Les types primitifs que nous avons découvert, les entiers, les réels et les booléens sont suffisamment intuitifs et simples à manipuler pour ne pas passer trop de temps avec eux. Par contre, les autres types, comme les chaînes de caractères, les listes, les tuples, les ensembles ainsi que les dictionnaires méritent un approfondissement, surtout pour ces derniers puisque nous les avons très peu utilisés. Il faut préciser tout de suite que ces types là représentent des objets (programmation objet).

Quelques notions sur les objets

Les objets sont des entités (des variables) qui possèdent intérieurement une ou plusieurs autres variables, que nous nommons ***« attributs »****, mais plus surprenant, qui intègrent également des fonctionnalités associées à ces attributs, que nous nommons* ***« méthodes »*** *qui décrivent tous les comportements possibles relatifs à ces objets.*

La programmation objet est extrêmement séduisante puisque nous n'avons plus les ***« données »*** *d'un côté et les* ***« fonctions »*** *de l'autre, mais au contraire une fusion* ***(encapsulation)*** *entre ces deux éléments. Ainsi, les* ***« méthodes »*** *sont parfaitement adaptées à ce que peut faire réellement l'objet en question. Les* ***« attributs »*** *ne sont pas accessibles directement, seules les* ***« méthodes »*** *le sont, ce qui se fait très simplement au travers de l'opérateur point* ***« . »****.*

La description de l'ensemble de ces éléments ***(« attributs » + « méthodes »)*** *est déclarée dans ce que nous appelons une* ***« classe »****. En réalité, une* ***« classe »*** *est tout simplement un* ***« type »****. Nous créerons nos propres classes ultérieurement lors d'une étude consacrée à la programmation objet.*

```
>>> x = 5 # Déclaration d'une variable entière
>>> type(x)
<class 'int'> # type ou classe « int »

>>> liste = [-1, 8.9, 'bonjour'] # Déclaration d'un nouvel objet associé à la classe « list »
>>> type(liste)
<class 'list'> # type ou classe « list »
```

Objet = identité *(nom de la variable)* ***+ état*** *(valeurs des attributs)* ***+ comportement*** *(méthodes associées)****.***

Les chaînes de caractères – classe « str »

Une chaîne de caractère peut se définir en première approximation comme une suite quelconque de caractères. Dans un script Python, nous pouvons délimiter une telle suite soit par des apostrophes (simples quotes), soit par des guillemets (double quotes).

Exemples

```
>>> message = '"bien", tu peux continuer'
>>> retour = "je suis d'accord"
>>> print(message, ',', retour)
"bien", tu peux continuer , je suis d'accord
```

Remarquez l'utilisation des guillemets pour délimiter uns chaîne dans laquelle nous trouvons des apostrophes, ou l'utilisation des apostrophes pour délimiter une chaîne qui contient des guillemets.

Pour insérer aisément des caractères spéciaux dans une chaîne, nous pouvons encore délimiter la chaîne à l'aide de triples guillemets ou de triples apostrophes. Dans ce cas de figure, la chaîne n'est pas du tout interprétée et délivrée tel qu'elle est déclarée. Son apparence sera systématiquement identique à sa déclaration.

Exemples

```
>>> message = ''' "Bien", tu peux continuer
  (en retour) je suis d'accord'''
>>> print(message)
 "Bien", tu peux continuer
  (en retour) je suis d'accord
```

Accès aux caractères d'une chaîne

À la différence des données numériques, qui sont des entités singulières, les chaînes de caractères constituent un type de données composites. Nous entendons par là une entité bien définie (objet) qui est faite elle-même d'un ensemble d'entités (attributs) plus petites, en l'occurrence : les caractères.

En fonction des circonstances, nous souhaitons traiter la chaîne de caractères, tantôt comme un seul objet, tantôt comme une collection de caractères. Python permet d'accéder séparément à chacun des caractères, ou même plusieurs consécutifs, au travers de l'opérateur crochets ***« [ ] »****.*

Python considère donc une chaîne de caractères comme un objet de la catégorie des séquences (la liste aussi), lesquelles sont des collections ordonnées d'éléments. Cela signifie simplement que les caractères d'une chaîne sont toujours disposés dans un certain ordre. Par conséquent, chaque caractère de la chaîne peut être désigné par sa place dans la séquence, à l'aide d'un index.

Pour accéder à un caractère bien déterminer, nous utilisons le nom de la variable (l'objet) qui contient la chaîne et nous lui accolons, entre deux crochets, l'index numérique qui correspond à la position du caractère dans la chaîne. Attention, la première position en informatique est toujours le zéro.

chaine.py

```
# Lire chaque caractère d'une chaîne
mot = input("Saisissez un mot : ")
print("Première lettre du mot :", mot[0])
print("Deuxième lettre du mot :", mot[1])
print("Dernière lettre du mot :", mot[-1])
print("Avant dernière lettre du mot :", mot[-2])
```

Shell Python : [évaluer chaine.py]

```
Saisissez un mot : Bienvenue
Première lettre du mot : B
Deuxième lettre du mot : i
Dernière lettre du mot : e
Avant dernière lettre du mot : u
```

Python propose une particularité vraiment intéressante et de très utile dans la pratique, c'est de pouvoir désigner l'emplacement d'un caractère par rapport à la fin de la chaîne. Vous le voyez, c'est très simple et intuitif, il suffit d'utiliser les indices négatifs. Ainsi, l'indice ***« -1 »*** *correspond à la dernière lettre de la chaîne de caractères.*

Il arrive souvent également que nous ayons besoin d'extraire une partie réduite d'une chaîne de caractères. Pour cela, Python propose une technique, ici aussi très simple et intuitive, que nous appelons le « slicing » (découpage en tranches). Elle consiste à indiquer entre crochets, à l'aide de l'opérateur « : », les indices correspondant au début et à la fin de la tranche que nous souhaitons extraire.

chaine.py

```
# Portions de chaîne
bonjour = "bonjour"
print(bonjour[1:3], bonjour[:3], bonjour[3:], bonjour[3:-2]+"ie",bonjour[1::3], bonjour[:0:-1]+"b", sep='-')
```

Shell Python : [évaluer chaine.py]

```
on-bon-jour-joie-oo-ruojnob
```

Dans la tranche ***[n:m]****, le* $n^{ième}$ *caractère est inclus, mais pas le* $m^{ième}$*. Par ailleurs, les indices de découpages proposent des valeurs par défaut : le premier indice non défini est considéré comme zéro* ***« 0 »****, tandis que le second indice omis prend par défaut la taille de la chaîne complète. Il est possible également d'intégrer le* ***pas*** *(par défaut à 1) en respectant la structure suivante* ***[n:m:p]****. À ce sujet sur le dernier traitement la chaîne est lue à l'envers.*

Multiples fonctionnalités associées aux chaînes de caractères

Python intègre de nombreuses méthodes qui permettent d'effectuer différents traitements spécifiques sur les chaînes de caractères, comme les conversions majuscules/minuscules, la recherche de mot, la longueur d'une chaîne, etc.

Remarquez au passage que la notion de majuscule ou de minuscule est bien associée à la notion de chaîne de caractères, et n'a aucun sens pour tout autre type d'objets. La programmation objet est très bien adaptée à ce genre de situation, puisque des méthodes spécifiques sont encapsulées dans les objets concernés.

