Mémento Python

Traduit et adapté de https://www.brianheinold.net/python/Python_Quick_Reference_Guide_Heinold.pdf par E. Batard
selon la licence originale https://creativecommons.org/licenses/by-nc-sa/3.0/deed.fr

1 Généralités

- **Commentaires** — Utilisez # pour mettre en commentaire jusqu'à la fin de la ligne.
- **Guillemets et apostrophes** — Utilisez une paire de guillemets (ou d'apostrophes) quand vous voulez du texte et ne les utilisez pas si vous voulez calculer un résultat numérique. La première ligne ci-dessous affiche le texte 2+2 et la suivante calcule 2 + 2 et affiche donc 4.

```
print("2 + 2")  # affiche 2 + 2
print(2 + 2)    # affiche 4
```

- **Import** — Par convention, les clauses d'importation s'écrivent au début de vos programmes.
- **Conversion : d'une chaîne de caractères vers un nombre -- fonctions *int()* et *float()***

```
nombreEntier = int(s)     # convertit la chaîne s en entier
nombreReel = float(s)     # convertit la chaîne s en nombre réel (avec décimales)
```

- **Conversion** : **d'un nombre vers une chaîne de caractères -- fonctions *str()***

  Utilisez la fonction *str()* pour convertir un nombre (entier ou pas) en chaîne de caractères. Par exemple, grâce à une conversion en chaîne, il est facile d'accéder aux différents chiffres qui composent un nombre :

```
nombre = 3141592654
s = str(nombre)
print("Le premier chiffre du nombre", nombre, "est", s[0])
```

  On pourrait aussi utiliser ce genre de conversion pour insérer un nombre dans le nom de quelque chose, comme le nom d'un fichier. Ainsi :

```
x = 3
s = "fichierVersion" + str(x) + ".txt" # s contient fichierVersion3.txt
```

  Cela pourrait servir dans un programme qui génère une série de fichiers ayant des noms similaires.

- **Demande d'une valeur** — Utilisez directement la fonction *input()* pour obtenir un texte et utilisez-la avec une fonction de conversion *int()* ou *float()* pour obtenir un nombre, respectivement entier ou réel.

```
nom = input("Quel est votre nom ?")
nombreEntier = int(input("Tapez un nombre entier :"))
```

- **Les arguments optionnels de print() : séparateur d'éléments et séparateur de lignes**
  - sep —utilisé pour définir ou supprimer ce qui sépare les éléments affichés par *une* instruction print(). Par défaut c'est un espace.

```
print(1, 2, 3, 4, 5, sep='')      # sep est une chaîne vide, donc 12345 est affiché
print(1, 2, 3, 4, 5, sep='/')     # sep est une barre oblique, donc 1/2/3/4/5 est affiché
```

  - end — utilisé pour définir comment print() passe, ou pas, à la ligne suivante. Par défaut print() passe à la ligne

```
for i in range(10):
    print(i, end="")
```

  Cette boucle affiche 0123456789 sur une seule ligne. Si au lieu de end="" il y avait end=" " (un espace), on aurait eu 0 1 2 3 4 5 6 7 8 9 toujours sur une seule ligne mais séparé par des espaces.

2 Opérations arithmétiques et mathématiques

Opérateur	Opération arithmétique	Exemple
+, -, *, /	Respectivement : addition, soustraction, multiplication	5 + 3 donne 8 5 – 3 donne 2 5 * 3 donne 15
/	Division réelle (avec décimales)	5 / 3 donne 1.6666666666666667
**	Elévation à la puissance	5**3 donne 125
%	Modulo (reste de la division entière)	19 % 5 donne 4
//	Division entière (euclidienne)	19 // 5 donne 3

- **Compléments**
  - Une formule comme 3*x* + 5 doit s'écrire avec le symbole de la multiplication, comme ceci :

```
y = 3*x + 5
```

  - L'ordre des opérations est important. Utilisez des parenthèses si nécessaire. Par exemple, pour calculer la moyenne de x, y et z, c'est :

```
moyenne = (x + y + z) / 3
```

  - Pour utiliser des fonctions mathématiques standard, il faut les importer du module math. Par exemple :

```
from math import cos, pi
print("Le cosinus de pi vaut", cos(pi))  # affiche -1
```

