

I Dictionnaire

- Trois fonctions importantes sur les dicts : `keys()`, `values()` et `items()`.
- La méthode `keys()` d'un dictionnaire retourne la liste de toutes les clés. Cette liste ne suit pas l'ordre dans lequel le dictionnaire a été défini (souvenez-vous, les éléments d'un dictionnaire ne sont pas ordonnés) mais cela reste une liste.
- La méthode `values()` retourne la liste de toutes les valeurs. La liste est dans le même ordre que celle retournée par `keys()`, on a donc

```
dico.values()[n] == dico[dico.keys()[n]]    pour toute valeur de n.
```

- La méthode `items` retourne une liste de tuples de la forme (*clé*, *valeur*). La liste contient toutes les données stockées dans le dictionnaire.

I-1 Fonctions sur les Dictionnaires

- *Dictionnaire* (dico) se dit également *map*.
- Pour illustrer les opérateurs applicables sur les Dictionnaires, notons que les exemples suivants créent tous le dico `{"one": 1, "two": 2, "three": 3}`:

```
a = dict(one=1, two=2, three=3)
b = {'one': 1, 'two': 2, 'three': 3}
c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
d = dict([('two', 2), ('one', 1), ('three', 3)])
e = dict({'three': 3, 'one': 1, 'two': 2})
a == b == c == d == e
# True
```

- **len(d)** : nombre d'éléments de *d*.
- **d[clé]** : renvoie la valeur associée à *clé*. Peut lever l'exception *KeyError* si *clé* n'est pas dans *d*. Voir la doc sur les sous classes de *Dict* et la fonction `__missing__(self, clé)` qui peut définir le comportement en cas d'absence de *clé* dans *d[clé]* (voir aussi INF TC2).
- **d[clé] = val** : affecte *val* à *d[clé]*.
- **del d[clé]** : supprime la valeur *d[clé]* de *d*. Lève l'exception *KeyError* si *clé* n'est pas dans *d*.
- **clef in d** : renvoie vrai si *d* a la clé *clef*, faux sinon.
- **clef not in d** : équivalent à `not clef in d`.
- **iter(d)** : renvoie un itérateur sur les clés dans *d*. C'est un raccourcis pour `iter(d.keys())`.
- **d.clear()** : supprime tous les éléments de *d*.
- **d.copy()** : fait une copie de *d*.
- **fromkeys(seq[, val])** : crée un nouveaux Dictionnaire en prenant les clés dans *seq* et valeur dans *val*.
- **d.fromkeys()** : est une méthode de la classe *Dict* qui renvoie un nouveau Dictionnaire avec la valeur par défaut *None*.
- **d.get(clé[, défaut])** : renvoie la valeur associée à *clé* si *clé* est dans *d*. Si *défaut* n'est pas donné, on prend *None*. Cette méthode n'émet donc pas d'exception *KeyError*.
- **d.items()** : renvoie une séquence (une *vue*, voir la doc) de couples (*clé*, *valeur*) de *d*.
- **d.keys()** : renvoie une *vue* des clés de *d*.
- **d.pop(clé[, défaut])** : si *clé* est dans *d*, on la supprime et on renvoie sa valeur ; sinon, renvoie *défaut*. Si *défaut* n'est pas fourni et *clé* n'est pas dans *d*, l'exception *KeyError* est levée.
- **d.popitem()** : supprime et renvoie un couple (*clé*, *val*) arbitraire dans *d*.
d.popitem() est utilisé pour itérer sur un Dictionnaire et supprimer les entrées. La même utilisation existe pour les ensembles (*set*). Si *d* est vide, une exception *KeyError* est levée.
- **d.setdefault(clé[, défaut])** : si *clé* est dans *d*, la valeur associée est renvoyée. Sinon, on insère le couple (*clé*, *défaut*) et renvoie *défaut*. La valeur par défaut de *défaut* est *None*.