Les opérateurs permettent également de réaliser des traitements spécifiques et à ce titre, bien entendu, ils sont considérés comme des méthodes. Il est d'ailleurs possible de redéfinir certains opérateurs, comme l'addition ***« + »*** *par exemple, qui permet de réaliser une concaténation entre deux chaînes de caractères. Surtout, nous venons d'utiliser très largement l'opérateur crochet* ***« [:] »*** *qui lui aussi a été redéfini pour la classe* ***« str »****.*

Je rappelle qu'il est nécessaire d'utiliser l'opérateur « . » pour délimiter l'objet (la variable) de sa méthode (la fonction associée).

chaine.py

```
# Opérateurs, méthodes et fonctions associées aux chaînes de caractères
un = str()                      # crée une chaîne vide (méthode appelée constructeur)
deux = ''                       # crée une nouvelle chaîne vide
nous = "moi,"
nous += ' toi'                  # concaténation de deux chaînes (opérateur +=)
phrase = nous + ", les enfants" # concaténation de chaînes (opérateur +)
majuscule = phrase.upper()      # mise en majuscule (méthode)
minuscule = majuscule.lower()   # mise en minuscule (méthode)
un += minuscule.capitalize()    # première lettre en majuscule (méthode)
titre = phrase.title()          # première lettre de chaque mot en majuscule (méthode)
deux += 'm' + un[1:]            # double concaténation (opérateurs += et +)
répétition = "oui " * 3         # répétition d'une chaîne n fois (opérateur *)
liste = répétition.split()      # découper une chaîne en une liste de chaînes
portions = un.split(',')        # découpage de la chaîne avec choix du séparateur

print(un, deux, majuscule, minuscule, titre, répétition, liste, portions, sep='\n')
print("longueur =", len(minuscule))       # longueur d'une chaîne (fonction)
print("position =", phrase.find('toi'))   # recherche d'un mot et donne la position (méthode)
print("nombre d'espace =", un.count(' ')) # compte le nombre de la sous-chaîne spécifiée (méthode)
```

```
réel = float("-12.568")                    # transforme une chaîne en valeur numérique réelle
entier = int("123")                        # transforme une chaîne en valeur numérique entière
nombre = str(réel + entier)          # transforme des valeurs numériques en chaîne (constructeur)
chaîne = str([12, -8.6, 'bonjour'])  # transforme un type quelconque (liste) en chaîne (constructeur)
print('nombre=', nombre, ', chaîne='+chaîne, sep='')   # concaténation (opérateur +)
for portion in portions:
    print('$'+portion.strip()+'$')   # enlève les espaces éventuels au début ou à la fin (méthode)

print(phrase.replace(', toi,', ' et toi avec')) # remplace une partie de chaîne par une autre (méthode)
```

Shell Python : [évaluer chaine.py]

```
Moi, toi, les enfants
moi, toi, les enfants
MOI, TOI, LES ENFANTS
moi, toi, les enfants
Moi, Toi, Les Enfants
oui oui oui
['oui', 'oui', 'oui']
['Moi', ' toi', ' les enfants']
longueur = 21
position = 5
nombre d'espace = 3
nombre=110.432, chaîne=[12, -8.6, 'bonjour']
$Moi$
$toi$
$les enfants$
moi et toi avec les enfants
```

Vous avez ci-dessus quelques opérateurs, méthodes et fonctions bien utiles pour gérer pas mal de situations pour le traitement des chaînes de caractères. Cette liste n'est bien entendu pas exhaustive.

*Nous pouvons nous poser la question pour la fonction **« len() »**, pourquoi ne pas en avoir fait une méthode avec par exemple l'écriture suivante **« minuscule.len() »**. Les développeurs de Python ont préféré créer une fonction générique qui fonctionne pour tous les types de séquence (chaînes, bytes, liste, tuple) plutôt que de créer une méthode pour chacune des classes.*

comparaison.py

```
# Comparaison de chaînes de caractères
précédent = ''
while précédent.lower() != 'fin':
    mot = input('Saisissez votre mot ("fin" pour quitter) : ')
    if  mot == précédent: print("C'est le même mot que le précédent")
    elif mot < précédent: print("Ce mot est avant le précédent dans l'ordre alphabétique")
    else:                 print("Ce mot est après le précédent dans l'ordre alphabétique")
    précédent = mot
print('Au revoir!')
```

Shell Python : [évaluer comparaison.py]

```
Saisissez votre mot ("fin" pour quitter) : premier
Ce mot est après le précédent dans l'ordre alphabétique
Saisissez votre mot ("fin" pour quitter) : premier
C'est le même mot que le précédent
Saisissez votre mot ("fin" pour quitter) : deuxième
Ce mot est avant le précédent dans l'ordre alphabétique
Saisissez votre mot ("fin" pour quitter) : troisième
Ce mot est après le précédent dans l'ordre alphabétique
Saisissez votre mot ("fin" pour quitter) : fin
Ce mot est avant le précédent dans l'ordre alphabétique
Au revoir!
```

*Dans les opérateurs redéfinis spécialement pour les chaînes de caractères, nous retrouvons évidemment les opérateurs de comparaison classique. **Attention** toutefois, pour que les comparaisons fonctionnent bien, vous devez avoir le même type de casse de caractères (majuscule/minuscule) et les caractères accentués ne sont pas gérés comme nous le souhaiterions en France (prise en compte uniquement du code **ASCII**). Seule les opérateurs d'égalité **« == ou != »** me semblent fiables*

Chaînes de caractères paramétrées

Grâce aux opérateurs de concaténation, nous avons vu dans le chapitre précédent que nous pouvions fusionner plusieurs bouts de chaîne, avec éventuellement des valeurs de variables numériques, pour composer une seule chaîne de caractères plus complexe.

*Cela n'est pas toujours très lisible, notamment s'il y a beaucoup de morceaux à prendre en considération, et surtout pour les valeurs numériques, puisque vous êtes dans l'obligation de les transformer en chaînes de caractères au préalable, à l'aide du constructeur **« str() »**.*

Python vous offre une autre possibilité bien plus pratique. 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 (chaîne de caractères paramétrée). Vous appliquez ensuite à cette chaîne paramétrée la méthode « format() », à laquelle vous fournirez comme arguments les divers objets à convertir en caractères et à insérer en remplacement des balises.

