# 5

# *Les structures de données*

# Les chaînes de caractères

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**.

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

```
>>> chaine = 'abcdefghijk'
>>> for i in aaa : print(i+'-',end=' ')

a- b- c- d- e- f- g- h- i- j- k-
```

L'instruction for permet donc d'écrire des boucles, dans lesquelles l'itération traite successivement tous les éléments d'une séquence donnée.

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

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

## 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.

```
>>> salut = 'bonjour à tous'
>>> salut[0] = 'B'
Traceback (most recent call last):
  File "<pyshell#367>", line 1, in <module>
    salut[0] = 'B'
TypeError: 'str' object does not support item assignment
>>> salut = salut[0].upper()+salut[1:]
>>> print(salut)
Bonjour à tous
```

## Les chaînes sont comparables

Tous les opérateurs de comparaison dont nous avons parlé à propos des instructions de contrôle de flux fonctionnent aussi avec les chaînes de caractères.

```
>>> 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")

Entrez un mot quelconque : (<enter> pour terminer)limonade
Le mot limonade se confond avec le mot 'limonade' dans l'ordre
alphabétique
Entrez un mot quelconque : (<enter> pour terminer)limonade <--espace
Le mot limonade  suit le mot 'limonade' dans l'ordre alphabétique
Entrez un mot quelconque : (<enter> pour terminer)limonade1
Le mot limonade1 suit le mot 'limonade' dans l'ordre alphabétique
Entrez un mot quelconque : (<enter> pour terminer)test
Le mot test suit le mot 'limonade' dans l'ordre alphabétique
Entrez un mot quelconque : (<enter> pour terminer)
```

## Les séquences d'octets

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. 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. 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, et vice-versa.

```
>>> import os
>>> os.chdir("F:\PYT - Python programmation objet\Scripts")
>>> chaine = "Amélie et Eugène\n"
>>> of =open("test.txt", "w")
>>> of.write(chaine)
17
>>> of.close()
>>> of =open("test.txt", "rb")
>>> octets =of.read()
>>> of.close()
>>> type(octets)
<class 'bytes'>
>>> print(octets)
b'Am\xe9lie et Eug\xe8ne\r\n'
>>> print(octets.decode())
Traceback (most recent call last):
  File "<pyshell#466>", line 1, in <module>
    print(octets.decode())
```

```
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xe9 in position
2: invalid continuation byte
>>> print(octets.decode(errors="replace"))
Am�lie et Eug�ne
>>> import locale
>>> print(locale.getdefaultlocale())
('fr_FR', 'cp1252')
>>> print(octets.decode('cp1252'))
Amélie et Eugène
```

## 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.

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.

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)
Alphabet grec (minuscule) :  αβγδεζηθικλμνξοπρςστυφχψω
```

**Atelier 5 - Exercice 1**

# Les listes

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.

Dans une liste on peut combiner des données de n'importe quel type, y compris des listes, des **`dictionnaires`** et des **`tuples`**. Pour accéder aux éléments d'une liste, on utilise les mêmes méthodes (index, découpage en tranches) que pour accéder aux caractères d'une chaîne.

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

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.

## 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.

