TD 03 Notion de fonction

Lycée Louis Thuillier - Informatique - PCSIB -

> *Exemple 1 :* ***Intérêt d'une fonction***
> *Imaginons qu'on a a précédemment créé un programme permettant de compter le nombre de diviseur d'un nombre entier qu'on avait demandé à l'utilisateur de préciser via la commande input(). Le programma fonctionne, fantastique !*
> *Mais il est possible qu'on souhait calculer ce nombres de diviseurs pour tout en tas d'autre nombres et qu'on souhaite également utiliser le résultat pour faire autres choses. Il faudra alors recopier le code précédemment écrit, en l'adaptant un peu. Ce qui n'est pas pratique ...*
>
> *On va alors chercher à créer une commande en Python qui, si je lui précise l'entier n sur lequel je souhaite travailler, me renvoie "automatiquement" le nombre de diviseur de n. On va vouloir créer* ***une fonction Python****.*

1 Notion de fonction informatique

Attention ! Même s'il y a des similitudes, un fonction en informatique est différent d'une fonction en math !!

1.1 Définition et appel

Une fonction est un objet à part entière (comme un nombre, une liste, ...) qui se définit via :
- un nom
- une liste de paramètres
- une liste d'instruction
- une valeur renvoyée

♡ **Commande ♡. Fonction**

```
def nom_fonction(parametre1,parametre2):
    instruction1
    instruciton2
    return valeur
```

Attention ! indentation !!
Attention ! la commande *return* marque la fin de la fonction

Il faut voir une fonction comme une recette de cuisine :
- un nom ∼ nom du plat à cuisiner
- une liste de paramètres ∼ nombre d'invités, ...
- une liste d'instruction ∼ les étapes de la recette
- une valeur renvoyée ∼ le plat servi

Lors de la définition de la fonction, on **apprend** à Python la fonction (∼on apprend la recette). Pour l'utiliser, il faut ensuite l'**appeler** en utilisant son nom et en précisant la valeur des paramètres (∼ on demande à Python d'appliquer la recette pour 3 personnes).

Exemple 2 :

```
>>> def fonction_puissance(x,n):
    result=x**n
    return result
...

>>> fonction_puissance(2,4)
16

>>> x=fonction_puissance(2,4)

>>> print(x+1)
17
```

▷ *j'apprends à Python la fonction : il ne renvoie donc rien*

▷ *pour utiliser la fonction, je l'appelle grâce à son nom et Python m'affiche la valeur renvoyée*

▷ *la valeur renvoyée peut être utilisée pour l'affectation d'une variable*

Attention ! Attention ! Attention ! Attention ! Attention ! Attention ! Attention ! Attention ! Attention ! Attention ! Attention ! Attention ! Attention !

Lorsqu'on définit la fonction, les paramètres ne sont pas affectés (∼ ils n'existent pas). C'est lorsque je ferai appelle à la fonction que je leur donnerai une valeur.
On remarque alors que la commande *input*() qui permet de demander à l'utilisateur une valeur **n'est quasiment plus utile** !

Application 1 :

1. *Ecrire une fonction racine qui prend en entrée un nombre et renvoie la valeur de sa racine carrée.*
2. *Ecrire une fonction somme qui prend en entrée deux entiers a et b et renvoie la somme de tous les entiers entre a et b inclus.*
3. *Ecrire une fonction minimum qui prend en argument deux nombres et qui affiche le minimum des deux.*
4. *(*) Même question mais avec 3 nombres.*

1.2 Fonction et procédure

Ainsi présentée, une fonction est très proche d'une fonction mathématiques. Néanmoins on peut faire d'autres choses.

Exemple 3 : Une fonction peut ne rien renvoyer : on appelle ça une ***procédure****.*

```
def dire_bonjour(nom,humeur):
    if humeur=="bonne":
        print("Bonjour ", nom ,", j'espère que tu vas bien !")
    elif humeur=="mauvaise":
        print("... Bonjour ...")
```

Attention ! ne pas confondre paramètres et variables ! !
Les paramètres dans la définition d'une fonction "n'existe pas" ! Ils ne servent qu'à la définir. Dans l'exemple précédent, la variable *nom* n'existe pas.

