5

Principaux types de données

Dans le chapitre 2, nous avons déjà manipulé des données de différents types : des nombres entiers ou réels, et des chaînes de caractères. Il est temps à présent d'examiner d'un peu plus près ces types de données, et également de vous en faire découvrir d'autres.

Les données numériques

Dans les exercices réalisés jusqu'à présent, nous avons déjà utilisé des données de deux types : les nombres *entiers* ordinaires et les nombres *réels* (aussi appelés nombres *à virgule flottante*). Tâchons de mettre en évidence les caractéristiques (et les limites) de ces concepts.

Le type integer

Supposons que nous voulions modifier légèrement notre précédent exercice sur la suite de Fibonacci, de manière à obtenir l'affichage d'un plus grand nombre de termes. A priori, il suffit de modifier la condition de bouclage, dans la deuxième ligne. Avec `while c <50:` , nous devrions obtenir quarante-neuf termes. Modifions donc légèrement l'exercice, de manière à afficher aussi le type de la variable principale :

```
>>> a, b, c = 1, 1, 1
>>> while c <50:
        print(c, ":", b, type(b))
        a, b, c = b, a+b, c+1
...
...
...  (affichage des 43 premiers termes)
...
44 : 1134903170 <class 'int'>
45 : 1836311903 <class 'int'>
46 : 2971215073 <class 'int'>
47 : 4807526976 <class 'int'>
48 : 7778742049 <class 'int'>
49 : 12586269025 <class 'int'>
```

Que pouvons-nous constater ?

Il semble que Python soit capable de traiter des nombres entiers de taille illimitée. La fonction **type()** nous permet de vérifier à chaque itération que le type de la variable **b** reste bien en permanence de ce type.

L'exercice que nous venons de réaliser pourrait cependant intriguer ceux d'entre vous qui s'interrogent sur la représentation « interne » des nombres dans un ordinateur. Vous savez probablement en effet que le « cœur » de celui-ci est constitué par un circuit intégré électronique (une « puce » de silicium) à très haut degré d'intégration, qui peut effectuer plus d'un milliard d'opérations en une seule seconde, mais seulement sur des nombres binaires de taille limitée : 32 bits actuellement[19]. Or, la gamme de valeurs décimales qu'il est possible d'encoder sous forme de nombres binaires de 32 bits s'étend de -2147483648 à +2147483647.

Les opérations effectuées sur des entiers compris entre ces deux limites sont donc toujours très rapides, parce que le processeur est capable de les traiter directement. En revanche, lorsqu'il est question de traiter des nombres entiers plus grands, ou encore des nombres réels (nombres « à virgule flottante »), les logiciels que sont les interpréteurs et compilateurs doivent effectuer un gros travail de codage/décodage, afin de ne présenter en définitive au processeur que des opérations binaires sur des nombres entiers, de 32 bits au maximum.

Vous n'avez pas à vous préoccuper de ces considérations techniques. Lorsque vous lui demandez de traiter des entiers quelconques, Python les transmet au processeur sous la forme de nombres binaires de 32 bits chaque fois que cela est possible, afin d'optimiser la vitesse de calcul et d'économiser l'espace mémoire. Lorsque les valeurs à traiter sont des nombres entiers se situant au-delà des limites indiquées plus haut, leur encodage dans la mémoire de l'ordinateur devient plus complexe, et leur traitement par le processeur nécessite alors plusieurs opérations successives, mais tout cela se fait automatiquement, sans que vous n'ayez à vous en soucier[20].

Vous pouvez donc effectuer avec Python des calculs impliquant des valeurs entières comportant un nombre de chiffres significatifs quelconque. Ce nombre n'est limité en effet *que par la taille de la mémoire disponible sur l'ordinateur utilisé*. Il va de soi cependant que les calculs impliquant de très grands nombres devront être décomposés par l'interpréteur en calculs multiples sur des nombres plus simples, ce qui pourra nécessiter un temps de traitement considérable dans certains cas.

