éduSCOL
Informer et accompagner les professionnels de l'éducation

Liberté • Égalité • Fraternité
RÉPUBLIQUE FRANÇAISE

MINISTÈRE DE L'ÉDUCATION NATIONALE ET DE LA JEUNESSE

VOIE GÉNÉRALE

2DE | 1RE | TLE

Numérique et Sciences informatiques

ENSEIGNEMENT
SPÉCIALITÉ

p-UPLETS NOMMÉS ET DICTIONNAIRES

- Histoire de l'informatique
- Représentation des données
- Traitement des données
- Interactions entre l'homme et la machine sur le Web
- Architectures matérielles et systèmes d'exploitation
- Langages et programmation
- Algorithmique

1. p-uplets nommés

1.1. Brève présentation

Un p-uplet nommé est un p-uplet, dont les composantes sont appelées via un descripteur au lieu d'un indice. Le principal intérêt de ce type est d'améliorer la lisibilité du code, et partant de réduire les risque d'erreurs.
Le type des p-uplets nommés n'existe pas nativement dans Python. On pourrait utiliser le module `collection.namedtuple` mais le programme invite à utiliser des dictionnaires pour limiter le nombre de syntaxes différentes à utiliser.
Voici un exemple pour montrer la syntaxe :

```
>>> monsieurX = {"nom": "X", "prenom": "Monsieur", "age": 47}
>>> monsieurX["age"]
47
```

Remarques :

- Les autres fonctionnalités des dictionnaires seront présentées dans la partie éponyme du présent document.
- Les p-uplets nommés sont normalement immutables alors que les dictionnaires (que nous utiliserons en Python pour les représenter) ne le sont pas. Ainsi, dans la suite, nous nous garderons de modifier les dictionnaires représentant des p-uplets.

1.2. Exemple d'utilisation

Les p-uplets nommés ont été introduits dans le programme de première de NSI principalement pour faciliter le traitement de données en tables.
Dans l'exemple ci-dessous, nous supposons disposer d'un fichier `notes.csv` qui contient les notes des élèves d'une classe.
Écrivons une fonction pour extraire les données, et les mettre dans un tableau de p-uplets nommés.
On commence par une version simple, où nous connaissons le nom et la position des colonnes qui nous intéressent : supposons que les quatre premières colonnes du fichier csv sont, dans cet ordre, nom, DS1, DS2, projet. Alors nous pouvons utiliser le code suivant :

```
def extrait_données(chemin):
```

```
    """ Entrée : le chemin d'accès d'un fichier csv, à données séparée par un point-virgule, dont les quatre premières colonnes sont nom, DS1, DS2, projet.
        Sortie : le tableau de p-uplets nommés représentant la table contenue dans le fichier csv.
    """
    entrée = open(chemin, "r")
    res = [] # Pour contenir le résultat.
    for ligne in entrée:
        données = ligne.strip().split(";")
        res.append(
            { "nom":données[0],
              "DS1":données[1],
              "DS2":données[2],
              "projet":données[3]
             })
    entrée.close()
    return res
```

Mais on peut faire beaucoup plus pratique. Supposons maintenant, comme c'est souvent le cas, que la première ligne contient le nom des colonnes. Nous allons pouvoir automatiquement récupérer ces noms, et les utiliser comme clefs dans nos p-uplets nommés.

```
def extrait_données(chemin):
    """ Entrée : le chemin d'accès d'un fichier csv, à données séparée par un point-virgule, dont la première ligne contient le nom des colonnes.
        Sortie : le tableau de p-uplets nommés représentant la table contenue dans le fichier csv.
    """
    entrée = open(chemin, "r")
    première_ligne = entrée.readline().strip()
    champs = première_ligne.split(";")
    nbChamps = len(champs)
    res = []

    for ligne in entrée:
        # Élève va contenir les données pour l'élève correspondant à la ligne en cours
        élève = {}
        données = ligne.strip().split(";")
        for i in range(nbChamps):
            # On peut utiliser enumerate(champs) aussi
            élève[champs[i]] = données[i]
        res.append(élève)
    entrée.close()
    return res
```

Remarque : On utilise ici le caractère mutable des dictionnaires Python pour créer nos p-uplets nommés (ligne 17). C'est un peu dommage puisque en théorie, un p-uplet est plutôt persistant...