Méthode en DS. Paramètres d'une fonction
On fera bien attention de ne pas utiliser des noms de variables comme paramètres d'une fonction !

> ***Application 2 :*** *Ecrire une procédure table_multiplication qui prend en entrée deux nombres n etp et affiche les p premiers multiples de n.*

1.3 Variables locales et globales

▶ Définition

Propriété. Toute variable définie au sein d'une fonction n'existe qu'au sein de cette fonction. Elle est "oublier" dès que l'exécution de la fonction est terminée.
On parle de **variable locale**.

En revanche, une variable définie en dehors d'une fonction peut être utilisé au sein de cette dernière.
On parle de **variable globale**.

```
c=3*10**8
def energie_masse(masse):
    energie=masse*c**2
    return energie
```

Propriété. Toute variable définie en dehors d'une fonction existe même au sein d'une fonction. On parle de **variable globale**.

▶ Manipulation de variable globale

Il est inefficace de modifier une variable globale dans une fonction : les modifications seront utilisées dans la fonction mais oubliées dès que la fonction est exécutée.

Méthode en DS. Modifier une variable globale
Sauf volonté explicite, on ne modifie pas de variable globale dans une fonction.

MAIS ....

Les procédures servent souvent à modifier des variables globale. On utilise pour cela la commande *global* qui permet de dire à Python qu'on "sait ce qu'on fait" et qu'on va modifier les variables globales suivante "en connaissance de cause".

```
>>> a=2
def f(x):
    a=3
    return(a*x)
...

>>> f(2)
6

>>> print(a)
2
```

```
>>> a=2
def f(x):
    global a
    a=3
    return(a*x)
...

>>> print(a)
2

>>> f(2)
6

>>> print(a)
3
```

1.4 Attention ! *return*, *print* et *input* Attention !

On ne confondra pas *print* et *return* ! !

▷ *print* **affiche** et ne permet donc pas des affectations
▷ *return* **renvoie** et permet donc des affectations

Une fonction sans *return* n'est pas très utile !

input ne sert pas dans une fonction ! On précise les valeurs des paramètres lors de l'appel de la fonction.

Pas de *input* dans une fonction !

Il existe évidemment des exceptions ...

> ***Application 3 :***
> 1. *Ecrire une fonction factorielle reçoit un entier naturel n et qui renvoie $n!$ si $n > 0$ et affiche "Donner un nombre positif !" sinon.*
> 2. *Ecrire une fonction test_parite qui prend en entrée un entier n et qui renvoie un booléen suivant la parité du nombre.*

2 Commentaire, documentation et intérêt des fonctions

Cette partie n'a pas d'application et est à travailler à la maison en autonomie.

2.1 Commenter un script

A présent qu'on commence à écrire des programmes conséquents, une clef de la programmation est de **commenter ses scripts**. On utilise pour se faire la commande # : tout se qui est écrit après ne sera pas lu par Python.

```
N=100 #nombre d'ittérations

for k in range(1,N+1) : #N+1 car on veut aller jusqu'à N
    #test de parité
    test=k%2==0
    if test==True : #cas où le nombre est pair
        print(k," est pair")
    else: #sinon le nombre est impair
        print(k," est impair")
```

L'intérêt est qu'une personne extérieur peut le lire et le comprendre sans trop de mal. Cette personne extérieur peut très bien être nous même quelque semaines/mois après l'écriture du code ! !

2.2 Documenter une fonction

Comme nous le verrons plus tard, l'intérêt d'une fonction est d'y faire appel même bien longtemps après l'avoir écrit. On peut même utiliser des fonctions qu'on n'a pas nous même écrit en les important depuis des bibliothèques.
Il est alors important de laisser au sein de la définition de la fonction un "manuel d'utilisateur". C'est ce qu'on appelle **documenter une fonction**.

En Python, on peut documenter une fonction à l'aide d'une "docstring" : c'est une chaîne de caractères délimitée par des triples guillemets que l'on place sous l'en-tête de la fonction.

