RND. Types. Construits

I/ Introduction

Dans la partie précédente, différents types de valeurs ont été présentées comme *int*, *bool*, *float* : ce sont des types **simples.** Il existe également le type *string*, qui est composé d'une **chaîne de caractères** et permet de traiter du texte et également assimilé à un type simple.

Rappel : chaque caractère correspond à un nombre attribué en fonction de la norme choisie (Unicode, ASCII etc.). **Un mot n'a aucun sens pour un ordinateur**, il est traité comme une suite de caractères indépendants.

On peut ainsi associer une **valeur** à une **variable**. Cela est suffisant si l'on n'utilise que peu de données mais s'il devient important, il faudra les « regrouper » par exemple sous forme de tableaux. Ainsi, **une seule variable** peut définir un **ensemble de valeurs**.

II/ Chaînes de caractères

Le langage Python propose le type *string* pour gérer les chaînes de caractères. Un grand nombre de ses **méthodes** sont communes aux types construits. On notera qu'une chaîne de caractères n'est **pas mutable**, ce qui signifie que l'on ne peut **pas** la **modifier**.

Voici quelques méthodes d'accès aux caractères d'un type *string* :

```
# Chaine de caractères"
s = "bonjour"

# Accès aux caractères
print(len(s))   # Affiche le nombre de lettres de 'bonjour' soit 7
print(s[1])     # Affiche 'o' : les indices commencent à 0, attention !
print(s[-1])    # Affiche le dernier caractère de 'bonjour' soit r. C'est l'équivalent de s[len(s)-1]
print(s[1:3])   # Affiche les caractères du second au troisème soit 'on'
print(s[1:])    # Affiche tous les caractères à partir du second soit 'onjour'
print(s[:3])    # Affiche les trois premiers caractères soit 'bon'
print(s[2:6:2]) # Affiche le troisième et le cinquième caractère  soit 'no'
s[2] = 'T'      # Génère une erreur car le type string n'est pas mutable
```

```
7
o
r
on
onjour
bon
no

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-1b85b794239f> in <module>
     10 print(s[:3])    # Affiche les trois premiers caractères soit 'bon'
     11 print(s[2:6:2]) # Affiche le troisième et le cinquième caractère  soit 'no'
---> 12 s[2] = 'T'      # Génère une erreur car le type string n'est pas mutable

TypeError: 'str' object does not support item assignment
```

Il existe bien entendu bien plus de méthodes pour le type *string*. Ne pas hésiter à aller consulter la documentation officielle du langage Python ou de consulter le livre de *Gérard Swinnen*.

Documentation : https://docs.python.org/fr/3.8/library/string.html

*A noter : une **méthode** (ou fonction-membre) est une fonction qui n'est valable que pour le type en question. Cela est une des bases de la programmation objet qui sera étudiée en Terminale.*

III/ N-uplets

Un n-uplet est une **suite ordonnée d'éléments** qui peuvent être **de n'importe quel type**. Tout comme le type *string*, il n'est **pas mutable**.

Exemple : t = 1,2,5,'yop' crée un tuple de quatre éléments.

Le langage Python utilise les parenthèses pour indiquer qu'il s'agit d'un type tuple.

Initialisation d'un type *tuple* :

```
t = 1,'3',(1,2),'bonjour'    # On peut tout à fait créer des tuples contenant des tuples.
print(t)                     # On remarquera la virgule en séparateur (obligatoire)
                             # Les parenthèses sont facultatives sauf si le tuple est vide
t = ()                       # Tuple vide => parenthèses obligatoires
print(t)

t = 3                        # Nombre entier
print(t)

t = 3,                         # 1-uplet (grâce à la virgule)
print(t)
```

```
(1, '3', (1, 2), 'bonjour')
()
3
(3,)
```

Les **méthodes d'accès aux éléments** sont **les mêmes** que celles du type *string*.

IV/ Listes

1/ Définition et construction

Une liste est une **suite d'éléments ordonnée** (comme le tuple) mais **mutable**. En langage Python, ce sont les crochets qui permettent de déterminer un objet de type *list*. Là encore, les types de chaque élément peuvent différer mais cela est rarement utile.

On peut tout à fait créer des « listes de listes » ce qui est très utile pour parcourir les pixels d'une image par exemple ou de déterminer un ensemble de points munis de coordonnées du type (y , f(y)) où *f* est une fonction mathématique.

Exemples :

- Liste1 = [3,6, « bonjour »,-5.4] est une liste.
- Liste2 = [[5,4],[2,-3],[-4,7]] est une liste de listes.

