



# Chapitre 5

## Les fonctions, modules et packages en Python

- ❑ Partie 1: Les fonctions
- ❑ Partie 2: Les Modules
- ❑ Partie 3: Les packages

# Partie 1: Les fonctions

- ❑ Les scripts que vous avez écrits jusqu'à présent étaient à chaque fois très courts, car leur objectif était seulement de vous faire assimiler les premiers éléments du langage.
- ❑ Lorsque vous commencerez à développer de véritables projets, vous serez confrontés à des problèmes souvent fort complexes, et les lignes de programme vont commencer à s'accumuler...
- ❑ Il arrivera souvent qu'une même séquence d'instructions doivent être utilisée à plusieurs reprises dans un programme, et on souhaitera bien évidemment ne pas avoir à la reproduire systématiquement.

**Solution** : décomposer un problème en plusieurs sous problèmes plus simples qui seront étudiés séparément (Ces sous-problèmes peuvent éventuellement être eux-mêmes décomposés à leur tour, et ainsi de suite)



Les **fonctions**, il s'agit de différentes des structures de sous-programmes qui ont été imaginées par les concepteurs des langages de haut niveau afin de résoudre les difficultés évoquées ci-dessus.

## Pourquoi créer des fonctions ?

1. Meilleure organisation du programme (regrouper les tâches par blocs : lisibilité → maintenance)
2. Éviter la redondance (pas de copier/coller → maintenance, meilleure réutilisation du code)
3. Possibilité de partager les fonctions (via des modules)
4. Le programme principal doit être le plus simple possible

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

```
def nomDeLaFonction(liste de paramètres):
```

```
...
```

```
    bloc d'instructions
```

```
...
```

- ❑ Vous pouvez choisir n'importe quel nom pour la fonction que vous créez, à l'exception des mots réservés du langage (ex: `input`, `print`, `random`,.... ).
- ❑ Ne pas utiliser de caractère spécial ou accentué (le caractère souligné « `_` » est permis). Comme c'est le cas pour les noms de variables,
- ❑ il vous est conseillé d'utiliser surtout des lettres minuscules, notamment au début du nom (les noms commençant par une majuscule seront réservés aux classes: concept non étudié dans ce cours).
- ❑ L'entête d'une fonction commence par : **def**
  - `def` est une *instruction composée*.
  - La ligne contenant cette instruction se termine obligatoirement par un double point, lequel introduit un bloc d'instructions que vous ne devez pas oublier d'indenter



- ❑ La **liste de paramètres** spécifie quelles informations, il faudra fournir en guise d'arguments lorsque l'on voudra utiliser cette fonction (Les parenthèses sont vides si la fonction ne nécessite pas d'arguments).
- ❑ Une fonction s'utilise pratiquement comme une instruction quelconque. Dans le corps d'un programme, un **appel de fonction** est constitué du nom de la fonction suivi de parenthèses.
  - Si c'est nécessaire, on place dans ces parenthèses le ou les arguments que l'on souhaite transmettre à la fonction.
  - il est possible de définir pour ces paramètres des valeurs par défaut (voir plus loin).

## Exemple:

```
def petit (a, b):  
    if (a < b):  
        d = a  
    else:  
        d = 0  
    return d
```

- **def** pour dire que l'on définit une fonction
- Le nom de la fonction est « **petit** »
- Les paramètres ne sont pas typés
- Noter le rôle du :
- Attention à l'indentation
- **return** renvoie la valeur
- **return** provoque immédiatement la sortie de la fonction



Procédure = Fonction sans **return**

# Les fonctions

Passage de paramètres par position

```
print(petit(10, 12))
```

Passer les paramètres selon les positions attendues  
La fonction renvoie **10**

Passage par nom. Le mode de passage que je préconise, d'autant plus que les paramètres ne sont pas typés.

```
print(petit(a=10,b=12))
```

Aucune confusion possible → **10**

```
print(petit(b=12, a=10))
```

Aucune confusion possible → **10**

En revanche...

```
print(petit(12, 10))
```

Sans instructions spécifiques, le passage par position prévaut  
La fonction renvoie → **0**

