Programmation fonctionnelle

Dernière mise à jour le : 19/04/2024

Introduction

La **programmation fonctionnelle** est née avec le langage LISP (List Processing) créé par John Mac Carthy en 1958. C'est une mise en oeuvre de ce qu'on appelle le *lambda-calcul*, une théorie inventée par Alonzo Church dans les années 1930 dont l'idée majeure est que tout programme informatique ou tout calcul peut s'écrire ou s'exprimer uniquement avec des *fonctions*.

Le langage LISP a eu ensuite de nombreux descendants dont Common Lisp, Scheme, Haskell, Caml puis son extension objet OcamL (qui est un langage enseigné actuellement en classe préparatoire MP2I).

Les idées introduites par ces langages ont été reprises dans la plupart des langages modernes qui offrent la possibilité, entre autres, de programmer en suivant un **paradigme fonctionnel**. En particulier, on verra qu'il est possible d'écrire en Python dans un style fonctionnel.

Exemple introductif : traitement de listes

En Lisp, les listes se notent avec des parenthèses et le traitement de listes repose sur 2 primitives `car` et `cdr` pour accéder au premier élément et au reste d'une liste. Par exemple, en Lisp, le code suivant définie une fonction récursive qui renvoie le dernier élément d'une liste.

```
(de dernier (l)
  (cond ((null (cdr l)) (car l))
        (t (dernier (cdr l)))
  )
)
```

Pour appeler la fonction sur la liste formée des nombres 2, 4, 3 et 9 on écrirait `(dernier '(2 4 3 9))` qui renverrait la valeur `9`.

On remarque la notation préfixée des opérateurs et l'usage abondant des parenthèses qui servent à la fois à structurer les données et les programmes.

Si vous souhaitez avoir une explication détaillée du code de cette fonction, lisez ce qu'en dit ChatGPT via Bing Copilot : https://copilot.microsoft.com/sl/kGV04eBTleu.

À condition de définir au préalable les fonctions primitives, on peut, en Python, définir la fonction `dernier`, selon le paradigme fonctionnel.

```python
# Fonctions primitives

def listevide():
    return []

def premier(L):
    return L[0]
```

```
def reste(L):
    return L[1:]
```

On a déjà écrit une version impérative de la fonction `dernier` :

```
def dernier(L):
    """Version impérative."""
    while reste(L) != listevide(): # tant que le reste de la liste n'est pas vide
        L = reste(L) # on passe au reste
    return premier(L) # on renvoie le premier élément de la dernière paire
```

Une version fonctionnelle de cette fonction peut s'écrire ainsi :

```
def dernier(L):
    """Version fonctionnelle."""
    return premier(L) if reste(L) == listevide() else dernier(reste(L))
```

```
>>> dernier([2, 4, 3, 9])
9
```

Exemple introductif : calcul du pgcd

Pour calculer le pgcd de deux nombres entiers positifs donnés, on peut, dans un style *impératif*, écrire une boucle qui soustrait alternativement le plus petit du plus grand jusqu'à ce qu'on obtienne 0.

```
def pgcd(a, b):
    """Version impérative."""
    while a != b:
        if a > b:
            a = a - b
        else:
            b = b - a
    return a
```

```
>>> pgcd(56, 184)
8
```

Dans un style fonctionnel, on écrit une fonction qui renvoie le résultat en précisant la valeur dans les cas simples et en se ramenant, le cas échéant, par appel récursif, à des cas plus simples.

```
def pgcd(a, b):
    """Version fonctionnelle."""
    return a if b == 0 else pgcd(b, a) if b > a else pgcd(a-b, b)
```

```
>>> pgcd(56, 184)
8
```

Le paradigme fonctionnel

L'usage de toutes ou une partie de quatre notions suivantes est caractéristique de la programmation fonctionnelle :

- la notion de **fonction** qui est évidemment centrale,
- la **composition** de fonctions, qui permet de faire deux calculs successifs (par exemple `dernier(reste(L))` applique la fonction `dernier` au résultat de l'application de la fonction `reste` à la liste `L`),
- la possibilité d'écrire des **expressions conditionnelles**, en Python sous la forme : `exp1 if cond else exp2`,
- la **récursion** qui permet de réduire un calcul complexe à un cas plus simple.

Par contre, on n'a pas utilisé :

- de variable,
- d'instruction élémentaire : affichage ou affectation,
- de boucle pour répéter des instructions
- de séquence pour enchaîner des instructions
- d'instruction conditionnelle.

Ces notions sont, quant à elles, les briques de base de la programmation impérative.

