

Institut Préparatoire aux Études d'Ingénieurs de Tunis



Leila Ben Othman

Première année de classes préparatoires
MP-PC

Année universitaire: 2016 - 2017

Les fonctions en PYTHON

Plan

① Fonction PYTHON

- Définition

- Objet retourné

- Appel de fonction

② Paramètres d'une fonction

- Passage de paramètres par position

- Passage de paramètres par nom

- Paramètres par défaut

- Mode de passage des paramètres

③ Variables locales/globales

④ Fonction locale/globale

⑤ Fonction lambda

⑥ Module

⑦ Fonctions récursives

Avant de commencer:

Cette partie suppose que les notions suivantes sont bien assimilées:

- Programmation modulaire;
- Paramètres formels et effectifs;
- Variables locales et globales;
- Mode de passage des paramètres

⇒ Voir partie: Fonctions et Procédures en algorithmique.

Fonction PYTHON

Une fonction en PYTHON:

- est un bloc d'instructions;
- prend (éventuellement) des paramètres non typés;
- retourne un objet, ou plusieurs objets en même temps.
- peut ne rien retourner (procédure).

Définition d'une fonction

En PYTHON, une fonction est définie à l'aide du mot clé **def**:

```
def nom_fonction(paramètres formels):  
    .....  
    .....<instructions>.....  
    .....  
    return(objet)
```

- N'oubliez pas les ":" c'est pour l'indentation.
- L'instruction **return** renvoie l'objet à retourner.
- L'instruction **return** provoque la sortie de la fonction.
- Procédure = Fonction sans **return**.

Objet retourné

- L'objet retourné par l'instruction **return** peut-être de n'importe quel type, *c'est-à-dire*, une fonction PYTHON peut retourner un entier (*int*), une chaîne de caractère (*str*), un dictionnaire (*dict*), *etc.*
- L'instruction **return** peut retourner plusieurs objets en même temps. Dans ce cas, la fonction retourne un tuple composé des objets retournés.
- On peut omettre l'instruction **return**. Dans ce cas, la fonction ne retourne rien (procédure) ou plus précisément, elle retourne la valeur *None* (*NoneType*).

Exemples de fonctions

```
def somme(a,b):  
    return(a+b)
```

```
def quotient_reste(a,b):  
    return(a//b,a%b)
```

```
def affiche(a):  
    print('La valeur est ',a)
```

Appel - Fonction renvoyant un ou plusieurs objets

Une fonction renvoyant un ou plusieurs objets est appelée comme suit:

- 1 Récupération du résultat de la fonction dans une variable

```
res=nom_fonction(paramètres effectifs)
```

- 2 Affichage du résultat de la fonction

```
print('le résultat est ', nom_fonction(paramètres effectifs))
```

Remarque: Il faudra néanmoins faire attention lorsque la fonction retourne plusieurs objets en même temps car il s'agit d'un tuple.

Appel - Fonction ne renvoyant pas d'objet

Une fonction ne renvoyant pas d'objet est appelée comme suit:

```
nom_fonction(paramètres effectifs)
```

Remarque: Attention, si vous faites appel à une fonction ne renvoyant pas d'objet avec la syntaxe `res=nom_fonction(paramètres effectifs)`. La variable `res` contiendra la valeur *None*.

Exemple

```
#définition des fonctions

#fonction qui retourne un objet (int)
def somme(a,b):
    return(a+b)

#fonction qui retourne deux objets en même temps, c'est donc un tuple
def quotient_reste(a,b):
    return(a//b,a%b)

#fonction qui ne retourne rien
def afficher(a):
    print('La valeur est ',a)

#Script principal

x=int(input('donner un entier'))
y=int(input('donner un entier'))
s=somme(x,y)
afficher(s)
qr=quotient_reste(x,y)
#qr est un tuple
afficher(qr)
```

Exemple - suite

```
#On peut manipuler le tuple qr retourné par la fonction quotient_reste
q=qr[0]
print("quotient= ",q)
r=qr[1]
print("reste = ", r)

#On peut parcourir les éléments du tuple comme ceci
print('premier parcours')
for i in qr:
    afficher(i)

#On peut également parcourir les éléments du tuple comme cela
print('deuxième parcours')
for i in range(len(qr)):
    afficher(qr[i])
```

Passage de paramètres par position

- Le passage de paramètres par position est le mode de passage où l'ordre (la position) des paramètres est important.
- Chaque paramètre formel sera substitué par le paramètre effectif correspondant selon sa position lors de l'appel.