```
>>> nombres[0] = 17
>>> nombres
[17, 38, 10, 25]
>>> stuff[3][1] = "Isabelle"
>>> stuff
[5000, 'Brigitte', 3.1416, ['Albert', 'Isabelle', 1947]]
>>> nombres[0] = nombres[2:]
>>> nombres
[[10, 25], 38, 10, 25]
```

## Les listes sont des objets

Les listes sont des objets à part entière, et vous pouvez donc leur appliquer un certain nombre de méthodes particulièrement efficaces.

| | |
|---|---|
| **x in l** | vrai si x est un des éléments de l |
| **x not in l** | réciproque de la ligne précédente |
| **l + t** | concaténation de l et t |
| **l * n** | concatène n copies de l les unes à la suite des autres |
| **len(l)** | nombre d'éléments de l |
| **min(l)** | plus petit élément de l, résultat difficile à prévoir lorsque les types des éléments sont différents |
| **max(l)** | plus grand élément de l |
| **sum(l)** | retourne la somme de tous les éléments |
| **del l[i:j]** | supprime les éléments d'indices entre i et j exclu. Cette instruction est équivalente à **l[i:j] = [ ]**. |
| **list(x)** | convertit x en une liste quand cela est possible |
| **l.count(x)** | Retourne le nombre d'occurrences de l'élément x. |
| **l.index(x)** | Retourne l'indice de la première occurrence de l'élément x dans la liste l. |
| **l.append(x)** | Ajoute l'élément x à la fin de la liste l. Si x est une liste, cette fonction ajoute la liste x en tant qu'élément, au final, la liste l ne contiendra qu'un élément de plus. |
| **l.extend(k)** | Ajoute tous les éléments de la liste k à la liste l. La liste l aura autant d'éléments supplémentaires qu'il y en a dans la liste k. |
| **l.insert(i,x)** | Insère l'élément x à la position i dans la liste l. |
| **l.remove(x)** | Supprime la première occurrence de l'élément x dans la liste l. S'il n'y a aucune occurrence de x, cette méthode déclenche une exception. |
| **l.pop([i])** | Retourne l'élément l[i] et le supprime de la liste. Le paramètre i est facultatif, s'il n'est pas précisé, c'est le dernier élément qui est retourné puis supprimé de la liste. |
| **l.reverse(x)** | Retourne la liste, le premier et dernier élément échange leurs places, le second et l'avant dernier, et ainsi de suite. |
| **l.sort( [key=None, reverse=False])** | Cette fonction trie la liste par ordre croissant. Le paramètre key est facultatif, il permet de préciser la fonction qui précise clé de comparaison qui doit être utilisée |

```
>>> nombres = [17, 38, 10, 25, 72]
>>> nombres.sort()
>>> nombres
[10, 17, 25, 38, 72]
>>> nombres.append(12)
>>> nombres
[10, 17, 25, 38, 72, 12]
>>> nombres.reverse()
>>> nombres
[12, 72, 38, 25, 17, 10]
>>> nombres.index(17)
4
```

```
>>> nombres.remove(38)
>>> nombres
[12, 72, 25, 17, 10]
>>> del nombres[2]
>>> nombres
[12, 72, 17, 10]
>>> del nombres[1:3]
>>> nombres
[12, 10]
```

## Techniques de slicing avancé pour modifier une liste

Il est possible d'utiliser à la place des « **del** » ou « **append** » pour ajouter ou supprimer des éléments dans une liste l'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']
>>> mots[1:2]
['fromage']
```

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.

La même démarche pour les suppressions et le 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']
```

## Création d'une liste de nombres

Si vous devez manipuler des séquences de nombres, vous pouvez les créer très aisément à l'aide de la fonction intégrée « **range** ». Elle renvoie une séquence

d'entiers que vous pouvez utiliser directement, ou convertir en une liste avec la fonction « **list** », ou convertir en tuple avec la fonction « **tuple** ».

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(5,13))
[5, 6, 7, 8, 9, 10, 11, 12]
>>> list(range(3,16,3))
[3, 6, 9, 12, 15]
>>> list(range(10, -10, -3))
[10, 7, 4, 1, -2, -5, -8]
```

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. 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, que l'on pourrait intituler **FROM**, **TO** et **STEP**

Toute boucle « **for** » peut s'appliquer sur un objet muni d'un itérateur tels que les chaînes de caractères, tuples, les listes, les dictionnaires, les ensembles.

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

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

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

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

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

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

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

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

Une conséquence importante du typage dynamique est que le type de la variable utilisée avec l'instruction « **for** » est redéfini continuellement au fur et à mesure du parcours : même si les éléments d'une liste sont de types différents, on peut parcourir cette liste à l'aide de for sans qu'il ne s'ensuive une erreur, car le type de la variable de parcours s'adapte automatiquement à celui de l'élément en cours de lecture.