Les balises de la chaîne paramétrée sont constituées d'accolades, contenant le numéro du paramètre (ou pas si l'ordre est respecté) avec éventuellement des indications de formatage particulier.

format.py

```
# Création de chaînes paramétrées
nom = input("Votre nom : ")
prénom = input("Votre prénom : ")
âge = int(input("Votre âge : "))
identité = "Je m'appelle {0} {1} et j'ai {2} ans."
print(identité.format(prénom.capitalize(), nom.upper(), âge))
```

Shell Python : [évaluer format.py]

```
Votre nom : rémy
Votre prénom : emmanuel
Votre âge : 45
Je m'appelle Emmanuel RÉMY et j'ai 45 ans.
```

*Dans la chaîne « identité », la balise **« {0} »** est remplacée par le prénom, la balise **« {1} »** est remplacée par le nom, et enfin la dernière balise est remplacée par l'âge. Remarquez au passage que la valeur entière est automatiquement transformée dans sa portion de chaîne équivalente. Comme les arguments sont donnés dans l'ordre précis prévu par les paramètres, il est aussi possible de supprimer la numérotation des balises*

format.py

```
# Création de chaînes paramétrées
nom = input("Votre nom : ")
prénom = input("Votre prénom : ")
âge = int(input("Votre âge : "))
identité = "Je m'appelle {} {} et j'ai {} ans."
print(identité.format(prénom.capitalize(), nom.upper(), âge))
```

*Dans l'exemple ci-dessous, nous fabriquons nos chaînes paramétrées directement dans la fonction **« print() »**. L'avantage, je le répète est d'avoir une conversion automatique des valeurs numériques vers des arguments de type chaîne. Ainsi, la lecture du code ci-dessous devient beaucoup plus lisible.*

format.py

```
# Création de chaînes paramétrées
PI = 3.141592654
rayon = float(input("Saisissez la valeur du rayon : "))
print("La circonférence d'un cercle de rayon {} est {}".format(rayon, 2*PI*rayon))
print("La surface d'un disque de rayon {} est {}".format(rayon, PI*rayon**2))
print("Le volume d'une sphère de rayon {} est {}".format(rayon, 4*PI*rayon**3/3))
```

Shell Python : [évaluer format.py]

```
Saisissez la valeur du rayon : 3
La circonférence d'un cercle de rayon 3.0 est 18.849555924
La surface d'un disque de rayon 3.0 est 28.274333886
Le volume d'une sphère de rayon 3.0 est 113.09733554400002
```

*Afin d'éviter d'avoir autant de chiffres significatifs après la virgule, il est possible de spécifier dans les balises des indications particulières de formatage (en conjonction ou non avec des numéros d'ordre). Pour cela, nous pouvons indiquer le nombre de chiffres avant (justification à droite) ou après la virgule pour les flottants (avec l'indicateur **« f »**).*

format.py

```
# Création de chaînes paramétrées
PI = 3.141592654
rayon = float(input("Saisissez la valeur du rayon : "))
print("La circonférence d'un cercle de rayon {} est {:0.2f}".format(rayon, 2*PI*rayon))
print("La surface d'un disque de rayon {} est {:0.2f}".format(rayon, PI*rayon**2))
print("Le volume d'une sphère de rayon {} est {:0.2f}".format(rayon, 4*PI*rayon**3/3))
```

Shell Python : [évaluer format.py]

```
Saisissez la valeur du rayon : 3
La circonférence d'un cercle de rayon 3.0 est 18.85
La surface d'un disque de rayon 3.0 est 28.27
Le volume d'une sphère de rayon 3.0 est 113.10
```

*Nous pouvons également représenter les valeurs entières dans différents formats, en décimal **« d »**, en binaire **« b »**, en hexadécimal **« x ou X »**. À noter que si le même argument doit être représenté plusieurs fois dans la chaîne paramétrée, l'indication du numéro d'ordre est particulièrement judicieuse.*

format.py

```
# Création de chaînes paramétrées

octet = int(input("Saisissez la valeur décimale d'un octet (0 à 255) : "))
conversion = "Le nombre décimal '{0:d}' vaut '{0:02X}' en hexadécimale et '{0:08b}' en binaire"
print(conversion.format(octet))
```

Shell Python : [évaluer format.py]

```
Saisissez la valeur décimale d'un octet (0 à 255) : 77
Le nombre décimal '77' vaut '4D' en hexadécimal et '01001101' en binaire
```

*Dans le cas des entiers, pour spécifier le nombre de chiffres significatifs, vous devez précéder le nombre de chiffres de la valeur **« 0 »**.*

Il existe une dernière possibilité pour paramétrer votre chaîne, c'est de nommer directement les variables à prendre en compte à l'intérieur de vos balises. Voici un exemple :

format.py

```
# Création de chaînes paramétrées
n = input("Votre nom : ")
p = input("Votre prénom : ")
a = int(input("Votre âge : "))
identité = "Je m'appelle {prénom} {nom} et j'ai {âge} ans."
print(identité.format(prénom = p.capitalize(), nom = n.upper(), âge = a))
```

Shell Python : [évaluer format.py]

```
Votre nom : rémy
Votre prénom : emmanuel
Votre âge : 45
Je m'appelle Emmanuel RÉMY et j'ai 45 ans.
```

Les chaînes sont des séquences non modifiables

Attention, contrairement à ce que nous pourrions penser, vous ne pouvez pas modifier le contenu d'une chaîne existence en Python. Par exemple, il n'est pas possible de modifier un caractère, grâce à l'opérateur « [ ] », pour proposer une majuscule à la place d'une minuscule. Vous êtes obligé de créer une nouvelle variable qui prendra en compte la modification.

chaine.py

```
# Chaînes non modifiables
bienvenue = 'bienvenue!'
bienvenue[0] = 'B'       # erreur au moment de l'interprétation
print(bienvenue)
```

Shell Python : [évaluer chaine.py]

```
Retraçage (dernier appel le plus récent) :
  Fichier "/home/manu/CloudStation/Projets python/chaine.py", ligne 4, dans <module>
    bienvenue[0] = 'B'
builtins.TypeError: 'str' object does not support item assignment
```

Une erreur survient dès que vous tentez de modifier la chaîne de caractères.

chaine.py

```
# Chaînes non modifiables
bienvenue = 'bienvenue!'
bienvenue = 'B' + bienvenue[1:]  # création d'une nouvelle chaîne de caractères
print(bienvenue)
```

Shell Python : [évaluer chaine.py]

```
Bienvenue!
```

La seule solution consiste donc à recréer une nouvelle variable même si cette dernière porte le même nom que précédemment (variables dynamiques).

Les listes – classe « list »

Les listes sont des collections ordonnées d'objets. Comme les chaînes de caractères, les listes font partie d'un type général que nous appelons séquences. Vous pouvez créer des listes de n'importe quelle longueur. Les listes peuvent contenir n'importe quel type d'objets. Bien plus, nous pouvons mélanger différents types d'objets dans une même liste.

Également, comme les chaînes de caractères, les objets placés dans une liste sont rendus accessibles par l'intermédiaire d'un index (le nombre qui indique l'emplacement de l'objet dans la séquence). Nous retrouvons ainsi une certaine homogénéité entre la façon de manipuler les chaînes de caractères et les listes. Par contre, contrairement aux chaînes de caractères, les listes sont modifiable. Cela nous permet de construire des listes de grande taille, morceau par morceau, d'une manière dynamique.

listes.py

```
premiers = [1, 3, 5, 7, 11, 13, 17, 23]                                    # liste d'entiers
jours = ['Lundi', 'Mardi', 'Mercredi', 'Jeudi', 'Vendredi', 'Samedi', 'Dimanche'] # liste de chaînes
mélange = [18, 6.3, 'Bonjour', premiers]                    # liste d'éléments de différents types
vide = []                                                                  # liste vide
vierge = list()              # autre liste vide (constructeur)

print(premiers, jours, mélange, vide, vierge, sep='\n')
print(premiers[-1], jours[2], mélange[1:]) # Prendre un élément ou une partie de la liste (opérateur [])

mélange[0] = mélange[1] - 3.3 # modification d'un élément de la liste (opérateur [])
print(mélange)               # le premier élément de la liste devient un réel à la place d'un entier
print(mélange[-1][:5])       # afficher une partie de liste d'un élément de la liste
```