Initialisation d'un type *list* :

```
tab = [1,"coucou",2,-3.1,(6,7)]    # Création d'une liste
print(tab)

tab2 = list(tab)      # Création d'une autre liste
print(tab2)

tab3 = []             # Création d'une liste vide
for i in range(10) :  # Ajout de 10 éléments dans la liste tab3
    tab3.append(i)

print(tab3)
```

```
[1, 'coucou', 2, -3.1, (6, 7)]
[1, 'coucou', 2, -3.1, (6, 7)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

2/ Construction par compréhension

La construction d'une liste par **compréhension** est très utile par sa concision de code. L'instruction est de la forme [*expression(i)* for i in *objet condition*] .
Expression(i) dépend de i (cela peut être une fonction), **objet** peut être un objet itérable ou une instruction à partir de range(), **condition** est un test avec l'instruction *if* mais est facultative.

A noter : il peut y avoir des boucles *for* imbriquées.

Compréhension de *list* :

```
tab = [1,3,5,7,-10,3]   # Création d'une liste
print(tab)              # Affichage de la liste

for i in tab :          # Parcours de la liste et affichage des éléments
    print(i,end=" ")    # séparemment

tab1 = [2*i for i in range(10)]  # Création d'une liste des 10 premiers
print(tab1)                      # nombres pairs


tab2 = [i for i in range(20) if not i%2]  # Même chose que précédemment
print(tab2)

def carre(x) :          # fonction carré
    return x**2

tab3 = [(x,carre(x)) for x in range(20)]   # Création d'une liste de 20 tuples
print(tab3,end=" ")                        # de la forme (x,x²)
```

```
[1, 3, 5, 7, -10, 3]
1 3 5 7 -10 3 [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36), (7, 49), (8, 64), (9, 81),
9), (14, 196), (15, 225), (16, 256), (17, 289), (18, 324), (19, 361)]
```

Ces exemples permettent de comprendre que les listes sont très adaptées à la manipulation d'éléments de type numérique.

3/ Accès aux éléments

Comme pour les type *string* et *tuple*, on accède aux éléments par l'opérateur [*indice*]. Avec une liste, on peut également les modifier directement à partir de leur indice.

Accès aux éléments et modification de liste :

```
tab = [1,3,"coucou",7,-7,3]  # Création d'une liste
print(tab[0])                # Affiche le premier élément
print(tab[-1])               # Affiche le dernier élément
print(tab[1:4])              # Affiche du second au quatrième élément
print(tab[2][1])             # Affiche la deuxième lettre de "coucou"

tab[2] = "bonjour"           # "coucou" est changé en "bonjour"
print(tab)
```

```
1
3
[3, 'coucou', 7]
o
[1, 3, 'bonjour', 7, -7, 3]
```

On peut **parcourir une liste** de deux façons.

- A partir des indices
- En utilisant le mot clé **in** du langage Python

```
liste = ['toto',5,-2,[4,5],'bonjour']

for i in range(len(liste)) :  # A partir des indices
    print(liste[i],end=" ")

print("\n")

for element in liste :        # Avec le mot clé 'in'
    print(element,end=" ")
```

```
toto 5 -2 [4, 5] bonjour

toto 5 -2 [4, 5] bonjour
```

4/ Méthodes des listes

Voici quelques **méthodes** de l'objet *list* qui sont **à connaître** : append(*valeur*), insert(*index*,*valeur*), remove(*valeur*), pop(*index*), index(*valeur*), reverse(), sort().

A noter : ces **méthodes** ne **renvoient aucune valeur** : ce qui signifie que la liste initiale (avant modification) est perdue.

La méthode **append(*valeur*)** de *list* :

```
# Création d'une liste
list1 = [5,10,2]

list1.append(-4)   # Ajout de -4 en fin de liste
print(list1)
```

```
[5, 10, 2, -4]
```

```
# Création d'une liste vide
list2 = []

for n in range(10) :     # Liste des 10 premiers nombres pairs
    list2.append(2*n)

print(list2)
```

La méthode **insert(index,*valeur*)** de *list* :

```
# Création d'une liste
list1 = [2,5,-2,7,10]

list1.insert(3,-10)  # Insertion de -10 à la 4ème place

print(list1)
```

```
[2, 5, -2, -10, 7, 10]
```

La méthode **remove(*valeur*)** de *list* :

