Python – Listes chaînées

Une première implantation simple

Nous allons commencer par représenter les listes à l'aide de couples `(tete, queue)`. La liste vide vaudra `None`.

```
l1 = (4, (3, (9, (1, None))))
```

Pour simplifier, on pourra noter `nil` à la place de `None`.

```
nil = None
```

Exercice 1 : Écrire les fonctions de l'interface suivante :

Fonction	Description
`est_vide(liste)`	Renvoie un booléen indiquant si `liste` est vide ou non.
`tete(liste)`	Renvoie la valeur du premier maillon de `liste`, qui ne doit pas être vide.
`queue(liste)`	Renvoie la liste sur laquelle pointe le premier maillon de `liste`, qui ne doit pas être vide.
`cons(valeur, liste)`	Renvoie une nouvelle liste correspondant à l'ajout de `valeur` en début de `liste`.

Dans les fonctions `tete` et `queue`, il faut rajouter une assertion pour vérifier que la liste n'est pas vide. Il est possible de rajouter un texte en écrivant :

```
assert not est_vide(liste), 'liste vide'
```

```
>>> est_vide(l1)
False
>>> est_vide(nil)
True
>>> tete(l1)
4
>>> queue(l1)
(3, (9, (1, None)))
>>> cons(2, l1)
(2, (4, (3, (9, (1, None)))))
>>> tete(nil)
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
  File "liste_chainee.py", line 10, in tete
    assert not est_vide(liste), 'liste vide'
AssertionError: liste vide
```

On peut parcourir une liste chaînée de façon itérative :

```
def approche_iterative(liste):
    l = liste
    while not est_vide(l):
        t = tete(l)
        ...
        l = queue(l)
```

Exercice 2 : En utilisant l'approche itérative, écrire chacune des fonctions suivantes :

Fonction	Description
`longueur(liste)`	Renvoie le nombre d'éléments de `liste`. La longueur de la liste vide est de 0.
`appartient(v, liste)`	Renvoie un booléen indiquant si `v` est dans `liste`
`n_ieme(i, liste)`	Renvoie la valeur de l'élément d'indice `i` de `liste` s'il existe et sinon une erreur est provoquée par une assertion, avec le message `'indice non trouvé'`

```
>>> longueur(nil)
0
>>> longueur(l1)
4
>>> appartient(1, l1)
True
>>> appartient(4, l1)
True
>>> appartient(7, l1)
False
>>> appartient(2, nil)
False
>>> n_ieme(0, l1)
4
>>> n_ieme(3, l1)
1
```

Exercice 3 : Écrire la fonction `compter(val, liste)` qui renvoie le nombre de fois où `val` apparaît dans `liste`.

```
>>> compter(3, l1)
1
>>> compter(79, l1)
0
>>> compter(3, nil)
0
>>> compter(5, cons(5, cons(2, cons(5, nil))))
2
```

Exercice 4 : Écrire la fonction `somme(liste)` qui renvoie la somme des valeurs de `liste`. Si la liste est vide, la somme est nulle.

```
>>> somme(l1)
17
>>> somme(nil)
0
>>> somme(cons(5, cons(2, cons(5, nil))))
12
```

Exercice 5 : Écrire la fonction `maximum(liste)` qui renvoie la plus grande valeur de `liste`. La liste ne doit pas être vide. Attention, il faut modifier un peu la structure de la boucle par rapport aux fonctions précédentes.

```
>>> maximum(l1)
9
>>> maximum(nil)
Traceback (most recent call last):
  ...
    assert not est_vide(liste), 'liste vide'
AssertionError: liste vide
>>> maximum(cons(-5, cons(-2, cons(-5, nil))))
-2
```

Une autre approche

La structure des listes chaînées permet d'utiliser une autre approche, qu'on appelle **récursive**. L'idée c'est que la fonction s'appelle elle-même sur la queue de la liste afin d'obtenir le sous-résultat et d'en déduire le résultat final.
Par exemple, on peut définir la longueur d'une liste comme étant 0 si elle est vide et sinon, c'est 1 plus la longueur du reste de la liste. On peut l'écrire ainsi :

```
def longueur(liste):
    if est_vide(liste):
        return 0
    else:
        return 1 + longueur(queue(liste))
```