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

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

## Les opérations sur des listes

On peut appliquer aux listes les opérateurs « **+** » (concaténation) et « ***** » (multiplication). L'opérateur « ***** » est particulièrement utile pour créer une liste de n éléments identiques.

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

```
>>> deuxieme = [10]*len(sept_zeros)
>>> deuxieme
[10, 10, 10, 10, 10, 10, 10]
```

## Le 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
```

## La copie d'une liste

Attention une simple affectation ne créez pas une véritable copie d'une liste que vous souhaitez recopier dans une nouvelle variable. 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.

```
>>> 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.

En fait, les noms fable et phrase désignent tous deux un seul et même objet en mémoire. Ainsi l'objet phrase est une référence de l'objet fable.

```
>>> phrase2 = phrase[0:len(phrase)]
>>> phrase[4] ='romps'
>>> phrase
['Je', 'plie', 'mais', 'ne', 'romps', 'point']
>>> phrase2
['Je', 'plie', 'mais', 'ne', 'casse', 'point']
```

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 parenthèses, ou encore la définition de longues listes, de grands tuples ou de grands. 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.

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

## Les nombres aléatoires – histogrammes

Le module « **random** », Python propose une série de fonctions permettant de générer des nombres aléatoires qui suivent différentes distributions mathématiques.

Les fonctions les plus couramment utilisées sont :

- **choice(sequence)** : renvoie un élément au hasard de la séquence fournie.
- **randint(a, b)** : renvoie un nombre entier compris entre a et b.
- randrange(a,b,c) : renvoie un nombre entier tiré au hasard d'une série limitée d'entiers entre a et b, séparés les uns des autres par un certain intervalle, défini par c.
- **random()** : renvoie un réel compris entre 0.0 et 1.0.
- **sample(sequence, k)** : renvoie k éléments uniques de la séquence.
- **seed([salt])** : initialise le générateur aléatoire.
- **shuffle(sequence[, random])** : mélange l'ordre des éléments de la séquence (dans l'objet lui-même). Si random est fourni, c'est un callable qui renvoie un réel entre 0.0 et 1.0. « **random** » est pris par défaut.
- **uniform(a, b)** : renvoie un réel compris entre a et b.

```
>>> from random import *
>>> def list_aleat(n):
        s = [0]*n
        for i in range(n):
            s[i] =random()
        return s

>>> list_aleat(3)
[0.6735559443377152, 0.09987607185190805, 0.6063188589461471]
>>> list_aleat(3)
[0.5182167354579681, 0.21973841027737828, 0.36650995653460494]

>>> for i in range(15):
```

```
        print(randrange(3, 13, 3), end =' ')

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

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.

```
>>> import random
>>> good_work = ['Excellent travail!',
        'Très bonne analyse',
        'Les résultats sont là !']
>>> bad_work = ["J'ai gratté la copie pour mettre des points",
        'Vous filez un mauvais coton',
        'Que se passe-t-il ?']
>>> ok_work = ['Bonne première partie mais soignez la présentation',
        'Petites erreurs, dommage !',
        'Des progrès']

>>> def auto_corrector(student):
        note = random.randint(1, 20)
        if note < 8:
            appreciation = random.choice(bad_work)
        elif note < 14:
            appreciation = random.choice(ok_work)
        else:
            appreciation = random.choice(good_work)
        return '%s: %s, %s' %(student,
               note, appreciation)

>>> students = ['Bernard', 'Robert', 'René', 'Gaston',
    'Églantine', 'Aimé', 'Robertine']

>>> for student in students :
        print(auto_corrector(student))

Bernard: 16, Très bonne analyse
Robert: 2, Que se passe-t-il ?
René: 11, Petites erreurs, dommage !
Gaston: 12, Petites erreurs, dommage !
Églantine: 3, J'ai gratté la copie pour mettre des points
Aimé: 18, Excellent travail!
Robertine: 2, J'ai gratté la copie pour mettre des points
```

**Atelier 5 - Exercice 2 - Exercice 3**

# Les tuples

Python propose un type de données appelé « **tuple** », 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 comprise entre parenthèses. Ce terme n'est pas un mot anglais ordinaire : il s'agit d'un néologisme informatique.