```
# Création d'une liste
list1 = [2,5,-2,7,10,5]

list1.remove(5)  # Supprime le premier '5' de la liste
print(list1)

list1.remove(8) # Pas de '8', message d'erreur !!
print(list1)
```

```
[2, -2, 7, 10, 5]

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-5-5f904a911e37> in <module>
      5 print(list1)
      6
----> 7 list1.remove(8) # Pas de '8', message d'erreur !!
      8 print(list1)

ValueError: list.remove(x): x not in list
```

On sera attentif au fait que cette remove(valeur) ne supprime que la **première occurrence de la valeur (**s'il y a).

Attention : un message d'erreur apparaît si la valeur n'est pas dans la liste et stoppe le programme.
On peut utiliser un test pour éviter cela.

```
# Création d'une liste
list1 = [2,5,-2,7,10,5]

if 8 in list1 :     # Test de présence
    list1.remove(8) # Pas de '8', pas de tentative de suppression

print(list1)
```

```
[2, 5, -2, 7, 10, 5]
```

La méthode **pop(*index*)** :

```
# Création d'une liste
list1 = [2,5,-2,7,10,5]

list1.pop(2)      # Suppression de la valeur à la 3ème position
print(list1)

list1.pop(6)      # Erreur, il n'y a pas de 7ème élément !
```

```
[2, 5, 7, 10, 5]

---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-9-876b233a1e3a> in <module>
      5 print(list1)
      6
----> 7 list1.pop(6)      # Erreur, il n'y a pas de 7ème élément !

IndexError: pop index out of range
```

Attention : un message d'erreur apparaît si la l'index dépasse la position du dernier élément de la liste et stoppe le programme
On peut utiliser un test pour éviter ce cas là :

```
# Création d'une liste
list1 = [2,5,-2,7,10,5]

list1.pop(2)      # Suppression de la valeur à la 3ème position
print(list1)

index = 6
if index < len(list1) :  # Comparaison de l'index avec la longueur de la liste
    list1.pop(index)     # Il ne se passe rien

print(list1)
```

```
[2, 5, 7, 10, 5]
[2, 5, 7, 10, 5]
```

A savoir : la fonction **len(*tab*)** donne le nombre d'éléments de l'objet *tab*. Elle est très utile car la plupart du temps, on ne connaît pas la taille d'une *list* (chargement de données d'un fichier par exemple).

Attention : les indices commençant à 0, on accède au dernier élément d'une *list* par l'instruction tab[len(tab) – 1].

A savoir : particularités du langage Python dans l'accès aux éléments :

```
# Création d'une liste
list1 = [2,5,-2,7,10,0]

print(list1[2])   # Affiche le 3ème élément

print(list1[len(list1) - 1])   # Affiche le dernier élément
print(list1[-1])               # dans chaque cas

print(list1[len(list1) - 2])   # Affiche l'avant dernier élément
print(list1[-2])               # dans chaque cas

print(list1[len(list1)])    # Erreur !
```

```
-2
0
0
10
10

---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-13-6ff0d4d9a4fd> in <module>
     10 print(list1[-2])               # dans chaque cas
     11
---> 12 print(list1[len(list1)])    # Erreur !

IndexError: list index out of range
```

Les méthodes **reverse()** et **sort()** :

```
# Création d'une liste
list1 = [2,5,-2,7,10,0]

list1.reverse()   # Inverse l'ordre des éléments de la liste
print(list1)

list1.sort()      # Trie la liste dans l'ordre croissant
print(list1)

list1.sort(reverse = True)   # Trie la liste dans l'ordre décroissant
print(list1)
```

```
[0, 10, 7, -2, 5, 2]
[-2, 0, 2, 5, 7, 10]
[10, 7, 5, 2, 0, -2]
```

5/ Copie de listes

Il y a une différence très importante entre les objets de type *list* et ceux de type *string* ou *tuple* : il est possible de **modifier un élément** d'un type ***list*** par **affectation**.

Attention : **une affectation entre deux listes ne crée pas une nouvelle liste**. Ce qui a des conséquences si l'on **modifie un élément**, il le sera pour les **deux listes**.

