10

Approfondir les structures de données

Jusqu'à présent, nous nous sommes contentés d'opérations assez simples. Nous allons maintenant passer à la vitesse supérieure. Les structures de données que vous utilisez déjà présentent quelques caractéristiques que vous ne connaissez pas encore, et il est également temps de vous faire découvrir d'autres structures plus complexes.

Le point sur les chaînes de caractères

Nous avons déjà rencontré les chaînes de caractères au chapitre 5. 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.

Indiçage, extraction, longueur

Petit rappel du chapitre 5 : les chaînes sont des *séquences* 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 :

```
>>> nom = 'Cédric'
>>> print(nom[1], nom[3], nom[5])
é r c
```

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 : ainsi -1 désignera le dernier caractère, -2 l'avant-dernier, etc. :

```
>>> print(nom[-1], nom[-2], nom[-4], nom[-6])
c i d C
>>>
```

Si l'on désire déterminer le nombre de caractères présents dans une chaîne, on utilise la fonction intégrée **len()** :

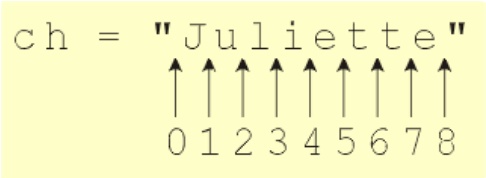
```
>>> print(len(nom))
6
```

Extraction de fragments de chaînes

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* (« 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 :

```
>>> ch = "Juliette"
>>> print(ch[0:3])
Jul
```

Dans la tranche **[n,m]**, le $n^{ième}$ caractère est inclus, mais pas le $m^{ième}$. Si vous voulez mémoriser aisément ce mécanisme, il faut vous représenter que les indices pointent des emplacements situés ***entre*** les caractères, comme dans le schéma ci-dessous :



Au vu de ce schéma, il n'est pas difficile de comprendre que **ch[3:7]** extraira « **iett** »

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(ch[:3])        # les 3 premiers caractères
Jul
>>> print(ch[3:])        # tout ce qui suit les 3 premiers caractères
iette
```

Les caractères accentués ne doivent pas faire problème :

```
>>> ch = 'Adélaïde'
>>> print(ch[:3], ch[4:8])
Adé aïde
```

Concaténation, répétition

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 !
```

Remarquez au passage 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 : on doit alors utiliser des symboles différents pour l'addition et la concaténation, par exemple.

Exercices

10.1 Déterminez vous-même ce qui se passe, dans la technique de *slicing*, lorsque l'un ou l'autre des indices de découpage est erroné, et décrivez cela le mieux possible. (Si le second indice est plus petit que le premier, par exemple, ou bien si le second indice est plus grand que la taille de la chaîne).

10.2 Découpez une grande chaîne en fragments de 5 caractères chacun. Rassemblez ces morceaux dans l'ordre inverse. La chaîne doit pouvoir contenir des caractères accentués.

10.3 Tâchez d'écrire une petite fonction **trouve()** qui fera exactement le contraire de ce que fait l'opérateur d'indexage (c'est-à-dire les crochets **[ ]**). Au lieu de partir d'un index donné pour retrouver le caractère correspondant, cette fonction devra retrouver l'index correspondant à un caractère donné.
En d'autres termes, il s'agit d'écrire une fonction qui attend deux arguments : le nom de la chaîne à traiter et le caractère à trouver. La fonction doit fournir en retour l'index du premier caractère de ce type dans la chaîne. Ainsi par exemple, l'instruction :

```
print(trouve("Juliette & Roméo", "&"))
```

devra afficher : **9**
Attention : il faut penser à tous les cas possibles. Il faut notamment veiller à ce que la fonction renvoie une valeur particulière (par exemple la valeur -1) si le caractère recherché n'existe pas dans la chaîne traitée. Les caractères accentués doivent être acceptés.

10.4 Écrivez une fonction **compteCar()** qui compte le nombre d'occurrences d'un caractère donné dans une chaîne. Ainsi :

```
print(compteCar("ananas au jus","a"))
```

devra afficher : **4**

```
print(compteCar("Gédéon est déjà là","é"))
```

devra afficher : **3.**

Parcours d'une séquence : l'instruction for ... in ...

Il arrive très souvent que l'on doive traiter l'intégralité d'une chaîne caractère par caractère, du premier jusqu'au dernier, pour effectuer à partir de chacun d'eux une opération quelconque. Nous appellerons cette opération un *parcours*. En nous limitant aux outils Python que nous connaissons déjà, nous pouvons envisager d'encoder un tel parcours à l'aide d'une boucle, articulée autour de l'instruction **while** :

```
>>> nom ="Joséphine"
>>> index =0
>>> while index < len(nom):
...   print(nom[index] + ' *', end =' ')
...   index = index +1
...
J * o * s * é * p * h * i * n * e *
```

Cette boucle *parcourt* donc la chaîne **nom** pour en extraire un à un tous les caractères, lesquels sont ensuite imprimés avec interposition d'astérisques. Notez bien que la condition utilisée avec l'instruction **while** est `index < len(nom)`, ce qui signifie que le bouclage doit s'effectuer jusqu'à ce que l'on soit arrivé à l'indice numéro 9 (la chaîne compte en effet 10 caractères). Nous aurons effectivement traité tous les caractères de la chaîne, puisque ceux-ci sont indicés de 0 à 9.

Le *parcours d'une séquence* est une opération très fréquente en programmation. Pour en faciliter l'écriture, Python vous propose une structure de boucle plus appropriée que la boucle **while**, basée sur le couple d'instructions **for ... in ... :**

Avec ces instructions, le programme ci-dessus devient :

```
>>> nom ="Cléopâtre"
>>> for car in nom:
...     print(car + ' *', end =' ')
...
C * l * é * o * p * â * t * r * e *
```

Comme vous pouvez le constater, cette structure de boucle est plus compacte. Elle vous évite d'avoir à définir et à incrémenter une variable spécifique (un « compteur ») pour gérer l'indice du caractère que vous voulez traiter à chaque itération (c'est Python qui s'en charge). La structure **for ... in ...** ne montre donc que l'essentiel, à savoir que variable **car** contiendra successivement tous les caractères de la chaîne, du premier jusqu'au dernier.

L'instruction **for** permet donc d'écrire des boucles, dans lesquelles *l'itération traite successivement tous les éléments d'une séquence donnée*. Dans l'exemple ci-dessus, la séquence était une chaîne de caractères. L'exemple ci-après démontre que l'on peut appliquer le même traitement aux *listes* (et il en sera de même pour les *tuples* étudiés plus loin) :

```
liste = ['chien', 'chat', 'crocodile', u'éléphant']
for animal in liste:
    print('longueur de la chaîne', animal, '=', len(animal))
```

L'exécution de ce script donne :

```
longueur de la chaîne chien = 5
longueur de la chaîne chat = 4
longueur de la chaîne crocodile = 9
longueur de la chaîne éléphant = 8
```

L'instruction **for ... in ... :** est un nouvel exemple d'*instruction composée*. N'oubliez donc pas le double point obligatoire à la fin de la ligne, et l'indentation pour le bloc d'instructions qui suit.

Le nom qui suit le mot réservé **in** est celui de la séquence qu'il faut traiter. Le nom qui suit le mot réservé **for** est celui que vous choisissez pour la *variable* destinée à contenir successivement

tous les éléments de la séquence. Cette variable est définie automatiquement (c'est-à-dire qu'il est inutile de la définir au préalable), et *son type est automatiquement adapté* à celui de l'élément de la séquence qui est en cours de traitement (rappelons en effet que dans le cas d'une liste, tous les éléments ne sont pas nécessairement du même type). Exemple :

```
divers = ['lézard', 3, 17.25, [5, 'Jean']]
for e in divers:
    print(e, type(e))
```

L'exécution de ce script donne :

```
lézard <class 'str'>
3 <class 'int'>
17.25 <class 'float'>
[5, 'Jean'] <class 'list'>
```

Bien que les éléments de la liste **divers** soient tous de types différents (une chaîne de caractères, un entier, un réel, une liste), on peut affecter successivement leurs contenus à la variable **e**, sans qu'il s'ensuive des erreurs (ceci est rendu possible grâce au *typage dynamique* des variables Python).

Exercices

10.5 Dans un script, écrivez une fonction qui recherche le nombre de mots contenus dans une phrase donnée.

10.6 Écrivez un script qui recherche le nombre de caractères "e", "é", "è", "ê", "ë" contenus dans une phrase donnée.

Appartenance d'un élément à une séquence : l'instruction in utilisée seule

L'instruction **in** peut être utilisée indépendamment de **for**, pour *vérifier si un élément donné fait partie ou non d'une séquence*. Vous pouvez par exemple vous servir de **in** pour vérifier si tel caractère alphabétique fait partie d'un groupe bien déterminé :

```
car = "e"
voyelles = "aeiouyAEIOUYàâéèêëùîï"
if car in voyelles:
    print(car, "est une voyelle")
```

D'une manière similaire, vous pouvez vérifier l'appartenance d'un élément à une liste :

```
n = 5
premiers = [1, 2, 3, 5, 7, 11, 13, 17]
if n in premiers:
    print(n, "fait partie de notre liste de nombres premiers")
```

> *Cette instruction très puissante effectue donc à elle seule un véritable parcours de la séquence. À titre d'exercice, écrivez les instructions qui effectueraient le même travail à l'aide d'une boucle classique utilisant l'instruction while.*

Exercices

10.7 Écrivez une fonction **estUneMaj()** qui renvoie « vrai » si l'argument transmis est une majuscule. Tâchez de tenir compte des majuscules accentuées !

