


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

Dans cette nouvelle partie, nous allons étudier une autre notion incontournable de tout langage de programmation qui se respecte : les fonctions. Nous allons notamment définir ce qu'est une fonction et comprendre l'intérêt d'utiliser ces structures puis nous verrons comment créer nos propres fonctions en Python ainsi que certains concepts avancés relatifs aux fonctions.

Qu'est-ce qu'une fonction ?

Une fonction est un bloc de code nommé. Une fonction correspond à un ensemble d'instructions créées pour effectuer une tâche précise, regroupées ensemble et qu'on va pouvoir exécuter autant de fois qu'on le souhaite en "l'appelant" avec son nom. Notez "qu'appeler" une fonction signifie exécuter les instructions qu'elle contient.

L'intérêt principal des fonctions se situe dans le fait qu'on va pouvoir appeler une fonction et donc exécuter les instructions qu'elle contient autant de fois qu'on le souhaite, ce qui constitue au final un gain de temps conséquent pour le développement d'un programme et ce qui nous permet de créer un code beaucoup plus clair.

Il existe deux grands "types" de fonctions en Python : les fonctions prédéfinies et les fonctions créées par l'utilisateur.

Les fonctions prédéfinies Python

Les fonctions prédéfinies sont des fonctions déjà créées et mises à notre disposition par Python. Dans ce cours, nous avons déjà utilisé des fonctions prédéfinies comme la fonction `print()` ou la fonction `type()` par exemple.

Ici, je tiens à rappeler qu'en programmation rien n'est magique : la "programmation"... ne représente que des séries d'instruction programmées.

Lorsqu'on a utilisé `print()` pour afficher des données pour la première fois ou `type()` pour connaître le type d'une donnée, on ne s'est pas posé la question de ce qu'il se passait en arrière plan.

En fait, ces deux fonctions sont des fonctions complexes et qui contiennent de nombreuses lignes d'instructions leur permettant d'accomplir une tâche précise : l'affichage d'un résultat ou la détermination du type d'une valeur en l'occurrence.

Cette complexité nous est cachée : nous n'avons qu'à appeler nos fonctions pour qu'elles fassent leur travail et n'avons pas à écrire la série d'instructions qu'elles contiennent à chaque fois et c'est tout l'intérêt des fonctions.

Python, dans sa version 3.7.4, met à notre disposition quasiment 70 fonctions prédéfinies (sans compter les fonctions des modules ou extensions dont nous parlerons plus tard). Vous pouvez déjà trouver la liste ci-dessous. Nous serons amenés à utiliser la plupart d'entre elles dans ce cours ; nous les définirons à ce moment là. Considérez le tableau ci-dessous comme une simple référence.

Liste des fonction prédéfinies Python 3.7.4

abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__
complex()	hasattr()	max()	round()	

Les fonction Python définies par l'utilisateur

En plus des fonction prédéfinies, Python nous laisse la possibilité de définir nos propres fonctions. Ces fonctions ne seront bien évidemment disponibles et utilisables que dans l'espace où elles ont été définies, c'est-à-dire uniquement au sein de nos scripts et non pas pour l'ensemble des développeurs utilisant Python.

On va vouloir créer nos propres fonctions Python lorsque nos programmes utilisent de manière répétées une même série d'instructions : plutôt que de réécrire ces instructions à chaque fois, autant utiliser une fonction !

Les fonctions vont aussi être de très bons outils d'abstraction lorsqu'on voudra distribuer notre code : on préférera souvent fournir des fonctions à utiliser aux autres développeurs plutôt que de les laisser se débrouiller avec des séries d'instructions "sauvages".

Pour définir une nouvelle fonction en Python, nous allons utiliser le mot clef `def` qui sert à introduire une définition de fonction. Ce mot clef doit être suivi du nom de la fonction, d'une paire de parenthèses au sein desquelles on pourra fournir une liste de paramètres (nous reviendrons là dessus plus tard) et de `:` pour terminer la ligne comme ceci

```
def ma_fonction():.
```

Le nom d'une fonction Python doit respecter les normes usuelles concernant les noms : un nom de fonction doit commencer par une lettre ou un underscore et ne contenir que des caractères alphanumériques classiques (pas d'accent ni de cédille ni aucun caractère spécial).

Notez que les noms de fonctions sont sensibles à la casse en Python, ce qui signifie que les fonctions `ma_fonction()`, `Ma_fonction()`, `ma_FONction()` et `MA_FONCTION()` par exemple seront des fonctions bien différentes pour Python.