Exemple :

```
>>> a, b, c = 3, 2, 1
>>> while c < 15:
        print(c, ": ", b)
        a, b, c = b, a*b, c+1

1 :  2
2 :  6
3 :  12
4 :  72
5 :  864
6 :  62208
7 :  53747712
8 :  3343537668096
9 :  179707499645975396352
10 :  60085879430566732227015542518579 2
```

[19] La plupart des ordinateurs de bureau actuels contiennent un microprocesseur à registres de 32 bits (même s'il s'agit d'un modèle « dual core ». Les processeurs « 64 bits » seront cependant bientôt monnaie courante.

[20] Les précédentes versions de Python disposaient de deux types d'entiers : « integer » et « long integer », mais la conversion entre ces deux types est devenue automatique dès la version 2.2.

```
11 :  1079788315649669138143849229447384578592430704390307 84
12 :  6488003054466075279073683736910497769500103428422804289182764945618623 4
582611607420928
13 :  7005669890111832002923764139957621692162454505797269791738369231327175 4
883621235064434673400268965204696103008832506249008437424702378475 52
14 :  4545280764562657998563629404824935120516823987072294615140165565565839 8
642227616335815123825782460196980206141536747116094173550514227947953005917 00
969504226930790382476340558291752968319462245039335017547760330040127583682 56
>>>
```

Dans l'exemple ci-dessus, la valeur des nombres affichés augmente très rapidement, car chacun d'eux est égal au produit des deux termes précédents.

Vous pouvez bien évidemment continuer cette suite mathématique plus loin si vous voulez. La progression continue avec des nombres de plus en plus gigantesques, mais la vitesse de calcul diminue au fur et à mesure.

Note complémentaire : les entiers de valeur comprise entre les deux limites indiquées plus haut occupent chacun 32 bits dans la mémoire de l'ordinateur. Les très grands entiers occupent une place variable, en fonction de leur taille.

Le type float

Vous avez déjà rencontré précédemment cet autre type de donnée numérique : le type « nombre réel », ou « nombre à virgule flottante », désigné en anglais par l'expression *floating point number*, et que pour cette raison on appellera type **float** sous Python.

Ce type autorise les calculs sur de très grands ou très petits nombres (données scientifiques, par exemple), avec un degré de précision constant.

Pour qu'une donnée numérique soit considérée par Python comme étant du type **float**, il suffit qu'elle contienne dans sa formulation un élément tel qu'un point décimal ou un exposant de 10.

Par exemple, les données :

```
      3.14      10.      .001      1e100      3.14e-10
```

sont automatiquement interprétées par Python comme étant du type ***float***.

Essayons donc ce type de données dans un nouveau petit programme (inspiré du précédent) :

```
>>> a, b, c = 1., 2., 1              # => a et b seront du type 'float'
>>> while c <18:
...     a, b, c = b, b*a, c+1
...     print(b)

2.0
4.0
8.0
32.0
256.0
8192.0
2097152.0
17179869184.0
3.6028797019e+16
```

```
6.18970019643e+26
2.23007451985e+43
1.38034926936e+70
3.07828173409e+113
4.24910394253e+183
1.30799390526e+297
         Inf
         Inf
```

Comme vous l'aurez certainement bien compris, nous affichons cette fois encore une série dont les termes augmentent extrêmement vite, chacun d'eux étant égal au produit des deux précédents. Au huitième terme, nous dépassons déjà largement la capacité d'un *integer*. Au neuvième terme, Python passe automatiquement à la notation scientifique (« e+n » signifie en fait : « fois dix à l'exposant n »). Après le quinzième terme, nous assistons à nouveau à un dépassement de capacité (sans message d'erreur) : les nombres vraiment trop grands sont tout simplement notés « inf » (pour « infini »).

Le type *float* utilisé dans notre exemple permet de manipuler des nombres (positifs ou négatifs) compris entre 10^{-308} et 10^{308} avec une précision de 12 chiffres significatifs. Ces nombres sont encodés d'une manière particulière sur 8 octets (64 bits) dans la mémoire de la machine : une partie du code correspond aux 12 chiffres significatifs, et une autre à l'ordre de grandeur (exposant de 10).

