Recherche dans une table

I Introduction

Un fois qu'un ensemble de données est chargé dans une table, il devient possible d'exploiter ces données à l'aide des opérations de manipulation de tableaux. On va, par exemple, pouvoir extraire des données cibles, tester la présence de certaines données ou faire des statistiques. Ces opérations sont appelées des *requêtes*. Prenons comme exemple de table pour la suite la table suivante.

prénom	jour	mois	année	projet
Brian	1	1	1942	programmer avec style
Grace	9	12	1906	production de code machine
Linus	28	12	1969	un petit système d'exploitation
Donald	10	1	1938	tout sur les algorithmes
Alan	23	6	1912	déchiffrer des codes secrets
Blaise	19	6	1623	machine arithmétique
Margaret	17	8	1936	atterrissage d'un module lunaire
Alan	1	4	1922	ce qu'un programmeur doit savoir
Joseph Marie	7	7	1752	programmer des dessins

II Recherche

Considérons la fonction suivante, qui renvoie **True** ou **False** selon la présence ou l'absence d'un élément **v** dans un tableau **t**.

```
def appartient(v, t):
  for x in t:
    if v == x: return True
  return False
```

Cette fonction pourrait être appliquée directement à une table de données. Il suffirait pour que cela ait du sens que la "valeur" **v** cherchée ait la forme d'une ligne complète. On pourrait donc par exemple chercher dans la table une ligne qui soit précisément la suivante.

```
{"prénom": "Donald", "jour": "10", "mois": "1", \
 "année": "1938", "projet": "tout sur les algorithmes"}
```

Faire ceci demande à l'avance de savoir ce que contient une ligne, ce qui ne correspond pas à l'utilisation courante d'une telle table. On cherchera en général plutôt à savoir s"il y a dans la table une ligne pour laquelle le prénom est **Donald**, pour obtenir ensuite certaines des informations associées.

II.1 Recherche en fonction d'un attribut clé

Une fonction cherchant dans une table un élève désigné par son prénom et renvoyant **True** ou **False** selon sa présence ou son absence peut être obtenue par une légère modification de la fonction précédente.

```
def appartient(p, eleves):
  for e in eleves:
    if e["prénom] == p: return True
  return False
```

Comme auparavant, on y passe en revue l'ensemble des élément du tableau, c'est-à-dire des lignes de la table, mais le test d'égalité est fait sur la composante précise nous servant de critère de cherche.

II.2 Récupération d'une donnée simple

A partir de cette base, on peut déduire d'autres fonctions qui, au lieu de seulement indiquer la présence ou l'absence d'un élève, renvoient certaines des informations associées. La fonction

```
def projet_de(p, eleves):
  for e in eleves:
    if e["prénom] == p:
      return e["projet"]
```

permet ainsi de récupérer le projet d'un élève désigné par son prénom. La fonction renvoie **None** si aucun élève n'a le prénom **p**. Que se passe-t-il cependant si deux élèves portent le même prénom ? Si on ne s'en est pas déjà convaincu à la simple lecture du code, une petite expérience permet de constater qu'un seul projet est renvoyé, en l'occurrence celui de l'élève dont la ligne apparaît en premier.

```
>>> projet_de("Alan", eleves)
"déchiffrer des codes secrets"
```

On peut donc vouloir préciser la recherche, en tenant compte par exemple à la fois du prénom et de l'année de naissance. La condition d'une instruction **if** pouvant être arbitrairement complexe, il est tout à fait possible d'enrichir le test **e[prénom] == p** de sorte qu'il consulte plusieurs des attributs de la ligne inspectée.

```
def projet_de(p, a, eleves):
  for e in eleves:
    if e["prénom] == p and e["année"] == a:
      return e["projet"]
```

III Agrégation

Les opérations d'*agrégation* combinent les données de plusieurs lignes pour produire un résultat et en particulier une statistique sur ces données. On pourra, par exemple, compter le nombre de lignes répondant à une certaine condition, ou calculer la valeur moyenne d'un attribut.

III.1 Comptage d'occurences

Les fonction visant à compter le nombre d'occurrences d'un élément dans un tableau peuvent, comme les fonctions de recherche, être adaptées pour compter dans une table le nombre de lignes validant une certaine condition. Ainsi, la fonction suivante compte le nombre d'élèves dont l'année de naissance est égale à l'année **a** donné en paramètre.

```
def eleves_nes_en(a, eleves):
  nb = 0
  for e in eleves:
    if e["année"] == a:
      nb += 1
  return nb
```

III.2 Sommes et moyennes

De manière générale, les opérations d'agrégation peuvent être réalisées en utilisant des accumulateurs qui enregistrent progressivement un bilan du parcours de la table. Pour calculer l'âge moyen de notre classe à la fin de l'année **a**, on peut ainsi utiliser un accumulateur pour la somme des âges.

```
def age_moyen(a, eleves):
  somme = 0
  for e in eleves:
    somme += a - e["année"]
  return somme / len(eleves)
```