10.8 Écrivez une fonction **chaineListe()** qui convertisse une phrase en une liste de mots.

10.9 Utilisez les fonctions définies dans les exercices précédents pour écrire un script qui puisse extraire d'un texte tous les mots qui commencent par une majuscule.

Les chaînes sont des séquences non modifiables

Vous ne pouvez pas modifier le contenu d'une chaîne existante. En d'autres termes, vous ne pouvez pas utiliser l'opérateur **[ ]** dans la partie gauche d'une instruction d'affectation. Essayez par exemple d'exécuter le petit script suivant (qui cherche intuitivement à remplacer une lettre dans une chaîne) :

```
salut = 'bonjour à tous'
salut[0] = 'B'
print(salut)
```

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 du genre : *TypeError: 'str' object does not support item assignment*. Cette erreur est 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:]
print salut
```

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 (à partir d'un morceau de la précédente, soit, mais qu'importe : il s'agit bien d'une *nouvelle* chaîne).

Les chaînes sont comparables

Tous les opérateurs de comparaison dont nous avons parlé à propos des instructions de contrôle de flux (c'est-à-dire les instructions **if ... elif ... else**) fonctionnent aussi avec les chaînes de caractères. Cela peut vous être utile pour trier des mots par ordre alphabétique :

```
while True:
    mot = input("Entrez un mot quelconque : (<enter> pour terminer)")
    if mot =="":
        break
    if mot < "limonade":
        place = "précède"
    elif mot > "limonade":
        place = "suit"
    else:
        place = "se confond avec"
    print("Le mot", mot, place, "le mot 'limonade' dans l'ordre alphabétique")
```

Ces comparaisons sont possibles, parce que dans toutes les normes d'encodage, les codes numériques représentant les caractères ont été attribués dans l'ordre alphabétique, tout au moins pour les caractères non accentués. Dans le système de codage *ASCII*, par exemple, A=65, B=66, C=67, etc.

Comprenez cependant que cela ne fonctionne bien que pour des mots qui sont tous entièrement en minuscules, ou entièrement en majuscules, et qui ne comportent aucun caractère accentué. Vous savez en effet que les majuscules et minuscules utilisent des ensembles de codes distincts. Quant aux caractères accentués, vous avez vu qu'ils sont encodés en dehors de l'ensemble constitué par les caractères du standard *ASCII*. Construire un algorithme de tri alphabétique qui prenne en compte à la fois la casse des caractères et tous leurs accents n'est donc pas une mince affaire !

La norme Unicode

À ce stade, il peut être utile de s'intéresser aux valeurs des identifiants numériques associés à chaque caractère. Sous Python 3, les chaînes de caractères (données de type string) sont désormais des chaînes Unicode[59], ce qui signifie que les identifiants numériques de leurs caractères sont uniques (il ne peut exister qu'un seul caractère typographique pour chaque code) et universels (les identifiants choisis couvrent la gamme complète de tous les caractères utilisés dans les différentes langues du monde entier).

À l'origine du développement des technologies informatiques, alors que les capacités de mémorisation des ordinateurs étaient encore assez limitées, on n'imaginait pas que ceux-ci seraient utilisés un jour pour traiter d'autres textes que des communications techniques, essentiellement en anglais. Il semblait donc tout-à-fait raisonnable de ne prévoir pour celles-ci qu'un jeu de caractères restreint, de manière à pouvoir représenter chacun de ces caractères avec un petit nombre de bits, et ainsi occuper aussi peu d'espace que possible dans les coûteuses unités de stockage de l'époque. Le jeu de caractères *ASCII*[60] fut donc choisi en ce temps là, avec l'estimation que 128 caractères suffiraient (à savoir le nombre de combinaisons possibles pour des groupes de 7 bits[61]). En l'étendant par la suite à 256 caractères, on put l'adapter aux exigences du traitement des textes écrits dans d'autres langues que l'anglais, mais au prix d'une dispersion des normes (ainsi par exemple, la norme *ISO-8859-1 (latin-1)* codifie tous les caractères accentués du français ou de l'allemand (entre autres), mais aucun caractère grec, hébreu ou cyrillique. Pour ces langues, il faudra respectivement utiliser les normes *ISO-8859-7, ISO-8859-8, ISO-8859-5*, bien évidemment incompatibles, et d'autres normes encore pour l'arabe, le tchèque, le hongrois...

---

[59] Dans les versions de Python antérieures à la version 3.0, les chaînes de caractères de type *string* étaient en fait des séquences d'octets (qui pouvaient représenter des caractères, mais avec un certain nombre de limitations assez gênantes), et il existait un deuxième type de chaîne, le type *unicode* pour traiter les chaînes de caractères au sens où nous l'entendons désormais.

[60] ASCII = *American Standard Code for Information Interchange*

[61] En fait, on utilisait déjà les octets à l'époque, mais l'un des bits de l'octet devait être réservé comme bit de contrôle pour les systèmes de rattrapage d'erreur. L'amélioration ultérieure de ces systèmes permit de libérer ce huitième bit pour y stocker de l'information utile : cela autorisa l'extension du jeu ASCII à 256 caractères (normes ISO-8859, etc.).

L'intérêt résiduel de ces normes anciennes réside dans leur simplicité. Elles permettent en effet aux développeurs d'applications informatiques de considérer que *chaque caractère typographique est assimilable à un octet*, et que par conséquent une chaîne de caractères n'est rien d'autre qu'une séquence d'octets. C'est ainsi que fonctionnait l'ancien type de données *string* de Python (dans les versions antérieures à la version 3.0).

Toutefois, comme nous l'avons déjà évoqué sommairement au chapitre 5, les applications informatiques modernes ne peuvent plus se satisfaire de ces normes étriquées. Il faut désormais pouvoir encoder, dans un même texte, tous les caractères de n'importe quel alphabet de n'importe quelle langue. Une organisation internationale a donc été créée : le *Consortium Unicode*, laquelle a effectivement développé une norme universelle sous le nom de `Unicode`. Cette nouvelle norme vise à donner à tout caractère de n'importe quel système d'écriture de langue un nom et un identifiant numérique, et ce de manière unifiée, quelle que soit la plate-forme informatique ou le logiciel.

Une difficulté se présente, cependant. Se voulant universelle, la norme *Unicode* doit attribuer un identifiant numérique *différent* à plusieurs dizaines de milliers de caractères. Tous ces identifiants ne pourront évidemment pas être encodés sur un seul octet. À première vue, ils serait donc tentant de décréter qu'à l'avenir, chaque caractère devra être encodé sur deux octets (cela ferait 65536 possibilités), ou trois (16777216 possibilités) ou quatre (plus de 4 milliards de possibilités). Chacun de ces choix rigides entraîne cependant son lot d'inconvénients. Le premier, commun à tous, est que l'on perd la compatibilité avec la multitude de documents informatiques préexistants (et notamment de logiciels), qui ont été encodés aux normes anciennes, sur la base du paradigme « un caractère égale un octet ». Le second est lié à l'impossibilité de satisfaire deux exigences contradictoires : si l'on se contente de deux octets, on risque de manquer de possibilités pour identifier des caractères rares ou des attributs de caractères qui seront probablement souhaités dans l'avenir ; si l'on impose trois, quatre octets ou davantage, par contre, on aboutit à un monstrueux gaspillage de ressources : la plupart des textes courants se satisfaisant d'un jeu de caractères restreint, l'immense majorité de ces octets ne contiendraient en effet que des zéros.

Afin de ne pas se retrouver piégée dans un carcan de ce genre, la norme *Unicode* ne fixe aucune règle concernant le nombre d'octets ou de bits à réserver pour l'encodage. Elle spécifie seulement la valeur numérique de l'identifiant associé à chaque caractère. En fonction des besoins, chaque système informatique est donc libre d'encoder « en interne » cet identifiant comme bon lui semble, par exemple sous la forme d'un entier ordinaire. Comme tous les langages de programmation modernes, Python s'est donc pourvu d'un type de données « chaîne de caractères » (le type `string`) qui respecte scrupuleusement la norme *Unicode*, et la représentation « interne » des codes numériques correspondants est sans importance pour le programmeur.

Nous verrons un peu plus loin dans ce chapitre qu'il est effectivement possible de placer dans une chaîne de ce type un mélange quelconque de caractères issus d'alphabets différents (qu'il s'agisse de caractères *ASCII* standards, de caractères accentués, de symboles mathématiques ou de caractères grecs, cyrilliques, arabes, etc.), et que chacun d'eux est effectivement représenté « en interne » par un code numérique unique.

Séquences d'octets : le type bytes

À ce stade de nos explications, il devient urgent de préciser encore quelque chose.

Nous avons donc vu que la norme *Unicode* ne fixe en fait rien d'autre que des valeurs numériques, pour tous les identifiants standardisés destinés à désigner de manière univoque les caractères des alphabets du monde entier (plus de 240000 en novembre 2005). Elle ne précise en aucune façon la manière dont ces valeurs numériques doivent être encodées concrètement sous forme d'octets ou de bits.

Pour le fonctionnement *interne* des applications informatiques, cela n'a pas d'importance. Les concepteurs de langages de programmation, de compilateurs ou d'interpréteurs pourront décider librement de représenter ces caractères par des entiers sur 8, 16, 24, 32, 64 bits, ou même (bien que l'on n'en voie pas l'intérêt !) par des réels en virgule flottante : c'est leur affaire et cela ne nous concerne pas. Nous ne devons donc pas nous préoccuper du format réel des caractères, à l'intérieur d'une chaîne *string* de Python.

Il en va tout autrement, par contre, pour les entrées/sorties. Les développeurs que nous sommes devons absolument pouvoir préciser sous quelle forme exacte les données sont attendues par nos programmes, que ces données soient fournies par l'intermédiaire de frappes au clavier ou par importation depuis une source quelconque. De même, nous devons pouvoir choisir le format des données que nous exportons vers n'importe quel dispositif périphérique, qu'il s'agisse d'une imprimante, d'un disque dur, d'un écran...

Pour toutes ces entrées ou sorties de chaînes de caractères, nous devrons donc toujours considérer qu'il s'agit *concrètement* de séquences d'octets, et utiliser divers mécanismes pour convertir ces séquences d'octets en chaînes de caractères, ou vice-versa.

Python met désormais à votre disposition le nouveau type de données **bytes**, spécifiquement conçu pour traiter les séquences (ou chaînes) d'octets. Les données de type *bytes* sont très similaires aux données de type *string*, mais avec la différence que ce sont strictement des séquences d'octets, et non des séquences de caractères. Les caractères peuvent bien entendu être *encodés* en octets, et les octets *décodés* en caractères, mais pas de manière univoque : du fait qu'il existe plusieurs normes d'encodage/décodage, la même chaîne *string* peut être convertie en plusieurs chaînes *bytes* différentes.

À titre d'exemple[62], nous allons effectuer à la ligne de commande un petit exercice d'écriture/lecture d'un fichier texte, en exploitant quelques possibilités de la fonction **open()** que nous n'avions pas encore rencontrées jusqu'ici. Nous veillerons à faire cet exercice avec une chaîne contenant quelques caractères accentués, ou d'autres symboles *non-ASCII* :

