Chapitre 1

Introduction à Python

Ce TP est une introduction à la programmation en Python, qui doit être vu comme un guide. Il ne faut pas hésiter à aller consulter les très nombreux tutoriels disponibles sur internet sur le sujet. Il présuppose des connaissances générale d'un autre langage (boucle, conditions, etc.).

Lorsqu'il est demandé de **taper** un code, il est tout à fait autorisé de le **copier-coller** depuis ce pdf.

Vous trouverez un excellent document de référence sur Python ici.

Certaines questions contienne des explication sur des vidéos. Merci de penser à les regarder chez soi ou de venir en TP avec un casque.

Ce premier chapitre doit être fait en 3 séances (4 maximum) de TP de 1h30, finissez chez vous si besoin. Si vous connaissez déjà Python, ou si vous allez vite, vous pouvez commencer le chapitre suivant plus tôt.

1.1 Introduction

Le langage Python est un langage **interprété** [1], c'est-à-dire que le code source du programme est exécuté pas à pas par un programme (*l'interpréteur*).

Il est possible d'utiliser l'interpréteur *à la volée* : on lui écrit ligne à ligne le code.

Ouvrir un terminal (sous linux) et taper la commande (ne pas taper le $) :

```
$ python3.6
```

L'interpréteur Python est lancé. Taper par exemple (suivi de la touche *entrée* :

```
>>> 3+5
```

De même avec :

```
>>> print(''bonjour'')
```

Enfin :

```
>>> exit()
```

Utiliser l'interpréteur à la volée n'est pas chose aisée et, bien sûr, il est possible de faire autrement.

1. en réalité, c'est un peu plus complexe

Q. 1 : Créer un fichier `test.py` (dans un répertoire dédié au module préalablement créer).

Q. 2 : A l'aide `demacs` par exemple, mettre comme seule ligne du fichier :

```
print("mon code")
```

en veillant à bien mettre cette instruction en début de ligne.

Q. 3 : Dans un terminal, dans le bon répertoire, taper la commande

```
python3.6 test.py
```

En python, l'indentation est très importante, c'est elle qui sert à délimiter les blocs d'instruction (là où dans beaucoup de langage on se sert d'accolades).

Dans le fichier `test.py`, ajouter un espace devant l'instruction `print`. Essayer d'interpréter le code à nouveau. Qu'observe t-on ?

1.2 Les variables

Comme tout langages de programmation évolué, Python utilise des variables :

- Il est inutile de les déclarer,
- Il est inutile de les typer (attention, contrairement à ce qu'on lit parfois, les variables Python sont typées, mais elles le sont *à la volée*, lors de l'interprétation du code, il est donc inutile d'indiquer le type au programme.
- L'affectation se fait avec l'opérateur `=`.

Dans le fichier `test.py`, taper et interpréter le code suivant :

```
a=2
b=3

c=(a+b)/2
print(c)
print(type(c))

a=2.2
b=3.3
c=(a+b)/2
print(c)
print(type(c))

d=int(c)
print(d)
print(type(d))
```

Q. 1 : Qu'observez-vous ? La variable `c` change-t-elle de type ?

Q. 2 : Si l'on ajoute une variable de type `int` à une variable de type `float`, quel est le type de la variable obtenue ? (modifier le programme)

Q. 3 : Quel est le type d'une variable qui contient `"toto"` ?(modifier le programme)

Q. 4 : Quel est le type d'une variable qui contient `'A'` ?(modifier le programme)

Q. 5 : Quel est le type d'une variable qui contient `True` ?(modifier le programme)

Q. 6 : Quel est le type d'une variable qui contient `False` ?(modifier le programme)

On peut trouver sur ce site une liste des types Python.
On peut noter les fonctions suivantes qui sont utiles pour *caster* les variables :

- `int()`, qui transforme en entier,
- `float()`, qui transforme en réel,
- `str()`, qui transforme en chaîne de caractères.

1.3 Les listes

Les listes sont des structures dynamique permettant de stocker, de façon ordonnée, des éléments en nombre variable, qui peuvent être de types différent. Par exemple

```
l=[1,2,3.4,'A']
```

est une liste `l` qui contient 4 éléments, deux entiers, un *floatant* et une chaîne de caractères. Le nombre d'éléments d'un liste sa longueur (que l'on peut obtenir avec la fonction `len()`. Les éléments d'une liste sont indexés de 0 à longueur de la liste moins 1, et l'accès au i-ème élément se fait classiquement à l'aide de crochets.

