Récursivité & Programmation fonctionnelle

Thibault PENNING

Rappels

Définition 0.1 (Fonction)**. Une fonction est un fragment de code qui rend un service déterminé.**

Une fonction permet de décomposer le code en plus petit morceau, servant une meilleure compréhension du déroulement du processus. Cette dernière permet en effet d'abstraire un ensemble d'instruction en un nom unique plus facile à comprendre.

Ainsi, dans la plupart des cas, l'appel a une fonction est *bloquante*, et l'ordinateur exécutera d'abord le contenu de la fonction, puis une fois cette dernière totalement exécutée, reprendra les instructions se situant après. Nous verrons plus en détails comment cela est réalisé dans un autre chapitre

Il convient aussi de **nommer une fonction proprement** afin qu'il soit le plus rapidement possible de comprendre son utilité. Le plus souvent ce dernier sera simplement le nom de l'action qu'il effectue, ou le lien entre l'entrée et la sortie.

En Python, nous définissons une fonction ainsi :

On utilise le mot clef `def` qui permet à l'interpréteur de savoir qu'une définition de fonction va être faite, puis nous ajoutons le nom de la fonction, suivi de parenthèses et de `:`. Le corps de la fonction doit impérativement suivre, et sa délimitation est définie par l'indentation.

```
def nom():                                                    ①
    pass                                                      ②

def salutation():
    print("Bonjour tout le monde !")
```

① Définition de l'interface de la fonction
② Corps de la fonction où sera exécutées toutes les instructions souhaitées.

L'appel d'une fonction se fait dans une instruction, en écrivant le nom de la fonction suivie de parenthèse.

```
nom()                                                    ①
salutation()                                             ②
```

① Appel la fonction `nom`. Il ne se passe rien
② Appel de la fonction `salutation`. Le terminal affiche `Bonjour tout le monde !`

Définition 0.2 (Arguments)**.** Les arguments ou paramètres d'une fonction sont des variables dont la valeur est fixée lors de l'exécution d'une fonction.

Cela permet d'obtenir un résultat différent entre 2 appels d'une fonction en changeant simplement l'entrée de cette dernière.

Le code d'une fonction devient ainsi adaptatif en fonction du contexte, et les arguments permettent de définir ce dernier.

Nous pouvons prendre comme exemple une fonction permettant d'afficher un message dans la console. Il serait ainsi possible d'écrire l'ensemble du code permettant la communication avec le système d'exploitation, gérer l'affichage des caractères, etc, dans une seule fonction, dont un argument serait uniquement la chaine à afficher. Si nous voulons afficher une chaine de caractère, nous pouvons donc appeler cette fonction et donner en variable la chaine à afficher ! Cette fonction existe d'ailleurs et s'appelle `print`.

Pour définir un argument lors de la création de la fonction, on indique facilement le nom de la variable qui va prendre la valeur de l'argument dans la parenthèse qui suit le nom de la fonction. Si on veut utiliser plusieurs arguments, on les place les un à la suite des autres, séparé par des virgules

```python
def ma_fonction(argument1, argument2):                   ①
    pass

def argumentPlusGrand(A, B):
    if A > B:
        print("A est le plus grand")
    elif B > A:
        print("B est le plus grand")
    else:
        print("A et B ont la même valeur")
```

① Définition d'une fonction `ma_fonction` prenant 2 arguments en entrée.

Les outils modernes permettent d'afficher les noms de ces arguments, il convient donc de les nommées de façon claire, de telle sorte que la définition de la fonction servent d'auto-documentation.

Pour appeler une fonction prenant un argument on utilise la syntaxe suivante :

```
ma_fonction(1, 2)                                                    ①
```

① Ici `argument1` prendra la valeur 1 et `argument2` la valeur 2.