```
>>> tup = (5000, "Brigitte",
        3.1416, ["Albert", "René", 1947])
>>> print(tup)
(5000, 'Brigitte', 3.1416, ['Albert', 'René', 1947])
>>> tup2 = 5000, "Brigitte", \
        3.1416, ["Albert", "René", 1947]
>>> tup2
(5000, 'Brigitte', 3.1416, ['Albert', 'René', 1947])
>>> tup2[0]
5000
>>> tup2[0] = 1
Traceback (most recent call last):
  File "<pyshell#59>", line 1, in <module>
    tup2[0] = 1
TypeError: 'tuple' object does not support item assignment
```

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.

| | |
|---|---|
| **x in s** | vrai si x est un des éléments de s |
| **x not in s** | réciproque de la ligne précédente |
| **s + t** | concaténation de s et t |
| **s * n** | concatène n copies de s les unes à la suite des autres |
| **s[i]** | retourne le i$^{ème}$ élément de s |
| **s[i:j]** | retourne un « **tuple** » contenant une copie des éléments de s d'indices i à j exclu |
| **s[i:j:k]** | retourne un « **tuple** » contenant une copie des éléments de s dont les indices sont compris entre i et j exclu, ces indices sont espacés de k : $i$, $i + k$, $i + 2k$, $i + 3k$, ... |
| **len(s)** | nombre d'éléments de s |
| **min(s)** | plus petit élément de s, résultat difficile à prévoir lorsque les types des éléments sont différents |
| **max(s)** | plus grand élément de s |
| **sum(s)** | retourne la somme de tous les éléments |

Les « **tuples** » composés d'un seul élément ont une écriture un peu particulière puisqu'il est nécessaire d'ajouter une virgule après l'élément, sans quoi l'analyseur syntaxique de Python ne le considérera pas comme un « **tuples** » mais comme l'élément lui-même, et supprimera les parenthèses qu'il analyserait comme superflues.