Par exemple, le code suivant :

```
l=[1,2,3.4,'A']
print(l)
print(len(l))
print(l[0])
l[2]=True
print(l)
```

va produire la sortie

```
[1, 2, 3.4, 'A']
4
1
[1, 2, True, 'A']
```

Qu'indique comme erreur l'interpréteur Python si on essaye d'accéder à un indice de liste supérieur à la longueur de la liste ?

Il est possible dans une liste d'utiliser des indices négatifs permettant de compter à partir de la fin de la liste.

Qu'affiche le code suivant ?

```
l=[1,2,3.4,'A']
print(l[-1])
print(l[-2])
print(l[-3])
```

1.3.1 Faire des tranches (*slice*)

Une manipulation très pratique sur les listes repose sur l'utilisation des *slice*.

Si `l` est une liste, `l[a:b]`, représente aussi une liste qui est constitué des éléments de `l` compris entre l'indice `a` et l'indice `b-1`. De plus :

- Si `a` est omis, il vaut implicitement 0.
- Si `b` est omis, il vaut implicitement `len(l)`.

Tester le code suivant :

```
l=[0,1,2,3,4,5,6,7,8,9]
print(l)
print(l[2:5])
print(l[:3])
print(l[-4:-2])
print(l[-4:])
print(l[-4:9])
print(l[8:1])
print(l[:])
```

Il est possible d'ajouter un troisième entier et d'écrire `l[a:b:c]`.

Q. 1 : Tester le code suivant :

```
l=[0,1,2,3,4,5,6,7,8,9]
print(l)
print(l[1:7:1])
print(l[1:7:2])
print(l[1:8:3])
print(l[::3])
```

Q. 2 : A quoi sert `c` ?

Q. 3 : Que se passe-t-il si `c` est négatif ? Essayer avec différentes valeurs de `a` et de `b` ?

Q. 4 : Que vaut la liste `l[::-1]` ?

Les *slice* sont des outils très pratiques pour la lisibilité et la concision du code. Il est possible de faire des choses bien plus complexes et avancées en les utilisant. Ceux qui sont intéressés pour approfondir peuvent aller voir ce site.

1.3.2 Fonctions et méthodes utiles sur les listes

Il existe de nombreuses fonctions et méthodes applicables aux listes

Les méthodes `pop()` et `append()` permettent respectivement de supprimer le dernier élément d'une liste et d'en rajouter un à la fin.

Tester le code suivant.

```
l=[0,1,2,3,4,5,6,7,8,9,'A','B','C',0,1,2,3]
x=l.pop()
print(l)
print(x)
l.append('S')
print(l)
```

- La méthode `insert(i,x)` permet d'insérer l'élément `x` à l'indice `i` de la liste.
- La méthode `remove(x)` supprime de la liste **la première occurrence** de `x`.

1.4 Les fonctions

La syntaxe pour écrire un fonction en Python est la suivante (attention à l'indentation) :

```
def PlusUn(x):
    resultat=x+1
    return resultat
```

Un gros avantage du typage à la volée est l'universalité d'une fonction.

Tester le code suivant.

```
def Double(x):
    return x+x

print(Double(3))
print(Double(3.5))
print(Double("ABC"))
print(Double([1,2,3]))
```

Il est possible de faire d'écrire des fonctions récursives.

Écrire une fonction qui calcule $n!$.

On peut utiliser un nom de fonction comme paramètre dans une fonction.

Tester le code suivant.

```
def PlusUn(x):
    return x+1

def Double(x):
    return x+x

def Repeter(x,f):
    return f(f(x))

print(Repeter(5,PlusUn))
print(Repeter(5,Double))
```

1.5 Les structures de contrôle

L'utilisation des structures de contrôle est intuitive en Python. Le point essentiel réside dans l'utilisation de l'indentation pour marquer les blocs.

1.5.1 Conditions

La syntaxe d'une condition est

```
if condition:
    instruction1
```

ou

```
if condition:
    instruction1
else:
    instruction2
```

Les conditions utilisant des comparaison classiques entre variables.

Que vous inspire le code suivant (le tester) sur l'indentation ?

```
x=1
if x >= 0 :
    x=x+1
else:
    x=x-1
print(x)

x=1
if x >= 0 :
    x=x+1
else:
    x=x-1
    print(x)
```

Il est possible d'utiliser, pour les conditions, les opérateur booléens `or`, `and` et `not`, par exemple

```
x=1.4
if x >= 0 and x <=2 and not (x==1):
    print(str(x)+" est compris entre 0 et 2, mais ne vaut pas 1.")
```

Les opérateurs pour comparer étant `==`, `!=`, `<=`, `>=`, `<`, `>` (il est possible d'en définir d'autres).

