ALG. Listes_Chaînées. Corrigé

Classe `Cellule` de base pour les exercices
Méthodes `__init__(self)` et `__str__(self)`

```
# Définit un élément d'une liste chaînée
class Cellule :
    def __init__(self,val,suiv) :
        self.valeur = val
        self.suivante = suiv

    # Affichage des valeurs de la liste
    def __str__(self) : # Permet de surcharger "print()"
        liste = self
        affichage = ""
        # Tant que l'on n'est pas en fin de liste
        while liste is not None :
            affichage += str(liste.valeur)
            # Pour éviter le ";" à la fin de la liste
            if liste.suivante is not None :
                affichage += " ; "

            liste = liste.suivante
        # Affiche les éléments séparés par un " ; "
        return affichage
```

Méthode `add_back(self)`

```
    # Ajoute un élément en fin de liste
    def add_back(self,val) :
        liste = self

        # Recherche du dernier élément
        while liste.suivante is not None :
            liste = liste.suivante

        liste.suivante = Cellule(val,None)

        return self
```

Correction de l'exercice 1 + jeu de tests

```
23 # Correction de l'exercice 1
24 def listN(n) :
25     lst = None
26     while n > 0 :
27         lst = Cellule(n,lst)
28         n-= 1
29     return lst
30
31 # Exercice 1 : jeu de tests
32 print(listN(3)) # Attendu : 1 ; 2 ; 3
33 print(listN(0)) # Attendu : None
```

```
1 ; 2 ; 3
None
```

Remarque : on pensera à tester le cas d'une liste vide (cas particulier).

Correction de l'exercice 2

```
def equal(lst1,lst2) :
    # Cas d'une liste au moins nulle
    if lst1 is None :
        return lst2 is None
    if lst2 is None :
        return lst1 is None

    while True : # boucle dite `infinie`
        # Si les valeurs sont différents : on renvoie `False`
        if lst1.valeur != lst2.valeur :
            return False

        lst1 = lst1.suivante
        lst2 = lst2.suivante

        # Attention à retester le cas d'une liste devenue nulle
        if lst1 is None :
            return lst2 is None
        if lst2 is None :
            return lst1 is None
```

Jeux de tests

```
23 lst1 = Cellule(2, Cellule(3, None))
24 lst2 = Cellule(1, None)
25 lst3 = Cellule(3, Cellule(2, None))
26 lst4 = Cellule(2, Cellule(3, None))
27 lst5 = None
28
29 print(equal(lst1,lst2)) # Attendu : False
30 print(equal(lst1,lst3)) # Attendu : False
31 print(equal(lst1,lst4)) # Attendu : True
32 print(equal(lst1,lst5)) # Attendu : False
33
34
```

```
False
False
True
False
```

Remarques :

- ***La boucle `while True` est une boucle dite infinie : très utilisée en programmation lorsque l'on est sûr qu'elle se termine (c'est le cas ici puisque les listes sont finies) ou que c'est l'utilisateur qui y mettra fin (lorsque l'on quitte un logiciel par exemple).***
- ***On note une redondance des tests de listes vides : la programmation récursive sera ici très efficace pour rendre le code plus clair.***

Correction de l'exercice 3

```
def concat(lst1,lst2) :
    # Si la liste de départ est vide
    if lst1 is None :
        return lst2

    # On part de l1
    lst3 = lst1

    # On ajoute ensuite les éléments de lst2
    while lst2 is not None  :
        lst3.add_back(lst2.valeur)
        lst2 = lst2.suivante

    return lst3
```

Jeux de tests

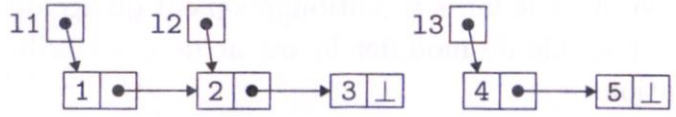
```
17 l2 = Cellule(2, Cellule(3, None))
18 l1 = Cellule(1, l2)
19 l3 = Cellule(4, Cellule(5, None))
20 print(l1) # Attendu : 1 ; 2 ; 3
21 print(l2) # Attendu : 2 ; 3
22 l4 = concat(l1,l3)
23 print(l4) # Attendu : 1 ; 2 ; 3 ; 4 ; 5
24 print(l1) # Attendu : 1 ; 2 ; 3
25 print(l2) # Attendu : 2 ; 3
```

```
1 ; 2 ; 3
2 ; 3
1 ; 2 ; 3 ; 4 ; 5
1 ; 2 ; 3 ; 4 ; 5
2 ; 3 ; 4 ; 5
```

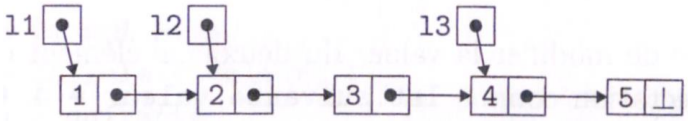
On constate que les listes l1 et l2 ont été modifiées.
Par construction, l1.suivante pointe sur l2 puis l4 vaut bien l1 + l3 soit 1 ; 2 ; 3 (l1) ; 4 ; 5 (l3).
Le problème est que le dernier élément de l1 pointe désormais sur le premier élément de l3 (à cause de l'appel à la fonction `concat(l1,l3)`) . Même situation pour la liste l2 car l1 pointe sur elle !

Voici un schéma :

Avant la création de l4



Après la création de l4



Ce sont des effets de bords, on peut faire appel à la fonction `deepcopy()` qui permet une copie en profondeur des éléments d'une liste et permet de les éviter.

Correction de l'exercice 3 (avec deepcopy)

```
from copy import deepcopy

def concat(lst1,lst2) :
    # Si la liste de départ est vide
    if lst1 is None :
        return lst2

    # On part d'une copie en profondeur l1
    lst3 = deepcopy(lst1)

    # On ajoute ensuite les éléments de lst2
    while lst2 is not None  :
        lst3.add_back(lst2.valeur)
        lst2 = lst2.suivante

    return lst3
```

Jeux de tests (avec deepcopy)

```
19 l2 = Cellule(2, Cellule(3, None))
20 l1 = Cellule(1, l2)
21 l3 = Cellule(4, Cellule(5, None))
22 print(l1) # Attendu : 1 ; 2 ; 3
23 print(l2) # Attendu : 2 ; 3
24 l4 = concat(l1,l3)
25 print(l4) # Attendu : 1 ; 2 ; 3 ; 4 ; 5
26 print(l1) # Attendu : 1 ; 2 ; 3
27 print(l2) # Attendu : 2 ; 3
```

```
1 ; 2 ; 3
2 ; 3
1 ; 2 ; 3 ; 4 ; 5
1 ; 2 ; 3
2 ; 3
```

On évitera tout de même d'abuser de la fonction `deepcopy()` car elle est coûteuse en temps d'exécution.