Chapitre n°2

Programmation fonctionnelle

Table des matières

1. Paradigme de programmation

Un **paradigme de programmation** est un ensemble (plus ou moins formel) de règles et de concepts sous-jacent à un programme informatique. En voici trois exemples (parmi d'autres) :

- La **programmation impérative** consiste à n'utiliser que des instructions de base (affectations, opérations, instructions conditionnelles, boucles, etc.) exécutées l'une après l'autre. Il s'agit de la manière la plus simple d'envisager un programme informatique.
- La **programmation fonctionnelle** repose sur la construction de **fonctions**. La résolution d'un problème donné consiste à associer différentes fonctions de base jouant le rôle de « briques » élémentaires dans la construction du programme.
- La **programmation récursive** consiste à subdiviser un problème complexe en sous-problèmes identiques de plus petite taille (la dichotomie s'y prête bien).

Selon le problème à résoudre, certains paradigmes sont plus efficaces / simples que d'autres :

Code Python

```
# version impérative / itérative
def fact(n): # calcule factorielle n
    p = 1
    for i in range(1,n+1):
        p = p*i
    return p
```

```python
# version récursive
def fact(n): # calcule factorielle n
    if n == 1:
        return 1
    else:
        return n*fact(n-1)
```

On voit que la méthoe récursive est plus courte et ne fait intervenir aucune variable.

Les différents paradigmes peuvent être combinés : dans les deux exemples ci-dessus, la programmation fonctionnelle intervient aussi puisqu'on définit des fonctions.

Certains langages de programmation sont dits « multi-paradigmes » et permettent de programmer du code en respectant différents paradigmes de programmation. C'est le cas de Python, qui permet d'utiliser tous les paradigmes ci-dessus et d'autres encore (par exemple la programmation orientée objet)

2. Programmation fonctionnelle

La programmation fonctionnelle consiste à écrire une ou plusieurs fonction(s) informatique(s). Chaque fonction informatique `f` possède une structure parfaitement analogue à celle d'une fonction mathématique $f : E \to S$:

- Une fonction informatique `f` possède une ou plusieurs **entrée(s)**, aussi appelé(s) **argument(s)**, jouant le même rôle qu'un élément $e \in E$ pour la fonction mathématique f.
- Une fonction `f` retourne une (unique) **sortie**. Pour une entrée `e` donnée, l'instruction `f(e)` retourne toujours la même sortie `s`, de la même manière que pour une fonction mathématique f, tout élément $e \in E$ possède une unique image $s = f(e)$. [1].
- Un fonction mathématique f possède un ensemble de départ E et un ensemble d'arrivée $\mathcal{S}$. Si on pousse la parallèle à l'informatique, une fonction informatique `f` devrait donc imposer un ensemble de valeurs possibles pour son (ou ses) entrée(s) `e` et de même pour sa sortie `s`. En Python, on peut préciser les types des entrées et sorties à travers la **signature** de `f`, mais cela n'a qu'un rôle indicatif.

Une fonction peut donc être représentée comme une « boîte » avec une ou plusieurs entrées, et retournant une unique sortie :

Fig. 1 – représentation schématique du fonctionnement d'une fonction en Informatique

En informatique (comme en mathématiques), il faut d'abord **définir** la fonction `f`, avec le mot-clé `def`, avant de pouvoir l'utiliser. La définition d'une fonction n'est à faire qu'une seule fois : il faut compiler la partie correspondante du code pour qu'elle soit prise en compte.

Une fois la fonction `f` définie, on peut faire un **appel** de cette fonction. Il faut alors préciser, pour chaque entrée, une valeur particulière (appelée une **instance**). La fonction retourne alors la sortie correspondant à l'instance fournie.

1. Y compris lorsque la fonction fait intervenir de l'aléatoire : dans ce cas, $s = f(e)$ suit toujours la même loi de probabilité, pour un même e fixé.