Correspondances :

- En programmation fonctionnelle, la **composition** remplace la **séquence**.
- L'écriture d'**expressions** - en particulier conditionnelles - remplace l'écriture d'**instructions**.
- La **récursion** remplace l'**itération**.

Les fonctions d'ordre supérieur

Dans un langage fonctionnel, une fonction est une expression comme une autre, qui peut donc être passée en paramètre à une autre fonction, ou être renvoyée comme résultat.

Une **fonction d'ordre supérieur** est une fonction qui prend une ou plusieurs fonctions en paramètre, ou qui renvoie une fonction (ou les deux).

Fondamentalement, cela permet de considérer un programme comme une donnée et donc de fabriquer des programmes qui fabriquent ou testent ou vérifient d'autres programmes.

On a déjà vu une tel usage, avec la fonction `sorted` qui peut prendre un paramètre - nommé `key` - une fonction calculant la clé sur laquelle effectuer le tri.

```
>>> help(sorted)

Help on built-in function sorted in module builtins:

sorted(iterable, /, *, key=None, reverse=False)
    Return a new list containing all items from the iterable in ascending order.

    A custom key function can be supplied to customize the sort order, and the
    reverse flag can be set to request the result in descending order.
```

Pour comparer deux éléments, ce sont leurs clés qui sont comparées.

```
>>> animaux = ['veau', 'vache', 'cochon', 'anaconda', 'chat', 'chien', 'ver',
               'poule', 'souris']
>>> sorted(animaux)
['anaconda', 'chat', 'chien', 'cochon', 'poule', 'souris', 'vache', 'veau', 'ver']
```

On peut utiliser la fonction `len` pour trier les chaines selon leur longueur.

```
>>> sorted(animaux, key=len)
['ver', 'veau', 'chat', 'vache', 'chien', 'poule', 'cochon', 'souris', 'anaconda']
```

Dans ce dernier exemple, la fonction `len` est passée en paramètre à la fonction `sorted`.

Si on souhaite, on peut trier la liste selon la deuxième lettre de chaque mot.

```
def deuxieme_lettre(mot):
    return mot[1]
```

```
>>> sorted(animaux, key=deuxieme_lettre)
['vache', 'veau', 'ver', 'chat', 'chien', 'anaconda', 'cochon', 'poule', 'souris']
```

La notation `lambda`

La notation `lambda` permet de définir une fonction, puis être nommée - ou pas. On utilise pour cela la syntaxe suivante :

```
lambda x: 2*x
```

qui désigne une fonction prenant en paramètre un nombre `x` et renvoyant le double de `x`.

```
>>> type(lambda x: 2*x)
<class 'function'>
```

On peut la nommer si nécessaire, puis l'utiliser, de la façon suivante :

```
>>> double = lambda x: 2*x
>>> double(16)
32
```

```
>>> addition = lambda x, y: x + y
>>> addition(2, 3)
5
```

Souvent, on va utiliser la notation `lambda` pour définir une fonction *anonyme*, c'est-à-dire sans avoir besoin de la nommer. Par exemple, une telle fonction anonyme peut aussi être fournie comme clé à la fonction `sorted` (ou toute autre fonction ayant pour paramètre une fonction).

Dans cet exemple, on trie des points selon leur distance de Manhattan au point (0,0) :

```
>>> points = [(2, 4), (3, 5), (1, 1), (7,5), (3,1), (5,0)]
>>> sorted(points, key = lambda p: p[0] + p[1])
[(1, 1), (3, 1), (5, 0), (2, 4), (3, 5), (7, 5)]
```

Les fonctions `map`, `filter` et `reduce`

Les fonctions `map`, `filter` et `reduce` sont des fonctions d'ordre supérieur communes en programmation fonctionnelle. Elle sont aussi présentes en Python, via la distribution standard ou des modules classiques.

La fonction `map`

La fonction `map` permet d'appliquer la même fonction à tous les éléments d'une liste et de construire la liste des résultats. C'est une fonction qui prend donc en paramètres :

- une fonction (celle à appliquer aux différents éléments)
- une liste contenant les éléments auxquels on va appliquer la fonction

```
>>> map(lambda x: 2*x, [1, 2, 3, 4, 5])
<map at 0x23ec3a54a48>
```

Cette fonction renvoie un *itérateur* auquel on peut appliquer la fonction `list` pour calculer effectivement la liste des valeurs sucessives :

```
>>> list(map(lambda x: 2*x, [1, 2, 3, 4, 5]))
[2, 4, 6, 8, 10]
```

```
>>> list(map(lambda x: 2**x, range(10)))
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
>>> list(map(len, ['veau', 'vache', 'cochon', 'anaconda', 'ver']))
[4, 5, 6, 8, 3]
```

