Paradigmes de programmation : programmation fonctionnelle

I : Généralités sur les paradigmes de programmation

Plus de 2500 langages de programmation ont été créés depuis 1950 (Wikipédia) et de nouveaux langages continuent d'être créés régulièrement. Cette multiplicité s'explique par la diversité des problématiques, contraintes, besoins ou critères auxquels doivent répondre les langages de programmation. Ces critères étant parfois contradictoires entre eux, il n'existe pas de "meilleur" langage de programmation. Voici une liste de quelques critères :

- expressivité : pour simplifier l'écriture des programmes grâce à de nombreuses instructions de haut niveau,
- lisibilité : pour faciliter la lecture et le raisonnement sur les programmes,
- efficacité : pour écrire des programmes s'exécutant rapidement grâce à une génération de code optimisé,
- portabilité : pour que les programmes soient facilement exécutables sur plusieurs OS ou architectures,
- sureté : pour aider à l'écriture de programmes corrects sans bugs,
- maintenabilité : pour faciliter la maintenance des programmes et logiciels

Les mêmes besoins ou contraintes conduisant souvent à des solutions similaires, on peut distinguer des familles de langages (on parle de *paradigmes de programmation*) selon les concepts qu'ils mettent en œuvre. Voici quelques paradigmes que nous avons déjà rencontrés ou que nous rencontrerons :

- paradigme *impératif* : manipulation de structures de données modifiables (tableaux, dictionnaires …) en utilisant notamment des boucles (ex : Python !),
- paradigme *évènementiel* : lors de l'occurrence d'un évènement, une fonction gestionnaire d'évènement est appelée et fait ce qui est attendu, par exemple dans le cas d'une interface homme-machine (ex : JavaScript),
- paradigme *orienté objet* : on définit des objets qui contiennent des données mais aussi des méthodes (ex : Python permet de programmer facilement ces concepts),
- paradigme *orienté requêtes* : pour faciliter l'interrogation des bases de données (ex : SQL),
- paradigme *fonctionnel* : pour améliorer la sureté et la maintenabilité en mettant en avant la notion de fonction et en interdisant les *effets de bord*.

Il est important de comprendre que si certains langages imposent un paradigme particulier, la plupart des langages laissent le choix au programmeur qui peut utiliser simultanément plusieurs paradigmes avec un même langage. Un programmeur doit donc connaître plusieurs paradigmes et connaître leurs avantages respectifs.

II : Programmation fonctionnelle : les fonctions sont des valeurs comme les autres

On dit qu'en programmation fonctionnelle les fonctions sont des valeurs comme les autres car :
- les fonctions peuvent elles-mêmes être passées en arguments à des fonctions,
- les fonctions peuvent elles-mêmes être renvoyées par des fonctions,
- les fonctions peuvent elles-mêmes être stockées dans des structures de données (tableaux, dictionnaires).

Illustrons cela sur quelques exemples (qui montrent au passage que Python offre cet aspect du paradigme fonctionnel) :

II.1 : des fonctions peuvent être passées en argument à des fonctions : clef de tri pour les tables

```
>>> ma_table = [("Ahmed", 8, "Blois"), ("Jane", 7, "Aix"), ("Zoé", 10, "Alès")]
>>> table_triee = sorted(ma_table)
>>> table_triee
[('Ahmed', 8, 'Blois'), ('Jane', 7, 'Aix'), ('Zoé', 10, 'Alès')]

>>> def clef_de_tri_b(triplet):
        '''Renvoie la longueur de triplet[2] (triplet[2] de type str).'''
        return len(triplet[2])
>>> table_triee = sorted(ma_table, key = clef_de_tri)
>>> table_triee
[('Jane', 7, 'Aix'), ('Zoé', 10, 'Alès'), ('Ahmed', 8, 'Blois')]
```

Remarques :

II.2 : des fonctions peuvent être renvoyées par des fonctions : contraire d'une fonction booléenne

```
>>> def contraire(f):
        '''
        f est une fonction booléenne ayant un seul paramètre.
        contraire renvoie une fonction g à un seul paramètre telle que :
        - si f(x) est True alors g(x) est False
        - si f(x) est False alors g(x) est True
        '''
        def g(x):
            return not f(x)
        return g

>>> def pair(n):
        return n%2 == 0

>>> impair = contraire(pair)
>>> pair(42)
True
>>> impair(42)
False
```

