IngeSUP - Cours 08 - Les fonctions 2

Sommaire ¶

Prérequis

- Module 16 : Les fonctions en Python – Les bases (https://courses.ionisx.com/courses/ref/m184/x/courseware/533425cca7814a619f38de67bf5bb024/)
- Première partie du cours sur les fonctions : Cours 07 - Les fonctions 1 (./Cours%2007%20-%20Les%20fonctions%201.ipynb)

Rappel sur les fonctions

Une fonction est un bloc d'instructions que l'on peut appeler à tout moment dans programme. Les fonctions ont plusieurs intérêts, notamment :

- **la réutilisation du code** : éviter de répéter les mêmes séries d'instructions à plusieurs endroits d'un programme ;
- **la modularité** : découper une tâche complexe en plusieurs sous-tâches plus simples.

Définir une fonction

Au cours des chapitres précédents, on a déjà rencontré de nombreuses fonctions telles que `print` ou `len` . Chacune de ces fonctions reçoit un argument et effectue une action (la fonction `print` affiche un objet à l'écran), ou renvoie une valeur (la fonction `len` renvoie la taille d'une liste).

On a également appris à définir nos propres fonctions : **il faut déclarer nos fonctions avant de les utiliser**. De manière générale, la syntaxe d'une déclaration de fonctions est la suivante.

```
    def <nom_fonction>(<paramètres>):   # En-tête de la fonction
        <instruction1>
        <instruction2>                  # Corps de la fonction
        ...
        return <valeur>                 # return n'est pas obligatoire
```

On décrit dans le corps de la fonction **les traitements à effectuer** sur les paramètres et on spécifie **la valeur que doit renvoyer la fonction**.

Considérons l'exemple simple suivant :

Entrée [ ]:

```
def factorielle(n):
    a = 1
    for k in range(n,0,-1):
        a *= k   # a *= k ça veut dire a = a*k | x += 1 <=> x = x+1
    return a
```

La fonction `factorielle` prend en argument un objet *n* (que l'on supposera être un entier naturel), calcule la `factorielle` de *n* à l'aide d'une variable *a* et renvoie cette valeur.

On constate que rien ne se passe lorsque la fonction est déclarée. Il faut appeler la fonction en fournissant une valeur à l'entier *n* pour que le code soit exécuté.

Entrée [ ]:

```
factorielle(5)
```

Entrée [ ]:

```
factorielle(7)
factorielle(8)
```

> Il faut bien faire la différence entre **la déclaration** et **l'appel** de la fonction. Lorsqu'une fonction est **déclarée**, aucun code n'est exécuté. Il faut **appeler** la fonction pour que le code soit exécuté.

Portée des variables

Une fonction peut utiliser des variables définies à **l'extérieur** de cette fonction.

Entrée [ ]:

```
a = 2   # Variable définie à l'extérieur de la fonction

def f(x):
    return a * x    # return 2 * x parce que a=2
```

On dit que les variables définies à l'extérieur d'une fonction sont des variables **globales**.

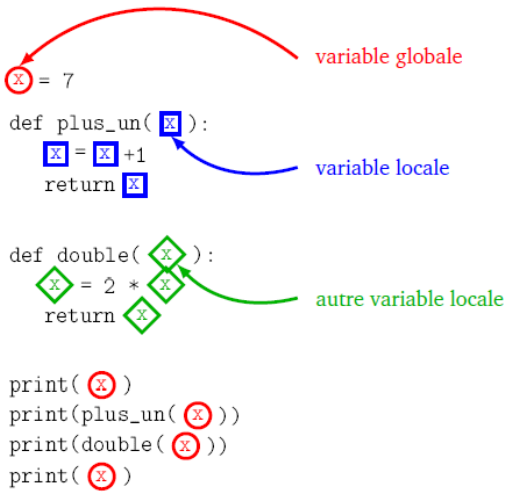
> De manière générale, il est plutôt déconseillé d'utiliser des variables globales car elles rendent le debuggage du code difficile.

En tout cas, quand on utilise une variable globale dans une fonction, il faut que celle-ci soit déclarée avant l'appel de la fonction.

> **Note**
>
> Les variables définies à l'intérieur d'une fonction sont appelées variables locales. Elles n'existent pas en dehors de la fonction.
>
> S'il existe une variable dans une fonction qui porte le même nom qu'une variable dans le programme (comme le x dans l'exemple ci-dessous), c'est comme si il y avait deux variables distinctes ; **la variable locale n'existe que dans la fonction**.

Pour bien comprendre la portée des variables, on colorie dans l'exemple suivant, les variables globales de la fonction en rouge, et les variables locales avec une couleur par fonction.

Le petit programme suivant définit une fonction qui ajoute un et une autre qui calcule le double.



Question 1: Quel sera l'affichage du code ci-dessus ?

