V. Fonctions en Python

Lorsqu'une tâche doit être réalisée plusieurs fois par un programme avec seulement des paramètres différents, on peut l'isoler au sein d'une fonction. Cette approche est également intéressante si la personne qui définit la fonction est différente de celle qui l'utilise. Par exemple, nous avons déjà utilisé la fonction **print()** qui avait été définie par quelqu'un d'autre.

Nous avons déjà rencontré diverses fonctions prédéfinies : **print**(), **input**(), **range**(), **len**().

1. *Définition d'une fonction-def*

La syntaxe Python pour la définition d'une fonction est la suivante :

```
def nom_fonction(liste de paramètres):
        bloc d'instructions
```

Vous pouvez choisir n'importe quel nom pour la fonction que vous créez, à l'exception des mots-clés réservés du langage, et à la condition de n'utiliser aucun caractère spécial ou accentué (le caractère souligné « _ » est permis). Comme c'est le cas pour les noms de variables, on utilise par convention des minuscules, notamment au début du nom (les noms commençant par une majuscule seront réservés aux classes).

⇒ ***Corps de la fonction***

Comme les instructions **if**, **for** et **while**, l'instruction **def** est une instruction composée. La ligne contenant cette instruction se termine obligatoirement par un deux-points **:**, qui introduisent un bloc d'instructions qui est précisé grâce à l'indentation. Ce bloc d'instructions constitue le **corps de la fonction**.

⇒ ***Fonction sans paramètre***

```
def compteur3():
        i = 0
        while i < 3:
                print(i)
                i = i + 1


print("bonjour")
compteur3()
compteur3()
```

➲ Affichage après exécution :

```
bonjour
0
1
2
0
1
2
```

En entrant ces quelques lignes, nous avons défini une fonction très simple qui compte jusqu'à 2. Notez bien les parenthèses, les deux-points, et l'indentation du bloc d'instructions qui suit la ligne d'en-tête (c'est ce bloc d'instructions qui constitue le corps de la fonction proprement dite).

Après la définition de la fonction, on trouve le programme principal qui débute par l'instruction **print("bonjour")**. Il y a ensuite au sein du programme principal, l'appel de la fonction grâce à **compteur3().**

Nous pouvons maintenant réutiliser cette fonction à plusieurs reprises, autant de fois que nous le souhaitons.

Nous pouvons également l'incorporer dans la définition d'une autre fonction.

```
def compteur3():
        i = 0
        while i < 3:
                print(i)
                i = i + 1

def double_compteur3():
        compteur3()
        compteur3()

print("bonjour")
double_compteur3()
```

Une première fonction peut donc appeler une deuxième fonction, qui elle-même en appelle une troisième, etc.

Créer une nouvelle fonction offre l'opportunité de donner un nom à tout un ensemble d'instructions. De cette manière, on peut simplifier le corps principal d'un programme, en dissimulant un algorithme secondaire complexe sous une commande unique, à laquelle on peut donner un nom explicite.

Une fonction est donc en quelque sorte une nouvelle instruction personnalisée, qu'il est possible d'ajouter librement à notre langage de programmation.

- ***Fonction avec paramètre***

```
def compteur(stop):
        i = 0
        while i < stop:
                print(i)
                i = i + 1

compteur(4)
compteur(2)
```

Affichage après exécution :

```
0
1
2
3
0
1
```

➲ Utilisation d'une variable comme argument

L'argument que nous utilisons dans l'appel d'une fonction peut être une variable.

```
def compteur(stop):
        i = 0
        while i < stop:
                print(i)
                i = i + 1
a = 5
compteur(a)
```

Affichage après exécution :

```
0
1
2
3
4
```

Dans l'exemple ci-dessus, l'argument que nous passons à la fonction `compteur()` est le contenu de la variable `a`. A l'intérieur de la fonction, cet argument est affecté au paramètre `stop`, qui est une tout autre variable.

Notez donc bien dès à présent que :

-Le nom d'une variable que nous passons comme argument n'a rien à voir avec le nom du paramètre correspondant dans la fonction.