  - Deux fonctions n'ont pas besoin d'être importées : la valeur absolue *abs()* et l'arrondi *round()* :

```
print("La valeur absolue de -3 est", abs(-3))              # affiche 3
print("4/3 arrondi à deux décimales vaut ", round(4/3, 2)) # affiche 1.33
```

- **Nombres aléatoires**
  - **Tirer au hasard un nombre entier entre deux bornes** — Pour utiliser des nombres tirés au hasard (ou presque), il faut importer le module random et utiliser la fonction *randint()* ainsi :

```
from random import randint
x = randint(1, 5) # affecte à x un nombre entre 1 et 5, bornes incluses
```

    Voir aussi le paragraphe Tirer au hasard un ou plusieurs éléments d'une liste, plus loin.

3 L'instruction *If* et ses variantes

- **Instruction *if* courantes**

```
if x == 3:                          # si x vaut 3
if x != 3:                          # si x vaut autre chose que 3
if x >= 1 and x <= 5:               # si x est entre 1 et 5 (bornes comprises)
                                    # on peut aussi écrire en Python
                                    #  if 1 <= x <= 5:
if x == 1 or x == 2:                # si x vaut 1 ou 2
if nom == "elvis":                  # pour comparer des chaînes de caractères
if (x == 3 or x == 5) and y == 2:   # les parenthèses permettent de simplifier des
                                    # formules complexes
```

- ***if* avec *else***

```
if nombreATester == nombreATrouver:
    print("Bravo, vous avez trouvé !")
else:
    print("C'est raté.")
```

- ***if* avec *elif* et éventuellement *else*** est utile quand il faut tester une possibilité parmi plusieurs.

```
if saisie == "oui":
    print("Le montant de votre note est 50.00 CHF")
elif saisie == "non":
    print("Le montant de votre note est 45.00 CHF")
else:
    print("Saisie non reconnue")
```

4 Les boucles *For* sur des nombres (entiers)

- **Généralités**
  - Pour afficher coucou 50 fois :

```
for i in range(50):
    print("coucou")
```

  - L'*indentation* sert à indiquer les instructions qui devront être répétées. La boucle ci-dessous affiche alternativement oui et non 50 fois. Et finit par afficher Je ne sais pas une seule fois.

```
for i in range(50):
    print("oui")
    print("non")
print("Je ne sais pas")
```

  - A tout moment la variable *i* dans une boucle *for* permet de savoir à quelle étape on se trouve. Dans les exemples précédents, cette variable vaut initialement *0* et augmente de 1 en 1 jusqu'à atteindre un de moins que le paramètre de range. L'exemple ci-dessous affiche la valeur de *i* à chaque *itération* (= chaque tour de la boucle), ce qui donnera les nombres entiers entre 0 et 49. (Vous êtes libre de choisir le nom de la variable, *i* est juste un exemple ici.)

```
for i in range(50):
    print(i)
```

- **La fonction *range*** — Voici quelques exemples d'utilisation de *range* (mot anglais qui signifie intervalle)

Appels	Valeurs obtenues dans un *for*
range(10)	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
range(5,10)	5, 6, 7, 8, 9
range(1,10,2)	1, 3, 5, 7, 9
range(10,0,-1)	10, 9, 8, 7, 6, 5, 4, 3, 2, 1

Attention : Le point délicat est que la dernière valeur d'un *range* est toujours un de moins que la valeur finale indiquée. Par exemple, range(1,10) s'arrête à 9.

5 Les chaînes de caractères

- **Les bases**

Instruction ou appel	Description
s = s+"!!"	Ajoute !! à la fin de la chaîne s
s = "a"*10	Affecte 10 fois a à s (équivalent à s="aaaaaaaaaa")
s.count(" !!")	Le nombre de fois qu'une chaîne (ici !!) apparaît dans s
s = s.upper()	Passe toutes les lettres en majuscules (y compris accentuées)
s = s.replace("Salut","Bonjour")	Remplace chaque Salut par Bonjour dans s
If "!!" in s:	Teste s'il y a au moins une fois !! dans s
s.index("a")	Position du premier a dans s Exception ValueError s'il n'y en a pas
If s.isalpha():	Teste si chaque caractère de s est une lettre (éventuellement accentuée)
If s.isdigit():	Teste si chaque caractère de s est un chiffre
len(s)	Renvoie la taille de s = nombre de caractères

