Numérique et sciences informatiques
Classe de terminale

II Structures de données
A Structures de données linéaires
1. Rappels sur les dictionnaires

I Découvrir les dictionnaire en Python

La notion de « dictionnaire », objet de type **`dict`** en Python, permet de former des variables contenant une succession de données similaire à un « tableau ». La grande différence entre un dictionnaire et un tableau tient au fait que l'accès aux informations contenues dans cette variable se fera :

* *non pas par un indice* (de type **`int`**) indiquant *l'emplacement* de cette information dans la succession ;
* mais par une **clé d'entrée** qui peut être de n'importe quel type *non mutable* (**`str`**, **`int`**, **`float`**, etc.).

Dans un dictionnaire, les données contenues dans la variable *ne seront pas associées à un certain ordre* donc elles n'auront *pas d'indice*.

I.1 Un premier exemple

On veut former une certaine variable `Hero` qui contiendrait avec différentes informations sur le personnage « James Bond ». Ces informations sont les suivantes :

* prénom : James
* nom : Bond
* âge : 38 ans
* pays : Royaume Uni
* métier : agent secret

Nous allons comparer deux solutions :

* la première consiste à créer une variale de type **`list`** ;
* la seconde à créer une variable de type **`dict`**.

a) 1ère solution : former une variable de type `list`

• On peut procèder de la façon suivante :

```
>>> Hero = ["James", "Bond", 38, "Royaume-Uni", "agent secret"]
```

• Pour exploiter cette variable de type **list**, il faut alors *bien se souvenir que* le 1er élément du tableau indique le prénom, le 2ème indique le nom, le troisième indique l'âge, etc.
Si on veut placer ces informations dans des variables distinctes, il faut alors écrire par exemple le code suivant :

```
>>> metierHero = Hero[4]
>>> nomHero = Hero[1]
...
```

b) 2ème solution : former une variable de type `dict`

```
>>> Hero = {"prénom":"James", "nom":"Bond", "age":38, "pays":"Royaume-Uni", "métier":"agent secret"}
```

```
>>> type(Hero)
<class 'dict'>
```

• On peut alors accèder à chaque information contenue dans la variable 'Hero' de type **`dict`** par la **clé** qui lui est associée :
* La clé "`prénom`" est associée à la valeur "`James`" ;
* La clé "`nom`" est associée à la valeur "`Bond`" ;
etc.

```
>>> Hero["prénom"]
'James'
>>> Hero["pays"]
'Royaume-Uni'
```

• On peut modifier la valeur associée à une clé donnée en utilisant une syntaxe très similaire à celle utilisée pour les tableaux :

```
>>> Hero["pays"] = "Grande-Bretagne"
>>> Hero
{'prénom': 'James', 'nom': 'Bond', 'age': 38, 'pays':
'Grande-Bretagne', 'métier': 'agent secret'}
```

• On peut aussi ajouter la nouvelle valeur "`007`" associée à la nouvelle clé "`numCode`" de la façon suivante :

```
>>> Hero['numCode']="007"
>>> Hero
{'prénom': 'James', 'nom': 'Bond', 'age': 38, 'pays':
'Grande-Bretagne', 'métier': 'agent secret', 'numCode':
'007'}
```

I.2 Règles fondamentales pour créer et manipuler les objets de type `dict`

• Pour **créer un dictionnaire vide**, il suffit d'écrire :

```
monDico = {}
```

ou bien

```
monDico = dict()
```

• Pour **créer un dictonnaire non vide** on doit écrire une instruction de la forme suivante :

```
monDico = {cle1 : valeur1 , cle2 : valeur2 ...}
```

• Les éléments importants de la syntaxe :
  * On ouvre et on ferme la valeur attribuée à la variable `monDico` avec des *accolades* : **`{`** et **`}`** ;
  * On associe une valeur à chaque clé en les séparant par le signe **`:`** ;
  * On sépare chaque association *clé* **:** *valeur* par une virgule : **`,`** ;
  * les objets utilisés *pour les valeurs* sont *de n'importe quel type* : **`int`**, **`float`**, **`str`**, **`list`**, **`dict`** etc.
  * les objets utilisés *pour les clés* sont de n'importe quel *type non mutable* : par exemple **`int`**, **`float`**, **`str`**, **`tuple`** *mais pas* **`list`** ni **`dict`.**

• Pour **accéder à la valeur associée à une clé**, on procède d'une façon similaire l'accès à une valeur dans un tableau mais on utilise une clé et non un indice :

```
var = monDico[cle]
```

Attention ! On utilise ici les *crochets* **[** et **]** (comme pour les tableaux) et non les accolades !