# Les fonctions

```
fonction_petit.py - D:\Travaux\univer...
File Edit Format Run Options Window Help

# -*- coding: utf-8 -*-

#écriture de la fonction
def petit(a,b):
    if (a < b):
        d = a
    else:
        d = 0
    return d

**** PROGRAMME PRINCIPAL ****

#saisie de x et y
x = int(input("x : "))
y = int(input("y : "))

#appel de la fonction
res = petit(a=x,b=y)

#affichage avec transtypage
print("résultat : " + str(res))

#pour bloquer la fermeture de la console
input("pause...")
```

Définition de la fonction

Programme principal

Appel de la fonction dans le programme principal (on peut faire l'appel d'une fonction dans une autre fonction)

2 exécutions du programme

```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help

>>> ----- RESTART -----
>>>
x : 15
y : 12
résultat : 0
pause...

>>> ----- RESTART -----
>>>
x : 5
y : 8
résultat : 5
pause...

Ln: 45 Col: 0
```

## Paramètres par défaut

- Affecter des valeurs aux paramètres **dès la définition de la fonction**
- Si l'utilisateur omet le paramètre lors de l'appel, cette valeur est utilisée
- Si l'utilisateur spécifie une valeur, c'est bien cette dernière qui est utilisée
- Les paramètres avec valeur par défaut doivent être regroupées en dernière position dans la liste des paramètres

## Exemple

```
def ecart(a,b,epsilon = 0.1):  
    d = math.fabs(a - b)  
    if (d < epsilon):  
        d = 0  
    return d  
  
ecart(a=12.2, b=11.9, epsilon = 1) #renvoie 0  
ecart(a=12.2, b=11.9) #renvoie 0.3
```

La valeur utilisée est epsilon = 0.1 dans ce cas



**Exemple:** Calcule le volume d'une sphère à l'aide de la formule :  $V = \frac{4}{3} * \pi * r^3$

```
def cube(n):  
    return n**3  
def volumeSphere(r):  
    return 4 * 3.1416 * cube(r) / 3  
r = input('Entrez la valeur du rayon : ')  
print 'Le volume de cette sphère vaut', volumeSphere(r)
```

- Ce programme comporte trois parties : les deux fonctions **cube()** et **volumeSphere()**, et ensuite le corps principal du programme.
- Dans le corps principal du programme, il y a un appel de la fonction **volumeSphere()**.
- A l'intérieur de la fonction **volumeSphere()**, il y a un appel de la fonction **cube()**.

## Passage de paramètres :

*#écriture de la fonction*

```
def modifier_non_mutable(a,b):
    a = 2 * a
    b = 3 * b
    print(a,b)
```

*#appel*

```
x = 10
y = 15
```

```
modifier_non_mutable(x,y)
print(x,y)
```

*#écriture de la fonction*

```
def modifier_mutable(a,b):
    a.append(8)
    b[0] = 6
    print(a,b)
```

*#appel pour les listes*

```
lx = [10]
ly = [15]
```

```
modifier_mutable(lx,ly)
print(lx,ly)
```

Les paramètres sont toujours passés par référence (référence à l'objet), mais ils sont modifiables selon qu'ils sont **mutables** (dictionnaire\*, liste\*, etc.) ou **non mutables** (types simples, tuples\*, etc.).

20 45

*\* à voir plus tard*

10 15

[10, 8] [6]

[10, 8] [6]

C'est ce qui est pointé par la référence qui est modifiable, pas la référence elle-même. Ex. `b = [6]` ne sera pas répercuté à l'extérieur de la fonction.

## Renvoyer plusieurs valeurs avec return

`return` peut envoyer plusieurs valeurs simultanément.  
La récupération passe par une affectation multiple.