> **Note**
>
> Il est aussi possible de donner des valeurs par défaut aux arguments. Ainsi si on oublie de donner une valeur à cet argument, la valeur de défaut seras utiliser, sinon c'est celle de l'utilisateur qui primera.
> La syntaxe est la suivante : on ajoute après le nom de l'argument, un `=` puis la valeur de défaut que la variable doit prendre.
>
> **Exemple 0.1.**
>
> ```
> def ma_fonction(argument="Bonjour"):
>     print(argument)
>
> >>> ma_fonction()
> Bonjour
>
> >>> ma_fonction("Salut")
> Salut
> ```
>
> Les arguments ayant une valeur par défaut doivent toujours se trouver après les arguments sans valeurs par défaut (pour éviter toute confusion lorsque le nombre d'arguments donné est plus faible).

Définition 0.3 (Valeur de retour)**.** Une valeur de retour est la valeur que produit une fonction.

Intuitivement, cela revient à remplacer l'appel à cette fonction par la valeur retournée, rendant l'appel à cette fonction équivalente a utilisé une valeur en dur.

On retourne une valeur grâce à la syntaxe suivante :

```
def nom():
    return ValeurARetourner
```

Lorsque l'on retourne une valeur la fonction s'arrête immédiatement, indépendamment si des instructions sont présentes par la suite.

> **i** Une fonction sans retour ?
>
> On appelle cela une procédure.
> En Python, toute fonction doit avoir une valeur de retour. Si vous ne précisez rien, la fonction retourne `None`.

Une procédure permet en général de simplifier le code, en regroupant les morceaux de code se répétant, afin d'alléger la lecture du code.

> **!** La portée des variables d'une fonction
>
> **Définition 0.4.** On appelle **portée** (*ou scope*) d'une variable l'ensemble des parties du code ayant connaissance de cette dernière.
>
> On appelle *variable à portée globale*, plus couramment **variable globale** une variable dont la portée est constituée de l'intégralité du programme.
> On appelle *variable à portée locale*, plus couramment **variable locale** une variable dont la portée n'est pas locale, i.e. il existe une portion de code qui n'a pas accès à cette variable. De manière générale toute variable définie dans une fonction est une variable locale, et tout variable défini autre part est une variable globale. Ce comportement est (et a été) sujet à changement. D'autre langage définissent les variables de boucle comme une valeur locale, par exemple.

Exemple 0.2. Ainsi dans le programme suivant :

```
a = 1
b = 2

def f(x):
    res = a*x+b
    return res
```

La variable `a`, `b` et `f` sont globales, et `x` et `res` sont locales.

Dans ce cas :

```
def f(x):
    def g(y):
        return x+y
    return g(2)
```

Ici seul `f` est globale. `g` et `x` ne sont accessibles que dans `f` et `y` seulement dans `g`.

> **Retourner plusieurs valeurs**
>
> Il est possible de retourner plusieurs valeurs en même temps avec la syntaxe suivante :
>
> ```
> def f():
>     a = 1
>     b = 2
>
>     return a, b
> ```
>
> En réalité, une seule valeur est retournée : celle d'un tuple contenant toutes les valeurs. Il est donc possible d'utiliser toutes les techniques sur les tuples comme le dépaquetage des valeurs.

Récursivité

Définition 0.1 (Fonction récursive)**.** Définir une fonction récursivement consiste en une définition d'une fonction qui fait appel à elle-même.

Une fonction définie ainsi est appelée **fonction récursive**.

Ainsi la fonction suivante est définie récursivement :

```
def f(x):
    f(x)
```

> **Note**
>
> Ici la fonction `f(x)` seras exécuter à l'infinie, ce qui peut être considéré comme un bug.

> **Important 1: Évitez les problèmes avec les définitions récursives**
>
> Afin d'éviter tous soucis dû à une fonction récursive qui ne se termine pas, le conseil suivant doit être réalisé :
> **Toute fonction définie récursivement, doit posséder un branchement (condition `if`) dans lequel une des branches fait un appel récursif, et l'autre retourne une valeur sans appel récursif.**
> L'argument demande ainsi un équivalent à un variant de boucle, qui permet l'arrêt de la fonction.

