

Programmation modulaire sous Python

Procédures et fonctions

Découpage en modules des applications

Découpage des programmes

PROCÉDURES ET FONCTIONS

Pourquoi créer des fonctions ?

1. Meilleure organisation du programme (regrouper les tâches par blocs : lisibilité → maintenance)
2. Eviter 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

Qu'est-ce qu'un module sous Python ?

1. Module = fichier « .py »
2. On peut regrouper dans un module les fonctions traitant des problèmes de même nature ou manipulant le même type d'objet
3. Pour charger les fonctions d'un module dans un autre module / programme principal, on utilise la commande `import nom_du_module`
4. 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)

Un 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**

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

Un exemple de programme

```
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...

Ln: 45 Col: 0
```

Portée des variables, imbrications des fonctions, ...

PLUS LOIN AVEC LES FONCTIONS

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 regroupés 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

#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]

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.

[10, 8] [6]

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.

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

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
 $x = 10 + 99$

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.

Création et utilisation des modules

LES MODULES

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
- C'est un pas supplémentaire vers la modularité : un module maximise la réutilisation et facilite le partage du code

Modules standards

- Des modules standards prêts à l'emploi sont livrés avec la distribution Python. Ex. random, math, os, hashlib, etc.
- Ils sont visibles dans le répertoire « Lib » de Python

Voir la liste
complète sur



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

```
# -*- 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


```
#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.

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

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

#taxe à 5.5%
def pttc_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__', 'pttc_alimentaire', 'pttc_normal', 'pttc_reduit']
>>> help(tva)
Help on module tva:

NAME
    tva - #taxe à 10%

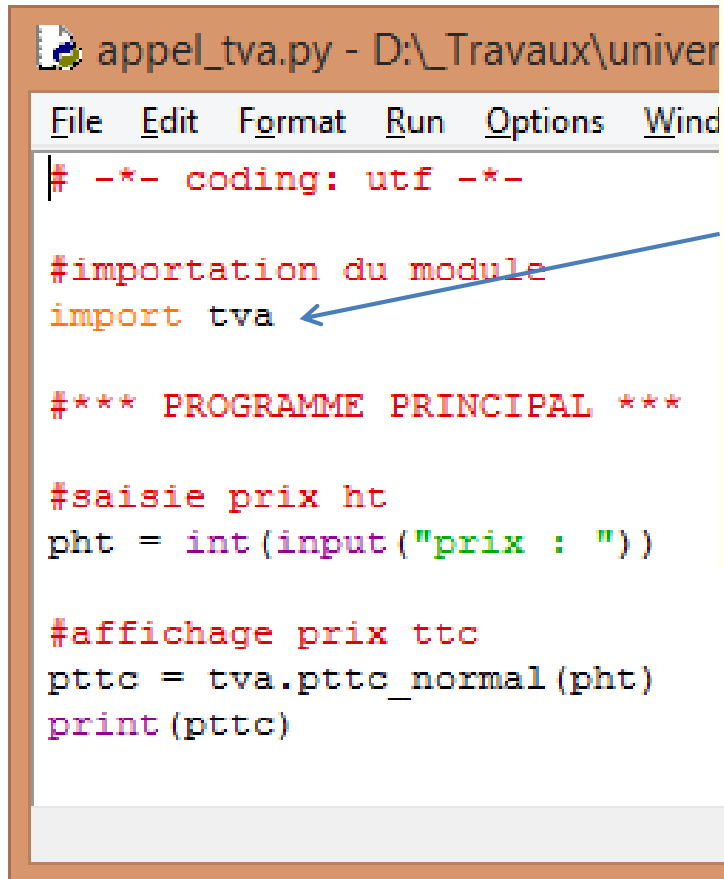
FUNCTIONS
    pttc_alimentaire(p)
        #taxe à 5.5%

    pttc_normal(p)
        #taxe à 20%

    pttc_reduit(p)
        #taxe à 10%

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

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



```
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 courrant
- 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
```

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 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 (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

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\supports_de_cours\informa
tique\python\slides\exemples\b\tva.py