On utilise le résultat par exemple ainsi :

```
notes = extrait_données("notes.csv")
notes[0]["DS1"] # Renvoie la note du premier élève au DS1
```

On voit bien comme l'utilisation d'un p-uplet nommé rend la syntaxe agréable et claire !
À titre d'application, rajoutons dans notre tableau la moyenne de chaque élève :

```
def notes_et_moyennes(chemin):
    """ Entrée : comme ci-dessus
        Sortie : comme ci-dessus, sauf que chaque p-uplet nommé contiendra aussi
            un champ "moyenne" contenant la moyenne de l'élève.
    """
    données = extrait_données(chemin)
    for e in données : # e parcourt l'ensemble des élèves
        ds1 = float(e["DS1"])
        ds2 = float(e["DS2"])
        projet = float(e["projet"])
        e["moyenne"] = (ds1 + ds2 + 2 * projet) / 4
        # On note la clarté de ces lignes. Merci les p-uplets nommés !
    return données
```

Prolongements :

- ▷ Le code ci-dessus ne fonctionne que si les notes sont rangées précisément dans trois colonnes nommée "DS1", "DS2", et "projet". Une piste d'amélioration serait de rajouter une ligne dans notre fichier csv qui indiquerait le type de la colonne. Ce type pourrait être choisi parmi "DS", "DM", "projet", "autre", etc. La lecture de cette information permettrait alors de repérer automatiquement les colonnes utiles au calcul de la moyenne, d'effectuer les conversions en flottant des colonnes idoines directement dans la fonction `extrait_données`, et de déterminer les coefficients pour automatiser le calcul des moyennes.
- ▷ On peut maintenant de calculer le rang de chaque élève dans la classe : ce serait une application des algorithmes de tri vus pas ailleurs.

2. Dictionnaires

2.1. Description du type

Décrire un type revient essentiellement à donner les opérations permises, voici donc les opérations permises par un dictionnaire.
Nous fixons deux ensembles C et V. L'ensemble C sera appelé l'ensemble des « clefs » et V l'ensemble des « valeurs ». Un dictionnaire permet d'associer à un élément de C un élément de V. Autrement dit, un dictionnaire est une fonction (au sens mathématique) de C dans V.
Selon le point de vue adopté ici, une clef peut ne pas avoir de valeur associée, autrement dit un dictionnaire n'est pas forcément une application de C vers V.
Les opérations permises par le type des dictionnaires sont :

- ▷ Créer un dictionnaire vide.
- ▷ Ajouter une association : étant donné $c \in C$ et $v \in V$, décider que v sera associée à c. Si une valeur était déjà associée à c, la nouvelle écrase l'ancienne.
- ▷ Lire la valeur associée à une clef : étant donné $c \in C$, renvoyer la valeur associée à c. Si aucune valeur n'est associée à c, une erreur est déclenchée.

De plus, un dictionnaire doit permettre de réaliser ces trois opérations de base efficacement. Selon les implémentations, le temps d'exécution sera en $O(\log(n))$, où n est le nombre de données enregistrées, voire en $O(1)$. Se reporter au paragraphe 2.4.

À ces trois opérations, on rajoute souvent une opération permettant de supprimer une entrée, et une permettant de savoir si une clef est présente dans un dictionnaire (pour éviter un `try`... `except` pour rattraper l'erreur déclenchée par une clef non présente). En outre, on peut aussi ajouter une opération permettant de renvoyer toutes les clefs d'un dictionnaire, ce qui permettra de parcourir toutes les données du dictionnaire. Ceci dit, si le but principal est de parcourir toutes les données enregistrées, un simple tableau ferait tout aussi bien l'affaire. L'intérêt du dictionnaire est de pouvoir trouver (ou tester la présence) une clef précise très rapidement ; les questions de rapidité seront évoquées au paragraphe 2.4.

