Dictionnaires

Dictionnaires : les clefs peuvent être n'importe quel type non mutable :

- un tuple dont les éléments sont non mutables.
- un namedtuple (dont les éléments sont non mutables).
- un frozenset
- pas une liste.
- mais pas un tuple dont certains éléments sont une liste.

Un dictionnaire peut avoir une clef à None (mais bien sûr, une seule !).
Manipulations élémentaires :

- `d = {}` : initialisation à vide.
- on peut aussi faire `d = dict()`.
- `d = {'toto': 1, 'titi': 2}` : initialisation.
- on peut aussi définir un dictionnaire avec une liste de paires (tuples à 2 valeurs) : `d = dict([('a', 1), ('b', 2)])`
- `d['toto'] = 5` : affectation d'une valeur.
- les clefs d'un dictionnaire peuvent être des tuples (de constantes) ! (mais pas des listes)
- `del d['toto']` : destruction d'une clef.
- `d.keys()` : renvoie un itérateur sur les clefs. Si on veut la liste des clefs, faire `list(d.keys())`
- `'toto' in d` : test de l'existence d'une clef.
- `key in d` ou `key not in d` : également pour tester la présence d'une clef.
- `len(d)` : renvoie le nombre d'items dans le dictionnaire.

Pour boucler sur les clefs d'un dictionnaire :

```
for x in d:
  print(x)
```

Pour boucler sur un dictionnaire en récupérant en même temps les clefs et les valeurs :

```
for k, v in d.items():
  print(repr(k) + ' : ' + repr(v))
```

Autres fonctions sur les dictionnaires :

- `d.clear()` : efface tout.
- `d.copy()` : shallow copie
- `d.get(key)` : renvoie la valeur si clef présente, sinon None

- `d.get(key, defaultVal)` : renvoie la valeur si clef présente, sinon defaultVal
- `d.items()` : renvoie un itérateur sur le dictionniare renvoyant les paires (key, value).
- `d.pop(key)` : enlève la clef et renvoie la valeur (KeyError si clef n'existe pas).
- `d.pop(key, val)` : enlève la clef si elle existe et renvoie la valeur. Renvoie val si clef n'existe pas (sans lever une exception).
- `d.popitem()` : renvoie une paire (key, value) au hasard et la retire du dictionnaire.
- `d.values()` : renvoie un itérateur sur les valeurs.
- `d.setdefault(key, defaultVal)` : si la clef existe, renvoie sa valeur, si la clef n'existe pas, insère la clef avec la valeur defaultVal et renvoie sa valeur.

`d2 = dict(d)` : fait une copie indépendante du dictionnaire.
Dictionnaire de compréhension : `d = {x: 2 * x for x in range(4)}` donne le dictionnaire `{0: 0, 1: 2, 2: 4, 3: 6}`.
Inversion des clefs/valeurs d'un dictionnaire :

- quand les valeurs sont uniques :

```
invD = {value:key for key, value in d.items()}
```

- quand les valeurs ne sont pas uniques :

```
invD = {}
for k, v in d.items():
    invD.setdefault(v, []).append(k)
```

Pour supprimer les doublons d'une liste sans changer l'ordre : `collections.OrderedDict.fromkeys(myList).keys()`

Fonctions

Définition d'une fonction :

```
def myFunc(n):
       print(n)
       return 1
```

Documentation d'une fonction : mettre la chaîne de documentation en première ligne du corps :

```
def myFunc(n):
    """Chaine de documentation."""
    print(n)

```

- La première ligne doit commencer par une majuscule et terminer par un point, et être la description générale de la fonction.
- on peut ensuite ajouter des détails e n sautant une ligne, et alors les triples quotes sont vraiment indispensables !

```
def myFunc(n):
    """Chaine de documentation.

       et des précisions sur la documentation,
       avec encore une troisième ligne"""

    print(n)
```

Les paramètres sont passés par valeur (mais c'est la référence qui est passée par valeur quand c'est une variable de type liste ou dictionnaire ou tuple).
Par défaut, les variables d'une fonction sont locales :

- on ne peut pas accéder à la valeur d'une variable locale avant de l'avoir initialisée (n'est pas initialisée à None par défaut !).
- on ne peut pas modifier une variable globale, mais on peut par contre la lire (dans une fonction, la variable est cherchée successivement dans la table des symboles locale, puis globale, puis built-in)

Valeur de retour d'une fonction :

