tup)
('André', 'b', 'c', 'd', 'e')
```

Vous pouvez déterminer la taille d'un tuple à l'aide de `len()`, le parcourir à l'aide d'une boucle `for`, utiliser l'instruction `in` pour savoir si un élément donné en fait partie, etc., exactement comme vous le faites pour une liste. Les opérateurs de concaténation et de multiplication fonctionnent aussi. Mais puisque les tuples ne sont pas modifiables, vous ne pouvez pas utiliser avec eux, ni l'intruction `del` ni la méthode `remove()` :

```
>>> tu1, tu2 = ("a","b"), ("c","d","e")
>>> tu3 = tu1*4 + tu2
>>> tu3
('a', 'b', 'a', 'b', 'a', 'b', 'a', 'b', 'c', 'd', 'e')
>>> for e in tu3:
...     print(e, end=":")
...
a:b:a:b:a:b:a:b:c:d:e:
>>> del(tu3[2])
TypeError: 'tuple' object doesn't support item deletion
```

Les tuples sont préférables aux listes partout où l'on veut être certain que les données transmises ne soient pas modifiées par erreur au sein d'un programme. En outre, les tuples sont moins *gourmands* en ressources système que les listes : ils occupent moins de place en mémoire et peuvent être traités plus rapidement par l'interpréteur. Afin que vous compreniez l'utilité des tuples, les paragraphes suivants se proposent de vous en donner quelques exemples d'application.

3.2 Affectation de tuple

Python permet d'affecter simultanément des valeurs à plusieurs variables :

```
>>> a, b = 3, 4
```

En réalité, cette **affectation parallèle** correspond à une affectation de tuple. En effet, la partie gauche de l'instruction ci-dessus correspond à un tuple de variables alors que celle de droite est un tuple d'entiers. Dès lors, chaque valeur est affectée à la variable correspondante de manière simultanée. Cette affectation parallèle est en fait une simplification d'écriture de l'affectation de tuple suivante :

```
>>> (a, b) = (3, 4)
>>> print((a, b))                    # afficher le tuple (a, b)
(3, 4)
>>> print(a, b)                      # afficher le contenu des variables a et b
3 4
```

Rappelons que l'affectation parallèle convient bien pour échanger les valeurs de deux variables `a` et `b`. Sans les tuples, cet échange nécessiterait la création d'une variable *tampon* stockant temporairement l'une des deux valeurs à échanger :

```
>>> tampon = a
>>> a = b
>>> b = tampon
```

Avec l'affectation de tuple, cet échange peut se faire de manière plus élégante :

```
>>> a, b = b, a
```

Plus généralement, la partie droite d'une affectation parallèle peut être une séquence de type quelconque (chaîne de caractères, liste ou tuple). Ainsi, par exemple, pour décomposer une adresse email en un nom d'utilisateur et un nom de domaine, il est possible d'écrire :

```
>>> adresse = 'monty@python.org'
>>> nom, domaine = adresse.split('@')   # conversion de l'adresse en une liste de mots
>>> print(nom)
monty
>>> print(domaine)
python.org
```

Comme nous le remarquons dans l'exemple ci-dessus, la valeur retournée par la méthode `split` est une liste de deux éléments : le premier élément (*'monty'*) est affecté à la variable `nom` alors que le deuxième (*'python.org'*) est affecté à la variable `domaine`.

3.3 Fonction retournant plusieurs valeurs

A strictement parler, une fonction ne peut retourner qu'une seule valeur. Or, si la valeur retournée est un tuple, l'effet est similaire à retourner plusieurs valeurs. Si vous désirez, par exemple, faire la division entière d'un nombre x par un autre y tout en obtenant le reste de celle-ci, il n'est pas nécessaire de calculer d'abord $x//y$ puis $x\%y$. En effet, Python possède une fonction prédéfinie `divmod` qui prend deux arguments entiers et retourne comme valeur de retour un tuple de deux valeurs, le quotient et le reste de la division entière du premier argument par le second. Il est dès lors possible de stocker ce résultat dans un tuple :

```
>>> resultat = divmod(7, 3)
>>> print(resultat)
(2, 1)
```

ou utiliser l'affectation parallèle pour stocker les deux valeurs séparément :

```
>>> quotient, reste = divmod(7, 3)
>>> print(quotient)
2
>>> print(resultat)
1
```

Pour définir une fonction comme celle présentée ci-dessus qui retourne un tuple de valeurs, il suffit de séparer chaque valeur à retourner par une virgule dans l'instruction `return`, comme le suggère la définition de la fonction `min_max` suivante. Cette fonction retourne un tuple de deux valeurs correspondant à l'élément minimal et à l'élément maximal d'une séquence `seq` transmise en argument :

```
def min_max(seq):
    return min(seq), max(seq)      # min et max sont des fonctions prédéfinies
```

3.4 Parcours d'une liste de tuples

Alors que les listes sont généralement utilisées pour stocker des séquences de données de même type, les tuples servent, quant à eux, à représenter des entités caractérisées par des données de types différents. Chaque élément d'un tuple fournit une partie de l'information totale mémorisée dans le tuple. Ainsi, par exemple, les tuples pourraient être utilisés pour représenter les élèves d'une école, chaque tuple contenant les informations propres à un élève particulier comme son nom et son âge. La population d'une école pourrait dès lors être représentée par une liste de tuples, comme le suggère l'exemple suivant :