2.2. En Python

Le type des dictionnaires est présent nativement dans Python, comme on l'a déjà vu. Voici la syntaxe des opérations de base ci-dessus, où l'on a au besoin $c \in C$ et $v \in V$:

- ▷ Créer un dictionnaire vide : `d = {}`.
- ▷ Associer v à la clé c : `d[c] = v`.
- ▷ Renvoyer la valeur associée à la clé c : `d[c]`.
- ▷ Voir si c est une clef de d : `c in d`.
- ▷ Supprimer l'entrée correspondant à c dans d : `del d[c]`. Déclenche une erreur si la clef c n'est pas présente dans d.
  On peut trouver cette syntaxe troublante... Un `del(d, c)` aurait été plus clair.
  On peut préférer utiliser `d.pop(c)` qui en plus renvoie l'élément supprimé.
- ▷ La méthode `keys` renvoie l'ensemble des clefs de d. On peut donc parcourir un dictionnaire à l'aide d'une boucle de la forme `for c in d.keys():`.

NB : il serait maladroit d'utiliser `c in d.keys()` pour tester si la clef c est dans le dictionnaire d. En effet, ceci conduirait Python à calculer la liste de toutes les clefs de d, puis à voir si c est parmi elles, pour une complexité en $O(n)$ où n est le nombre de clefs dans d, alors que précisément un dictionnaire est optimisé pour rechercher efficacement une clef particulière.
On remarquera la présence de procédures permettant de modifier un dictionnaire. Ainsi les dictionnaires de Python sont-ils modifiables, avec les conséquences habituelles (faire attention en les copiant, possibilité de les modifier par une procédure).
En outre, on retiendra que ces opérations de base sont, en gros[1], en $O(1)$.

2.3. Premier exemple : dépouillement d'une urne

À l'issue d'une élection à scrutin uninominal, on récupère un tableau contenant tous les noms inscrits sur les bulletins trouvés dans l'urne. Par exemple :

```
urne = ["Maurice", "Roger", "Maurice", "Marie", "Marie", "Jeanne", ...]
```

Nous voulons déterminer le vainqueur de l'élection, en un seul parcours de l'urne. Le plus pratique est d'utiliser un dictionnaire qui à chaque nom associera son nombre de voix.
Notons qu'un des avantages d'un dictionnaire est qu'il n'y a pas besoin de savoir à l'avance qui sont les candidats, ni même combien il y en a.
On commence par une procédure permettant d'incrémenter la valeur associée à une clef s'il y en a, ou de l'initialiser à 1 dans le cas contraire :

```
def incr_dico(d,c):
    """ Entrée : Un dictionnaire d
                 Une clef c
        Sortie : Rien (ceci est une procédure)
        Effet : - Si c est une clef dans d, la valeur associée est incrémentée
                - Sinon, elle est initialisée à 1.
    """
    if c in d:
        d[c] = d[c] + 1
    else:
        d[c] = 1
```

On écrit alors facilement le programme final :

```
def dépouillement(urne):
    d = {}
    nb_voix_max = 0
    vainqueur = ""
```

1. Voir le paragraphe suivant pour le sens de ce « en gros ».

```
    for nom in urne:
        incr_dico(d, nom)
        if d[nom] > nb_voix_max :
            nb_voix_max = d[nom]
            vainqueur = nom
    return vainqueur
```

Remarque : Cette fonction ne prend pas en compte le cas de deux vainqueur ex-aequo.