La fonction `filter`

La fonction `filter` permet de tester un prédicat sur tous les éléments d'une liste et de construire la liste des éléments le vérifiant. C'est une fonction qui prend donc en paramètres :

- une fonction renvoyant un prédicat
- une liste d'éléments sur lesquels le prédicat est testé

On peut construire la liste des éléments pour lesquels le prédicat est vrai en appliquant la fonction `list` après `filter` :

```
def est_pair(n):
    return n % 2 == 0
```

```
>>> list(filter(est_pair, range(20)))
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

```
>>> list(filter(lambda c: len(c) <= 4, ['veau', 'vache', 'cochon', 'anaconda', 'ver']))
['veau', 'ver']
```

On peut bien sûr combiner les fonctions `map` et `filter` :

```
# on construit la liste des carrés pairs des entiers compris entre 0 et 5
>>> (filter(est_pair, map(lambda x: x**2, range(6))))
[0, 4, 16]
```

Néanmoins, l'usage de `map` et de `filter` peut être évité en Python grâce à l'écriture par compréhension qui effectue le même calcul en construisant la liste des résultats :

```
>>> [x**2 for x in range(6) if est_pair(x)]
[0, 4, 16]
```

La fonction `reduce`

Cette fonction est accessible via le module `functools` (documentation : https://docs.python.org/fr/3/library/functools.html).

```
from functools import reduce
```

La fonction `reduce` permet de généraliser des opérateurs à tous les éléments d'une liste.

Elle prend en paramètres :

- une fonction `f` possédant **exactement deux** paramètres
- une liste d'éléments
- (et éventuellement un paramètre optionnel qui permet de préciser la valeur à renvoyer pour une liste vide (l'élément neutre).)

La fonction `reduce` va alors appliquer la fonction `f` aux éléments de la liste, deux à deux et de façon cumulative jusqu'à ce que tous les éléments aient été visités.

Concrètement, `reduce(f, [1, 2, 3, 4, 5]))` va calculer `f(f(f(f(1, 2), 3), 4), 5)`. C'est donc une fonction qui permet de généraliser la somme, le produit, etc.

```
>>> reduce(lambda x, y: x + y, range(1, 11))  # somme des entiers de 1 à 10
55
```

```
>>> reduce(lambda x, y: x * y, range(1, 11))  # produit des entiers de 1 à 10
3628800
```

Une synthèse

Voici un exemple de programme combinant toutes ces fonctions. L'exemple est inspiré du problème 1 du projet Euler :

> Si on liste tous les entiers naturels inférieurs à 10 qui sont multiples de 3 ou de 5, on obtient 3, 5, 6 and 9. La somme des carrés de ces multiples est 151.

Le programme ci-dessous donne la réponse en une seule ligne :

```
>>> reduce(lambda m, n: m+n, map(lambda n: n*n, filter(lambda n: n%3 == 0 or n%5 == 0, range(10))))
151
```

Analysez bien cette ligne pour la comprendre. Une version itérative pour trouver la réponse aurait été bien plus longue à écrire.

Transparence référentielle

Une autre caractéristique importante de la programme fonctionnelle s'appelle la **transparence référentielle**. Il s'agit d'un principe qui prévoit que lorsqu'on applique plusieurs fois la même fonction aux mêmes arguments, alors on obtient à chaque fois le même résultat.

Ce n'est pas toujours le cas, principalement lorsque notre fonction manipule des objets *mutables* :

```
def ajouter(liste, x):
    "Version impérative"
    liste.append(x)
    return liste
```

Si on applique cette fonction sur la liste

```
>>> une_liste = [0, 1, 2, 3, 4]
```

elle ne renvoie pas toujours le même résultat, même avec des expressions identiques :

```
>>> ajouter(une_liste, 5)
[0, 1, 2, 3, 4, 5]
```

```
>>> ajouter(une_liste, 5)
[0, 1, 2, 3, 4, 5, 5]
```

Comme on le peut le voir, notre fonction `ajouter` ne respecte pas la transparence référentielle.

Par ailleurs, comme elle modifie un élément externe à la fonction (la liste `liste` passée en paramètre), on dit qu'elle possède un **effet de bord**.

```
>>> une_liste = [0, 1, 2, 3, 4]
>>> ajouter(une_liste, 5)
>>> une_liste  # est modifiée par la fonction
[0, 1, 2, 3, 4, 5]
```

La programmation fonctionnelle *pure* prévoit que les fonctions ne doivent pas avoir d'effets de bord, donc elle décourage l'utilisation de variables *mutables*. Cela n'empêche en rien d'écrire une fonction qui fait le même travail que précédemment. En effet, au lieu de modifier un objet, il suffit d'en créer un nouveau.