```
ecole = []                  # l'école est représentée par une liste de tuples

nom, age = input("nom et âge de l'élève séparés d'une virgule \
ou -1,-1 pour terminer: ").split(',')

while 1:
    age = int(age)
    if age == -1:
        break
    eleve = (nom, age)      # un étudiant est représenté par un tuple
    ecole.append(eleve)
    nom, age = input("nom et âge de l'élève séparés d'une virgule ou -1,-1 pour terminer
        :
    ").split(',')           # récupérer les caractéristiques du prochain élève

for nom, age in ecole:
    print(nom, "est un(e) élève de", age, "ans.")
```

Comme nous pouvons le constater dans les deux dernières lignes du code ci-dessus, à chaque itération de la boucle, Python sélectionne le prochain tuple de la liste `ecole` et affecte la valeur des éléments qui le composent aux variables `nom` et `age`. Si la liste était composée de tuples formés de 3 éléments, le parcours se ferait à l'aide de 3 variables, de 4 si les tuples de la liste possèdaient 4 éléments, etc...

4 Les dictionnaires

Les types de données composites que nous avons abordés jusqu'ici (*chaînes*, *listes* et *tuples*) sont tous des **séquences**, c'est-à-dire des suites ordonnées d'éléments. Dans une séquence, il est facile d'accéder à un élément quelconque à l'aide d'un index entier, mais à la condition expresse de connaître son emplacement.

Les **tableaux associatifs,** que nous allons découvrir ici au travers des **dictionnaires**, constituent un autre type de données composite. De manière générale, un tableau associatif est un type de données associant à un ensemble de **clés** un ensemble correspondant de **valeurs**. Plus précisément, chaque clef est associée à une valeur comme le font en mathématiques les fonctions. Alors qu'une séquence associe des entiers consécutifs à des valeurs de différents types, un tableau associatif fait quant à lui correspondre des clés d'un type arbitraire (alphabétique, numérique ou même composite) à des valeurs d'un autre type. Un annuaire téléphonique peut dès

lors être vu comme un tableau associatif dans lequel les noms constituent les clefs et les numéros de téléphone les valeurs.

En Python, les **dictionnaires** constituent un exemple de tableau associatif qui ressemble aux listes dans la mesure où ils sont modifiables mais qui s'en différencient par le fait que les éléments que nous pouvons y enregistrer ne sont pas disposés dans un ordre immuable. En outre, l'accès à un élément constituant le dictionnaire ne se fait plus à l'aide d'un indice entier mais à partir d'une **clé** d'un type presque quelconque. Finalement, comme dans les listes, les éléments mémorisés dans un dictionnaire peuvent être de n'importe quel type : des valeurs numériques, des chaînes, des listes, des tuples, des dictionnaires, et même des fonctions, des **classes** ou des **instances de classes** (voir plus loin).

4.1 Création d'un dictionnaire et accès à ses éléments

À titre d'exemple, nous allons créer un dictionnaire de langue, pour la traduction de termes informatiques anglais en français.

Puisque le type **dictionnaire** est un type modifiable, nous pouvons commencer par créer un dictionnaire vide, puis le remplir petit à petit. Du point de vue de la syntaxe, on reconnaît un dictionnaire au fait que ses éléments sont enfermés dans une paire d'accolades. Un dictionnaire vide sera donc noté **{ }** :

```
>>> dico = {}
>>> dico['computer'] = 'ordinateur'
>>> dico['mouse'] ='souris'
>>> dico['keyboard'] ='clavier'

>>> print(dico)
{'computer': 'ordinateur', 'keyboard': 'clavier', 'mouse': 'souris'}
```

Comme vous pouvez l'observer dans la dernière ligne ci-dessus, un dictionnaire apparaît dans la syntaxe Python sous la forme d'une série d'éléments séparés par des virgules, le tout étant enfermé entre deux accolades. Chacun de ces éléments est lui-même constitué d'une paire d'objets : une **clé** et une **valeur**, séparés par un double point.

Dans un dictionnaire, les clés correspondent à des index et les éléments correspondent donc à des paires **clé-valeur**. Remarquez que, contrairement à ce qui se passe avec les listes, il n'est pas nécessaire de faire appel à une méthode particulière pour ajouter de nouveaux éléments à un dictionnaire : il suffit de créer une nouvelle paire clé-valeur. Ce procédé ne fonctionne pas avec les listes :

```
>>> dico['screen'] = 'ecran'
>>> print(dico)
{'screen': 'écran', 'computer': 'ordinateur', 'mouse': 'souris', 'keyboard': 'clavier'}

>>> liste =['jambon', 'salade', 'confiture', 'chocolat']
>>> liste[4] ='salami'