Question 2: Expliquez alors l'affichage du code suivant

Entrée [ ]:

```
def test():
    b = 5
    print(a, b)

a = 2    # à partir d'ici on est à l'extérieur de la fonction
b = 7
test()
print(a,b)
b = 4
print(a,b)
```

Il est tout de même possible de forcer la main à Python et de modifier une variable globale dans une fonction à l'aide du mot clé ***global***.

Cette instruction permet d'indiquer - à l'intérieur de la définition d'une fonction - quelles sont les variables à traiter globalement, **c'est-à-dire qu'elles auront la même valeur partout dans le code**.

Entrée [ ]:

```
def test():
    global b
    b = 5
    print(a, b)

a = 2
b = 7
test()
print(a, b)
```

Question 3: Expliquez l'affichage du code précédent

Méthodes : Fonctions et suites numériques

Les suites mathématiques peuvent être définies par des algorithmes. Dans ce cas l'algorithme prend comme entrée un entier naturel et renvoie un réel. L'entier naturel correspond à l'indice de la suite et le réel à la valeur du terme correspondant. Il existe plusieurs cas de figure.

La définition explicite

Une suite est définie de manière explicite lorsque l'on connaît directement la valeur de n'importe quel U_n en fonction de n.

> Une suite (U_n) définie sous forme explicite est donnée par son terme général U_n exprimé en fonction de n. Elle est reliée à une fonction f telle que $U_n = f(n)$.

Exemple

Soit la suite (U_n) définie sur $\mathbb{N}$ par :

$\forall n \in \mathbb{N}, U_n = 2n - 1$

Cette suite est définie de façon explicite.

Pour tout entier naturel n, on a $U_n = f(n)$ où f est la fonction affine $x \to 2x - 1$.

> Pour coder cette suite il suffit de coder la fonction affine en Python

```
def U(n):
    return 2*n-1
```

ou

```
U = lambda n : 2*n-1
```

Le calcul de chaque terme est direct, par exemple pour afficher le terme d'indice 83:

```
print(U(83))    # Affiche 165
```

La définition par récurrence simple

Lorsqu'une suite est définie par récurrence, on ne peut pas calculer directement la valeur du terme U_n en fonction de n. On sait uniquement calculer la valeur du terme U_n en fonction de celui qui précède.

> Une suite (U_n) définie par récurrence est donnée par son terme initial u_{n0} et une relation reliant chaque terme au terme suivant. On introduit alors une fonction f telle que :

$$\begin{cases} u_{n0} \in \mathbb{R} \\ \forall n \geq n_0 \in \mathbb{N}, U_{n+1} = f(U_n) \end{cases}$$

Exemple

On considère la suite u définie sur $\mathbb{N}$ par :

$$\begin{cases} u_0 = 2 \\ \forall n \in \mathbb{N}, U_{n+1} = 2U_n - 1 \end{cases}$$

On a, pour tout entier naturel n, $u_{n+1} = f(u_n)$ où f est la fonction affine $x \to 2x - 1$.

> Coder la suite en python revient donc à créer une fonction qui prend en entrée un entier n (qui correspond au terme dont on veut la valeur) et à renvoyer la valeur calculée pour ce terme.

Pour ça:

1. On associe le premier terme à une variable (par exemple *u0*).
2. Si le terme demandé lors de l'appel à la fonction est égal au premier terme on renvoie *u0*.
3. Sinon on crée une boucle qui va du **deuxième terme au rang n+1** (pour s'arrêter au rang ***n***, c'est ainsi que fonctionne la boucle **for** en python).

4. Dans cette boucle on applique la fonction f, telle que $U_{n+1} = f(U_n)$ à la variable qui contient le premier terme (par exemple *u0*). Le résultat est stocké dans la variable qui contenait déjà le premier terme (on écrase ainsi l'ancienne valeur de *u0*).
5. A la fin de la boucle on renvoie la valeur ainsi calculée.

Codons la suite u définie ci-dessus:

Entrée [ ]:

```
def u(n):
    # 1. On associe le premier terme à une variable.
    u0 = 2
    # 2. Si le terme demandé lors de l'appel à la fonction est égal au premier terme on le renvoie.
    if n == 0:
        return u0
    else:
        # 3. Sinon on crée une boucle qui va du deuxième terme au rang n+1
        for i in range(1,n+1):
            # Dans cette boucle, on applique la fonction associée au premier terme et on écrase sa valeur
            u0 = 2*u0-1
        # 5. A la fin de la boucle on renvoie la valeur ainsi calculée.
        return u0

print(u(0))
print(u(1))
print(u(2))
print(u(3))
print(u(60))
```

La définition par récurrence double

Lorsqu'une suite est définie par récurrence double, on calcule la valeur du terme U_n en fonction des deux termes qui précèdent.

Exemple

Illustrons l'utilité du principe de récurrence double grâce à un exemple.

