1 Introduction

A l'instar de la notion de **variable**, la notion de **fonction** reflète deux modes de pensée différents en Mathématiques et en Informatique. En Mathématiques, elle correspond à *une relation entre deux ensembles*, alors qu'en Informatique, elle fait référence à *un procédé de calcul*.

L'idée générale d'une fonction informatique est donc d'associer un nom à une séquence d'instructions particulière (on parle alors d'*encapsulation*) afin de pouvoir la réutiliser ultérieurement dans l'algorithme, voire dans un autre algorithme.

Par exemple, la fonction `type` vue précédemment fait référence à une séquence d'instructions permettant d'obtenir le type d'une donnée, et peut être utilisée quand on veut et dans n'importe quel algorithme. De même, la fonction `sqrt` fait référence à une séquence d'instructions permettant d'obtenir la racine carrée d'un nombre.

Ce procédé d'association (ou d'encapsulation) permet également de mettre en relief les trois parties d'un algorithme :

- les entrées correspondent aux paramètres de la fonction ;
- le corps correspond à la séquence d'instructions à laquelle la fonction fait référence ;
- les sorties correspondent aux données qui sont renvoyées par la fonction à l'issue de son exécution.

Par exemple, les fonctions `round`, `abs` et `sqrt` vues précédemment sont des fonctions qui prennent une valeur numérique en entrée et renvoient un résultat, également de type numérique.

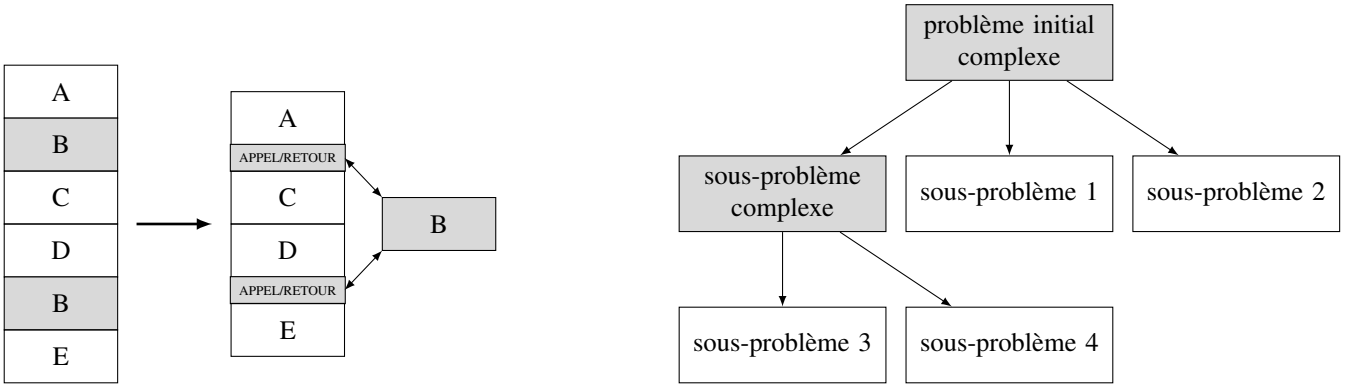
Noter que pour ces fonctions, on ne connaît pas précisément la séquence d'instructions qui est utilisée. C'est là tout l'intérêt de l'encapsulation : la fonction peut avoir été écrite par quelqu'un d'autre, il suffit juste de savoir ce que l'on doit donner en entrée et ce que l'on obtient en sortie...

2 Définition et utilités

Définition 2.1 Une *fonction* est définie par un *nom* et correspond à une séquence d'instructions réalisant une tâche précise, en utilisant un ou plusieurs *arguments*, voire aucun. Ces arguments sont les *paramètres* de la fonction. La fonction renvoie un résultat.

Les fonctions ont plusieurs utilités :

- mettre en relief les données et les résultats (entrées et sorties de la fonction) ;
- permettre la réutilisation dans d'autres algorithmiques ;
- éviter les répétitions (voir la figure 1a ci-dessous) ;
- décomposer une tâche complexe (voir la figure 1b ci-dessous).