```
# Création d'une liste
list1 = [2,5,-2,7,10,0]

list2 = list1   # Affectation d'une nouvelle liste

list1[1] = 0     # list1 est modifié ... mais list2 aussi

print(list1)
print(list2)
```

```
[2, 0, -2, 7, 10, 0]
[2, 0, -2, 7, 10, 0]
```

Pour éviter ces désagréments, on peut utiliser l'une de ces méthodes (il y en a d'autres) :

```
from copy import deepcopy

# Création d'une liste
list1 = [2,5,-2,7,10,0]

list2 = list(list1)
list3 = list1[:]
list4 = deepcopy(list1)   # Nécessite d'importer la fonction 'deepcopy'

list1[1] = 0     # On modifie un élément de list1

print("list1 =",list1)
print("list2 =",list2)
print("list3 =",list3)
print("list4 =",list4)
```

```
list1 = [2, 0, -2, 7, 10, 0]
list2 = [2, 5, -2, 7, 10, 0]
list3 = [2, 5, -2, 7, 10, 0]
list4 = [2, 5, -2, 7, 10, 0]
```

Plus de détails sur les références partagées à ce lien : https://www.youtube.com/watch?v=XkFYa8genoI
Auteur : Cours Python 3, durée : 10 min 56 sec

Remarque : en pratique, il est très rare d'avoir à affecter directement une liste à une autre sans en modifier ses valeurs.

6/ Tableaux et matrices

Une liste peut être composée d'éléments de différents types y compris de listes. Une liste composée de *n* listes de longueurs *p* est appelée **matrice** (*n*,*p*) avec *n* lignes et *p* colonnes.

Exemple : une matrice peut être utilisée pour représenter une image : chaque **élément** représente un pixel et chaque **liste** une ligne de pixels.

```
from PIL import Image
import numpy as np

im1 = Image.open("Logo.Python.png")   # Chargement de l'image logoPython
matrix = np.array(im1)                # Transformation en tableau

print("Pixel =",matrix[10][12])       # Accès au pixel de la 11ème ligne
                                      # et 13ème colonne

print("Ligne de pixels =",matrix[10]) # Affiche les pixels de la 11ème ligne
```

```
[ 24  66 101 255]
[[255 255 255 255]
 [255 255 255 255]
 [255 255 255 255]
 [255 255 255 255]
 [255 255 255 255]
 [255 255 255 255]
 [255 255 255 255]
 [138 158 176 255]
 [112 138 161 255]
 [109 136 159 255]
 [110 137 159 255]
 [115 140 162 255]
 [ 24  66 101 255]
 [ 54  94 128 255]
```

Remarque : il s'agit ici d'une image *png* qui prend en charge la transparence, chaque pixel est donc composé de 4 octets : rouge, vert, bleu et alpha avec des valeurs comprises entre 0 et 255.

Exemples de création de matrices :

```
matrix = []      # Création d'une liste vide
for n in range(4) :
    ligne = [4*n + i for i in range(1,5)]   # Création d'une ligne par compréhension
    matrix.append(ligne)                    # Ajout de la liste à la liste matrix

print(matrix)
print(matrix[0])    # Affiche la première ligne
print(matrix[3][2]) # Affiche l'élément à la 4ème ligne et 3ème colonne
```

```
[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 15, 16]]
[1, 2, 3, 4]
15
```

Méthode de construction par compréhension de matrice à connaître :

```
matrix = [[4*n + i for i in range(1,5)] for n in range(4)]  # Création par compréhension

print(matrix)          # Même affichage que précédemment
print(matrix[0])
print(matrix[3][2])
```

```
[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 15, 16]]
[1, 2, 3, 4]
15
```

Pour **copier** une **matrice** sans **effet de bord**, on utilise la méthode *deepcopy* :

```
from copy import deepcopy

matrix = [[4*n + i for i in range (1,5)] for n in range(4)]
matrix2 = deepcopy(matrix)   # Copie profonde de matrix

matrix[3][2]= 0              # Modification d'un élément de matrix

print("matrix =",matrix)
print("matrix2 = ", matrix2) # Seule matrix a bien été modifiée
```

```
matrix = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 0, 16]]
matrix2 =  [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 15, 16]]
```

V/ Dictionnaires

1/ Définition et construction

Un dictionnaire, un objet de type *dict*, ressemble à une liste de liste de deux éléments.

- Le premier élément est appelé **clé** et est de *type* numérique ou string la plupart du temps (objet **non mutable**).
- Le second élément est appelé **valeur** et est **relié à sa clé**. Il peut être de tout type.

Sa construction est proche de cette d'une liste de liste à deux éléments et le langage Python permet la même souplesse.
On utilise les accolades {} pour délimiter le dictionnaire et les deux points : pour séparer la clé de la valeur.

Exemples de construction d'un dictionnaire :