-Ces noms peuvent être identiques si vous le voulez, mais vous devez bien comprendre qu'ils ne désignent pas la même chose (en dépit du fait qu'ils puissent contenir une valeur identique).

- ***Fonction avec plusieurs paramètres***

La fonction suivante utilise trois paramètres : **start** qui contient la valeur de départ, **stop** la borne supérieure exclue comme dans l'exemple précédent et **step** le pas du compteur.

```
def compteur_complet(start, stop, step):
        i = start
        while i < stop:
                print(i)
                i = i + step


compteur_complet(1, 7, 2)
```

➲ Affichage après exécution :

```
1
3
5
```

Pour définir une fonction avec plusieurs paramètres, il suffit d'inclure ceux-ci entre les parenthèses qui suivent le nom de la fonction, en les séparant à l'aide de virgules.

Lors de l'appel de la fonction, les arguments utilisés doivent être fournis dans le même ordre que celui des paramètres correspondants (en les séparant eux aussi à l'aide de virgules). Le premier argument sera affecté au premier paramètre, le second argument sera affecté au second paramètre, et ainsi de suite.

2. *Variables locales, 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 à distinguer de l'espace de noms **global** où se trouvait 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. C'est par exemple le cas des variables **start**, **stop**, **step** et `i` dans l'exemple précédent.

A chaque fois que nous définissons des variables à l'intérieur du corps d'une fonction, ces variables ne sont accessibles qu'à la fonction elle-même. On dit que ces variables sont des **variables locales** à la fonction. Une variable locale peut avoir le même nom qu'une variable de l'espace de noms global mais elle reste néanmoins indépendante.

Les contenus des variables locales sont stockés dans l'espace de noms local qui est inaccessible depuis l'extérieur de la fonction.

Les variables définies à l'extérieur d'une fonction sont des variables globales. Leur contenu est « visible » de l'intérieur d'une fonction, mais la fonction ne peut pas le modifier.

➲ Exemple

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

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

➲ Affichage après exécution :

```
2 5
2 7
```

- ***Utilisation d'une variable globale - global***

Il peut se faire par exemple que vous ayez à définir une fonction qui soit capable de modifier une variable globale. Pour atteindre ce résultat, il vous suffira d'utiliser l'instruction **global**. Cette instruction permet d'indiquer - à l'intérieur de la définition d'une fonction - quelles sont les variables à traiter globalement.

On va ici créer une fonction qui a accès à la variable globale **b**.

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

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

➲ Affichage après exécution :

```
2 5
2 5
```

3. *Fonctions et procédures*

pour les puristes, les fonctions que nous avons décrites jusqu'à présent ne sont pas tout à fait des fonctions au sens strict, mais plus exactement des procédures. Une « vraie » fonction (au sens strict) doit en effet renvoyer une valeur lorsqu'elle se termine. Une « vraie » fonction peut s'utiliser à la droite du signe égale dans des expressions telles que y = sin(a). On comprend aisément que dans cette expression, la fonction sin() renvoie une valeur (le sinus de l'argument) qui est directement affectée à la variable y.

Voici un exemple extrêmement simple :

```
def cube(w):
    return w**3
```

```
>>> cube(3)
27
>>> a = cube(4)
>>> a
64
```

4. *Valeurs par défaut pour les paramètres*

Dans la définition d'une fonction, il est possible de définir un argument par défaut pour chacun des paramètres. On obtient ainsi une fonction qui peut être appelée avec une partie seulement des arguments attendus.

➲ Exemples :

```
def politesse(nom, titre ="Monsieur"):
    print("Veuillez agréer,", titre, nom, ", mes salutations distinguées.")
```

```
>>> politesse("ESSADDOUKI")
Veuillez agréer, Monsieur ESSADDOUKI , mes salutations distinguées.
>>> politesse('Sara', 'Mademoiselle')
Veuillez agréer, Mademoiselle Sara , mes salutations distinguées.
```

Lorsque l'on appelle cette fonction en ne lui fournissant que le premier argument, le second reçoit tout de même une valeur par défaut. Si l'on fournit les deux arguments, la valeur par défaut pour le deuxième est tout simplement ignorée.