Un tel code peut naturellement être adapté pour ne calculer la moyenne que d'une certaine catégorie d'élèves. Pour calculer la moyenne d'âge des élèves portant un certain prénom **p**, on peut utiliser la fonction suivante, qui utilise un deuxième accumulateur pour enregistrer le nombre d'élèves pris en compte dans la moyenne.

```
def age_moyen_de(p, a, eleves):
  somme = 0
  nb = 0
  for e in eleves:
   if e["prénom"] == p:
     somme += a - e["année"]
     nb += 1
 return somme / nb
```

IV Sélection de lignes

Les opérations présentées dans les deux sections précédentes analysent la table pour produire un résultat simple, sous la forme d'une unique valeur (résultat d'un test, valeur d'un attribut, valeur agrégée, etc.). Une autre opération courante, appelée *sélection*, consiste à produire une nouvelle table en extrayant de la table d'origine toutes les lignes vérifiant une certaine condition. On pourra par exemple sélectionner l'ensemble des lignes valides, ou l'ensemble des lignes relatives à des élèves nés avant 1940. Pour réaliser ce genre d'opération, on peut réutiliser la technique de construction d'un tableau par compréhension. Cette construction peut en effet intégrer une condition déterminant les éléments devant ou non être effectivement inclus dans le tableau.

```
>>> tab = [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> [i for i in tab if i \% 2 == 1]
[1, 1, 3, 5, 13, 21, 55, 89]
```

Ce même principe peut être appliqué à une table de données, c'est-à-dire à un tableau de dictionnaires, en faisant porter la condition sur les valeurs de certains attributs. Ainsi, l'instruction

```
t = [e for e in eleves if e["mois"] <= 2]
```

construit et stocke dans **t** une nouvelle table qui contient tous les dictionnaires de la table **eleves** concernant des élèves nés en janvier ou février. On parle de *sélection* des lignes vérifiant cette condition, ou encore de *filtrage*. La condition introduite dans le **if** peut être une condition arbitraire. En particulier, il est possible de combiner des conditions sur les valeurs de différents attributs et on pourra par exemple sélectionner des lignes des élèves nés avant le 15 février avec le code suivant.

```
t = [e for e in eleves if e["mois"] == 1 \
                       or e["mois"] == 2 abd e["jour"] < 15]
```

Lorsque la condition est complexe, il est même envisageable de la déléguer à une fonction écrite pour l'occasion.

```
def signe_sagittaire(e):
  return e["mois"] == 11 and e["jour"] >= 23 \
      or e["mois"] == 12 and e["jour"] <= 21

t = [e for e in eleves if signe_sagittaire(e)]
```


Info

Sélection ligne à ligne. Une manière alternative de sélectionner des lignes consiste à créer d'abord un tableau vide, auquel on ajoute une à une les lignes sélectionnées. Ainsi, à la fin du programme

```
t = []
for e in eleves:
  if date_valide(e)
    t.append(e)
```

la variable **t** désigne un tableau nouvellement créé contenant, dans l'ordre, les lignes de la table **eleves** présentant des dates valides. Rappelons que l'opération **append** modifie le tableau auquel elle est appliquée. Il faut donc éviter de l'utiliser sur les données d'origine mais uniquement sur un tableau que l'on a créé soi-même, comme le **t** qui est ici initialisé comme tableau vide. Autre rappel : écrite **t = eleves** ne suffit pas pour faire de **t** un nouveau tableau.

IV.1 Application : sélection des lignes vides

Dans un fichier tel que **eleves.csv** d'après le tableau présenté en introduction, la colonne **jour** n'est censé contenir que des nombres compris entre **1** et **31**, voir **1** et **28**, **29** ou **30** en fonction du mois et de l'année. Cependant, rien ne nous garantit que ce soit toujours le cas au moment où l'on charge le fichier, qui peut être corrompu ou invalide pour de multiples raisons. Plutôt que d'arrêter le programme à la première ligne invalide rencontrée, on peut utiliser une opération de sélection pour construire une table contenant chaque ligne valide du fichier de données et ignorant les autres. En première approximation, on peut par exemple extraire les lignes pour lesquelles à la fois le jour est compris entre **1** et **31**, le mois entre **1** et **12** et l'année antérieur à **2020**.

```
t = [e for e in eleves if 1 <= int(e["jour"]) <= 31 \
                      and 1 <= int(e["mois"]) <= 12 \
                      and int(e["année"]) <= 2020]
```