Ces fonctions peuvent en général être réduites (à quelques modifications près) sous la forme suivante :

```
def f(n, autre_arg):
    if n <= 0:
        return 0
    else:
        return f(n-1, autre_arg)
```

Les fonctions récursives sont particulièrement utiles, car elles permettent de résoudre des problèmes qui sont eux même définie récursivement.

Un des exemples les plus célèbres est de déterminer la valeur de n factoriel (noté $n!$).

On note que :

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \times (n-1)! & \text{sinon} \end{cases}$$

Cette définition est une définition courante dans de nombreux problèmes. Ici, on peut donc créer facilement une fonction factorielle en utilisant une définition récursive. On remarquera que cette définition rentre dans le cas défini par Important 1.

```
def factorielle(n):
    if n == 1:
        return 1
    else:
        return factorielle(n-1) * n
```

Remarquer le lien entre la définition théorique de la fonction factorielle et son implémentation.

Il est bien sûr possible d'utilisé plusieurs appels récursifs. Il est à noter que lors de ces appels, ces derniers sont bloquants, il faudra donc attendre le retour du premier appel de fonction pour que le deuxième appel puisse être fait.

Un exemple classique est introduit par la fonction de Fibonacci suivante :

$$F(n) = \begin{cases} 1 & \text{pour } n = 0 \vee n = 1 \\ F(n-1) + F(n-2) & \forall n \geq 2 \end{cases}$$

Son implémentation est donc la suivante :

```python
def fibo(n):
    if n == 1 or n == 2:
        return 1
    else:
        return fibo(n-1) + fibo(n-2)
```

> **Note**
>
> Le variant permettant de terminer la récursion n'est pas toujours un seul entier passez en argument. Par exemple si nous réalisons une dichotomie, entre les index a et b, le variant n'est ni a ni b mais $b - a$. De même, lors du calcul du maximum d'une liste chainée on peut considérer la taille de l'entrée comme le variant (valeur que nous ne connaissons pas a priori).

Programmation fonctionnelle

Définition 0.1 (Effet de bord)**.** Un effet de bord (ou *side-effect* en anglais) est la modification d'un état (d'une variable en général) en dehors de son référentiel local

Exemple 0.1. On peut citer comme effet de bord :

- Modification d'une variable globale dans une fonction.
- Modification d'une liste dans une fonction (dû au passage par référence).
- Écriture dans un fichier
- ...

En général la présence d'un effet de bord produira un comportement non prévu par le programmeur. Cela se traduira in fine par un bug. Dans l'idéal, il est préférable de limiter et de détecter au plus tôt que possible les effets de bord.

Introduction

> **Idée**
>
> Utilisé les fonctions comme une base pour l'écriture de tout le programme.
> Cela dérive de la vision mathématique d'un programme et d'une fonction

! Programmation impérative

La première programmation que vous avez vue est dite **impérative**. Cette dernière se base sur un ensemble d'état et les transitions entre chacune de ces dernières. Pour cela les programme se repose sur un grand nombre de variables, et des boucles ou des conditions qui font évoluer leur valeur.

Nombreux sont les avantages de la programmation fonctionnelle. On citera entre autre :

- **Plus de sécurité** (notamment car on limite les effets de bord, la programmation fonctionnelle évitant le changement d'état et préfèrent l'immuabilité).
- **Plus compréhensible**. La programmation fonctionnelle demande souvent une décomposition en petite fonction, nommé, dont on connait l'entrée et la sortie. Cela génère naturellement une décomposition du programme en petite étape dont on connait les tenants et aboutissement.
- **Maintenabilité**. Dû particulièrement au point précédent, il est habituellement plus facile de déterminé où le problème se situe et comment le corriger, ainsi que la possibilité de rajouter de nouvelles fonctionnalités en se basant sur les anciennes.