- une fonction peut renvoyer un argument avec return.
- une fonction qui n'a pas de return ou un return sans valeur renvoie None.
- une fonction peut renvoyer un tuple qui est récupéré dans plusieurs variables à condition que le nombre de variables matche le nombre de valeurs du tuple :

```
def myFunc():
    return (1, 'a')

(x, y) = myFunc() # valide
(x, y, z) = myFunc() # invalide
```

- convention lorsqu'on veut récupérer seulement certains arguments de retour d'une fonction qui renvoie une liste ou un tuple : on utilise '_' pour ceux qu'on ne veut pas récupérer (mais c'est seulement une convention). Par exemple : `x, _, y, _ = myFunction()`

On peut definir une fonction avec des valeurs par défaut :

- `def myFunc(a, b = 4):`
- on peut alors appeler myFunc(0, 1) ou myFunc(0)

Attention aux valeurs par défaut mutables !

- la valeur par défaut n'est évaluée qu'une seule fois !

```
def f(x, y = []):
    y.append('a')
    print(y)
f(1); f(2); f(3)
```

imprime

```
['a']
['a', 'a']
['a', 'a', 'a']
```

- si on ne veut pas ce comportement, mettre une valeur par défaut à None, et fixer la valeur dans la fonction si la valeur vaut None.
- en bref, éviter les objets mutables comme valeurs par défaut dans la définition d'une fonction !

On peut appeler une fonction avec des arguments précisés par nom :

```
def myFunc(a, b = 4):
...
myFunc(a = 1, b = 5)
myFunc(a = 2)
```

Les arguments par défaut sont évalués dans le scope de la définition :

```
a = 3
def myFunc(x = a):
    print(x)
a = 4
myFunc()
```

va imprimer 3 et non 4 !
Avec la formulation **, l'argument recoit le dictionnaire de tous les arguments précisés par nom :

```
def myFunc(**args):
        for kw in args.keys():
                print(kw, ':', args[kw])
```

Avec la formulation *, l'argument reçoit le tuple de tous les arguments :

```
def myFunc(*args):
        for kw in args:
                print(kw)
```

On peut mélanger tout cela :

```
def myFunc(a, *b, **c):
    print(a)
    print(b)
    print(c)

myFunc(4, 5, 7, x = 4, y = 8)
```

imprime :

```
4
(5, 7)
{'titi': 8, 'toto': 4}
```

Unpacking :

- si fonction définie par :

```
def myFunc(**args):
   for kw in args.keys():
          print(kw, ':', args[kw])
```

  on peut l'appeler avec `myFunc(**{'a' = 3, 'b' = 4})`.

- si fonction définie par :

```
def myFunc(a, b):
   return a + b
```

  on peut aussi l'appeler avec `myFunc(**{'a' = 3, 'b' = 4})` ou également `myFunc(*[3, 4])` ou même `myFunc(*(3, 4))`.

lambda function (fonction anonyme) : lambda x, y: x * y

Variable statique dans une fonction, qui garde la mémoire entre 2 appels :

- on peut définir une variable statique dans une fonction, par exemple :

```
def myFunc(a):
    if getattr(myFunc, 'myVar', None) is None:
        # initialisation au premier appel
        myFunc.myVar = 0
    myFunc.myVar += a
    print(myFunc.myVar)

myFunc(1) # affiche 1
myFunc(2) # affiche 3
```

Une fonction peut être stockée dans une variable avant d'être utilisée :

```
def myFunc(x):
    return x + 1

f = myFunc
print(f(3)) # donne 4
```

Si myFunc est une fonction définie :

- `myFunc.__name__` renvoie son nom : `myFunc`
- si on met cette fonction dans une autre variable : `myFunc2 = myFunc`, `myFunc2.__name__` renvoie `myFunc`

On peut définir une fonction à l'intérieur d'une fonction, pour que la fonction englobante retourne une fonction :

```
def buildMyFunc(a):
    def myFunc(x):
        return a * x
    return myFunc

f = buildMyFunc(2)
f(3) # renvoie 6
```

Test si un attribut est une fonction (ou méthode) : `callable(myMeth)` : renvoie True si myMeth est une fonction ou méthode, False sinon.
Appel des attributs par nom :

- `hasattr(myObj, myField)` renvoie True si l'objet a le champ défini, False sinon.
- `callable(myMeth)` : renvoie True si myMeth est une fonction ou méthode.
- appel d'une méthode par son nom sur un objet : `l = [6, 3, 5, 1, 9]; getattr(l, 'sort')()` : tri la liste en appliquant la méthode sort.
- pour fixer la valeur d'un champ : `setattr(myObj, 'myField', 35)`

`abs(-2)` : valeur absolue du nombre.
Il n'y a pas de fonction standard sign en python pour avoir le signe d'un nombre !

