SOMMAIRE

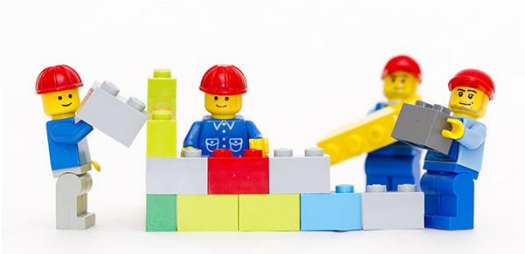
Chapitre 6 : Les fonctions

Les fonctions

I. Introduction

Dans ce chapitre, nous allons apprendre à structurer nos programmes en petits bouts, dans le but de rendre les grands programmes plus lisible et plus facile à maintenir. Ces petits bouts de code sont justement ce qu'on appelle… des fonctions !

Donc un programme sera vu comme un rassemblement de petit bloc de codes, comme le jeu de lego.



Le but des fonctions est de simplifier le code source, pour ne pas avoir à retaper le même code plusieurs fois d'affilée. Elles permettent de regrouper plusieurs instructions dans un bloc qui sera appelé grâce à un nom.

Nous avons utilisé pas mal de fonctions depuis le début de ce tutoriel. On citera pour mémoire print, input et range, sans compter quelques autres. Mais vous devez bien vous rendre compte qu'il existe un nombre incalculable de fonctions déjà construites en Python. Toutefois, vous vous apercevrez aussi que, très souvent, un programmeur crée ses propres fonctions. C'est le premier pas que vous ferez, dans cette partie, vers la **modularité**.

II. Définition d'une fonction

Une fonction exécute des actions et renvoie un résultat. C'est un **morceau de code** qui sert à faire quelque chose de précis.
On dit qu'une fonction possède une entrée et une sortie. La fig. suivante représente une fonction schématiquement.

Lorsqu'on appelle une fonction, il y a trois étapes.

1. L'entrée: on fait « rentrer » des informations dans la fonction (en lui donnant des informations avec lesquelles travailler).
2. Les calculs : grâce aux informations qu'elle a reçues en entrée, la fonction travaille.
3. La sortie : une fois qu'elle a fini ses calculs, la fonction renvoie des résultats. C'est ce qu'on appelle la sortie, ou encore le retour.

Concrètement, on peut imaginer par exemple une fonction appelée triple qui calcule le triple du nombre qu'on lui donne, en le multipliant par 3 (fig. suivante). Bien entendu, les fonctions seront en général plus compliquées.

La fonction « triple » multiplie le nombre en entrée par 3

III. Déclaration d'une fonction

On crée une fonction selon le schéma suivant :

```
def  nom_de_la_fonction  (parametre1 ,  parametre2 , parametre3 ,  parametreN ) :
  # Bloc d'instructions
```

- **def**, mot-clé qui est l'abréviation de « define » (définir, en anglais) et qui constitue le prélude à toute construction de fonction.

- Le nom de la fonction, qui se nomme exactement comme une variable (nous verrons par la suite que ce n'est pas par hasard). N'utilisez pas un nom de variable déjà instanciée pour nommer une fonction.

- La liste des paramètres qui seront fournis lors d'un appel à la fonction. Les paramètres sont séparés par des virgules et la liste est encadrée par des parenthèses ouvrante et fermante (là encore, les espaces sont optionnels mais améliorent la lisibilité).

- Les deux points, encore et toujours, qui clôturent la ligne.

Les parenthèses sont obligatoires, quand bien même votre fonction n'attendrait aucun paramètre.

Nous allons reprendre le code de la table de multiplication pour initier la syntaxe de création d'une fonction.

Nous allons emprisonner notre code calculant la table de multiplication par 7 dans une fonction que nous appellerons **table_par_7**.