#écriture de la fonction

```
def extreme(a,b):  
    if (a < b):  
        return a,b  
    else:  
        return b,a
```

#appel

```
x = 10  
y = 15  
vmin,vmax = extreme(x,y)  
print(vmin,vmax)
```

vmin =10 et vmax=15

Remarque : Que se passe-t-il si nous ne mettons qu'une variable dans la partie gauche de l'affectation ?

```
#ou autre appel  
v = extreme(x,y)  
print(v)  
#quel type pour v ?  
print(type(v))
```

v = (10, 15)

<class 'tuple'>

v est un « tuple », une collection de valeurs, à voir plus tard.



## Variables locales et globales

1. Les variables définies localement dans les fonctions sont uniquement visibles dans ces fonctions.
2. Les variables définies (dans la mémoire globale) en dehors de la fonction ne sont pas accessibles dans la fonction
3. Elles ne le sont uniquement que si on utilise un mot clé spécifique

```
#fonction
def modif_1(v):
```

```
    x = v
```

```
#appel
x = 10
modif_1(99)
print(x) → 10
```

x est une variable locale,  
pas de répercussion

```
#fonction
def modif_2(v):
```

```
    x = x + v
```

```
#appel
x = 10
modif_2(99)
print(x)
```

x n'est pas assignée ici,  
l'instruction provoque  
une **ERREUR**

```
#fonction
def modif_3(v):
```

```
    global x
    x = x + v
```

```
#appel
x = 10
modif_3(99)
print(x) → 109
```

On va utiliser la variable  
globale x. L'instruction  
suivante équivaut à  
x = 10 + 99

## Utilisation des listes et des dictionnaires

Nous pouvons aussi passer par une structure intermédiaire telle que la **liste** ou le **dictionnaire** d'objets. Les objets peuvent être de type différent, au final l'outil est très souple. (nous verrons plus en détail les listes et les dictionnaires plus loin)

```
#écriture de la fonction
def extreme_liste(a,b):
    if (a < b):
        return [a,b]
    else:
        return [b,a]
```

```
#appel
x = 10
y = 15
res = extreme_liste(x,y)
print(res[0])
```

```
#écriture de la fonction
def extreme_dico(a,b):
    if (a < b):
        return {'mini' : a,'maxi' : b}
    else:
        return {'mini' : b,'maxi' : a}
```

```
#appel
x = 10
y = 15
res = extreme_dico(x,y)
print(res['mini'])
```



Les deux fonctions renvoient deux objets différents

Notez l'accès à la valeur minimale selon le type de l'objet

## Imbrication des fonctions:

### Fonctions locales et globales

Il est possible de définir une fonction dans une autre fonction. Dans ce cas, elle est locale à la fonction, elle n'est pas visible à l'extérieur.

#écriture de la fonction

```
def externe(a):
```

```
    #fonction imbriquée
```

```
    def interne(b):
```

```
        return 2.0 * b
```

```
#on est dans externe ici
```

```
return 3.0 * interne(a)
```

#appel

```
x = 10
```

```
print(externe(x)) → renvoie 60
```

```
print(interne(x)) → provoque une erreur
```

La fonction interne() est imbriquée dans externe, elle n'est pas exploitable dans le prog. principal ou dans les autres fonctions.



## Fonctions récursives:

- ❑ Une fonction récursive est une fonction qui s'appelle elle-même. Les fonctions récursives permettent d'obtenir une efficacité
- ❑ redoutable dans la résolution de certains algorithmes comme le tri rapide 2 (en anglais quicksort).
- ❑ Oublions la recherche d'efficacité pour l'instant et concentrons-nous sur l'exemple de la fonction mathématique factorielle.
- ❑ Nous vous rappelons que la factorielle s'écrit avec un ! et se définit de la manière suivante :

$$3! = 3 * 2 * 1 = 6$$

$$4! = 4 * 3 * 2 * 1 = 24$$

$$n! = n * n-1 * n-2 * \dots * 2 * 1$$

## Fonctions récursives: fonction factorielle

```

1 def calc_factorielle(nb):
2     if nb == 1:
3         return 1
4     else:
5         return nb * calc_factorielle(nb - 1)
6
7 # prog principal
8 print(calc_factorielle(4))
    
```