Exemple:

- `quotient_reste(25,5)` $\Rightarrow$ renvoie (5,0)
- `quotient_reste(5,25)` $\Rightarrow$ renvoie (0,5)

Passage de paramètres par nom

- Le passage de paramètre par nom consiste à **faire appel à la fonction** en précisant *paramètre formel=paramètre effectif*

Example:

- quotient_reste(a=25,b=5) $\Rightarrow$ renvoie (5,0)
- quotient_reste(b=5,a=25) $\Rightarrow$ renvoie (5,0)

Paramètres par défaut (1)

- Le principe de paramètres par défaut consiste à affecter des valeurs aux paramètres **lors de la définition de la fonction**.
- Si le paramètre n'est pas précisé lors de l'appel, c'est la valeur par défaut qui sera utilisée.
- Si la valeur du paramètre est précisée lors de l'appel, c'est bien cette dernière qui sera utilisée.
- Les paramètres avec valeur par défaut doivent être regroupés à la fin dans la liste des paramètres.

Paramètres par défaut (2)

1 Définition de la fonction

```
# ici, le paramètre b possède une valeur par défaut égale à 5
def quotient_reste(a,b=5):
    return(a//b,a%b)
```

2 Appel de la fonction

```
# quand le deuxième paramètre n'est pas spécifié, PYTHON utilise la
valeur par défaut
(1) quotient_reste(25) ⇒ renvoie (5,0)
# quand le deuxième paramètre est spécifié, PYTHON utilise ce dernier
(2) quotient_reste(25,2) ⇒ renvoie (12,1)
```

Paramètres par défaut (3)

Attention, cette définition n'est pas correcte car les paramètres par défaut doivent être placés après les paramètres obligatoires

```
def quotient_reste(a=25,b):  
    return(a//b,a%b)
```

Cette définition est correcte

```
def quotient_reste(b,a=25):  
    return(a//b,a%b)
```

Mode de passage des paramètres

- Les paramètres sont toujours passés par référence, mais ils sont modifiables selon qu'ils sont mutables (listes, ensembles, dictionnaires) ou non mutables (types simples, chaînes de caractères et les tuples).

Exemple 1: on va essayer de modifier un objet non mutable (*int*)

```
def modifier_non_mutable(a):  
    a=99  
    print('La valeur locale est',a)
```

```
x=int(input('Donner un entier'))  
modifier_non_mutable(x)  
print('La valeur de x est',x)
```

Testez et interprétez

Exemple 2: on va essayer de modifier un objet mutable (*list*)

```
def modifier_mutable(L):  
    L.append(16)
```

```
maListe=[i for i in range(5)]  
print('La liste avant appel à la fonction',maListe)  
modifier_mutable(maListe)  
print('La liste après appel à la fonction',maListe)
```

Testez et interprétez

Variables locales/globales

- Une variable locale est une variable définie au niveau d'une fonction. Elle n'est visible que dans cette fonction.
- Par défaut, toute variable utilisée dans une fonction est une variable locale à cette fonction.
- Une variable globale est une variable définie au niveau du programme principal. Elle est donc visible par n'importe quelle fonction définie au niveau de ce programme.
- Pour qu'une fonction utilise une variable globale définie au niveau du programme principal, il faut employer le mot clé **global**.

De façon générale, il est déconseillé de faire usage de variables globales!!!

Variables locales/globales

Testez et interprétez

```
def f1(v):  
    x=v  
    # appel  
    x=10  
    f1(5)  
    print(x)
```

```
def f2(v):  
    x=x+v  
    # appel  
    x=10  
    f2(5)  
    print(x)
```

```
def f3(v):  
    global x  
    x=x+v  
    # appel  
    x=10  
    f3(5)  
    print(x)
```

Fonction locale/globale

- Il est possible de définir une fonction dans une autre fonction. Dans ce cas, on parle de fonction locale. Elle n'est visible que par la fonction dans laquelle elle a été définie.

```
def max3(a,b,c):  
    def max2(a,b):  
        if a>b:  
            return a  
        else:  
            return b  
    m=max2(a,b)  
    return(max2(m,c))  
  
x=24  
y=8  
z=12  
maxim=max3(x,y,z)  
print('le max est',maxim)
```

L'appel à la fonction max2 dans le script principal provoquera une erreur.

