


Cours	Cours	
BTS SNIR	Python	
2 ^{ème} année	Les fonctions (suite)	

Dans cette leçon, nous allons voir comment faire pour que nos fonctions retournent explicitement une valeur et comprendre l'intérêt de faire retourner une valeur à nos fonctions Python.

Présentation de l'instruction `return` et cas d'utilisation

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 est affiché dès que la fonction a fini de s'exécuter et ensuite car nous ne pouvons pas utiliser ce résultat pour effectuer de nouvelles opérations.

Or, en programmation, nous voudrions souvent récupérer le résultat d'une fonction afin de l'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.

Premier exemple d'utilisation de `return` en Python

Imaginons que nous soyons en train de créer un programme relativement complexe qui effectue des séries de calculs intermédiaires pour finalement arriver à un résultat final.

Notre programme va être composé de différentes fonctions qui vont se charger d'effectuer ces différents calculs à la suite les uns des autres. Certaines fonctions vont fonctionner différemment ou même ne pas s'exécuter du tout en fonction du résultat renvoyé par la fonction précédente dans la chaîne de fonctions.

Ce type de situations est très fréquent en programmation : on exécute une première fonction qui renvoie un résultat et on injecte ce résultat dans la fonction suivante et etc. On va pouvoir faire cela avec une instruction `return`.

Pour cela, créons par exemple une fonction très simple qui renvoie la différence entre deux nombres.

```
[>>> #Création d'un fonction qui calcule la différence entre deux nombres
[... def difference(a,b):
[...     return a - b
[...
[>>> #On exécute difference() en lui passant deux nombres et on stocke le résultat dans une
      variable x
[... x = difference(10, 12) #x contient la valeur -2
```

Ici, on utilise `return` afin de demander à notre fonction de retourner son résultat. On stocke ensuite ce résultat dans une variable `x` dont on pourra se resservir dans la suite du script.

Utiliser `return` pour retourner plusieurs valeurs

Une fonction ne peut retourner qu'une donnée à la fois. Cependant, Python met à notre disposition des types de données composites comme les listes ou les tuples par exemple.

On va donc pouvoir utiliser `return` pour faire retourner "plusieurs valeurs" à la fois à nos fonctions ou pour être tout à fait exact pour leur faire retourner une donnée composite.

Pour cela, on va préciser les différentes valeurs que doit retourner `return` en les séparant par des virgules. Les valeurs retournées seront retournées dans un tuple.

```
[>>> def ordre(a, b):
[...     if a <= b:
[...         return a, b
[...     else:
[...         return b, a
[...
[>>> ordonne = ordre(50, 2)
[>>> print(ordonne)
(2, 50)
```

Les fonctions récursives

Nous avons vu dans les leçon précédente qu'une fonction pouvait exécuter une autre fonction, par exemple dans le cas où on demande à une fonction d'exécuter une fonction `print()` pour afficher une valeur.

Vous devez savoir qu'une fonction peut également s'appeler elle même dans son exécution : c'est ce qu'on appelle la récursivité. Lorsqu'on définit une fonction récursive, il faudra toujours faire bien attention à fournir une condition qui sera fausse à un moment ou l'autre au risque que la fonction s'appelle à l'infini.

L'exemple de fonction récursive par excellence est la définition d'une fonction qui calculerait une factorielle. La factorielle d'un nombre est le produit des nombres entiers inférieurs ou égaux à celui-ci; la factorielle de 4 par exemple est égale à $4 * 3 * 2 * 1$.

Créons immédiatement cette fonction :

```
[>>> def factorielle(n):  
[...     if n <= 1:  
[...         return 1  
[...     else:  
[...         return n * factorielle(n-1)  
[...  
[>>> factorielle(4)  
24  
[>>> factorielle(0)  
1
```

Ici, la condition de sortie de notre fonction est atteinte dès que la valeur passée en argument atteint ou est inférieure à 1. Expliquons comment fonctionne cette fonction en détail. Si on passe une valeur inférieure ou égale à 1 à notre fonction au départ, on retourne la valeur 1 et la fonction s'arrête.

Si on passe une valeur strictement supérieure à 1, on retourne cette valeur et on appelle `factorielle(n-1)`. Si `n-1` représente toujours une valeur strictement supérieure à 1, on retourne cette valeur et on appelle à nouveau notre fonction avec une valeur diminuée de 1 et etc. Jusqu'à ce que la valeur passée à `factorielle()` atteigne 1.