Nous allons ensuite placer la liste des différentes instructions de notre fonction à la ligne suivant sa définition et en les indentant par rapport à la définition afin que Python comprenne que ces instructions appartiennent à notre fonction. Notez que la première instruction d'une fonction peut être une chaîne de caractères littérale qui sera alors utilisée comme chaîne de documentation de la fonction.

Créons immédiatement deux fonctions `bonjour()` et `BONJOUR()` toutes simples dont le but va être d'afficher un message en exécutant elles-mêmes une fonction `print()`. On va faire cela comme cela :

```
[>>> def bonjour():
[...     print("Bonjour à tous")
[...
[>>> def BONJOUR():
[...     print("BONJOUR A TOUS")
[...
```

Nous avons ici défini nos deux premières fonctions. Ces fonctions ne sont pas très utiles ici : elles se contentent simplement d'exécuter une fonction `print()` mais c'est déjà un bon début !

Maintenant que nos fonctions sont créées, nous allons devoir les appeler pour exécuter le code qu'elles contiennent. Pour cela, nous allons utiliser leur nom suivi d'un couple de parenthèses. On va pouvoir appeler nos fonctions autant de fois qu'on le souhaite dans notre script : c'est tout l'intérêt des fonctions !

```
>>> def bonjour():  
[...     print("Bonjour à tous")  
[...  
>>> def BONJOUR():  
[...     print("BONJOUR A TOUS")  
[...  
>>> bonjour()  
Bonjour à tous  
>>> BONJOUR()  
BONJOUR A TOUS  
>>> bonjour()  
Bonjour à tous  
>>> bonjour()  
Bonjour à tous
```

Les paramètres et arguments des fonctions

Les fonctions que nous avons créées ci-dessus se contentent d'exécuter toujours la même fonction `print()` et donc de renvoyer toujours le même message.

Elles ne sont pas très utiles en l'état. Un autre aspect fondamental des fonctions est qu'elles vont pouvoir accepter des informations qui viennent de l'extérieur, c'est-à-dire qui sont externes à leur définition et qui vont les amener à produire des résultats différents. Souvent même, les fonctions vont avoir besoin qu'on leur passe des informations externes pour fonctionner normalement.

C'est par exemple le cas des fonctions `print()` et `type()` : ces deux fonctions permettent d'afficher un message et de déterminer le type d'une donnée. Pour afficher un message, `print()` va avoir besoin qu'on lui passe les données qu'elle doit afficher. De même, `type()` va avoir besoin qu'on lui fournisse la donnée dont elle doit déterminer le type.

Ces informations dont vont avoir besoin certaines fonctions pour fonctionner et qu'on va passer à nos fonctions entre le couple de parenthèses sont appelées des arguments ou des paramètres.

Pour rester très simple et très schématique ici, on parle de "paramètres" lorsqu'on définit une fonction, c'est-à-dire lorsqu'on indique dans la définition de la fonction que telle fonction a besoin d'une, de deux... informations pour fonctionner et on parle "d'arguments" pour désigner les valeurs effectivement passées à une fonction lorsqu'on l'utilise.

Illustrons cela immédiatement en reprenant et en modifiant notre fonction `bonjour()` afin qu'elle affiche "Bonjour " suivi du nom de quelqu'un qui va lui être fourni ultérieurement. Pour réaliser cela, on va indiquer dans la définition de la fonction que celle-ci a besoin d'un paramètre pour fonctionner.

On peut donner n'importe quel nom à ce paramètre dans la définition puisque celui-ci sera dans tous les cas remplacé par la valeur effective passée lors de l'appel à la fonction. Pour une meilleure lisibilité, il est cependant conseillé de fournir des noms descriptifs et paramètres des fonctions. On peut par exemple l'appeler `prenom` dans notre cas :

```
[>>> def bonjour(prenom):
[...     print("Bonjour " + prenom)
[...
[>>> bonjour("Pierre")
Bonjour Pierre
[>>> bonjour("Mathilde")
Bonjour Mathilde
[>>> bonjour("Florian")
Bonjour Florian
```

Expliquons ce code. On crée une nouvelle fonction `bonjour()` dont le rôle est d’afficher “Bonjour” suivi du prénom de quelqu’un. Pour que cela fonctionne, il va falloir lui passer un prénom lorsqu’on appelle notre fonction en argument de celle-ci.