Exercices

5.1 Écrivez un programme qui convertisse en radians un angle fourni au départ en degrés, minutes, secondes.

5.2 Écrivez un programme qui convertisse en degrés, minutes, secondes un angle fourni au départ en radians.

5.3 Écrivez un programme qui convertisse en degrés Celsius une température exprimée au départ en degrés Fahrenheit, ou l'inverse.
La formule de conversion est : $T_F = T_C \times 1{,}8 + 32$.

5.4 Écrivez un programme qui calcule les intérêts accumulés chaque année pendant 20 ans, par capitalisation d'une somme de 100 euros placée en banque au taux fixe de 4,3 %

5.5 Une légende de l'Inde ancienne raconte que le jeu d'échecs a été inventé par un vieux sage, que son roi voulut remercier en lui affirmant qu'il lui accorderait n'importe quel cadeau en récompense. Le vieux sage demanda qu'on lui fournisse simplement un peu de riz pour ses vieux jours, et plus précisément un nombre de grains de riz suffisant pour que l'on puisse en déposer 1 seul sur la première case du jeu qu'il venait d'inventer, deux sur la suivante, quatre sur la troisième, et ainsi de suite jusqu'à la 64e case.
Écrivez un programme Python qui affiche le nombre de grains à déposer sur chacune des 64 cases du jeu. Calculez ce nombre de deux manières :
- le nombre exact de grains (nombre entier) ;
- le nombre de grains en notation scientifique (nombre réel).

Les données alphanumériques

Jusqu'à présent nous n'avons manipulé que des nombres. Mais un programme d'ordinateur peut également traiter des caractères alphabétiques, des mots, des phrases, ou des suites de symboles quelconques. Dans la plupart des langages de programmation, il existe pour cet usage des structures de données particulières que l'on appelle « chaînes de caractères ».

> *Nous apprendrons plus loin (au chapitre 10) qu'il ne faut pas confondre les notions de « chaîne de caractères » et « séquence d'octets » comme le faisaient abusivement les langages e programmation anciens (dont les premières versions de Python). Pour l'instant, contentons-nous de nous réjouir que Python traite désormais de manière parfaitement cohérente toutes les chaînes de caractères, ceux-ci pouvant faire partie d'alphabets quelconques*[21].

Le type string

Une donnée de type *string* peut se définir en première approximation comme une suite quelconque de caractères. Dans un script python, on peut délimiter une telle suite de caractères, soit par des apostrophes (simple quotes), soit par des guillemets (double quotes). Exemples :

```
>>> phrase1 = 'les oeufs durs.'
>>> phrase2 = '"Oui", répondit-il,'
>>> phrase3 = "j'aime bien"
>>> print(phrase2, phrase3, phrase1)
"Oui", répondit-il, j'aime bien les oeufs durs.
```

Les 3 variables **phrase1**, **phrase2**, **phrase3** sont donc des variables de type *string*.

Remarquez l'utilisation des guillemets pour délimiter une chaîne dans laquelle il y a des apostrophes, ou l'utilisation des apostrophes pour délimiter une chaîne qui contient des guillemets. Remarquez aussi encore une fois que la fonction **print()** insère un espace entre les éléments affichés.

Le caractère spécial « \ » (*antislash*) permet quelques subtilités complémentaires :

- En premier lieu, il permet d'écrire sur plusieurs lignes une commande qui serait trop longue pour tenir sur une seule (cela vaut pour n'importe quel type de commande).
- À l'intérieur d'une chaîne de caractères, l'*antislash* permet d'insérer un certain nombre de codes spéciaux (sauts à la ligne, apostrophes, guillemets, etc.). Exemples :

```
>>> txt3 = '"N\'est-ce pas ?" répondit-elle.'
>>> print(txt3)
"N'est-ce pas ?" répondit-elle.
```

