Introduction à la Récursivité

Mise en situation

Une fonction récursive est une fonction qui s'appelle elle-même.
Les techniques de programmation récursives sont particulièrement puissantes dans la résolution de certains types de problèmes (parcours d'arbres, de labyrinthe, tri rapide …).
Les fonctions récursives sous leur aspect souvent "épuré" cachent un certain nombre de notions fondamentales et le mécanisme de la récursivité n'est pas toujours véritablement compris par les étudiants qui peuvent réaliser des programmes récursifs qui ne sont en réalité que des usines à gaz ...

Problématique : comment l'interpréteur d'un langage exécute-t-il une fonction récursive ?
Notions abordées : Variable locale/globale, pile.
Prérequis: avoir déjà une bonne base de l'algorithmique itérative

Appel d'une fonction classique

Avant d'aborder une fonction récursive il est impératif de comprendre ce que fait l'interpréteur de langage lorsqu'on fait un appel de fonction classique.

Exécutons les 2 programmes ci-dessous et voyons les différences:

```
1  def toto():
2      a = 10
3      print ("Dans la fonction a= ",a)
4
5  a = 5
6  toto()
7  print ("A l'extérieure de la fonction a= ",a)
8
```

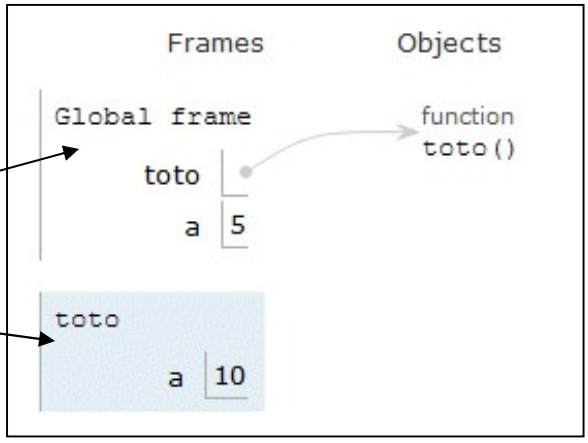
```
Dans la fonction a=  10
A l'extérieure de la fonction a=  5
```

Dans ce **1er programme**, il y a DEUX variables "a", elles portent le même nom mais sont différentes.
Lorsque l'interpréteur python lit la première variable 'a' (ligne 5), elle est créée en mémoire.
Lors de l'appel de la fonction toto(), l'interpréteur va **créer** une nouvelle variable 'a' (ligne 2) dans un espace mémoire associée à la fonction

Tester le programme dans python tutor :
http://www.pythontutor.com/visualize.html#mode=edit

Zone mémoire variables globales

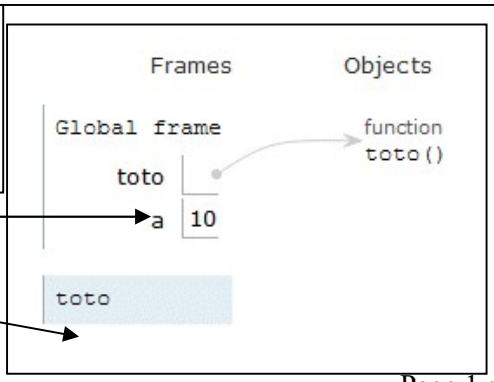
Zone mémoire variables locales de la fonction toto




```
def toto():
    global a
    a = 10
    print ("Dans la fonction a= ",a)

a = 5
toto()
print ("A l'extérieure de la fonction a= ",a)
```

```
Dans la fonction a=  10
A l'extérieure de la fonction a=  10
```

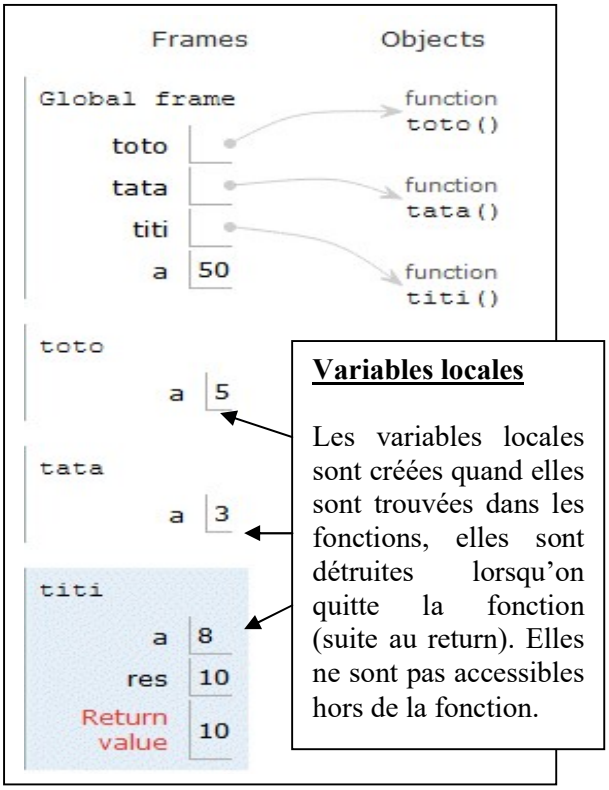
Pas de variable locale à toto, c'est la variable **globale** a qui est modifiée . Cette variable est unique en mémoire