Ex : ce comportement est parfaitement analogue à celui observé en mathématiques. Par exemple, voici la définition de la fonction carré :

$$\begin{cases} \mathbb{R} \longrightarrow \mathbb{R}_+ \\ x \longmapsto f(x) = x^2 \end{cases}$$

définie par le choix des ensembles de départ, d'arrivée (de l'ensemble $\mathbb{R}$ dans l'ensemble $\mathbb{R}_+$) et par son expression ($f(x) = x^2$). On peut ensuite appeler cette fonction : pour chaque appel, l'image $f(x)$ est déterminée en fonction de l'instance de l'entrée x choisie :

$$f(0) = 0 \quad ; \quad f(2) = 4 \quad ; \quad f(-3) = 9 \quad ; \quad \text{etc.}$$

Ici, la première instance choisie est $x = 0$, la seconde instance choisie est $x = 2$, la troisième instance choisie est $x = -3$, etc.

3. Utilisation de fonctions en Python

En Python, la déclaration d'une fonction respecte la syntaxe suivante :

Code Python

```
def nom_de_la_fonction(argument1, argument2):

    # # # # # # # # #
    #  Instructions #
    # # # # # # # # #

    return sortie
```

Lors de la déclaration d'une fonction en Python :

- la déclaration de la fonction débute avec le mot-clef `def` ;
- la ligne comportant le mot-clef `def` termine par le symbole `:` ;
- l'intégralité de la définition de la fonction est indentée.
- `nom_de_la_fonction` désigne le nom de la fonction (utilisé ultérieurement lors de l'appel de cette fonction) ;
- le mot-clef **return** permet de retourner la sortie déterminée à partir de l'instance utilisée lors de l'exécution de la fonction ;

⚠ Attention ⚠

Le mot-clé **return** met fin à l'exécution de la fonction.

La définition d'une fonction peut tout-à-fait comporter plusieurs mot-clefs `return`, mais l'exécution de cette fonction prendra fin dès qu'un de ces mots-clefs aura été atteint lors de l'exécution du programme :

Code Python

```
def f(x):
    if x < 0:
        return "x est négatif"
    elif x > 0:
        return "x est positif"

    return "x est nul"

    return "cette sortie ne peut pas être retournée par la fonction !"
```

- Une fonction peut admettre n'importe quel nombre d'arguments :

Code Python

```
def hum(): # 0 argument
    return 42

def carre(x):
    return x ** 2

def ajout(L, a): # 2 arguments
    return L + [a]
```

Il faut alors appeler la fonction avec le bon nombre d'arguments, sinon cela cause une erreur :

Code Python

```
>>> carre(-3.0)
9
>>> carre()
Traceback [...]
TypeError: foo() missing 1 required positional argument: 'x'
>>> carre(0, 2)
Traceback [...]
TypeError: foo() takes 1 positional argument but 2 were given
>>> carre("plop")
Traceback [...]
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

Dans le dernier exemple, le nombre d'arguments était correct mais il y a eu une erreur de type car `x**2` n'a pas de sens si `x` est une chaîne de caractères. On peut ajouter une **signature** à la fonction pour indiquer à l'utilisateur le type attendu en argument et le type de la sortie :

Code Python