```
>>> chaine = "Amélie et Eugène\n"
>>> of =open("test.txt", "w")
>>> of.write(chaine)
17
>>> of.close()
```

[62]Pour cet exemple, nous supposons que la norme d'encodage par défaut sur votre système d'exploitation est Utf-8. Si vous utilisez un système d'exploitation ancien, utilisant par exemple la norme Latin-1 (ou Windows-1252), les résultats seront légèrement différents en ce qui concerne les nombres et valeurs des octets, mais vous ne devriez pas avoir de mal à interpréter ce que vous obtenez.

Avec ces quelques lignes, nous avons enregistré la chaîne de caractères **chaine** sous la forme d'une ligne de texte dans un fichier, de la manière habituelle. Effectuons à présent une relecture de ce fichier, mais en veillant à ouvrir celui-ci *en mode « binaire »*, ce qui peut se faire aisément en transmettant l'argument **"rb"** à la fonction **open()**. Dans ce mode, les octets sont transférés à l'état brut, sans conversion d'aucune sorte. La lecture avec **read()** ne nous fournit donc plus une chaîne de caractères comme au chapitre précédent, mais bien une chaîne d'octets, et la variable qui les accueille est pour cette raison automatiquement typée comme variable du type *bytes* :

```
>>> of =open("test.txt", "rb")          # "rb" => mode lecture (r) binaire (b)
>>> octets =of.read()
>>> of.close()
>>> type(octets)
<class 'bytes'>
```

En procédant ainsi, nous ne récupérons donc pas notre chaîne de caractères initiale, mais bien sa traduction concrète en octets, dans une donnée de type *bytes*. Essayons d'afficher cette donnée à l'aide de la fonction **print()** :

```
>>> print(octets)
b'Am\xc3\xa9lie et Eug\xc3\xa8ne\n'
```

Que signifie ce résultat ?

Lorqu'on lui demande d'afficher une donnée de type *bytes* à l'aide la fonction **print()**, Python nous en fournit en fait une *représentation*, entre deux apostrophes pour indiquer qu'il s'agit d'une chaîne, mais celles-ci précédées d'un **b** minuscule pour spécifier qu'il s'agit d'une chaîne d'octets (*bytes)*, avec les conventions suivantes :

- Les octets de valeur numérique comprise entre 32 et 127 sont représentés par le caractère correspondant du code ASCII.
- Certains octets de valeur numérique inférieure à 32 sont représentés de manière conventionnelle, comme par exemple le caractère de fin de ligne.
- Les autres octets sont représentés par leur valeur hexadécimale, précédée de « **\x** ».

Dans le cas de notre exemple, on voit que les caractères non accentués de la chaîne utilisée ont été encodés chacun à l'aide d'un seul octet correspondant à son code ASCII : nous les reconnaissons donc directement. Les caractères accentués, par contre (ils n'existent pas dans le code ASCII), sont encodés chacun sur deux octets : \xc3 et \xa9 pour le « é », \xc3 et \xa8 pour le « è ». Cette forme particulière d'encodage correspond à la norme *Utf-8*, que nous décrirons un peu plus en détail dans les pages suivantes.

La représentation obtenue avec **print()** nous aide à reconnaître notre chaîne initiale, certes, mais elle ne nous montre pas assez bien qu'il s'agit en fait d'octets. Essayons donc autre chose. Vous savez que l'on peut aussi examiner le contenu d'une séquence, élément par élément, à l'aide d'une boucle de *parcours*. Voyons ce que cela donne ici :

```
>>> for oct in octets:
...     print(oct, end =' ')
...
65 109 195 169 108 105 101 32 101 116 32 69 117 103 195 168 110 101 10
```

Cette fois, nous voyons très clairement qu'il s'agit bien d'octets : le parcours nous en restitue toutes les valeurs numériques, en notation décimale.

Du fait que les caractères accentués sont encodés sur deux octets en *Utf-8*, la fonction **len()** ne nous renvoie pas la même valeur pour la chaîne de caractères, et pour son équivalent encodé en *Utf-8* dans une chaîne d'octets :

```
>>> len(chaine)
17
>>> len(octets)
19
```

Les opérations d'extraction d'éléments, de slicing, etc., fonctionnent de manière analogue avec des données de type *byte* et de type *string*, quoique avec des résultats différents, bien entendu :

```
>>> print(chaine[2], chaine[12], "---", chaine[2:12])
é g --- élie et Eu
>>> print(octets[2], octets[12], "---", octets[2:12])
195 117 --- b'\xc3\xa9lie et E'
```

Attention : Vous ne pouvez pas *enregistrer* une chaîne d'octets telle quelle dans un fichier texte. Exemple :

```
>>> of =open("test.txt", "w")
>>> of.write(octets)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not bytes
```

Pour enregistrer une séquence d'octets, il faut toujours ouvrir le fichier *en mode binaire*, en utilisant l'argument **"wb"** au lieu de **"w"** dans l'instruction **open()**.

Remarquons pour terminer que nous pouvons définir une variable de type *bytes* et lui affecter une valeur littérale, en utilisant la syntaxe : **var = b'***chaîne de caractères ASCII***'** .

L'encodage Utf-8

Tout ce qui précède nous indique que la chaîne de caractères initiale de notre exemple a dû être automatiquement convertie, lors de son enregistrement dans un fichier, en une chaîne d'octets encodés suivant la norme *Utf-8*. La séquence d'octets dont nous avons traité jusqu'ici correspond donc à une forme particulière d'encodage numérique, pour la chaîne de caractères : « Amélie et Eugène »

Même si cela peut vous paraître à première vue un peu compliqué, dites-vous bien que malheureusement l'encodage idéal n'existe pas. En fonction de ce que l'on veut en faire, il peut être préférable d'encoder un même texte de plusieurs manières différentes. C'est pour cette raison qu'ont été définies, en parallèle avec la norme *Unicode*, plusieurs *normes d'encodage* : *Utf-8, Utf-16, Utf-32,* et quelques variantes. Toutes ces normes utilisent les mêmes identifiants numériques pour encoder les caractères, mais elles diffèrent sur la manière d'enregistrer concrètement ces identifiants sous forme d'octets. Ne vous affolez pas, cependant : vous ne serez vraisemblablement jamais confronté qu'à la première d'entre elles (*Utf-8*). Les autres ne concerneront que certains spécialistes de domaines « pointus ».

La norme d'encodage *Utf-8* est désormais la norme préférentielle pour la plupart des textes courants, parce que :

- d'une part, elle assure une parfaite compatibilité avec les textes encodés en « pur » *ASCII* (ce qui est le cas de nombreux codes sources de logiciels), ainsi qu'une compatibilité partielle avec les textes encodés à l'aide de ses variantes « étendues », telles que *Latin-1* ;
- d'autre part, cette nouvelle norme est celle qui est la plus économe en ressources, tout au moins pour les textes écrits dans une langue occidentale.

Suivant cette norme, les caractères du jeu *ASCII* standard sont encodés sur un seul octet. Les autres seront encodés en général sur deux octets, parfois trois ou même quatre octets pour les caractères les plus rares.

À titre de comparaison, rappelons ici que la norme la plus couramment utilisée avant *Utf-8* par les francophones, était la norme *Latin-1* (elle est encore largement répandue, en particulier dans les environnements de travail Windows). Cette norme permettait d'encoder sur un seul octet un jeu de caractères accentués restreint, correspondant aux principales langues de l'Europe occidentale (Français, Allemand, Portugais, etc.).

Les normes Utf-16 et Utf-32 encodent systématiquement tous les caractères sur deux octets pour la première, et quatre octets pour la seconde. Ces normes ne sont utilisées que pour des usages très spécifiques, comme par exemple pour le traitement interne des chaînes de caractères par un compilateur. Vous ne les rencontrerez guère.

Conversion (encodage/décodage) des chaînes

Avec les versions de Python antérieures à la version 3.0, comme dans beaucoup d'autres langages, il fallait fréquemment convertir les chaînes de caractères d'une norme d'encodage à une autre. Du fait des conventions et des mécanismes adoptés désormais, vous ne devrez plus beaucoup vous en préoccuper pour vos propres programmes traitant des données récentes.

Il vous arrivera cependant de devoir convertir des fichiers encodés suivant une norme ancienne et/ou étrangère : un programmeur digne de ce nom doit être capable d'effectuer ces conversions. Python vous fournit fort heureusement les outils nécessaires, sous la forme de *méthodes* des objets concernés.

Conversion d'une chaîne bytes en chaîne string

Considérons par exemple la séquence d'octets obtenue à la fin de notre précédent petit exercice. Si nous savons que cette séquence correspond à un texte encodé suivant la norme *Utf-8*, nous pouvons la *décoder* en chaîne de caractères à l'aide de la méthode **decode()**, avec l'argument `"Utf-8"` (ou indifféremment : `"utf-8"`, `"Utf8"` ou `"utf8"`) :

