Corrigé Sujet 3 épreuve pratique NSI

Michel Billaud (michel.billaud@laposte.net)

31 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/33184/download

3 Exercice 1 : delta encoding

Sujet résumé : écrire une fonction `delta` qui, à partir d'une liste de nombres, retourner la liste des différences entre une donnée et celle qui la précède. Exemples

```
>>> delta([1000, 800, 802, 1000, 1003])
[1000, -200, 2, 198, 3]
>>> delta([42])
42
```

3.1 Résolution

On ne saurait trop remercier le rédacteur du sujet qui prend la peine de donner l'exemple d'une liste avec une seule donnée pour qu'on pense à ce cas particulier :

- la première donnée n'a pas de prédécesseur,
- et la première valeur du résultat est la première donnée.

Faut-il en faire un cas particulier dans le programme ? Les choses se simplifient si on a l'idée de considérer que

- chaque donnée a un prédécesseur
- y compris pour la première, le prédécesseur fictif O.

3.2 Solution

On s'en tire facilement avec une boucle sur la liste, et une variable qui sert à mémoriser le prédecesseur

```
derniere = 0
for donnee in liste:
   ...
   derniere = donnee
```

à chaque étape, on ajoute un élément au résultat, une liste qui est initialement vide. Cet élément est la différence entre la donnée et la précédente

```
resultat = []                   # avant la boucle
...
resultat = donnee - derniere    # dans la boucle
....
return resultat                 # à la fin
```

Here we go

```
def delta(liste):
    resultat = []
    derniere = 0
    for donnee in liste:
       resultat.append(donnee - derniere)
       derniere = donnee
    return resultat
```

4 Solution 2, boucle sur indices

On peut aussi avoir l'idée d'utiliser les indices, parce que

En remarquant que `resultat[i]` est égal à `liste[i] - liste[i-1]`, on peut aussi avoir l'idée de faire une boucle sur les indices.

Boucle qui partira à l'indice 1, puisque `resultat[0]` doit être traité comme un cas particulier (`liste[-1]` n'existe pas) :

```
def delta2(liste):
    resultat = [ liste[0] ]                    # cas particulier
    for i in range(1, len(liste)):             # de 1 à taille-1
        resultat.append(liste[i] - liste[i-1])
    return resultat
```

5 Exercice 2 : représentation infixe de l'arbre d'une expression arithmétique

Sujet résumé : Compléter le code suivant

```
def expression_infixe(e):
    s = ...                              # 1
    if e.gauche is not None:
        s = s + expression_infixe(...) # 2
    s = s + ...                          # 3
    if ... is not None:                  # 4
        s = s + ...                      # 5
    if ...:                              # 6
        return s

    return '('+ s +')'
```

5.1 Résolution

Si on reprend le sujet, un `Noeud` (classe fournie) peut représenter deux types d'objets

- les nombres, qui sont des feuilles
- l'application d'un opérateur binaire à deux sous expressions

D'après l'exemple fourni, on infère que

- dans le premier cas, la valeur du nombre est dans l'attribut `valeur` (ce qui est raisonnable) et les attributs `gauche` et `droite` sont égaux à `None` ;
- dans le second cas, `valeur` contient l'opérateur, et les noeuds des sous-expressions sont référencées par `gauche` et `droite`.

Ce qu'il faut produire :

- dans le cas d'un nombre : le nombre lui-même ;

- dans le cas d'une expression : une parenthèse ouvrante, la sous-expression de gauche, l'operateur, la sous-expression de droite, et une parenthèse fermante.

Une solution simple, directe et de bon goût serait de traduire ceci en employant la méthode `est_une_feuille` aimablement fournie

```
def expression_infixe(n):
    if n.est_une_feuille() :
        return str(n.valeur)
    else:
        return "(" + expression_infixe(n.gauche) \
               + n.valeur \
               + expression_infixe(n.droit) + ")"
```

Remarque : Le paramètre est renommé `n` parce qu'il s'agit d'un `Noeud` et que "noeud" commence par un "n". Mettre `e` suggère qu'il s'agit d'une expression, mais le sujet utilise le mot "expression" à propos de la représentation textuelle. A témoin `expression_infixe` qui retourne une chaîne. À un moment, il faut essayer d'être cohérent.

Mais le pire est à venir : puisqu'il s'agit d'un sujet d'examen, il faut se plier aux lubies de l'examinateur, et faire rentrer les belles rondeurs de notre solution dans les trous carrés mis à notre disposition.