Remarques :

II.3 : des fonctions peuvent être stockées dans des structures de données

```
def fois_deux(n):
    return 2 * n

def plus_quatre(n):
    return n + 4

def moins_sept(n):
    return n - 7

def transformer_lourd(n):
    if n%3 == 0:
        return fois_deux(n)
    elif n%3 == 1:
        return plus_quatre(n)
    else:
        return moins_sept(n)

def transformer_elegant(n):
    tab = [fois_deux, plus_quatre, moins_sept]
    return tab[n%3](n)
```

Remarques :

III : Structures de données immuables

On sait qu'en Python les tableaux de type `list` sont *mutables* : on peut modifier un élément du tableau sans le redéfinir en entier (il en est de même pour les dictionnaires). Cette possibilité possède un avantage : si on a un tableau de grande taille et qu'on souhaite en modifier un seul élément, cela se fait en temps constant quelle que soit la taille du tableau. En revanche il y a un inconvénient : on peut parfois effectuer une modification non désirée, ce qui peut être source de bugs.

En revanche les p-uplets de type `tuple` sont *immuables* : une fois définis, on ne peut les modifier ce qui garantit une certaine sûreté du code (il en est de même pour les chaînes de caractères). Mais si on souhaite modifier un seul élément d'un p-uplet, cela nécessite de d'abord le recopier (sauf l'élément que l'on souhaite modifier) puis de le réaffecter à lui-même :

```
>>> tab = ['O' for i in range(500000)]
>>> tab[42] = 'X'

>>> puplet = tuple('O' for i in range(500000))
>>> puplet = tuple(puplet[i] if i != 42 else 'X' for i in range(len(puplet)))
```

Remarques :

Il convient de noter que si les tableaux (de type `list`) sont mutables, Python offre parfois la possibilité de les muter ou pas et laisse le choix au programmeur, par exemple pour les deux méthodes de tri intégrées à Python :

```
>>> tab = [13, 4, 9, 21, 19]
>>> tab_trie = sorted(tab)
>>> tab
[13, 4, 9, 21, 19]
>>> tab_trie
[4, 9, 13, 19, 21]
```

```
>>> tab = [13, 4, 9, 21, 19]
>>> tab.sort()
>>> tab
[4, 9, 13, 19, 21]
```

Remarques :

La programmation fonctionnelle se caractérise par le fait de ne pas exploiter le caractère mutable des structures de données (alors que le paradigme impératif exploite au contraire cette possibilité). En fait, ce qui importe pour respecter le paradigme fonctionnel est de respecter l'idée suivante :

"Quand on applique deux fois la même fonction aux mêmes arguments, on obtient le même résultat"

Conséquence 1 : pour respecter le paradigme fonctionnel on s'interdit d'utiliser des fonctions qui mutent des données (pas *d'effets de bords*). Un des deux exemples donnés ci-dessous crée un effet de bord qui enfreint le paradigme fonctionnel et constitue une source d'erreur fréquente. La différence entre ces deux exemples est donc à connaître :

```
>>> def plus_six(x):
        '''renvoie x augmenté de 6'''
        x = x + 6
        return x
>>> n = 7
>>> p = plus_six(n)
>>> p
13
>>> n
7
```

```
>>> def complete_avec_six(tab):
        '''renvoie tab avec l'élément 6 ajouté à la fin'''
        tab.append(6)
        return tab
>>> tab_1 = [1, 2, 3, 4, 5]
>>> tab_2 = complete_avec_six(tab_1)
>>> tab_2
[1, 2, 3, 4, 5, 6]
>>> tab_1
[1, 2, 3, 4, 5, 6]
```

Remarques :

On peut facilement trouver une fonction qui appliquée à l'argument `tab_1` donne deux fois un résultat différent (ce qui montre que le paradigme fonctionnel n'est pas respecté). On peut par exemple prendre la fonction …

Conséquence 2 : pour respecter le paradigme fonctionnel, on s'interdit les réaffectations. En effet …