Tester le code suivant :

```
l=[1,2]
k=[2,3]
r=[1,2]
print(l==k)
print(r==l)
```

Écrire une fonction qui retourne la valeur absolue du paramètre passé.

1.5.2 Boucles

La boucle `while` s'utilise classique avec la syntaxe suivante :

```
while condition :
    instructions
```

Q. 1 : Écrire une fonction (en utilisant une boucle `while`) qui prend comme paramètre une liste et un élément `x` et qui compte le nombre d'occurrences de `x` dans la liste (sans utiliser une méthode qui le fait déjà).

Q. 2 : Écrire une fonction qui étant donné un nombre x calcule le plus grand entier n tel que

$$\sum_{i=1}^{n} \frac{1}{i} \leq x.$$

Q. 3 : Écrire une fonction qui calcule une valeur de π^2 à 10^{-2} près en utilisant la formule :

$$\sum_{i=1}^{+\infty} \frac{1}{i^2} = \frac{\pi^2}{6}.$$

La boucle `for` s'utilise à l'aide des listes (ou d'autres structures dites *itérables* avec la syntaxe suivante :

```
for element in liste :
    instructions
```

Tester par exemple :

```
l=[1,2,3,4]
for x in l :
    print(x)
```

Q. 1 : Écrire une fonction (en utilisant une boucle `for`) qui prend comme paramètre une liste et un élément `x` et qui compte le nombre d'occurrences de `x` dans la liste (sans utiliser une méthode qui le fait déjà).

Q. 2 : Écrire une fonction qui étant donné un entier n calcule

$$\sum_{i=1}^{n} \frac{1}{i}$$

Q. 3 : En utilisant une boucle écrire une fonction qui retourne une nouvelle liste contenant les mêmes élément que la première, mais en ordre inverse.

1.5.3 Listes en compréhension

Python dispose d'un mécanisme très pratique (temps de programmation, compacité et clarté du code) pour définir des listes. On peut utiliser la syntaxe suivante :

```
maliste = [f(x) for x in list if condition(x)]
```

Cela crée une liste `maliste` contenant, dans l'ordre, tous les `f(x)` pour `x` dans `liste` vérifiant `condition`.

```
l=[1,2,3,4,5,6,7]
print([x*x for x in l if x >=3])
```

va produire une liste qui prend tous les carrés des éléments de `l` pour des éléments plus grands que 3 :

```
[9, 16, 25, 36, 49]
```

1.5.4 Exercices sur les listes

L'objectif de cette section est de pratiquer un peu plus la manipulation des listes Python. Certaines méthodes ou fonctions de Python permettent de le faire très rapidement. C'est bien et utile de le voir et le savoir, mais prenez aussi le temps de les écrire *à la main* afin de vous familiariser avec Python.

Q. 1 : Écrire une fonction qui cherche la présence d'un élément dans une liste triée en utilisant une approche dichotomique.

Q. 2 : Écrire une fonction qui calcule la somme des éléments d'une liste.

Q. 3 : Écrire une fonction qui calcule la somme des éléments d'une liste dentiers, mais uniquement des éléments impaires.

Q. 4 : Écrire une fonction qui trie une liste d'entier.

Q. 5 : Écrire une fonction qui supprime, dans une liste d'entier, tous les doublons (en préservant l'ordre, on ne garde que la première occurrence de chaque valeur).

1.6 Un peu plus sur les variables

Maintenant que vous connaissez les bases de Python, on va ouvrir un peu la boîte pour permettre une programmation plus avancée.

Pour les sections **??** et **??** vous pouvez regarder (avec un casque si vous êtes en TP) cette vidéo.

1.6.1 Références, identifiants

En Python, toute information est codé dans un objet qui possède au minimum trois propriétés :

- Un identifiant (que l'on peut connaître avec la fonction `id()`,
- Un type (que l'on peut connaître avec la fonction `type()`. Contrairement à ce que l'on entend parfois, Python est bien un langage typé. Le typage se fait cependant de façon dynamique et non explicite (en général).
- une valeur, qui peut être modifiable ou non, à laquelle on accède par différents moyens.