Raisonnement :

- le `return` final se charge d'ajouter des parenthèses à `s`. On ne va donc pas le faire ailleurs. Et ça permet de conclure que `s` doit être une chaîne.
- Et on déduit que si on arrive à ce `return`, c'est que le `return` précédent s'est occupé du cas où il n'y avait pas de parenthèses : les nombres.
- pour le trou 6 on aurait donc

```
if e.est_une_feuille():              #6
    return s
```

- mais pour cela il faut que `s` contienne la chaine qui représente la valeur (voir plus loin)
- pour le trou 2, on ajoute dans une expression infixe après avoir testé `e.gauche`
- on peut donc imaginer qu'il s'agit de l'expression de gauche,

```
if e.gauche is not None:
    s = s + expression_infixe(e.gauche)    #2
```

- on a vu que `s` était une chaîne. Elle est initialisé dans le trou 1, on l'étend à droite en 2, 3 et 5, et on l'encadre à la fin.
- Donc, puisqu'en 2 on y met l'expression de gauche, qui ne sere précédée par rien sinon la parenthèse ouvrante ajoutée en bout de course, on déduit que `s` est nécessairement la chaîne vide au départ

```
s = ''                          #1
```

- le sort des trous 5 et 6 est réglé par symétrie

```
if e.droit is not None:                    # 4
        s = s + expression_infixe(e.droit)   # 5
```

- ne reste plus qu'à régler que le cas de la valeur dans le trou 3. Or il y a un petit problème : avec la représentation choisie pour les données, l'attribut `valeur` contient soit un nombre, soit une chaine. Et on ne peut pas concaténer un nombre à `s`. Il faut donc une conversion

```
s = s + str(e.valeur)                      # 3
```

Et donc voila :

```
def expression_infixe(e):
    s = ''                                 #1
    if e.gauche is not None:
        s = s + expression_infixe(e.gauche)  #2
    s = s + str(e.valeur)                  #3
    if e.droit is not None:                #4
        s = s + expression_infixe(e.droit)   #5
    if e.est_une_feuille():                #6
        return s
    return '('+ s +')'
```

5.2 Critique de l'exercice : la méthode `__str__`

Croyant certainement bien faire, le rédacteur du sujet a fourni une méthode `__str__` qui fabrique une représentation sous forme de chaine de la valeur d'un noeud, et permettrait de remplir le trou 3 ainsi

```
    s = s + e                          #3
```

La méthode `__str__` est là pour fournir la représentation textuelle d'un objet. Dans la représentation textuelle d'un "noeud binaire", les sous-expressions doivent être représentées.

Dévoyer `__str__` pour en faire un accesseur/convertisseur de type pour un attribut en particulier, c'est une idée tordue.

5.3 Critique de l'exercice : mauvais usage de l'abstraction.

Quand, dans la définition d'un type par une classe, on introduit une méthode, c'est avec l'objectif de fournir une **abstraction** qui permet de manipuler l'objet sans revenir aux détails d'implémentation.

Dans `expression_infixe`, on ne devrait pas tester

```
if e.gauche is not None:
```

mais

```
if not e.est_une feuille():
```

Si on le fait, on obtient

```
def expression_infixe(e):
    s = ''                                 #1
```

```
    if not e.est_une_feuille():
        s = s + expression_infixe(e.gauche)  #2
    s = s + str(e.valeur)                    #3
    if not e.est_une_feuille():              #4
        s = s + expression_infixe(e.droit)   #5
    if e.est_une_feuille():                  #6
        return s
    return '('+ s +')
```

qu'on montre bien qu'on s'embarque dans des complications, en testant 3 fois la même chose, dans le but irraisonné d'avoir l'instruction 3 en commun.

En séparant les cas dès le départ

```
def expression_infixe(e):
    if e.est_une_feuille():
        s = str(e.valeur)                  # conversion
    else:
        s  expression_infixe(e.gauche)
        s = s + e.valeur                    # chaine
        s = s + expression_infixe(e.droit)
        s = '(' + s + ')'
    return s
```

5.4 Hors sujet : utilisation de deux classes

En utilisant 2 classes "basiques" pour implémenter les deux types d'arbres

```
class Nombre:
    def __init__(self, valeur):
        self.valeur = valeur

class Binaire:
    def __init__(self, gauche, operateur, droite):
        self.gauche = gauche
        self.operateur = operateur
        self.droite = droite
```

on a aussi une solution élégante

```
def infixe(arbre):
    if isinstance(arbre, Nombre):
        return str(arbre.valeur)
    elif isinstance(arbre, Binaire):
        return '(' + infixe(arbre.gauche)\
            + arbre.operateur \
            + infixe(arbre.droite) + ')'
    else:
        raise RuntimeError("Type invalide")

exemple = Binaire( Binaire(Nombre(3), '*', Binaire(Nombre(8),'+', Nombre(7))),
    '-',  Binaire(Nombre(2), '+', Nombre(1)))
```

```
print (infixe(exemple))
```