

# Programmation Python

## Procédures et fonctions

Dr. Yousfi Souheib

1<sup>er</sup> mars 2021

## Pourquoi créer des fonctions ?

- ➊ Meilleure organisation du programme (regrouper les tâches par blocs : lisibilité et maintenance)
- ➋ Éviter la redondance (pas de copier/coller : maintenance, meilleure réutilisation du code)
- ➌ Possibilité de partager les fonctions (via des modules)
- ➍ Le programme principal doit être le plus simple possible

## Un module sous Python ?

- ➊ Module = fichier « .py »
- ➋ On peut regrouper dans un module les fonctions traitant des problèmes de même nature ou manipulant le même type d'objet
- ➌ Pour charger les fonctions d'un module dans un autre module/programme principal, on utilise la commande `import nom_du_module`
- ➍ Les fonctions importées sont chargées en mémoire. Si collision de noms, les plus récentes écrasent les anciennes.

## Fonction ?

- Fonction : Bloc d'instructions
- Prend (éventuellement) des paramètres en entrée (non typés)
- Renvoie une valeur en sortie (ou plusieurs valeurs, ou pas de valeurs du tout : procédure)

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

- 1 def : définir une fonction, "petit" est son nom
- 2 Les paramètres ne sont pas typés
- 3 Noter le rôle du " :"
- 4 Attention à l'indentation
- 5 return provoque immédiatement la sortie de la fonction

## Procédure

Fonction sans return

# Appels des 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

# Appels des fonctions

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

# Valeur par défaut

## 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

# Visibilité des variables

## 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  $x = 10 + 99$

# 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.



## Modules

- Un module est un fichier « .py » contenant un ensemble de variables, fonctions et classes que l'on peut importer et utiliser dans le programme principal (ou dans d'autres modules).
- Le mot clé "import" permet d'importer un module

## Modules standards

- Des modules standards prêts à l'emploi Ex. random, math, os, etc.
- Ils sont visibles dans le répertoire « Lib » de Python, ou bien <https://docs.python.org/3/library/>

# Exemple d'utilisation de 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

# Autres utilisations possibles

```
#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)
```

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

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

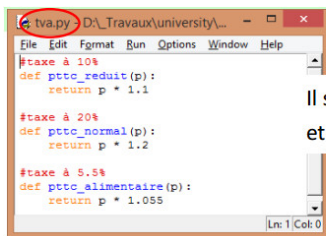
#utilisation directe
seed(None)
value = random()
logv = log(value)
abslog = pow(logv,2.0)
```

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.

# Création d'un module personnalisé

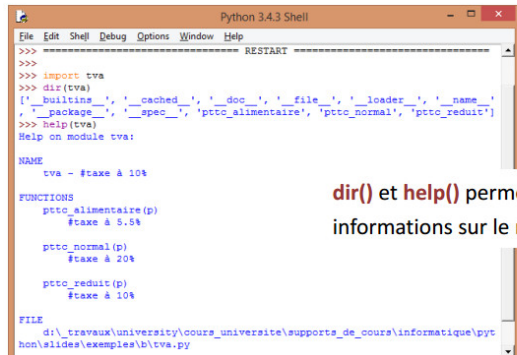


```
tva.py D:\Travaux\university\...
File Edit Format Run Options Window Help
#taxe à 10%
def ptto_reduit(p):
    return p * 1.1

#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
```

Il suffit de créer un fichier **nom\_module.py**,  
et d'y implémenter les fonctions à partager.



```
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\pyt
hon\slides\exemples\b\tva.py
```

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

# Création d'un module personnalisé

```
tva.py - D:\Travaux\university\Co...
File Edit Format Run Options Window Help

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

#taxe à 10%
def pttc_reduit(p):
    """tva intermédiaire - ex. travaux aménagement
    """
    return p * 1.1

#taxe à 20%
def pttc_normal(p):
    """tva normale
    """
    return p * 1.2

#taxe à 5.5%
def pttc_alimentaire(p):
    """tva produits alimentaires - mais au
    """
    return p * 1.055
```

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

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

Python 3.4.3 (vs4.4.3:9b73f1c3e601, 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
    pttc_alimentaire(p)
        tva produits alimentaires - mais aussi travaux amélioration

    pttc_normal(p)
        tva normale

    pttc_reduit(p)
        tva intermédiaire - ex. travaux aménagement

FILE
    d:\travaux\university\cours_universite\programmation_python\informa
    ti...
```