- **Indices de chaîne et *slices*** — Supposons que s contienne "abcdefghij"

Expression	Résultat	Description
s[0]	a	Le premier caractère de s
s[1]	b	Le second caractère de s
s[-1] s[len(s)-1]	j	Le dernier caractère de s **Attention** : la formule avec len() peut donner des indices négatifs
s[-2] s[len(s)-2]	i	L'avant-dernier caractère de s **Attention** : la formule avec len() peut donner des indices négatifs
s[:3]	abc	Les trois premiers caractères
s[-3:]	hij	Les trois derniers caractères
s[2:5]	cde	Les caractères en position 2, 3 et 4
s[5:]	fghij	Les caractères de la position 5 jusqu'au dernier
s[:]	abcdefghij	Une copie de toute la chaîne
s[1:7:2]	bdf	Les caractères en position 1, 3 et 5, c'est-à-dire un sur deux de la position 1 à la position 6
s[::-1]	jihgfedcba	L'inverse de s

Attention : Les *slices* (mot anglais qui signifie tranches) sont comme **range** et n'incluent *jamais* la dernière position.

- **Utilisation des slices avec la fonction *index()***

Expression	Description
s[s.index(" ")+1]	Le caractère après le premier espace
s[:s.index(" ")]	Tous les caractères avant le premier espace
s[s.index(" ")+1:]	Tous les caractères après le premier espace

- **Quelques exemples utiles**

Exemple	Description
s[0].upper()+s[1:]	Passe l'initiale en majuscule sans changer le reste
s=s.replace("$","")	Elimine tous les symboles $ dans s
int(s[-4:])	Convertit les 4 derniers caractères de s en un nombre
If s[-1].isalpha():	Teste si le dernier caractère est une lettre

- **Parcours des caractères d'une chaine : première méthode**

```
for i in range(len(s)):
    print(s[i])
```

La variable i désigne la position courante dans s ; s[i] est le caractère situé à cette position.

Exemple *:* Affichage alterné des caractères de deux chaînes, s et t, de même longueur.

```
for i in range(len(s)):
    print(s[i], t[i], end = "")
```

- **Parcours des caractères d'une chaine : deuxième méthode**

```
for c in s:
    print(c)
```

Dans ce cas la variable de contrôle de la boucle, c, prend successivement comme valeur chacun des caractères de la chaîne s.

Exemple *:* Pour compter combien il y a de lettres dans s

```
nbLettres = 0
for c in s:
  if c.isalpha(): #les fonctions isalpha() et isdigit() s'appliquent aussi aux caractères uniques
     nbLettres = nbLettres + 1
print("il y a", nbLettres, "lettres dans", s)
```

Attention de ne pas mélanger les deux types de boucles. Si vous avez une boucle du style for i in s: et que vous essayez de faire quelque chose avec s[i], ça ne pourra pas marcher. Le second type de boucle est simplement un raccourci du premier type de boucle et est bien pratique dans certains cas.

- **Les caractères spéciaux**

Caractère	Effet
'\n'	Caractère de passage à la ligne
'\t'	Caractère de tabulation
"\'" et '\"'	Caractères apostrophe et guillemet : pratique si vous devez mettre des guillemets ou des apostrophes dans une chaîne
'\\'	Caractères antislash ou barre oblique inverse lui-même (cf remarque)

Remarque : vous pouvez aussi préfixer une chaîne littérale (c'est-à-dire écrite entre guillemets ou apostrophes) dans votre programme par r (pour *raw*) de façon à ne pas avoir à doubler le \, par exemple dans une spécification de fichier.

- **Formatage des chaînes**

Le formatage de chaînes est utilisé pour rendre l'affichage plus lisible. Voici quelques exemples :

Code de formatage	Description
"{:.2f}".format(*num*)	Contraint l'affichage de *num* à toujours avoir 2 chiffres après la virgule
"{:10.2f}".format(*num*)	*num* à 2 décimales aussi mais aligné
"{:10d}".format(*num*)	Utilisez d pour les entiers et f pour les nombres réels
"{:10s}".format(*nom*)	Utilisez s pour les chaînes

Remarque : dans les exemples ci-dessus, 10 représente la taille maximale d'un nombre ou une chaîne, mais vous pourriez le remplacer par la taille du plus grand nombre ou chaîne que vous voulez afficher. Quand vous utilisez 10 (ou un autre nombre) Python réserve 10 caractères pour afficher le nombre ou la chaîne et les caractères non utilisés seront remplis par des espaces, ce qui aura pour effet de justifier (=aligner) à gauche les chaînes et de justifier à droite les nombres.