Un peu de vocabulaire : fonction vs procédure en Python

Par définition, toute fonction est censée renvoyer une valeur. Une fonction qui ne renvoie pas de valeur n'est pas une fonction : on appelle cela en programmation une procédure.

En Python, en fait, même les fonctions sans instruction `return` explicite renvoient une valeur qui est `None`. Le valeur `None` est une valeur qui correspond justement à l'absence de valeur. Cette valeur sert à indiquer "il n'y a pas de valeur".

L'interpréteur Python l'ignore lorsque c'est la seule valeur qui est renvoyée mais elle existe tout de même et c'est la raison pour laquelle on appelle les fonctions qui ne possèdent pas de `return` explicite des fonctions en Python.

```
[>>> def inutile():  
[...     print("Pas très utile comme fonction")  
[...  
[>>> inutile()  
Pas très utile comme fonction  
[>>> print(inutile())  
Pas très utile comme fonction  
None
```

Nous allons découvrir maintenant un concept fondamental lié aux fonctions et aux variables qui est celui de portée des variables. Nous allons comprendre les conditions d'accès et d'utilisation des différentes variables dans un script.

Définition de la portée des variables en Python

En Python, nous pouvons déclarer des variables n'importe où dans notre script : au début du script, à l'intérieur de boucles, au sein de nos fonctions, etc.

L'endroit où on définit une variable dans le script va déterminer l'endroit où la variable va être accessible c'est-à-dire utilisable.

Le terme de "portée des variables" sert à désigner les différents espaces dans le script dans lesquels une variable est accessible c'est-à-dire utilisable. En Python, une variable peut avoir une portée locale ou une portée globale.

Variables globales et variables locales en Python

Les variables définies dans une fonction sont appelées variables locales. Elles ne peuvent être utilisées que localement c'est-à-dire qu'à l'intérieur de la fonction qui les a définies. Tenter d'appeler une variable locale depuis l'extérieur de la fonction qui l'a définie provoquera une erreur.

```
[>>> #On définit une variable locale
[... def portee():
[...     local = 10
[...     print(local)
[...
[>>> portee()
10
[>>> #On tente d'appeler notre variable "local" depuis le contexte global :
[... print(local)
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: name 'local' is not defined
```

Cela est dû au fait que chaque fois qu'une fonction est appelée, Python réserve pour elle (dans la mémoire de l'ordinateur) un nouvel espace de noms (c'est-à-dire une sorte de dossier virtuel). Les contenus des variables locales sont stockés dans cet espace de noms qui est inaccessible depuis l'extérieur de la fonction.

Cet espace de noms est automatiquement détruit dès que la fonction a terminé son travail, ce qui fait que les valeurs des variables sont réinitialisées à chaque nouvel appel de fonction.

Les variables définies dans l'espace global du script, c'est-à-dire en dehors de toute fonction sont appelées des variables globales. Ces variables sont accessibles (= utilisables) à travers l'ensemble du script et accessible en lecture seulement à l'intérieur des fonctions utilisées dans ce script.

Pour le dire très simplement : une fonction va pouvoir utiliser la valeur d'une variable définie globalement mais ne va pas pouvoir modifier sa valeur c'est-à-dire la redéfinir. En effet, toute variable définie dans une fonction est par définition locale ce qui fait que si on essaie de redéfinir une variable globale à l'intérieur d'une fonction on ne fera que créer une autre variable de même nom que la variable globale qu'on souhaite redéfinir mais qui sera locale et bien distincte de cette dernière.

```
>>> #Variable x globale
... x = 10
>>> def portee():
...     print(x)
...
>>> portee()
10
>>>
>>> def portee2():
...     x = 20 #Variable x locale
...     print(x)
...
>>> portee2()
20
```

Modifier une variable globale depuis une fonction

Dans certaines situations, il serait utile de pouvoir modifier la valeur d'une variable globale depuis une fonction, notamment dans le cas où une fonction se sert d'une variable globale et la manipule.

Cela est possible en Python. Pour faire cela, il suffit d'utiliser le mot clef `global` devant le nom d'une variable globale utilisée localement afin d'indiquer à Python qu'on souhaite bien modifier le contenu de la variable globale et non pas créer une variable locale de même nom.

```
>>> #Utilisation du mot clef global
... x = 10
>>> def portee():
...     global x
...     x = 5
...
>>> portee()
>>> print(x)
5
```