(a) Eviter les répétitions

(b) Décomposer une tâche complexe

FIGURE 1 – Quelques utilités d'une fonction

En conséquence : **les algorithmes sont plus courts et plus lisibles.**

3 Syntaxe et exemples

Pour définir des *fonctions* en Python on utilise le mot-clé **def**. La syntaxe pour la création d'une fonction est la suivante :

```
def nom_fonction(liste_paramètres) :          # ne pas oublier les :
   """ documentation de la fonction """       # chaîne de caractères entre triple guillemets
   bloc d'instructions                        # indentation obligatoire
   return résultat                            # renvoie le résultat de la fonction
# pour terminer, on stoppe l'indentation
```

Nota Bene :

- la liste des paramètres peut être vide ;
- la documentation de la fonction est très importante. Elle fournit des informations sur les types des paramètres, sur ce qu'elle fait et sur ce qu'elle renvoie. On accède à cette documentation en utilisant l'instruction `help(nom_fonction)` ;
- l'instruction `return` stoppe l'exécution de la fonction et renvoie une ou plusieurs valeurs. De cette façon on peut récupérer le résultat renvoyé par une fonction. Si plusieurs valeurs sont renvoyées, elles le sont sous forme d'un tuple que l'on peut affecter, soit à une seule variable, soit à plusieurs variables à l'aide d'une affectation multiple ;
- si l'instruction `return` est omise, la fonction renvoie `None` et on parle alors de *procédure*.

On voir la notion de fonction comme étant un mini-algorithme puisque les paramètres sont les données d'entrée, les instructions composant le corps de la fonction (partie indentée) correspondent au traitement des données, les valeurs renvoyées sont les données de sortie.

Exemple 3.1 La fonction `distance_parcourue` ci-dessous prend en argument la vitesse en km/h et le temps en secondes et renvoie la distance parcourue en mètres :

Algorithme
```
fonction distance_parcourue(vitesse,temps)
   vitesse_ms ← vitesse*1000/3600
   distance ← vitesse_ms * temps
   renvoyer distance
fin fonction
```

Programmation Python
```
def distance_parcourue(vitesse,temps):
   vitesse_ms = vitesse*1000/3600
   distance = vitesse_ms * temps
   return distance
```

Exercice 3.2 Pour bien comprendre comment fonctionne une fonction, regardons en détail quelques exemples via ce Notebook.

4 Appel d'une fonction

Pour utiliser une fonction, il faut l'« appeler » et lui passer des valeurs par arguments si elle possède des paramètres. Les valeurs passées doivent bien sûr être dans le même ordre que les paramètres et du même type que ceux-ci.

Exemple 4.1 Pour connaître la distance parcourue par un véhicule roulant à 72 km/h pendant 2 minutes, on saisit l'instruction

```
distance_parcourue(72,120)
```

qui renvoie la valeur `2400.0`. On dit que l'on procède à *l'appel de la fonction* `distance_parcourue` pour les valeurs 72 et 120.

Remarque 4.2 Les paramètres d'une fonction peuvent être eux-mêmes des fonctions.
Considérons par exemple les trois fonctions suivantes :

```
def cube(x):
    """
    Fonction cube
    """
    return x**3
```

```
def monAffine(x):
    """
    Fonction affine
    x -> 2x + 3
    """
    return 2*x + 3
```

```
def composition(f,g,x):
    """
    Renvoie f(g(x))
    """
    return f(g(x))
```

La troisième fonction prend en paramètres deux fonctions `f` et `g`. En tapant l'instruction `composition(cube,monAffine,1)` on obtient ainsi le cube de 5, soit 125. En tapant `composition(monAffine,cube,2)` on calcule cette fois l'image du cube de 2 (soit 8) par la fonction affine $x \mapsto 3x + 2$ et on obtient 19. La fonction `composition` est réutilisable avec d'autres fonctions compatibles, c'est-à-dire telles que l'ensemble image de g est inclus dans l'ensemble de définition de f, par exemple on ne peut pas calculer la racine carrée d'un nombre négatif, ni calculer l'inverse de zéro.

5 Portée des variables : variables locales et variables globales

Lorsqu'une fonction est appelée, Python réserve pour elle (dans la mémoire de l'ordinateur) un espace de noms. Cet espace de noms local à la fonction est différent de l'espace de noms global où se trouvent les variables du programme principal. Dans l'espace de noms local, nous aurons des variables qui ne sont accessibles qu'au sein de la fonction.

- Les variables qui sont définies dans le corps d'une fonction sont des variables *locales*, c'est-à-dire qu'elles n'existent qu'à l'intérieur de cette fonction. En particulier, une fois la fonction exécutée, ces variables sont détruites et n'existent plus. Une variable locale peut avoir le même nom qu'une variable de l'espace de noms global mais elle reste néanmoins indépendante.
- Les variables qui sont définies en dehors du corps de toutes fonctions sont appelées variables *globales*. En Python ces variables sont accessibles à l'intérieur du corps d'une fonction uniquement en *lecture*, autrement dit elles leur contenu est « visible », mais la fonction ne peut pas le modifier sauf si on les redéclare explicitement avec le mot-clé `global`.