Notons que la table directement obtenue par lecture du fichier CSV selon la technique vue dans le chapitre précédent ne contient que des chaînes de caractères. On a donc besoin de la fonction **int** pour convertir des chaînes de caractères décrivant des nombres en vrais nombres Python. Cette traduction peut néanmoins elle-même échouer si la chaîne convertie ne correspond pas à l'écriture d'un nombre. Si par une malencontreuse faute de frappe le jour de naissance de Donald est enregistré dans notre fichier comme **"1à"**, l'appel de **int** échouera et le programme sera interrompu sans produire aucune table. On préférerait cependant simplement la compter comme une ligne invalide et passer à la suite. Pour tester si une chaîne de caractère **c** représente bien un nombre avant de la convertir avec **int**, on peut donc utiliser l'opération **c.isdecimal()**. Le programme suivant inclus ce test dans une fonction **est_valide** et produit enfin une table validée dans laquelle on trouve seulement les lignes valides, où les valeurs des attributs numériques ont effectivement étés traduites.

```
def est_valide(e):
  jour = e["jour"]
  mois = e["mois"]
  annee = e["année"]
  return jour.isdecimal() and 1 <= int(jour) <= 31 \
     and mois.isdecimal() and 1 <= int(mois) <= 12 \
     and annee.isdecimal() and int(annee) <= 2020

def conversion(e):
return {"prénom":e["prénom"], "jour":int(e["jour"]),
        "mois":int(e["mois"]), "année":int(e["année"]),
        "projet":e["projet"] }

table_validee = [ conversion(e) for e in table if est_valide(e) ]
```

V Sélection de lignes et colonnes

La construction de tableau par compréhension peut effectuer un calcul.

```
>>> [i * i for i in range(7)]
[0, 1, 4, 9, 16, 25, 36]
```

Cette possibilité peut être utilisée dans le cas particulier d'une table de données, par exemple pour sélectionner une colonne particulière plutôt que l'intégralité de chaque ligne. Ainsi, l'instruction

```
t = [e["prénom"] for e in eleves]
```

construit un tableau contenant les prénoms de chaque élèves de la table **eleves**. On appel cette opération une *projection*. La projection sur une colonne peut évidemment être combinée à la sélection d'un ensemble de lignes. Les prénoms des élèves nés avant le 15 février peuvent donc être obtenus avec la seul ligne suivante.

```
[e["prénom"] for e in eleves \
           if e["mois"] == 1
           or e["mois"] == 2 and e["jour"] < 15]
```

L'opération de projection est notamment utilisée pour construire une nouvelle table contenant seulement une partie des colonnes de la table d'origine. Pour cela, plutôt que d'extraire uniquement la valeur d'un attribut, on peut reconstruire un nouveau dictionnaire contenant les valeurs des attributs qui nous intéressent. Ainsi, l'instruction

```
t = [{"prénom": e["prénom"], "projet" : e["projet"]}
     for e in eleves
```

construit pour chaque ligne de la table **eleves** un n-uplet contenant un attribut **"prénom"** et un attribut **"projet"**, dont les valeurs sont exactement les valeurs des attributs de même nom dans la ligne considérée. Autrement dit, on extrait de la table **eleves** une table ne conservant que les prénoms et les projets, ou encore on effectue une projection sur ces deux colonnes. Plus généralement, On peut au passage effectuer n'importe quelle opération sur les lignes au moment de faire une sélection ou une projection et notamment produire une table contenant de nouvelles colonnes. Ainsi, l'instruction suivante crée une nouvelle table associant chaque élève à son âge à la fin de l'année 2019.

```
t = [{"prénom": e["prénom"], "âge" : 2019 - e["année"]}
     for e in eleves
```

Info

Un avant-goût des bases de données. Les opérations d'extraction présentées ici préparent à la manipulation de bases de données en classe de terminale. Ces opérations correspondent en effet de près aux requêtes **SELECT ... FROM ... WHERE ...** du langage SQL. Par exemple, notre sélection

```
[{"prénom": e["prénom"], "projet" : e["projet"]} \
 for e in eleves if e["mois"] == 1 or e["mois"] == 2 and e["jour"] < 15]
```

qui récupère les prénoms et projets des élèves nés avant le 15 février pourrait être produite en SQL avec la requête suivante.

```
SELECT e.prénom, e.projet
FROM eleves AS e
WHERE e.mois = 1 OR (e.mois = 2 AND e.jour <15)
```

Après **SELECT** apparaissent les colonnes sur lesquelles on souhaite projeter, qui correspondent à la partie principale de notre construction par compréhension (mais avec une notation allégée). Après **FROM** vient la table dans laquelle on fait la sélection, après **AS** le nom donné à la ligne en cours d'examen et après **WHERE** la condition désignant les lignes qui doivent être sélectionnées. Ces éléments correspondent donc précisément, dans l'ordre, au **in**, au **for** et au **if** de la construction par compréhension. A noter, la requête **SELECT** permet également, au delà de la simple projection, de faire quelques opérations arithmétiques simples et des opérations d'agrégation.