Voyons maintenant le cas de figure suivant :

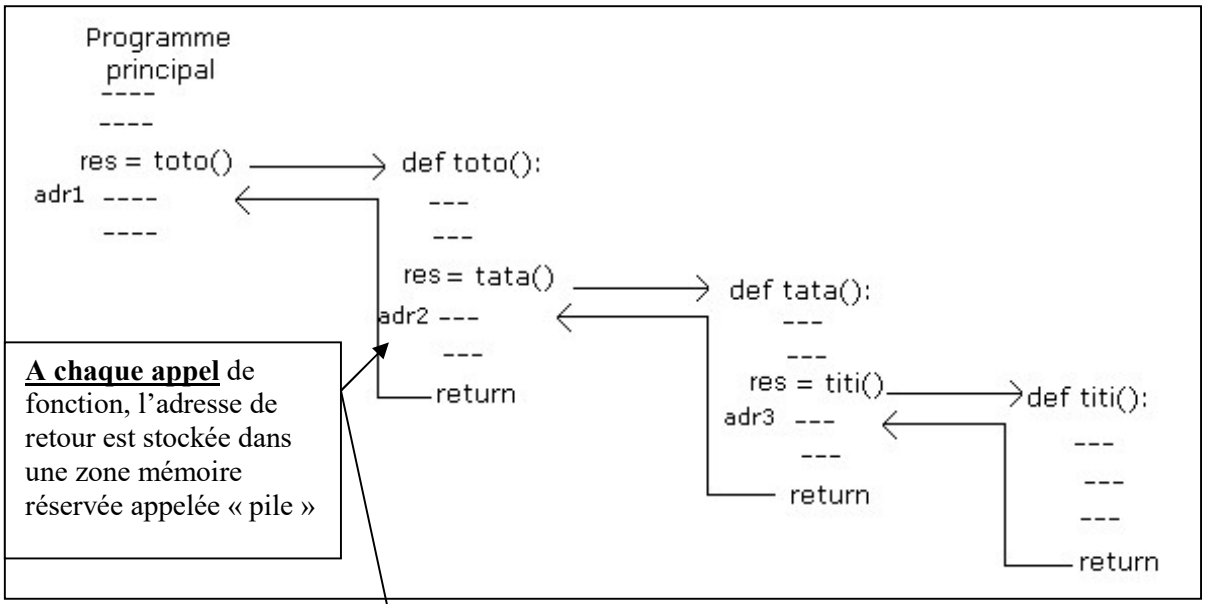
```
1   def toto():
2       a = 5
3       res = tata()
4       print ("adr2")
5       return (res)
6
7   def tata():
8       a = 3
9       res = titi()
10      print ("adr3")
11      return (res)
12
13  def titi():
14      a = 8
15      print ("je suis dans titi")
16      res = 10
17      return (res)
18
19  a = 50
20  res = toto()
21  print ("adr1")
22  print ("res = ",res)
23
```

```
je suis dans titi
adr3
adr2
adr1
res =  10
```



Variables locales

Les variables locales sont créées quand elles sont trouvées dans les fonctions, elles sont détruites lorsqu'on quitte la fonction (suite au return). Elles ne sont pas accessibles hors de la fonction.



A chaque appel de fonction, l'adresse de retour est stockée dans une zone mémoire réservée appelée « pile »

	Res=toto()	Res=tata()	Res=titi()	Return de titi()	Return de tata()	Return de toto()
			Adr3			
		Adr2	Adr2	Adr2		
	Adr1	Adr1	Adr1	Adr1	Adr1	
PILE	**PILE**	**PILE**	**PILE**	**PILE**	**PILE**	**PILE**

`res = toto()` : On empile l'adresse Adr1 de retour
`res = tata()` : On empile l'adresse Adr2 de retour
`res = titi()` : On empile l'adresse Adr3 de retour

Lorsque l'interpréteur rencontre « return » il va récupérer (« dépiler ») la dernière adresse empilée pour savoir où revenir.