6 Les listes

- **Ce qui fonctionne de la même façon pour les listes et les chaînes**
  - Les indices and les *slices* (entre crochets)
  - L'opérateur **in** pour vérifier si quelque chose est présent dans une liste ou une chaîne
  - Les méthodes count() et index()
  - + et * servent aussi pour concaténer (=ajouter) et répéter des listes
  - Les deux types de boucle **for**

- **Les fonctions utiles sur les listes** (elles prennent une liste en *paramètre*)

Fonction	Résultat
max(*li*)	Renvoie le plus grand élément dans la liste *li*
min(*li*)	Renvoie le plus petit élément dans la liste *li*
sum(*li*)	Renvoie la somme des éléments de la liste *li*
len(*li*)	Renvoie le nombre d'éléments dans la liste *li*

- **Quelques méthodes utiles sur les liste** (notation *pointée*) Si la liste s'appelle li

Méthode	Description
li.append(*x*)	Ajoute x à la fin de la liste
li.sort()	Trie la liste *li* dans l'ordre croissant (paramètre reverse=true pour l'ordre décroissant)
li.count(*x*)	Renvoie le nombre de fois que *x* apparaît dans la liste
li.index(*x*)	Renvoie la position de la première *occurrence* (= apparition) of *x* dans ili
li.reverse()	Inverse la liste
li.remove(*x*)	Supprime la première occurrence de *x* dans la liste *li*
li.pop(*p*)	Supprime l'élément en position *p* et renvoie sa valeur
li.insert(*p*,*x*)	Insère *x* à la position *p* dans la liste *li*

- **Pour modifier un élément existant d'une liste**

  Le code ci-dessous modifie le premier élément de la liste maliste en 99.

```
maliste[0] = 99
```

- **Pour construire une liste élément par élément**

  Voici un programme qui construit une liste de 100 nombres aléatoires compris entre 1 et 20 :

```
from random import randint

maliste = []
for i in range(100):
    maliste.append(randint(1,20)
```

  Bon à savoir : La technique générale pour construire une liste est de démarrer avec une liste vide et d'ajouter (avec *append()*) les éléments un par un.

- **Utiliser input pour obtenir une liste**

  Si vous voulez demander à l'utilisateur de saisir une liste *li*, vous pouvez utiliser ceci :

```
li = eval(input("Tapez une liste entre crochets")) #sans eval() li contiendra une chaîne
```

  L'utilisateur doit saisir la liste exactement comme s'il l'écrivait dans un programme Python, en utilisant des crochets et en séparant les éléments par des virgules.

- **Tirer au hasard un ou plusieurs élément(s) d'une liste**

  Ces fonctions sont très pratiques. Pour les utiliser, il faut les importer au préalable du module random.

Appel	Résultat
choice(*li*)	Renvoie un élément de *l*i tiré au hasard
choices(*li*, k = *v)*	Renvoie une liste de *v* éléments de *li* tirés au hasard **Attention** : le même élément peut apparaître plusieurs fois **Attention** : il *faut* écrire k= avant le nombre d'éléments voulus
sample(*li*, *n*)	Renvoie une liste de *n* éléments de *li* tirés au hasard **Attention** : un même élément ne peut apparaître qu'une seule fois **Attention** : *n* doit être inférieur à la longueur de *li*
shuffle(*li*)	Met les éléments de *li* dans un ordre au hasard

- **The méthode *split()***

  La méthode *split()* est une méthode très utile pour découper une chaîne. Par exemple :

```
"s=abc de fgh i jklmn"
listeRésultat = s.split()
print(listeRésultat [0], listeRésultat [-1])
```

  Ce programme affiche abc et jklmn.