```
def ajouter(liste, x):
    """Version fonctionnelle."""
    return liste[:] + [x]  # liste[:] crée une copie de liste (équivalent à liste.copy())
```

> La concaténation de deux listes en Python (opération `+`) construit une nouvelle liste, caractéristique d'une approche fonctionnelle (contrairement à `.append()` caractéristique d'une approche impérative).

La nouvelle fonction ne modifie plus la liste passée en paramètre (elle n'a plus d'effet de bord) :

```
>>> une_liste = [0, 1, 2, 3, 4]
>>> ajouter(une_liste, 5)
>>> une_liste  # la liste de départ n'est pas modifiée
[0, 1, 2, 3, 4]
```

Et donc l'application de la fonction donne toujours le même résultat si on lui passe les mêmes arguments (elle respecte la transparence référentielle) :

```
>>> une_liste = [0, 1, 2, 3, 4]
>>> ajouter(une_liste, 5)
[0, 1, 2, 3, 4, 5]
>>> ajouter(une_liste, 5)
[0, 1, 2, 3, 4, 5]
```

Avec cette version fonctionnelle, on peut écrire des programmes équivalents à ceux de la version impérative. Cela suppose simplement de modifier légèrement la façon dont on les écrit. Ainsi, pour ajouter 5 puis 6 à notre liste de départ, au lieu d'écrire

```
# version impérative
>>> une_liste = [0, 1, 2, 3, 4]
>>> ajouter(une_liste, 5)
>>> ajouter(une_liste, 6)
>>> une_liste
[0, 1, 2, 3, 4, 5, 6]
```

on écrirait :

```
# version "fonctionnelle"
>>> une_liste = [0, 1, 2, 3, 4]
>>> une_autre = ajouter(une_liste, 5)
>>> encore_une_autre = ajouter(une_autre, 6)
>>> encore_une_autre
[0, 1, 2, 3, 4, 5, 6]
```

Encore mieux, pour ne pas utiliser d'affectations mais plutôt la *composition* de fonctions pour faire des calculs successifs (caractéristique d'une approche fonctionnelle), on écrirait :

```
>>> ajouter(ajouter([0, 1, 2, 3, 4], 5), 6)
[0, 1, 2, 3, 4, 5, 6]
```

Bilan

- La programmation fonctionnelle est un paradigme de programmation qui considère le calcul en tant qu'**évaluation de fonctions** mathématiques.
- En particulier, en programmation fonctionnelle, il n'y a pas :
  - de boucles : les itérations sont remplacées par la **récursion**
  - de séquences d'instructions : elles sont remplacées par la **composition** (de fonctions)
- Dans un langage fonctionnel, **une fonction est une expression comme une autre**, qui peut donc être passée en paramètre à une autre fonction, ou être renvoyée comme résultat. Une fonction qui prend en paramètre d'autres fonctions ou qui en renvoie s'appelle une *fonction d'ordre supérieur*.
- Les fonctions `map`, `filter` et `reduce` sont des fonctions d'ordre supérieur communes en programmation fonctionnelle. Elles permettent d'itérer, d'effectuer des tests et d'effectuer des calculs, donc d'écrire la plupart des programmes impératifs déjà écrits en NSI, par des évaluations de fonctions.
- La programmation fonctionnelle respecte en outre le principe de **transparence référentielle** qui prévoit qu'un programme a les mêmes effets et les mêmes sorties si on lui donne les mêmes entrées. Cela impose de fait de ne pas modifier une variable existante, donc de ne pas utiliser de variables mutables. En fonctionnel, on ne modifie pas un objet on en crée un nouveau.
- L'activité du programmeur peut sembler assez différente en programmation impérative ou fonctionnelle. Effectivement, l'ordre dans lequel on appréhende les problèmes à résoudre diffère :
  - en impératif, on doit penser la suite des ordres à appliquer pour progresser des données vers le résultat,
  - en fonctionnel, on commence par écrire l'expression du résultat à calculer.

Références

- Documents ressources de l'équipe éducative du DIU EIL, Université de Nantes, Christophe DECLERCQ (licence CC BY-NC-SA 4.0)
- Document ressource sur Eduscol : Le paradigme fonctionnel
- Articles Wikipédia : Programmation fonctionnelle, Transparence référentielle, Fonction pure

---

Germain BECKER, Lycée Mounier, ANGERS

Voir en ligne : info-mounier.fr/terminale_nsi/langages_prog/programmation-fonctionnelle