Programmation fonctionnelle

1. Historique

La programmation fonctionnelle est née en même temps que la programmation impérative.

- Le premier langage fonctionnel, le langage **LISP**, a été créé en 1958 par John McCarthy. Il a connu est fort succès, notamment avec beaucoup d'application en intelligence artificielle. Par ailleurs, ce fut un des premiers langages pour lequel fut développée la notion de machine virtuelle. C'est un langage réputé pour le nombre colossal de parenthèses qu'il requiert.
- Le langage **CAML**, apparu en 1985, a été développé par l'INRIA rajoutant quelques aspects impératifs afin de faciliter certaines tâches de programmation. Un peu plus tard est apparu OOCAML, une version orientée objet de CAML.
- Le langage **HASKELL** (1990) est un langage fonctionnellement très pur, très efficace à l'exécution, et beaucoup moins lourd à utiliser que LISP. Par ailleurs, une bibliothèque très riche est disponible.

2. La programmation fonctionnelle, c'est quoi ?

2.1. Premier exemple

Soit le programme suivant :

```
i = 6

def fct():
    if i > 5:
        return True
    else:
        return False

print(fct())
```

- Si `i = 6`, `fct()` renvoie `True`.
- Si `i = 4`, `fct()` renvoie `False`.

Pour un même appel à la fonction, `fct()` peut renvoyer `True` ou `False`. Par conséquent la fonction `fct` n'est pas une fonction pure. Son résultat dépend d'une valeur extérieure (ici `i`).

Maintenant le programme ci-après :

```
def fct(i):
    if i > 5:
        return True
    else:
        return False

print(fct(6))
```

- `fct(6)` renvoie toujours `True`.
- `fct(4)` renvoie toujours `False`.

La fonction ci-dessus est une fonction pure, car la valeur renvoyée par la fonction `fct` (`True` ou `False`) dépend uniquement du paramètre passé à la fonction.

Le paradigme fonctionnel cherche à éviter au maximum les **effets de bord**, dit autrement, en programmation fonctionnelle on va éviter de modifier les valeurs associées à des variables. Pour ce faire, on va chercher au maximum à utiliser les fonctions (d'où le nom de programmation fonctionnelle), mais ces fonctions ne devront pas modifier les variables : en programmation fonctionnelle, on s'efforce de coder des fonctions qui ne modifient pas l'état courant des variables.
Les fonctions utilisées en programmation fonctionnelle sont parfois appelées "**fonction pure**" : le résultat renvoyé par une fonction pure doit uniquement dépendre des paramètres passés à la fonction et pas des valeurs externes à la fonction (elle ne doit pas non plus engendrer d'effet de bord).

2.2. Deuxième exemple

Le programme suivant respecte-t-il le paradigme fonctionnel ? Pourquoi ?

```
liste = [4, 7, 3]

def ajout(i):
    liste.append(i)
```

Non, car il y a un effet de bord. La fonction **ajout** modifie le tableau **liste**.

Le programme suivant respecte-t-il le paradigme fonctionnel ? Pourquoi ?

```
liste = [4, 7, 3]

def ajout(i, liste):
    tab = liste + [i]
    return tab
```

Oui car la fonction **`ajout`** ne modifie aucune variable. Elle crée une nouvelle liste **`tab`** à partir de la liste **`liste`** et du paramètre **`i`**. La fonction renvoie ainsi la nouvelle liste ainsi créée. La grandeur existante (**`liste`**) n'est jamais modifiée donc aucun risque d'effet de bord.

2.3. Troisième exemple

Le programme suivant respecte-t-il le paradigme fonctionnel ? Pourquoi ?

```
liste = [1, 2, 3]

def somme(liste):
    s = 0
    for element in liste:
        s = s + element
    return s
```

Non, ici les variables **`s`** et **`element`** sont modifiées dans la boucle **`for`**. Une variable doit être définit une seule fois et ne pas être modifiée.
Par conséquent dans les fonctions pures, il ne peut pas y avoir de boucles **`for`** ou **`while`**. Par conséquent, elles doivent être faite par récursivité.

```
liste = [1, 2, 3]
def somme(liste):
    if len(liste) == 0:
        return 0
    else:
        return liste[0] + somme(liste[1:])
```

2.4. Le lambda calcul (λ-calcul)

Les langages fonctionnels sont des langages qui reposent sur le **λ-calcul**, crée par Church en 1925.
Le principe du **λ-calcul** consiste à considérer les fonctions comme des données comme les autres.

Par exemple, la fonction qui consiste à ajouter 1 à un nombre x s'écrit :

- En **λ-calcul** : **λx.(x + 1)**
- En mathématiques : `f(x) = x + 1`
- En Python : `lambda x: x + 1`

Tester le programme suivant :

```
f = (lambda x: x + 1)
```

Et vérifier le bon résultat.

```
>>> f(3)
4
```

Un autre exemple, soit une fonction prenant en entrée deux nombres et retournant leur somme.
À l'aide du λ-calcul, cela donne :

```
somme = (lambda x, y: x + y)
```

On peut vérifier le résultat :

```
>>> somme(2, 3)
5
```

Par conséquent on peut manipuler les fonctions comme des données. Cela implique de pouvoir les passer en paramètres à d'autres fonctions.

✍ Déterminer l'affichage obtenu avec le script ci-dessous. L'implémenter et vérifier votre solution.

```
fonction = lambda x, f: x + f(x)
print(fonction(2, lambda x: x * x))
```

3. Pour résumé

En programmation fonctionnelle :

- Il n'y a pas d'affectation (lorsqu'on écrit a = 1, a peut être remplacé par 1 tout au long du programme).
- Une fonction donne toujours le même résultat pour un ou des mêmes paramètres d'entrées.
- Sans affectations, ni boucles, les fonctions récursives sont utilisées.
- Une fonction peut prendre d'autres fonctions en paramètres (λ-calcul).

C'est une bonne pratique car :

- Cela minimise les effets de bord.
- Cela facilite le débogage.
- Cela facilite les tests unitaires des fonctions.

Source : https://pixees.fr/informatiquelycee/n_site/nsi_term_paraProg_fct.html
https://mermet.users.greyc.fr/Enseignement/EnseignementInformatiqueLycee/Havre/Bloc4/progFonctionnelle.html