- on peut cependant en créer une facilement : `sign = lambda x: (x > 0) - (x < 0)`. Alors :
  - `sign(2)` : donne `1`.
  - `sign(-2)` : donne `-1`.
  - `sign(0)` : donne `0`.
- ou alors, on peut utiliser `numpy.sign(2)`, une fonction du package numpy (qui renvoie un int64 ou float64).

Arrays

Arrays :

- c'est une séquence qui permet de représenter de manière compacte une liste de valeurs toutes du même type (élémentaire). Sa taille n'est pas fixe contrairement aux arrays numpy.
- faire `from array import array` pour l'utiliser.

Définition d'une array :

- `a = array('d', [2.3, 5.4, 3.2, 2.7])` : définit une array de type double.
- les différents types sont :
  - c : caractère.
  - b : entiers signés sur 1 octet.
  - B : entiers non signés sur 1 octet.
  - i : entiers signés sur 2 octets.
  - I : entiers non signés sur 2 octets.
  - f : float sur 4 octets.

    - d : double sur 8 octets.
- pour avoir la place occupée par un élément : `a.itemsize`.
- pour avoir le type (lettre ci-dessus) : `a.typecode`.
- pour définir une array de caractères, donner une chaîne : `a = array('c', 'abcde')`.
- `a.tolist()` : renvoie une liste normale.

Lectures et modifications d'une array :

- `a[0], a[1:3]` et toutes les opérations sur les séquences sont valables.
- `a.append(5.3)` : rajoute une valeur à la fin.
- `a.extend([2.1, 8.3])` : rajoute à la fin la liste ou l'array (qui doit alors avoir le même type).
- `a.count(3.2)` : renvoie le nombre de valeurs qui sont à 3.2 dans l'array.
- `a.index(3.2)` : renvoie le plus petit index (origine à 0) dont la valeur est 3.2.
- `a.insert(2, 7.7)` : insère la valeur à la position 2 (origine à 0) et décale les valeurs après.
- `a.insert(-1, 7.7)` : insère la valeur à l'avant-dernière position.
- `a.remove(7.7)` : enlève la première valeur qui vaut 7.7 (et décale le reste).
- `a.pop(2)` : enlève la valeur à la position 2 (origine à 0) et la renvoie.
- `a.pop()` : enlève la dernière valeur et la renvoie (équivalent à `a.pop(-1)`.
- `a.reverse()` : renverse la liste en place.

Exceptions

Syntaxe des exceptions :

```
while 1:
    try:
        x = int(raw_input('number:'))
        break
    except ValueError:
        print('try again')
```

Une clause except peut avoir plusieurs exceptions :

- séquentiellement (mais seul le premier qui matche sera exécuté, même si plusieurs matchent) :
    - `try:`

```
    f = open('toto')
    s = f.readline()
    i = int(string.strip(s))
except IOError:
    print('I/O error')
except ValueError:
    print('could not convert')
except: # all other exceptions
    print('unexpected')
    raise # reraise exception after message printing
else: # done if no exception raised
    f.close()
```

- except sans type d'exception catche tous les types.
- raise tout seul dans un except : lève à nouveau la même exception.
- ou récupération de différents types d'exception en même temps :

```
try:
    ...
except (RuntimeError, TypeError, NameError):
    ...
```

Les exceptions sont des classes :

- elles dérivent toutes de la classe BaseException, mais si on veut créer ses propres classes d'exception, les faire dériver de la classe Exception, plutôt que BaseException (voir ci-dessous).
- lors de l'appel à sys.exit(), une exception SystemExit est levée. SystemExit dérive directement de BaseException, donc si on catche BaseException, on va aussi la catcher !
- lors d'un Ctrl-C, une exception KeyboardInterrupt est levée. KeyboardInterrupt dérive aussi directement de BaseException, donc si on catche BaseException, on va aussi la catcher !
- on peut définir ses propres classes d'exception en les faisant dériver de la classe Exception (éviter de les faire dériver de BaseException).
- on peut catcher toutes les exceptions "normales" et examiner leur type ou le message associé :

```
try:
    x = 'a' + 8
except Exception as e:
    print(type(e))
    print(str(e))
```

- StandardError est une exception dérivée de Exception et de laquelle la plupart des exceptions dérivent (notamment RuntimeError, KeyError, ...)