```
>>> def ajout(L:list,a:float)->list                # signature de ajout
>>> ...
```

Dans l'exemple ci-dessus, on indique à l'utilisateur que `L` doit être une liste et `a` doit être un flottant pour que `ajout(L,a)` fonctionne : la sortie sera alors une liste.

4. Portée des variables

En programmation fonctionnelle, le code se sépare en deux environnements :

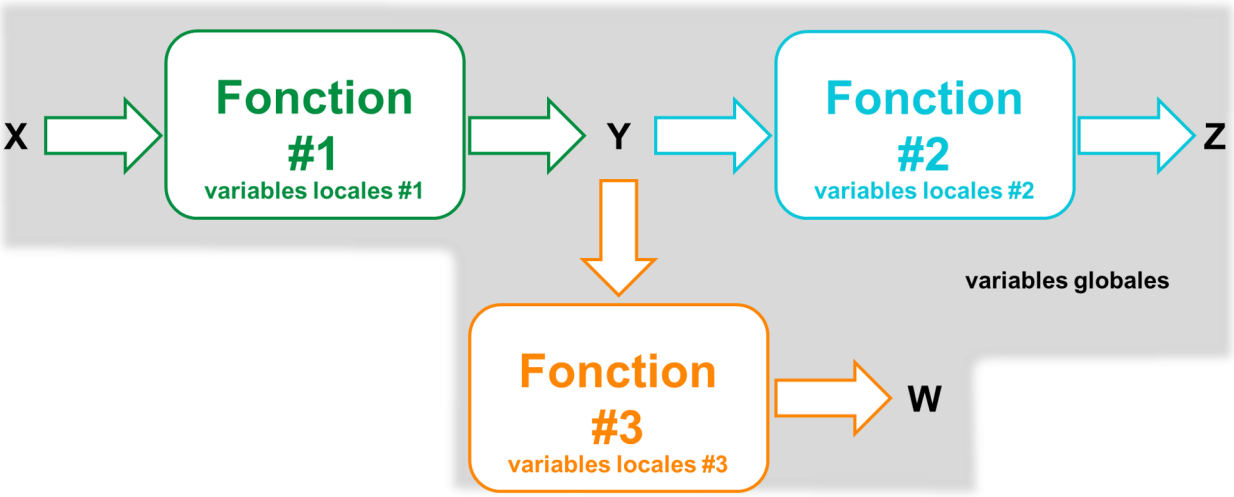

Fig. 2 – X,Y,Z,W sont des variables globales. Chaque fonction peut avoir des variables locales.

- Les **variables locales** sont des variables qui sont utilisées uniquement dans la fonction. Ce sont des variables muettes : elles sont effacées lorsque la fonction se termine.
- Les **variables globales** sont des variables générées en-dehors de toute fonction. Elles peuvent cependant être utilisées et modifiées par des fonctions.

Code Python

```
varGlobale = 42

def double(arg):
    varLocale = 2 * arg
    return varLocale

# On suppose qu'on a compilé le code ci-dessus

>>> varGlobale
42

>>> double(varGlobale)
84

# Les variables locales n'existent pas en-dehors la fonction
>>> arg
Traceback [...]
NameError: name 'arg' is not defined
>>> varLocale
Traceback [...]
NameError: name 'varLocale' is not defined
```

On voit ici que les variables locales `arg` et `varLocale` n'existent "qu'à l'intérieur" de la fonction `double`. De manière similaire, l'écriture de la fonction mathématique

$$f : x \mapsto \sum_{k=1}^{10} x^k$$

fait intervenir deux variables, k et x, toutes deux muettes. Écrire la ligne ci-dessus n'introduit pas (ou ne modifie pas) en soi les variables x et k. Ainsi, on pourrait techniquement écrire :

On pose $x = 1$ et $k = 0$.
On pose $f : x \mapsto \sum_{k=1}^{10} x^k$.
Alors $f(x) = 10$ et $f(k) = 0$ (puisque $x = 1$ et $k = 0$).

Mais écrire les choses ainsi n'est pas conseillé du tout ! De même en informatique :

⚠ Attention ⚠

En programmation fonctionnelle, les variables locales et globales ne doivent (idéalement) jamais avoir le même nom.

Bien que les variables locales n'existent plus en dehors de la fonction, les variables globales existent partout, y compris à l'intérieur des fonctions. On peut donc écrire :

Code Python

```
varGlobale = 7

def f(arg):
    return arg * varGlobale          # Un exemple de MAUVAISE pratique...

>>> f(4)
28
```

Un premier problème de cette fonction est que si `varGlobale` n'a pas été définie, alors la fonction `f` renverra une erreur.
Ainsi, bien que ce soit autorisé en Python, on évite que les fonctions nécessitent des variables globales pour fonctionner. L'équivalent en mathématiques de cet exemple serait :

On pose $y = 7$.
On pose $f : x \mapsto xy$.
Alors $f(4) = 28$.

Ce n'est techniquement pas faux, mais la fonction f dépend également d'un paramètre y. On devrait donc la noter f_y ou mieux la redéfinir comme une fonction de deux variables $f : (x, y) \mapsto xy$. C'est aussi ce qu'on recommande de faire en Python :

Code Python