```
>>> ch_car = octets.decode("utf8")
>>> ch_car
'Amélie et Eugène\n'
>>> type(ch_car)
<class 'str'>
```

Le *parcours* de la chaîne obtenue nous fournit bien des caractères, cette fois :

```
>>> for c in ch_car:
...     print(c, end =' ')
...
A m é l i e   e t   E u g è n e
```

Conversion d'une chaîne string en chaîne bytes

Pour convertir une chaîne de caractères en une séquence d'octets, encodée suivant une norme particulière, on utilise la méthode **encode()**, qui fonctionne de manière parfaitement symétrique à la méthode **decode()** décrite précédemment. Convertissons par exemple la même chaîne de caractères, à la fois en *Utf-8* et en *Latin-1* pour comparaison :

```
>>> chaine = "Bonne fête de Noël"
>>> octets_u = chaine.encode("Utf-8")
>>> octets_l = chaine.encode("Latin-1")
>>> octets_u
b'Bonne f\xc3\xaate de No\xc3\xabl'
>>> octets_l
b'Bonne f\xeate de No\xebl'
```

Dans les séquences d'octets ainsi obtenues, on voit clairement que les caractères accentués « ê » et « ë » sont encodés à l'aide de deux octets dans le cas de la séquence *Utf-8*, et à l'aide d'un seul octet dans le cas de la séquence *Latin-1*.

Conversions automatiques lors du traitement des fichiers

Il vous faut à présent reconsidérer ce qui se passe lorsque vous souhaitez mémoriser des chaînes de caractères dans un fichier texte.

Jusqu'à présent, en effet, nous n'avons pas attiré votre attention sur le problème constitué par la norme d'encodage de ces chaînes, parce que la fonction **open()** de Python dispose fort heureusement d'un paramétrage par défaut qui convient à la plupart des situations modernes concrètes. Lorsque vous ouvrez un fichier en écriture, par exemple, en choisissant `"w"` ou `"a"` comme deuxième argument pour **open()**, Python encode automatiquement les chaînes à enregistrer en suivant la norme par défaut de votre système d'exploitation (dans nos exemples, nous avons considéré qu'il s'agissait de *Utf-8*), et la conversion inverse est effectuée lors des opérations de lecture[63]. Nous avons donc pu aborder l'étude des fichiers, au chapitre précédent, sans vous encombrer l'esprit avec des explications trop détaillées.

Dans les petits exercices des pages précédentes, nous avons encore exploité sans le dire cette facilité offerte par Python. Mais voyons à présent comment enregistrer des textes en leur appliquant un encodage différent de celui qui est prévu par défaut, ne serait-ce que pour nous assurer que l'encodage réalisé soit bien celui que nous voulons (nous devons absolument procéder ainsi si nous souhaitons que nos scripts puissent être utilisés sur différents OS).

La technique est simple. Il suffit d'indiquer cet encodage à **open()** à l'aide d'un argument supplémentaire : `encoding ="`*norme_choisie*`"`. À titre d'exemple, nous pouvons refaire les exercices des pages précédentes en forçant cette fois l'encodage en *Latin-1* :

[63] Dans les versions de Python antérieures à la version 3.0, les chaînes de caractères devaient toujours être converties en séquences d'octets avant d'être enregistrées. L'ancien type *string* étant par ailleurs équivalent au type bytes actuel, aucune conversion n'était effectuée automatiquement lors des opérations de lecture/écriture de fichiers.

```
>>> chaine ="Amélie et Eugène\n"
>>> of =open("test.txt", "w", encoding ="Latin-1")
>>> of.write(chaine)
17
>>> of.close()
>>> of =open("test.txt", "rb")
>>> octets =of.read()
>>> of.close()
>>> print(octets)
b'Am\xe9lie et Eug\xe8ne\n'
```

… etc.

(À vous d'effectuer divers contrôles et essais sur cette séquence d'octets, si vous le souhaitez).

C'est pareil lorsque vous ouvrez un fichier *en lecture*. Par défaut, Python considère que le fichier est encodé suivant la norme par défaut de votre système d'exploitation, mais ce n'est évidemment pas une certitude. Essayons par exemple de ré-ouvrir sans précaution le fichier *test.txt* que nous venons de créer dans les lignes précédentes :

```
>>> of =open("test.txt", "r")
>>> ch_lue =of.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3.1/codecs.py", line 300, in decode
    (result, consumed) = self._buffer_decode(data, self.errors, final)
UnicodeDecodeError: 'utf8' codec can't decode bytes in position 2-4:
invalid data
```

Le message d'erreur est explicite : supposant que le fichier était encodé en *Utf-8*, Python n'a pas pu le décoder[64]. Tout rentre dans l'ordre si nous précisons :

```
>>> of =open("test.txt", "r", encoding ="Latin-1")
>>> ch_lue =of.read()
>>> of.close()
>>> ch_lue
'Amélie et Eugène\n'
```

Dans les scripts élaborés, il sera probablement toujours préférable de préciser l'encodage supposé pour les fichiers que vous traitez, quitte à devoir demander cette information à l'utilisateur, ou à imaginer des tests plus ou moins élaborés pour le déterminer de façon automatique (ce qui est loin d'être évident !).

Cas des scripts Python

Les scripts Python que vous écrivez sont eux-mêmes des textes, bien entendu.
Suivant la configuration de votre logiciel éditeur, ou de votre OS, ces textes pourront donc se retrouver encodés suivant différentes normes. Afin que Python puisse les interpréter correctement, il vous est conseillé d'y inclure toujours l'un des pseudo-commentaires suivants (obligatoirement à la 1e ou à la 2e ligne) :

```
# -*- coding:Latin-1 -*-
```

[64] En informatique, on appelle **codec** (*codeur/décodeur*) tout dispositif de conversion de format. Vous rencontrerez par exemple de nombreux codecs dans le monde du multimedia (codecs audio, vidéo...). Python dispose de nombreux codecs pour convertir les chaînes de caractères suivant les différentes normes en vigueur.

Ou bien :

```
# -*- coding:Utf-8 -*-
```

… en indiquant l'encodage effectivement utilisé, bien évidemment !

Ainsi l'interpréteur Python sait décoder correctement les chaînes de caractères littérales que vous avez utilisées dans le script. Notez que vous pouvez omettre ce pseudo-commentaire si vous êtes certain que vos scripts sont encodés en *Utf-8*, car c'est cet encodage qui est désormais la norme par défaut pour les scripts Python[65].

Accéder à d'autres caractères que ceux du clavier

Voyons à présent quel parti vous pouvez tirer du fait que tous les caractères possèdent leur identifiant numérique universel *Unicode*. Pour accéder à ces identifiants, Python met à votre disposition un certain nombre de fonctions prédéfinies :

La fonction **ord(ch)** accepte n'importe quel caractère comme argument. En retour, elle fournit la valeur de l'identifiant numérique correspondant à ce caractère. Ainsi `ord("A")` renvoie la valeur 65, et `ord("Ĩ")` renvoie la valeur 296.

La fonction **chr(num)** fait exactement le contraire, en vous présentant le caractère typographique dont l'identifiant Unicode est égal à **num**. Pour que cela fonctionne, il faut cependant que deux conditions soient réalisées :

- la valeur de **num** doit correspondre effectivement à un caractère existant (la répartition des identifiants unicode n'est pas continue : certains codes ne correspondent donc à aucun caractère)
- votre ordinateur doit disposer d'une description graphique du caractère, ou en d'autres termes connaître le dessin de ce caractère, que l'on appelle *un glyphe*. Les systèmes d'exploitation récents disposent cependant de bibliothèques de glyphes très étendues, ce qui devrait vous permettre d'en afficher des milliers à l'écran.

Ainsi, par exemple, `chr(65)` renvoie le caractère **A**, et `chr(1046)` renvoie le caractère cyrillique Ж.

Vous pouvez exploiter ces fonctions prédéfinies pour vous amuser à explorer le jeu de caractères disponible sur votre ordinateur. Vous pouvez par exemple retrouver les caractères minuscules de l'alphabet grec, en sachant que les codes qui leur sont attribués vont de 945 à 969. Ainsi le petit script ci-dessous :

