Listes, tuples, dictionnaires, sets...

Types / Structures de données python :

- scalaires (`int` , `float` , `bool` , `NoneType`)
- conteneurs :
  - `str`
  - `bytes`
  - `list`
  - `tuple`
  - `dict`
  - `set`
  - ...

Une des forces du langage est de proposer **en standard** des structures de données puissantes (qu'on a aussi dans d'autres langages, mais souvent dans des libs séparées).

Comparatif des structures de données (duck-typing)

- conteneur (on peut utiliser `in`)
- mutable (on peut modifier le contenu)
- séquence (on peut utiliser `[.]`)
- hachable (récursivement non mutable)

Liste ou tuple ?

- Caractère mutable ou pas
- Sémantique :
    - une liste est une collection d'éléments plutôt de même type (une liste d'entiers, une liste de chaînes, une liste de tuples...).
    - un tuple est plutôt un enregistrement (avec un nombre fixe de valeurs) : *(nom, prénom, age, numéro sécu)*

Dans les programmes, il est question de p-uplet nommés (ce ne sont ni des tuples ni des tuples nommés, mais des dictionnaires...)

Tuples nommés

```
from collections import namedtuple

Identite = namedtuple('Identite', ['prenom', 'nom', 'age'])
p1 = Identite('Bilbon', 'Sacquet', 131)

>>> p1.nom
    'Bilbon'
```

Dataclasses (3.7)

```
from dataclasses import dataclass

@dataclass
class Identite:
    prenom: str
    nom: str
    age: int


p1 = Identite("Bilbon", "Sacquet", 131)

>>> p1.nom
   'Bilbon'
```

Dictionnaires

Autres noms : Tables de hachage, tableaux associatifs

Objectif :

On donne une clé, le dictionnaire renvoie la valeur associée à cette clé.

Pourquoi pas des listes ?

On pourrait faire pareil ainsi :

```
def clevaleur(cles, valeurs, lacle):
    k = cles.index(lacle)
    return valeurs[k]

les_cles = ['prenom', 'nom', 'age']
les_valeurs = ['Bilbon', 'Saquet', 131]

>>> clevaleur(les_cles, les_valeurs, 'age')
    131
```

Quel est l'intérêt des dictionnaires alors ?

Intérêt

L'intérêt est le temps d'accès.

Obtenir la valeur associée à la clé se fait en **O(1)**, alors que l'implémentation naïve proposée est en O(n).

Complexité des opérations sur les dictionnaires :

- appartenance d'une clé : O(1)
- ajout, modification d'un élément à partir de la clé : O(1)
- récupération d'un élément à partir de la clé : O(1)
- appartenance d'une valeyr : O(n)
- effacement d'une valeur : O(1)

Utiliser des dictionnaires en pratique

Création

```
di1 = {}

di2 = dict([("nom": "Potter"), ("prenom": "Harry"),
            ("age": 17)])
di2["Baguette"] = "Houx"
di2["age"] = 11

di3 = {k: k**3 for k in range(0,10)}
```

Rappel des valeurs

```
print(di2["nom"])
v = 2 * di2["age"]
print(di2["maison"])
# ERREUR !!!
print(di2.get("maison", "Gryfondor"))
```

Parcours

```
for k, v in di2.items():
    print(k, v)
```

Faits à connaître

- Une clé est nécessairement unique
- Une clé doit être hachable (récursivement non mutable)
- Les dictionnaires ne sont pas ordonnés (pas vraiment exact)

Comment ça marche ?

Vous voulez vraiment le savoir ?

> CPython’s dictionaries are implemented as resizable hash tables. Compared to B-trees, this gives better performance for lookup (the most common operation by far) under most circumstances, and the implementation is simpler.
>
> Dictionaries work by computing a hash code for each key stored in the dictionary using the hash() built-in function. The hash code varies widely depending on the key and a per-process seed; for example, “Python” could hash to -539294296 while “python”, a string that differs by a single bit, could hash to 1142331976. The hash code is then used to calculate a location in an internal array where the value will be stored. Assuming that you’re storing keys that all have different hash values, this means that dictionaries take constant time – O(1), in Big-O notation – to retrieve a key.

Ensembles : des dictionnaires particuliers (en qque sorte)

Un ensemble est à peu près un dictionnaire sans valeur (juste les clés).

Bon nombre d'**opérations sur les ensembles** sont disponibles (intersection, union, différence, différence symmétrique)

Pour les mêmes raisons que dans un dictionnaire :

- chaque élément doit ête hachable
- chaque élément est unique
- le test d'appartenance est en O(1)

Si vous devez éliminer les doublons de quelque chose, pensez aux ensembles (facilité d'utilisation + rapidité)

```
lst= [1, 2, 3, 2, 2, 4, 3, 1, 1, 5, 2, 3, 4]
print(set(lst))
{1, 2, 3, 4, 5}
```

(a priori en O(n) alors qu'une implémentation naïve serait en $O(n^2)$ avec des listes non triées et O(nlog(n)) avec des listes triées.)