[21] Ceci constitue donc l'une des grandes nouveautés de la version 3 de Python par rapport aux versions précédentes. Dans celles-ci, une donnée de type *string* était en réalité une *séquence d'octets* et non une *séquence de caractères*. Cela ne posait guère de problèmes pour traiter des textes contenant seulement les caractères principaux des langues d'Europe occidentale, car il était possible d'encoder chacun de ces caractères sur un seul octet (en suivant par exemple la norme Latin-1). Cela entraînait cependant de grosses difficultés si l'on voulait rassembler dans un même texte des caractères tirés d'alphabets différents, ou simplement utiliser des alphabets comportant plus de 256 caractères, des symboles mathématiques particuliers, etc. (Vous trouverez davantage d'informations à ce sujet au chapitre 10).

```
>>> Salut = "Ceci est une chaîne plutôt longue\n contenant plusieurs lignes \
... de texte (Ceci fonctionne\n de la même façon en C/C++.\n\
...     Notez que les blancs en début\n de ligne sont significatifs.\n"
>>> print(Salut)
Ceci est une chaîne plutôt longue
 contenant plusieurs lignes de texte (Ceci fonctionne
 de la même façon en C/C++.
    Notez que les blancs en début
 de ligne sont significatifs.
```

Remarques

- La séquence `\n` dans une chaîne provoque un saut à la ligne.
- La séquence `\'` permet d'insérer une apostrophe dans une chaîne délimitée par des apostrophes. De la même manière, la séquence `\"` permet d'insérer des guillemets dans une chaîne délimitée elle-même par des guillemets.
- Rappelons encore ici que la casse est significative dans les noms de variables (il faut respecter scrupuleusement le choix initial de majuscules ou minuscules).

Triple quotes

Pour insérer plus aisément des caractères spéciaux ou « exotiques » dans une chaîne, sans faire usage de l'*antislash*, ou pour faire accepter l'*antislash* lui-même dans la chaîne, on peut encore délimiter la chaîne à l'aide de *triples guillemets* ou de *triples apostrophes* :

```
>>> a1 = """
... Usage: trucmuche[OPTIONS]
... { -h
...   -H hôte
... }"""

>>> print(a1)

Usage: trucmuche[OPTIONS]
{ -h
  -H hôte
}
```

Accès aux caractères individuels d'une chaîne

Les chaînes de caractères constituent un cas particulier d'un type de données plus général que l'on appelle des *données composites*. Une donnée composite est une entité qui rassemble dans une seule structure un ensemble d'entités plus simples : dans le cas d'une chaîne de caractères, par exemple, ces entités plus simples sont évidemment les caractères eux-mêmes. En fonction des circonstances, nous souhaiterons traiter la chaîne de caractères, tantôt comme un seul objet, tantôt comme une collection de caractères distincts. Un langage de programmation tel que Python doit donc être pourvu de mécanismes qui permettent d'accéder séparément à chacun des caractères d'une chaîne. Comme vous allez le voir, cela n'est pas bien compliqué.

Python considère qu'une chaîne de caractères est un objet de la catégorie des *séquences*, lesquelles sont des *collections ordonnées d'éléments*. Cela signifie simplement que les caractères

d'une chaîne sont toujours disposés dans un certain ordre. Par conséquent, chaque caractère de la chaîne peut être désigné par sa place dans la séquence, à l'aide d'un *index*.

Pour accéder à un caractère bien déterminé, on utilise le nom de la variable qui contient la chaîne et on lui accole, entre deux crochets, l'index numérique qui correspond à la position du caractère dans la chaîne.

Attention cependant : comme vous aurez l'occasion de le vérifier par ailleurs, les données informatiques sont presque toujours numérotées *à partir de zéro* (et non à partir de un). C'est le cas pour les caractères d'une chaîne.