- **d.update([couples])** : met à jour *d* avec les paires dans *couples* en remplaçant éventuellement les clés existants dans *d*. Renvoie *None*.
d.update() accepte soit un autre dico soit un itérable sur couples (*clé,valeur*). Si l'argument *keyword=..* est donné, *d* est mis à jour avec ces valeurs de la forme (*clé,valeur*).
Exemple : `d.update(red=1, blue=2)`.
- **d.values()** : renvoie une vue des valeurs de *d*. Voir la doc pour les vues (*view*).
- L'opérateur **d1==d2** s'applique si et seulement si *d1* et *d2* ont les mêmes paires (*clé, val*). Les autres comparaisons ('<', '<=', '>=', '>') lèvent l'exception `TypeError`.

I-2 Exemples

```
capitals = { 'Iowa': 'DesMoines', 'Wisconsin': 'Madison' }
print(capitals['Iowa'])
# DesMoines

capitals['Utah'] = 'SaltLakeCity'
print(capitals)
# { 'Iowa': 'DesMoines', 'Utah': 'SaltLakeCity', 'Wisconsin': 'Madison' }

capitals['California'] = 'Sacramento'
print(len(capitals))
# 4

for k in capitals:
    print(capitals[k], " is the capital of ", k)
# DesMoines is the capital of Iowa
# SaltLakeCity is the capital of Utah
# Madison is the capital of Wisconsin
# Sacramento is the capital of California
```

- D'autres exemples :

```
phone_ext={'david':1410, 'brad':1137}
phone_ext
# { 'brad': 1137, 'david': 1410 }

phone_ext.keys()           # Renvoie les clés de phone_ext
# dict_keys(['brad', 'david'])

list(phone_ext.keys())
# ['brad', 'david']

"brad" in phone_ext
# True

1137 in phone_ext
# False                     # 1137 n'est pas une clé

phone_ext.values()         # Renvoie les valeurs de phone_ext
# dict_values([1137, 1410])

list(phone_ext.values())
# [1137, 1410]

phone_ext.items()
# dict_items([('brad', 1137), ('david', 1410)])

phone_ext.get("kent")
# Rien ! La clé n'y est pas.

phone_ext.get("kent", "NO ENTRY") # Si on veut récupérer la réponse en cas d'absence de la clé.
'NO ENTRY'

del phone_ext["david"]
phone_ext
# { 'brad': 1137 }
```

I-3 Dictionnaire et tuple

```
params = {"server": "CRI", "database": "master", "uid": "chef", "pwd": "secret"}
params.keys()
# ['server', 'uid', 'database', 'pwd']

params.values()
# ['CRI', 'chef', 'master', 'secret']

params.items()
# [('server', 'CRI'), ('uid', 'chef'), ('database', 'master'), ('pwd', 'secret')]
```

- En liste en compréhension :

```
params = {"server": "CRI", "database": "master", "uid": "chef", "pwd": "secret"}
params.items()
# [('server', 'CRI'), ('uid', 'chef'), ('database', 'master'), ('pwd', 'secret')]

[k for k, v in params.items()]
# ['server', 'uid', 'database', 'pwd']

[v for k, v in params.items()]
# ['CRI', 'chef', 'master', 'secret']

["%s=%s" % (k, v) for k, v in params.items()]
# ['server=CRI', 'uid=chef', 'database=master', 'pwd=secret']
```

I-4 Exercice : les mots et le dictionnaire

- On veut écrire une fonction `compterMots()` ayant un argument (une chaîne de caractères) et qui renvoie un dictionnaire qui contient la fréquence de tous les mots de la chaîne entrée.
- Donner cette fonction.

```
# fonction
def compterMots(texte):
    dict = {}
    listeMots = texte.split()

    for mot in listeMots:
        if mot in dict:
            dict[mot] = dict[mot] + 1
        else:
            dict[mot] = 1
    return dict

# programme principal
res = compterMots("Ala Met Asn Glu Met Cys Asn Glu Hou Ala Met Gli Asn Asn")
for c in res.keys():
    print(c, "---->", res[c])
```

I-5 Exemple : tableau de tableau avec Dictionnaire

- Le type dictionnaire (ou tableau associatif) permet de représenter des tableaux structurés. En effet, à chaque clé un dictionnaire associe une valeur, et cette valeur peut elle-même être une structure de donnée (liste, tuple ou un dictionnaire ...).
- Soit le tableau suivant représentant des informations physico-chimiques sur des éléments simples (température d'ébullition (Te) et de fusion (Tf), numéro (Z) et masse (M) atomique :