        ***** IndexError: list assignment index out of range  *****
```

Finalement, veuillez constater que l'ordre dans lequel les éléments apparaissent à la troisième ligne de notre exemple ne correspond pas à celui dans lequel nous les avons fournis. Cela n'a strictement aucune importance : nous n'essaierons jamais d'extraire une valeur d'un dictionnaire à l'aide d'un index numérique. Au lieu de cela, nous utiliserons les clés :

```
>>> print(dico['mouse'])
souris
```

Dès lors, des opérations comme la concaténation et l'extraction d'un groupe d'éléments contigus ne peuvent donc tout simplement pas s'appliquer aux dictionnaires. Si vous essayez tout de même d'utiliser la technique de *slicing* aux dictionnaires, Python lèvera une erreur lors de l'exécution du code :

```
>>> print(dico['computer':'keyboard'])

          ***** Erreur : TypeError: unhashable type *****
```

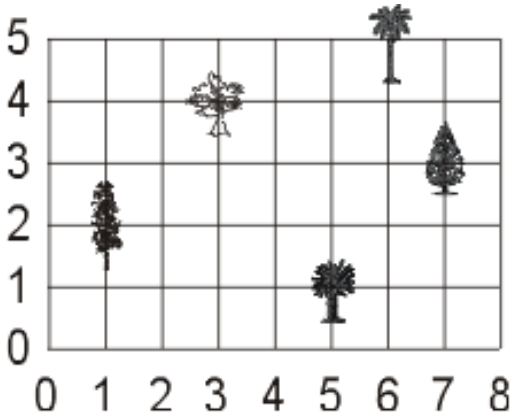
Dans l'exemple ci-dessus, nous avons décrit un dictionnaire dont les clés sont à chaque fois des valeurs de type *string*. En fait, nous pouvons utiliser en guise de clés n'importe quel type de données **non modifiables** : des entiers, des réels, des chaînes de caractères, et même des tuples.

Considérons par exemple que nous voulions répertorier les arbres remarquables situés dans un grand terrain rectangulaire. Nous pouvons pour cela utiliser un dictionnaire, dont les clés seront des tuples indiquant les coordonnées $(x;y)$ de chaque arbre :

```
>>> arb = {}
>>> arb[(1,2)] = 'Peuplier'
>>> arb[(3,4)] = 'Platane'
>>> arb[6,5] = 'Palmier'
>>> arb[5,1] = 'Cycas'
>>> arb[7,3] = 'Sapin'

>>> print(arb)
{(3, 4): 'Platane', (6, 5): 'Palmier',
(5, 1): 'Cycas', (1, 2): 'Peuplier',
(7, 3): 'Sapin'}

>>> print(arb[(6,5)])
Palmier
```



Vous pouvez remarquer que nous avons allégé l'écriture à partir de la troisième ligne, en profitant du fait que les parenthèses délimitant les tuples sont facultatives.

4.2 Opérations sur les dictionnaires

Tout comme aux chaînes de caractères, aux listes et aux tuples, il est possible d'appliquer aux dictionnaires un certain nombre d'opérations se présentant sous la forme de fonctions ou de méthodes. Pour illustrer notre propos, créons un nouveau dictionnaire, destiné cette fois à contenir l'inventaire d'un stock de fruits. Les index (ou clés) seront les noms des fruits et les valeurs seront les masses de ces fruits répertoriés dans le stock (les valeurs sont donc cette fois des données de type numérique) :

```
>>> invent = {'pommes': 430, 'bananes': 312, 'oranges' : 274, 'poires' : 137}
>>> print(invent)
{'oranges': 274, 'pommes': 430, 'bananes': 312, 'poires': 137}
```

Il est possible d'appliquer au dictionnaire `invent` un certain nombre de fonctions intégrées :

- `len(dico)` renvoie la longueur d'un dictionnaire, ou en d'autres termes, son nombre d'éléments :

```
>>> print(len(invent))
4
```

- `del(dico[ ])` supprime un ou plusieurs éléments d'un dictionnaire à partir de leur(s) clé(s) :

```
>>> del(invent['pommes'])
>>> print(invent)
{'poires': 137, 'bananes': 312, 'oranges': 274}
```

De plus, les dictionnaires étant des objets à part entière, vous pouvez donc leur appliquer un certain nombre de méthodes particulièrement efficaces. En voici quelques-unes :

- `keys()` : renvoie la séquence des *clés* utilisés dans le dictionnaire. Cette séquence peut être utilisée telle quelle dans les expressions ou convertie en liste ou en tuple si nécessaire, avec les fonctions intégrées correspondantes `list` ou `tuple`:

```
>>> print(invent.keys())
dict_keys(['poires', 'bananes', 'oranges'])
>>> list(invent.keys())
['poires', 'bananes', 'oranges']
>>> tuple(invent.keys())
('poires', 'bananes', 'oranges')
```

- `values()` : renvoie la séquence des *valeurs* mémorisées dans le dictionnaire :

```
>>> print(invent.values())
dict_values([137, 312, 27])
>>> list(invent.values())
[137, 312, 274]
>>> tuple(invent.values())
(137, 312, 274)
```

- `items()` : extrait du dictionnaire une séquence équivalente de tuples :