i Note

La programmation fonctionnelle reste cependant généralement assez compliqué au début, car elle est très déroutante. Ne vous en faite pas, avec de l'entrainement, elle ne vous posera plus de problème. Vous verrez alors que certains algorithmes sont plus aisés à décrire en programmation fonctionnelle qu'avec de la programmation impérative.

i Historique

Historiquement la programmation impérative s'est imposée car elle était liée à la manière dont le matériel fonctionne : un processeur et la mémoire associée n'est rien d'autre qu'une machine contenant des états et les faisant évoluer. Le langage contrôlant le processeur est donc un langage impératif.
Les premiers langages fonctionnels viennent rapidement cependant, notamment sous l'impulsion d'une recherche plus *"théorique"* de l'informatique.
Aujourd'hui beaucoup de domaine utilise la programmation fonctionnelle, en dehors de la recherche. On parle particulièrement de domaine où la sécurité est essentielle, comme le domaine bancaire ou les parties critiques de service internet.

Definitions

Définition 0.2 (Valeur de première classe)**.** Une valeur est dit de premières classes (ou *first-class citizen* en anglais) lorsqu'elle se comporte *comme n'importe quelle valeur, "comme les*

autres". Cela insinue que la valeur supporte les opérations permises par le langage. On peut citer :

- Être passé en argument
- Retourner par une fonction
- Assigné à une variable

Exemple 0.2. On peut citer comme type étant de première classe : les entiers, les flottants, les tableaux, ...

> **!** Important
>
> **En Python, une fonction est une valeur de première classe**

Les conséquences sont que :

- Une fonction peut être assignée à une variable (lors de la création d'une fonction, une variable du même nom est créé et la valeur prise permet d'exécuter la fonction).
- Une fonction peut prendre une autre fonction en argument.
- Une fonction peut en retourner une autre.

On verras par la suite une application concrète de ces conséquences.

Définition 0.3 (Fonction d'ordre supérieur)**.** Une fonction est dite d'ordre supérieur lorsqu'elle possède au moins l'une de ces propriétés :

- Elle prend une autre fonction en argument
- Elle retourne une fonction

> **i** Note
>
> On peut voir une fonction d'ordre supérieur comme une fonction qui n'agis pas seul, pour son exécution elle a besoin de d'autre fonction. Cela permet d'enchainer plusieurs bloc, comme des briques de lego, où chaque fonction est responsable d'une petite fonctionalité, et où en les enchainant les une à la suite des autres, bous obtenons un comportmeent plus complexe, repondant a notre problème.

Exemple 0.3. Créons une fonction d'ordre superieur, que nous appelerons `applique` qui prendras en entrée une fontion et un tableau et retournera un tableau, de même taille que le précédent, et donc chacune de ses valeurs est la valeurs de la liste passé dans la fonction.

```
map(f, tbl) -> [f(tbl[0]), f(tbl[1]), ..., f(tbl[n])]
```

Nous définissons la fonction ainsi :

```
from typing import Any, Callable, List
def applique(f:Callable[[Any], Any], tbl:List[Any])->List[Any]:
    res = []                                                    ①
    for elm in tbl:
        res.append(f(elm))                                      ②
    return res
```

① Le tableau que l'on va retourner
② On appel la fonction passé en argument sur tout les élément de `tbl`. On remarque que `f` s'utilise ici comme n'importe qu'elle autre fonction

On vas ensuite définir la fonction a appliquer :

```
def carre(x):
    return x*x
```

Un exemple d'utilisation de la fonction précédentes :

Exemple 0.4.

```
l = [1,2,3,4]
l2 = applique(carre, l)                                         ①

print(l2)
# [1, 4, 9, 16]
```

① Ici on obtiendras une liste contenant les carrées des nombres de `l`

Un autre exemple de fonction d'ordre supérieur (cette fois ci qui retourne une fonction) :