2.4. Sous le capot : brève description d'une table de hachage

Il existe différentes manières d'implémenter les dictionnaires. Python a choisi les tables de hachage. Au programme de terminale figurent les arbres de recherche qui sont une autre manière d'implémenter des dictionnaires.
Voici en quelques mots le principe d'une table de hachage. Cela vous permettra d'avoir une meilleure idée de la complexité des opérations élémentaires.
Tout d'abord, Python dispose d'une fonction qui à tout objet persistant [2] associe un nombre entier. Cette fonction s'appelle « fonction de hachage » et nous n'allons pas nous préoccuper de savoir comment elle fonctionne. Par contre nous supposerons que son temps d'exécution est en $O(1)$. Remarquons tout de même qu'il est évident qu'une telle fonction existe puisque tout objet est finalement enregistré comme une suite de bits, laquelle peut toujours être interprétée comme un nombre écrit en base deux. Cette fonction s'appelle **hash**.
Ensuite, une table de hachage est un tableau de tableaux de couples. Notons d la longueur du tableau principal. Pensons-y comme à une commode contenant d tiroirs. Chacun de ces tiroirs va contenir une liste de couples *(clef, valeur)*. Et c'est la fonction de hachage qui va indiquer dans quelle tiroir doit être rangée chaque donnée. Pour être précis, soit $(c, v) \in C \times V$ un couple *(clef, valeur)*. Celui-ci sera ajouté dans la case d'indice **hash**(c) % d de notre table. La réduction modulo d nous assure de tomber sur une case valide de la table.
Voici un exemple pour $d = 3$. Initialement, une table de hachage vide ressemblera à :

```
[ [],
  [],
  []
]
```

Je veux associer la valeur 2 à la clef `"bananes"` (mettons que je sois en train de gérer les stocks d'une épicerie). Le hash de `"banane"` est -4289889455651360333, ce qui modulo 3 donne 1, donc notre couple `("banane", 3)` ira case 1. Et notre table devient :

```
[ [],
  [("banane", 3)],
  []
]
```

J'associe à présent la valeur 5 à la clef `"celeri"`. Sachant que **hash**`("celeri") % 3` vaut 2, j'obtiens :

```
[ [],
  [("banane", 3)],
  [("celeri", 5)]
]
```

Enfin, j'ajoute 6 navets. Mais **hash**`("navet") % 3` vaut 1 : ils arrivent dans la même case que les bananes :

```
[ [],
  [("banane", 3), ("navet", 6)],
  [("celeri", 5)]
]
```

2. Python ne permet pas que les clefs soient mutables, car la moindre modification d'une clef empêcherait par la suite de retrouver la valeur associée.

On comprend le principe de la recherche : étant donnée une clef, il suffit de calculer son hash pour savoir dans quelle case de la table de hachage la chercher. Si par malheur il y avait beaucoup d'éléments dans cette case, la recherche pourrait être lente… C'est pourquoi une « bonne » table de hachage devra faire en sorte que ces cases ne soient pas trop remplies. D'une part, une « bonne » fonction de hachage devra faire en sorte de bien répartir les éléments, et d'autre part, la table sera automatiquement agrandie dès que le ratio nombre d'éléments sur nombre de cases devient trop important (supérieur à 2/3 en Python). Cette agrandissement coûtera du temps mais arrive rarement… Au final, la complexité d'une insertion et d'une lecture dans une table de hachage sera « en moyenne » en $O(1)$.

2.5. Deuxième exemple : tri par dénombrement

Programmation

La méthode de tri que nous allons présenter ici ne s'applique qu'aux tableaux d'entiers. En outre, ce tri n'est pas en place : nous allons créer un tableau distinct du tableau initial, qui contiendra les mêmes éléments mais dans l'ordre croissant. Soit `t` un tableau d'entiers. Pour trier `t`, la méthode consiste en :

1. Déterminer l'ensemble des nombres apparaissant dans `t`, et combien de fois chacun apparaît.
2. Grâce à ces informations, construire un tableau `ttrié` contenant les mêmes éléments, mais dans l'ordre.

Comme toujours la première question à se poser est la manière d'enregistrer les données nécessaires. Ici, il nous faut enregistrer pour chaque valeur apparaissant dans `t` le nombre de fois qu'elle apparaît. On pourrait utiliser un tableau `nb` tel que pour tout i, `nb[i]` contienne le nombre de i dant `t`. Ceci aurait plusieurs défauts :

- ▷ Éventuellement de nombreuses cases inutiles (si `t` contient 12, le tableau `nb` devra avoir au moins 13 cases, et tout $i \in [0, 12]$ qui n'apparaît pas dans `t` donne lieu à une case inutile).
- ▷ Que faire si `t` contient des nombres négatifs ?
- ▷ La création de `nb` va être laborieuse car il faudra l'agrandir à chaque fois qu'on rencontre un élément de `t` plus grand que les précédents.