Dans la définition de la fonction, on va donc indiquer que notre fonction a besoin d’un paramètre pour fonctionner. Ici, on choisit le mot “prenom” pour définir notre paramètre. On aurait aussi bien pu choisir “toto”. Ensuite, dans le corps de notre fonction, on utilise une fonction `print()` qui va afficher “Bonjour” suivi du prénom qu’on aura indiqué lorsqu’on utilisera la fonction.

Ici, je vous rappelle que le mot qu’on utilise comme paramètre lors de la définition de la fonction sera remplacé par la valeur passée en argument lors de l’appel à la fonction. On va donc utiliser notre paramètre dans `print()` afin que cette fonction affiche bien “Bonjour” suivi d’un prénom.

On utilise ensuite notre fonction plusieurs fois, en lui passant un prénom différent à chaque fois en argument. La valeur passée va se substituer au paramètre défini lors de la définition de la fonction.

Le code de notre fonction n’est cependant pas très optimisé ici : en effet, on utilise de la concaténation dans `print()` or la concaténation ne va fonctionner que si la valeur passée est bien une chaîne de caractères. Si on passe un chiffre en argument de `bonjour()`, Python renverra une erreur.

Ici, il va être plus efficace de passer le texte et l’argument comme deux arguments différents de `print()`. En effet, vous devez savoir que `print()` est capable d’accepter un nombre infini d’arguments qu’elle affichera à la suite. Cela résout notre problème de type de valeurs :

```
[>>> def bonjour(p):
[...     print("Bonjour ", p)
[...
[>>> bonjour("Pierre")
Bonjour Pierre
[>>> bonjour(45)
Bonjour 45
```

Créer des fonctions acceptant un nombre variable d'arguments

Dans le chapitre précédent, nous avons défini ce qu'étaient des paramètres et des arguments. Nous avons créé une fonction `bonjour()` qui avait besoin qu'on lui passe un argument pour fonctionner comme ceci :

```
>>> def bonjour(p):  
[...     print("Bonjour ", p)  
[...  
[>>> bonjour("Pierre")  
Bonjour Pierre
```

Cette définition impose qu'on passe un et un seul argument à notre fonction pour qu'elle fonctionne : si on tente de l'appeler sans argument ou en lui passant plusieurs arguments Python renverra une erreur.

Dans certaines situations, nous voudrions créer des fonctions plus flexibles qui pourront accepter un nombre variable d'arguments. Cela peut être utile si on souhaite créer une fonction de calcul de somme par exemple qui devra additionner les différents arguments passés sans limite sur le nombre d'arguments et sans qu'on sache à priori combien de valeurs vont être additionnées.

En Python, il existe deux façons différentes de créer des fonctions qui acceptent un nombre variable d'arguments. On peut :

- Définir des valeurs de paramètres par défaut lors de la définition d'une fonction ;
- Utiliser une syntaxe particulière permettant de passer un nombre arbitraire d'arguments.

Préciser des valeurs par défaut pour les paramètres d'une fonction

On va déjà pouvoir préciser des valeurs par défaut pour nos paramètres. Comme leur nom l'indique, ces valeurs seront utilisées par défaut lors d'un appel à la fonction si aucune valeur effective (si aucun argument) n'est passée à la place.

Utiliser des valeurs par défaut pour les paramètres de fonctions permet donc aux utilisateurs d'appeler cette fonction en passant en omettant de passer les arguments relatifs aux paramètres possédant des valeurs par défaut.

On va pouvoir définir des fonctions avec des paramètres sans valeur et des paramètres avec des valeurs par défaut. Attention cependant : vous devez bien comprendre qu'ici, si on omet de passer des valeurs lors de l'appel à la fonction, Python n'a aucun moyen de savoir quel argument est manquant. Si 1, 2, etc. arguments sont passés, ils correspondront de facto au premier, aux premier et deuxième, etc. paramètres de la définition de fonction.