  Par défaut, *split()* effectue un découpage sur les espaces au sens large : un ou plusieurs espaces, tabulations et passage à la ligne. Pour découper selon d'autres caractères, par exemple des barres obliques (/), écrivez quelque chose comme ceci:

```
s = "31/5/2013"
listeRésultat = s.split("/")
jour = int(listeRésultat[0])
```

  L'appel à int() de la dernière ligne est nécessaire si on veut faire des calculs avec la variable jour.

7 Les fonctions et procédures

Les fonctions et procédures servent à décomposer les gros programmes en parties plus faciles à comprendre, à écrire et à faire évoluer. De plus, les fonctions servent à donner un nom à des actions (on parle de *procédures* dans ce cas) ou des calculs qui sont faits à plusieurs reprises. Ainsi si vous devez modifier le comportement d'une fonction, vous n'avez qu'à intervenir à un seul endroit et pas à chaque endroit où on a besoin de l'action ou du calcul correspondant, c'est-à-dire à chaque endroit où on appelle la fonction.

Voici une petite procédure :

```
def dessiner_rectangle():
    print("**********")
    print("**********")
    print("**********")
```

Cette procédure pourrait être écrite au début de votre programme.

On peut aussi ajouter des *paramètres* pour qu'une procédure ou fonction puisse s'adapter à différents cas de figure. Pour avoir des rectangles de différentes tailles, il faut permettre à l'utilisateur de votre procédure, *la procédure* ou *fonction appelante*, d'indiquer la taille voulue. Pour cela, on peut modifier la procédure comme ceci (on pourrait utiliser des boucles aussi) :

```
def dessiner_rectangle(largeur):
    print("*" * largeur)
    print("*" * largeur)
    print("*" * largeur)
```

Voici comment on peut utiliser, ou *appeler*, chacune des fonctions que nous venons de voir.

```
dessiner_rectangle()      # appel de la première fonction
dessiner_rectangle (10)   # appel de la version améliorée avec un paramètre
```

Les procédures servent à effectuer des actions (comme des affichages pour dessiner_rectangle()). Les fonctions, quant à elles, renvoient une valeur. Python ne fait pas de différence syntaxique entre procédures et fonctions et appelle tout des *fonctions*.

Voici un exemple de « vraie » fonction. Elle prend en paramètre le montant d'une addition et le pourcentage pour le service que lui fournit le code qui l'appelle et renvoie le montant total à payer grâce à **return** :

```
def calculer_addition(montant, service):
    total_a_payer = montant + montant * service/100
    return total_a_payer
```

Et voici deux façons d'appeler cette fonction :

```
print(calculer_addition (24.99, 20))
x = calculer_addition(24.99, 20) / 2        # on partage l'addition !
```

Dans ces deux appels, on constate l'effet de **return**. Il permet de récupérer le résultat du calcul dans l'instruction appelante et d'utiliser ce résultat pour faire autre chose (ici l'afficher ou bien le diviser par deux).

8 Les boucles *While*

Les boucles **for** servent à répéter des actions ou des calculs quand on (vous ou Python) connaît à l'avance le nombre de fois que ces actions et calculs doivent être répétés. Quand ce nombre de fois est inconnu car il dépend d'une condition variable, il faut utiliser une boucle **while**.
Habituellement avec une boucle **while** on continue à exécuter la boucle jusqu'à ce que quelque chose impose d'arrêter.

Voici un exemple qui demande à un utilisateur de proposer un nombre jusqu'à ce qu'il trouve la bonne réponse :

```
proposition = 0 # pour démarrer la boucle
while proposition != 24: # le nombre à trouver est 24
    proposition = int (input("Que vaut 6 fois 4 ? "))
    if proposition != 24:
        print("Ce n'est pas ", proposition, " ! Essayez encore...")
print("Bravo, c'est tout bon.")
```

Voici un autre exemple qui parcourt une liste maliste jusqu'à trouver un nombre supérieur ou égal à 5 (en supposant qu'il y en ait forcément un).

```
pos = 0
while maliste[pos] <= 5:
    pos = pos + 1 # ou pos += 1
```

Quand on parcourt une liste, ou une chaîne de caractères, avec une boucle **while**, il faut gérer l'évolution des indices nous-même (alors qu'une boucle **for** s'en occupe pour nous). Il nous faut donc une variable comme pos (pour position) qu'il faut initialiser à 0 avant l'instruction **while** et qu'il ne faut surtout pas oublier d'incrémenter, par exemple avec pos = pos + 1 dans la boucle.