Exemple :

```
>>> ch = "Christine"
>>> print(ch[0], ch[3], ch[5])
C i t
```

Vous pouvez recommencer l'exercice de l'exemple ci-dessus en utilisant cette fois un ou deux caractères « non-ASCII », tels que lettres accentuées, cédilles, etc. Contrairement à ce qui pouvait se passer dans certains cas avec les versions de Python antérieures à la version 3.0, vous obtenez sans surprise les résultats attendus :

```
>>> ch ="Noël en Décembre"
>>> print(ch[1],ch[2],ch[3],ch[4],ch[8],ch[9],ch[10],ch[11],ch[12])
o ë l   D é c e m
```

Ne vous préoccupez pas pour l'instant de savoir de quelle manière exacte Python mémorise et traite les caractères typographiques dans la mémoire de l'ordinateur. Sachez cependant que la technique utilisée exploite la norme internationale **unicode**. qui permet de distinguer de façon univoque n'importe quel caractère de n'importe quel alphabet. Vous pourrez donc mélanger dans une même chaîne des caractères latins, grecs, cyrilliques, arabes, ... (y compris les caractères accentués), ainsi que des symboles mathématiques, des pictogrammes, etc. Nous verrons au chapitre 10 (voir page 123) comment faire apparaître d'autres caractères que ceux qui sont directement accessibles au clavier.

Opérations élémentaires sur les chaînes

Python intègre de nombreuses *fonctions* qui permettent d'effectuer divers traitements sur les chaînes de caractères (conversions majuscules/minuscules, découpage en chaînes plus petites, recherche de mots, etc.). Une fois de plus, cependant, nous devons vous demander de patienter : ces questions ne seront développées qu'à partir du chapitre 10 (voir page 123).

Pour l'instant, nous pouvons nous contenter de savoir qu'il est possible d'accéder individuellement à chacun des caractères d'une chaîne, comme cela a été expliqué dans la section précédente. Sachons en outre que l'on peut aussi :

- assembler plusieurs petites chaînes pour en construire de plus grandes. Cette opération s'appelle *concaténation* et on la réalise sous Python à l'aide de l'opérateur + (cet opérateur réalise donc l'opération d'addition lorsqu'on l'applique à des nombres, et l'opération de concaténation lorsqu'on l'applique à des chaînes de caractères). Exemple :

```
a = 'Petit poisson'
b = ' deviendra grand'
c = a + b
print(c)
petit poisson deviendra grand
```

- déterminer la longueur (c'est-à-dire le nombre de caractères) d'une chaîne, en faisant appel à la fonction intégrée **len()** :

```
>>> ch ='Georges'
>>> print(len(ch))
7
```

Cela marche tout aussi bien si la chaîne contient des caractères accentués :

```
>>> ch ='René'
>>> print(len(ch))
4
```

- Convertir en nombre véritable une chaîne de caractères qui représente un nombre. Exemple :

```
>>> ch = '8647'
>>> print(ch + 45)
```

→ *** erreur *** : on ne peut pas additionner une chaîne et un nombre

```
>>> n = int(ch)
>>> print(n + 65)
8712                    # OK : on peut additionner 2 nombres
```

Dans cet exemple, la fonction intégrée **int()** convertit la chaîne en nombre entier. Il serait également possible de convertir une chaîne de caractères en nombre réel, à l'aide de la fonction intégrée **float()**.

Exercices

5.6 Écrivez un script qui détermine si une chaîne contient ou non le caractère « e ».

5.7 Écrivez un script qui compte le nombre d'occurrences du caractère « e » dans une chaîne.

5.8 Écrivez un script qui recopie une chaîne (dans une nouvelle variable), en insérant des astérisques entre les caractères.
Ainsi par exemple, « `gaston` » devra devenir « `g*a*s*t*o*n` »

5.9 Écrivez un script qui recopie une chaîne (dans une nouvelle variable) en l'inversant.
Ainsi par exemple, « `zorglub` » deviendra « `bulgroz` ».

5.10 En partant de l'exercice précédent, écrivez un script qui détermine si une chaîne de caractères donnée est un *palindrome* (c'est-à-dire une chaîne qui peut se lire indifféremment dans les deux sens), comme par exemple « radar » ou « s.o.s ».

Les listes (première approche)

Les chaînes que nous avons abordées à la rubrique précédente constituaient un premier exemple de *données composites*. On appelle ainsi les structures de données qui sont utilisées pour regrou-

per de manière structurée des ensembles de valeurs. Vous apprendrez progressivement à utiliser plusieurs autres types de données composites, parmi lesquelles les *listes*, les *tuples* et les *dictionnaires*[22]. Nous n'allons cependant aborder ici que le premier de ces trois types, et ce de façon assez sommaire. Il s'agit-là en effet d'un sujet fort vaste, sur lequel nous devrons revenir à plusieurs reprises.

