

Chapitre 5 : Fonctions et sous programmes

Yves Guidet pour Édugroupe

V1.4.8 May 2, 2017

Tous les langages de programmation (sauf peut-être COBOL) connaissent la notion de fonction.

Cette notion est plus ou moins proche de celle de fonction mathématique, selon les langages.

En C et dans ses descendants (C++, Java, Perl, PHP, ...) une fonction peut modifier ses arguments et ne pas retourner de résultat.

On va voir que c'est le mot-clé *def* qui déclare une fonction.
Pour ce qui est des arguments, il y a deux écoles :

- ▶ les langages de scripts (shells Unix, Perl, .BAT du DOS) se contentent de récupérer les arguments par leur « numéro », ou, ce qui revient au même dans une liste
- ▶ les langages évolués déclarent un certain nombre de paramètres « formels », avec leur type.

En Python, il y a bien des paramètres typés, et ce langage se démarque ainsi des autres langages de script, mais sans déclaration de type. Ceci introduit une forme de polymorphisme, on y reviendra.

```
>>> def carre(x):  
...     return x**2  
...  
>>> carre(2)  
4  
>>>
```

- ▶ indentation obligatoire.
- ▶ return pas obligatoire
 - ▶ retourne *None* dans ce cas

Valeurs par défaut (arguments optionnels)

Considérons la fonction :

```
def coucou(n=5, lang='Python'):  
    for i in range(n):  
        print "j'adore programmer en ", lang
```

On imagine très bien ce que feront les appels suivants :

```
coucou()  
print  
coucou(3)  
print  
coucou(2, 'Perl') # bad example ?
```

Fonctions : arguments

► arguments non typés

```
>>> def multiplier(x,y):  
...     return x*y  
...  
>>> multiplier(2,3)  
6  
>>> multiplier(3.1415,5.23)  
16.430045000000003  
>>> multiplier('to',2)  
'toto'  
>>>
```

Eh oui multiplier une chaîne par un entier a un sens.

Fonctions : renvoient quoi ?

- capables de renvoyer plusieurs valeurs à la fois

```
>>> def carre_et_cube(x):  
...     return x**2,x**3  
...  
>>> carre_et_cube(2)  
(4, 8)  
>>>
```

- Python renvoie un objet séquentiel

Remarque importante

Python se comporte comme un langage interprété ; redéfinir une fonction ne provoque aucune erreur mais écrase l'ancienne définition :

```
>>> def f(n):  
...     return n + 1  
...  
>>> f(5)  
6  
>>> def f(n):  
...     return n + 2  
...  
>>> f(5)  
7
```

Ce qui rend impossible toute forme de *surcharge*.

Ce sont les appels du genre « `*args` » et « `**kwargs` ».

- ▶ `*args` permet de passer les arguments via un *tuple* à la fonction, tandis que
- ▶ `**kwargs` permet de les passer un *dictionnaire*.

On obtient ainsi un nombre variable d'arguments, comme le montre cet exemple :

```
def toto(*argu):  
    print "toto : j'ai reçu ", len(argu), " arguments"
```

```
toto(29, 11, 1953)  
toto('Michaël', "XXX")
```

Les 2 appels afficheront dont 3 et 2.

Remarque sur **args*

On peut trouver cette notation **args* dans un **appel** de fonction, comme ici :

```
yves@bella:cartesHebdo$ python
>>> def f(*args):
...     print args
...
>>> f()
()
>>> f(1, 2)
(1, 2)
>>> f(*'abc')
('a', 'b', 'c')
```

La chaîne **abc** étant un *énumérable*, c'est la **liste** de ses caractères que reçoit la fonction *f*.

****kwargs** (I/II)

Cette fois, les arguments seront passés dans un *dictionnaire*.
Exemple :

```
def toto(**kwarg):  
    print "toto : j'ai reçu ", len(kwarg), " argument(s) ", kwarg  
    for x in kwarg:  
        print x, kwarg[x]
```

Rappelons que la boucle *for* énumérera les *clés* du *dictionnaire*.

