

1 Les dictionnaires

Et si je dois acheter 5 pains, 2 artichauts, 2 plaquettes de beurre et 1 bouteille de vin ?

Il faut cette fois *associer*, d'une part, des entités (**beurre**, **pain**, etc), et d'autre part des informations sur chaque entité (ici une quantité entière mais ce pourrait être une information aussi complexe que l'on veut). La structure de données qui mémorise cela est classiquement appelée une « liste d'association » mais en raison de la ressemblance avec un dictionnaire qui associe à chaque mot une définition, Python appelle ce type `dict` (comme dictionnaire).

L'ensemble des dictionnaires est constitué des expressions de la forme

$$\{ e_1:d_1 , \dots , e_n:d_n \}$$

où les e_i sont des données *élémentaires* quelconques et les d_i sont des données quelconques. Par donnée élémentaire, on entend ici une donnée qui n'est pas elle-même une liste ou un dictionnaire.

```
>>> coursesPrecises = { "baguette":2 , "vin":1 , "saumon":2 }
>>> coursesPrecises[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 1
>>> coursesPrecises["vin"]
1
```

On constate que l'on n'accède pas au contenu d'un dictionnaire par un indice entier donnant la position dans le dictionnaire mais, et c'est plus simple, par le nom des entités appartenant au dictionnaire.

Un dictionnaire peut être hétérogène :

```
>>> chelou = { "vin":2 , (4,5,6):"c'est un triplet" , 3.5:78 }
>>> type(chelou)
<type 'dict'>
>>> chelou[(4,5,6)]
"c'est un triplet"
>>> chelou[3.5]
78
>>> chelou[2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 2
```

L'accès à un dictionnaire se fait par les entités, pas par les informations qui lui sont associées.

On peut compléter ou modifier un dictionnaire avec une instruction d'affectation :

```
>>> coursesPrecises
{'baguette': 2, 'saumon': 2, 'vin': 1}
>>> coursesPrecises["saumon"]=1
>>> coursesPrecises
{'baguette': 2, 'saumon': 1, 'vin': 1}
>>> coursesPrecises["gateau"]=8
>>> coursesPrecises
{'baguette': 2, 'saumon': 1, 'vin': 1, 'gateau': 8}
```

On peut aussi supprimer une entité (et son information associée) avec `del`, obtenir le nombre d'entités avec `len` et tester si une entité est dans le dictionnaire avec `in`.

```

>>> del coursesPrecises["vin"]
>>> coursesPrecises
{'baguette': 2, 'saumon': 1, 'gateau': 8}
>>> len(coursesPrecises)
3
>>> "gateau" in coursesPrecises
True
>>> "chou" in coursesPrecises
False

```

Enfin, on peut *itérer* un bloc d'instructions sur toutes les entités d'un dictionnaire, un peu comme avec un `while` sur une chaîne de caractères ou sur une liste. Pour cela on utilise `for...in...`

```

>>> def total (d) :
...     s=0
...     for e in d :
...         s = s + d[e]
...     return s
...
>>> total(coursesPrecises)
11
>>> total(chelou)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in total
TypeError: unsupported operand type(s) for +: 'int' and 'str'

```

Moralité : un dictionnaire avec des entités ayant toutes le même type et des informations ayant toutes le même type, c'est souvent mieux !

2 Exemples de programmes sur les dictionnaires

Un dictionnaire peut être typiquement utilisé pour collecter un ensemble d'informations à propos d'une collection d'objets. Prenons comme premier exemple un stock de médicaments :

```

>>> NbreDeBoites = {
...     "Doliprane":10,
...     "Efferalgan":3,
...     "Dafalgan":1,
...     "Levothyrox":0,
...     "Kardegic":15,
...     "Spasfon":20,
...     "Tahor":11,
...     "Voltarene":4,
...     "Methadone Aphp":5,
...     "Eludril":6,
...     "Ixxprim":3,
...     "Paracetamol Biogaran":2
... }

```

On peut alors par exemple calculer le nombre total de boîtes de médicaments dans un tel stock :

```

>>> def totalDico (d) :
...     r=0
...     for produit in d :
...         r = r + d[produit]
...     return(r)
...
>>> totalDico(NbreDeBoites)
80

