

Exercice 1 : dictionnaires et composition des espèces chimiques

Conventions

La composition d'une espèce chimique est représentée par un dictionnaire dont les clés sont les symboles atomiques et les valeurs les nombres d'atomes correspondants. Ainsi l'éthanol C_2H_6O est décrit par

```
ethanol = {"C": 2, "H": 6, "O": 1}
```

On dispose par ailleurs d'une table des masses molaires atomiques, dictionnaire associant à chaque symbole sa masse molaire exprimée en $g \cdot mol^{-1}$:

```
masses = {"H": 1.008, "C": 12.011, "N": 14.007, "O": 15.999, "S": 32.06}
```

Dans tout le sujet, une composition est supposée n'associer à chaque symbole qu'un entier strictement positif : un élément absent de l'espèce n'apparaît pas comme clé, plutôt que d'être associé à la valeur 0.

Partie A — Exploiter une composition

1. Écrire une fonction `nombre_atomes` de signature

```
def nombre_atomes(compo: dict[str, int]) -> int:
```

qui renvoie le nombre total d'atomes de l'espèce décrite par `compo`.

Correction.

```
def nombre_atomes(compo: dict[str, int]) -> int:
    """
    Renvoie le nombre total d'atomes de l'espèce de composition compo.
    compo associe à chaque symbole atomique son nombre d'occurrences.
    """
    total = 0
    for symbole in compo:
        total = total + compo[symbole]
    return total
```

Itérer sur un dictionnaire, c'est itérer sur ses *clés* : la variable `symbole` prend successivement pour valeur chacun des symboles atomiques, et non les nombres d'atomes.

2. On souhaite savoir si un élément donné entre dans la composition d'une espèce.
 - (a) Écrire une fonction `contient(compo, symbole)` renvoyant un booléen, **en parcourant** le dictionnaire `compo`.
 - (b) Combien de comparaisons cette fonction effectue-t-elle, dans le meilleur et dans le pire des cas, si `compo` possède n clés ?
 - (c) Quelle construction de Python permet d'obtenir le même résultat en une seule ligne ? Quel est alors le coût de l'opération ?

Correction.

```
def contient(compo: dict[str, int], symbole: str) -> bool:
    """
    Renvoie True si et seulement si symbole est l'un des symboles atomiques
    apparaissant dans la composition compo.
    """
    for s in compo:
        if s == symbole:
            return True
    return False
```

Le nombre de comparaisons appartient à $\llbracket 1; n \rrbracket$: une seule si le symbole cherché est la première clé rencontrée, n s'il est en dernière position ou absent. Le coût est donc linéaire en le nombre de clés.

L'expression `symbole in compo` produit le même booléen. Elle n'effectue pas de parcours : le dictionnaire est une table de hachage, la clé est convertie en un indice de case par une fonction de hachage, et seul le contenu de cette case est examiné. Le coût est constant en moyenne, ce qui constitue l'intérêt essentiel de cette structure par rapport à une liste de couples.

3. Écrire une fonction `symboles_inconnus(compo, masses)` renvoyant la liste des symboles atomiques de `compo` qui ne figurent pas dans la table `masses`. Cette liste est vide lorsque la masse molaire de l'espèce est calculable.

Correction.

```
def symboles_inconnus(compo: dict[str, int],
                      masses: dict[str, float]) -> list[str]:
    """
    Renvoie la liste des symboles atomiques apparaissant dans compo mais
    absents de la table masses. La liste renvoyee est vide si tous les
    symboles de compo sont des cles de masses.
    """
    inconnus = []
    for symbole in compo:
        if symbole not in masses:
            inconnus.append(symbole)
    return inconnus
```

Les deux dictionnaires jouent ici des rôles très différents : `compo` est parcouru, `masses` est seulement interrogé.

4. Écrire une fonction `masse_molaire(compo, masses)` renvoyant la masse molaire de l'espèce, en $\text{g} \cdot \text{mol}^{-1}$. La spécification énoncera une précondition portant sur `symboles_inconnus`.

Correction.

```
def masse_molaire(compo: dict[str, int],
                  masses: dict[str, float]) -> float:
    """
    Renvoie la masse molaire, en g/mol, de l'espece de composition compo.
    masses associe a chaque symbole atomique sa masse molaire en g/mol.
    Precondition : symboles_inconnus(compo, masses) est la liste vide.
    """
    m = 0.0
    for symbole in compo:
        m = m + compo[symbole] * masses[symbole]
    return m
```

La précondition évite d'avoir à tester l'appartenance dans la boucle : sans elle, l'accès `masses[symbole]` déclencherait une `KeyError`. Rejeter la vérification dans une fonction distincte, appelée une fois en amont, est préférable à un test répété à chaque tour de boucle.

Partie B — Construire une composition

1. Écrire une procédure `ajouter(compo, symbole, n)` qui ajoute n atomes de l'élément `symbole` à la composition `compo`, celle-ci étant modifiée en place. La clé sera créée si elle n'est pas déjà présente.

Correction.

```
def ajouter(compo: dict[str, int], symbole: str, n: int) -> None:
    """
    Ajoute n atomes de l'element symbole a la composition compo, qui est
    modifiee en place. La cle symbole est creee si elle est absente.
    Precondition : n est un entier strictement positif.
    """
    if symbole in compo:
        compo[symbole] = compo[symbole] + n
    else:
        compo[symbole] = n
```

Ce motif — tester la présence de la clé, l'incrémenter ou l'initialiser — est le schéma fondamental du comptage par dictionnaire ; il se retrouve à l'identique dans le dénombrement des occurrences des mots d'un texte. L'annotation `-> None` signale une procédure : il n'y a rien à renvoyer, la modification du dictionnaire passé en argument étant visible par l'appelant.

2. Écrire une fonction `composition(atomes)` qui, à partir d'une liste de symboles atomiques telle que `["H", "O", "H"]`, renvoie la composition correspondante, ici `{"H": 2, "O": 1}`. On réutilisera la procédure précédente.

Correction.

```
def composition(atomes: list[str]) -> dict[str, int]:
    """
    Renvoie la composition de l'espece obtenue en assemblant les atomes de
    la liste atomes, dont chaque element est un symbole atomique.
    """
    compo = {}
    for symbole in atomes:
        ajouter(compo, symbole, 1)
    return compo
```

Le dictionnaire est ici construit par accumulation, à partir du dictionnaire vide `{}` — et non `()` ni `[]`. L'ordre des clés du résultat est celui des premières apparitions dans la liste, mais aucune question ne doit en dépendre.

3. Écrire une fonction `fusion(c1, c2)` renvoyant la composition de l'assemblage de deux espèces. Les dictionnaires `c1` et `c2` devront être intacts après l'appel.

Correction.

```
def fusion(c1: dict[str, int], c2: dict[str, int]) -> dict[str, int]:
    """
    Renvoie la composition de l'assemblage des especes de compositions c1
    et c2. Les dictionnaires c1 et c2 ne sont pas modifies.
    """
    resultat = {}
    for symbole in c1:
        resultat[symbole] = c1[symbole]
    for symbole in c2:
        ajouter(resultat, symbole, c2[symbole])
    return resultat
```

L'affectation `resultat = c1` ne crée aucun dictionnaire : elle donne un second nom au même objet en mémoire. Toute modification ultérieure de `resultat` altérerait donc `c1`, et l'appelant retrouverait

son argument corrompu. La première boucle réalise une copie effective, que l'on peut aussi obtenir par `dict(c1)`.