- Ligne 8, on appelle la fonction `calc_factorielle()` en passant comme argument l'entier 4.
- Dans la fonction, la variable locale qui récupère cet argument est `nb`.
- Au sein de la fonction, celle-ci se rappelle elle-même (ligne 5), mais cette fois-ci en passant la valeur 3.
- Au prochain appel, ce sera avec la valeur 2, puis finalement 1. Dans ce dernier cas, le test `if nb == 1:` est vrai et l'instruction `return 1` sera exécutée.

```

Appel à fact(3)
3*fact(2) = ?
Appel à fact(2)
1*fact(1) = ?
Appel à fact(1)
1*fact(0) = ?
Appel à fact(0)
Retour de la valeur 1
1*1
Retour de la valeur 1
2*1
Retour de la valeur 2
3*2
Retour de la valeur 6.
    
```

## Partie 2: Les modules

❑ **Définition:** fichier indépendant permettant de scinder un programme en plusieurs scripts. Ce mécanisme permet d'élaborer efficacement des bibliothèques de fonctions ou de classes.

❑ **Avantages :**

- réutilisation du code ;
- la documentation et les tests peuvent être intégrés au module ;
- réalisation de services ou de données partagés ;
- partition de l'espace de noms du système.

## Import d'un module

❑ Deux syntaxes possibles :

- la commande `import <nom_module>` importe la totalité des objets du module :

```
import tkinter
```

- la commande `from <nom_module> import obj1, obj2...` n'importe que les objets `obj1, obj2...` du module :

```
from math import pi, sin, log
```

❑ Il est conseillé d'importer dans l'ordre :

1. les modules de la bibliothèque standard ;
2. les modules des bibliothèques tierces ;
3. Les modules personnels



## ❑ Exemple: Notion d'« auto-test »

Un module `cube_m.py`. Remarquez l'utilisation de « l'auto-test » qui permet de tester le module seul :

```
def cube(y):  
    """Calcule le cube du paramètre <y>."""  
    return y**3  
  
# Auto-test -----  
if __name__ == "__main__": # False lors d'un import ==> ignoré  
    help(cube)             # affiche le docstring de la fonction  
    print("cube de 9 :", cube(9)) # cube de 9 : 729
```

• Utilisation de ce module. On importe la fonction `cube()` incluse dans le fichier `cube_m.py`.

```
from cube_m import cube  
for i in range(1, 4):  
    print("cube de", i, "=", cube(i), end=" ")  
# cube de 1 = 1 cube de 2 = 8 cube de 3 = 27
```

## 1. Bibliothèque standard:

❑ Il existe une série de modules que vous serez probablement amenés à utiliser si vous programmez en Python. Exemple:

- **math** : fonctions et constantes mathématiques de base (sin, cos, exp, pi...).
- **sys** : interaction avec l'interpréteur Python, passage d'arguments (cf. plus bas).
- **os** : dialogue avec le système d'exploitation (cf. plus bas).
- **random** : génération de nombres aléatoires.
- **time** : accès à l'heure de l'ordinateur et aux fonctions gérant le temps.
- **urllib** : récupération de données sur internet depuis Python.

▪ **Tkin**  
grand

Voir la liste  
complète sur



<https://docs.python.org/3/library/> objets

## Exemple: d'utilisation des modules standards:

```
# -*- coding: utf -*-  
  
#importer les modules  
#math et random  
import math, random  
  
#générer un nom réel  
#compris entre 0 et 1  
random.seed(None)  
value = random.random()  
  
#calculer le carré de  
#son logarithme  
logv = math.log(value)  
abslog = math.pow(logv,2.0)  
  
#affichage  
print(abslog)
```

Si plusieurs modules à importer, on les met à la suite en les séparant par « , »

Préfixer la fonction  
à utiliser par le  
nom du module

## Autre utilisation possible:

```
#définition d'alias
import math as m, random as r

#utilisation de l'alias
r.seed(None)
value = r.random()
logv = m.log(value)
abslog = m.pow(logv, 2.0)
```

```
#importer le contenu
#des modules
from math import log, pow
from random import seed, random

#utilisation directe
seed(None)
value = random()
logv = log(value)
abslog = pow(logv, 2.0)
```

L'alias permet d'utiliser des noms plus courts dans le programme.

