tnsi_14_progr_fonctionnelle

February 20, 2023

1 Une approche de la programmation fonctionnelle

1.1 Introduction

La programmation fonctionnelle est un paradigme de programmation, basé sur l'utilisation de fonctions de type "fonctions mathématiques".
Une des premières conséquences est qu'il n'y a -en théorie- pas d'affectations : une fonction mathématique renvoie un résultat uniquement suivant ses variables d'entrée -i.e. ses paramètres-, mais ne modifie pas de variables. Notamment les effets de bords sont interdits. De même, le résultat d'une fonction ne dépend que des variables/paramètres d'entrées.
Dans la cellule de code ci-dessous, quelles sont les fonctions qui vérifient le paradigme fonctionnel ? Quelles sont celles qui ne le vérifient pas ? Pourquoi ?

[ ]:
```
def fct_1(x):
    if x > 5:
        return True
    else:
        return False

def fct_2():
    if y > 5:
        return True
    else :
        return False

def fct_3(x):
    if x > 5:
        y = y + x
    else:
        y = y - x
    return y

def fct_4(tab):
    tab.append(3)
    return tab

def fct_5(tab):
    return tab + [3]
```

```
y = 6
tab = [0, 1, 2]
```

Remarque 1 : faire le lien avec les objets muables et immuables.
Remarque 2 : en programmation fonctionnelle, on peut écrire `u = 3`, mais ici on définit la valeur de la **constante** `u`. De même si l'on écrit `u = x + y`. Par contre, `u = u + 1` n'est pas de la programmation fonctionnelle.
Remarque 3 : en contradiction avec la remarque 2, on utilise parfois des variables locales en programmation fonctionnelle. Pourquoi ? Parce que les contours de la programmation fonctionnelle sont flous, allant de la vision très stricte n'utilisant que des compositions de fonctions, avec aucun effet de bord, jusqu'à des approches plus tolérantes. Les fonctions mathématiques n'utilisent pas de boucles, mais peuvent par contre utiliser des définitions récursives (comme les suites qui ne sont que des fonctions de $\mathbb{N}$ vers $\mathbb{N}$). La récursivité remplace les boucles en programmation fonctionnelle (note pour les matheux : la récursivité correspond à la composition d'une fonction par elle-même, on reste dans le vocabulaire et le champ des fonctions).
Transformer le programme suivant, qui calcule le plus grand commun diviseur de `a` et `b` en version fonctionnelle.

[ ]:
```
def pgcd(a, b):
    while b != 0:
        reste = a % b
        a = b
        b = reste
    return a

pgcd(18513,2992)
```

1.2 Fonctions en tant que paramètres ou résultats

L'usage des fonctions comme paramètre d'une autre fonction est un standard de la programmation fonctionnelle.
Comprendre l'exemple ci-dessous et recoder en fonctionnel la fonction qui est en impératif.

[ ]:
```
def calcul(f, variable):
    return f(variable)

def carre(x):
    return x*x

def double(x):
    return 2*x

def concat(tab):
    if len(tab) == 0:
        return ""
    else :
        return str(tab[0]) + concat(tab[1:])
```

```
def somme(tab):
    somme = 0
    for i in range(len(tab)):
        somme = somme + tab[i]
    return somme

g = carre     # en fonctionnel, on peut affecter une fonction à une variable
h = double
f_c = concat
f_s = somme
print(calcul(carre, 3), calcul(double, 3), calcul(g, 3), calcul(h, 3))
print(calcul(f_c, [1,2,3,4]))
print(calcul(somme, [1,2,3,4]))
```

1.2.1 Fonctions Lambda

Dans l'exemple précédent, les fonctions carré et double n'ont pas besoin d'une définition comme on le fait usuellement. On peut utiliser les fonctions λ. Ces fonctions reposent sur le λ-calcul, inventé par Alonzo Church dans les années 1930.
La fonction mathématique $x \to 2x + 3$ s'écrit $\lambda x.(2x + 3)$ en λ-calcul, et se code `lambda x : 2*x + 3` en Python.
On peut donc écrire plus simplement les fonctions carré et double (à faire pour cette dernière) :

[ ]:
```
print(calcul(lambda x : x*x, 28))

# ou bien, mais moins utilisé, cf. remarque juste après :
carre_lambda  = lambda x : x*x
print(calcul(carre_lambda, 759))
```

Les λ-expressions (fonctions λ) sont notamment utilisées lorsque l'usage en est local, et que donc la fonction n'est pas réutilisée par ailleurs : les constructions lambda sont des fonctions anonymes.
Remarque : on peut mettre un test dans une fonction lambda, cf ci-dessous.

[ ]:
```
# La fonction suivante renvoie True si le nombre est un multiple de 5
mult_5 = lambda x : True if x % 5 == 0 else False
print(mult_5(25))
print(mult_5(26))
```

1.2.2 Fonction comme résultat d'une fonction

Comprendre et expliquer ce que fait le code ci-après.