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

```
>>> jour = ['lundi', 'mardi', 'mercredi', 1800, 20.357, 'jeudi', 'vendredi']
>>> print(jour)
['lundi', 'mardi', 'mercredi', 1800, 20.357, 'jeudi', 'vendredi']
```

Dans cet exemple, la valeur de la variable **jour** est une liste.

Comme on peut le constater dans le même exemple, les éléments individuels qui constituent une liste peuvent être de types variés. Dans cet exemple, en effet, les trois premiers éléments sont des chaînes de caractères, le quatrième élément est un entier, le cinquième un réel, etc. (nous verrons plus loin qu'un élément d'une liste peut lui-même être une liste !). À cet égard, le concept de liste est donc assez différent du concept de « tableau » (*array*) ou de « variable indicée » que l'on rencontre dans d'autres langages de programmation.

Remarquons aussi que, comme les chaînes de caractères, les listes sont des *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 l'on peut donc accéder à chacun d'entre eux individuellement si l'on connaît son *index* dans la liste. 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.

Exemples :

```
>>> jour = ['lundi', 'mardi', 'mercredi', 1800, 20.357, 'jeudi', 'vendredi']
>>> print(jour[2])
mercredi
>>> print(jour[4])
20.357
```

À la différence de ce qui se passe pour les chaînes, qui constituent un type de données *non-modifiables* (nous aurons plus loin diverses occasions de revenir là-dessus), il est possible de changer les éléments individuels d'une liste :

```
>>> print(jour)
['lundi', 'mardi', 'mercredi', 1800, 20.357, 'jeudi', 'vendredi']
>>> jour[3] = jour[3] +47
>>> print(jour)
['lundi', 'mardi', 'mercredi', 1847, 20.357, 'jeudi', 'vendredi']
```

On peut donc remplacer certains éléments d'une liste par d'autres, comme ci-dessous :

```
>>> jour[3] = 'Juillet'
>>> print(jour)
['lundi', 'mardi', 'mercredi', 'Juillet', 20.357, 'jeudi', 'vendredi']
```

[22] Vous pourrez même créer vos propres types de données composites, lorsque vous aurez assimilé le concept de ***classe*** (voir page 163).

La *fonction intégrée* **len()**, que nous avons déjà rencontrée à propos des chaînes, s'applique aussi aux listes. Elle renvoie le nombre d'éléments présents dans la liste :

```
>>> print(len(jour))
7
```

Une autre *fonction intégrée* permet de supprimer d'une liste un élément quelconque (à partir de son index). Il s'agit de la fonction **del()** [23] :

```
>>> del(jour[4])
>>> print(jour)
['lundi', 'mardi', 'mercredi', 'juillet', 'jeudi', 'vendredi']
```

Il est également tout à fait possible d'ajouter un élément à une liste, mais pour ce faire, il faut considérer que la liste est un *objet*, dont on va utiliser l'une des *méthodes*. Les concepts informatiques *d'objet* et de *méthode* ne seront expliqués qu'un peu plus loin dans ces notes, mais nous pouvons dès à présent montrer « comment ça marche » dans le cas particulier d'une liste :

```
>>> jour.append('samedi')
>>> print(jour)
['lundi', 'mardi', 'mercredi', 'juillet', 'jeudi', 'vendredi', 'samedi']
>>>
```

Dans la première ligne de l'exemple ci-dessus, nous avons appliqué la *méthode* **append()** à *l'objet* **jour**, avec *l'argument* 'samedi'. Si l'on se rappelle que le mot « append » signifie « ajouter » en anglais, on peut comprendre que la méthode **append()** est une sorte de *fonction* qui est en quelque manière attachée ou intégrée aux objets du type « liste ». L'argument que l'on utilise avec cette fonction est bien entendu l'élément que l'on veut ajouter à la fin de la liste.

