LE PARADIGME FONCTIONNEL

1. Introduction

Le paradigme fonctionnel est un paradigme de programmation qui reprend les principes du lambda-calcul introduit par 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 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) : 1958 ;
- SML (Standard Meta Language) : 1983 ;
- CAML (Categorical Abstract Machine Language) : 1987, puis son extension objet OCAML ;
- Haskell : 1990 ;
- Clojure : 2007.

Mais certains aspects du paradigme fonctionnel ont fini par être intégrés dans des langages impératifs, car ils présentent certains avantages :

- fonctions pures ;
- fonctions d'ordre supérieur ;
- lambda-expressions ;
- évaluation paresseuse.

2. 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 ;
- 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'utilise pas les notions suivantes qui sont les briques de base de la programmation impérative :

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

- d'instruction conditionnelle.

Si on veut faire une correspondances entre les deux paradigmes :

- 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.

3. Opérateur conditionnelle ternaire (inline if)

La syntaxe de l'opérateur ternaire est :

```
expression_si_vrai if condition else expression_si_faux
```

On appelle cela opérateur ternaire car il prend 3 paramètres : une condition, une expression si la condition est vrai et une expression si la condition est fausse.

```
a = int(input())
print('Pair' if a%2==0 else 'Impair')
```

```
50
Pair
13
Impair
```

4. Fonctions pures

Une **fonction pure** est une fonction qui ne modifie rien ; elle ne fait que renvoyer des valeurs en fonction de ses paramètres. Elle n'a pas d'effet de bord.

Les modifications qu'une fonction peut effectuer sur l'état du système sont appelées **effets de bord**. Un affichage à l'écran, la modification d'un des paramètres de la fonction, la modification d'une variable non locale à la fonction sont des exemples d'effet de bord.

Une fonction pure est ainsi un analogue informatique d'une fonction mathématique.

La fonction suivante a un effet de bord, la modification d'un objet référencé en paramètres :

```
def retirer_dernier(liste) :
    liste.pop()

l = [1, 2, 3]
retirer_dernier(l)
```

L'inconvénient de ce type de fonction est qu'elles complexifient énormément la compréhension du code. Une fonction pure doit renvoyer la valeur calculée sans modifier ses paramètres. Ainsi, on peut réécrire le traitement précédent de la façon suivante :

```
def retirer_dernier_pure(liste) :
    retour = liste.copy()
    retour.pop()
    return retour

l1 = [1, 2, 3]
l2 = retirer_dernier_pure(l1)
```

Dans ce dernier cas, le fait que l'appel à *retirer_dernier_pure* ne modifie par l1 est bien plus intuitif.

Pour faciliter l'écriture de fonctions pures en Python, on peut :

- utiliser au maximum des données non mutables, c'est à dire non modifiable ;
- copier systématiquement au début des fonctions les paramètres référençant des données mutables et utiliser ces copies dans la fonction.

La fonction *retirer_dernier_pure* ci-dessus est bien une fonction pure. Cependant, si on veut suivre complètement le paradigme fonctionnel, il ne faut pas utiliser une séquence d'instruction mais uniquement de la composition de fonction. On peut alors réécrire cette fonction ainsi :

```
def retirer_dernier_fonctionnelle(liste) :
    return liste[:-1]
```

L'instruction `liste[:-1]` utilise le *slicing* pour retourner la copie d'une sous-liste de `Liste`.

S'interdire les séquences d'instruction n'est pas forcément à rechercher en Python, mais essayer de n'écrire que des fonctions pures permet de limiter les risques de bugs et facilite la relecture des programmes. Il s'agit donc d'un style de programmation à privilégier.

5. 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.

Les **fonctions d'ordre supérieur** sont des fonctions qui ont au moins une des propriétés suivantes :

- elles prennent une ou plusieurs fonctions en entrée ;
- elles renvoient une fonction.

