J.Courtin *PC* LVH Caen*

INFORMATIQUE

Les dictionnaires en Python

2 - Les dictionnaires et ensembles

A - Les ensembles

Ce sont des objets spécifiques à Python, de «programmation haut niveau», c-à-d que c'est une structure de données abstraite proche de notre intuition mathématique.

On va retrouver dans le code toutes les opérations mathématiques ensemblistes.

Création d'un ensemble : **Un ensemble est une collection d'éléments distincts**

```
>>> E1={1,3,5,7,9}
>>> E1
set([9, 3, 1, 5, 7])
```

```
>>> Ep={2,3,5,7,11,13,17,19}
>>> Ep
set([2, 3, 5, 7, 11, 13, 17, 19])
```

```
>>> E0={2,3,2,5,3,7,5,1}
>>> E0
set([1, 2, 3, 5, 7])
```

=> L'ordre des éléments n'est donc pas nécessairement celui de la saisie !

```
>>> mySet={'Toto', 3.14159, True, lambda x : x**2}
>>> mySet
set([3.14159, True, 'Toto', <function <lambda> at 0x2c5770>])
```

```
>>> vide=set()
>>> vide
set([])
```

Un ensemble peut être vide ou contenir des éléments de types distincts.

La commande set() fonctionne avec tout itérable :

```
>>> E2=set([0,2,4,6,8,10])
>>> E2
set([0, 2, 4, 6, 8, 10])
```

```
>>> E3=set(range(1,11))
>>> E3
set([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

```
>>> Setoto=set("Toto")
>>> Setoto
set(['T', 'o', 't'])
```

Un ensemble ne contient jamais deux fois le même élément.

Opérations sur les ensembles :

```
>>> E1 | E2
set([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

>>> E1.union(E2)
set([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

```
>>> E1 & E2
set([])

>>> E1.intersection(E2)
set([])
```

```
>>> Ep & E2
set([2])

>>> E2 | Ep
set([0, 2, 3, 4, 5, 6, 7, 8, 10, 11, 13, 17, 19])
```

Union Intersection Différence

```
>>> set(['A','B','C']) - set(['C','D','E'])
set(['A', 'B'])

>>> set(['A','B','C']).difference(set(['C','D','E']))
set(['A', 'B'])
```

R–S

```
>>> set(['A','B','C']).symmetric_difference(set(['C','D','E']))
set(['A', 'B', 'E', 'D'])
>>>
>>> set(['A','B','C']) ^ set(['C','D','E'])
set(['A', 'B', 'E', 'D'])
```

R∪S – R∩S

Test d'appartenance : les ensembles sont optimisés pour cela !

```
>>> 7 in E2
False
>>> 7 in E1
True
```

```
>>> "Toto" in mySet
True
>>> "Toto" in Setoto
False
```

```
>>> {0,4}.issubset(E2)
True
>>> 2 in E2&Ep
True
```

```
>>> E2.issuperset({2,4,6})
True
>>> (E1-E2).isdisjoint(E2-E1)
True
```

Bien que l'ordre ne soit pas maintenu : on peut passer en revue les éléments par itération.

```
>>> for i in mySet:
...     print(i)
...
```

→

```
3.14159
True
Toto
<function <lambda> at 0x2c5770>
```

Recherche d'élément ou appartenance :

→ **des structures de données optimisées**

Trouver un élément dans une liste --- ou --- trouver un élément dans un ensemble :

```
>>> import timeit   #utilisation du module timeit !!
>>> timeit.timeit(stmt='-1 in L', setup='L=range(1000)', number=1000000)
34.591264963150024
```

```
>>> timeit.timeit(stmt='-1 in S', setup='S=set(range(1000))', number=1000000)
0.09568214416503906
```

	Liste	Set
n=1	0.1277	0.0957
n=10	0.4514	0.0943
n=100	3.5651	0.0963
n=1000	34.5913	0.0957
	O(N)	O(1)

Chronométrage
(temps en seconde)

Les ensembles et les dictionnaires utilisent une fonction de Hash, ce qui rend ces structures optimales pour la recherche et l'ajout d'un élément.

B - Les dictionnaires

L'idée du dictionnaire s'appuie sur le concept de correspondance **{clé : valeur}**
Un dictionnaire est donc un **ensemble de clé : valeur** => **chaque clé est unique !**

Exemple : Dictionnaire de la langue française {nom_commun : définition}
Fichier de données {# de ligne : tuple de données}

Cette notion est donc centrale dans l'étude des bases de données.