```
s = ""                          # chaîne vide
i = 945                         # premier code
while i <= 969:                 # dernier code
    s += chr(i)
    i = i + 1
print("Alphabet grec (minuscule) : ", s)
```

devrait afficher le résultat suivant :

```
Alphabet grec (minuscule) : αβγδεζηθικλμνξοπρςστυφχψω
```

[65] Dans les versions de Python antérieures à la version 3.0, l'encodage par défaut était ASCII.

Exercices

10.10 Écrivez un petit script qui affiche une table des codes *ASCII*. Le programme doit afficher tous les caractères en regard des codes correspondants. À partir de cette table, établissez la relation numérique simple reliant chaque caractère majuscule au caractère minuscule correspondant.

10.11 Modifiez le script précédent pour explorer les codes situés entre 128 et 256, où vous retrouverez nos caractères accentués (parmi de nombreux autres). La relation numérique trouvée dans l'exercice précédent reste-t-elle valable aussi pour les caractères accentués du Français ?

10.12 À partir de cette relation, écrivez une fonction qui convertit tous les caractères minuscules en majuscules, et vice-versa (dans une phrase fournie en argument).

10.13 Explorez la gamme des caractères *Unicode* disponibles sur votre ordinateur, à l'aide de boucles de programmes similaires à celle que nous avons nous-même utilisée pour afficher l'alphabet grec. Trouvez ainsi les codes correspondant à l'alphabet cyrillique, et écrivez un script qui affiche celui-ci en majuscules et en minuscules.

Les chaînes sont des objets

Dans les chapitres précédents, vous avez déjà rencontré de nombreux *objets*. Vous savez donc que l'on peut agir sur un objet à l'aide de *méthodes* (c'est-à-dire des fonctions associées à cet objet).
Sous Python, les chaînes de caractères sont des objets. On peut donc effectuer de nombreux traitements sur les chaînes de caractères en utilisant des méthodes appropriées. En voici quelques-unes, choisies parmi les plus utiles[66] :

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

```
>>> c2 ="Votez pour moi"
>>> a = c2.split()
>>> print(a)
['Votez', 'pour', 'moi']
>>> c4 ="Cet exemple, parmi d'autres, peut encore servir"
>>> c4.split(",")
['Cet exemple', " parmi d'autres", ' peut encore servir']
```

- **join(liste)** : rassemble une liste de chaînes en une seule (cette méthode effectue donc l'action inverse de la précédente). Attention : 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
```

[66] Il s'agit de quelques exemples seulement. La plupart de ces méthodes peuvent être utilisées avec différents paramètres que nous n'indiquons pas tous ici (par exemple, certains paramètres permettent de ne traiter qu'une partie de la chaîne). Vous pouvez obtenir la liste complète de toutes les méthodes associées à un objet à l'aide de la fonction intégrée **dir()**. Veuillez consulter l'un ou l'autre des ouvrages de référence (ou la documentation en ligne de Python) si vous souhaitez en savoir davantage.

```
>>> print("---".join(b))
Bête---à---manger---du---foin
```

- **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
```

- **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-Noël Hébèrt"
>>> print(ch.upper())
MAÎTRE JEAN-NOËL 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, aujourd'hui !"
>>> print(b3.capitalize())
"Quel beau temps, aujourd'hui !"
```

- **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()** : enlève les espaces éventuels au début et à la fin de 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
```

Dans la plupart de ces méthodes, il est possible de préciser quelle portion de la chaîne doit être traitée, en ajoutant des arguments supplémentaires. Exemples :

```
>>> print(ch9.index("e"))      # cherche à partir du début de la chaîne
4                              # et trouve le premier 'e'
>>> print(ch9.index("e",5))    # cherche seulement à partir de l'indice 5
8                              # et trouve le second 'e'
>>> print(ch9.index("e",15))   # cherche à partir du caractère n° 15
29                             # et trouve le quatrième 'e'
```

Etc.

Comprenez bien qu'il n'est pas possible de décrire toutes les méthodes disponibles, ainsi que leur paramétrage, dans le cadre restreint de ce cours. Si vous souhaitez en savoir davantage, il vous faut consulter la documentation en ligne de Python (*Library reference*), ou un bon ouvrage de référence.

Fonctions intégrées

À toutes fins utiles, rappelons également ici que l'on peut aussi appliquer aux chaînes 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.
- **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, b = 17, ["Émile", 7.65]
  >>> ch =str(a) +" est un entier et " +str(b) +" est une liste."
  >>> print(ch)
  17 est un entier et ['Émile', 7.65] est une liste.
  ```

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 particulièrement 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 à présent que votre programme construise une chaîne de caractères à partir de ces données, par exemple une phrase telle que la suivante : « La solution est devenue rouge, et sa température atteint 12,7 °C ».

Vous pouvez construire cette chaîne en assemblant des morceaux à l'aide de l'opérateur de concaténation (le symbole **+**), mais il vous faudra alors utiliser aussi la fonction intégrée **str()** pour convertir en chaîne de caractères la valeur numérique contenue dans la variable de type *float* (faites l'exercice).

Python vous offre une autre possibilité.

Vous pouvez préparer une chaîne « patron » contenant l'essentiel du texte invariable, avec des *balises* particulières aux endroits (*les champs*) 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. Un exemple vaut certainement mieux qu'un long discours :

```
>>> 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 ="L'aire d'un disque de rayon {} est égale à {}."
>>> print(ch.format(r, pi * r**2))
L'aire d'un disque de rayon 4.7 est égale à 69.397944.
```

Voilà pour le principe de base. La technique devient cependant bien plus intéressante encore 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.

Si vous insérez les indications suivantes dans la dernière balise, par exemple, vous obtiendrez respectivement :

```
avec {:8.2f}   :     L'aire d'un disque de rayon 4.7 est égale à    69.40.
avec {:6.2e}   :     L'aire d'un disque de rayon 4.7 est égale à 6.94e+01.
```

Dans le premier essai, le résultat est formaté de manière à comporter un total de 8 caractères, dont 2 chiffres après le point décimal. Dans le second, le résultat est présenté en notation scientifique (e+01 signifie : « x 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 largement 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
```

Formatage des chaînes « à l'ancienne »

Les versions de Python antérieures à la version 3.0 utilisaient une technique de formatage légèrement différente et un peu moins élaborée, qui reste encore utilisable. Il est cependant fortement conseillé d'adopter plutôt celle que nous avons décrite dans les paragraphes précédents. Nous expliquons sommairement ici l'ancienne convention, parce que vous risquez de la rencontrer encore dans les scripts de nombreux programmeurs. Elle consiste à formater la chaîne en assemblant deux éléments à l'aide de l'opérateur **%** . À gauche de cet opérateur, la chaîne « patron » contenant des balises commençant toujours par %, et à droite (entre parenthèses) le ou les objets que Python devra insérer dans la chaîne, en lieu et place des balises.

Exemple :

```
>>> coul ="verte"
>>> temp = 1.347 + 15.9
>>> print("La couleur est %s et la température vaut %s °C" % (coul, temp))
La couleur est verte et la température vaut 17.247 °C
```

La balise `%s` joue le même rôle que `{}` dans la nouvelle technique. Elle accepte n'importe quel objet (chaîne, entier, float, liste...). Vous utiliser aussi d'autres balises plus élaborées, telles que `%8.2f`, ou `%6.2e`, qui correspondent aux `{:8.2f}` et `{:6.2e}` de la nouvelle technique. C'est donc équivalent pour les cas les plus simples, mais soyez persuadés que les possibilités de la nouvelle formulation sont beaucoup plus étendues.

Exercices

10.14 Écrivez un script qui recopie en *Utf-8* un fichier texte encodé à l'origine en *Latin-1*, en veillant en outre à ce que chaque mot commence par une majuscule.
Le programme demandera les noms des fichiers à l'utilisateur. Les opérations de lecture et d'écriture des fichiers auront lieu en mode texte ordinaire.

10.15 Écrivez un script qui compte le nombre de mots contenus dans un fichier texte.

10.16 Vous disposez d'un fichier contenant des valeurs numériques. Considérez que ces valeurs sont les diamètres d'une série de sphères. Écrivez un script qui utilise les données de ce fichier pour en créer un autre, organisé en lignes de texte qui exprimeront « en clair » les autres caractéristiques de ces sphères (surface de section, surface extérieure et volume), dans des phrases telles que :

```
Diam.  46.20 cm Section  1676.39 cm² Surf.  6705.54 cm² Vol.  51632.67 cm³
Diam. 120.00 cm Section 11309.73 cm² Surf. 45238.93 cm² Vol. 904778.68 cm³
Diam.   0.03 cm Section     0.00 cm² Surf.     0.00 cm² Vol.      0.00 cm³
Diam.  13.90 cm Section   151.75 cm² Surf.   606.99 cm² Vol.   1406.19 cm³
Diam.  88.80 cm Section  6193.21 cm² Surf. 24772.84 cm² Vol. 366638.04 cm³
etc.
```

Le point sur les listes

Nous avons déjà rencontré les listes à plusieurs reprises, depuis leur présentation sommaire au chapitre 5. Les listes sont des collections ordonnées d'objets. Comme les chaînes de caractères, les listes font partie d'un type général que l'on appelle *séquences* sous Python. Comme les caractères dans une chaîne, les objets placés dans une liste sont rendus accessibles par l'intermédiaire d'un *index* (un nombre qui indique l'emplacement de l'objet dans la séquence).

Définition d'une liste – accès à ses éléments

Vous savez déjà que l'on délimite une liste à l'aide de crochets :

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

Dans le dernier exemple ci-dessus, nous avons rassemblé un entier, une chaîne, un réel et même une liste, pour vous rappeler que l'on peut combiner dans une liste des données de n'importe quel type, y compris des listes, des dictionnaires et des tuples (ceux-ci seront étudiés plus loin).

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. Nous allons approfondir cette distinction tout au long des exemples suivants.

Les listes sont modifiables

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). Exemples :

```
>>> 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.