Au	Te/Tf	2970	1063
	Z/A	79	196.967
Ga	Te/Tf	2237	29.8
	/A	31	69.72

- Affectez les données de ce tableau à un dictionnaire dict python de façon à pouvoir écrire par exemple :

```
print(dict["Au"]["Z/A"][0]) # affiche : 79
```

```
dict = {"Au": {"Te/Tf": (2970, 1063),
               "N/M atomique": (79, 196.967)},
        "Ga": {"Te/Tf": (2237, 29.8),
               "N/M atomique": (31, 69.72)}}

# programme principal
print(dict["Au"], "\n")
print(dict["Ga"]["Te/Tf"], "\n")
print("Numero atomique de l'or :", dict["Au"]["N/M atomique"][0], "\n")
print("Masse atomique de l'or :", dict["Au"]["N/M atomique"][1])
```

📄 pour copier un Dictionnaire :

→ utiliser la fonction générique pour copier (du module `copy`) : `copy.copy()` ou `copy.deepcopy()`. En même temps, Dict a sa propre méthode `copy()`, au même titre que pour la liste : ce sont des objets mutables. Pour les listes, on peut utiliser la tranche `new_L=L[:]`.

I-6 Dictionnaires (Dict) et Vues

- Les objets renvoyés par les fonctions **dict.keys()**, **dict.values()** et **dict.items()** sont des **vues** (*view*). Ce sont des vues dynamiques des dicos (au sens Bases de données relationnelles). Ce qui veut dire que si le contenu d'un Dico change, les vues reflètent ces changements.
- Les vues sont itérables et transmettent par *yield* leur contenus et acceptent le test d'appartenance (voir ci-dessous).
→ Ce qui veut dire que des dicos de grandes tailles peuvent être manipulés sans encombrer la mémoire.

☞ Noter cependant que les vues ne sont pas des générateurs : un accès à une vue n'épuise pas ses éléments (comme c'est le cas des générateurs).

I-6-a Les fonctions de Vue

- Les fonctions de manipulation des vues :
 - **len(dictview)** : renvoie le nombre d'entrées du Dictionnaire (ou de la *vue*).
 - **iter(dictview)** : renvoie un itérateur sur les clés, les valeurs ou sur les couples (*clé, valeur*).
L'itération sur une vue sera dans un ordre quelconque (mais pas aléatoirement). L'itération varie selon les implantations de Python et dépend de l'historique des insertions et suppression dans le dico.
 - Si le dico n'a pas subi de modification, la vue correspondra à l'ordre des insertions. Ce qui autorise la création des paires (*clé, valeur*) en utilisation *zip()* (création de couples) :

```
pairs = zip(d.values(), d.keys())
```
 - Une autre manière de créer la même liste est : `pairs = [(v, k) for (k, v) in d.items()]`.
 - L'itération sur une vue pendant les ajouts et suppressions sur le dico peut lever l'exception *RuntimeError* ou échouer d'itérer sur certaines entrées du dico.
 - **x in dictview** : renvoie True si x est dans le dico. x peut être une clé, une valeur ou un couple (*clé, valeur*)
- Une vue sur les clés du dico est un ensemble (*set* sans doublon mais non ordonnée) puisque les clés du dico sont uniques (et hashées).
- Si toutes les *valeurs* des couples (*clé, valeur*) sont hashables et que les couples (*clé, valeur*) sont uniques alors la vue est également un ensemble (*set*). Mais les valeurs ne sont pas traitées comme un ensemble (*set*) puisqu'elles risquent de se répéter.
- Pour les clés et les couples traités comme ensembles (*sets*), toutes les opérations (telles que `==`, `<` ou `)` de la classe abstraite *collections.abc.Set* sont disponibles.