De façon générale, on peut résumer cette approche ainsi :

```
def approche_recursive(liste):
    if est_vide(liste):
        ...
    else:
        quelque chose avec tete(liste)
        ...
        appel de approche_recursive(queue(liste))
```

Exercice 6 : Compléter la fonction récursive `appartient(val, liste)` qui renvoie un booléen indiquant si `val` appartient à `liste`.

```
def appartient(val, liste):
    if est_vide(liste):
        return ...
    elif ...: # Quelques chose avec tete(liste)
        return ...
    else:
        return appartient(..., ...)
```

Exercice 7 : Compléter la fonction récursive `n_ieme(i, liste)` qui renvoie l'élément en position `i` dans `liste` s'il existe. Sinon, une assertion provoque une erreur.

```
def n_ieme(i, liste):
    assert not est_vide(liste), 'indice non trouvé'
    if i == 0:
        return ...
    else:
        return n_ieme(..., queue(liste))
```

Exercice 8 : Écrire une version récursive de `compter(val, liste)`.

Exercice 9 : Écrire une version récursive de `somme(liste)`.

Exercice 10 : Écrire une version récursive de `maximum(liste)`.

Pour aller plus loin

Pour chacune des fonctions suivantes, vous pourrez utiliser la méthode de votre choix, même si l'approche récursive est généralement plus simple.

Exercice 11 : Écrire une fonction `copie(liste)` qui renvoie une nouvelle liste identique à `liste`.

```
>>> copie(nil) # None ne s'affiche pas
>>> copie(l1)
(4, (3, (9, (1, None))))
```

Exercice 12 : Écrire une fonction `insere_fin(val, liste)` qui renvoie une nouvelle liste qui correspond à `liste` avec `val` inséré à la fin.

```
>>> insere_fin(4, nil)
(4, nil)
>>> insere_fin(7, l1)
(4, (3, (9, (1, (7, None)))))
```

Exercice 13 : Écrire une fonction `insere_apres(val, i, liste)` renvoie une nouvelle liste qui correspond à `liste` avec `val` inséré après la `i`-ième valeur. S'il y a moins de `i` valeurs, il faut renvoyer une erreur avec une assertion.

```
>>> insere_apres(7, 0, l1)
(7, (4, (3, (9, (1, None)))))
>>> insere_apres(7, 2, l1)
(4, (3, (9, (7, (1, None)))))
```

Exercice 14 : Écrire une fonction `efface(val, liste)` qui renvoie une nouvelle liste qui correspond à `liste` où toutes les occurrences de `val` ont été enlevées.

```
>>> efface(3, l1)
(4, (9, (1, None)))
>>> efface(5, cons(5, cons(2, cons(5, nil))))
(2, None)
>>> efface(7, cons(5, cons(2, cons(5, nil))))
(5, (2, (5, nil)))
```

Exercice 15 : Écrire une fonction `concat(liste1, liste2)` qui renvoie une nouvelle liste dont les premiers éléments sont identique à ceux de `liste1` sauf que l'élément qui correspond au dernier de `liste1` pointe sur `liste2`. Ainsi, `concat(liste1, nil)` est équivalent à `copie(liste1)` et `concat(nil, liste2)` est exactement `liste2`.

```
>>> concat(l1, l1)
(4, (3, (9, (1, (4, (3, (9, (1, None))))))))
>>> concat(cons(5, cons(3, nil)), cons(7, nil))
(5, (3, (7, None)))
```

Exercice 16 : Écrire une fonction `texte(liste)` qui renvoie un texte correspondant à la liste sous la forme `val1::val2::...::nil`. On rappelle que **`str`**`(truc)` permet de convertir `truc` en texte.

```
>>> texte(nil)
'nil'
>>> texte(l1)
'4::3::9::1::nil'
```

Exercice 17 : Écrire une fonction `renverse(liste)` qui renvoie une nouvelle liste avec les éléments de `liste` dans l'ordre inverse. L'approche itérative est plus simple pour cette fonction. Si vous voulez faire une approche récursive, il faut faire rajouter un paramètre supplémentaire qui correspond à la liste déjà créée et qui est initialisée à `nil`.

```
>>> renverse(l1)
(1, (9, (3, (4, None))))
```