```
>>> invent.items()
dict_items([('poires', 137), ('bananes', 312), ('oranges', 274)])
>>> list(invent.items())
[('poires', 137), ('bananes', 312), ('oranges', 274)]
>>> tuple(invent.items())
(('poires', 137), ('bananes', 312), ('oranges', 274))
```

- `get(cle, valeur)` : extrait du dictionnaire la valeur associée à l'argument **cle** ou retourne le second argument si la clé n'existe pas dans le dictionnaire :

```
>>> invent.get('poires', 0)
137
>>> invent.get('kiwi', 0)
0
```

- `copy()` : effectue une *vraie copie* d'un dictionnaire :

```
>>> magasin = invent.copy()
>>> magasin['prunes'] = 561
>>> magasin
{'poires': 137, 'bananes': 312, 'prunes': 561, 'oranges': 274}
>>> invent
{'poires': 137, 'bananes': 312, 'oranges': 274}
```

Rappelons que la simple affectation d'un objet existant à une nouvelle variable crée seulement **une nouvelle référence** vers le même objet, et non un nouvel objet identique. Ainsi, l'instruction ci-dessous ne définit pas un nouveau dictionnaire, contrairement aux apparences, mais un **alias** :

```
>>> stock = invent
>>> print(stock)
{'poires': 137, 'bananes': 312, 'pommes': 430, 'oranges': 274}
>>> del(invent['bananes'])
>>> print(stock)
{'poires': 137, 'pommes': 430, 'oranges': 274}
```

4.3 Parcours d'un dictionnaire

Vous pouvez utiliser une boucle `for` pour traiter successivement tous les éléments contenus dans un dictionnaire, mais attention :

- au cours de l'itération, ce sont les **clés** utilisées dans le dictionnaire qui seront successivement affectées à la variable de travail, et non les **valeurs** ;
- l'ordre dans lequel les éléments seront extraits est **imprévisible** car un dictionnaire n'est pas une séquence.

Par exemple :

```
>>> invent ={"oranges":274, "poires":137, "bananes":312}
>>> for clef in invent:
        print(clef)

poires
bananes
oranges
```

Si vous souhaitez effectuer un traitement sur les valeurs, il vous suffit alors de récupérer chacune d'elles à partir de la clé correspondante :

```
>>> for clef in invent:
        print(clef, invent[clef])

poires 137
bananes 312
oranges 274
```

Cette manière de procéder n'est cependant pas idéale, ni en termes de performances ni même du point de vue de la lisibilité. Il est recommandé de plutôt faire appel à la méthode `items()` décrite ci-dessus :

```
for clef, valeur in invent.items():
    print(clef, valeur)

poires 137
bananes 312
oranges 274
```

Dans cet exemple, la méthode `items()` appliquée au dictionnaire `invent` renvoie une séquence de tuples (clef, valeur). Le parcours effectué sur cette séquence à l'aide de la boucle `for` permet d'examiner chacun de ces tuples un par un.

Finalement, d'une manière analogue à ce qui se passe pour les chaînes, les listes et les tuples, l'instruction `in` est utilisable avec les dictionnaires. Elle permet de savoir si un dictionnaire comprend une **clé** bien déterminée :

```
>>> if "poires" in invent:
        print("Nous avons", invent['poires'], "kg de poires en stock.")
    else:
        print("Nous n'avons pas de poires en stock. Sorry")

Nous avons 137 kg de poires en stock.
```

4.4 Histogramme des fréquences d'apparition des lettres dans un texte

Comme il ne s'agit pas de séquences, les dictionnaires se révèlent précieux pour gérer des ensembles de données amenées à être fréquemment ajoutées ou supprimées dans n'importe quel ordre. Ils remplacent avantageusement les listes lorsqu'il s'agit de traiter des ensembles de données non numériques.

Afin d'illustrer l'utilisation des dictionnaires nous vous proposons d'étudier le code d'une fonction permettant d'afficher l'histogramme des fréquences d'apparition des lettres de l'alphabet dans un texte donné :

```
def histo(texte):
    "histo(str texte)-->None. Affiche l'histogramme des fréquences d'apparition des
    \ caractères composant le texte entré"

    lettres = {}

    for c in texte:
        lettres[c] = lettres.get(c, 0) + 1

    lettres_triees = list(lettres.items())
    lettres_triees.sort()

    print("Histogramme des fréquences d'apparition des caractères du texte entré: \n")
    print("Caractère \t Fréquences \t Histogramme")

    for car, freq in lettres_triees:
        print(car, "\t \t", freq, "\t \t", "*" * freq)
```

Nous commençons par créer un dictionnaire vide : `lettres`. Ensuite, nous le remplissons en utilisant les caractères de l'alphabet en guise de clés. Les valeurs que nous mémoriserons pour chacune de ces clés seront les fréquences d'apparition des caractères correspondants dans le texte. Afin de calculer celles-ci, nous effectuons

un parcours de la chaîne de caractères `texte`. Pour chacun de ces caractères, nous interrogeons le dictionnaire à l'aide de la méthode `get()`, en utilisant le caractère en guise de clé, afin d'y lire la fréquence déjà mémorisée pour ce caractère. Si cette valeur n'existe pas encore, la méthode `get()` renvoie une valeur nulle. Dans tous les cas, nous incrémentons la valeur trouvée, et nous la mémorisons dans le dictionnaire, à l'emplacement qui correspond à la clé, c'est-à-dire au caractère en cours de traitement.

Pour fignoler notre travail, nous trions l'histogramme dans l'ordre alphabétique. Nous commençons donc par convertir le dictionnaire `lettres` en une liste de tuples à l'aide de la méthode `items()` que nous trions avec la méthode `sort()`. Remarquons que cette dernière ne peut s'appliquer qu'aux listes. Finalement, l'affichage de l'histogramme se fait en parcourant la liste de tuples `lettres_triees`. Ainsi, nous obtenons :