I-6-b Exemples

```
# Dictionnaire et Vues
dishes = {'eggs': 2, 'sausage': 1, 'bacon': 1, 'spam': 500}
keys = dishes.keys()
values = dishes.values()

keys
# dict_keys(['eggs', 'spam', 'sausage', 'bacon'])

values
# dict_values([2, 500, 1, 1])

dishes['eggs']
# 2
```

- Les vues reflètent les modifications :

```
# Modification d'une valeur
dishes['eggs'] += 1
dishes['eggs']
# 3

values
# dict_values([3, 500, 1, 1]) # La vue est modifiée

# Ajout d'une entrée
dishes['chesse'] = 300
dishes
# {'bacon': 1, 'chesse': 300, 'eggs': 3, 'sausage': 1, 'spam': 500} # La vue est modifiée

keys
# dict_keys(['eggs', 'chesse', 'spam', 'sausage', 'bacon']) # La vue est modifiée
```

- Itération sur une vue :

```
# itération
n = 0
for val in values:
    n += val
print(n)
# --> 805

# Les vues ne sont pas &puisées par un accès / itération
# clés et valeurs : on itère suivant le même ordre
list(keys)
# ['eggs', 'chesse', 'spam', 'sausage', 'bacon']

list(values)
# [3, 300, 500, 1, 1]
```

- Les vues reflètent les suppressions :

```
# Les vues sont des objets dynamiques et reflètent les modifications du dico
del dishes['eggs']
del dishes['sausage']

list(keys)
# ['chesse', 'spam', 'bacon']
```

- Les vues supportent les opérateurs ensemblistes :

```
# opérations sur les ensembles
keys & {'eggs', 'bacon', 'salad'}      # Intersection
# {'bacon'}

keys ^ {'sausage', 'juice'}           # différence symétrique : les éléments dans l'un ou l'autre mais pas
les deux
# {'bacon', 'chesse', 'juice', 'sausage', 'spam'}
```

II A propos du BE3

- Ci-dessous quelques fonctions utiles au BE3 (Monnaie).

II-1 Modification de la valeur d'une clé

- Soit le dictionnaire $D = \{25 : \{15 : \text{None}, 12 : \text{None}\}, 20 : \text{None}\}$.

Pour remplacer la valeur associée à un noeud (par exemple le noeud 15) et la remplacer par $\{3 : \text{None}\}$ dans ce dictionnaire, on écrira `remplacer_info_d_une_clef(D, 15, {3 : None})`.

```
def remplacer_info_d_une_clef(Dico, cle_recherchee, new_val_de_la_cle) :
    """ Rempacement de la partie info pour une clé dans un Dictionnaire """
    if not Dico : return False      # la clef est devenue None de proche en proche
    Lst_keys=Dico.keys()
    if cle_recherchee in Lst_keys :
        Dico[cle_recherchee]=new_val_de_la_cle
        return True                 # Pour terminer
    else :
        for k in Lst_keys :
            Rep=remplacer_info_d_une_clef(Dico[k], cle_recherchee, new_val_de_la_cle)
            if Rep : return True     # On termine
        return False
```

- Un test :

```
B= {28: {27: {4: None, 26: None, 20: None}, 21: {20: None, 14: None}, 5: {4: None}}}
remplacer_info_d_une_clef(B,26, {25:None, 19:None, 3:None})
print(B)

# {28: {27: {20: None, 26: {3: None, 25: None, 19: None}, 4: None}, 21: {20: None, 14: None}, 5: {4: None}}}
```

```
B
# {28:
#   {5:
#     {4: None},
#     21:
#     {14: None, 20: None},
#     27:
#     {4: None, 20: None,
#      26:
#      {3: None, 19: None, 25: None}}}}
```

II-2 Développer un niveau pour une valeur

- Il s'agit de développer un niveau pour un noeud (une valeur) dans un Dictionnaire. On assigne à *M_is_Val_noeud* un triplet (quand cela est possible) dont chaque élément est *M_is_Val_noeud - un_element_de_S* (*S* est la liste des pièces disponibles).
- Par exemple, pour le dictionnaire $D = \{25 : \{15 : \text{None}, 12 : \text{None}\}, 20 : \text{None}\}$, si on décide de développer un niveau pour la clé **12** et lui donner les descendants 12-3 et 12-5 (le stock de pièces serait $S = [3, 5]$), on obtiendra le nouveau dictionnaire $D = \{25 : \{15 : \text{None}, 12 : \{9 : \text{None}, 7 : \text{None}\}\}, 20 : \text{None}\}$

```
def developper_un_niveau_pour_un_noeud(Dico, S, M_is_Val_noeud) :
    """ On développe un noeud de l'arbre : crée un niveau pour ce noeud """
    niveau={}
    lst_cles_du_niveau=[M_is_Val_noeud - piece for piece in S if piece <= M_is_Val_noeud]
    niveau=niveau.fromkeys(lst_cles_du_niveau, None)
    remplacer_info_d_une_clef(Dico, M_is_Val_noeud, niveau)
```