La fonction *help* permet d'afficher la docstring. Elle apparaît également quand on tape le nom de la fonction dans le shell.

```
def test_parite(x):
    """ Prends en argument un nombre x, renvoie true si le nombre est
pair, False sinon"""
    if x%2==0:
        return(True)
    else :
        return(False)
```

```
>>> help(test_parite)
Help on function test_parite in module __main__:

test_parite(x)
    Prends en argument un nombre x, renvoie true si le nombre est pair, Fals
```

2.3 Intérêt des fonctions

Le grand intérêt des fonction en informatique est qu'on peut les appeler au sein même d'autre fonctions. Cela permet :

- ▷ de séparer la résolution d'un problème en plusieurs étapes. C'est la méthode par analyse descendante. Cette méthode est utilisée pour résoudre un problème complexe : en procédant par raffinement successifs, le problème principal est décomposé en sous-problèmes moins compliqués. Chaque sous-problème est ensuite décomposé en sous-problèmes de plus en plus simples jusqu'à parvenir à des problèmes tellement élémentaires que la solution en est évidente.

- ▷ de créer des bibliothèques de fonction "de la vie de tous les jours" auxquelles on pourra faire appel lorsqu'on travaille.
  Exemple : les fonctions valeur absolue, racine carrée, moyenne, ...

  L'intérêt de Python est que de nombreuses bibliothèque existe déjà. On peut demander à Python d'aller les utiliser via la commande *import*.

```
>>> sqrt(4)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
NameError: name 'sqrt' is not defined

>>> import math

>>> math.sqrt(4)
2.0
```

Python ne connaît que très peu de fonction. Par contre on peut lui demander "d'apprendre" toute les fonctions de la bibliothèque *math* qui contient un très grand nombre de fonctions usuelles.
On peut alors faire appel à ses fonctions en précisant *math.* devant le nom.

La personne qui a créé la bibliothèque *math* a documenter sa fonction *sqrt*. Je peux y faire appel avec la commande *help*.

```
>>> help(math.sqrt)
Help on built-in function sqrt in module math:

sqrt(x, /)
    Return the square root of x.
```

3 Exercices

A faire pendant les vacances !

Exercice 1 - Cubes :

Ecrire une procédure *cubes* qui, recevant un entier naturel n, affiche les cubes des entiers compris entre 0 et n.
Attention ! ce sont les cubes qui doivent être plus petit que n, pas les entiers : $k^3 < n$.

Exercice 2 - Année bissextile :

Les années divisibles par 4 sont bissextiles, sauf si elles sont divisibles par 100 mais les années divisibles par 400 sont quand même bissextiles. Par exemple, 2012 et 2000 étaient bissextiles, mais pas 1900 alors que 1600 oui.
Ecrire une fonction *bissextile* qui, recevant une année, renvoie $True$ si elle est bissextile et $False$ sinon.

Exercice 3 - Entiers parfaits :

Soit n un entier naturel. On appelle diviseur propre de n tout entier naturel différent de n qui divise n. On dit que n est parfait s'il est égal à la somme de ses diviseurs propres.
Attention ! Il ne faut pas compter n comme diviseur propre de n !

Exemple : $6 = 1 \times 2 \times 3 = 1 + 2 + 3$

1. Recopier et compléter le programme suivant de sorte à ce qu'il renvoie la somme des diviseurs propres de n.

```
def somme_diviseur(n):
    s=0
    i=1
    while ...
        if ...
            s+= i
        i+= 1
    return ...
```

2. Créer une fonction qui utilise la fonction *somme_diviseur*, qui prend en entrée un entier naturel et qui renvoie $True$ s'il est parfait, $False$ sinon.
3. Afficher les entiers parfaits pairs inférieurs ou égaux à 5000

Exercice 4 - Suite de Syracuse :

On considère l'algorithme suivant. Un entier naturel n non nul est donné. S'il est pair, on le divise par 2. S'il est impair, on le multiplie par 3 et on ajoute 1. Puis on recommence : si le nombre obtenu est pair, on le divise par 2, et s'il est impair on le multiplie par 3 et on ajoute 1. Et on continue...
1. Appliquer cet algorithme (à la main) pour $n = 1, 2, 3, 4, 5, 6, 7$. Que peut-on conjecturer ?
2. Ecrire une fonction qui, recevant un entier n, lui applique l'algorithme et affiche la suite des nombres obtenus jusqu'à ce qu'on arrive à 1.
3. Pour quel entier $6 < n < 1000$ le nombre d'itérations effectuées est-il maximal ?