[ ]:
```
def add_cst(cst):
    return lambda x : x + cst

add_2 = add_cst(2)
```

```
print(type(add_2), add_2)
```

Comprendre et expliquer ce que fait le code suivant.
Pour ceux qui sont rapides, écrire la fonction `compose` en tant que λ-expression : `comp = lambda ...`

[ ]:
```
def compose(f , g) :
    return lambda x : f(g(x))

f = compose (lambda x : x + 1 ,lambda x : x*x )
```

1.2.3 Pourquoi s'embêter avec un nouveau paradigme, qui de plus semble assez complexe ?

1. Parce qu'on aime bien ça (raison parfaitement valide).
2. Parce que sans effets de bord, les programmes sont bien plus simples à maintenir. La programmation est globalement plus propre. Il n'y a pas de problèmes de gestion de la mémoire comme il peut y en avoir en C par exemple.
3. Une fois habitué, certains considèrent le code comme facilement lisible, d'autant plus qu'en général une fonction est codée sur peu de lignes. Toutefois, il est important de ne pas se laisser emporter par son enthousiasme et de ne coder que des fonctions simples, comme en impératif. Notamment en ce qui concerne le λ-calcul utilisé avec les fonctions vues ci-après en complément. Certains d'entre vous, élèves, peuvent se sentir visés, à juste titre, par cette remarque !
4. Les programmes sont faciles à prouver : la même entrée donne toujours le même résultat, et le découpage en fonction permet de les tester individuellement.
5. Parce que si l'on avait commencé par ce type de programmation, c'est la programmation impérative qui paraitraît complexe (testé et confirmé).

1.3 Compléments

Remarque : les contours du programme étant flous, il semble à votre professeur que les constructions λ sont à la limite du programme, et que ce qui suit n'y est pas, hormis la construction des listes par compréhension.

En programmation fonctionnelle, on travaille souvent avec des flux de données. Ces flux sont gérés avec des **itérateurs**, pour répondre aux questions "y-t-il une donnée suivante ?" et "donne moi la donnée suivante". Ils sont crées avec des **générateurs**. Dans la suite de ce TD, on travaillera uniquement avec les objets itérables de type `list` et `dict`, i.e. avec les tableaux dynamiques et dictionnaires Python. Le parcours d'un itérable se fait avec `for élément in itérable`, comme quoi vous faites de la programmation fonctionnelle sans le savoir depuis longtemps. Pour ceux qui souhaitent en savoir plus sur itératuers et générateurs, une très courte introductione en est donnée dans le cours, avec quelques exercices.

1.3.1 `map`

La fonction `map` permet de donner les résultats d'une fonction appliquée à tous les éléments d'un itérable. La tester ci-dessous, et programmer une fonction équivalente `carte(fonction, liste)` qui a le même effet, et qui renvoie une liste. Pour ceux qui sont en avance, vous pouvez programmer

cette fonction pour qu'elle soit efficace sur une liste ou un dictionnaire, en renvoyant un objet du même type que celui passé en paramètre.

[ ]:
```
print(map(lambda x : x**2, [i for i in range(10)]))
print(list(map(lambda x : x**2, [i for i in range(10)])))

def carte(fonction, iterable):
    return

print(carte(lambda x : 3*x + 5, [0, 1, 2, 3]))
```

1.3.2 `filter`

La fonction `filter` filtre les résultats (on aurait pu s'en douter). Comprendre le code ci-dessous. Ecrire la construction correspondante de la liste résultante par compréhension (c'est également de la programmation fonctionnelle).
Exerices :
* Ecrire une instruction permettant de renvoyer les mots ne comportant pas de "e" dans une liste de mots. * Ecrire une instruction qui, étant donnée une liste d'entiers, renvoie la liste formée de la moitié des entiers pairs uniquement.

[ ]:
```
print(filter(lambda n : n%2 == 0, [i for i in range(10)]))

liste =['coucou', 'Hamlet', 'ca', 'caille', 'pas', 'trop', 'dans',
        'ton', 'bled', 'paume', '?', 'etre', 'ou', 'ne', 'pas',
        'etre', 'la', 'est', 'la', 'question']
```

1.3.3 `reduce`

La fonction `reduce` permet de renvoyer un résultat calculé à partir de tous les éléments d'un itérable. Deviner avant exécution le résultat du code ci-dessous. *Exercices* : * Ecrire une instruction permettant de savoir si tous les booléens d'une liste sont Vrais. * A l'aide de la documentation de `reduce`, et d'une réflexion sur la fonction λ utilisée, comprendre l'instruction donnant le nombre d'éléments dans une liste. https://docs.python.org/3.8/library/functools.html?highlight=reduce#functools.reduce

- Ecrire une instruction comptant le nombre de booléens Vrai dans une liste.
- Dans une λ-expression, on peut également mettre une condition sous la forme :
  `lambda variable : resultat_1 if condition1 else resultat_2`
  Ecrire une instruction donnant le maximum d'une liste en utilisant `reduce`.
- Ecrire la fonction `réduction(fonction, debut, liste)` qui reproduit l'effet de l'instruction `reduce` (on rajoute dans les paramètres un élément de début pour pouvoir initialiser le calcul).

[ ]:
```
from functools import reduce
from random import randint
print(reduce(lambda x, y : x*y, [i for i in range(1,6)]))
```

```
def reduction(fonction, debut, liste):
    return
```


frederic.mandon@ac-montpellier.fr, Lycée Jean Jaurès - Saint Clément de Rivière - France