```
>>> tuple()
```

```
()
>>> tuple('a')
('a',)
>>> color_and_note = ('rouge', 12, 'vert', 14, 'bleu', 9)
>>> colors = color_and_note[::2]
>>> print(colors)
('rouge', 'vert', 'bleu')
>>> notes = color_and_note[1::2]
>>> print(notes)
(12, 14, 9)
>>> color_and_note = color_and_note + ('violet',)
>>> print(color_and_note)
('rouge', 12, 'vert', 14, 'bleu', 9, 'violet')
>>> ('violet')
'violet'
>>> ('violet',)
('violet',)
```

Les « **tuples** » sont préférables aux listes partout où l'on veut être certain que les données transmises ne soient pas modifiées par erreur au sein d'un programme. En outre, les « **tuples** » 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** ».

## La création d'un dictionnaire

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 « **{ }** ».

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

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.

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.

## Les opérations sur les dictionnaires

Tout comme les listes, les objets de type dictionnaire proposent un certain nombre de méthodes.

| Nom | Description |
|---|---|
| `clear()` | Supprime tous les éléments du dictionnaire. |
| `copy()` | Renvoie une copie par références du dictionnaire. Lire la remarque sur les copies un peu plus bas. |
| `items()` | Renvoie sous la forme d'une liste de « **tuples** », es couples (clé, valeur) du dictionnaire. Les objets représentant les valeurs sont es copies complètes et non des références. |
| `keys()` | Renvoie sous la forme d'une liste l'ensemble des clés du dictionnaire. L'ordre de renvoi des éléments n'a aucune signification ni constance et peut varier à chaque modification du dictionnaire. |
| `values()` | Renvoie sous forme de liste les valeurs du dictionnaire. L'ordre de renvoi n'a ici non plus aucune signification mais sera le même que pour « **keys** » si la liste n'est pas modifiée entre-temps, ce qui permet de faire des manipulations avec les deux listes. |
| `get(cle,default)` | Renvoie la valeur identifiée par la clé. Si la clé n'existe pas, renvoie la valeur default fournie. Si aucune valeur n'est fournie, renvoie « **None** ». |
| `pop(cle,default)` | Renvoie la valeur identifiée par la clé et retire l'élément du dictionnaire. Si la clé n'existe pas, pop se contente de renvoyer la valeur default. Si le paramètre default n'est pas fourni, une erreur est levée. |
| `popitem()` | Renvoie le premier couple (clé, valeur) du dictionnaire et le retire. Si le dictionnaire est vide, une erreur est renvoyée. L'ordre de retrait des éléments correspond à l'ordre des clés retournées par « **keys** » si la liste n'est pas modifiée entre-temps. |
| `update(dic,**dic)` | Update permet de mettre à jour le dictionnaire avec les éléments du dictionnaire dic. Pour les clés existantes dans la liste, les valeurs sont mises à jour, sinon créées. Le deuxième argument est aussi utilisé pour mettre à jour les valeurs. |
| `setdefault(cle, default)` | Fonctionne comme « **get** » mais si clé n'existe pas et default est fourni, le couple (cle, default) est ajouté à la liste. |
| `fromkeys(seq, default)` | Génère un nouveau dictionnaire et y ajoute les clés fournies dans la séquence seq. La valeur associée à ces clés est default si le paramètre est fourni, « **None** » le cas échéant. |

Voici quelque exemples de ces syntaxes.

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

oranges
poires
bananes
>>> dico1 = {'a':1,'b':2}
>>> dico1.clear()
>>> dico1
```

```
{}
>>> dico = {'1': 'r', '2': [1,2]}
>>> dico2 = dico.copy()
>>> dico2
{'1': 'r', '2': [1, 2]}
>>> dico['2'].append('E')
>>> dico2['2']
[1, 2, 'E']
>>> dico = {'a': 1, 'b': 2}
>>> 'a' in dico
True
>>> 'c' not in dico
True
>>> a = {'a': 1, 'b': 1}
>>> a.items()
dict_items([('a', 1), ('b', 1)])
>>> a = {(1, 3): 3, 'Q': 4}
>>> a.keys()
dict_keys([(1, 3), 'Q'])
>>> a = {(1, 3): 3, 'Q': 4}
>>> a.values()
dict_values([3, 4])
>>> l = {1: 'a', 2: 'b', 3: 'c'}
>>> for i,j,k in l.items(),l.keys(),l.values() :
    print(i,j,k)

 (1, 'a') (2, 'b') (3, 'c')
1 2 3
a b c
>>> l.get(1)
'a'
>>> l.get(13)

>>> l.get(13, 7)
7
>>> l
{1: 'a', 2: 'b', 3: 'c'}
>>> l.pop(1)
'a'
>>> l
{2: 'b', 3: 'c'}
>>> l.pop(13, 6)
6
>>> l
{2: 'b', 3: 'c'}
>>> l = {1: 'a', 2: 'b', 3: 'c'}
>>> l.popitem()
(3, 'c')
>>> l.popitem()
(2, 'b')
>>> l.popitem()
(1, 'a')
>>> l
{}
```

```
>>> l = {1: 'a', 2: 'b', 3: 'c'}
>>> l2 = {3: 'ccc', 4: 'd'}
>>> l.update(l2)
>>> l
{1: 'a', 2: 'b', 3: 'ccc', 4: 'd'}
>>> l = {1: 'a', 2: 'b', 3: 'c'}
>>> l.setdefault(4, 'd')
'd'
>>> l
{1: 'a', 2: 'b', 3: 'c', 4: 'd'}
>>> l = {}

>>> l.fromkeys([1, 2, 3], 0)
{1: 0, 2: 0, 3: 0}
```