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* :

```
>>> stuff[3][1] = "Isabelle"
>>> stuff
[5000, 'Brigitte', 3.1415999999999999, ['Albert', 'Isabelle', 1947]]
```

Comme c'est le cas pour toutes les séquences, il ne faut jamais oublier que la numérotation des éléments commence à partir de zéro. Ainsi, dans l'exemple ci-dessus on remplace l'élément n° 1 d'une liste, qui est elle-même l'élément n° 3 d'une autre liste : la liste **stuff**.

Les listes sont des objets

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 :

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

>>> nombres.append(12)                      # ajouter un élément à la fin
>>> nombres
[10, 17, 25, 38, 72, 12]

>>> nombres.reverse()                       # inverser l'ordre des éléments
>>> nombres
[12, 72, 38, 25, 17, 10]

>>> nombres.index(17)                       # retrouver l'index d'un élément
4

>>> nombres.remove(38)                      # enlever (effacer) un élément
>>> nombres
[12, 72, 25, 17, 10]
```

En plus de ces méthodes, vous disposez encore de l'instruction intégrée **del**, qui vous permet d'effacer un ou plusieurs éléments à partir de leur(s) index :

```
>>> del nombres[2]
>>> nombres
[12, 72, 17, 10]
>>> del nombres[1:3]
>>> nombres
[12, 10]
```

Notez bien la différence entre la méthode **remove()** et l'instruction **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).

Exercices

10.17 Écrivez un script qui permette d'obtenir à l'écran les 15 premiers termes des tables de multiplication par 2, 3, 5, 7, 11, 13, 17, 19 (ces nombres seront placés au départ dans une liste) sous la forme d'une table similaire à la suivante :

```
 2   4   6   8  10  12  14  16  18  20  22  24  26  28  30
 3   6   9  12  15  18  21  24  27  30  33  36  39  42  45
 5  10  15  20  25  30  35  40  45  50  55  60  65  70  75
```

etc.

10.18 Soit la liste suivante : `['Jean-Michel', 'Marc', 'Vanessa', 'Anne', 'Maximilien', 'Alexandre-Benoît', 'Louise']`
Écrivez un script qui affiche chacun de ces noms avec le nombre de caractères correspondant.

10.19 Vous disposez d'une liste de nombres entiers quelconques, certains d'entre eux étant présents en plusieurs exemplaires. Écrivez un script qui recopie cette liste dans une autre, *en omettant les doublons*. La liste finale devra être *triée*.

10.20 Écrivez un script capable d'afficher la liste de tous les jours d'une année imaginaire, laquelle commencerait un jeudi. Votre script utilisera lui-même trois listes : une liste des noms de jours de la semaine, une liste des noms des mois, et une liste des nombres de jours que comportent chacun des mois (ne pas tenir compte des années bissextiles).
Exemple de sortie :

```
jeudi 1 janvier    vendredi 2 janvier    samedi 3 janvier    dimanche 4 janvier
```

... et ainsi de suite, jusqu'au jeudi 31 décembre.

Techniques de slicing avancé pour modifier une liste

Comme nous venons de le signaler, vous pouvez ajouter ou supprimer des éléments dans une liste en utilisant une instruction (**del**) et une méthode (**append()**) intégrées. Si vous avez bien assimilé le principe du « découpage en tranches » (*slicing*), vous pouvez cependant obtenir les mêmes résultats à l'aide du seul opérateur **[ ]**. L'utilisation de cet opérateur est un peu plus délicate que celle d'instructions ou de méthodes dédiées, mais elle permet davantage de souplesse :

Insertion d'un ou plusieurs éléments n'importe où dans une liste

```
>>> 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 :

Suppression /remplacement d'éléments

```
>>> mots[2:5] = []                  # [] désigne une liste vide
>>> mots
['jambon','fromage','saucisson', 'ketchup']

>>> mots[1:3] = ['salade']
>>> mots
['jambon', 'salade', 'ketchup']

>>> mots[1:] = ['mayonnaise', 'poulet', 'tomate']
>>> mots
['jambon', 'mayonnaise', 'poulet', 'tomate']
```

- À la première ligne de cet exemple, nous remplaçons la tranche [2:5] par une liste vide, ce qui correspond à un effacement.
- À la quatrième 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 7e ligne, nous remplaçons une tranche de deux éléments par une autre qui en compte 3.

Création d'une liste de nombres à l'aide de la fonction range()

Si vous devez manipuler des séquences de nombres, vous pouvez les créer très aisément à l'aide de cette fonction intégrée. Elle renvoie une séquence d'entiers[67] que vous pouvez utiliser direc-

[67] **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 fixés pour cet ouvrage d'initiation. Veuillez donc consulter la bibliographie, page 11, ou la documentatio en ligne de Python, si vous souhaitez des

tement, ou convertir en une *liste* avec la fonction **list()**, ou convertir en *tuple* avec la fonction **tuple()** (les tuples seront décrits un peu plus loin) :

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

La fonction **range()** génère par défaut une séquence de nombres entiers de valeurs croissantes, et différant d'une unité. 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).
Notez bien que 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.

Les arguments négatifs sont autorisés :

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

Parcours d'une liste à l'aide de for, range() et len()

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). Exemple :

éclaircissements.

```
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
```

Une conséquence importante du typage dynamique

Comme nous l'avons déjà signalé plus haut (page 127), 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. Exemple :

```
>>> 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).

Opérations sur les listes

On peut appliquer aux listes les opérateurs + (concaténation) et * (multiplication) :

```
>>> 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 autre liste **A**. Vous pouvez obtenir ce résultat de différentes manières, mais l'une des plus simples consistera à effectuer : `B = [0]*len(A).`

Test d'appartenance

Vous pouvez aisément déterminer si un élément fait partie d'une liste à l'aide de l'instruction **in** (cette instruction puissante peut être utilisée avec toutes les séquences) :

```
>>> v = 'tomate'
>>> if v in legumes:
...     print('OK')
...
OK
```

Copie d'une liste

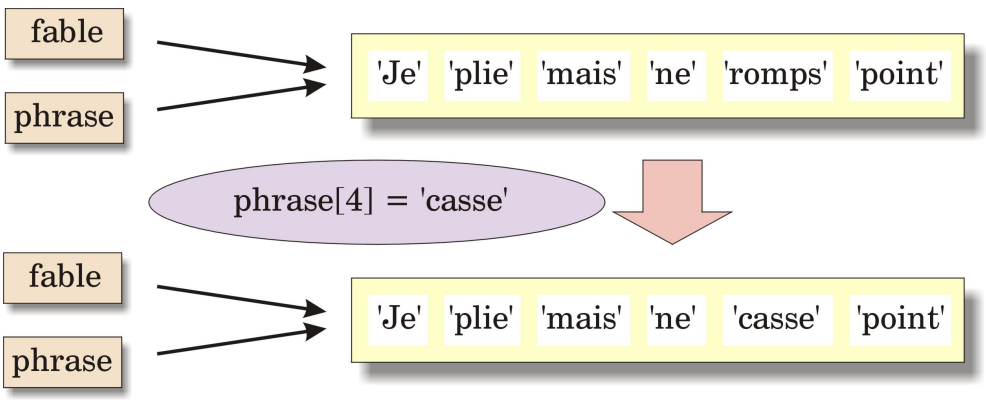
Considérons que vous disposez d'une liste **fable** que vous souhaitez recopier dans une nouvelle variable que vous appellerez **phrase**. La première idée qui vous viendra à l'esprit sera certainement d'écrire une simple affectation telle que :

```
>>> 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 :

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

Si la variable **phrase** contenait une véritable copie de la liste, cette copie serait indépendante de l'original 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.**

Nous verrons plus tard l'utilité des alias. Pour l'instant, nous voudrions disposer d'une technique pour effectuer une véritable copie d'une liste. Avec les notions vues précédemment, vous devriez pouvoir en trouver une par vous-même.

Petite remarque concernant la syntaxe

Python vous autorise à « étendre » une longue instruction sur plusieurs lignes, si vous continuez à encoder quelque chose qui est délimité par une paire de parenthèses, de crochets ou d'accolades. Vous pouvez traiter ainsi des expressions parenthésées, ou encore la définition de longues listes, de grands tuples ou de grands dictionnaires (voir plus loin). Le niveau d'indentation n'a pas d'importance : l'interpréteur détecte la fin de l'instruction là où la paire syntaxique est refermée.

Cette fonctionnalité vous permet d'améliorer la lisibilité de vos programmes. Exemple :