Shell Python : [évaluer listes.py]

```
[1, 3, 5, 7, 11, 13, 17, 23]
['Lundi', 'Mardi', 'Mercredi', 'Jeudi', 'Vendredi', 'Samedi', 'Dimanche']
[18, 6.3, 'Bonjour', [1, 3, 5, 7, 11, 13, 17, 23]]
[]
[]
23 Mercredi [6.3, 'Bonjour', [1, 3, 5, 7, 11, 13, 17, 23]]
[3.0, 6.3, 'Bonjour', [1, 3, 5, 7, 11, 13, 17, 23]]
[1, 3, 5, 7, 11]
```

Multiples fonctionnalités associées aux listes

Comme les listes sont des objets, elles possèdent, à l'image des chaînes de caractères, plein de fonctionnalités adaptées, soit aux travers de ses propres méthodes, soit à l'aide de fonctions génériques préfabriquées et spécialisées pour tous les types de séquence.

listes.py

```
nombres = [-8.9, 45, 7.8, -12, 33]
motif = "{} => {} => Nombre = {}"
print(motif.format('Liste', nombres, len(nombres))) # nombre d'éléments d'une liste (fonction)
nombres.append(25)                                  # ajout d'un élément en fin de liste (méthode)
print(motif.format('Ajout', nombres, len(nombres)))
nombres.remove(7.8)                                 # suppression d'un élément spécifique (méthode)
print(motif.format('Suppression', nombres, len(nombres)))
nombres.insert(2, -5.6)                             # insertion d'un nouvel élément par indice (méthode)
print(motif.format('Insertion', nombres, len(nombres)))
nombres.reverse()                                   # inversion de l'ordre des éléments (méthode)
print(motif.format('Inversion', nombres, len(nombres)))
                                                    # récupérer et supprimer la dernière valeur (méthode)
print('Enlever, récupérer dernier élément :', nombres.pop(), nombres)  # méthode inverse de append()
nombres.sort()                                      # mettre les éléments dans l'ordre (méthode)
print(motif.format('Ordre', nombres, len(nombres)))
nombres.extend([3, -1, 2])                          # concaténation de listes (méthode)
print(motif.format('Concaténation', nombres, len(nombres)))
                                                    # test d'appartenance (opérateur in) et position (méthode)
if 33 in nombres: print('33 fait parti de', nombres, 'à la position', nombres.index(33))
print('Progression', list(range(10)))               # création d'une liste avec la fonction range()
print('Progression', list(range(5, 15)))            # création d'une liste avec la fonction range()
print('Progression', list(range(3, 21, 3)))         # création d'une liste avec la fonction range()
print('Progression', list(range(19, -40, -7)))      # création d'une liste avec la fonction range()
print('Que des zéros =>', [0]*11)                   # création d'une liste avec que des 0 (opérateur *)
duplication = [1, 2, 3] * 4                         # dupliquer n fois une partie de liste (opérateur *)
                                                    # vérifier un nombre d'occurences (méthode)
print(duplication, 'possède', duplication.count(3), 'fois la valeur 3')
autre = nombres.copy()                              # copie d'une liste vers une autre variable (méthode)
autre.remove(-12)
print(nombres, autre)
```

Shell Python : [évaluer listes.py]

```
Liste => [-8.9, 45, 7.8, -12, 33] => Nombre = 5
Ajout => [-8.9, 45, 7.8, -12, 33, 25] => Nombre = 6
```

```
Suppression => [-8.9, 45, -12, 33, 25] => Nombre = 5
Insertion => [-8.9, 45, -5.6, -12, 33, 25] => Nombre = 6
Inversion => [25, 33, -12, -5.6, 45, -8.9] => Nombre = 6
Enlever, récupérer dernier élément : -8.9 [25, 33, -12, -5.6, 45]
Ordre => [-12, -5.6, 25, 33, 45] => Nombre = 5
Concaténation => [-12, -5.6, 25, 33, 45, 3, -1, 2] => Nombre = 8
33 fait parti de [-12, -5.6, 25, 33, 45, 3, -1, 2] à la position 3
Progression [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Progression [5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
Progression [3, 6, 9, 12, 15, 18]
Progression [19, 12, 5, -2, -9, -16, -23, -30, -37]
Que des zéros => [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3] possède 4 fois la valeur 3
[-12, -5.6, 25, 33, 45, 3, -1, 2] [-5.6, 25, 33, 45, 3, -1, 2]
```

Utilisation des opérateurs associés aux listes

Comme vous pouvez le remarquer, la classe **« list »** possède pas mal de méthodes internes qui permettent de résoudre différentes situations. Dans toutes ces situations, nous pouvons procéder différemment, mais cette fois-ci en utilisant uniquement les opérateurs, notamment les opérateurs **« [::] »**.

listes.py

```
nombres = [-8.9, 45, 7.8, -12, 33]
print('Liste initiale :', nombres)
nombres += [25]                                  # ajout d'un élément en fin de liste (opérateur +=)
print('Ajout en fin de liste :', nombres)
nombres += [-8.9, 14, 7.8, 24]                   # concaténation de listes (opérateur +=)
print('Concaténation :', nombres)
del nombres[0]                                   # suppression par indice (fonction)
print("Suppression d'un élément :", nombres)
del nombres[:2]                                  # suppression par indice (fonction)
print('Suppression des deux premiers:', nombres)
nombres[1:2] = []                                # suppression (opérateurs [:])
print("Suppression d'un élément:", nombres)
nombres[2:5] = []                                # suppression (opérateurs [:])
print("Suppression d'éléments:", nombres)
nombres[1:1] = [19]                              # insertion d'un nouvel élément (opérateurs [:])
print('Insertion', nombres)
autre = nombres + [18, -2, 45]                   # concaténation de listes (opérateur +)
print('Concaténation :', autre)
dernière = nombres[-1]                           # récupérer et supprimer la dernière valeur
nombres[-1:] = []
print('Récupérer et enlever dernière valeur', nombres, '=>', dernière)
nombres[1] = 2                                   # modifier un élément de la liste (opérateur [])
print('Modifier un élément', nombres)
nombres[2:] = [15, 33, 44]                       # enlever et ajouter des éléments (opérateur [:])
print('Modifier un élément', nombres)
duplication = nombres[:]                         # duplication de liste
duplication[0] = 0
print('Duplication', nombres, '=>', duplication)
nombres[:] = []                                  # vider la liste (opérateurs [:])
print('Vider la liste', nombres)
```

Shell Python : [évaluer listes.py]

```
Liste initiale : [-8.9, 45, 7.8, -12, 33]
Ajout en fin de liste : [-8.9, 45, 7.8, -12, 33, 25]
Concaténation : [-8.9, 45, 7.8, -12, 33, 25, -8.9, 14, 7.8, 24]
Suppression d'un élément : [45, 7.8, -12, 33, 25, -8.9, 14, 7.8, 24]
Suppression des deux premiers: [-12, 33, 25, -8.9, 14, 7.8, 24]
Suppression d'un élément: [-12, 25, -8.9, 14, 7.8, 24]
Suppression d'éléments: [-12, 25, 24]
Insertion [-12, 19, 25, 24]
Concaténation : [-12, 19, 25, 24, 18, -2, 45]
Récupérer et enlever dernière valeur [-12, 19, 25] => 24
Modifier un élément [-12, 2, 25]
Modifier un élément [-12, 2, 15, 33, 44]
Duplication [-12, 2, 15, 33, 44] => [0, 2, 15, 33, 44]
Vider la liste []
```

*Grâce aux opérateurs **« [::] »** qui nous permettent de travailler en tranche de liste **« slicing »**, offre une très grande souplesse d'écriture. Par contre, c'est un peu plus délicat à manipuler. Respectez bien les critères suivants :*

Si vous utilisez l'opérateur « [ ] » à gauche du signe « = » pour effectuer une insertion ou une suppression d'éléments dans une liste, vous devez obligatoirement y indiquer une tranche « : » dans la liste cible, et non un élément isolé dans cette liste

L'élément que vous fournissez à la droite du signe « = » 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 (une tranche de liste est une liste, vous devez donc avoir une liste à gauche et à droite du signe « = » pour que cela soit compatible).