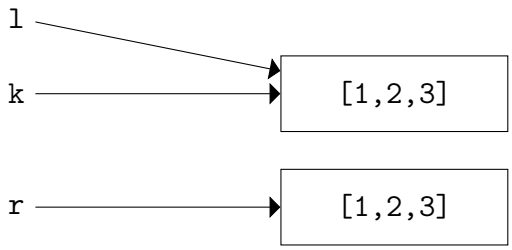
Une **variable** est un nom donné à un objet dans une partie de programme. Un objet peut avoir plusieurs noms. **Le contenu d'une variable est un identifiant d'objet**.

Q. 1 : Exécuter le code suivant.

```
l=[1,2,3]
k=l
r=[1,2,3]
print(id(l))
print(id(k))
print(id(r))
```

Q. 2 : Que remarquez-vous ?

L'explication est la suivante : `k`, `l` et `r` sont des variables, donc contiennent des identifiants. Les deux premières, `k` et `l`, contiennent le même identifiant alors que `r` en contient un différent. On représente souvent ce type de situation avec des schémas, en mettant les valeurs dans des rectangles et des flèches entre les variables et les objets qui correspondent à leur identifiant. Dans l'exemple ci-dessus cela donnerait :



1.6.2 Affectation, modifiables vs non-modifiables

Un objet ne peut ni changer de type, ni d'identifiant. Par ailleurs, on trouve parmi les objets dont la valeur est modifiable (*mutable*) :

- **Listes**,
- **Dictionnaires** (on verra plus tard),
- Set.

Du côté des objets dont la valeur n'est pas modifiable (*non-mutable*) ; on a :

- **Nombres**,
- **Tuples** (on verra plus tard),

- Chaînes (de caractères),
- Frozenset.

La création et destruction d'objets fait appel à des mécanismes complexes gérés par Python.

Regardons maintenant d'un peu plus près l'affectation dans une variable, qui se fait par un `=`. A gauche de ce `=` il y a toujours l'identifiant d'une variable, à droite il y a soit un autre identifiant de variable, soit une valeur fixe. Étudions les deux cas.

- Si `a` et `b` sont deux variables. On considère l'instruction

  ```
  a=b
  ```

  Dans ce cas, l'identifiant de `b` est associé à `a`.
- Si `a` et `valeur` est une donnée comme un entier fixé ou une chaîne de caractères explicite (ou une liste). Deux cas sont possibles, selon que `a` est modifiable ou non.
  - Si `a` est modifiable,

    ```
    a= [1,2,3]
    ```

    crée un nouvel objet dont la valeur est `[1,2,3]` et associe son identifiant à `a`.
  - Si `a` n'est pas modifiable,

    ```
    a=3
    ```

    réutilise un objet dont la valeur est `valeur` et associe son identifiant à `a` (s'il existe) ou le crée si besoin.

Il faut noter que modifier la valeur contenue dans un objet modifiable, avec par exemple une instruction comme (si `a` est une liste)

```
a[2]=3
```

ne modifie pas l'identifiant de `a`.

Illustrons cela. Taper (copier/coller) le code suivant.

```
l=[1,2,3]
k=l
r=[1,2,3]
print ("id(l)",id(l))
print ("id(k)",id(k))
print ("id(r)",id(r))
l[0]=2
r=[3,4,5]
print ("id(l)",id(l))
print ("id(k)",id(k))
print (k)
print ("id(r)",id(r))
```

Qu'observer vous ? Expliquer.

Même question avec :

```
a=1
b=a
c=1
ch="je suis content"
ch2="je suis content"
print("id(a)",id(a))
print("id(b)",id(b))
print("id(c)",id(c))
print("id(ch)",id(ch))
print("id(ch2)",id(ch2))
a=2
print("id(a)",id(a))
print("id(b)",id(b))
```

Que se passe-t-il lorsque l'on exécute le code suivant (expliquer).

```
a=1
k=[2,3]
r=[4,5]
l=[a,k,r]
a=2
k=[3,2]
r[0]=1
print(l)
```

1.6.3 Passage de paramètre des fonctions

Intéressons nous maintenant aux passages de paramètres dans une fonction. Lors de l'appel d'une fonction, une copie locale à la fonction de chaque paramètre est créée et liée au même identifiant que le paramètre (comme dans de nombreux langages de programmation). Et c'est sur cette copie locale que travaille l'appel de la fonction.

Ce point est développé dans cette vidéo.