```
couleurs = ['noir', 'brun', 'rouge',
            'orange', 'jaune', 'vert',
            'bleu', 'violet', 'gris', 'blanc']
```

Exercices

10.21 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 insère dans la seconde liste tous les éléments de la première, de telle sorte que chaque nom de mois soit suivi du nombre de jours correspondant : `['Janvier',31,'Février',28,'Mars',31, etc.]`.

10.22 Créez une liste **A** contenant quelques éléments. Effectuez une *vraie copie* de cette liste dans une nouvelle variable **B**. Suggestion : créez d'abord une liste **B** de même taille que **A** mais ne contenant que des zéros. Remplacez ensuite tous ces zéros par les éléments tirés de **A**.

10.23 Même question, mais autre suggestion : créez d'abord une liste **B** vide. Remplissez-la ensuite à l'aide des éléments de **A** ajoutés l'un après l'autre.

10.24 Même question, autre suggestion encore : pour créer la liste **B**, découpez dans la liste **A** une tranche incluant tous les éléments (à l'aide de l'opérateur **[:]**).

10.25 *Un nombre premier est un nombre qui n'est divisible que par un et par lui-même*. Écrivez un programme qui établit la liste de tous les nombres premiers compris entre 1 et 1000, en utilisant la méthode du *crible d'Eratosthène* :

- Créez une liste de 1000 éléments, chacun initialisé à la valeur 1.
- Parcourez cette liste à partir de l'élément d'indice 2 : si l'élément analysé possède la valeur 1, mettez à zéro tous les autres éléments de la liste, dont les indices sont des multiples entiers de l'indice auquel vous êtes arrivé.

Lorsque vous aurez parcouru ainsi toute la liste, les indices des éléments qui seront restés à 1 seront les nombres premiers recherchés.
En effet : A partir de l'indice 2, vous annulez tous les éléments d'indices pairs : 4, 6, 8,

10, etc. Avec l'indice 3, vous annulez les éléments d'indices 6, 9, 12, 15, etc., et ainsi de suite. Seuls resteront à 1 les éléments dont les indices sont effectivement des nombres premiers.

Nombres aléatoires – histogrammes

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 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 quelques-unes d'entre elles. Veuillez consulter la documentation en ligne pour découvrir les autres. Vous pouvez importer toutes les fonctions du module par :

```
>>> from random import *
```

La fonction ci-dessous permet de créer une liste de nombres réels aléatoires, de valeur comprise entre zéro et un. L'argument à fournir est la taille de la liste :

```
>>> def list_aleat(n):
...     s = [0]*n
...     for i in range(n):
...         s[i] =random()
...     return s
...
>>> list_aleat(3)
[0.37584811062278767, 0.03459750519478866, 0.7145643370381 24]
>>> list_aleat(3)
[0.8151025790264931, 0.3772866844634689, 0.8207328556071652]
```

Vous pouvez constater que nous avons pris le parti de construire d'abord une liste de zéros de taille n, et ensuite de remplacer les zéros par des nombres aléatoires.

Tirage au hasard de nombres entiers

Lorsque vous développerez des projets personnels, il vous arrivera fréquemment de souhaiter disposer d'une fonction qui permette de tirer au hasard un nombre entier entre certaines limites. Par exemple, si vous voulez écrire un programme de jeu dans lequel des cartes à jouer sont tirées au hasard (à partir d'un jeu ordinaire de 52 cartes), vous aurez certainement l'utilité d'une fonction capable de tirer au hasard un nombre entier compris entre 1 et 52.

Vous pouvez pour ce faire utiliser la fonction **randrange()** du module **random**. Cette fonction peut être utilisée avec 1, 2 ou 3 arguments.

Avec un seul argument, elle renvoie un entier compris entre zéro et la valeur de l'argument diminué d'une unité. Par exemple, **randrange(6)** renvoie un nombre 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. Par exemple, **randrange(2, 8)** renvoie un nombre compris entre 2 et 7.

Si l'on ajoute 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 nombres 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
```

Exercices

10.26 Écrivez un script qui tire au hasard des cartes à jouer. Le nom de la carte tirée doit être correctement présenté, « en clair ». Le programme affichera par exemple :

```
Frappez <Enter> pour tirer une carte :
Dix de Trèfle
Frappez <Enter> pour tirer une carte :
As de Carreau
Frappez <Enter> pour tirer une carte :
Huit de Pique
Frappez <Enter> pour tirer une carte :
```

etc.

Les tuples

Nous avons étudié jusqu'ici deux types de données composites : les *chaînes*, qui sont composées de caractères, et les *listes*, qui sont composées d'éléments de n'importe quel type. Vous devez vous rappeler une autre différence importante entre chaînes et listes : il n'est pas possible de changer les caractères au sein d'une chaîne existante, alors que vous pouvez modifier les éléments d'une liste. En d'autres termes, les listes sont des séquences modifiables, alors que les chaînes de caractères sont des séquences non-modifiables. Exemple :

```
>>> liste =['jambon','fromage','miel','confiture','chocolat']
>>> liste[1:3] =['salade']
>>> print(liste)
['jambon', 'salade', 'confiture', 'chocolat']

>>> chaine ='Roméo préfère Juliette'
>>> chaine[14:] ='Brigitte'
```

```
        ***** ==> Erreur: object doesn't support slice assignment  *****
```

Nous essayons de modifier la fin de la chaîne de caractères, mais cela ne marche pas. La seule possibilité d'arriver à nos fins est de créer une nouvelle chaîne, et d'y recopier ce que nous voulons changer :

```
>>> chaine = chaine[:14] +'Brigitte'
>>> print(chaine)
Roméo préfère Brigitte
```

Python propose un type de données appelé ***tuple***[68], qui est assez semblable à une liste mais qui, comme les chaînes, n'est pas modifiable.

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

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

Bien que cela ne soit pas nécessaire, 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, mais vous savez que c'est important.

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

Opérations sur les tuples

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

```
>>> print(tup[2:4])
('c', 'd')
>>> tup[1:3] = ('x', 'y')                ==> ***** 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')
```

Remarquez qu'il faut toujours au moins une virgule pour définir un tuple (le dernier exemple ci-dessus utilise un tuple contenant un seul élément : `'André'`).

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
```

[68] Ce terme n'est pas un mot anglais ordinaire : il s'agit d'un néologisme informatique.

```
>>> 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]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object doesn't support item deletion
```

Vous comprendrez l'utilité des tuples petit à petit. Signalons simplement ici qu'ils 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 (ils occupent moins de place en mémoire, et peuvent être traités plus rapidement par l'interpréteur).

Les dictionnaires

Les types de données composites que nous avons abordés jusqu'à présent (*chaînes*, *listes* et *tuples*) étaient 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 (un nombre entier), mais à la condition expresse de connaître son emplacement.

Les ***dictionnaires*** que nous découvrons ici constituent un autre *type composite*. Ils ressemblent aux listes dans une certaine mesure (ils sont modifiables comme elles), mais ce ne sont pas des séquences. Les éléments que nous allons y enregistrer ne seront pas disposés dans un ordre immuable. En revanche, nous pourrons accéder à n'importe lequel d'entre eux à l'aide d'un index spécifique que l'on appellera une ***clé***, laquelle pourra être alphabétique, numérique, ou même d'un type composite sous certaines conditions.

Comme dans une liste, les éléments mémorisés dans un dictionnaire peuvent être de n'importe quel type. Ce peuvent être des valeurs numériques, des chaînes, des listes, des tuples, des dictionnaires, et même aussi des fonctions, des *classes* ou des *instances* (voir plus loin)[69].

Création d'un dictionnaire

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'
```

[69] Les listes et les tuples peuvent eux aussi contenir des dictionnaires, des fonctions, des classes ou des instances, Nous n'avions pas mentionné tout cela jusqu'ici, afin de ne pas alourdir l'exposé.

```
>>> 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 : un index et une valeur, séparés par un double point.

Dans un dictionnaire, les index s'appellent des *clés*, et les éléments peuvent donc s'appeler des *paires clé-valeur*. Dans notre dictionnaire d'exemple, les clés et les valeurs sont des chaînes de caractères.

Veuillez à présent constater que l'ordre dans lequel les éléments apparaissent à la dernière ligne 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
```

Remarquez aussi que contrairement à ce qui se passe avec les listes, il n'est pas nécessaire de faire appel à une méthode particulière (telle que **append()**) pour ajouter de nouveaux éléments à un dictionnaire : il suffit de créer une nouvelle paire clé-valeur.

