Chapitre 3

Les Fonctions en Python

3.1 Les fonctions

3.1.1 Définition de fonction

Une fonction est un algorithme indépendant.
L'appel (avec ou sans paramètre) de la fonction déclenche l'exécution de son bloc d'instructions.
Une fonction se termine en retournant ou non une valeur

Remarque 1 :

1. Une fonction a aussi été appelée **sous-programme**
2. Nous avons déjà rencontré des fonctions comme, par exemple `print()`, `type()` ou `input()`
3. La structure d'une fonction est la suivante :

```
def Nom_De_La_Fonction(parametrel, parametre2, parametre3,....,parametreN):
        Bloc d'instructiors
```

Si nous décortiquons la définition de la fonction, on trouve dans l'ordre :
→ `def`, mot-clé qui est l'abréviation du « define » *(définir, en anglais)* et qui est le prélude à toute construction de fonction.
→ Le nom de Ia fonction, qui se nomme exactement comme une variable *(nous verrons par la suite que ce n'est pas par hasard)*. Par conséquent, 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 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.

4. Trois étapes sont toujours nécessaires à l'exécution d'une fonction :
   (a) Le programme appelant interrompt son exécution
   (b) La fonction appelée effectue son bloc d'instructions.
   (c) Une fois le bloc d'instructions de la fonction terminé,le programme appelant reprend alors son exécution

Remarque 2 :

1. Une fonction effectue une tâche. Pour cela, elle reçoit éventuellement des arguments et renvoie éventuellement quelque chose.
   L'algorithme utilisé au sein de la fonction n'intéresse pas directement l'utilisateur.

Par exemple, il est inutile de savoir comment la fonction `math.cos()` calcule un cosinus. On a juste besoin de savoir qu'il faut lui passer en argument un angle en radian et qu'elle renvoie le cosinus de cet angle. Ce qui se passe à l'intérieur de la fonction ne regarde que le programmeur.

2. Chaque fonction effectue, en général, une tâche unique et précise. Si cela se complique, il est plus judicieux d'écrire plusieurs fonctions *(qui peuvent éventuellement s'appeler les unes les autres)*. Cette **modularité** améliore la qualité générale et la lisibilité du code. Vous verrez qu'en Python, les fonctions présentent une grande flexibilité.
3. Le programmeur doit penser à concevoir et écrire des fonctions pour améliorer son programme. Il y gagnera sur plusieurs points :
   — **Le code des algorithmes est plus simple, plus clair et plus court.** Dans un algorithme, appeler une fonction se fait en une ligne et la fonction peut être appelée à plusieurs reprises
   — Une seule modification dans la fonction sera automatiquement répercutée sur tous les algorithmes qui utilisent cette fonction.
   — L'utilisation de fonctions génériques dans des algorithmes différents permet de réutiliser son travail et de gagner du temps

Exemple 1 :

Le premier exemple d'utilisation de fonction : somme et maximum de deux réels : une fonction ramène la somme de deux réels, une autre le plus grand des 2

1. **Fonction somme de 2 réels**

```
#-*-coding:Latin-1 -*



#algo Fonction somme de 2 réels

def somme(x,y):
    x=float(x)
    y=float(y)
    s=x+y
    print("La somme de ",x,"et ",y ," est ", s)
```

Et à l'appel de cette fonction, nous avons :

```
>>> somme(15,16)
La somme de  15.0 et  16.0  est  31.0
>>> somme(265,-256)
La somme de  265.0 et  -256.0  est  9.0
>>>
```

2. **Fonction maximum de 2 réels**

```
#-*-coding:Latin-1 -*



#algo Fonction maximum de 2 réels

def maximum(x,y):
    if (x>y):
        grd = x
    else:
        grd = y
    print("Le plus grand des 2 entiers ",x," et ",y," est", grd)
```

Et, comme tout à l'heure, à l'appel de cette fonction, nous avons :

```
>>> maximum(25,63)
Le plus grand des 2 entiers  25  et  63  est 63
>>> maximum(-96,-100)
Le plus grand des 2 entiers  -96  et  -100  est -96
>>>
```