Que se passe-t-il en mémoire lors de l'appel du code suivant ?

```
def f(x,l):
    x=x+1
    l[0]=l[0]*2
    l=[3]
    l[0]=1

k=[1]
f(k[0],k)
print(k)
```

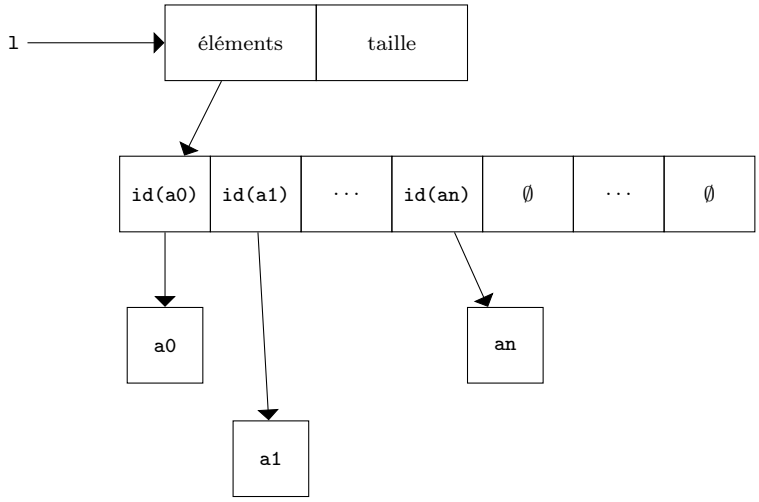
1.6.4 Approfondissement sur les listes

Les points abordés dans cette section sont développés dans cette vidéo pour le codage des listes et dans cette vidéo pour la copie de listes.

Nous allons entrer un peu dans le détail du codage des listes en mémoire. Lorsque l'on utilise une liste, par exemple avec la commande ci-dessous :

```
l=[a0,...,an]
```

L'objet créé en mémoire peut se schématiser ainsi



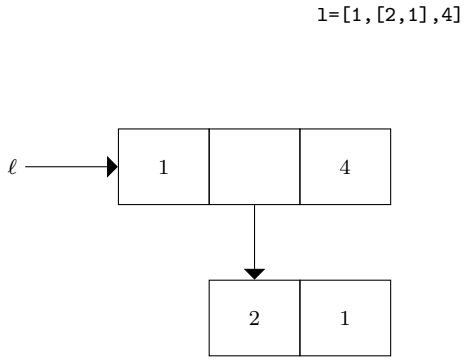
La liste `l` contient l'identifiant d'un élément qui contient deux informations, d'une part un autre identifiant (élément sur la figure), ainsi qu'un entier qui est le nombre d'éléments dans la liste.

- L'identifiant `elements` référence une suite contiguë de bloc mémoires (au moins au temps que la taille de la liste, souvent plus, comme on le verra plus tard). Ces identifiants sont ceux des éléments constituant la liste (pour les premiers).
- La taille de la liste est le nombre d'élément utilisé de façon effective par la liste dans le programme, les autres pouvant être vide.

Quand les éléments d'une liste sont non modifiable, on utilise en général une notation un peu plus compacte. Par exemple,

```
l=[1,[2,1],4]
```

sera représentée ainsi :



Lorsque l'on utilise des méthodes qui ajoutent ou suppriment un élément d'une liste, la taille utilisée en mémoire par la liste est dynamiquement modifiée.

La fonction `getsizeof()` du package `sys` permet de connaître la taille mémoire utilisée par un objet. En divisant par 8 (taille de codage d'un identifiant), on peut connaître le nombre de bloc d'identifiants allouée pour la liste. Utiliser le programme suivant.

```
import sys

l=[]
b=sys.getsizeof(l)
res=[]
for i in range(100):
    res.append((sys.getsizeof(l)-b)/8)
    l.append(i)
print res
```

1. Quel est, selon vous, le rôle de la variable `b` ?
2. Qu'observez-vous ?

Que se passe-t-il lors de l'exécution du code suivant (expliquez le résultat).

```
l=[1,2]
k=[3,l]
l.append(k)
l[2][1][0]=0
print(l)
```

L'opérateur `*` permet de créer une liste contenant n fois le même identifiant avec la syntaxe :

```
l=[a]*n
```

Tester le code suivant et expliquer la différence de résultat.

```
l1=[[1,2]]*4
l1[0][0]=0
print(l2)

l2=[]
for i in range(4):
   l2.append([1,2])
l2[0][0]=0
print(l2)
```

Terminons cette section sur les listes avec la copie de liste.