Lorsque vous utilisez les crochets « [] » sans la notion de tranche « : », il s'agit alors d'un seul élément de liste qui possède son propre type. De l'autre côté du signe « = », vous devez donc avoir une entité (variable ou valeur) correspondant à un seul élément (sans notion de liste) du même type (compatibilité dans l'affectation).

Transformation d'une chaîne en liste de mots et inversement

Il est souvent utile de découper une chaîne de caractères en plusieurs morceaux afin de pouvoir faire une analyse plus pointue et plus simple sur chaque mot plutôt que de prendre la chaîne dans sa globalité. Ce découpage nous donne finalement une liste de chaînes plus petites et se réalise grâce à la méthode **« split() »** de la classe **« str »**.

*Il est également possible de réaliser l'opération inverse, c'est-à-dire de fusionner une liste de chaînes pour en faire une seule globale, toujours grâce à une méthode de la classe **« str »**, il s'agit de la méthode **« join »**.*

listes.py

```
phrase = input('Saisissez votre phrase : ')
liste = phrase.split()                      # découpe une chaîne en une liste
chaîne = '.'.join(liste)                    # fusionne une liste en une seule chaîne
print(liste, chaîne, sep='\n')
if type(liste) is list: print(liste, 'est bien une liste')
if type(chaîne) is str: print('"{}" {}'.format(chaîne, 'est bien une chaîne de caractères'))
```

Shell Python : [évaluer listes.py]

```
Saisissez votre phrase : bienvenue à tout le monde
['bienvenue', 'à', 'tout', 'le', 'monde']
bienvenue.à.tout.le.monde
['bienvenue', 'à', 'tout', 'le', 'monde'] est bien une liste
"bienvenue.à.tout.le.monde" est bien une chaîne de caractères
```

*Quand nous appelons la méthode **« split() »**, celle-ci découpe la chaîne en fonction du paramètre par défaut (espace, tabulation et saut de ligne). Ici, la première case de la liste va donc du début de la chaîne au premier espace (non inclus), la deuxième case va du premier espace au second, et ainsi de suite jusqu'à la fin de la chaîne.*

*Vous avez bien sur la possibilité de choisir votre caractère (ou suites de caractères) séparateur. Il suffit alors de la préciser en argument de la méthode **« split() »**.*

*Pour réaliser l'opération inverse, nous utilisons la méthode **« join() »** de la chaîne réduite qui correspond au caractère (ou aux caractères) à introduire entre chaque élément de la liste. Cette méthode nous renvoie donc la chaîne définitive qui fusionne à la fois tous les éléments de la liste avec les caractères séparateurs.*

Parcours d'une liste et création de nouvelles listes au travers de l'itérative « for »

Utilisation de l'itérative **« for »** peut-être très utile, nous l'avons vu, pour parcourir une liste ou une chaîne (en réalité tout ce qui est séquence), mais également pour créer de nouvelles listes par des traitements spécifiques et conditionnels. Vous avez des exemples simples dans les algorithmes proposés ci-dessous :

listes.py

```
initiale = list(range(10))

# parcours d'une liste
print('Ensemble des éléments de la liste :', end=' ')
for n in initiale: print(n, end=' ')
print()

# création de nouvelles listes en parcourant une liste initiale
# et en proposant des traitements spécifiques ou conditionnels

carrés = [n**2 for n in initiale]
paires = [n for n in initiale if n%2==0]

print('Liste initiale', initiale)
print('Liste des carrés', carrés)
print('Liste des nombres paires', paires)
```

Shell Python : [évaluer listes.py]

```
Ensemble des éléments de la liste : 0 1 2 3 4 5 6 7 8 9
Liste initiale [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Liste des carrés [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
Liste des nombres paires [0, 2, 4, 6, 8]
```

Les tuples – classe « tuple »

Les tuples sont des séquences, à l'image des chaînes de caractères et des listes. Un tuple est une collection d'éléments de différentes natures, comme les listes. Par contre, une fois créés, les tuples, à la différence des listes, ne peuvent plus être modifiés : nous ne pouvons plus ajouter d'objets ou en retirer.

L'initialisation des tuples se fait en proposant définitivement un ensemble d'éléments, chacun séparé par une virgule. En réalité, les tuples sont très fréquemment utilisés dans Python, sans que nous nous en rendions compte. La liste des paramètres d'une fonction est considéré comme un tuple, l'affectation multiple l'est également. Partout où vous avez une écriture qui prend un ensemble d'éléments séparés par des virgules peut être considérée comme un tuple.

*Pour que cela soit peut-être plus clair, il est souvent conseillé d'utiliser des parenthèses pour délimiter les tuples. Cette syntaxe permet d'ailleurs de voir la différence entre les listes **« [ ] »** et les tuples **« () »**.*

*Toutes les opérations, les fonctions, que nous avons découvertes sur les listes s'appliquent bien entendu sur les tuples, sauf celles qui tentent de modifier le tuple en question. Pour créer un tuple, vous devez systématiquement avoir l'opérateur virgule **« , »** (c'est ce qui le définit), même si le tuple ne comporte qu'un élément.*

tuple.py

```
t1 = 1, 2.5, 9                                  # tuple de 3 valeurs
t2 = (3, 5.2, 6, -7.9)                          # tuple de 4 valeurs
t3 = 8,                                         # tuple de 1 élément
t4 = (9.3,)                                     # tuple de 1 élément
t0 = ()                                         # tuple vide
t5 = t1 + t2 + t3 + t4                          # concaténation
print('t5 =', t5, ': t0 =', t0)
a, b = 5, 9.6                                   # affectations multiples (tuples en interne)
(c, d) = (3, 4)                                 # écriture équivalente
print('a = {}, b = {}, c = {}, d = {}'.format(a, b, c, d))
t6, t7 = (1, 2), (11, 4, 5)                       # plusieurs tuples
t8 = t6*3 + t7[-2:] + t5[1:4] + (t7[0],) + t7[0:1]    # opérateurs (* + [:] ,)
print('t8 =', t8)
suite = tuple(range(10))                        # utilisation de la fonction range()
                                                # nombre d'éléments dans le tuple (fonction len())
print('suite = {} : nombre = {}'.format(suite, len(suite)))
```

Shell Python : [évaluer tuple.py]

```
t5 = (1, 2.5, 9, 3, 5.2, 6, -7.9, 8, 9.3) : t0 = ()
a = 5, b = 9.6, c = 3, d = 4
t8 = (1, 2, 1, 2, 1, 2, 4, 5, 2.5, 9, 3, 11, 11)
suite = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) : nombre = 10
```

*Pour récupérer un seul élément du tuple **« t7 »** pour qu'il soit considéré comme une partie de tuple, deux syntaxes vous sont proposées dans l'exemple précédent pour la création du tuple **« t8 »**. Soit nous devons utiliser les opérateurs **« , »** et **« () »**, soit nous utilisons une portion de tuple, grâce à l'opérateur **« [:] »**.*

À quoi servent les tuples

Nous pouvons légitimement nous poser la question de l'intérêt d'utiliser les tuples puisqu'ils sont très similaires aux listes avec des restrictions supplémentaires. En réalité, dans Python, il est très utilisé, notamment pour le retour de plusieurs valeurs dans les fonctions ou les méthodes, ce qui est d'ailleurs la grande particularité de ce langage par rapport aux autres (les fonctions ou les méthodes dans les autres langages ne renvoient toujours qu'une seule valeur au maximum).