9 Quelques techniques utiles (et pas seulement en Python !)

Décompte Un problème vraiment courant en programmation est de compter combien de fois quelque chose arrive. Le programme qui suit demande à l'utilisateur 10 nombres et compte combien de fois l'utilisateur saisit 7.

```
nb7 = 0
for i in range(10):
    nombre = int(input("Tapez un nombre "))
    if nombre == 7:
            nb7 = nb7 + 1
print("Vous avez saisi", nb7, "fois le nombre 7")
```

Dans un décompte, on commence pratiquement toujours à compter à partir de 0 (ici `nb7 = 0`) et à un moment ou un autre, on incrémente le nombre. Ici `nb7 = nb7 + 1`, mais uniquement si le nombre saisi vaut 7 car c'est ce qu'on compte.

Sommation Une technique proche du décompte revient à additionner diverses valeurs.
Ce programme additionne tous les nombres de 1 à 100 :

```
total = 0
for i in range(1, 101):
    total = total + i
print("1+2+...+100 vaut", total)
```

Trouver le maximum et le minimum Si vous avez une liste li et que vous voulez connaître le max ou le min, utilisez `max(li)` ou `min(li)`. Mais si ce que vous avez à faire est un peu plus compliqué, utilisez la technique ci-dessous. Dans cet exemple, on cherche le mot le plus long dans une liste de mots :

```
mot_le_plus_long = li[0] # on suppose que le mot le plus long est le premier
for mot in li:
        if len(mot) > len(mot_le_plus_long): # utilisez < au lieu de > pour avoir le mot le plus court
            mot_le_plus_long = mot # on a trouvé plus long que celui qu'on supposait être le plus long
```

On peut aussi initialiser la variable qui contient le max (ou min) supposé avec une valeur forcément « fausse ». Par exemple, si on initialise `mot_le_plus_long` avec la chaîne vide `""`, on est sûr qu'au moins un mot a davantage de caractères (ou alors tous les mots sont des chaînes vides et l'hypothèse est juste !).

Echanger les valeurs de deux variables Pour échanger les valeurs de `x` et `y`, il faut utiliser une variable intermédiaire comme ceci :

```
copie_de_x = x      # on recopie la valeur de x dans copie_de_x
x = y               # on peut alors écraser la valeur de x avec celle de y
y = copie_de_x      # maintenant que la valeur de y est dans x on peut écraser la valeur de y avec l'ancienne de x
```

En Python, vous pouvez utiliser ce raccourci :

```
x, y = y, x
```

Utiliser des listes pour réduire le code Il arrive souvent qu'on soit amené à écrire pratiquement la même chose un grand nombre de fois, comme dans l'exemple ci-dessous qui récupère le nom d'un mois à partir de son numéro.

```
if mois == 1:
    nomMois = "janvier"
elif mois == 2:
    nomMois = "février"
# suivi de 10 autres conditions pratiquement identiques...
```

Il est possible de raccourcir cette écriture façon significative en utilisant une liste des noms de mois `listeNomMois` :

```
listeNomMois = ["janvier", "février", "mars", "avril", "mai", "juin","juillet", "août", "septembre", "octobre", "novembre", "décembre"]
nomMois = listeNomMois[mois-1] # -1 car les indices commencent à 0
```

Si vous voyez que vous faites souvent du copier-coller de code sans faire de grands changements, pensez à définir une procédure, une fonction ou à utiliser une liste. Ça peut simplifier les choses.

10 Les dictionnaires

Un dictionnaire permet d'associer deux séries de données ensemble. Par exemple :

```
d = {"jan":31, "fév":28, "mar":31, "avr":30}  # nombre de jours des 4 premiers mois
```

Une fois le dictionnaire d défini ainsi, pour obtenir le nombre de jours en janvier, il suffit d'écrire d["jan"].

Un dictionnaire se comporte comme une liste dont les indices peuvent être autre chose que des nombres, comme des chaînes dans l'exemple. Ici les noms des mois sont les *clés* du dictionnaire et les nombres de jours sont ses *valeurs* toujours associées à une clé. Généralement on donne la clé pour accéder à la valeur associée.