```
dico = { "yes" : "oui", "no" : "non", "if" : "si" } # Construction directe
print(dico)

liste = [['a',1],['b',2],['c',3]]  # A partir d'une liste de liste à deux éléments
dico = dict(liste)
print(dico)

dico = dict(one=1,two=2,three=3)  # A partir de couples clé = valeur
print(dico)
```

```
{'yes': 'oui', 'no': 'non', 'if': 'si'}
{'a': 1, 'b': 2, 'c': 3}
{'one': 1, 'two': 2, 'three': 3}
```

La méthode de construction par compréhension est bien sûr disponible (et est à connaître) :

```
dico = {x:x**3 for x in range(10)}  # Construction des 10 premiers couples (x,x^3)
print(dico)
```

```
{0: 0, 1: 1, 2: 8, 3: 27, 4: 64, 5: 125, 6: 216, 7: 343, 8: 512, 9: 729}
```

On notera la **syntaxe** avec les accolades qui délimitent le dictionnaire et les deux points séparant clé et valeur.

2/ Accès aux éléments

Un dictionnaire comprend des couples *(clé, valeur).* On peut accéder simplement soit aux couples, soit aux clés, soit aux valeurs.

A savoir : la méthode **keys()** permet l'accès aux clés, la méthode **values()** celle aux valeurs et la méthode **items()** aux couples.

```
semaine = {"lundi" : 1 ,"mardi" : 2 ,"mercredi" : 3 ,"jeudi" : 4 ,"vendredi" : 5}

print(semaine.keys())   # Affiche les clés
print(semaine.values()) # Affiche les valeurs
print(semaine.items())  # Affiche les couples (clé, valeur)
```

```
dict_keys(['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi'])
dict_values([1, 2, 3, 4, 5])
dict_items([('lundi', 1), ('mardi', 2), ('mercredi', 3), ('jeudi', 4), ('vendredi', 5)])
```

A savoir : Le mot clé **in** permet de savoir **si la clé testée existe bien** et non la valeur.

```
clients = {"Pierre" : "Dupond", "Amélie" : "Aulfinger", "Léa" : "Meyer"}

print("Amélie" in clients)    # La clé 'Amélie' existe bien
print("Dupond" in clients)    # La clé 'Dupond' n'existe pas (même si la valeur existe)
```

```
True
False
```

Plusieurs méthodes pour parcourir un dictionnaire (à connaître) :

```
semaine = {"lundi" : 1 ,"mardi" : 2 ,"mercredi" : 3 ,"jeudi" : 4 ,"vendredi" : 5}

for jour in semaine.keys() :  # Parcourt les clés
    print(jour,end = " ")
print("\n")

for jour in semaine :         # Parcourt aussi les clés
    print(jour,end = " ")
print("\n")

for numero in semaine.values() : # Parcourt les valeurs
    print(numero,end = " ")
print("\n")

for couple in semaine.items() :  # Parcourt les couples (clé, valeur)
    print(couple,end=" ")
```

```
lundi mardi mercredi jeudi vendredi

lundi mardi mercredi jeudi vendredi

1 2 3 4 5

('lundi', 1) ('mardi', 2) ('mercredi', 3) ('jeudi', 4) ('vendredi', 5)
```

A savoir : On peut simplement accéder à une valeur si l'on connait sa clé à l'aide de l'opérateur **[*clé*]** (comme pour les listes sauf que l'on donne une clé plutôt qu'un indice :

```
clients = {"Pierre" : "Dupond", "Amélie" : "Aulfinger", "Léa" : "Meyer"}

print(clients['Léa'])   # Affiche la valeur correspondante à la clé Léa
print(clients['Didier'])# La clé n'existe : erreur et le programme s'arrête
```

```
Meyer

---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-45-c4f0c2f9e2ba> in <module>
      2
      3 print(clients['Léa'])   # Affiche la valeur correspondante à la clé Léa
----> 4 print(clients['Didier'])# La clé n'existe : erreur et le programme s'arrête

KeyError: 'Didier'
```

On remarque là aussi que s'il n'y a aucune valeur correspondante, le programme génère une erreur et s'arrête. Pour éviter cela, **deux méthodes possibles** (qui sont aussi **à connaître** mais vous avez l'habitude ☺)

- Vérifier d'abord si la clé existe avec le mot clé **in.**
- Utiliser la méthode **get(*clé*)** du type *dict*.