```
>>> test = "les saucisses suisses au saloir"
>>> histo(test)
Histogramme des fréquences d'apparition des caractères du texte entré:

Caractère   Fréquences   Histogramme
            4            ****
a           3            ***
c           1            *
e           3            ***
i           3            ***
l           2            **
o           1            *
r           1            *
s           10           **********
u           3            ***
```

4.5 Contrôle du flux d'exécution à l'aide d'un dictionnaire

Il arrive fréquemment que l'on ait à diriger l'exécution d'un programme dans différentes directions, en fonction de la valeur prise par une variable. Vous pouvez bien évidemment traiter ce problème à l'aide d'une série d'instructions `if - elif - else` , mais cela peut devenir assez lourd et inélégant si vous avez affaire à un grand nombre de possibilités :

```
materiau = input("Choisissez le matériau : ")

if materiau == 'fer':
    fonctionA()
elif materiau == 'bois':
    fonctionC()
elif materiau == 'cuivre':
    fonctionB()
elif materiau == 'pierre':
    fonctionD()
elif    ... etc ...
```

Les langages de programmation proposent souvent des instructions spécifiques pour traiter ce genre de problème, telles les instructions `switch` ou `case` du **C** ou du **Pascal**. Python n'en propose aucune, mais vous pouvez vous tirer d'affaire dans bien des cas à l'aide d'un dictionnaire :

```
materiau = input("Choisissez le matériau : ")

dico = {'fer':fonctionA,
        'bois':fonctionC,
        'cuivre':fonctionB,
        'pierre':fonctionD,
           ... etc ...}

dico[materiau]()
```

Les deux instructions ci-dessus pourraient être condensées en une seule, mais nous les laissons séparées pour bien détailler le mécanisme :

- La première instruction définit un dictionnaire `dico` dans lequel les clés sont les possibilités pour la variable `materiau`, et les valeurs, les noms des fonctions à invoquer en correspondance. Notez bien qu'il s'agit seulement des **noms** de ces fonctions, qu'il ne faut surtout pas faire suivre de parenthèses dans ce cas (sinon Python exécuterait chacune de ces fonctions au moment de la création du dictionnaire).
- La seconde instruction invoque la fonction correspondant au choix opéré à l'aide de la variable `materiau`. Le nom de la fonction est extrait du dictionnaire à l'aide de la clé, puis associé à une paire de parenthèses. Python reconnaît alors un appel de fonction tout à fait classique, et l'exécute.

Vous pouvez encore améliorer la technique ci-dessus en remplaçant cette instruction par sa variante ci-dessous, qui fait appel à la méthode `get()` afin de prévoir le cas où la clé demandée n'existerait pas dans le dictionnaire (vous obtenez de cette façon l'équivalent d'une instruction `else` terminant une longue série de `elif`) :

```
dico.get(materiau, fonctAutre)()
```

Lorsque la la valeur de la variable `materiau` ne correspond à aucune clé du dictionnaire, c'est la fonction `fonctAutre()` qui est invoquée.

5 Glossaire du chapitre

La liste suivante répertorie dans l'ordre alphabétique les principaux termes introduits dans ce chapitre. La connaissance de leur définition est indispensable à la suite de votre apprentissage :

alias : référence vers un objet déjà existant.

chaine de caractères : séquence non modifiable de caractères enfermés dans une paire de guillemets.

clé : valeur de type non-modifiable indexant les éléments d'un dictionnaire.

découpage (*slicing*) : partie d'une séquence formée d'éléments contigus.

dictionnaire : tableau associatif modifiable délimité par des accolades.

index : valeur entière utilisée pour sélectionner un élément d'une séquence.

liste : séquence modifiable de valeurs de type quelconque enfermées dans un paire de crochets.

méthode : fonction associée à un objet et appelée à l'aide de la notation pointée.

référence : association entre une variable et une valeur.

séquence : suite ordonnée d'éléments indexés par des entiers.

structure de données : structure logique destinée à contenir des données afin de leur donner une organisation permettant de simplifier leur traitement.

tableau associatif : structure de données associant à un ensemble de clefs un ensemble correspondant de valeurs.

tuple : séquence non-modifiable de valeurs de type quelconque enfermées dans une paire de parenthèses.

6 Exercices

Exercice 5.1
A l'aide du *slicing*, écrivez une fonction `inverse(str texte, int n) --> str` qui découpe la chaîne de caractères `texte` en blocs de n caractères (ou moins pour le dernier bloc) et les retourne dans l'ordre inverse. Ainsi, par exemple :

