Corrigé Sujet 1 épreuve pratique NSI

Michel Billaud (michel.billaud@laposte.net)

30 janvier 2022

Table des matières

1 Licence

2 Le sujet

Se trouve sur la page https://eduscol.education.fr/2661/banque-des-epreuves-pratiques-de-specialite-nsi, dans https://eduscol.education.fr/document/33178/download

3 Exercice 1 : nombre d'occurrences d'un caractère dans un mot

3.1 La question

> **Question** : Écrire une fonction `recherche` qui prend en paramètres `caractere`, un caractère, et `mot`, une chaîne de caractères, et qui renvoie le nombre d'occurrences de `caractere` dans `mot`, c'est-à-dire le nombre de fois où `caractere` apparaît dans `mot`.

Clairement, le souhait est de normaliser le choix des noms - ça facilite les corrections -, on ne laisse pas d'autre possibilité que de commencer par

```
def recherche(caractere, mot):
    ...
```

Critique : le nom imposé `recherche` est assez malheureux. L'usage est que le nom d'une fonction décrive le résultat qu'elle calcule. "Recherche", ça ne dit pas ce qui est recherché. La première position du caractère dans le mot ? juste un booléen qui dit si on l'a trouvé ? …

Bref, un nom comme `nombre_occurrences` ou `nb_occurrences` aurait été plus approprié.

3.2 Solution 1 : itération et comptage

Une solution simple, dans le style impératif, est de

- regarder le mot caractère par caractère, avec une boucle (`for c in mot: ...`)
- comparer avec le caractère cherché -`if c == caractere:`
- et si ça colle, ajouter 1 à un compteur (variable locale de la fonction)
  - initialisé à 0 avant la boucle
  - et dont la valeur sera retournée comme résultat de la fonction

```
def recherche(caractere, mot):
    nb = 0                      # nombre d'occurrences, 0 au départ
    for c in mot:               # pour chaque caractere c du mot
        if c == caractere:      #   - est-ce le bon ?
            nb += 1             #   - oui ! un de plus !
    return nb                   # et voila le total.
```

Je suppose que c'est la solution à laquelle les correcteurs s'attendent le plus. Mais on peut faire autrement.

3.3 Solution 2 : sous-liste en intension et comptage

Une chaîne de caractères est aussi une liste de caractères. À ce titre,

- on peut en extraire facilement la sous-liste de caractères qui nous plaisent (qui sont égaux à ceux qu'on cherche), par `[ c in mot if c == caractere ]`
- et il ne reste plus qu'à retourner la longueur(`len`) de cette liste.

```python
def recherche(caractere, mot):
    return len([c for c in mot if c == caractere])
```

3.4 Solution 3 : la méthode `count` des chaînes

Quand on connait le langage Python, on sait que les chaînes sont des séquences qui ont une méthode `count` qui retourne le nombre d'occurrences d'une sous-séquence.

```python
def recherche3(caractere, mot):
    return mot.count(caractere)
```

bref, c'est tout fait.

3.5 Solution 4 : récursion structurelle

Sachant que

- une chaîne est vide ou pas,
- si elle est vide, elle contient 0 occurrence de quoi que ce soit,
- si elle n'est pas vide, elle est composée d'un premier caractère et d'une suite ;
- ce caractère est celui qu'on cherche, ou pas ;

on peut s'orienter vers une solution récursive

```python
def recherche(caractere, mot):
    if len(mot) == 0:
        return 0
    else:
        premier = mot[0]
        suite = mot[1:]
        if premier == caractere:
            return 1 + recherche(caractere, suite)
        else:
            return recherche(caractere, suite)
```

qu'on peut présenter de diverses façons équivalentes, plus ou moins détaillées, par exemple en utilisant seulement une variable pour le nombre d'occurrences qui figurent dans la suite :

```python
def recherche(caractere, mot):
    if len(mot) == 0:
        return 0
    else:
        nb_occ_suite = recherche(mot[1:])
        if mot[0] == caractere:
            return 1 + nb_occ_suite
        else:
            return nb_occ_suite
```

ou encore, en jonglant avec les expressions conditionnelles de Python

```
def recherche (caractere, mot):
    return 0 if len(mot) == 0 else (
           recherche(caractere, mot[1:])
              + (1 if caractere == mot[0] else 0))
```

qui se lit :