Nous verrons plus loin qu'il existe ainsi toute une série de ces *méthodes* (c'est-à-dire des fonctions intégrées, ou plutôt « encapsulées » dans les objets de type « liste »). Notons simplement au passage que l'on applique une méthode à un objet *en reliant les deux à l'aide d'un point*. (D'abord le nom de la variable qui référence l'objet, puis le point, puis le nom de la méthode, cette dernière toujours accompagnée d'une paire de parenthèses.)

Comme les chaînes de caractères, les listes seront approfondies plus loin dans ces notes (voir page 143). Nous en savons cependant assez pour commencer à les utiliser dans nos programmes. Veuillez par exemple analyser le petit script ci-dessous et commenter son fonctionnement :

```
jour = ['dimanche','lundi','mardi','mercredi','jeudi','vendredi','samedi']
a, b = 0, 0
while a<25:
     a = a + 1
     b = a % 7
     print(a, jour[b])
```

[23] Il existe en fait tout un ensemble de techniques qui permettent de découper une liste en tranches, d'y insérer des groupes d'éléments, d'en enlever d'autres, etc., en utilisant une syntaxe particulière où n'interviennent que les index.
Cet ensemble de techniques (qui peuvent aussi s'appliquer aux chaînes de caractères) porte le nom générique de *slicing* (tranchage). On le met en œuvre en plaçant plusieurs indices au lieu d'un seul entre les crochets que l'on accole au nom de la variable. Ainsi jour[1:3] désigne le sous-ensemble ['mardi', 'mercredi'].
Ces techniques un peu particulières sont décrites plus loin (voir pages 123 et suivantes).

La 5e ligne de cet exemple fait usage de l'opérateur « *modulo* » déjà rencontré précédemment et qui peut rendre de grands services en programmation. On le représente par % dans de nombreux langages (dont Python). Quelle est l'opération effectuée par cet opérateur ?

Exercices

5.11 Soient les listes suivantes :

```
t1 = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
t2 = ['Janvier', 'Février', 'Mars', 'Avril', 'Mai', 'Juin',
      'Juillet', 'Août', 'Septembre', 'Octobre', 'Novembre', 'Décembre']
```

Écrivez un petit programme qui crée une nouvelle liste **t3**. Celle-ci devra contenir tous les éléments des deux listes en les alternant, de telle manière que chaque nom de mois soit suivi du nombre de jours correspondant :

```
['Janvier',31,'Février',28,'Mars',31, etc...].
```

5.12 Écrivez un programme qui affiche « proprement » tous les éléments d'une liste. Si on l'appliquait par exemple à la liste **t2** de l'exercice ci-dessus, on devrait obtenir :

```
Janvier Février Mars Avril Mai Juin Juillet Août Septembre Octobre
Novembre Décembre
```

5.13 Écrivez un programme qui recherche le plus grand élément présent dans une liste donnée. Par exemple, si on l'appliquait à la liste `[32, 5, 12, 8, 3, 75, 2, 15]`, ce programme devrait afficher :

```
le plus grand élément de cette liste a la valeur 75.
```

5.14 Écrivez un programme qui analyse un par un tous les éléments d'une liste de nombres (par exemple celle de l'exercice précédent) pour générer deux nouvelles listes. L'une contiendra seulement les nombres *pairs* de la liste initiale, et l'autre les nombres *impairs*. Par exemple, si la liste initiale est celle de l'exercice précédent, le programme devra construire une liste **pairs** qui contiendra `[32, 12, 8, 2]`, et une liste **impairs** qui contiendra `[5, 3, 75, 15]`. Astuce : pensez à utiliser l'opérateur **modulo** (%) déjà cité précédemment.

5.15 Écrivez un programme qui analyse un par un tous les éléments d'une liste de mots (par exemple : `['Jean', 'Maximilien', 'Brigitte', 'Sonia', 'Jean-Pierre', 'Sandra']`) pour générer deux nouvelles listes. L'une contiendra les mots comportant moins de 6 caractères, l'autre les mots comportant 6 caractères ou davantage.