Quelques opérations courantes sur les dictionnaires :

Expression	Description
d[k]	Récupère la valeur associée à la clé k
d[k] = 29	Modifie ou crée la valeur d associée à la clé k
if k in d:	Teste si k est une clé dans le dictionnaire d
list(d)	Renvoie une liste de toutes les clés dans d (raccourci pour list(d.keys())
list(d.values())	Renvoie une liste de toutes les valeurs dans d
list(d.items())	Renvoie une liste de toutes les paires (clé, valeur) dans d

Il y a différentes façons de parcourir les dictionnaires.

Par exemple, pour avoir les clés et les valeurs (une paire par ligne)

```
for k in d:
    print("la clé", k, "est associée à", d[k])
```

Pour avoir la liste des mois à 31 jours dans le dictionnaire d, on peut écrire :

```
résultat = []
for x in d:
  if d[x] == 31:
     résultat.append(x)
```

Les dictionnaires sont pratiques quand vous avez deux listes qui doivent être synchronisées. Par exemple, un dictionnaire serait utile si vous aviez un à représenter un questionnaire : les clés pourraient être les questions du quiz et les valeurs seraient les réponses aux questions. Dans le cas du *Scrabble,* vous pourriez avoir les lettres comme clés et, comme valeur, le nombre de points que rapporte la lettre.

11 Les listes à deux dimensions (voir aussi les tableaux du module *numpy*)

Voici comment créer une liste de 3 lignes sur 3 colonnes contenant des zéros :

```
exemple2D = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

Pour avoir accès aux différents éléments, il faut utiliser deux indices, un pour la ligne, l'autre pour la colonne (dans cet ordre !). Pour accéder à l'élément en ligne lig et colonne col, écrivez exemple2D[lig][col].

Quand on travaille sur des listes à deux dimensions, on a souvent recours à deux boucles for, une pour parcourir les lignes et l'autre pour parcourir les colonnes. Le programme qui suit affiche une liste de 10 × 5 :

```
for lig in range(10): # 10 lignes allant de 0 à 9
    for lig in range(5): # 5 colonnes allant de 0 à 4
        print(exemple2D[lig][col], end=" ") # affiche l'élément en ligne lig et en colonne col sur la même ligne
    print() # pour passer à la ligne
```

HORS CHAMP ALP

12 La compréhension de listes ou de dictionnaires

La *compréhension* de liste ou de dictionnaire (ou de *set* ou de *tuple*) permet des notations très compactes. Ainsi au lieu de la boucle donnée ci-dessus pour avoir la liste des mois à 31 jours dans le dictionnaire `d` du point 10., on peut écrire :

```
résultat = [m for m in d if d[m] == 31] # exemple de compréhension de liste
```

Et pour avoir un dictionnaire où ne figurent que les mois n'ayant pas 31 jours parmi ceux de du dictionnaire `d` du point 10., on peut écrire :

```
r = {m:d[m] for m in d if d[m] < 31} # exemple de compréhension de dictionnaire
```

13 Working with text files

- **Reading from files** — The line below opens a text file and reads its lines into a list:

```
L = [line.strip() for line in open('somefile.txt')]
```

1. Suppose each line of the file is a separate word. The following will print all the words that end with the letter `q`:

```
for word in L:
    if word[-1] == 'q': print(word)
```

2. Suppose each line of the file contains a single integer. The following will add 10 to each integer and print the results:

```
for line in L:
    print(int(line) + 10)
```

3. Suppose there are multiple things on each line; maybe a typical line looks like this:

   Luigi, 3.45, 86

   The entries are a name, GPA, and number of credits, separated by commas. The line below prints the names of the students that have a GPA over 3 and at least 60 credits: e

```
for line in L:
    M = line.split(",") if float(M[1]) > 3 and int(M[2]) >= 60:
        print(M[0])
```

- **Writing to text files** — Three things to do: open the file for writing, use the print statement with the optional `file` argument to write to the file, and close the file.

```
my_file = open("filename.txt", "w") print("This will show up in the file but not onscreen", file=my_file)
my_file.close()
```

The `"w"` is used to indicate that we will be writing to the file. This code will overwrite whatever is in the file if it already exists. Use an `"a"` in place of `"w"` to append to the file.