5. *Arguments avec étiquettes*

Dans la plupart des langages de programmation, les arguments que l'on fournit lors de l'appel d'une fonction doivent être fournis exactement dans le même ordre que celui des paramètres qui leur correspondent dans la définition de la fonction.

Python autorise cependant une souplesse beaucoup plus grande. Si les paramètres annoncés dans la définition de la fonction ont reçu chacun une valeur par défaut, sous la forme déjà décrite ci-dessus, on peut faire appel à la fonction en fournissant les arguments correspondants **dans n'importe quel ordre, à la condition de désigner nommément les paramètres correspondants**.

➲ Exemple :

```
def oiseau(voltage=100, action="danser la java"):
    print("Ce perroquet ne pourra pas", action)
    print("si vous le branchez sur", voltage, "volts !")
```

```
>>> oiseau(voltage=250, action="vous approuver")
Ce perroquet ne pourra pas vous approuver
si vous le branchez sur 250 volts !
>>> oiseau()
Ce perroquet ne pourra pas danser la java
si vous le branchez sur 100 volts !
```

6. Utilisation des fonctions dans un script

Pour cette première approche des fonctions, nous n'avons utilisé jusqu'ici que des fonctions dans des scripts et pas de programme principal dans le script. Veuillez donc essayer vous-même le petit programme ci-dessous, lequel calcule le volume d'une sphère a l'aide de la formule :

➲ Exemple :

```
import numpy as np

def cube(n):
        return n**3

def volume_sphere(r):
        return 4 / 3 * np.pi * cube(r)

r = float(input("Entrez la valeur du rayon : "))
print("Le volume de cette sphere vaut", volume_sphere(r))
```

A bien y regarder, ce programme comporte deux parties :

- Les deux fonctions cube() et volume_sphere()
- Le **corps principal du programme**.

Dans le corps principal du programme, il y a un appel de la fonction volume_sphere(). A l'intérieur de la fonction volume_sphere(), il y a un appel de la fonction cube().

Notez bien que les deux parties du programme ont été disposées dans un certain ordre :

- D'abord la définition des fonctions,
- Et ensuite le corps principal du programme.

Cette disposition est nécessaire, parce que l'interpréteur exécute les lignes d'instructions du programme l'une après l'autre, dans l'ordre où elles apparaissent dans le code source. Dans le script, la définition des fonctions doit donc précéder leur utilisation.

En fait, le corps principal d'un programme Python constitue lui-même une entité un peu particulière, qui est toujours reconnue dans le fonctionnement interne de l'interpréteur sous le nom réservé __main__ (le mot « main » signifie « principal », en anglais. Il est encadré par des caractères « souligné » en double, pour éviter toute confusion avec d'autres symboles). L'exécution d'un script commence toujours avec la première instruction de cette entité __main__, où qu'elle puisse se trouver dans le listing. Les instructions qui suivent sont alors exécutées l'une après l'autre, dans l'ordre, jusqu'au premier appel de fonction. Un appel de fonction est comme un détour dans le flux de l'exécution : au lieu de passer à l'instruction suivante, l'interpréteur exécute la fonction appelée, puis revient au programme appelant pour continuer le travail interrompu. Pour que ce mécanisme puisse fonctionner, il faut que l'interpréteur ait pu lire la définition de la fonction avant l'entité __main__, et celle-ci sera donc placée en général à la fin du script.

Dans notre exemple, l'entité __main__ appelle une première fonction qui elle-même en appelle une deuxième. Cette situation est très fréquente en programmation. Si vous voulez comprendre correctement ce qui se passe dans un programme, vous devez donc apprendre à lire un script, non pas de la première à la dernière ligne, mais plutôt en suivant un cheminement analogue à ce qui se passe lors de l'exécution de ce script. Cela signifie concrètement que vous devrez souvent analyser un script en commençant par ses dernières lignes !