Création explicite d'un dictionnaire :

```
>>> Clients = { 'riri': ('Duck',10,'pizza'),          'mickey': ('Mouse',25,'Hamburger'),
...             'donald': ('Duck',20,'Hotdog'),       'daisy': ('Duck',18,'pizza'),
...             'loulou': ('Duck',9,'salad'),         'minie':('Mouse',10,'pizza'),
...             'fifi': None,                         'picsou': ('Duck',999999999, 'Hamburger')  }

>>> Clients
{'riri': ('Duck', 10, 'pizza'), 'mickey':  etc....}
```

(nom, argent, plat préféré)

Le dictionnaire est optimisé pour trouver rapidement la valeur (ici un tuple) **associée à une clé**, dans une base de donnée très grande (milliards d'entrées).

Opérations de base sur un dictionnaire :

Valeur associée à la clé :

```
>>> Clients['daisy']
('Duck', 18, 'pizza')
```

Toutes les clés :

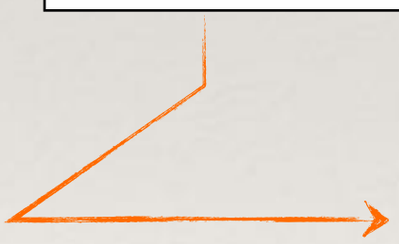
```
>>> Clients.keys()
['riri', 'mickey', 'donald', 'daisy', 'fifi', 'picsou', 'loulou', 'minie']
```

Toutes les valeurs :

```
>>> Clients.values()
[('Duck', 10, 'pizza'),  ('Mouse', 25, 'Hamburger'),  ('Duck', 20, 'Hotdog'),  ('Duck', 18, 'pizza'),
 None,  ('Duck', 999999999, 'Hamburger'),  ('Duck', 9, 'salad'),  ('Mouse', 10, 'pizza')]
```

Modifier le dictionnaire :

```
Clients['fifi']=('Duck',7,'Donut')
```



```
>>> Clients['fifi']
('Duck', 7, 'Donut')
```

Pop : renvoit la valeur / éjecte l'élément

```
>>> 'minie' in Clients
True

>>> Clients.pop('minie')
('Mouse', 10, 'pizza')

>>> 'minie' in Clients          #minie n'est plus
False                           #dans le dictionnaire
```

Le dictionnaire est donc (comme les ensembles) optimisé pour la recherche, l'insertion ou la modification d'un élément : tout ce dont on a besoin pour gérer une base de données.

Ajout d'un autre dictionnaire : **update**

```
>>> monDico={'A':1, 'B':2, 'C':3, 'D':2}
```

```
>>> complement={'E':3,'F':7}
```

```
>>> monDico.update(complement)
>>> monDico
{'A': 1, 'C': 3, 'B': 2, 'E': 3, 'D': 2, 'F': 7}
```

Ou d'un seul élément :

```
>>> monDico.update({'A':5})
>>> monDico
{'A': 5, 'C': 3, 'B': 2, 'E': 3, 'D': 2, 'F': 7}
```

Attention :

Le premier jeu **(clé : valeur)** est remplacé par le nouveau, car chaque clé (ici 'A') doit être unique !

Création à partir d'une liste de tuple : (clé, valeur)

```
>>> convert=[('A','T'), ('T','A'), ('C','G'), ('G','C')]     #  table de conversion ADN
                                                              #  Liste de tuple cle-valeur
>>> convertDict = dict(convert)
>>> convertDict
{'A': 'T', 'C': 'G', 'T': 'A', 'G': 'C'}
```

Itérations sur un dictionnaire

La structure de données est munie d'un itérateur sur les clés du dictionnaire

```
>>> for key in monDico:
...     print key, '  :  ', monDico[key]
...
A   :   5
C   :   3
B   :   2
E   :   3
D   :   2
F   :   7
```

Suppressions :