On définit la suite a_n par:

$$\begin{cases} a_0 = a_1 = 1 \\ \forall n \in \mathbb{N}, a_{n+2} = a_{n+1} + 2a_n \end{cases}$$

Dans cette configuration, il peut être difficile de trouver une fonction f telle que $U_{n+1} = f(U_n)$ comme dans l'exemple précédent.

Malgré ça, la démarche pour coder une telle suite est équivalente, sauf qu'on utilise deux termes génériques pour coder la suite, au lieu d'un seul.

> Comme avant, coder la suite en python revient à créer une fonction qui prend en entrée un entier n (qui correspond au terme dont on veut la valeur) et à renvoyer la valeur calculée pour ce terme.

Pour ça:

1. On associe les deux premiers termes à des variables (par exemple *u0* et *u1*).
2. Si le terme demandé lors de l'appel à la fonction est égal au premier terme (par exemple *u0*) ou au deuxième terme (par exemple *u1*), on les renvoie.
3. Sinon on crée une boucle qui va du **troisième terme au rang n+1** (pour s'arrêter au rang ***n***, c'est ainsi que fonctionne la boucle **for** en python).

4. Dans cette boucle, à chaque tour on applique les actions suivantes :
   - On applique la relation de récurrence aux deux premiers termes (par exemple *u0* et *u1*). On stocke le résultat dans une nouvelle variable (par exemple *u*).
   - La variable qui désigne le premier terme prend la valeur du deuxième terme (par exemple *u0* prend la valeur de *u1*).
   - La variable qui désigne le deuxième terme prend la valeur de la nouvelle variable (par exemple *u1* prend la valeur de *u*).

5. A la fin de la boucle on renvoie *u1*.

Les noms de variable sont donnés à titre d'exemple.

Codons la suite a définie ci-dessus:

Entrée [ ]:

```
def a(n):
    # 1. On associe les deux premiers termes à des variables
    a0 = a1 = 1
    # 2. Si le terme demandé lors de l'appel à la fonction est égal au premier terme ou au deuxième terme
    # on le renvoie.
    if n == 0:
        return a0
    elif n == 1:
        return a1
    else:
        # 3. Sinon on crée une boucle qui va du troisième terme au rang n+1
        for i in range(2, n+1):
            # On applique la relation de récurrence aux deux premiers termes.
            # On stocke le résultat dans une nouvelle variable.
            a = a1 + 2*a0

            # u0 prend la valeur de u1
            a0 = a1

            # u1 prend la valeur de u
            a1 = a

        return a1

print(a(1))
print(a(2))
print(a(3))
print(a(44))
```

Méthodes : Sommes arithmétiques

La notation mathématique utilise un symbole qui représente la **somme d'une suite de termes** : le symbole de sommation **Σ**, une forme élargie de la lettre grecque sigma capitale. Celui-ci est défini comme suit :

$$\sum_{i=m}^{n} a_i = a_m + a_{m+1} + a_{m+2} + \ldots + a_n$$

Ici *i* représente l'indice de sommation ; a_i est une variable indexée représentant chaque nombre successif de la série ; m est la limite inférieure de sommation, et n est la limite supérieure de sommation.

Cette somme d'un ensemble fini de nombres peut être calculée à l'aide du langage Python. L'idée est tout simplement de calculer au fur et à mesure les additions successives, en les ajoutant dans une seule variable (nommée par exemple *s* ou *somme*). Cette variable est initialisée 0.

Illustrons ceci à travers un exemple; voici un exemple montrant une somme de carrés.

$$\sum_{i=3}^{6} i^2 = 3^2 + 4^2 + 5^2 + 6^2 = 86$$

Voici la méthode pour programmer un tel calcul de sommes en Python :

Initialiser la somme à 0.

- Faire une boucle qui va de la limite inférieure de la sommation à la limite supérieure (incluse).
- A chaque tour de boucle :
  - Ajouter à la variable somme le contenu de la formule de la sommation.
- A la fin de la boucle retoruner la variable somme.

Codons la somme arithmétique précédente en langage Python. On a choisi pour cet exemple de mettre la limité inférieure (variable *inf*) et la limite supérieure (variable *sup*) en paramètres.

Entrée [ ]:

```
def somme_arithmetique(inf, sup):
    somme = 0
    for i in range(inf, sup+1):
        somme += i**2
    return somme

print(somme_arithmetique(3,6))
```

Information

Pourquoi avoir mis sup+1 pour la borne droite du for et pas tout simplement sup ?

Tout simplement parce que la borne droite du for en Python, correspond à une valeur qui n'est jamais atteinte. Si on veut être sur d'atteindre la valeur sup il faut donc mettre sup+1 et non pas sup seulement (auquel cas la boucle va s'arrêter à sup-1).

Exercices de TD

Vous pouvez maintenant vous exercer à partir du notebook TD 08 - Les fonctions 2 (../TD/TD%2008%20-%20Les%20fonctions%202.ipynb)