Exemple 0.5. Créons une fonction `mon_homothetie`, qui retourne une fonction multipliant toujours un entier par une constante entière :

```
def mon_homothetie(constante:int)->Callable[[int], int]:        ①
    def f(x):                                                   ②
        return constante * x                                    ③

    return f                                                    ④
```

① Ici la fonction prend un seul argument (`constante`)
② On créer (à l'interieur de la précédente fonction) une fonction f, qui prend un argument `x`. Pour appeller `f` il suffie de fournir `x` (nul besoin de `constante`)

③ Lorsque l'on créer la fonction, cette dernière se rappel de constante, et l'on peux l'utiliser comme si de rien n'était
④ On retourne la fonction comme n'importe quel valeur. Python s'occupe de tout !

Nous n'irons pas plus loin dans l'analyse de cette fonction, car cela sortirait amplement du programme.

Pour utiliser la fonction suivante :

```
g = mon_homothetie(2)                                                    ①

g(3)                                                                     ②
>>> 6

g(10)
>>> 20

g2 = mon_homothetie(10)

g2(10)
>>> 100
```

① On fait applele à `mon_homothetie`. On se rappeleras, que cette dernière retourne une fonction, prenant un entier et retournant un entier. `g` ici seras donc une fonction prenant en argument un entier, et retournant un entier.
② `g` s'utilise comme n'importe quelle autre fonction.

> ❗ Important
>
> Remarquont plusieurs point :
>
> - Lorsque nous voulons passez une fonction en argument (ou pour la retourner), nous utilisons le nom de la fonction, sans `()`
> - A l'inverse lorsque nous voulons exécuter une fonction, nous utilisons les `()` afin de demander son execution
> - Un variable n'est pas obligé d'avoir le même nom que la fonction qu'elle contient. De ce faite, lorsque l'on veu exécuter la fonction qu'elle contient, nous utilisons le nom de la variable et non celle de la fonction.

Définition 0.4 (Fonction Pure)**.** Une fonction est dite pure, lorsque pour les même causes (arguments), elle produits les même effets (valeurs de retour).

Cela insinue entre autre :

- Pas d'effet de bord
- Le retour est le même pour les même arguments

Si une fonction utilise un nombre aléatoire, ou modifie un argument passé en référence, alors elle n'est pas pure.

Exemple 0.6. Les propriété suivantes font que la fonctions ne seras pas pure :

- Modifier un fichier (effet de bord)
- Lire un fichier (le retour dépend aussi du contenu du fichier, et non pas que des arguments)
- Entrée ou sortie à l'utilisateur (`print` ou `input`)

> **Note**
>
> Une fonction pure est née de la conception mathématique d'une fonction. En général, une fonction pure peut être vu comme *"plus sûre"* et plu simple a tester que sa version impure.

Sans donner une définition précise de la programmation fonctionelle, on peut en rafiner la définition :

> **Note**
>
> La programmation fonctionelle va utilisé des fonctions (quasi exclusivement pure), souvent d'ordre supérieur et/ou récursive pour gérer le flux d'éxécution.

Dans la pratique

Ecrivons la fonction `somme` d'une liste chainée en paradigme impératif, et en fonctionelle, en utilisant ce que nous avons appris :

Exemple 0.7. Version **impérative** :

```
def somme(l:Liste_Chainee[int])->int:
    acc = 0
    while not l.EstVide():
        acc += l.Valeur()
        l = l.Suivant()
    return elm
```

Version **fonctionnelle**

```python
def somme_rec(acc:int, l:Liste_Chainee[int])-> int:
    if l.EstVide():
        return acc
    else:
        return somme_rec(acc + l.Valeur(), l.Suivant())

def somme(l:Liste_Chainee[int])->int:
    return somme_rec(0, l)
```

Définition 0.5 (Fonction annonyme)**.** Une fonction est dite annonyme lorsque sa création n'entraine pas la création d'une variable associé

> Astuce
>
> lorsque l'on définie une fonction avec la syntaxe usuel, on créer une variable du même nom.
> Ainsi
>
> ```python
> def f():
>     pass
> ```
>
> Entraine la création d'un variable nommé `f` contenant la fonction. C'est d'ailleur avec cette variable que vous intéragissez en utilisant la syntaxe d'appel `f()`.