Opérations sur les dictionnaires

Vous savez déjà comment ajouter des éléments à un dictionnaire. Pour en enlever, vous utiliserez l'instruction intégrée **del**. Créons pour l'exemple un autre 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 répertoriées 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}
```

Si le patron décide de liquider toutes les pommes et de ne plus en vendre, nous pouvons enlever cette entrée dans le dictionnaire :

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

La fonction intégrée `len()` est utilisable avec un dictionnaire : elle en renvoie le nombre d'éléments :

```
>>> print(len(invent))
3
```

Test d'appartenance

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[70] :

```
>>> if "pommes" in invent:
...     print("Nous avons des pommes")
... else:
...     print("Pas de pommes. Sorry")
...
Pas de pommes. Sorry
```

Les dictionnaires sont des objets

On peut appliquer aux dictionnaires un certain nombre de *méthodes* spécifiques :

La méthode **keys()** renvoie la séquence des *clés* utilisées 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()** et **tuple()** :

```
>>> print(dico.keys())
dict_keys(['computer', 'mouse', 'keyboard'])
>>> for k in dico.keys():
...     print("clé :", k, " --- valeur :", dico[k])
...
clé : computer  --- valeur : ordinateur
clé : mouse  --- valeur : souris
clé : keyboard  --- valeur : clavier
>>> list(dico.keys())
['computer', 'mouse', 'keyboard']
>>> tuple(dico.keys())
('computer', 'mouse', 'keyboard')
```

De manière analogue, la méthode **values()** renvoie la séquence des *valeurs* mémorisées dans le dictionnaire :

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

Quant à la méthode **items()**, elle extrait du dictionnaire une séquence équivalente de tuples. Cette méthode se révélera très utile plus loin, lorsque nous voudrons parcourir un dictionnaire à l'aide d'une boucle :

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

La méthode **copy()** permet d'effectuer une *vraie copie* d'un dictionnaire. Il faut savoir en effet que la simple affectation d'un dictionnaire existant à une nouvelle variable crée seulement *une nouvelle référence* vers le même objet, et non un nouvel objet. Nous avons déjà discuté ce phé-

[70] Dans les versions de Python antérieures à la version 3.0, il fallait faire appel à une méthode particulière (la méthode **has_key()**)pour effectuer ce test.

nomène (*aliasing*) à propos des listes (voir page 149). Par exemple, l'instruction ci-dessous ne définit pas un nouveau dictionnaire (contrairement aux apparences) :

```
>>> stock = invent
>>> stock
{'oranges': 274, 'bananes': 312, 'poires': 137}
```

Si nous modifions **invent**, alors **stock** est également modifié, et vice-versa (ces deux noms désignent en effet le même objet dictionnaire dans la mémoire de l'ordinateur) :

```
>>> del invent['bananes']
>>> stock
{'oranges': 274, 'poires': 137}
```

Pour obtenir une vraie copie (indépendante) d'un dictionnaire préexistant, il faut employer la méthode **copy()** :

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

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* (puisqu'un dictionnaire n'est pas une séquence).

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 :

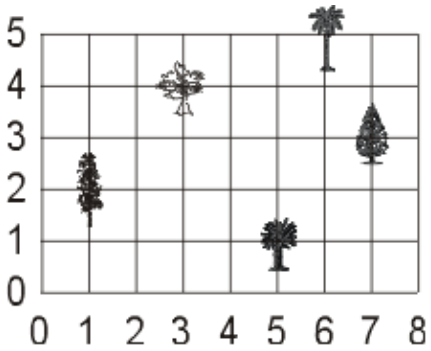
```
>>> for clef in invent:
...     print(clef, invent[clef])
...
poires 137
bananes 312
oranges 274
```

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 à la section précédente :

```
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 liste à l'aide de la boucle **for** permet d'examiner chacun de ces tuples un par un.

Les clés ne sont pas nécessairement des chaînes de caractères



Jusqu'à présent nous avons décrit des dictionnaires dont les clés étaient 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 (à utiliser avec prudence !).

Dans ce genre de construction, il faut garder à l'esprit que le dictionnaire contient des éléments seulement pour certains couples de coordonnées. Ailleurs, il n'y a rien. Par conséquent, si nous voulons interroger le dictionnaire pour savoir ce qui se trouve là où il n'y a rien, comme par exemple aux coordonnées (2,1), nous allons provoquer une erreur :

```
>>> print(arb[1,2])
Peuplier
>>> print(arb[2,1])

          ***** Erreur : KeyError: (2, 1)  *****
```

Pour résoudre ce petit problème, nous pouvons utiliser la méthode **get()** :

```
>>> arb.get((1,2), 'néant')
Peuplier
>>> arb.get((2,1), 'néant')
néant
```

Le premier argument transmis à cette méthode est la clé de recherche, le second argument est la valeur que nous voulons obtenir en retour si la clé n'existe pas dans le dictionnaire.

Les dictionnaires ne sont pas des séquences

Comme vous l'avez vu plus haut, les éléments d'un dictionnaire ne sont pas disposés dans un ordre particulier. Des opérations comme la concaténation et l'extraction (d'un groupe d'éléments contigus) ne peuvent donc tout simplement pas s'appliquer ici. Si vous essayez tout de même, Python lèvera une erreur lors de l'exécution du code :

```
>>> print(arb[1:3])

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

Vous avez vu également qu'il suffit d'affecter un nouvel indice (une nouvelle clé) pour ajouter une entrée au dictionnaire. Cela ne marcherait pas avec les listes[71] :

```
>>> invent['cerises'] = 987
>>> print(invent)
{'oranges': 274, 'cerises': 987, 'poires': 137}

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

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

Du fait qu'ils ne sont pas des séquences, les dictionnaires se révèlent donc particulièrement précieux pour gérer des ensembles de données où l'on est amené à effectuer fréquemment des ajouts ou des suppressions, dans n'importe quel ordre. Ils remplacent avantageusement les listes lorsqu'il s'agit de traiter des ensembles de données numérotées, dont les numéros ne se suivent pas.

Exemple :

```
>>> client = {}
>>> client[4317] = "Dupond"
>>> client[256] = "Durand"
>>> client[782] = "Schmidt"
```

etc.

[71] Rappel : les méthodes permettant d'ajouter des éléments à une liste sont décrites page 145.

Exercices

10.27 Écrivez un script qui crée un mini-système de base de données fonctionnant à l'aide d'un dictionnaire, dans lequel vous mémoriserez les noms d'une série de copains, leur âge et leur taille. Votre script devra comporter deux fonctions : la première pour le remplissage du dictionnaire, et la seconde pour sa consultation. Dans la fonction de remplissage, utilisez une boucle pour accepter les données entrées par l'utilisateur.
Dans le dictionnaire, le nom de l'élève servira de clé d'accès, et les valeurs seront constituées de tuples (âge, taille), dans lesquels l'âge sera exprimé en années (donnée de type entier), et la taille en mètres (donnée de type réel).
La fonction de consultation comportera elle aussi une boucle, dans laquelle l'utilisateur pourra fournir un nom quelconque pour obtenir en retour le couple « âge, taille » correspondant. Le résultat de la requête devra être une ligne de texte bien formatée, telle par exemple : « Nom : Jean Dhoute - âge : 15 ans - taille : 1.74 m ». Pour obtenir ce résultat, servez-vous du formatage des chaînes de caractères décrit à la page 140.

10.28 Écrivez une fonction qui échange les clés et les valeurs d'un dictionnaire (ce qui permettra par exemple de transformer un dictionnaire anglais/français en un dictionnaire français/anglais). On suppose que le dictionnaire ne contient pas plusieurs valeurs identiques.

Construction d'un histogramme à l'aide d'un dictionnaire

Les dictionnaires constituent un outil très élégant pour construire des *histogrammes*.

Supposons par exemple que nous voulions établir l'histogramme qui représente la fréquence d'utilisation de chacune des lettres de l'alphabet dans un texte donné. L'algorithme permettant de réaliser ce travail est extraordinairement simple si on le construit sur base d'un dictionnaire :

```
>>> texte ="les saucisses et saucissons secs sont dans le saloir"
>>> lettres ={}
>>> for c in texte:
...     lettres[c] =lettres.get(c, 0) + 1
...
>>> print(lettres)
{'t': 2, 'u': 2, 'r': 1, 's': 14, 'n': 3, 'o': 3, 'l': 3, 'i': 3, 'd': 1, 'e': 5,
'c': 3, ' ': 8, 'a': 4}
```

Nous commençons par créer un dictionnaire vide : **lettres**. Ensuite, nous allons remplir ce dictionnaire en utilisant les caractères de l'alphabet en guise de clés. Les valeurs que nous mémoriserons pour chacune de ces clés seront les fréquences 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()** doit renvoyer 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 pouvons encore souhaiter afficher l'histogramme dans l'ordre alphabétique. Pour ce faire, nous pensons immédiatement à la méthode **sort()**, mais celle-ci ne

peut s'appliquer qu'aux listes. Qu'à cela ne tienne ! Nous avons vu plus haut comment nous pouvions convertir un dictionnaire en une liste de tuples :

```
>>> lettres_triees = list(lettres.items())
>>> lettres_triees.sort()
>>> print(lettres_triees)
[(' ', 8), ('a', 4), ('c', 3), ('d', 1), ('e', 5), ('i', 3), ('l', 3), ('n', 3),
('o', 3), ('r', 1), ('s', 14), ('t', 2), ('u', 2)]
```

Exercices

10.29 Vous avez à votre disposition un fichier texte quelconque (pas trop gros). Écrivez un script qui compte les occurrences de chacune des lettres de l'alphabet dans ce texte (on simplifiera le problème en ne tenant pas compte des lettres accentuées).

10.30 Modifiez le script ci-dessus afin qu'il établisse une table des occurrences de chaque *mot* dans le texte. Conseil : dans un texte quelconque, les mots ne sont pas seulement séparés par des espaces, mais également par divers signes de ponctuation. Pour simplifier le problème, vous pouvez commencer par remplacer tous les caractères non-alphabétiques par des espaces, et convertir la chaîne résultante en une liste de mots à l'aide de la méthode **split()**.

10.31 Vous avez à votre disposition un fichier texte quelconque (pas trop gros). Écrivez un script qui analyse ce texte, et mémorise dans un dictionnaire l'emplacement exact de chacun des mots (compté en nombre de caractères à partir du début). Lorsqu'un même mot apparaît plusieurs fois, tous ses emplacements doivent être mémorisés : chaque valeur de votre dictionnaire doit donc être une liste d'emplacements.