En résumé :
Le *paradigme de la programmation fonctionnelle* repose tout d'abord sur l'idée d'un calcul qui se fait sans modifier les valeurs déjà construites mais plutôt en créant de nouvelles valeurs. Par ailleurs, comme son nom l'indique, le paradigme de la programmation fonctionnelle est centré sur les fonctions qui y sont des valeurs comme les autres pouvant être passées en arguments, renvoyées comme résultats ou encore stockées comme des données.
Un des intérêts du paradigme fonctionnel par rapport au paradigme impératif est de *privilégier la sûreté du code grâce à l'immuabilité des valeurs et/ou à l'absence d'effets de bord.*

Exercices

Exercice 1 (voir II.1) :

Ecrire une fonction `applique(f, t)` qui reçoit en arguments une fonction `f` et un tableau `t` et renvoie un nouveau tableau, de même taille, où la fonction `f` a été appliquée à chaque élément de `t`.
Par exemple si `f` est la fonction carré, `applique(f, [3, 7, 8, 9])` renvoie `[9, 49, 64, 81]`.

Exercice 2 (voir II.2) :

On considère la fonction `fonctionnelle_mystere` donnée ci-contre pour laquelle la fonction `f` passée en argument doit vérifier la précondition suivante : `f` est une fonction prenant un seul argument et renvoyant un nombre entier.

```
def fonctionnelle_mystere(f):
    def g(x):
        return f(x)*f(x)
    return g
```

- Comment décrire simplement `fonctionnelle_mystere(f)` ?
- Quelle est la valeur de `fonctionnelle_mystere(len)('chaton')` (où `len` est la méthode donnant la longueur d'un itérable) ?

Exercice 3 (voir II.2) :

Ecrire le code d'une fonction `double_impact` qui vérifie la spécification suivante :
- argument : fonction `f`

- préconditions : `f` prend en argument une chaîne de caractères et renvoie une chaîne de caractères

- valeur renvoyée : renvoie une fonction `g` prenant en argument une chaîne de caractères `s` et renvoyant une chaîne de caractères `g(s)` qui est la concaténation de `f(s)` et `f(s)`

- exemple : avec l'exemple ci-dessus, `res1` a pour valeur `'Jan-Claud'` et `res2` a pour valeur `'Jan-ClaudJan-Claud'`

```
def enlever_les_e(s):
   '''Enlève les 'e' de la chaîne de caractères s'''
    t = s.replace('e', '')
    return t

double_enleve = double_impact(enlever_les_e)

res1 = enlever_les_e('Jean-Claude')
res2 = double_enleve('Jean-Claude')
```

Exercice 4 (voir III) :

En Python l'opérateur `+` appliqué à deux tableaux de type `list` `s` et `t` renvoie une troisième tableau de type `list` qui est la concaténation de `s` et `t`.
D'après la documentation Python, la méthode `extend` utilisée ainsi : `s.extend(t)` permet quant à elle d'étendre `s` avec le contenu de `t`.
Pour accoler deux tableaux en respectant le paradigme fonctionnel, utilisez-vous l'opérateur `+` ou la méthode `extend` ?

Exercice 5 (voir III) :

Pour chacune des séquences d'instructions ou fonctions suivantes, indiquez si elle respecte ou non le paradigme fonctionnel.

```
tab = ['O', 'X', 'O', 'O', 'O', 'X', 'O']

def vise_les_ronds(k):
    if tab[k] == 'O':
        tab[k] = 'X'
        return 1
    else :
        return 0

s1 = vise_les_ronds(2)
s2 = vise_les_ronds(5)
score = s1 + S2
```

```
def mystere(x):
    '''x est un entier'''
    z = x + x
    return z

n = 5
x = mystere(n)
```

```
def mystere(x):
    '''x est un entier'''
    x = x + x
    return x

n = 5
y = mystere(n)
```

```
def mystere(x):
    '''x est un tableau'''
    z = x + x
    return z

n = [1, 2, 3]
y = mystere(n)
```

```
def mystere(x):
    '''x est un tableau'''
    x.extend(x)
    return x

n = [1, 2, 3]
y = mystere(n)
```

```
compteur = 0
def incremente():
    compteur = compteur + 1

incremente()
```