Chapitre 3

Programmation fonctionnelle

3.1 Paradigme fonctionnel

Le paradigme fonctionnel est un paradigme de programmation qui reprend les principes du lambda-calcul introduit par Alonzo Church dans les années 1930. L'idée fondamentale du lambda-calcul est de considérer que les fonctions sont des données comme les autres. Ainsi, elles peuvent être par exemple passées en paramètre à d'autres fonctions. D'autres principes découlent également de la thèse de Church :

- les fonctions sont des fonctions au sens mathématique du terme : elles se contentent de renvoyer une valeur en fonction de leurs arguments ;
- il n'y a pas de notion « d'état », ni à l'extérieur des fonctions, ni dans les fonctions. Un programme n'est donc qu'une composition de fonctions.

Le paradigme fonctionnel est donc centré sur l'entrée (ou l'argument), l'action et la sortie (ou résultat).

Exemple : À gauche, un programme calculant l'aire d'un rectangle en impératif, et à droite, en fonctionnel.

```
longueur=4
largeur=3
aire=longueur*largeur
print(aire)
```

```
def aireRectangle(longueur,largeur) :
    return longueur*largeur

print(aireRectangle(4,3))
```

Exemple : Un programme faisant intervenir des fonctions imbriquées.

```
def aireRectangle(longueur,largeur) :
    return longueur*largeur

def volumeParallepipede(longueur,largeur,hauteur) :
    return hauteur*aireRectangle(longueur,largeur)

print(volumeParallepipede(4,3,2))
```

Exemple : Un programme faisant intervenir une fonction prenant en argument une autre fonction.

```
def appliquerDeuxFois(fonction,argument) :
    return fonction(fonction(argument))

def ajouterCinq(x) :
    return x+5

print(appliquerDeuxFois(ajouterCinq,10))
```

Le paradigme fonctionnel a d'abord été implanté au sein de langages dédiés, plus ou moins « purement fonctionnel ». Parmi les langages dits fonctionnels, on peut citer LISP (List Processing) en 1958 ; SML (Standard Meta Language) en 1983 ; CAML (Categorical Abstract Machine Language) en 1987, puis son extension objet OCAML ; Haskell en 1990 ; Clojure en 2007.

3.2 Effets de bord et fonction pure

On appelle **effet de bord** la modification d'un état par une fonction en dehors de son environnement local. Par exemple la modification d'une variable globale appelée en argument.

Exemple : Une fonction ayant un effet de bord par modification d'une variable donnée en argument.

```
def increment(variable,seuil) :
    global variable
    while variable<seuil :
        variable+=1

variable=1
increment(10)
print(variable)
```

Une fonction dont le résultat ne dépend que des entrées et sans effets de bord est dite **pure**, elles sont donc l'essence de la programmation fonctionnelle.

Exemple : Une fonction dont la sortie ne dépend pas que de ses entrées. Elle n'est donc pas pure.

```
X=1

def sommeX(Y) :
    return X+Y
```

En programmation fonctionnelle, les « variables » sont en réalité des constantes, on ne modifie pas leurs valeurs. Mais on peut créer de nouveaux espaces mémoires par appels récursifs. Il n'y a donc pas de compteurs ni de boucles ; elles sont remplacées par les appels récursifs de fonctions.

Exemple : Fonction donnant le factoriel d'un entier naturel de façon récursive.

```
def factoriel(entier) :
    if entier==0 :
        return 1
    else :
        return entier * factoriel(entier-1)
```

3.3 Transparence référentielle

La **transparence référentielle** est le fait de pouvoir remplacer une variable par sa valeur et inversement sans changer le résultat du programme.

Exemple : La fonction ci-dessous ne vérifie pas la transparence référentielle.

```
def etend(L1,L2) :
    # ajoute à la fin de L1 les deux derniers éléments de L2
    L1.append(L2[-2])
    L1.append(L2[-1])
    return L1

L=[1,2,3]
Le1=etend(L,L)
Le2=etend(L,[1,2,3])
```

On a `Le1=[1,2,3,2,2]` et `Le2=[1,2,3,2,3]`. En effet, `L` étant modifié une première fois par la fonction `etend` lors de son appel, il devient `[1,2,3,2]` (on lui rajoute son avant dernier élément). Puis lors de la deuxième modification, il devient `Le1=[1,2,3,2,2]` (on lui rajoute son dernier élément) ; entre les deux ajouts, le dernier élément de `L` a changé contrairement à celui de la liste `[1,2,3]` qui, elle, reste constante.

3.4 Avantages et désavantages de la programmation fonctionnelle

Le paradigme fonctionnel présentent de nombreux avantages dont certains ont fini par être intégrés dans des langages impératifs.

— Il permet de structurer avec une grande clarté le code.

— Chaque processus est isolé, ce qui rend plus facile les phases de tests et de recherches d'erreurs.

— L'absence d'affectation évite les **effets de bord**.

— On bénéficie d'une **transparence référentielle**.

— Avec des fonctions pures, le parallélisme est facilité : des processus menés parallèlement sur un même jeu de données ne risquent pas de s'affecter en les modifiant.

Toutefois, la programmation fonctionnelle peut s'avérer assez abstraite et donc difficile à concevoir.

3.5 Exercices

Exercice 3.1. Dans le programme suivant, l'expression `seuil_depasse(5)` est-elle définie en dehors de tout contexte ? La fonction est-elle pure ?

```
def seuil_depasse(x) :
    return x > seuil


seuil = 10
test = seuil_depasse(5)

seuil = 4
test = seuil_depasse(5)
```

Exercice 3.2. Parmi les fonctions suivantes, lesquelles renvoient toujours la même valeur pour des entrées identiques ? Lesquelles sont pures ?

```
def somme(x,y) :
     return x + y

def somme2(x) :
    global y
    s = x + y
    y = y - 1
    return s

def f3(x) :
    return x + randint(0,10)
```

Exercice 3.3. On considère le programme ci-dessous.

```
def incrementeur(n) :
    def f(x) : return x + n
    return f

g = incrementeur(5)
h = incrementeur(7)
```

1. Quels sont les types de `f` et `g` ?
2. Que valent `f(2)` et `g(2)` ?

Exercice 3.4. On considère le programme ci-dessous.

```
def composer(f,g) :
    def h(x) : return f(g(x))
    return h
```

1. Quels sont les types attendus pour `f` et `g` ?
2. De quel type est le résultat renvoyé par `compose` ?