****kwargs** (II/II)

Les appels :

```
toto(prenom = 'Yves')  
toto(jour = 29, mois = 11)  
toto(mois = 11, jour = 29)
```

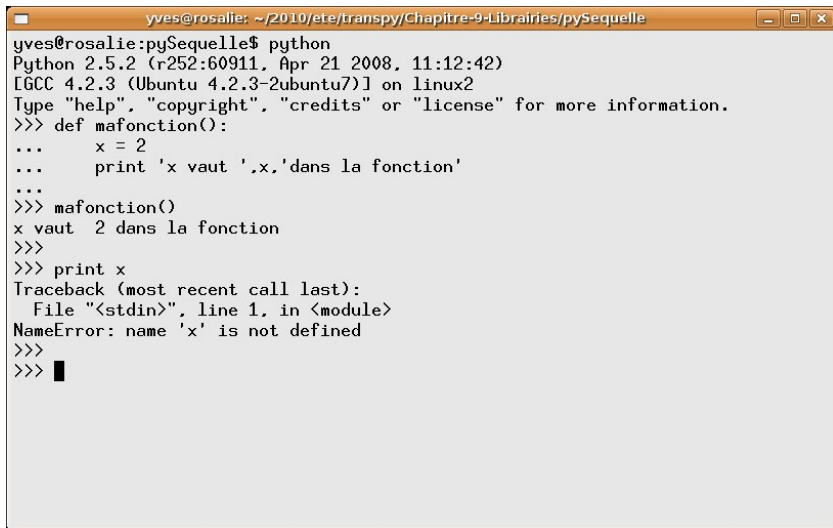
donneront donc à l'exécution :

```
toto : j'ai reçu 1 argument(s) {'prenom': 'Yves'}  
prenom Yves  
toto : j'ai reçu 2 argument(s) {'mois': 11, 'jour': 29}  
mois 11  
jour 29  
toto : j'ai reçu 2 argument(s) {'jour': 29, 'mois': 11}  
jour 29  
mois 11
```

On a déjà évoqué le problème de la surcharge, on trouvera une solution [ici <https://huit.re/4hdPjsjR>].

Variables locales

- ▶ variable globale : à la racine du module, visible dans tout le module



```
yves@rosalie: ~/2010/ete/transpy/Chapitre-9-Librairies/pySequelle
yves@rosalie:pySequelle$ python
Python 2.5.2 (r252:60911, Apr 21 2008, 11:12:42)
[GCC 4.2.3 (Ubuntu 4.2.3-2ubuntu7)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> def mafonction():
...     x = 2
...     print 'x vaut ',x,'dans la fonction'
...
>>> mafonction()
x vaut 2 dans la fonction
>>>
>>> print x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
>>>
>>> █
```

Variables locales (suite)

- ▶ attention aux types modifiables

```
>>> liste = [29,11,54]
>>> def mafonction():
...     liste[2] = 53
...
>>> mafonction()
>>> liste
[29, 11, 53]
>>>
```


Variables locales (suite et fin)

- ▶ liste en argument : modifiable au sein de la fonction

```
>>> def mafonction(x):  
...     x[1] = -15  
...  
>>> y = [1,2,3]  
>>> mafonction(y)  
>>> y  
[1, -15, 3]  
>>>
```

Il est possible en Python de définir une fonction à l'intérieur d'une autre fonction ; elle n'est visible que de la fonction qui l'englobe.

Opérateurs fonctionnels (map, lamda)

Ces deux opérateurs proviennent des langages fonctionnels (Lisp, Scheme).

map applique une fonction à tous les éléments d'une liste.

lambda vient en fait du "lambda-calcul", c'est en fait une façon de définir une fonction anonyme.

Bien sûr, *map* et *lambda* se combinent agréablement :

```
>>> map(lambda x: x + 1, [29, 11])  
[30, 12]
```

Attention *map* a été modifiée en Python3k : elle ne retourne plus une liste, mais un itérateur. Voir

<http://diveintopython3.org/porting-code-to-python-3-with-2to3.html#map>

(<http://tinyurl.com/3j8tt9y>).

Lire aussi ENI pour en savoir plus.

Exercices (fonctions)

- ▶ Reprendre les exercices précédents, y introduire des fonctions.
- ▶ une idée pour des arguments par défaut ?
- ▶ écrire une fonction calculant la moyenne des valeurs (numériques) passées en argument
- ▶ écrire une fonction calculant une puissance et dont l'appel se fera comme ceci : $r = \text{power}(a = 2, b = 4)$