Pensez aussi que les opérateurs, comme beaucoup d'autres langages, sont considérés comme des fonctions ou des méthodes. À ce titre, nous avons vu que l'affectation multiple propose effectivement en interne la création de tuples pour résoudre le traitement d'affectation à chacune des variables concernées.

Je vous propose de voir deux exemples qui vont nous permettre d'utiliser des syntaxes très concises pour résoudre simplement des traitements spécifiques grâce à l'utilisation de ces tuples qui autrement nous prendrait beaucoup plus de temps et de réflexion.

tuple.py

```
premiers = [1, 3, 5, 7, 11]

  # fonction enumerate() qui prend une liste en paramètre
  # et qui renvoie un tuple avec l'indice et la valeur de chaque élément
for valeur in enumerate(premiers):
    print(valeur)

for indice, premier in enumerate(premiers):
    print("Le {} nombre premier est {}".format(indice+1, premier))
```

Shell Python : [évaluer tuple.py]

```
(0, 1)
(1, 3)
(2, 5)
(3, 7)
(4, 11)
Le 1 nombre premier est 1
Le 2 nombre premier est 3
Le 3 nombre premier est 5
Le 4 nombre premier est 7
Le 5 nombre premier est 11
```

*Nous utilisons dans cet exemple la fonction **« enumerate() »** qui a la particularité de prendre une liste en argument et qui retourne un tuple avec la valeur de la position dans la liste suivie de la valeur de l'élément. Nous voyons bien d'ailleurs le résultat obtenu dans l'utilisation du premier **« for »**.*

*Puisque cette fonction retourne un tuple, il est tout à fait possible de récupérer chaque élément dans une variable séparée grâce à l'opérateur **« , »** (qui est d'ailleurs l'opérateur fondamental du tuple). C'est ce que nous avons fait dans le deuxième **« for »** avec la création des variables **« indice »** et **« premier »**.*

Nous pouvons également passer d'une liste vers un tuple très simplement en proposant une liste de variables, séparées par des virgules « , », dont le nombre correspond exactement au nombre d'éléments de la liste, et faire une simple affectation (multiple)

tuple.py

```
liste = [11, 22, 33]                    # création d'une liste
premier, deuxième, troisième = liste    # transformation d'une liste vers un tuple
print(premier, deuxième, troisième)
```

Shell Python : [évaluer tuple.py]

```
11 22 33
```

*Dans l'exemple qui suit, nous utilisons cette particularité. La méthode **« split() »** renvoie une liste automatiquement transformée en tuple grâce à la déclaration des variables **« entière »** et **« décimale »**.*

tuple.py

```
réel = input('Saisissez un nombre réel : ')
entière, décimale = réel.split('.')
motif = 'Le nombre {} possède la partie entière {} et la partie décimale {}'
print(motif.format(réel, entière, décimale))
```

Shell Python : [évaluer tuple.py]

```
Saisissez un nombre réel : 3.5
Le nombre 3.5 possède la partie entière 3 et la partie décimale 5
```

Nous aurons l'occasion de proposer plein d'autres exemples significatifs sur les tuples lors de l'étude des fonctions personnalisées

Les dictionnaires – classe « dict »

Les types de données composites que nous avons abordés jusqu'à présent (chaînes, listes et tuples) sont toutes des séquences, c'est-à-dire des suites ordonnées d'éléments. Dans une séquence, nous pouvons accéder à n'importe quel élément au moyen de son index (un numéro de case dont la première est 0), par contre cela oblige à connaître sa position dans la suite ordonnée.

*Les dictionnaires que nous abordons dans ce chapitre constituent un autre type composite. À ce titre, comme les listes, les dictionnaires sont des objets qui contiennent d'autres objets. Cependant, au lieu d'héberger des informations dans un ordre précis, elles sont associées à un élément de référence qui s'appelle une **« clé »**. C'est cette **« clé »**, qui peut-être de n'importe quel type, qui nous permettra de retrouver très facilement cette information ultérieurement.*

*Un dictionnaire n'est donc pas une séquence puisque les informations sont enregistrées plus en rapport avec ce que représente la **« clé »** plutôt qu'une mise en mémoire ordonnée dans le temps. Justement, l'intérêt du dictionnaire, c'est de pouvoir retrouver l'information instantanément, grâce à cette **« clé »** sans connaître précisément une position particulière dans la mémoire.*

*L'exemple typique et le répertoire téléphonique, où pour retrouver un numéro de téléphone, il suffit de connaître le nom de la personne concernée (le nom ici correspond à la **« clé »**). L'autre exemple, également significatif, c'est le dictionnaire, où à un mot particulier (la **« clé »**) correspond une définition associée (ce n'est peut-être pas pour rien qu'on ait choisi ce nom là pour définir ce type de collection).*

Comme dans les listes, les éléments mémorisés dans un dictionnaire, ainsi que les clés d'ailleurs, peuvent vraiment être de n'importe quel type. Ce peuvent être des valeurs numériques, des chaînes, des listes, des tuples, des dictionnaires, et même des fonctions, des classes ou des objets personnalisés.

dictionnaires.py

```
répertoire = {}                                          # création d'un dictionnaire vide
login = dict()                                           # création d'un autre dictionnaire vide
répertoire['marcel'] = '04-85-99-66-33'                  # ajout du premier élément dans le dictionnaire
répertoire['hugo'] = '06-15-48-26-53'                    # ajout d'un autre élément dans le dictionnaire
login['utilisateur'] = 'marcel'
login['mot de passe'] = '#&ùhuio45&'
print(répertoire, login, sep='\n')                       # afficher les deux collections complètes
print(login['utilisateur'], ':', répertoire['marcel']) # afficher un seul élément des dictionnaires
```

Shell Python : [évaluer dictionnaires.py]

```
{'hugo': '06-15-48-26-53', 'marcel': '04-85-99-66-33'}
{'mot de passe': '#&ùhuio45&', 'utilisateur': 'marcel'}
marcel : 04-85-99-66-33
```

*Les crochets **« [ ] »** délimitent les listes, les parenthèses **« ( ) »** délimitent les tuples et les accolades **« { } »** délimitent les dictionnaires. Nous pouvons observer que le résultat obtenu pour un dictionnaire apparaît sous la forme d'une série d'éléments séparés par des virgules **« , »**, le tout étant enfermé entre deux accolades **« {} »**. Chacun de ces éléments est lui-même constitué d'une paire d'objets : une **« clé »** et une **« valeur »** associée, séparées par un double point **« : »**. Dans cet exemple, les **« clés »** et les **« valeurs »** sont toutes les deux des chaînes de caractères.*