Dans l'exemple qui suit :

- La variable `a` est définie et initialisée avec la même valeur que `foo`. On dit que `a` est un alias de la fonction `foo`.
- La fonction `foo2` est une fonction d'ordre supérieur et prend en paramètre une autre fonction ;
- La fonction `foo3` est une fonction d'ordre supérieur et retourne une référence de la fonction `foo2`.

```
def foo(x):
    return x + 1
a = foo
print(a(0))

def foo2(f, x):
    return f(x)
print(foo2(foo, 10))

def foo3():
    return foo2
print(foo3()(foo, 100))
```

```
1
11
101
```

Python fourni déjà des fonctions d'ordre suppérieur, avec par exemple la fonction `sorted` qui trie une liste et peut prendre un paramètre - nommé `key` - une fonction calculant la clé sur laquelle effectuer le tri. Pour comparer deux éléments, ce sont leurs clés qui sont comparées.

```
>>>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.
```

```
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.

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

```
['ver', 'veau', 'chat', 'vache', 'chien', 'poule', 'cochon', 'souris', 'anaconda']
```

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

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

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

```
['vache', 'veau', 'ver', 'chat', 'chien', 'anaconda', 'cochon', 'p
oule', 'souris']
```

6. Les fonctions anonymes (lambda)

En Python, le mot-clé lambda est utilisé pour définir des **fonctions anonymes**, également appelées fonctions lambda.
Les fonctions lambda sont des fonctions qui ne sont pas définies avec un nom, mais qui peuvent être utilisées dans des expressions ou des fonctions.
Elles sont souvent utilisées dans des situations où une fonction simple est nécessaire pour une tâche spécifique, comme trier une liste ou filtrer des éléments.

Les fonctions lambda avec Python sont créées selon la syntaxe suivante :

```
lambda arguments : expression
```

Les arguments sont les paramètres de la fonction et l'expression est le corps de la fonction.

La fonction anonyme renvoie la valeur de l'expression évaluée comme n'importe quelle fonction Python classique. Remarquez **l'absence d'utilisation du mot-clé return**, contrairement à une fonction classique.

```
print(type(lambda x: 2*x))
```

```
<class 'function'>
```

Dans cet exemple, nous créons une fonction lambda qui prend un argument x et renvoie x au carré.
L'appel de la fonction avec 2 en argument retourne ainsi le nombre 4 (2 au carré).

```
square = lambda x : x ** 2
print(square(2))
```

```
4
```

Et dans cet exemple, on ne passe pas par une variable intermédiaire, on utilise directement l’expression.

```
Print( (lambda x : x ** 2)(2) )
```

```
4
```

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)]
```

7. Évaluation paresseuse

L’évaluation paresseuse (en anglais : lazy evaluation), appelée aussi appel par nécessité ou évaluation retardée est une technique d'implémentation des programmes récursifs pour laquelle l'évaluation d'un paramètre de fonction ne se fait pas avant que les résultats de cette évaluation ne soient réellement nécessaires. Ces résultats, une fois calculés, sont préservés pour des réutilisations ultérieures.

Voici un exemple d’implémentation d’évaluation paresseuse :

```
class Map:
    def __init__(self, l, f):
        self.l = l.copy()
        self.l_mapped = l.copy()
        self.l_calcule = [None]*len(l)
        for i in range(len(l)):
            self.l_mapped[i] = lambda : f(self.l[i])
        print("Je ferai les calculs plus tard...")

    def __getitem__(self, key):
        if self.l_calcule[key] is None :
            self.l_calcule[key] = self.l_mapped[key]()
        return self.l_calcule[key]

l = [0, 1, 2, 3, 4, 5, 5, 6, 7, 8, 9]

def carre(x):
    print("je calcule")
    return x**2

m = Map(l, carre)
print(m[9])    # Premier appel, la valeur est calculée.
print(m[9])    # Deuxième appel, la valeur a déjà été calculée.
```

```
Je ferai les calculs plus tard...
je calcule
81
81
```

8. `map` `filter` `reduce`

L'intérêt des trois fonctions qui suivent est de pouvoir effectuer des traitements sur tous les éléments d'une liste sans avoir à utiliser de boucles, qui sont interdites en programmation fonctionnelle.
(Les listes étant aussi interdites en programmation fonctionnelle, les fonctions suivantes acceptent d'autres types de structures en paramètre)

8.1. Fonction `map`

La fonction `map` est une fonction qui permet d'appliquer un traitement à tous les éléments d'une liste. Cette fonction ne modifie pas la liste de départ : elle renvoie un objet (itérable) encapsulant le résultat (le résultat n'est pas construit à l'appel) ; les valeurs sont calculées lorsqu'elles sont requises ; c'est une mise en œuvre du principe d'**évaluation paresseuse**. Le traitement est bien sûr spécifié via une fonction.

```
def carre(n) :
    print('calcul de',n,'x',n)
    return n * n

l1 = [1, 2, 3, 4]
res = map(carre, l1)
print(type(res))
```

```
<class 'map'>
```

Par la suite, les instructions suivantes donnent :

```
l2 = list(res)
print(l2)
```

```
calcul de 1 x 1
calcul de 2 x 2
calcul de 3 x 3
calcul de 4 x 4
[1, 4, 9, 16]
```

Comme on peut le voir, c'est la fonction (c'est-à-dire son nom) qui est passée en paramètre, pas son appel (il n'y a pas de parenthèses après le nom de la fonction).