3.1.2 Définir une fonction

La signature d'une fonction décrit les éléments permettant de l'appeler correctement
En Python, la signature d'une fonction est seulement donnée par le nom de la fonction

Remarque 3 :

1. **Attention**, dans d'autres langages comme C++ ou Java la signature est beaucoup plus importante ; elle est donnée par :
   — Le nom de la fonction
   — Le type des paramètres
   — Le type de la valeur retournée
   Ainsi, dans ces langages, deux fonctions différentes peuvent avoir le même nom, mais des types de variables différents ; ce seront donc des fonctions différentes.
2. En Python, il est impossible de définir 2 fonctions avec le même nom : si vous le faites, l'ancienne fonction est écrasée par la nouvelle

Exemple 2 :

Nous vous proposons, ici, un exemple où une fonction écrase une autre. Nous vérifions, en plus que l'adresse de stockage est identique

```
>>> def exemple():
        print("fonction à écraser")


>>> exemple
<function exemple at 0x00000265F90F2708>
#Adresse de stockage de la fonction exemple()
>>> exemple()
fonction à écraser
>>> def exemple():
        print("Fonction qui écrase")


>>> exemple()
Fonction qui écrase
>>> exemple
<function exemple at 0x00000265F90F25E8>
#C'est bien la même adresse de stockage de la fonction exemple()
>>>
```

Exercice 1 :

Ecrire un programme qui affiche tous les nombres impairs entre 0 et n, dans l'ordre croissant, le nombre n étant donné par l'opérateur

3.1.3 Renvoi de résultats : la fonction « *return* »

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

Remarque 4 :

Jusqu'à présent, nos fonctions n'ont fait qu'afficher leur résultat après qu'on les ait appelées.
En pratique, cette façon de procéder est rarement utilisée et ceci pour deux raisons :

→ D'une part, nous n'avons aucun contrôle sur le résultat affiché puisque celui ci est affiché dès que la fonction a fini de s'exécuter

→ Ensuite car nous ne pouvons pas utiliser ce résultat pour effectuer de nouvelles opérations dans la suite

Or, en programmation, nous voudrons souvent récupérer le résultat d'une fonction afin de le ré-utiliser dans le reste de notre script. Pour cela, il va falloir qu'on demande à notre fonction de retourner *(renvoyer)* le résultat de ses opérations. Nous allons pouvoir faire cela en `Python` grâce à l'instruction `return`.
Attention cependant : l'instruction `return` va terminer l'exécution d'une fonction, ce qui signifie qu'on placera généralement cette instruction en fin de fonction puisque le code suivant une instruction `return` dans une fonction ne sera jamais lu ni exécuté.

Exemple 3 :

```
>>> def carre_cube(x):
        return x**2, x**3
>>> carre_cube(2)
(4, 8)
# Affectation multiple
>>> z1,z2 = carre_cube(7)
>>> z1
49
>>> z2
343
>>>
```

Exercice 2 :

Ecrire la fonction qui prend en entrée la température en degrés Celsius et la convertit en degré Farheneit.
La formule de conversion est donnée par : $C° = (F° - 32) \times \frac{5}{9}$

3.1.4 Fonction et arguments positionnels

Lorsqu'on définit une fonction def fct(x, y) : les arguments x et y sont appelés arguments positionnels *(en anglais positional arguments)*.
Il est strictement obligatoire de les préciser lors de l'appel de la fonction.
De plus, il est nécessaire de respecter le même ordre lors de l'appel que dans la définition de la fonction.

Exemple 4 :

Cet exemple montre l'importance du positionnement. Nous faisons ici le calcul de la fonction $f(x, y) = xy^2$

```
  def fois(x,y):
    return x*(y**2)

fois(2,1)
2
fois(1,2)
4
```

```
  def fois(x,y):
    return x*(y**2)

fois(2)
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    fois(2)
TypeError: fois() missing 1 required
       positional argument: 'y'
```

On constate que passer un seul argument à une fonction qui en attend deux conduit à une erreur.

3.1.5 Fonction et arguments par mots clefs

Un argument défini avec une syntaxe def fct(arg=val) : est appelé argument par mot-clé *(en anglais keyword argument)*.
Le passage d'un tel argument lors de l'appel de la fonction est facultatif.