Savoir-faire

A retenir

Une fois un jeu de données chargé dans une table, il devient possible de les manipuler avec les opérations usuelles de Python pour en **extraire** des informations immédiates ou de nouvelles tables. On peut utiliser des **conditions** arbitraires sur l'ensemble des attributs d'une ligne pour désigner la ou les lignes d'intérêt, puis construire un résultat en utilisant les **attributs** des lignes sélectionnées.

VI Exercices

Exercice

Exercice 1

Imaginons un bon de commande représenté par une table de données dans laquelle chaque ligne correspond à un produit commandé et contient quatre attributs : la référence du produit et sa désignation (deux chaînes de caractères), le prix unitaire (un nombre décimal), la quantité commandée (un nombre entier). Voici un exemple de cette table.

réf.	désignation	prix	qté
18635	lot crayons HB	2,30	1
15223	stylo rouge	1,50	3
20112	cahier petits carreaux	3,50	2

1) Écrire une fonction **verifie_quantites** qui analyse un bon de commande et renvoie **True** si pour chaque produit commandé la quantité est bien positive.
2) Écrire une fonction **nombre_produit** qui renvoie le nombre total de produits demandé dans un bon de commande donné en argument (en ne comptant que les quantités positives).
3) Écrire une fonction **purge_commande** qui prend en paramètre un bon de commande **b** et renvoie un nouveau bon de commande dans lequel seuls les produits commandés en quantités strictement supérieures à 0 sont conservés.
4) Écrire une fonction **prix** qui renvoie le prix total d'un bon de commande après l'avoir purgé.

Exercice

Exercice 2

Dans le contexte de l'exercice précédent, on se donne en plus un dictionnaire **poids_produits** dont les clés sont les numéros de référence de tous les produits du catalogue, et les valeurs associées sont les poids exprimés en grammes.

1) Écrire un fonction **poids_commande** qui renvoie le poids total d'une commande, en supposant que les quantités sont bien toutes positives.
2) Écrire une fonction **articles_lourds** qui renvoie un nouveau bon de commande dans lequel seuls les produits dont le poids unitaire dépasse 20 grammes.

Exercice

Exercice 3

Comme pour l'exercice précédent, on se donne en plus un dictionnaire **tarifs** dont les clés sont les numéros de référence de tous les produits du catalogue, et les valeurs associées sont les prix unitaires.

1) Écrire une fonction **verifie_commande** qui analyse un bon de commande et renvoie **True** si les tarifs sur les bons.
2) Écrire une fonction **cherche_erreurs** qui analyse un bon de commande **b** et renvoie une nouvelle table contenant, pour chaque ligne de **b** erronée, la référence du produit, le prix indiqué dans le bon de commande et le prix du catalogue.

Exercice

Exercice 4

Rappelons qu'une table peut contenir des attributs indéfinis (avec la valeur **None**).

1) Écrire une fonction **nb_indefinis** qui analyse une table et renvoie le nombre d'attributs de ses lignes valant **None**.
2) Écrire une fonction **nb_lignes_incompletes** qui analyse une table et renvoie le nombre de ligne comportant au moins un attribut valant **None**.

Exercice

Exercice 5

On considère un registre de ventes d'appartements représenté par une table de données **registre** dont chaque ligne décrit un bien vendu avec quatre attributs : **lat** (latitude), **long** (longitude), **surface**, **prix**. Les deux premiers attributs ont pour valeur des nombres décimaux, et les deux derniers des nombres entiers. Voici un exemple de ce registre.

lat.	long.	surface	prix
48,6938	6,1893	91	169000
48,6907	6,1809	19	55000
48,6955	6,1811	75	176000

1) Écrire une fonction **surface_sup(s, registre)** qui renvoie le nombre d'appartements vendus dont la surface est supérieur ou égal à **s**.
2) Écrire une fonction **prix_inf(p, registre)** qui renvoie le nombre d'appartements vendus dont le prix est inférieur ou égal à **p**.
3) Écrire une fonction **surface_sup_prix_inf(s, p, registre)** qui renvoie le nombre d'appartements vendus pour lesquels à la fois la surface est supérieur à **s** et le prix est inférieur à **p**.
4) Écrire une fonction **prix_m2_max(registre)** qui calcule le prix par mètre carré le plus élevé.
5) Écrire une fonction **prix_moyen(registre)** qui calcule le prix moyen des appartements vendus.
6) Écrire un programme **prix_moyen_familial(registre)** qui calcule le prix moyen d'un appartement dont la surface est comprise entre 70 et 100 mètres carrés.