• Pour **modifier la valeur associée à une clé**, on procède d'une façon similaire à la modification d'une valeur dans un tableau :

```
monDico[cle] = nouvelleValeur
```

• Pour **ajouter des entrées**, on écrit :

```
monDico[nouvelleCle] = uneValeur
```

• Attention ! Si la clé est déjà présente dans le dictionnaire, cette instruction ne créera pas une nouvelle entrée mais elle aura pour effet de modifier la valeur associée à cette clé
En revanche, il est tout à fait possible que la *valeur associée* à une nouvelle clé soit déjà présente dans le dictionnaire.

Exemple :

```
>>> monTriangle = {"nbCotes" : 3 , "perimetre" : 21 , "aire" : 15 }
>>> monTriangle["nbAngles"] = 3
>>> monTriangle
{'nbCotes': 3, 'perimetre': 21, 'aire': 15, 'nbAngles': 3}
```

• Pour **tester la présence d'une *clé* dans le dictionnaire**, on utilise **`in`**. L'instruction

```
uneCle in monDico
```

renvoie **`True`** ou **`False`**.

Exemple :

```
>>> "aire" in monTriangle
True
>>> "longueur" in monTriangle
False
```

• Attention ! Le mot-clé **`in`** ne permet pas de tester la présence d'une *valeur* dans la variable.

Exemple :

```
>>> 3 in monTriangle
False
```

En effet, `3` n'est pas ici une *clé* mais une *valeur* contenue dans la variable `monTriangle`.

• On peut **supprimer une information** contenue dans la variable avec le mot-clé **`del`** de la façon suivante :

```
del monDico[cle]
```

Exemple :

```
>>> del monTriangle["perimetre"]
>>> monTriangle
{'nbCote': 3, 'aire': 15, 'nbAngles': 3}
```

• On peut **connaître le nombre d'infomations contenues** dans la variable avec la fonction **`len`**`(...)` comme pour les objets de type **`list`**, **`tupple`** et **`str`** :

```
>>> len(Hero)
6
```

I.3 Deux erreurs courantes

a) Pas d'indice pour les dictionnaires

Dans une variable de type **`dict`**, les informations ne sont pas classées dans un ordre bien défini. Elles n'ont donc pas d'indice associé. On ne peut donc pas récupérer une valeur avec un numéro d'indice.

Exemple :

```
>>> nomUsage = Hero[1]
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    nomUsage = Hero[1]
KeyError: 1
```

b) Il faut connaître les clés présentes dans le dictionnaire pour appeler une valeur

Si on essaie d'obtenir une information associée à une clé qui n'existe pas, on obtient un message d'erreur :

```
>>> Hero["numSecu"]
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    Hero["numSecu"]
KeyError: 'numSecu'
```

I.4 Parcourir un dictionnaire

Pour parcourir un dictionnaire on peut utiliser une boucle `for` qui nous permettra d'obtenir toutes les clés une à une. Par exemple, si on veut afficher successivement toutes les clés avec leur valeur associée, on peut écrire :

```
for cle in monDico :
  print(cle, " :", monDico[cle])
```

Exemple :

```
>>> for c in Hero :
  print(c, ':', Hero[c])

prénom : James
nom : Bond
age : 38
pays : Grande-Bretagne
numCode : 007
```

Attention ! Puisque les entrées d'un dictionnaire n'ont pas d'ordre défini, il n'est pas possible de prévoir dans quel ordre les entrées seront affichées. On sait seulement qu'elles le seront toutes.

I.5 Obtenir toutes les clés, toutes les valeurs ou tous les couples (clé , valeur)

• On peut **obtenir un tableau de toutes les *clés* d'un dictionnaire** en utilisant la méthode **`keys()`** associée à tous les objets de type **`dict`**. On pourra écrire une instruction de la forme suivante :

```
tabCles = list(monDico.keys())
```

Exemple :

```
>>> tabCles = list(Hero.keys())
>>> tabCles
['prenom', 'nom', 'age', 'pays', 'numCode']
```

Remarque : L'instruction `tabCle = list(Hero)` donnerait le même résultat.

• De la même façon, on peut **obtenir un tableau de toutes les *valeurs* d'un dictionnaire** en utilisant la méthode **`values()`** associée à tous les objets de type **`dict`**. On pourra écrire une instruction de la forme suivante :

```
tabValeurs = list(monDico.values())
```

Exemple :

```
>>> tabValeurs = list(Hero.values())
>>> tabValeurs
['James', 'Bond', 38, 'Grande Bretagne', '007']
```

Remarque : Si une même valeur apparaît plusieurs fois dans le dictionnaire alors elle apparaîtra dans la variable de type list avec le même nombre d'occurrence.