```
# Un exemple de BONNE pratique
varGlobale = 7

def f(arg1,arg2):
    return arg1 * arg2

>>> f(4,varGlobale)
28
```

Ici, toutes les informations nécessaires au bon déroulement de la fonction sont passées en argument : la fonction est bien définie et est utilisable indépendamment de toute variable globale.

Un **effet de bord** consiste à modifier une variable globale lors de l'exécution d'une fonction. La programmation fonctionnelle de Python empêche automatiquement ces effets de bord, excepté lorsque la variable globale en question est une *liste*.

Code Python

```
# Un exemple avec effet de bord
L = [3, 5, 7] # liste globale

def fonc(M):
    M[0] = M[0] - 3
    return M

>>> fonc(L)
[0, 5, 7]
>>> L
[0, 5, 7]
```

Cet effet de bord est rarement souhaitable. Dans l'exemple ci-dessus, si on réalise l'instruction `fonc(L)` plusieurs fois d'affilée, le résultat retourné sera à chaque fois différent : `[0, 5, 7]`, puis `[-3, 5, 7]`, etc. Afin d'éviter ce phénomène, il suffit de créer une copie de la liste passée en argument, puis de travailler uniquement avec la copie :

Code Python

```
# Un exemple sans effet de bord
L = [3, 5, 7]

def fonc(M):
    copie = M.copy() # on ne touche plus à liste M à présent
    copie[0] = copie[0] - 3
    return copie

>>> fonc(L)
[0, 5, 7]
>>> L
[3, 5, 7]
```

⚠ Attention ⚠

Attention à ne pas modifier une liste passée en argument.

5. Sorties multiples

Il est en fait possible pour une fonction informatique de retourner plusieurs sorties, séparées avec une virgule `,`. Cela revient en fait à retourner un tuple qui contient toutes ces sorties.

Code Python

```
def F(x):
    return x - 2, x + 2                    # identique à return (x-2,x+2)

>>> # Utilisation de la capacité d'affectation parallèle
>>> F(8)
(6,10)

>>> a,b = F(8)
>>> a
6
>>> b
10
```

L'instruction `a,b = F(8)` fonctionne, car cela revient à écrire `a,b = (6,10)`, ce qui revient à faire les affectations `a=6` et `b=10`. Cependant, l'instruction `a,b,c = F(8)` renverrait une erreur : il faut exactement autant de variables à gauche que de valeurs à droite pour réaliser ces affectations multiples.

6. Rapide aperçu de la POO

La programmation orientée objet (POO) est un paradigme de programmation qui consiste à penser tout élément du programme comme un "objet". Un objet est une structure de données munie d'attributs (données intrinsèques de cet objet) et de méthodes (fonctions associées à cet objet).

Par exemple, un complexe `z=3+2j` est muni de l'attribut `z.real` qui retourne `3` et d'une méthode `z.conjugate()` qui retourne `3-2j`. `z` possède donc l'attribut `real` (la partie réelle est une donnée intrinsèque de toute complexe) mais aussi une méthode `conjugate()` qui ne prend aucun argument (le conjugué n'est pas une donnée intrinsèque d'un complexe, cela ne peut être qu'une méthode).

Comme les méthodes sont des fonctions, on met toujours des parenthèses. Les listes disposent également de méthodes, notamment `L.append(x)` et `L.copy()`. Par choix, elles n'ont pas d'attribut notable : pour avoir la longueur d'une liste, on utilise `len(L)`.

À l'issue de ce cours

À l'issue de ce cours, je suis capable :

- ☐ d'expliquer ce qu'est un paradigme de programmation ;
- ☐ de définir les notions d'entrée(s), de sortie, d'instance ;
- ☐ de déclarer une fonction sous Python ;
- ☐ d'appeler une fonction préalablement déclarée sous Python ;
- ☐ d'utiliser correctement le mot-clef `return` ;
- ☐ d'expliquer le concept de portée des variables, de définir les notions de variables locales et globales ;
- ☐ d'éviter les effets de bord, notamment lors de l'utilisation de listes passées en entrée d'une fonction.