8.2. Fonction `filter`

La fonction `filter` est un autre exemple de fonction d'ordre supérieur s'appliquant à des listes. Elle prend en premier paramètre une fonction à valeur booléenne appelée filtre, et une liste en deuxième paramètre. En résultat, elle renvoie un itérable ne contenant que les valeurs de la liste pour lesquels le filtre renvoie la valeur `True`.

```
def est_pair(n) :
    return n % 2 == 0
l1 = [1,3, 4, 6, 7, 9, 10]
res = filter(est_pair, l1)
l2 = list(res)
```

```
[4, 6, 10]
```

8.3. Fonction `reduce`

La fonction `reduce` du module `functools` est une fonction d'ordre supérieur qui permet d'effectuer un pliage (ou réduction) d'une liste. Un pliage consiste à itérer un calcul sur les éléments d'une liste. On peut résumer ainsi l'algorithme de la fonction `reduce` :

```
def reduce(fonction, liste, init) :
    accumulateur = init
    for element in liste :
        accumulateur = fonction(accumulateur, element)
    return accumulateur
```

Ainsi, pour calculer la somme ou le produit des éléments d'une liste, on peut utiliser `reduce` ainsi :

```
from functools import reduce

def addition(x, y) :
    return x + y

def multiplication(x, y) :
    return x * y

l1 = [1, 2, 3, 4]
somme = reduce(addition, l1, 0)
produit = reduce(multiplication, l1, 1)
print(somme, produit)
```

```
10 24
```

9. Curryfication

La **curryfication** est la transformation d'une fonction à plusieurs arguments en une fonction à un argument qui retourne une fonction sur le reste des arguments. L'opération inverse est possible et s'appelle la décurryfication.

Ici, la fonction somme prend 2 paramètres. On va la curryfier la fonction pour passer ces paramètres consécutivement. Si on ne passe pas tous les paramètres, la fonction n'est pas exécutée et nous avons toujours une fonction.

```
def somme(a, b, c):
    return a + b + c

# On curryfie somme
c_somme = lambda a: lambda b: lambda c : somme(a, b, c)
print(c_somme(1)(2)(3))
print(c_somme(1))
```

```
6
<function <lambda>.<locals>.<lambda> at 0x1327ad0>
```

Ici, on décurryfie la fonction, pour pouvoir passer les paramètres d'un seul coup.

```
nc_somme = lambda a, b, c: c_somme(a)(b)(c)
print(nc_somme(1, 2, 3))
```

```
6
```

L'utilité de la curryfication est de pouvoir découper le code et le rendre plus compréhensible grâce à la séparation des responsabilités.

```
def somme(a, b, c):
    return a + b + c

c_somme = lambda a: lambda b: lambda c : somme(a, b, c)
r = c_somme(1) # Résultat partiel, r est une fonction
r = c_somme(2) # Résultat partiel, r est une fonction
r = c_somme(3) # Résultat final, r est une valeur
print(r)
```

```
6
```

10. Transparence référentielle

La transparence référentielle est une propriété des expressions d'un langage de programmation qui fait qu'une expression peut être remplacée par sa valeur sans changer le comportement du programme.
Définition

Une expression est référentiellement transparente si elle peut être remplacée par sa valeur sans changer le comportement du programme (c'est-à-dire que le programme a les mêmes effets et les mêmes sorties pour les mêmes entrées, quel que soit son contexte d'exécution).
Une expression est référentiellement opaque si elle n'est pas référentiellement transparente.

Si toutes les fonctions impliquées dans l'expression sont pures, c'est-à-dire si elles ont toujours les mêmes valeurs de retour pour les mêmes arguments et si elles sont sans effets de bord, alors l'expression est référentiellement transparente. Mais la réciproque est fausse : une expression référentiellement transparente peut impliquer des fonctions impures.

La transparence référentielle est la pierre angulaire de la programmation fonctionnelle. Elle permet notamment la **mémoïsation** des fonctions référentiellement transparentes (c'est-à-dire la mémorisation des valeurs précédemment calculées).

11. Récursion et récursion terminale

…