Le code pour mettre notre table de multiplication par 7 dans une fonction serait donc :

```
def table_par_7():
    nb = 7
    i = 0 # Notre compteur ! L'auriez-vous oublié ?
    while i < 10: # Tant que i est strictement inférieure à 10,
        print(i + 1, "*", nb, "=", (i + 1) * nb)
        i = i + 1 # On incrémente i de 1 à chaque tour de boucle.
```

Quand vous exécutez ce code à l'écran, il ne se passe rien. Une fois que vous avez retrouvé les trois chevrons, essayez d'appeler la fonction :

```
1 * 7 = 7
2 * 7 = 14
3 * 7 = 21
4 * 7 = 28
5 * 7 = 35
6 * 7 = 42
7 * 7 = 49
8 * 7 = 56
9 * 7 = 63
10 * 7 = 70
```

Bien, c'est, exactement ce qu'on avait réussi à faire aux séances précédentes et l'intérêt ne saute pas encore aux yeux. L'avantage est que l'on peut appeler facilement la fonction et réafficher toute la table sans avoir besoin de tout réécrire !

Mais, si on saisit des paramètres pour pouvoir afficher la table de 5 ou de 8… ?

Oui, ce serait déjà bien plus utile. Le code de la fonction est :

```
def table(nb):
    i = 0
    while i < 10: # Tant que i est strictement inférieure à 10,
        print(i + 1, "*", nb, "=", (i + 1) * nb)
        i = i + 1 # On incrémente i de 1 à chaque tour de boucle.
```

Et là, vous pouvez passer en argument différents nombres, table(8) pour afficher la table de multiplication par 8 par exemple.

On peut aussi envisager de passer en paramètre le nombre de valeurs à afficher dans la table.

```
def table(nb, max):
  i = 0
  while i < max: # Tant que i est strictement inférieure à la variable max,
    print(i + 1, "*", nb, "=", (i + 1) * nb)
    i = i + 1
```

Si vous tapez à présent **table(11, 20)**, l'interpréteur vous affichera la table de 11, de 1*11 à 20*11.

Dans le cas où l'on utilise plusieurs paramètres sans les nommer, comme ici, il faut respecter l'ordre d'appel des paramètres, cela va de soi. Si vous commencez à mettre le nombre d'affichages en premier paramètre alors que, dans la définition, c'était le second, vous risquez d'avoir quelques surprises. Il est possible d'appeler les paramètres dans le désordre mais il faut, dans ce cas, préciser leur nom: nous verrons cela plus loin.

IV. L'instruction return :

Ce que nous avons fait était intéressant, mais nous n'avons pas encore fait le tour des possibilités de la fonction. Et d'ailleurs, même à la fin de ce chapitre, il nous restera quelques petites fonctionnalités à voir. Si vous vous souvenez bien, il existe des fonctions comme **print** qui ne renvoient rien (attention, « renvoyer » et « afficher » sont deux choses différentes) et des fonctions telles que **input** qui renvoient une valeur. Vous pouvez capturer cette valeur en plaçant une variable devant (exemple **variable =input()**). En effet, les fonctions travaillent en général sur des données et renvoient le résultat obtenu, suite à un calcul par exemple.

Prenons un exemple simple : une fonction chargée de mettre au carré une valeur passée en argument. Je vous signale au passage que Python en est parfaitement capable sans avoir à coder une nouvelle fonction, mais c'est pour l'exemple.

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

L'instruction *return* signifie qu'on va **renvoyer** la valeur, pour pouvoir la récupérer ensuite et la stocker dans une variable par exemple. Cette instruction arrête le déroulement de la fonction, le code situé après le return ne s'exécutera pas.

```
variable = carre(5)
```

La variable contiendra, après exécution de cette instruction, 5 au carré, c'est-à-dire 25.

V. Les fonctions anonymes

Python permet la création de fonctions anonymes (i.e. sans nom et donc non définie par def) à l'aide du mot-clé lambda. Une fonction anonyme ne peut pas avoir d'instruction return et doit forcément retourner une expression. De telles fonctions permettent de représenter tout sous forme de fonctions sans réellement en définir explicitement.

Exemples de syntaxes de déclaration de quelques fonctions :

Avec def	Avec lambda
def cree(n) : return n*n	cree = lambda n : n*n

def absolute(n) :
    if n>=0 :
        return n
    return -n
	absolute = lambda n : n if n>=0 else -n

def min(a,b) :
    if a>b :
        return a
    return b
	min = lambda a,b : a if a>b else b