- Pour développer le dictionnaire (l'arbre) jusqu'à ce que le noeud contenant zéro soit inséré :

```
def creer_arbre(S, M) :
    B={}
    B[M]=None
    # On fait le premier niveau
    Lst_niv = [M]

    if 0 not in Lst_niv :      # On n'a pas fini
        while True :
            for val_noeud in Lst_niv :
                developper_un_niveau_pour_un_noeud(B, S, val_noeud)

            # Récupérer les nouveaux noeuds générés
            Lst_niv=[v - piece for v in Lst_niv for piece in S if piece <= v]
            if 0 in Lst_niv : break      # On a fini

    # 0 a été généré
    print(B)
```

- Un exemple : on développe un arbre pour le montant **M12** étant donné le stock de pièces $S = [1, 10]$.

```
M=12
S=[1,10]
creer_arbre(S, M)

# {12: {2: {1: {0: None}}, 11: {1: None, 10: {0: None, 9: None}}}}
```

→ C'est à dire

```
{12:
  {2:                               # 12-10
    {1:                             # 2-1
      {0: None}                     # 1-1
    },
    11:                             # 12-1
    {1: None,                       # 11-10
      10:                            # 11-1
      {0: None,                     # 10-10
        9: None                     # 10-1
      }
    }
  }
}
```

- Il vous reste à reconstituer le chemin qui mène de la racine à zéro et extraire les pièces utilisées.

II-3 Deux autres fonctions

- Test de la présence d'une clé dans un dico :

```
def clef_presente_dans_Dico(Dico, cle_recherchee) :
    Lst_keys=Dico.keys()
    if cle_recherchee in Lst_keys :
        return True

    for k in Lst_keys :
        if Dico[k]== None : continue
        if clef_presente_dans_Dico(Dico[k], cle_recherchee) : return True
    return False

# — TEst
B= {28: {27: {26: None, 20: {19: None, 13: None}}, 5: None, 21: {20: None, 14: {13: None, 7: {0: None, 6: None}}}}}
print(clef_presente_dans_Dico(B, 28)) # True
print(clef_presente_dans_Dico(B, 29)) # False
print(clef_presente_dans_Dico(B, 7))  # True
```

- Extraction de l'information associée à un clé dans un dico :

```
# Les clés sont unique : si une clé est répétée, leur info est identique
def info_d_une_clef(Dico, cle_recherchee) :
    Lst_keys=Dico.keys()
    if cle_recherchee in Lst_keys :
        return Dico[cle_recherchee]

    # for k in Lst_keys :
    #     if Dico[k]== None : continue
    #     Rep=info_d_une_clef(Dico[k], cle_recherchee)
    #     if Rep : return Rep

    # On peut aussi écrire la boucle comme ceci :
    for d in iter(Dico.values()) : # chaque éléments est un Dict d'un niveau de Dico (en largeur)
        if not d : continue # Si None, on passe (car d.keys() ne passera pas
        if cle_recherchee in d.keys() : return d[cle_recherchee]
        elif d != None : return info_d_une_clef(d, cle_recherchee)
    return None

# — TEst
B= {28: {27: {26: None, 20: {19: None, 13: None}}, 5: None, 21: {20: None, 14: {13: None, 7: {0: None, 6:
None}}}}}

print(info_d_une_clef(B, 28))
# {27: {26: None, 20: {19: None, 13: None}}, 5: None, 21: {20: None, 14: {13: None, 7: {0: None, 6: None}}}}

print(info_d_une_clef(B, 26))
# None

print(info_d_une_clef(B, 20))
# {19: None, 13: None}
```