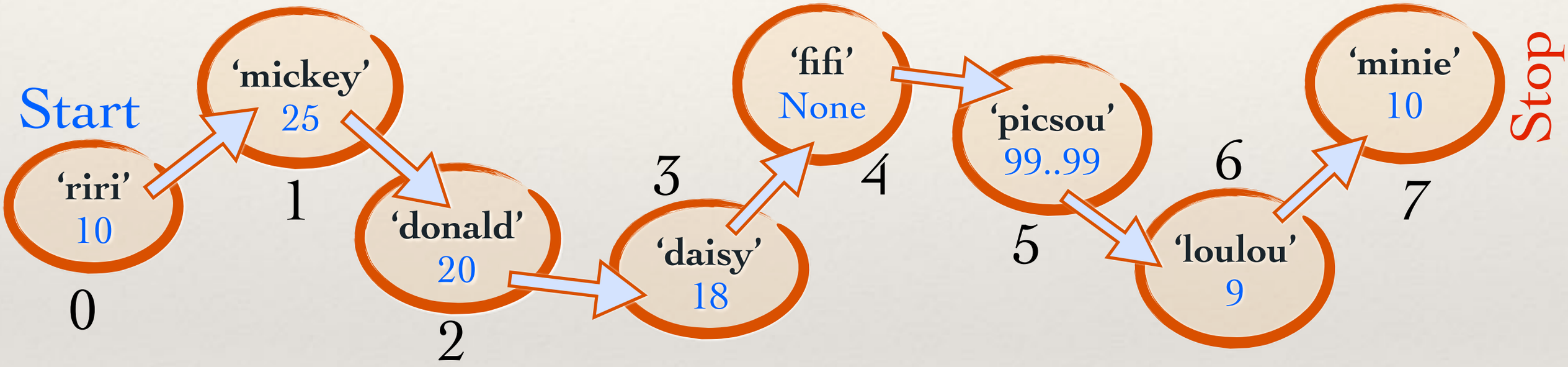
```
>>> del monDico['C']                    # suppression d'une entrée (cle : valeur)
>>> monDico
{'A': 5, 'B': 2, 'E': 3, 'D': 2, 'F': 7}

>>> monDico.clear()                     # vider le dictionnaire
>>> monDico
{}
```

Un point algorithmique : la table de Hachage

Cas d'une liste : Structure de données ?

- Une liste est simplement la donnée d'un élément de départ et d'une longueur
- Chaque élément connait seulement, son unique successeur → .next

[On peut la munir d'un itérateur qui s'arrête lorsque l'indice dépasse la longueur de la liste]



PB : Comment **trouver**, **insérer** ou **supprimer** un élément dans une liste ?

Ex : >>> L[7]

→ Il n'y a pas d'autre choix que de parcourir toute la liste : **coût en temps O(N)**

Rq : L'insertion (ou la supression) en elle même n'est pas un problème : **coût en temps O(1)** mais encore faut-il trouver l'élément en question.

=> **En pratique cette implémentation est beaucoup trop lente**

La table de Hachage permet de trouver un élément dans un temps O(1)

clé : «donald»

>>>hash(«donald»)
2

fonction de Hash

Gestion des «collisions»

>>>hash(«mickey»)
6

clé : «mickey»

Indices	Tableau contenant des listes d'éléments		
0			
1			
2	donald : 12		
3	minie : 8		
4			
5	On parcourt la liste de l'élément 6 du tableau		
6	riri : 5	mickey : 25	loulou : 9
7			
8	picsou : 11		
9	fifi : 8		
10			
11			
12	daisy : 15		
13			
14			
15			

Les collisions sont très rares

Les clés d'un dictionnaire doivent être *hashables* :

- Tous les types immutables de python
  (numériques, chaînes de caractères, tuples, etc. mais pas les listes ni les dictionnaires !)

- Tous les types disposant des méthodes ***__hash__*** et ***__eq__*** ou ***__cmp__***

```
>>> Dico = { [1,2] : 3 }     # cle -> de type list

Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

```
hash(...)    hash(object) -> integer

             Return a hash value for the object.  Two objects with the same value have
             the same hash value.  The reverse is not necessarily true, but likely.
```

Exemples de fonction de Hash :

```
>>> hash(2**20);   2**20
1048576
1048576

>>> hash(2**50);   2**50
262144
1125899906842624L
```

```
>>> hash(2**31-1); 2**31-1
2147483647
2147483647L

>>> hash(2**31);  2**31
-2147483648
 2147483648L
```

```
>>> hash("Toto");
-1620250612



>>> hash("T");
-2132869547
```