```
>>> inverse("abcdefghijklmnopqrstuvwxyz123456789", 4)
'7893456yz12uvwxqrstmnopijklefghabcd'
```

Proposez une version itérative et une version récursive de la fonction `inverse`.

Exercice 5.2
Ecrivez une fonction `trouve(str texte, str car, int debut) --> int` qui fournit comme valeur de retour l'indice positif du premier caractère `car` dans la chaîne `texte` situé à partir de l'indice `debut`. Par défaut, `debut` correspondra au premier caractère de la chaîne `texte`. Si le caractère n'est pas trouvé, la fonction retournera -1. Ainsi, par exemple :

```
>>> chaine = "Juliette et Matthieu"
>>> trouve(chaine, "e", 5)
7
>>> trouve(chaine, "f")
-1
```

Exercice 5.3
Ecrivez une fonction `echange(str texte, int i, int j) --> str` qui retourne, si possible, la chaine de caractères `texte` après avoir échangé son i-ème et son j-ème caractère. La fonction retournera -1 si l'opération n'a pas pu se faire. Ainsi, par exemple :

```
>>> chaine = "Conserve"
>>> echange(chaine, 3, 6)
'Converse'
>>> echange(chaine, 3, 3)
'Conserve'
>>> echange(chaine, 3, 8)
-1
```

Exercice 5.4
Ecrivez une fonction `minMaj(str texte) --> str` qui retourne la chaine de caractères `texte` après avoir converti tous ses caractères minuscules en majuscules et inversement. Nous supposerons que la phrase entrée ne comporte pas de caractères spéciaux. Ainsi, par exemple :

```
>>> minMaj("Voici Une Phrase Test")
'vOICI uNE pHRASE tEST'
```

Exercice 5.5
A partir des listes

```
total = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
mois = ['Janvier','Février','Mars','Avril','Mai','Juin','Juillet',
'Août','Septembre','Octobre','Novembre','Décembre']
```

écrivez un programme qui affiche chaque mois de l'année en correspondance avec le nombre de jours qui le composent. Ainsi, votre programme affichera :

```
Le mois de Janvier a 31 jours.
Le mois de Février a 28 jours.
...
```

Exercice 5.6
Ecrivez une fonction `mois(int n) --> str` qui retourne le nom du $n^{\text{ème}}$ mois de l'année. Ainsi, par exemple :

```
>>> mois(5)
mai
```

Exercice 5.7
Ecrivez un programme qui analyse un par un tous les éléments d'une liste d'entiers entrés par l'utilisateur pour générer deux nouvelles listes. L'une contiendra seulement les nombres pairs de la liste initiale alors que la deuxième contiendra uniquement les impairs. Le programme affichera la liste initiale d'entiers entrés par l'utilisateur ainsi que les deux listes générées.

Exercice 5.8
Ecrivez une fonction `eleMax(list liste, int debut, int fin)--> int` qui retourne le plus grand élément présent dans la liste d'entiers `liste` entre les indices `debut` et `fin`. Par défaut, les paramètres `debut` et `fin` prennent les valeurs 0, respectivement la taille de la liste moins 1. Ainsi, par exemple :

```
>>> eleMax([32, 5, 75, 8, 3, 12, 2, 15])
75
>>> eleMax([32, 5, 75, 8, 3, 12, 2, 15], 3)
15
>>> eleMax([32, 5, 75, 8, 3, 12, 2, 15], 3, 6)
12
```

Exercice 5.9
Ecrivez une fonction `select(list liste, int taille) --> list` qui retourne la liste des mots situés dans la liste entrée en premier argument dont la taille est strictement plus petite que le deuxième argument transmis à la fonction. Ainsi, par exemple :

```
>>> select(['Jean', 'Maximilien', 'Brigitte', 'Sonia', 'Jean-Pierre', 'Sandra'], 10)
['Jean', 'Brigitte', 'Sonia', 'Sandra']
```

Exercice 5.10
Ecrivez une fonction `indexPattern(str ch, str pattern) --> list` qui retourne la liste des index de début de chaque occurrence de la sous-chaîne `pattern` dans la chaîne `ch`. Si la chaîne `ch` ne contient aucune occurrence de la sous-chaîne `pattern`, la fonction `indexPattern` retournera une liste vide. Ainsi, par exemple :

```
>>> chaine = "Tu sembles regarder les lessiveuses faire leur travail."
>>> indexPattern(chaine, 'lesi')
[]
>>> indexPattern(chaine, 'les')
[7, 20, 24]
>>> indexPattern(chaine, 'le')
[7, 20, 24, 42]
>>> indexPattern(chaine, 'l')
[7, 20, 24, 42, 53]
```

Exercice 5.11
Ecrivez une fonction `table(base, debut, fin)` qui retourne la liste des multiples du paramètre `base` situés entre les position `debut` et `fin` de la table de multiplication de `base`. Par défaut, les paramètres `debut` et `fin` auront pour valeur 1 et 10. La fonction `table` devra utiliser la fonction `range`. Ainsi, par exemple :