On pourrait imbriquer les fonctions comme ceci autant qu'on veut théoriquement mais on est en pratique limité par l'espace mémoire attribuée à la pile. Cet espace peut varier suivant les langages de programmation. Avec python 3.7, on est limité à 996 appels imbriqués avant que l'interpréteur ne génère un message d'erreur pile pleine. Dans les anciens langage basic (années 1980 sur les TO7-70) , on était limité à 8 imbrications de sous programmes et la notion de variables globales et locales n'existaient pas ...

A RETENIR : Quelque soit le langage ou la méthode de programmation (itérative ou récursive), dès le moment où dans votre programme vous avez un appel de fonction, la pile va OBLIGATOIREMENT sauvegarder l'adresse de retour de l'appel : c'est l'empilement

De même, lorsque vous avez un return (ou une sortie de procédure c-a-d de fonction sans renvoie de données) , vous avez forcément un dépilement. Notons qu'en python, le return n'est pas obligatoire si on ne renvoie pas de données. Il n'empêche que le retour de fonction (avec ou sans return) se fait grâce au **dépilement**. C'est fondamental d'avoir cela en mémoire si on veut comprendre « finement » ce qui se passe dans les programmes récursifs en particulier.

Comprendre vraiment la notion de récursivité

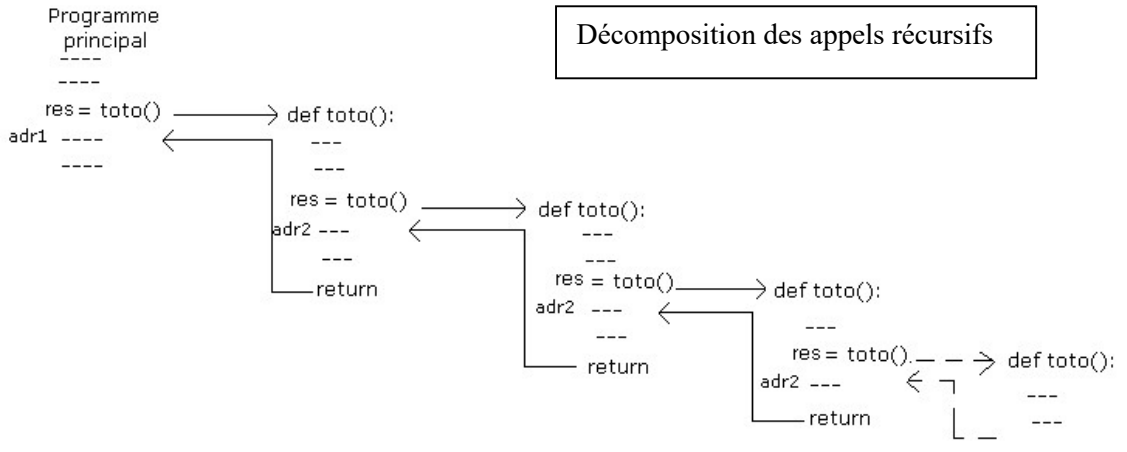
Testons le programme « récursif » suivant :

```
def toto() :
    global n
    a = 1
    n = n+1
    print (n,end=" ")
    res = toto()
    return

# programme principal :
n = 0
res = toto()
```

Dans le programme principal, on appelle la fonction `toto()` puis dans `toto()`, on appelle à nouveau `toto()` : on voit que c'est une boucle sans fin , mais en réalité il va s'arrêter sur une erreur de pile (maximum de récursions atteinte au bout 996 appels et 2954 sous Ipython anaconda ...)

```
RecursionError: maximum recursion depth exceeded
```

Décomposition des appels récursifs



C'est exactement comme ce qu'on a vu précédemment pour des appels de fonctions imbriquées, sauf que maintenant toutes les fonctions s'appellent toto() ! Pour la pile, on a va avoir l'adresse adr1 suivi d'un **empilement d'adresse adr2**, normal puisque c'est la même fonction qui est appelée. Pour la variable « a », à chaque appel de la fonction, c'est une nouvelle variable locale « a » qui est crée (comme antérieurement).

Comme notre programme d'exemple est stupide, il boucle sans fin car ne fait jamais de return. Il manque ce qu'on appelle une condition d'arrêt de la récursion ce qui fait qu'on ne fait que des empilement et pas de dépilement. Mais maintenant, on va voir un programme (un peu plus intelligent) de calcul factoriel et pouvoir comprendre vraiment ce qui se passe.