Expliquer le résultat du code suivant en vous appuya,t sur ce qui a été vue avant.

```
l=[1,2,3]
copiedel=l
l[0]=0
print(copiedel)
```

On souhaite parfois avoir une copie qui soit un nouvel objet. On peut le faire avec une boucle (par exemple), mais l'utilisation du package `copy` permet de le faire facilement (comme avec d'autres objets).

```
import copy

l=[1,2,3]
copiedel=copy.copy(l)
l[0]=0
```

```
print(copiedel)
```

donnera comme résultat :

```
[1, 2, 3]
```

Plus précisément, la fonction `copy.copy()` crée une liste de la même taille et recopie les idenfiants des éléments de la première liste dans la seconde.

Que remarquez vous sur l'exécution du code suivant :

```
l=[1,[2,3],2]
copiedel=copy.copy(l)
l[0]=0
print(copiedel)
l[1][0]=0
print(copiedel)
print(l)
```

On a parfois de créer une copie complètement disjointe, pour ne pas se retrouver dans la situation précédente. On peut pour cela utiliser la fonction `deepcopy()` du package `copy`

Que remarquez vous sur l'exécution du code suivant :

```
l=[1,[2,3],2]
copiedel=copy.deepcopy(l)
l[0]=0
print(copiedel)
l[1][0]=0
print(copiedel)
print(l)
```

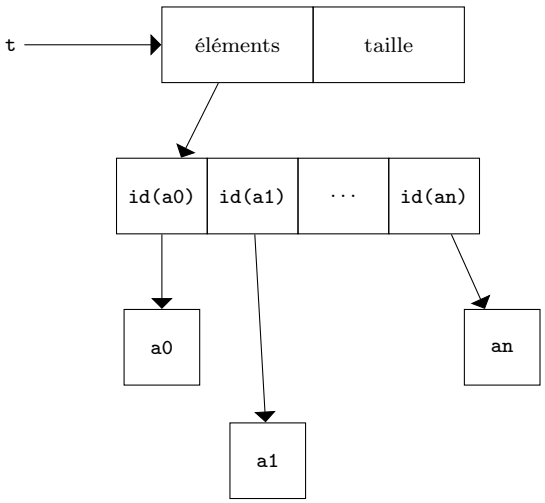
1.6.5 Tuples et dictionnaires

Le type tuple permet de coder des collections à la différence des listes qu'ils ne sont pas modifiable (on précisera un peu plus loin ce point). On utilise une syntaxe proche des listes, mais avec des () au lieux des [].

En mémoire un tuple est codé comme une liste :

```
t=(a0,...,an)
```

sera codé en mémoire par :



On en peut ni modifier la taille d'un tuple (il faut en créer un nouveau si on veut ajouter un élément), ni changer les identifiants qu'il contient. En revanche, ce qui est pointé par les identifiants peut changer.

Q. 1 : Que donne le code suivant ?

```
t=(3,4)
t[0]=1
```

Q. 2 : Même question avec :

```
t=([1,2],4)
t[0][0]=0
```

Q. 3 : Qu'est-ce que cela illustre ?

Les fonctions Python, lorsqu'elles retournent plusieurs arguments, retournent des tuples. Par ailleurs, comme on va le voir, les tuples, sous certaines conditions, peuvent être utilisés comme clés de dictionnaire.

Un dictionnaire est un tableau associatif qui à une clé associe une valeur.

- Les clés sont nécessairement des objets hachables et donc *non mutables*.
- *Mutable*.
- Plus souple qu'une liste...
- Mais moins efficace (en temps/mémoire)...
- Même si optimisé.

Pour définir un dictionnaire on utilise la syntaxe

```
{cle1:valeur,...}
```

la liste des clés/valeurs est accessible via les fonctions `keys()` et `values()`.

Un petit exemple :

```
D={'a':3,'b':"maison",1:[3,4],6:(4,2)}
print D.keys()
print D.values()
D["rouge"]=1
D[1].append(5)
D['a']=1
print D
```

donnera

```
['a', 1, 'b', 6]
[3, [3, 4], 'maison', (4, 2)]
{'a': 1, 1: [3, 4, 5], 'b': 'maison', 6: (4, 2), 'rouge': 1}
```

Le codage en interne des dictionnaires est assez complexe et utilise des tables de hachage. Nous n'en verrons pas plus. Il faut néanmoins noter qu'il n'y a pasde mécanisme de réduction de taille sur les dictionnaires comme sur les listes.