Cette écriture permet de désigner nommément les fonctions à importer. Elle nous épargne le préfixe lors de l'appel des fonctions. Mais est-ce vraiment une bonne idée ?

N.B.: Avec « \* », nous les importons toutes (ex. `from math import *`). Là non plus pas besoin de préfixe par la suite.



## 2. Bibliothèques tierces :

- ❑ Outre les modules intégrés à la distribution standard de Python, on trouve des bibliothèques dans tous les domaines :
  - scientifique ;
  - bases de données ;
  - tests fonctionnels et contrôle de qualité ;
  - 3D ;
  - ...
- ❑ Le site [pypi.python.org/pypi](https://pypi.python.org/pypi) (*The Python Package Index*) recense des milliers de modules et de packages !

## 2. Bibliothèques tierces :

❑ Cette bibliothèque permet de calculer en tenant compte des unités du système SI (Système International d'unités).

▪ Voici un exemple

```
>>> from unum.units import *
>>> distance = 100*m
>>> temps = 9.683*s
>>> vitesse = distance / temps
>>> vitesse
10.327377878756584 [m/s]
>>> vitesse.asUnit(mile/h)
23.1017437978 [mile/h]
>>> acceleration = vitesse/temps
>>> acceleration
1.0665473385063085 [m/s2]
```

## 3. Bibliothèques (module) personnalisé

### Création d'un module personnalisé

Il suffit de créer un fichier `nom_module.py`,  
et d'y implémenter les fonctions à partager.

```
tva.py - D:\Travaux\university\...
File Edit Format Run Options Window Help
#taxe à 10%
def ptto_reduit(p):
    return p * 1.1

#taxe à 20%
def ptto_normal(p):
    return p * 1.2

#taxe à 5.5%
def ptto_alimentaire(p):
    return p * 1.055
Ln: 1 Col: 0
```

```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
>>> ===== RESTART =====
>>>
>>> import tva
>>> dir(tva)
['_builtins_', '__cached__', '__doc__', '__file__', '__loader__', '__name__',
 '__package__', '__spec__', 'ptto_alimentaire', 'ptto_normal', 'ptto_reduit']
>>> help(tva)
Help on module tva:

NAME
    tva - #taxe à 10%

FUNCTIONS
    ptto_alimentaire(p)
        #taxe à 5.5%

    ptto_normal(p)
        #taxe à 20%

    ptto_reduit(p)
        #taxe à 10%

FILE
    d:\travaux\university\cours_universite\supports_de_cours\informatique\pyc
hon\slides\exemples\b\tva.py
Ln: 28 Col: 4
```

`dir()` et `help()` permettent d'obtenir des  
informations sur le module (liste du contenu)

## Importation d'un module personnalisé:

```
appel_tva.py - D:\Travaux\univer
File Edit Format Run Options Wind
# -*- coding: utf -*-
#importation du module
import tva
*** PROGRAMME PRINCIPAL ***
#saisie prix ht
pht = int(input("prix : "))
#affichage prix ttc
pttc = tva.pttc_normal(pht)
print(pttc)
```

Python cherche automatiquement le module dans le « search path » c.-à-d.

- le dossier courant
- les dossiers listés dans la variable d'environnement **PYTHONPATH** (configurable sous Windows)
- les dossiers automatiquement spécifiés à l'installation. On peut obtenir la liste avec la commande **sys.path** (il faut importer le module **sys** au préalable).

Pour connaître et modifier le répertoire courant, utiliser les fonctions du module **os** c.-à-d.

```
import os
```

```
os.getcwd() # affiche le répertoire de travail actuel
```

```
os.chdir(chemin) # permet de le modifier
```

```
Ex. os.chdir("c:/temp/exo") #noter l'écriture du chemin
```



## Documentation:

tva.py - D:\Travaux\university\Cours\_Universite\Supports\_de\_cour...