Fonction lambda

- La fonction lambda est utilisée lorsque la fonction est une expression mathématique.
- Une fonction lambda est définie à l'aide du mot clef **lambda**.

Syntaxe de définition d'une fonction lambda:

- ① `nom_fonction=lambda x:expression`
- ② `nom_fonction=lambda x,y:expression`
- ③ `nom_fonction=lambda x: expression1 if condition else expression2`
- ④ `nom_fonction=lambda x,y: expression1 if condition else expression2`

Fonction lambda

```
In [1]: f=lambda x:x**2
```

```
In [2]: f(2)
```

```
Out [2]: 4
```

```
In [3]: somme=lambda x,y:x+y
```

```
In [4]: somme(5,4)
```

```
Out [4]: 9
```

```
In [5]: valeur_absolue=lambda x:x if x>0 else -x
```

```
In [6]: valeur_absolue(5)
```

```
Out [6]: 5
```

```
In [7]: valeur_absolue(-5)
```

```
Out [7]: 5
```

Module- Bibliothèque

- Un module est un ensemble de fonctions définies dans un fichier .py
- Un module favorise la réutilisation du code et facilite le partage ⇒ Modularité
- On a déjà utilisé et importé les modules `math`, `copy` et `random`.
- Pour charger les fonctions d'un module, on utilise la commande `import`
 - `import nom_module`
 - `import nom_module as alias` (celle que je vous recommande)
 - `from nom_module import *`
 - `from nom_module import nom_fonction`
- On va prochainement utiliser les modules `numpy` et `scipy` pour la manipulation des tableaux et faire du calcul scientifique/statistique.

À vous de jouer: créez vos propres modules de fonctions.

Les fonctions récursives

Factorielle

$$\begin{cases} 0! = 1 \\ n! = n \times (n-1) \times \dots \times 1 \end{cases}$$

```
def fact(n):  
    res=1  
    for i in range(2, n+1):  
        res=res*i  
    return(res)
```

⇒ version itérative de la fonction factorielle

Les fonctions récursives

Une autre façon de formaliser la fonction factorielle

Factorielle

$$\begin{cases} 0! = 1 \\ n! = n \times (n-1)! \end{cases}$$

```
def fact(n):  
    if n==0:  
        return(1)  
    else:  
        return(n*fact(n-1))
```

⇒ version récursive de la fonction factorielle

Les fonctions récursives

Qu'est-ce qu'une fonction récursive?

- Une fonction récursive est une fonction qui contient un appel à elle-même.

Comment écrire une fonction récursive?

- Identifier le cas de base
- Identifier le cas général qui effectue la récursion (les appels récursifs)

Les fonctions récursives

Pourquoi un cas de base?

- Le cas de base est le cas qui précise la **condition d'arrêt** des appels récursifs.
- Dans la fonction factorielle, il faut arrêter les appels récursifs lorsque $n=0$ car dans ce cas on connaît le résultat.
- Attention: Absence de cas de base $\Rightarrow$ suite infinie d'appels récursifs!

Les fonctions récursives

Comment ça marche?

- Les appels récursifs se déroulent selon le principe LIFO (Last-In-First-Out ou dernier arrivé premier servi).
- On peut représenter ces appels avec une structure en pile.

Les fonctions récursives

Le calcul de $\text{fact}(4)$

Appel	n	Appel récursif	Résultat
$\text{fact}(0)$	0	arrêt	1
$\text{fact}(1)$	1	$1 \times \text{fact}(0)$	$1 \times 1 = 1$
$\text{fact}(2)$	2	$2 \times \text{fact}(1)$	$2 \times 1 = 2$
$\text{fact}(3)$	3	$3 \times \text{fact}(2)$	$3 \times 2 = 6$
$\text{fact}(4)$	4	$4 \times \text{fact}(3)$	$4 \times 6 = 24$