```
>>> table(5)
[5, 10, 15, 20, 25, 30, 35, 40, 45, 50]
>>> table(5, 7)
[35, 40, 45, 50]
>>> table(5, 3, 6)
[15, 20, 25, 30]
```

Exercice 5.12
Ecrivez une fonction `enleveDoublons(liste)` qui retourne la liste entrée en omettant ses éventuels doublons et en la retournant triée.

Exercice 5.13
Ecrivez une fonction `myMap(fonction f, list liste) --> list` qui retourne une liste contenant tous les éléments de `liste` auxquels a été appliquée la fonction f. Ainsi, par exemple :

```
>>> def impair(n): return 2*n + 1
>>> myMap(impair, [0, 1, 2, 3, 4])
[1, 3, 5, 7, 9]
>>> myMap(lambda n: n**2, [0, 1, 2, 3, 4])
[0, 1, 4, 9, 16]
```

Exercice 5.14
Ecrivez une fonction `myFilter(fonction f, list liste) --> list` qui retourne une liste contenant uniquement les éléments de `liste` retournant `True` lorsqu'ils sont donnés en arguments à la fonction `f`. Ainsi, par exemple :

```
>>> from ex4_9 import estPremier
>>> myFilter(estPremier, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15])
[2, 3, 5, 7, 11, 13]
>>> myFilter(lambda n: not n%2, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
[0, 2, 4, 6, 8, 10]
```

Exercice 5.15
Ecrivez une fonction `totalMots(str phrase) --> int` qui compte le nombre de mots composant une phrase donnée en argument et qui

(1) utilise la méthode `split`.

(2) n'utilise pas la méthode `split`.

La chaîne entrée est supposée ne contenir aucun caractère spécial à l'exception des espaces. Ainsi, par exemple

```
>>>totalMots("Ceci est une phrase permettant de tester la fonction")
9
```

Exercice 5.16
Ecrivez une fonction `select(str phrase, str car) --> list` qui retourne la liste de tous les mots de la phrase entrée qui comportent le caractère `car`. Ainsi, par exemple :

```
>>> select("Ceci est une phrase pour tester la fonction", "e")
['Ceci', 'est', 'une', 'phrase', 'tester']
```

Exercice 5.17
Ecrivez une fonction `plusLong(phrase)--> str` qui retourne le mot le plus long de la phrase entrée. Ainsi,

```
>>> plusLong("Voici une phrase permettant de tester la fonction")
'permettant'
```

Exercice 5.18
Créez une liste `A` contenant quelques éléments. Effectuez une vrai copie de cette liste dans une nouvelle variable `B`. Proposez au moins deux techniques et vérifiez que la liste `B` est une réelle copie de la `A` et non un alias.

Exercice 5.19
Ecrivez une fonction `generePremiers(int n) --> list` qui établit la liste de tous les nombres premiers compris entre 1 et `n` en utilisant la méthode du crible d'Eratosthène :

(1) Créez une liste de `n+1` éléments, chacun initialisé à 1.

(2) Parcourez cette liste à partir de l'élément d'indice 2 : si l'élément analysé possède la valeur 1, mettez à 0 tous les autres éléments de la liste dont les indices sont des multiples de l'indice auquel vous êtes arrivé.

(3) Déduisez de cette méthode la liste des nombres premiers situés entre 1 et `n`.

Exercice 5.20
En utilisant la fonction `randrange` du module `random`, écrivez un script simulant la distribution des 36 cartes d'un jeu de Jass conventionnel. Le programme tirera au hasard des cartes à jouer à partir d'une liste de couleurs (Coeur, Carreau, Trèfle et Pique) et d'une liste de valeurs (Six, Sept, Huit, ..., Roi, As). Le programme affichera par exemple :

```
>>> Appuyez sur <Enter> pour tirer une carte:
Dix de Trèfle
>>> Appuyez sur <Enter> pour tirer une carte:
As de Carreau
>>> Appuyez sur <Enter> pour tirer une carte:
Huit de Pique
>>> Appuyez sur <Enter> pour tirer une carte:
...
Toutes les cartes sont sorties.
```

Le programme ne retournera jamais une carte déjà tirée et avertira l'utilisateur lorsque toutes les cartes ont été tirées.

Exercice 5.21
Ecrivez une fonction `taille(list liste) --> list` qui, à partir d'une liste de mots, retourne une liste de tuples formés de chacun des mots et de sa taille respective. Ainsi,

```
>>> taille(['Jean', 'Maximilien', 'Brigitte', 'Sonia', 'Jean-Pierre', 'Sandra'])
[('Jean', 4), ('Maximilien', 10), ('Brigitte', 8), ('Sonia', 5),
('Jean-Pierre', 11), ('Sandra', 6)]
```

Le corps principal de votre programme permettra à l'utilisateur d'entrer tous les mots composant la liste en une seule opération :

```
Liste de mots séparés par des virgules: Jean, Maximilien, Brigitte, Sonia, Jean-Pierre
[('Jean', 4), ('Maximilien', 10), ('Brigitte', 8), ('Sonia', 5), ('Jean-Pierre', 11)]
```

Exercice 5.22
Ecrivez une fonction `division(int n, int d) --> tuple` qui retourne le quotient `q` et le reste `r` de la division entière du nombre `n` par le diviseur `d`. Ainsi, comme $22 = 5 \cdot 4 + 2$, il s'ensuit que