On peut exécuter du code si aucune exception levée :

```
try:
    ...
except TypeError:
    ...
else:
    print('Aucune exception n'a eu lieu')
```

Certaines exceptions ont des arguments que l'on peut alors récupérer :

```
except NameError as e: # un argument, e, contenant les arguments de
l'exception.
    print(arg:', e)
```

Les exceptions non traitées remontent d'appel en appel.
Pour catcher toutes les exceptions en récupérant leur message :

```
try:
    ...
except Exception as e:
   print(str(e))
   ...
```

Levée d'exception explicite :

- `raise NameError('badName')` ou aussi `raise NameError, 'badName'` : lève une exception NameError avec un argument.
- on peut lever une exception avec plusieurs arguments de types quelconques (récupérables sous forme de tuple) : `raise Exception(arg1, arg2)`
- on peut alors récupérer ces arguments en faisant :

```
except Exception as e:
  print(e.args)
```

finally : code toujours executé, qu'il y ait eu levée d'exception ou non :

```
try:
    1 / 0
except TypeError:
    print('type error')
finally: # toujours exécuté, avant de lever l'exception
    print('always done')
```

Traceback : permet de voir la pile des appels lors d'une erreur :

- faire `import traceback`.
- `traceback.print_exc()` : imprime sur stderr la pile des appels de la dernière exception.
- `traceback.format_exc()` : permet de récupérer la pile des appels sous forme de chaîne de caractères.

Pour avoir la pile des appels :

- `import inspect; stack = inspect.stack()` : renvoie une liste avec un élément par élément de la pile avec le premier élément étant l'appel à cette fonction stack()

- pour avoir le nom de la fonction appelante la fonction contenant cet ordre : `stack[1][3]`
- pour avoir la ligne ou la fonction est appelée : `stack[1][2]`

Pour avoir la liste des numéros de lignes après une exception :

```
info = sys.exc_info()[2]
while True:
  print(str(info.tb_lineno)+ ' at ' + info.tb_frame.f_code.co_filename)
  info = info.tb_next
  if info is None:
    break
```

Assertions en python :

- `assert x == 2` : vérifie si la condition est vraie, et si non, lance une exception AssertionError.
- `assert x == 2, 'x ne valait pas 2'` : idem, mais en plus, affiche le message d'erreur indiqué.

les assertions sont vérifiées, sauf si on fournit à python l'option -O est fournie (python -O ...)

Fichiers

Ouverture d'un fichier : `fh = open('/tmp/toto', 'w')` : le mode peut être 'r' pour read ('r' fournit des strings, 'rb' fournit des bytes), 'w' pour write, 'a' pour append, 'r+' pour read et write, ou omis (read par défaut).
attention, sous windows, il y a aussi les modes 'rb', 'wb', 'r+b' avec b pour binaire : en effet, en mode ascii, les retours chariots sont altérés, en mode binaire, ils ne le sont pas !
Fonctions sur les file handles :

- `fh.read(1000)` : lit 1000 caractères.
- `content = fh.read()` : lit tout le fichier (ou le reste) (chaîne vide si fichier fini).
- attention : `fh.read()` est bloquant. Si ce qu'on lit est en cours d'écriture, il va attendre que l'EOF final soit disponible ! Si on veut faire un read non bloquant, il faut transformer le file handle en non bloquant : `import fcntl; fcntl.fcntl(fh, fcntl.F_SETFL, fcntl.fcntl(fh, fcntl.F_GETFL) | os.O_NONBLOCK)`
- `fh.readline()` : lit la ligne suivante (chaîne vide si fichier fini).
- `fh.readlines()` : renvoie une liste de toutes les lignes.
- `fh.write('myString')` : écrit une string
- `fh.tell()` : renvoie la position courante dans le fichier.
- `fh.close()` : ferme le fichier
- `fh.seek(offset, from_what)` : repositionne le curseur :
  - si from_what = 0 : offset à partir du début

    - si from_what = 1 : offset à partir de la position courante
    - si from_what = 2 : offset à partir de la fin
    - si from_what omis : il vaut 0.