- retourner 0 si le mot est vide
- ou alors le nombre d'occurrences du caractère dans la suite, avec 1 de plus si le premier caractère du mot était le bon.

4 Exercice 2 : rendu de monnaie

4.1 La question

Un bout de programme Python à trous, à compléter

```
Pieces = [100,50,20,10,5,2,1]

def rendu_glouton(arendre, solution=[], i=0):
    if arendre == 0:
        return ...                                  # 1
    p = pieces[i]                                   # BUG
    if p <= ... :                                   # 2
        solution.append(...)                        # 3
        return rendu_glouton(arendre - p, solution, i)
    else :
        return rendu_glouton(arendre, solution, ...)  # 4
```

comportement attendu

```
>>>rendu_glouton_r(68,[],0)
[50, 10, 5, 2, 1]
>>>rendu_glouton_r(291,[],0)
[100, 100, 50, 20, 20, 1]
```

Critiques :

- Erreur dans le sujet. Le nom de la variable globale `Pieces` commence par une majuscule, il est utilisé sans.
- Les concepteurs du sujet ont du mal à respecter les conventions de programmation, qui disent que les morceaux d'un nom (de variable, de méthode) sont séparés par des "underscores". Donc `a_rendre`, et pas `arendre`.
- Et le nom d'une constante globale (les pièces) devrait être en majuscule (`PIECES`).

À ces considérations syntaxiques, dont les rédacteurs du sujet semblent ignorer autant l'existence https://www.python.org/dev/peps/pep-0008 que l'importance (y compris pédagogique), s'ajoute

- Manque de cohérence : si on définit des valeurs par défaut aux paramètres, c'est pour ne pas avoir à les indiquer lors des appels. Les exemples devraient être

```
>>>rendu_glouton_r(68)
[50, 10, 5, 2, 1]
>>>rendu_glouton_r(291)
[100, 100, 50, 20, 20, 1]
```

4.2 Résolution

Il n'y a jamais que 4 trous à compléter.

Quelques observations

- l'exécution des exemples montre qu'il faut retourner une liste. Ca servira surement pour le trou numéro 1.
- au cours des appels, le second paramètre est une liste vide au départ (valeur par défaut) qui se remplit (trou 3).
- le troisième paramètre `i` est un indice dans le tableau `Pieces` (ligne du bug, initialement la première. Il faudra bien le faire avancer un jour (trou 4).

Il parait donc raisonnable

- que quand on n'a plus de monnaie à rendre (`arendre == 0`), on retourne la solution qu'on a construite ; `return solution # 1` ;
- qu'on compare la valeur d'une pièce avec la somme à rendre `if p <= arendre:  #2` pour savoir si il faut rendre cette pièce ou pas ;
- que si oui, on ajoute la pièce dans la solution `solution.append(p) # 3`.
- et que sinon, on rende la monnaie avec des pièces de valeurs plus faibles, qui sont plus loin dans la liste `return rendu_glouton(arendre, solution, i + 1) # 4`

4.3 Corrigé

```
PIECES = [100, 50, 20, 10, 5, 2, 1]

def rendu_glouton(a_rendre, solution = [], i = 0):
    if a_rendre == 0:
        return solution                                    # 1
    p = PIECES[i]
    if p <= a_rendre:                                      # 2
        solution.append(p)                                 # 3
        return rendu_glouton(a_rendre - p, solution, i)
    else :
        return rendu_glouton(a_rendre, solution, i + 1)  # 4
```