• Enfin, on **peut obtenir un tableau de tuples à deux places formés de chaque clé et de sa valeur associée** avec la méthode **`items()`** associée à tous les objets de type **`dict`**. On pourra écrire une instruction de la forme suivante :

```
tabEntrees = list(monDico.items())
```

Exemple :

```
>>> tabEntrees = list(Hero.items())
>>> tabEntrees
[('prenom', 'James'), ('nom', 'Bond'), ('age', 38),
('pays', 'Grande Bretagne'), ('numCode', '007')]
```

II Se servir d'une variable dictionnaire

II.1 Créer une fonction qui place le nombre d'occurrence dans un dictionnaire

Imaginons que l'on ait créé un tableau `monTab` contenant 500 nombres entiers choisis aléatoirement entre 1 et 10. On voudrait que la fonction `occurrences(...)` permette de déterminer rapidement le nombre d'occurrences du nombre 1, du nombre 2, etc. jusqu'au nombre 10 dans la variable `monTab`. Le résultat de cette fonction se présenterait sous la forme d'une variable `nbOccu` de type dictionnaire.

Par exemple, si, après avoir créé la fonction `occurrences(...)`, on execute les deux instructions suivantes :

```
monTab = [randint(1, 10) for i in range(500)]
print(occurrences(monTab))
```

On pourrait obtenir par exemple le résultat

```
{7: 58, 10: 48, 1: 41, 3: 43, 8: 49, 4: 51, 2:
51, 6: 61, 9: 42, 5: 56}
```

Ceci signifie que :
* le 7 est apparu 58 fois dans le tableau 'monTab' ;
* le 10 apparaît 48 fois ;
Etc.

On veut donc écrire une fonction `occurrences(tab)` qui :
* prend comme argument une variable `tab` de type **`list`** ;
* qui renvoie un objet de type **`dict`** qui contient le nombre d'occurrences des différentes données qui apparaissent dans le tableau `tab`.

1. Dans le dossier « programmation Python », créer un dossier « Dictionnaire »
Puis dans IDLE, créer un fichier « occurrences.py » dans le dossier « Dictionnaire ».

2. Dans la première ligne du programme

a) écrire l'importation de la fonction `randint(...)` :

```
from random import randint
```

b) écrire l'en-tête de la fonction `occurrences(...)` :

```
def occurrences(tab):
  ...
```

c) Ecrire le corps de la fonction `occurrences(...)` en traduisant en Python l'algorithme suivant :

```
  créer dico comme un dictionnaire vide
  pour e dans tab
       si e est une clé de dico
            incrémenter de 1 la valeur associée à
la clé elem dans dico
       sinon
            créer la clé e dans dico en lui
affectant la valeur 1
       fin si
  fin pour
  renvoyer dico
```

Remarque : Pour traduire cet algorithme en Python, il faut utiliser plusieurs des éléments de syntaxe sur les dictionnaires indiqué dans la partie I.2 du cours.

d) Tester la fonction que vous venez d'écrite avec le programme principal déjà donné plus haut :

```
monTab = [randint(1, 10) for i in range(500)]
print(occurrences(monTab))
```

e) Si tout semble bien fonctionner, testez votre fonction sur le tableau suivant :

```
[3, 4, 4, 5, 2, 3, 5, 3, 4, 1]
```

Vous devez obtenir le résultat suivant

```
{3: 3, 4: 3, 5: 2, 2: 1, 1: 1}
```

f) Constatez que la fonction `occurrences(...)` que vous avez écrite fonctionne même si vous l'appliquer à une chaîne de caractères (de type **`str`**) plutôt qu'à un tableau (de type **`list`**). La fonction `occurrences(...)` appliquée à une variable de type **`str`** détermine le nombre d'occurrences des différents caractères utilisés dans cette variable.

Par exemple si vous exécutez l'instruction suivante

```
maPhrase = "pour qui sont ces serpents qui
sifflent sur vos tetes ?"
print(occurrences(maPhrase))
```

Vous devez obtenir :

```
{'p': 2, 'o': 3, 'u': 4, 'r': 3, ' ': 10, 'q': 2, 'i': 3, 's': 8,
'n': 3, 't': 5, 'c': 1, 'e': 6, 'f': 2, 'l': 1, 'v': 1, '?': 1}
```