Les clés peuvent être de n'importe quel type de données non modifiables : des entiers, des réels, des chaînes de caractères, et même des tuples.

```
>>> arb = {}
>>> arb[(1,2)] = 'Peuplier'
>>> arb[(3,4)] = 'Platane'
>>> arb[6,5] = 'Palmier'
>>> arb[5,1] = 'Cycas'
>>> arb[7,3] = 'Sapin'
>>> print(arb)
{(1, 2): 'Peuplier', (3, 4): 'Platane', (6, 5): 'Palmier', (5, 1):
'Cycas', (7, 3): 'Sapin'}
>>> print(arb[(6,5)])
Palmier
>>> print(arb[1,2])
Peuplier
>>> print(arb[2,1])
Traceback (most recent call last):
  File "<pyshell#68>", line 1, in <module>
    print(arb[2,1])
KeyError: (2, 1)
>>> arb.get((2,1), 'néant')
'néant'
>>> print(arb[1:3])
Traceback (most recent call last):
  File "<pyshell#69>", line 1, in <module>
    print(arb[1:3])
TypeError: unhashable type: 'slice'
```

**Atelier 5 - Exercice 4**

# Atelier 5

## Exercice n° 1

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.

Écrivez un script qui recopie un fichier texte ou binaire en remplaçant tous ses espaces par le groupe de trois caractères -*-. Le fichier à copier sera fourni encodé selon la norme **`Latin-1`**, et le fichier destinataire devra être encodé en **`Utf-8`**. Le type du fichier source et les noms des 2 fichiers seront demandés en début de script.

Écrivez une fonction « **`voyelle`** », qui renvoie vrai si le caractère fourni en argument est une voyelle. Écrivez une fonction « **`compteVoyelles`** », qui renvoie le nombre de voyelles contenues dans une phrase donnée.

Explorez la gamme des caractères Unicode disponibles sur votre ordinateur, à l'aide de boucles de programmes similaires à celle que nous avons nous-mêmes 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.

Écrivez une fonction « **`estUnChiffre`** » qui renvoie vrai, si l'argument transmis est un chiffre, et faux sinon. Testez ainsi tous les caractères d'une chaîne en la parcourant à l'aide d'une boucle.

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

## Exercice n° 2

Écrivez un script qui génère la liste des carrés et des cubes des nombres de 20 à 40.

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.

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.

Écrivez un script qui recherche le mot le plus long dans une phrase donnée (l'utilisateur du programme doit pouvoir entrer une phrase de son choix).

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).

Écrivez une fonction permettant de trier une liste. Cette fonction ne pourra pas utiliser la méthode intégrée « **`sort`** » de Python : vous devez donc définir vous-même l'algorithme de tri.

Soient les listes suivantes :

```
t1=[31,28,31,30,31,30,31,31,30,31,30,31]
```

```
t2=['Janvier','Février','Mars','Avril','Mai','Juin','Juil
let','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.

## Exercice n° 3

Créez une liste A contenant quelques éléments. Effectuez une vraie copie de cette liste dans une nouvelle variable B. Options :

- 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.
- créez d'abord une liste B vide. Remplissez-la ensuite à l'aide des éléments de A ajoutés l'un après l'autre.
- créer la liste B en affectant une tranche incluant tous les éléments de la liste A (à l'aide de l'opérateur **`[ : ]`** )

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.

Écrivez une fonction « **`list_aleat`** » qui permette de créer une liste de nombres réels aléatoires, en utilisant la méthode « **`append`** » pour construire la liste petit à petit à partir d'une liste vide.

Écrivez une fonction « **`imprime_liste`** » qui permette d'afficher ligne par ligne tous les éléments contenus dans une liste de taille quelconque. Le nom de la liste sera fourni en argument. Utilisez cette fonction pour imprimer la liste de nombres aléatoires générés par la fonction « **`list_aleat`** ».

## Exercice n° 4

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 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 Dupont - âge : 15 ans - taille : 1.74 m.

Il faut pouvoir enregistrer le dictionnaire résultant dans un fichier texte, et également de reconstituer le dictionnaire à partir du fichier correspondant. Chaque ligne de votre fichier texte correspondra à un élément du dictionnaire et elle sera formatée de manière à bien séparer la clé et la valeur.