```
>>> division(22, 5)
(4, 2)
```

Exercice 5.23
Un tableau de m lignes et n colonnes peut être représenté en Python à l'aide d'une liste de m listes contenant chacune n éléments. Ainsi, par exemple, le tableau à 3 lignes et 4 colonnes

1	a	b	*
T	v	a	b
A	a	a	7

peut être représenté par la variable

```
>>> monTableau = [['1', 'a', 'b', '*'], ['T', 'v', 'a', 'b'], ['A', 'a', 'a', '7']]
```

Ecrivez une fonction `listeCoord(list tableau, str car) --> list of tuples` qui retourne une liste de tuples correspondant aux coordonnées (ligne ; colonne) des cellules du `tableau` contenant le caractère `car`. Ainsi, appliquée au tableau ci-dessus et au caractère `a`, la fonction `listeCoord` doit retourner

```
>>> listeCoord(monTableau, 'a')
[(1, 2), (2, 3), (3, 2), (3, 3)]
```

Le corps principal de votre programme permettra à l'utilisateur d'entrer les caractères de chaque ligne du tableau à analyser séparés par des virgules ainsi que le caractère à trouver puis affichera le tableau entré ainsi que la liste des coordonnées des cellules contenant le caractère cherché :

```
Caractères de la première ligne (<Enter> pour arrêter): 1, a, b, *
Caractères de la prochaine ligne (<Enter> pour arrêter): T, v, a, b
Caractères de la prochaine ligne (<Enter> pour arrêter): A, a, a, 7
Caractères de la prochaine ligne (<Enter> pour arrêter):

Caractère à chercher: a

Tableau à parcourir:

1 | a | b | *
-----------
T | v | a | b
-----------
A | a | a | 7

Coordonnées des cellules avec a:

[(1, 2), (2, 3), (3, 2), (3, 3)]
```

Exercice 5.24
Ecrivez une fonction `cleValeur(dict dico) --> dict` qui échange les clés et les valeurs du dictionnaire `dico`. On supposera que le dictionnaire entré ne contient pas plusieurs valeurs identiques. Ainsi,

```
>>> dico1 = {'computer': 'ordinateur', 'keyboard': 'clavier', 'mouse': 'souris'}
>>> dico2 = cleValeur(dico1)
>>> print(dico2)
{'ordinateur': 'computer', 'clavier': 'keyboard', 'souris': 'mouse'}
```

Exercice 5.25
Ecrivez une fonction `histoLettres(str texte) --> dict` qui retourne le dictionnaire des fréquences d'apparition des lettres composant la chaîne de caractères `texte`. Le dictionnaire retourné aura comme clefs les lettres de l'alphabet apparaissant au moins une fois dans le texte entré et comme valeurs la fréquence d'apparition des lettres dans le texte entré. Les espaces du texte ne seront pas comptabilisés. Ainsi,

```
>>> histoLettres("les saucisses suisses au saloir")
{'a': 3, 'c': 1, 'e': 3, 'i': 3, 'l': 2, 'o': 1, 's': 10, 'r': 1, 'u': 3}
```

Le corps principal du programme permettra d'afficher, par ordre alphabétique, l'histogramme des fréquences d'apparition des lettres d'un texte entré par l'utilisateur.

Exercice 5.26
Ecrivez une fonction `histoMots(str texte) --> dict` qui retourne le dictionnaire des fréquences d'apparition des mots composant la chaîne de caractères `texte`. Le corps principal du programme affichera l'histogramme des fréquences d'apparition des mots composant le texte entré par l'utilisateur par ordre décroissant de fréquence selon le format suivant :

```
Entrez un texte: "le corbeau et le renard"
Histogramme des fréquences d'apparition des mots du texte entré:

Mot       Fréquence      Histogramme
le        2              **
corbeau   1              *
et        1              *
renard    1              *
```

Exercice 5.27
Ecrivez une fonction `positionsMots(str phrase) --> dict` qui prend en argument une chaîne de caractères `phrase` et qui retourne un dictionnaire répertoriant pour chaque mot composant `phrase` la liste de ses positions d'apparition dans `phrase`. Le premier mot d'une phrase sera considérée en position 1.

```
>>> positionsMots("la vie est à la vie ce que la mort est à la mort")
{'à': [4, 12], 'est': [3, 11], 'la': [1, 5, 9, 13], 'vie': [2, 6], 'ce': [7],
'mort': [10, 14], 'que': [8]}
```

La partie principale de votre programme utilisera la fonction `positionsMots` pour afficher le mot apparaissant le plus souvent dans une phrase entrée par l'utilisateur, sa fréquence d'apparition et la position de sa dernière occurrence dans la phrase. L'affichage du résultat respectera le format de l'exemple suivant :

```
>>> Entrez une phrase: la vie est à la vie ce que la mort est à la mort
    Le mot "la" apparait 4 fois dont une dernière à la position 13.
```