Exemple 5.1 Une fois l'instruction `distance_parcourue(72,120)` exécutée (voir les exemples 3.1 et 4.1) les variables `vitesse_ms` et `distance` n'existent plus car ce sont des variables locales à la fonction `distance_parcourue`.

Exemple 5.2 Les instructions ci-dessous renvoient une erreur du type `UnboundLocalError`, c'est-à-dire que l'on a essayé de modifier une variable non définie dans la fonction `modif()`.

```
a = 3

def modif():
    a += 1
    return a

modif()
```

Pour pouvoir modifier une variable globale à l'intérieur d'une fonction (et donc modifier globalement sa valeur), il faut utiliser le mot-clef **`global`**. Ainsi, en ajoutant l'instruction `global a` juste avant de modifier la valeur de la variable `a`, on peut modifier la valeur de `a` : à l'issue de l'exécution de la fonction `modif`, la variable `a` contient la valeur `4`.

```
a = 3

def modif():
    global a
    a += 1
    return a

modif()
a
```

6 Exercices

Exercice 6.1 (Comprendre et manipuler des fonctions) Compléter le Notebook suivant.

Exercice 6.2 (QCM)

1. On considère la fonction suivante :

```
def double(x):
    return 2*x
```

Parmi les affirmations suivantes, lesquelles sont vraies ?

(a) l'instruction `double(5)` renvoie `25`
(b) les instructions `a = 2 ; double(a)` renvoient `4`
(c) `x` est un paramètre de la fonction
(d) la fonction s'appelle `x`

2. On considère la fonction suivante :

```
def somme(x,y):
    s = x + y
    return s
```

Parmi les affirmations suivantes, lesquelles sont vraies ?

(a) les instructions `a = -1 ; somme(a,a)` donnent une erreur car `x` et `y` doivent être deux variables différentes

(b) `s` est une variable locale à la fonction

(c) la fonction a trois paramètres

(d) l'instruction `somme(2,3)` renvoie `5`

3. On considère le code `Python` suivant :

```
def aireRect(longueur,largeur):
    return longueur*largeur

A = aireRect(5,2)
B = aireRect(A,A)
A = aireRect(A,A)
```

Parmi les affirmations suivantes, lesquelles sont vraies à la fin de l'exécution de ce code ?

(a) la variable `B` contient le résultat de `100*100`

(b) les variables `A` et `B` contiennent les mêmes valeurs

(c) la variable `A` contient la valeur `100`

(d) la variable `A` contient la valeur `10`

4. On considère la fonction suivante :

```
def discr(a,b,c):
    a = b = c = 1
    return b**2 - 4*a*c
```

Parmi les affirmations suivantes, lesquelles sont vraies ?

(a) l'instruction `discr(2,5,1)` renvoie `17`

(b) l'instruction `discr(2,5,1)` renvoie `-3`

(c) les paramètres `a`, `b` et `c` ne servent à rien

(d) l'instruction `discr(1,1,1)` renvoie `-3`

5. On considère le code Python suivant :

```
def neModifiePas(x):
    x = 10
    print("Valeur de x :",x)

x = 200
print("Valeur de x :",x)
neModifiePas(x)
print("Valeur de x :",x)
```

Quelles sont les valeurs affichées lors de l'exécution de ce code ?

(a) `200`, `10` et `200`

(b) `200`, `200` et `200`

(c) aucune

(d) `200`, `10` et `10`

Exercice 6.3 (Créer ses propres fonctions) Compléter le Notebook suivant.

Exercice complémentaire 6.4 (Conversions de température)
Le **degré celsius** (symbole $°C$) est l'unité de l'échelle de température Celsius.
Le **kelvin** (symbole K) est l'unité de température thermodynamique.

La formule suivante $C = K - 273,15$, où C est la température en degré Celsius et K la température en degré kelvin, permet de passer d'une unité à l'autre.

1. Lorsque $K = 12$, quelle est la valeur de C ?
2. Lorsque $C = 25$, quelle est la valeur de K ?
3. Lorsque $C = 0$, quelle est la valeur de K ?
4. Ecrire deux fonctions de conversion : l'une permettant de convertir des degrés Celsius en degré Kelvin et l'autre convertissant des degrés Kelvin en degré Celsius. On n'oubliera de pas de les documenter.