```

On peut aussi calculer la liste des médicaments à renouveler dans le stock, en fonction du nombre minimum de boîtes que l'on souhaite :

```
>>> def renouveler(d,mini) :
...     commande=[]
...     for article in d :
...         if d[article] < mini :
...             commande=commande+[article]
...     return(commande)
...
>>> renouveler(NbreDeBoites,4)
['Levothyrox', 'Paracetamol Biogaran', 'Dafalgan', 'Ixprim', 'Efferalgan']
>>> renouveler(NbreDeBoites,2)
['Levothyrox', 'Dafalgan']
```

Si les clefs doivent être des données élémentaires, les informations qui leur sont associées peuvent en revanche être aussi complexes que l'on veut. Entre autres, ce peuvent être elles-mêmes des dictionnaires, comme pour ce second exemple de programmes sur les dictionnaires.

```
>>> anniv = { "Pierre" : {"jour":3,"mois":"jan","an":1965} ,
...           "Paul" : { "jour":18,"mois":"nov","an":1998} ,
...           "Irène" : {"jour":25,"mois":"mar","an":1982} }
>>> anniv["Irène"]
{'mois': 'mar', 'jour': 25, 'an': 1982}
>>> anniv["Paul"]["an"]
1998
```

Supposons que l'on veuille écrire une fonction `lesAges` qui prend en entrée un dictionnaire `d` d'anniversaires similaire à `anniv` et une date courante `t`, et retourne en sortie la liste des âges des personnes du dictionnaire `d`.

```
def lesAges (d,t) :
    r = []
    for p in d :
        r = r + [...??...]
    return r
```

calculer l'âge d'une personne en fonction de son « dictionnaire anniversaire » et du dictionnaire représentant la date courante est de prime abord compliqué. Dans de tels cas, il faut toujours procéder de la manière suivante :

- on fait l'hypothèse qu'il existe des fonctions qui résolvent les problèmes compliqués (ici faire la différence de deux dates en nombre d'années entières : `diffDates(t1,t2)`);
- on programme comme si ces fonctions existaient (ici, `r = r + [diffDates(d[p],t)]`); naturellement Python n'acceptera cette programmation qu'après avoir réellement programmé ces fonctions;
- on s'attaque ensuite, une par une, aux fonctions identifiées par le processus précédent, et on leur applique exactement la même approche;
- la programmation est terminée lorsque les fonctions deviennent suffisamment simples pour ne plus avoir besoin de faire appel à des fonctions hypothétiques.

Ici, pour programmer `diffDates`, si la date de `t1` vient après celle de `t2` dans l'année, il suffit de faire la soustraction des années; sinon il faut soustraire 1 au résultat :

```
def diffDates (t1,t2) :
    if apresAnnee(t1,t2) :
        return t1["an"] - t2["an"]
    else :
        return t1["an"] - t2["an"] - 1
```

La fonction hypothétique qu'il faut maintenant programmer est `apresAnnee`. Ce qui est difficile est alors de comparer des mois qui ne sont pas des entiers mais des chaînes de caractères. Qu'à cela ne tienne, on fait l'hypothèse que la fonction `numero` fournit le numéro du mois.

```

def apresAnnee (u,v) :
    if numero(u["mois"]) > numero(v["mois"]) :
        return True
    elif numero(u["mois"]) == numero(v["mois"]) :
        return u["jour"] > v["jour"]
    else :
        return False

```

Ensuite, lorsque l'on veut programmer la fonction `numero`, on se rend vite compte qu'il est préférable d'avoir un tableau plutôt qu'une fonction :

```

numero={"jan":1, "Jan":1, "janv":1, "Janv":1, "janvier":1, "janvier":1,
        "fev":2, "Fev":2, "fevrier":2, "Fevrier":2, "fév":2, "Fév":2,
        "février":2, "Février":2,
        "mar":3, "Mar":3, "mars":3, "Mars":3,
        ...
}

```

et du coup il faut revoir `apresAnnee`, en mieux :

```

def apresAnnee (u,v) :
    if not (u["mois"] in numero) or not (v["mois"] in numero)
        print("mois mal formé !")
    elif numero[u["mois"]] == numero[v["mois"]] :
        return u["jour"] >= v["jour"]
    else :
        return numero[u["mois"]] > numero[v["mois"]]

```