File Edit Format Run Options Window Help

```
"""Module pour calcul des prix TTC
Application de différents niveaux de TVA
"""
```

```
#taxe à 10%
```

```
def ptto_reduit(p):
    """tva intermédiaire - ex. travaux aménagement
    """
    return p * 1.1
```

```
#taxe à 20%
```

```
def ptto_normal(p):
    """tva normale
    """
    return p * 1.2
```

```
#taxe à 5.5%
```

```
def ptto_alimentaire(p):
    """tva produits alimentaires - mais au
    """
    return p * 1.055
```

Python 3.4.3 Shell

File Edit Shell Debug Options Window Help

Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (Intel)] on win32

Type "copyright", "credits" or "license()" for more information.

```
>>> import tva
```

```
>>> help(tva)
```

Help on module tva:

### NAME

tva

### DESCRIPTION

Module pour calcul des prix TTC  
Application de différents niveaux de TVA

### FUNCTIONS

ptto\_alimentaire(p)  
tva produits alimentaires - mais aussi travaux amélioration

ptto\_normal(p)  
tva normale

ptto\_reduit(p)  
tva intermédiaire - ex. travaux aménagement

### FILE

d:\\_travaux\university\cours\_universite\supports\_de\_cours\informa  
tique\python\slides\exemples\b\tva.py

Documentez vos modules, vous  
facilitez le travail des  
programmeurs qui les utilisent.

## Partie 2: Les packages

- ❑ Un *package* correspond à un répertoire sur le système de fichier : il a un nom (nom du *package*), et contient des fichiers (les modules). Les règles de nom des *packages* sont donc les mêmes que pour les modules.
- ❑ Les *packages* (paquets) sont des modules, mais qui peuvent contenir d'autres modules.
- ❑ Avant Python 3.5, pour être un *package*, un répertoire devait contenir un fichier `__init__.py`. Ce n'est plus obligatoire aujourd'hui, mais c'est toujours utile. Quand un *package* est importé, c'est en fait son module `__init__` qui l'est.

❑ Si nous créons un sous-répertoire **mypackage** dans le répertoire courant, et que nous y écrivons le fichier **\_\_init\_\_.py** suivant :

```
1 def myfunction():  
2     return None
```

❑ Alors **mypackage** est utilisable comme un module contenant une fonction myfunction.

```
1 >>> import mypackage  
2 >>> mypackage.myfunction()
```

❑ L'intérêt principal des *packages* étant tout de même de contenir plusieurs modules. On peut ainsi ajouter un fichier **operations.py** au répertoire **mypackage**.

```
1 def addition(a, b):  
2     return a + b  
3  
4 def soustraction(a, b):  
5     return a - b  
6  
7 def multiplication(a, b):  
8     return a * b
```

❑ On note aussi qu'il n'est pas nécessaire d'importer **mypackage** pour pouvoir importer **mypackage.operations**.

❑ Cela revient à disposer d'un module `mypackage.operations`. Mais ce module n'est par défaut pas importé dans le *package*: `import mypackage` ne donne par défaut pas accès à opérations, il faudra importer explicitement ce dernier.

```
1 >>> import mypackage.operations
2 >>> mypackage.operations.addition(1, 2)
3 3
4 >>> from mypackage import operations
5 >>> operations.soustraction(1, 2)
6 -1
7 >>> from mypackage.operations import multiplication
8 >>> multiplication(1, 2)
9 2
```

❑ Pour donner accès au module **operations** directement en important **mypackage**, il est nécessaire de toucher au fichier **\_\_init\_\_.py**. Ce fichier correspondant à ce qui est chargé à l'importation du *package*, nous pouvons importer **mypackage.operations**, ce qui le rendra directement accessible.

`__init__.py`

```
1 import mypackage.operations
```

Puis en console :

```
1 >>> import mypackage
2 >>> mypackage.operations.addition(1, 2)
3 3
```

❑ Un *package* est un niveau d'indirection supérieur au module, mais il est aussi possible d'avoir des *packages* de *packages*, et *packages* de *packages* de *packages*, et plus encore : vers l'infini et au-delà !