*Veuillez également constater que l'ordre dans lequel les éléments nous apparaissent ne correspondent pas à celui dans lequel nous les avons enregistrés dans le temps. Si nous regardons bien, l'affichage se fait à priori dans l'ordre alphabétique des **« clés »**, puisque ces **« clés »** sont des chaînes de caractères.*

*Pour introduire de nouvelles valeurs dans un dictionnaire, nous indiquons entre crochets **« [ ] »** la **« clé »** à laquelle nous souhaitons accéder. Si la clé n'existe pas, elle est ajoutée au dictionnaire avec la **« valeur »** spécifiée après le signe **« = »**. Sinon, l'ancienne **« valeur »** à l'emplacement indiqué est remplacée par la nouvelle.*

*Pour accéder à une **« valeur »** particulière, il suffit de préciser la **« clé »** correspondante au dictionnaire souhaité grâce, encore une fois, à l'utilisation des crochets **« [ ] »**.*

Résumons un peu tout cela : nous avons des dictionnaires, qui peuvent contenir d'autres objets. Nous plaçons ces objets et nous y accédons grâce à des clés. Un dictionnaire ne peut pas contenir deux clés identiques (comme nous l'avons évoqué, la seconde écrase la première). En revanche, rien n'empêche d'avoir deux valeurs identiques dans le dictionnaire.

Multiples fonctionnalités associées aux dictionnaires

À l'image de ce que nous avons fait lors des chapitres précédents, il existe un certain nombre de fonctionnalités intéressantes pour travailler avec les dictionnaires : choisir le type de **« clé »** et de **« valeur »**, supprimer des éléments déjà intégrés, parcourir un dictionnaire, avec l'ensemble des éléments, uniquement avec les clés, uniquement avec les valeurs, etc.

dictionnaires.py

```
échiquier = {}                                      # création d'un dictionnaire vide
corbeille = {'bananes':3, 'pommes':5, 'poires':7}  # création d'un dictionnaire déjà rempli

échiquier['A', 1] = 'Tour', 'Blanche'   # ajout d'un élément : la clé et la valeur sont des tuples
échiquier['F', 7] = 'Reine', 'Noire'    # le tuple de la clé est une chaîne et un nombre entier
échiquier['D', 2] = 'Pion', 'Blanc'     # le tuple de la valeur sont deux chaînes
échiquier['E', 2] = 'Pion', 'Blanc'     # les valeurs peuvent être identiques pour des clés différentes
échiquier['D', 4] = 'rien'              # ici la valeur n'est pas un tuple mais une simple chaîne
échiquier['Noires'] = 16                # les clés aussi peuvent être de nature différente
échiquier['Blanches'] = 14

if ('D', 2) in échiquier: del échiquier['D', 2]   # appartenance et suppression d'un élément par sa clé
print(échiquier.pop(('A', 1)))                    # récupération et suppression d'un élément par sa clé
                                                  # ATTENTION aux doubles parenthèses
print(échiquier)                        # affichage du dictionnaire au complet
print(corbeille)                        # affichage du dictionnaire au complet

print(corbeille.keys())                 # ensemble des clés du dictionnaire
print(list(corbeille.keys()))           # ensemble des clés du dictionnaire sous forme de liste
print(tuple(corbeille.keys()))          # ensemble des clés du dictionnaire sous forme de tuple

print(corbeille.values())               # ensemble des valeurs du dictionnaire
print(list(corbeille.values()))         # ensemble des valeurs du dictionnaire sous forme de liste
print(tuple(corbeille.values()))        # ensemble des valeurs du dictionnaire sous forme de tuple

print(corbeille.items())                # ensemble des éléments du dictionnaire
print(list(corbeille.items()))          # ensemble des éléments du dictionnaire sous forme de liste
print(tuple(corbeille.items()))         # ensemble des éléments du dictionnaire sous forme de tuple
```

```
for clé in corbeille: print(clé, end=' ') # parcours d'un dictionnaire (les clés par défaut)
print()
for clé in corbeille.keys(): print(clé, end=' ') # résultat identique
print()
for valeur in corbeille.values(): print(valeur, end=', ') # parcours d'un dictionnaire pour leurs valeurs
print()
for clé, valeur in corbeille.items():                    # parcours d'un dictionnaire (clés + valeurs)
    print('Le nombre de {} est {}'.format(clé, valeur))   # les tuples sont omniprésents

fruits = corbeille.copy()                                # copie réelle d'un dictionnaire
corbeille.clear()                                        # vider un dictionnaire
print('fruits = {}, la corbeille = {}'.format(fruits, corbeille))
print('Nombre de types de fruit :', len(fruits))         # Nombre d'éléments d'un dictionnaire
```

Shell Python : [évaluer dictionnaires.py]

```
('Tour', 'Blanche')
{'Noires': 16, ('E', 2): ('Pion', 'Blanc'), ('F', 7): ('Reine', 'Noire'), 'Blanches': 14, ('D', 4):
'rien'}
{'pommes': 5, 'bananes': 3, 'poires': 7}
dict_keys(['pommes', 'bananes', 'poires'])
['pommes', 'bananes', 'poires']
('pommes', 'bananes', 'poires')
dict_values([5, 3, 7])
[5, 3, 7]
(5, 3, 7)
dict_items([('pommes', 5), ('bananes', 3), ('poires', 7)])
[('pommes', 5), ('bananes', 3), ('poires', 7)]
(('pommes', 5), ('bananes', 3), ('poires', 7))
pommes bananes poires
pommes bananes poires
5, 3, 7,
Le nombre de pommes est 5
Le nombre de bananes est 3
Le nombre de poires est 7
fruits = {'pommes': 5, 'bananes': 3, 'poires': 7}, la corbeille = {}
Nombre de types de fruit : 3
```

*Dans cet exemple de script, nous voyons qu'il possible de créer un dictionnaire avec dès le départ un certain nombre d'éléments. Attention toutefois à la syntaxe ! Penser aux **« : »** pour séparer la **« clé »** de sa **« valeur »** et à l'opérateur **« , »** pour séparer chaque élément constituant le dictionnaire.*

*Pour supprimer un élément du dictionnaire, il suffit de faire appel à la fonction générique **« del() »**. Par contre, encore une fois, vous avez besoin de la **« clé »** pour identifier l'élément à supprimer. Vous pouvez aussi faire appel à la méthode intégrée **« pop() »** si vous souhaitez récupérer l'élément avant de le supprimer.*

*Lorsque vous parcourrez un dictionnaire à l'aide de l'itérative **«for … in ... »**, par défaut c'est la **« clé »** de chaque élément qui est délivrée. Si vous désirez récupérer successivement les **« valeurs »**, vous devez passer alors par la méthode **« values() »**. Si vous souhaitez tout récupérer à chaque itération, c'est cette fois-ci la méthode **« items() »** qui est préférable. Enfin, la méthode **« keys() »** nous renvoie l'ensemble des **« clés »** présentes actuellement dans la collection.*

*L'opérateur **« in »**, sans le **« for »**, en association avec l'opérateur **« if »** nous permet également de tester si il existe bien une clé (unique) dans le dictionnaire.*

Lorsque nous étudierons les fonctions, nous reviendrons sur les dictionnaires, pour le passage des arguments, pour les appels automatiques de fonction, etc.