```python
d1 = {"one" : 1,"two" : 2,"three" : 3}    # Création d'un dictionnaire

if "one" in d1 :
    print(d1["one"])     # Affiche bien la valeur correspondate

if "four" in d1 :
    print(d1["four"])    # N'affiche rien, la clé "four" n'existe pas

print(d1.get("one"))     # Affiche la valeur correspondante
print(d1.get("four"))    # Affiche "None" (pas de clé "four")
print(d1.get("four","clé inexistante"))   # Affiche "clé inexistante"
```

```
1
1
None
clé inexistante
```

3/ Copie de dictionnaire

On retrouve les mêmes problèmes de références partagées qu'avec les listes: on utilisera là encore la fonction *deepcopy()* si l'on a besoin de copier un dictionnaire.

```python
from copy import deepcopy

d1 = {"one" : 1,"two" : 2,"three" : 3}    # Création d'un dictionnaire
d2 = deepcopy(d1)   # Copie profonde de d1

d1["four"] = 4      # Ajout d'un élément dans d1

print("d1=",d1)
print("d2=",d2)     # Seul d1 est modifié
```

```
d1= {'one': 1, 'two': 2, 'three': 3, 'four': 4}
d2= {'one': 1, 'two': 2, 'three': 3}
```

4/ Autres méthodes utiles

Voici quelques méthodes utiles avec un objet type *dict*. La liste n'est bien sûr pas exhaustive.

- L'expression **dict[clé] = valeur** ajoute le couple (clé, valeur) au dictionnaire.
- La fonction ***del*(dict[clé]))** efface un élément du dictionnaire.
- La méthode ***clear*()** vide le dictionnaire.
- L'opérateur **&** permet d'obtenir des éléments communs à deux dictionnaires.

```python
semaine = {"lundi" : 1 ,"mardi" : 2 ,"mercredi" : 3 ,"jeudi" : 4 ,"vendredi" : 5}
semaine2 = {"lundi" : 1 ,"mardi" : 2 ,"mercredi" : 3 ,"samedi" : 5 }

semaine["samedi"] = 6    # Ajout du couple (samedi,6)
print(semaine)

if "samedi" in semaine : # Supprime la clé samedi et sa valeur(si elle existe)
    del(semaine["samedi"])
print(semaine)

print("Clés communes :",semaine.keys() & semaine2.keys())  # Affiche les clés en commun
print("Couples commun :",semaine.items() & semaine2.items())     # Idem (couple (clé, valeur))

semaine.clear()    # Vide le dictionnaire semaine
print("Dictionnaire vide :",semaine)
```

```
{'lundi': 1, 'mardi': 2, 'mercredi': 3, 'jeudi': 4, 'vendredi': 5, 'samedi': 6}
{'lundi': 1, 'mardi': 2, 'mercredi': 3, 'jeudi': 4, 'vendredi': 5}
Clés communes : {'mercredi', 'lundi', 'mardi'}
Couples commun : {('mercredi', 3), ('lundi', 1), ('mardi', 2)}
Dictionnaire vide : {}
```

5/ Listes ou dictionnaires ?

Il est vrai que l'on peut remplacer un objet *dict* comme une liste de listes à deux éléments.
Exemple :

```python
DictSemaine = {"lundi" : 1 ,"mardi" : 2 ,"mercredi" : 3 ,"jeudi" : 4 }
ListSemaine = [ ["lundi",1] , ["mardi",2] , ["mercredi",3] , ["jeudi", 4] ]
```

Il faut toutefois bien saisir les différences entre les objets *list* et *dict* notamment le fait qu'une **liste est mutable** mais **pas** les **clés** d'un **dictionnaire**.

Avantage au dictionnaire	Avantage à la liste de listes
• Peu de modifications de données. • Une donnée non modifiable. • Unicité d'une donnée. • Beaucoup de recherches d'éléments (dans un fichier texte par exemple). • Peu / pas d'ajouts d'éléments.	• Beaucoup de modifications de données. • Peu de recherches d'éléments en particulier. • De nombreux parcours de groupes d'éléments ou de l'objet en entier. • Plusieurs données identiques. • Pas de nécessité de trier systématiquement.

A noter : cette notion de **« donnée non modifiable et unique »** est primordiale et largement utilisée en informatique (par exemple : date de naissance, numéro de sécurité sociale, numéro de facture etc.) et permet de sécuriser les échanges : c'est le principe des bases de données qui seront étudiées en Terminale.

A noter : les éléments d'un objet *dict* sont systématiquement triés dès qu'il est modifié : cela prend beaucoup de temps mais en contrepartie, toute recherche d'élément en particulier est bien plus efficace que dans une liste.