Programme de calcul factoriel en version récursive et version itérative :

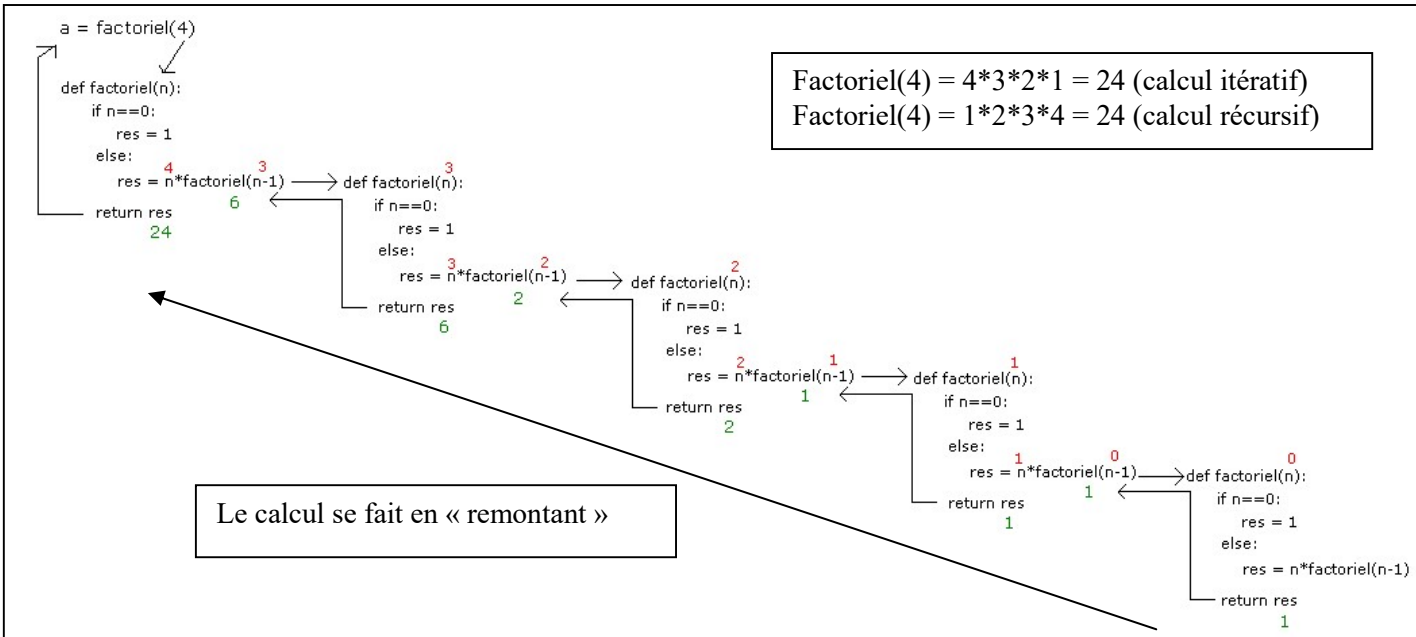
Version récursive

```
def factoriel(n):
    if n == 0:
        res = 1          # condition d'arrêt
    else:
        res = n*factoriel(n-1)   # appel récursif avec n-1

    return res
print ("Factoriel(4) =",factoriel(4))
```

```
Factoriel(4) = 24
```

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

print (factoriel(4))
```

Version itérative



Remarques :

1- Le calcul factoriel se fait en « remontant », il faut en effet que la routine arrive jusqu'à la condition d'arrêt pour pouvoir faire le calcul (à savoir n = 0).
Dans tout programme récursif il faut impérativement une **condition d'arrêt de la récursion !**

2- Les variables res et n sont locales et donc crées à chaque nouvel appel de fonction et détruites à chaque sortie de fonction !

3- A chaque appel récursif, on dit que l'on descend d'un niveau et à chaque retour de fonction on remonte d'un niveau (Attention, en python on peut avoir des procédures récursives sans return, mais il est alors sous entendu !)

La profondeur de la récursion donne ici la limite de notre programme : on ne pourra pas calculer `factoriel(5000)` par contre en version itérative, la seule limite sera la capacité de calcul de python à traiter les grands nombres !

L'utilisation de la récursivité peut sembler assez abstraite, les programmes récursifs sont souvent déroutants à première vue, surtout pour les débutants. Cependant, **lorsque le principe est bien maîtrisé (il faut beaucoup de pratique !)**, cela peut faciliter grandement la résolution de certains problèmes : cela devient presque une façon de penser ou de raisonner : certains problèmes sont plus facile à aborder récursivement qu'itérativement. Un bon programmeur est capable d'utiliser les deux techniques (récursive ou itérative) et ce **à bon escient.**

Exercice à faire : Tester le programme en rajoutant des print avant et après chaque appel récursif pour afficher le contenu des variables. S'aider aussi de python tutor.
http://www.pythontutor.com/visualize.html#mode=edit