Programmation fonctionnelle

1 Les paradigmes de programmation

Le mot paradigme vient du mot grec ancien $\pi\alpha\rho\alpha\delta\epsilon\iota\gamma\mu\alpha$ qui signifie « modèle ». En informatique, ce mot a été adopté pour distinguer les différentes façons d'envisager l'écriture d'un programme.
Il existe de nombreux paradigmes de programmation :

Impératif	Structuré	Fonctionnel
Orienté objet	Concurrent	Distribué
Evénementiel	Logique	Modélisation
...		

Le paradigme *impératif* est celui qui permet de manipuler des structures de données modifiables (variabes, tableaux, etc.) en utilisant notamment des boucles (`while`, `for`, etc.).
Le paradigme *orienté objets*, avec les concepts de classe et de méthode, ont été présentés dans un chapitre dédié.
Dans ce chapitre, nous allons présenter le paradigme *fonctionnel* qui repose sur les principes suivants :

- Programmer uniquement avec des fonctions.
- Pas d'affectation.
- Programmation sans état.
- Pas de mise à jour des structures de données.
- Les fonctions sont des objets de première classe (on peut prendre en argument des fonctions et on peut renvoyer une fonction)

2 Fonctions en tant que paramètres

2.1 Fonctions passées en arguments

Les fonctions sont comme des objets en Python, elles peuvent donc être passées en argument à d'autres fonctions.
Dans l'exemple ci-dessous, nous avons créé une fonction `saluer` qui prend une autre fonction comme argument :

```
def crier(texte):
    return texte.upper()

def chuchoter(texte):
    return texte.lower()

def saluer(f):
    salutations = f("Bonjour, je suis créé par une fonction \
                    passée en argument.")
    print(salutations)
```

```
>>> saluer(crier)
BONJOUR, JE SUIS CRÉÉ PAR UNE FONCTION PASSÉE EN ARGUMENT.
>>> saluer(chuchoter)
bonjour, je suis créé par une fonction passée en argument.
```

Cette méthode est utilisée par les fonctions de tries comme la fonction `sorted` qui permet de trier les éléments d'une liste par odre croissant :

```
>>> noms = ['Yang', 'Robert', 'Tom', 'Gates']
>>> sorted(noms)
['Gates', 'Robert', 'Tom', 'Yang']
```

L'odre croissant correspond ici à l'ordre alphabétique. On peut cependant classer cette liste selon un autre ordre, par exemple la taille, pour cela il faut passer la fonction correspondante en paramètre :

```
>>> sorted(noms, key=len)
['Tom', 'Yang', 'Gates', 'Robert']
```

2.2 Fonctions anonymes

Une *fonction anonyme* est une fonction n'ayant pas de nom. Python propose une notion de fonction anonyme sous la forme d'une construction `lambda` selon la syntaxe :

```
lambda arguments: image
```

Par exemple, pour créer « à la volée » la fonction $x \to 2 * x + 1$:

```
lambda x: 2*x+1
```

On peut le lire ainsi : « Une fonction qui prend x en paramètre et renvoie $2 * x + 1$. »
Vous pouvez appliquer la fonction ci-dessus à un argument en entourant la fonction et son argument de parenthèses :

```
>>> (lambda x: 2*x+1)(5)
11
```

Une fonction lambda étant une expression, elle peut être nommée. Par conséquent, vous pouvez écrire le code précédent comme suit :

```
>>> f = lambda x: 2*x + 1
>>> f(5)
11
```

La fonction lambda ci-dessus équivaut à écrire ceci :

```
def f(x):
    return 2*x + 1
```

Les fonctions qui prennent plus d'un argument sont exprimées en lambdas Python en listant les arguments et en les séparant par une virgule (,) mais sans les entourer de parenthèses :

```
lambda x, y: x + y
```

2.3 Fonctions renvoyées comme résultats

En programmation fonctionnelle, les fonctions ne se contentent pas de pouvoir recevoir d'autres fonctions en argument, elles peuvent également renvoyer une fonction comme résultat.

Par exemple, pour calculer la dérivée f' d'une fonction f où l'on approche $f'(x)$ par son taux d'accroissement $(f(x+h)-f(x))/h$ pour une valeur de h asser petite, on peut écrire une fonction Python `derive` qui prend en paramètre une fonction `f` et *renvoie une fonction* qui approche la dérivée de `f` :

```
def derive(f):
    h = 1e-7
    return lambda x: (f(x+h) - f(x))/h
```

On peut constater que cela fonctionne bien sur une fonction dont on connait la dérivée :

```
>>> f = lambda x: x*x
>>> df = derive(f)
>>> df(3)
6.000000087880153
```

Un autre exemple est le code suivant qui crée une fonction qui peut être utilisée pour vérifier si un nombre est pair, positif ou négatif en fonction de la chaîne de caractères donnée en paramètre.

```
def creer_verificateur(s):
    if s == 'pair':
        return lambda n: n%2 == 0
    elif s == 'positif':
        return lambda n: n>=0
    elif s == 'negatif':
        return lambda n: n<0
    else:
        raise ValueError('Requête inconnue')
```

Il est utilisé ci-dessous pour créer trois fonctions qui valideront le type d'un nombre :

```
>>> f1 = creer_verificateur('pair')
>>> f2 = creer_verificateur('positif')
>>> f3 = creer_verificateur('negatif')
>>> f1(5), f2(5), f3(5)
(False, True, False)
```

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

Les fonctions `map`, `filter` et `reduce` sont les fondements de la programmation fonctionnelle.

3.1 La fonction `map`

La fonction `map` permet d'appliquer la même fonction à tous les éléments d'une liste. Elle a deux paramètres : une fonction et une liste. Elle renvoie un itérable contenant les images par la fonction des éléments de la liste initiale.
Par exemple, on peut obtenir la liste des longueurs d'une liste de mots :

```
>>> mots = ['banane','oeuf','poisson']
>>> list(map(len, mots))
[6, 4, 7]
```

ou les carrés des éléments d'une liste :

```
>>> list(map(lambda x: x * x, [2, 4, 5]))
[4, 16, 25]
```

3.2 La fonction `filter`

La fonction `filter` permet de séléctionner les éléments d'une liste vérifiant une condition donnée. Elle a deux paramètres : une fonction qui revoie un booléen (`True` ou `False`) et une liste. Elle renvoie un itérable contenant les éléments de la liste initiale qui ont pour image `True`.
Par exemple, on peut obtenir la liste des éléments pairs d'une liste de nombre :

```
>>> def est_pair(x):
        return x % 2 == 0
>>> list(filter(est_pair, [3, 4, 6, 12, 53]))
[4, 6, 12]
```

ou les éléments positifs d'une liste de nombres :

```
>>> nombres = [23, -7, 5, 42, -12]
>>> list(filter(lambda x : x > 0, nombres))
[23, 5, 42]
```

3.3 La fonction `reduce`

Contrairement à `map` et `filter` qui sont des fonctions intégrées à Python, pour utiliser `reduce`, vous devez l'importer depuis un module appelé `functools` :

```
from functools import reduce
```

`reduce` applique une fonction de deux arguments de manière cumulative aux éléments d'une liste, en commençant éventuellement par un argument initial.
Par exemple, on peut obtenir le produit des éléments d'une liste de nombre :

```
>>> reduce((lambda x, y : x * y ), [1, 2, 3, 4])
24
```