Les ensembles – classe « set »

Il est possible d'implémenter la théorie des ensembles en Python. Un ensemble **« set »** est également un objet conteneur, composé d'autres objets non ordonnés, mais avec la particularité que deux objets identiques ne peuvent pas coexister dans la collection, chaque élément doit être unique.

Si vous vous souvenez bien de vos cours de mathématiques, l'intérêt de la manipulation des ensembles est surtout accès sur des notions d'appartenance, d'inclusion, d'union, d'intersection, de différence entre plusieurs ensembles, etc. Il existe bien entendu un certain nombre d'opérateurs ou de méthodes adaptés à ce genre de critères que nous allons découvrir au travers d'un exemple relativement complet.

ensembles.py

```
un, deux, trois, quatre = {1, 2, 3, 4}, {-1, 0, 1, 2, 5}, {1, 2}, {7, 8, 9}

                                          # déclaration de plusieurs ensembles pré-remplis
un.add(5)                                 # ajout d'un élément dans l'ensemble
deux.add(2)                               # ajout non effectué, déjà présent
deux.remove(-1)                           # suppression d'un élément de l'ensemble
```

```
print('un =', un)
print('deux =', deux)
print('trois =', trois)
print('quatre =', quatre)

print('Union (un, deux) =', un.union(deux))                # union de deux ensembles (méthode)
print('Union (un, quatre) =', un | quatre)                 # union de deux ensembles (opérateur)
print('Intersection (un, deux) =', un.intersection(deux)) # intersection de deux ensembles (méthode)
print('Intersection (un, quatre) =', un & quatre)          # intersection de deux ensembles (opérateur)
print('Différence (un, deux) =', un.difference(deux))     # différence entre deux ensembles (méthode)
print('Différence (deux, un) =', deux - un)               # différence entre deux ensembles (méthode)
print('Différence symétrique (un, deux) =', un ^ deux)     # différence symétrique de deux ensembles

print('trois inclus dans un :', trois.issubset(un))                    # inclusion (méthode) ?
print('deux inclus dans un :', deux <= un)                             # inclusion (opérateur) ?
print('un sur-ensemble de trois :', un.issuperset(trois))              # sur-ensemble (méthode) ?
print('un sur-ensemble de deux :', un >= deux)                         # sur-ensemble (opérateur) ?
print('aucun élément commun (un, quatre) :', un.isdisjoint(quatre))    # aucun élément commun ?
print('aucun élément commun (un, deux) :', un.isdisjoint(deux))        # aucun élément commun ?

cinq = un.copy()                                                       # copie d'un ensemble
print('Égalité un et cinq :', cinq == un)                              # test d'égalité
un.clear()                                                             # vider un ensemble
print('cinq =', cinq, ': un =', un)

cinq |= deux                                  # ajout d'un ensemble d'éléments (opérateur)
cinq |= {11, 13}                              # ajout de plusieurs éléments (opérateur)
print('Ajout : cinq =', cinq)


cinq ^= trois                                 # suppression d'un ensemble d'éléments (opérateur)
cinq ^= {0, 13}                               # suppression de plusieurs éléments (opérateur)
print('Suppression : cinq =', cinq)

cinq &= deux                                  # garder ce qui est commun (opérateur)
print('Commun : cinq =', cinq)
print("Nombre d'élément de deux :", len(deux))  # nombre d'élément (fonction)
```

Shell Python : [évaluer ensembles.py]

```
un = {1, 2, 3, 4, 5}
deux = {0, 1, 2, 5}
trois = {1, 2}
quatre = {8, 9, 7}
Union (un, deux) = {0, 1, 2, 3, 4, 5}
Union (un, quatre) = {1, 2, 3, 4, 5, 7, 8, 9}
Intersection (un, deux) = {1, 2, 5}
Intersection (un, quatre) = set()
Différence (un, deux) = {3, 4}
Différence (deux, un) = {0}
Différence symétrique (un, deux) = {0, 3, 4}
trois inclus dans un : True
deux inclus dans un : False
un sur-ensemble de trois : True
un sur-ensemble de deux : False
aucun élément commun (un, quatre) : True
aucun élément commun (un, deux) : False
Égalité un et cinq : True
cinq = {1, 2, 3, 4, 5} : un = set()
Ajout : cinq = {0, 1, 2, 3, 4, 5, 11, 13}
Suppression : cinq = {3, 4, 5, 11}
Commun : cinq = {5}
Nombre d'élément de deux : 4
```

*Comme pour le dictionnaire, la délimitation se fait au travers de l'opérateur accolade **« {} »**, par contre chaque élément est distinct, alors que pour le dictionnaire, chaque élément est un couple **« clé / valeur »**. Les méthodes et les opérateurs mis à disposition sont bien de nature à traiter les ensembles, ce n'est pas la même démarche que pour les autres collections.*

Le script proposé ne travaille que sur des ensembles d'entiers, mais retenez bien que cette collection, comme les autres, peut stocker des éléments de n'importe quelle nature : des réels, des chaînes, des listes, des tuples, des objets, etc.

Changer de type de collection

Toutes les collections que nous venons de voir sont des objets. Grâce aux constructeurs (méthodes automatiquement appelée en phase de création des objets) de chacune d'entre elles, nous pouvons transformer une collection en une autre

très facilement. Nous passons donc d'une collection à une autre en appelant explicitement le constructeur correspondant à la collection cible, qu'elle une séquence ordonnée ou pas. Voici d'ailleurs un exemple qui nous montre ces fonctionnalités :

ensembles.py

```
# Changer de type de collection (ordonnée ou pas)

liste = [1, 2, 3, 4, 5]
ensemble = {1, 2, 3, 4, 5}
tuples = (1, 2, 3, 4, 5)

chaîne = 'Bienvenue !'
# Tout mettre sous forme de listes
print('Que des listes :')
print(list(ensemble))
print(list(tuples))
print(list(chaîne))


# Tout mettre sous forme d'ensembles
print('Que des ensembles :')
print(set(liste))
print(set(tuples))
print(set(chaîne))

# Tout mettre sous forme de tuples
print('Que des tuples :')
print(tuple(ensemble))
print(tuple(liste))
print(tuple(chaîne))
```

Shell Python : [évaluer ensembles.py]

```
Que des listes :
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
['B', 'i', 'e', 'n', 'v', 'e', 'n', 'u', 'e', ' ', '!']
Que des ensembles :
{1, 2, 3, 4, 5}
{1, 2, 3, 4, 5}
{'u', 'n', 'i', 'e', '!', 'v', 'B', ' '}
Que des tuples :
(1, 2, 3, 4, 5)
(1, 2, 3, 4, 5)
('B', 'i', 'e', 'n', 'v', 'e', 'n', 'u', 'e', ' ', '!')
```

*Nous n'avons pas travailler avec les dictionnaires, puisque chaque élément est composé de deux entités, la **« clé »** et la **« valeur »** qui n'existent pas dans les autres collections.*

*Par ailleurs, nous avons travailler uniquement qu'avec des valeurs entières. Si nous avions pris des caractères à la place, nous aurions pu également passer d'une collection quelconque vers une chaîne de caractères en utilisant le constructeur spécifique **« str() »**.*