Exemple 5 :

1. Les arguments par mot-clé sont pris dans l'ordre dans lesquels on les passe lors de l'appel. Comment pourrions-nous faire si on souhaitait préciser l'argument par mot-clé z et garder les valeurs de x et y par défaut ? Simplement en précisant le nom de l'argument lors de l'appel :

```
def fct(x=0, y=0, z=0):
    return x, y, z

fct()
(0, 0, 0)
fct(15,23,5)
(15, 23, 5)
fct(z=5)
(0,0,5)
def fct(x=0, y=0, z=0):
    return x, y+x, z+y+x

fct()
(0, 0, 0)
fct(15,23,5)
(15, 38, 43)
fct(z=12,y=-5,x=5)
(5, 0, 12)
```

2. Que se passe-t-il lorsque nous avons un mélange d'arguments positionnels et par mot-clé ? Et bien les arguments positionnels doivent toujours être placés avant les arguments par mot-clé :

```
def fct(a, b, x=0, y=0, z=0):
    return a, b, a+x, b+y, a-z

fct(2,3,z=3,x=4,y=5)
(2, 3, 6, 8, -1)
fct(1,5)
(1, 5, 1, 5, 1)
fct(z=1)
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    fct(z=1)
TypeError: fct() missing 2 required positional arguments: 'a' and 'b'
```

On peut toujours passer les arguments par mot-clé dans un ordre arbitraire à partir du moment où on précise leur nom. Par contre, si les deux arguments positionnels a et b ne sont pas passés à la fonction, `Python` renvoie une erreur.

Exercice 3 :

Ecrire une fonction qui nous donne $n!$ *(sans utiliser la récursivité)*

Exercice 4 :

Calculez et affichez à l'écran 2^k avec k variant de 0 à n inclus.

Exercice 5 :

Créez une fonction qui calcule la distance euclidienne dans $\mathbb{R}^3$ entre 2 points $A\,(x_A, y_A, z_A)$ et $B\,(x_B, y_B, z_B)$

Exercice 6 :

Écrire une fonction `suite` qui, étant donnés deux arguments n et a renvoie la valeur de u_n, où la suite $(u_k)_{k\in\mathbb{N}}$ est la suite définie par :

$$\begin{cases} u_0 = 1 \\ u_{n+1} = \dfrac{1}{2}\left(u_n + \dfrac{a}{u_n}\right) \end{cases}$$

C'est la suite des babyloniens.

3.1.6 Variables locales, variables globales

1. Une variable déclarée à la racine du module est une variable globale ; elle est donc dite globale lorsqu'elle est créée dans le programme principal. Elle sera visible partout dans le programme.
2. Une variable déclarée à l'intérieur d'une fonction est une variable locale ; elle n'existera et ne sera visible que lors de l'exécution de ladite fonction.

Exemple 6 :

1. Un premier exemple très *(trop !)* simple

```
#-*-coding:Latin-1 -*
#algo Variable_Locale_Variable_Globale

x="bonjour"
def test(y):
        z=" tout le monde"
        print(y+z)
```

   Dans cet exemple, `x` est une variable globale et `y` est une variable locale

2. Par défaut, tout identificateur utilisé dans le corps d'une fonction est local à celle-ci.
   Par exemple :

```
#-*-coding:Latin-1 -*
#algo Variable_Locale_Variable_Globale_02


def fonc(y) : # y, x et z sont affectés dans fonc => locaux
    x = 3 # ce nouvel x est local et masque le x global
    z = x + y
    return z

x=26
fonc(x)
29
x #on demande d'imprimer la variable globale x
26
```

3. Autre exemple :

```
import math
def volume_boule(r):
    volume = 4/3 * math.pi * r**3
    return volume

volume_boule(3)
113.09733552923254

v=volume_boule(5)
print("Le volume de la boule de rayon 5 est ",v)
```

```
Le volume de la boule de rayon 5 est  523.5987755982989
print(volume)
#Réponse de l'ordinateur:
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    print(volume)
NameError: name 'volume' is not defined
```

La variable `volume` est une variable locale, uniquement liée à la fonction. Elle n'existe pas dans le programme général