(offset peut être négatif)

Pour boucler sur les lignes d'un fichier :

```
fh = open('myFile')
for line in fh:
  print(line)
```

attention : ça fait du buffering, contrairement à un simple readline() ! Donc utiliser readline() si on veut l'éviter.

On peut récupérer directement la liste des lignes d'un fichier en faisant : `[x.replace(' ', '') for x in open('myFile')`

Lecture d'un fichier avec fermeture automatique de celui-ci à la fin, comme s'il y avait un finally (même en cas d'exception) :

```
with open('myfile.txt') as fh:
    for line in fh:
        print(line)
```

Diverses fonctions :

- `os.environ` : renvoie un dictionnaire de tous les variables d'environnement avec leur valeur
- `os.getcwd()` : renvoie le directory courant.
- `os.chdir(myDir)` : change le directory courant.
- `os.getuid(), os.geteuid(), os.getgid(), os.getegid()` : renvoient les uid ou gid, ou les uid ou gid effectifs
- `os.getlogin()` : renvoie le login.
- `os.getpid()` : renvoie le process id (PID).
- `os.getenv('MY_VAR'), os.putenv('MY_VAR', '1'), os.unsetenv('MY_VAR')` lit, fixe ou détruit une variable d'environnement (valable pour les sous-process lancés)
- `os.umask(0o002)` : positionne le umask à 002. Attention, un nombre en octal doit être précédé de 0o !
- `os.chmod('myfile', 0o755)` : fait un changement des droits. Attention, un nombre en octal doit être précédé de 0o !
- `os.uname()` : renvoie un tuple avec les infos de uname.
- `os.listdir('/myDir')` : renvoie la liste des fichiers (entrées) du directory.
- `os.mkdir(myDir)` : crée un directory. `os.mkdir(myDir, 0o777)` en indiquant le mode, mais en appliquant par dessus le umask qui vient restreindre les droits. Lève une exception si le directory existe déjà.

- `os.makedirs(myDir)` : crée le directory en créant tous les intermédiaires si nécessaire.
- `os.remove(myFile)` : détruit un fichier (pas un directory).
- `os.unlink(myFile)` : comme os.remove().
- `os.rmdir(myDir)` : détruit un directory seulement s'il est vide.
- `os.removedirs(myDir)` : détruit un directory et si son parent est vide, le détruit aussi et remonte comme ça toute l'arborescence jusqu'à un directory non vide. Pour détruire un directory et tout ce qu'il contient, utiliser shutil.rmtree
- `os.rename(oldFile, newFile)` : renomme un fichier ou directory. Attention : écrase le fichier destination s'il existe ! Attention : ne marche pas entre filesystems (utiliser shutil.move() pour cela).
- `os.symlink(myFile, myLink)` : crée un lien symbolique.
- `os.symlink('myFile', '/myDir/myFile2')` : Pour faire un lien relatif de myFile vers myFile2 dans le directory /myDir
- `os.link(myFile, myLink)` : crée un hard lien (spécifique unix/linux).
- `os.walk(myDir)` : renvoie un generator qui renvoie à chaque fois un tuple avec (dirpath, dirnames, filenames) où dirpath est le directory courant (chemin complet à partir de myDir inclus), dirnames la liste de tous les sous-directories, et filenames la liste des fichiers (sans les directories).

```
for dirpath, dirnames, filenames in os.walk('docs'):
    print(dirpath, dirnames, filenames)
```

- `os.stat(myFile)` : permet d'avoir les infos sur l'inode comme la date de dernière modification, la taille du fichier ou le user :

```
import stat
print(os.stat(myFile)[stat.ST_MTIME])
```

  - `stat.ST_UID` : le user id
  - `stat.ST_GID` : le group id
  - `stat.ST_SIZE` : la taille
  - `stat.ST_ATIME` : le timestamp de la dernière date d'accès.
  - `stat.ST_MTIME` : le timestamp de la dernière date de modification
- accès en lecture ou écriture :
  - Pour tester si un directory peut être écrit : `os.access(myDir, os.W_OK | os.X_OK)`
  - Pour tester si un fichier peut être lu : `os.access(myFile, os.R_OK)`

- `os.kill(myPid, 9)` : kill le process (utiliser `import signal; os.kill(myPid, signal.SIGKILL)` pour être plus propre.

Appels systèmes :

- `os.system(myCommand)` : avec retour du statut, mais on ne peut pas récupérer le stdout du programme appelé. Cette méthode est déconseillée.
- l'appel est bloquant.
- voir aussi le module subprocess
-