Non, l'idéal ici est d'utiliser un dictionnaire, qui à chaque élément de `t` associe son nombre d'occurrence.
Commençons par écrire une fonction pour créer ce dictionnaire. On réutilise `incr_dico` de la partie 2.3.

```
def compte(t):
    """ Entrée : un tableau t
        Sortie : un dictionnaire qui à chaque élément de t associe
            son nombre d'occurrences.
    """

    res = {}
    for x in t:
        incr_dico(res, x)
    return res
```

Maintenant, il s'agit de construire le tableau trié à l'aide du dictionnaire. pour parcourir *dans l'ordre* toutes les valeurs prises par `t`, le plus simple est d'utiliser une boucle `for`, depuis le minimum jusqu'au maximum de `t`. Pour disposer de ces extrema, modifions la fonction précédente :

```
def compte(t):
    """ Entrée : un tableau t
        Sortie : un triplet (dictionnaire qui à chaque élément de t
            associe son nombre d'occurrences, mir(t), max(t)).
    """

    res = {}
    mini, maxi = t[0], t[0]
    for x in t:
```

```
        incr_dico(res, x)
        mini = min(mini, x)
        maxi = max(maxi, x)
    return res, mini, maxi
```

On écrit alors la fonction finale :

```
def tri_dénombrement(t):
    nb, mini, maxi = compte(t)
    res = []
    for i in range(mini, maxi + 1):
        if i in nb:
            # Il faut rajouter nb[i] fois i dans res.
            for k in range(nb[i]):
                res.append(i)
    return res
```

Remarque : On peut par une petite pythonnerie raccourcir un peu le code : la méthode `get` d'un dictionnaire permet de renvoyer la valeur associée à une clef, en précisant une valeur par défaut à renvoyer si cette clef n'est pas présente. Ainsi, dans la fonction précédente, un `nb.get(i, 0)` renverrait le nombre de i dans `t`, si i apparaît effectivement dans `t`, et 0 sinon.
La fonction peut donc se réécrire :

```
def tri_dénombrement(t):
    nb, mini, maxi = compte(t)
    res = []
    for i in range(mini, maxi + 1):
        # Il faut rajouter nb[i] fois i dans res.
        for k in range(nb.get(i, 0)):
            res.append(i)
    return res
```

2.6. Complexité

La terminaison de cette fonction etc claire puisqu'elle n'emploie ni boucle conditionnelle, ni récursivité. Par contre, cet exemple est intéressant car non trivial du point du vue de la complexité. Soit `t` un tableau et n son nombre d'éléments, et calculons la complexité de `tri_dénombrement(t)`.
Tout d'abord, `incr_dico` est toujours en $O(1)$, d'où `compte(t)` est en $O(n)$.
On constate ensuite que la complexité de la fonction principale va dépendre du minimum et du maximum de `t`. Notons m et M ces deux nombres.

- ▷ La boucle principale s'effectue $M - m$ fois.
- ▷ Pour chaque valeur de i, la boucle intérieur tourne au maximum n fois.

Ceci nous indique donc une complexité en $O\left((M-m)n\right)$.
Mais ce résultat, correct, est beaucoup trop grossier ! L'idée est la complexité maximale de la boucle interne (à savoir au plus n itérations) *ne peut pas* être atteinte à chaque itération de la boucle externe.
En effet, la boucle intérieure s'effectue *au total* n fois car à chaque itération elle ajoute un élément à `res`, et celui-ci finit avec n éléments. La ligne 7 est donc exécutée au total n fois, et notre fonction effectue n `append`.
Si nous ne comptons que les `append`, nous pouvons affirmer que la complexité est en $O(n)$. Cependant, n'oublions pas le `get` qui est quant à lui effectué $M - m$ fois.
Au total, nous avons n `append` et $M - m$ `get`, ce qui nous fait une complexité en $O(n) + O(M - m)$, autrement dit en $O\left(n + M - m\right)$.
Il s'agit donc d'un tri particulièrement efficace si l'amplitude des nombres à trier n'est pas trop grande.

Remarque : Si l'on souhaite également compter le coût de gérer les boucles for elles-même, on trouve que la boucle externe a un coût en $O(M-m)$, et la boucle interne a un coût *total* en $O(M-m)$.