En Python, pour créer une fonction annonyme, on utilise la syntaxe suivante :

```python
lambda var1, var2, ..., varn : expression
```

Ici l'expression est l'expression du retour. Pas besoin de `return`, l'utilisation d'une fonction annonyme retourne la seul ligne qu'elle contient.

Exemple 0.8. Créons une fonction annonyme qui renvoie le carré de sa valeur passez en argument.

```python
carre = lambda x : x*x

carre(2)
>>> 4
```

> **Note**
>
> Si on n'assigne pas la fonction anonyme à une variable, ou que l'on ne la passe pas en argument ou en retour d'une fonction, la fonction est perdu est n'est donc pas utilisable. Ainsi, en général, on l'utilise dans un des 3 cas précédent cité.

Des fonctions utiles

La fonction `map`

La fonction `map` est une fonction qui prend en entrée une fonction et une séquence (`list`, `set`, `tuple`, ...) et qui retourne un itérateur, où chaque valeur est prise dans la séquence d'entrée, puis passez dans la fonction, et enfin retourner. En Python `map` ne s'utilise pas seul est est souvent accompagné d'une fonction transformant l'ittérateur en une séquence, comme la fonction `list`.

```
map(f, l)->Iterator




list(map(f, tbl))
>>>[ f(tbl[0]), f(tbl[1]), ..., f(tbl[n])]
```

La fonction `map` est ici très similaire a notre fonction `applique`. Cf Exemple 0.3.

La fonction `filter`

La fonction `filter` est une fonction prennant en entrée une fonction retournant un booléen, et une sequence et qui retourne un itérateur contenant toute les valeurs de la liste, tel que passez dans la fonction, cette dernière renvoie `True`.

La fonction fait donc office de fonction filtrante. On ne garde seulement les élément qui vérifie une propriété tester par la fonction.

Comme pour `map`, on utilise souvent une fonction de conversion.

```
filter(fct:Callable[[Any], bool], l)->Iterator
```

Exemple 0.9.

```
l = [1,2,3,4,5,6,20,11]

l2 = list(filter(lambda x: x%2 == 0, l))                    ①

l2
>>> [2,4,6,20]
```

① Ici la fonction annonyme `lambda x : x%2 == 0` test si le nombre est paire. Ainsi la ligne complète ne conserve que les nombre paire dans `l`.

> **!** Important
>
> `filter` et `map` retorune tout deux des itérateur. N'oublier jamais de réutiliser la fonction `list`, `set` ou `tuple` pour les convertir dans la séquence de votre choix !
> Cela vous eviteras des erreurs, souvent hardu a trouver, venant de la mauvaise utilisation d'un itérateur !

La fonction `reduce`

La fonction `reduce` du module `functools` prend en entrée une fonction prenant 2 argument et qui retourne une valeur du même type, prend aussi une liste et enfin un élément, et retourne la valeur de la fonction appelé recursivement, ou le premier argument est pris dans la séquence, et le deuxième est le resultat des appelle récursif. Le troisième argument sert d'initialiseur pour les appel récursif.

```
functools.reduce(f, tbl, start)
>>> f(tbl[n], f(tbl[n-1], f(... f(tbl[1], f(tbl[0], start)))))
```

Exemple 0.10 (Somme avec reduce)**.** Par exemple, lorsque nous voulons faire la somme d'une liste (disons [1,2,3,4,5]), nous pouvons réécrire la somme ainsi :

$$1+2+3+4+5 = (((((0+1)+2)+3)+4)+5)$$

On peut donc écrire une fonction sommant les élément d'une liste ainsi :

```
import functools

def somme(tbl:List[int])->int:
    return functools.reduce(lambda x, y: x+y, tbl, 0)
```