Pour cette raison, on placera toujours les paramètres sans valeur par défaut au début et ceux avec valeurs par défaut à la fin afin que les arguments passés remplacent en priorité les paramètres sans valeur.

```
[>>> def prez(prenom, age = 18, nat = "Francais"):
[...     print("Je m'appelle ", prenom, ", j'ai au moins ", age, " ans et je suis probablement ",
[...         nat)
[...
[>>> prez("Pierre")
Je m'appelle Pierre , j'ai au moins 18 ans et je suis probablement Francais
[>>>
[>>> prez("Pierre", 29)
Je m'appelle Pierre , j'ai au moins 29 ans et je suis probablement Francais
[>>>
[>>> prez("Pierre", 29, "Belge")
Je m'appelle Pierre , j'ai au moins 29 ans et je suis probablement Belge
```

Si on souhaite s'assurer que les valeurs passées à une fonction vont bien correspondre à tel ou tel paramètre, on peut passer à nos fonctions des arguments nommés. Un argument nommé est un argument qui contient le nom d'un paramètre présent dans la définition de la fonction suivi de la valeur qu'on souhaite passer comme ceci : `argument = valeur`.

On va pouvoir passer les arguments nommés dans n'importe quel ordre puisque Python pourra faire le lien grâce au nom avec les arguments attendus par notre fonction. Notez cependant qu'il faudra ici passer les arguments nommés en dernier, après les arguments sans nom. Par ailleurs, aucun argument ne peut recevoir de valeur plus d'une fois. Faites donc bien attention à ne pas passer une valeur à un argument sans le nommer puis à repasser cette valeur en le nommant par inadvertance.

```
[>>> def prez(prenom, age = 18, nat = "Francais"):
[...     print("Je m'appelle ", prenom, ", j'ai au moins ", age, " ans et je suis probablement ",
[...         nat)
[...
[>>> prez("Pierre")
Je m'appelle Pierre , j'ai au moins 18 ans et je suis probablement Francais
[>>>
[>>> prez("Pierre", 29)
Je m'appelle Pierre , j'ai au moins 29 ans et je suis probablement Francais
[>>>
[>>> prez("Pierre", 29, "Belge")
Je m'appelle Pierre , j'ai au moins 29 ans et je suis probablement Belge
[>>>
[>>> prez("Mathilde", nat="Suisse", age=27)
Je m'appelle Mathilde , j'ai au moins 27 ans et je suis probablement Suisse
```

Passer un nombre arbitraire d'arguments avec `*args` et `**kwargs`

La syntaxe `*args` (remplacez "args" par ce que vous voulez) permet d'indiquer lors de la définition d'une fonction que notre fonction peut accepter un nombre variable d'arguments. Ces arguments sont intégrés dans un tuple. On va pouvoir préciser 0, 1 ou plusieurs paramètres classiques dans la définition de la fonction avant la partie variable.

```
[>>> def somme(*args):
[...     s = 0
[...     for n in args:
[...         s += n
[...     print("Somme : ", s)
[...
[>>> somme(1, 2)
Somme : 3
[>>> somme(1, 2, 3, 4)
Somme : 10
[>>> somme(30, 100, 2)
Somme : 132
```

Ici, on utilise une boucle `for` pour itérer parmi les arguments : tant que des valeurs sont trouvées, elles sont ajoutées à la valeur de `s`. Dès qu'on arrive à court d'arguments, on `print()` le résultat.

De façon alternative, la syntaxe `**kwargs` (remplacez “kwargs” par ce que vous voulez) permet également d’indiquer que notre fonction peut recevoir un nombre variable d’arguments mais cette fois-ci les arguments devront être passés sous la forme d’un dictionnaire Python.

```
[>>> def prez(**kwargs):
...     for i, j in kwargs.items():
...         print(i, j)
...
>>> prez(prenom="Pierre", age=29, sport="trail")
prenom Pierre
age 29
sport trail
```

Dans cet exemple, j’utilise la méthode Python `items()` dont le rôle est de récupérer les différentes paires clefs : valeurs d’un dictionnaire. Nous reparlerons des méthodes lorsque nous aborderons l’orienté objet. Pour le moment, vous pouvez considérer qu’une méthode est l’équivalent d’une fonction.

Séparer des données pour les passer à une fonction

Les syntaxes `*args` et `**kwargs` peuvent être utilisées pour réaliser les opérations inverse de celles présentées ci-dessus, à savoir séparer des données composites pour passer les valeurs ou éléments de ces données un à un en arguments des fonctions.

On utilisera la syntaxe `*args` pour séparer les arguments présents dans une liste ou un tuple et la syntaxe `**kwargs` pour séparer les arguments présents dans un dictionnaire et fournir des arguments nommés à une fonction.

```
[>>> def somme(a, b, c):
...     s = a + b + c
...     print(s)
...
>>> x = [1, 2, 3]
>>> somme(*x)
6
```