PROGRAMMATION ITERATIVE ET PROGRAMMATION RECURSIVITE

#### **Objectifs**

* Être capable d'écrire un programme récursif
* Être capable d'analyser le fonctionnement d'un programme récursif.

I/ **Pile d'exécution**

Analysez puis testez le programme suivant :

def fctA():

print ("Début fonction fctA")

i=0

while i<5:

print(f"fctA {i}")

i = i + 1

print ("Fin fonction fctA")

def fctB():

print ("Début fonction fctB")

i=0

while i<5:

if i==3:

fctA()

print("Retour à la fonction fctB")

print(f"fctB {i}")

i = i + 1

print ("Fin fonction fctB")

fctB()

Vous devriez obtenir l'enchainement suivant :

Début fonction fctB

fctB 0

fctB 1

fctB 2

Début fonction fctA

fctA 0

fctA 1

fctA 2

fctA 3

fctA 4

Fin fonction fctA

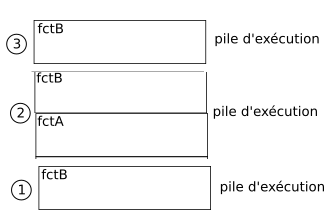
Retour à la fonction fctB

fctB 3

fctB 4

Fin fonction fctB

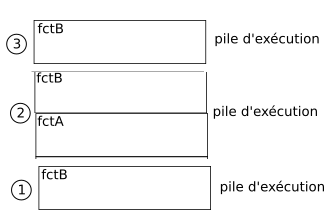
Dans l'exemple ci-dessus, nous avons une fonction (fctB) qui appelle une autre fonction (fctA). La principale chose à retenir de cet exemple est que l'exécution de fctB est interrompue pendant l'exécution de fctA. Une fois l'exécution de fctA terminée, l'exécution de fctB reprendra là où elle avait été interrompue.

Pour gérer ces fonctions qui appellent d'autres fonctions, le système utilise une "pile d'exécution". Une pile d'exécution permet d'enregistrer des informations sur les fonctions en cours d'exécution dans un programme. On parle de pile, car les exécutions successives "s'empilent" les unes sur les autres. Si nous nous intéressons à la pile d'exécution du programme étudié ci-dessus, nous obtenons le schéma suivant :

Nous pouvons "découper" l'exécution de ce programme en 3 parties :

1. la fonction fctB s'exécute jusqu'à l'appel de la fonction fctA
2. l'exécution de la fctB est mise en "pause" pendant l'exécution de la fonction fctA
3. une fois que l'exécution de fctA est terminée, on termine l'exécution de la fonction fctB

Il est important de bien comprendre que la fonction située au sommet de la pile d'exécution est en cours d'exécution. Toutes les fonctions situées "en dessous" sont mises en pause jusqu'au moment où elles se retrouveront au sommet de la pile. Quand une fonction termine son exécution, elle est automatiquement retirée du sommet de la pile (on dit que la fonction est dépilée).

La pile d'exécution permet de retenir la prochaine instruction à exécuter au moment où une fonction sera sortie de son ""état de pause" (qu'elle se retrouvera au sommet de la pile d'exécution) :



Dans l'exemple ci-dessus, on retrouve une variable i dans les deux fonctions : fctA et fctB. La variable i présente dans la fonction fctA n'a rien à voir avec la variable i présente dans la fonction fctB (elles portent le même nom, mais elles représentent 2 adresses mémoires différentes). Il est très important de bien comprendre que les variables créées dans une fonction ne "sortent" pas de la fonction : chaque fonction possède sa propre liste de variable, comme déjà dit ci-dessus la variable i de la fonction fctB est différente de la variable i de la fonction fctA.

La pile d'exécution conserve une "trace" des valeurs des variables lorsqu'une autre fonction est exécutée. Par exemple la valeur de i (fctB) est conservée au moment de l'exécution de fctA. Quand l'exécution de fctA se termine est que l'exécution de fctB "reprend", la valeur référencée par i (fctB) a été "conservée" (voilà pourquoi on reprend l'exécution de fctB avec un "fctB 3").

### II/ **Fonction récursive**

Une fonction peut s'appeler elle-même, on parle alors de fonction **récursive**.

Analysez puis testez le programme suivant :

def fctA():

print ("Hello")

fctA()

fctA()

Comme vous pouvez le constater, nous avons une erreur dans la console Python :

RecursionError: maximum recursion depth exceeded while calling a Python object  
 (*profondeur de récursion maximale dépassée lors de l’appel d’un objet Python)*

Dans le cas où une fonction s'appelle elle-même (fonction récursive), on retrouve le même système de pile d'exécution. Dans l'exemple traité ci-dessus, les appels s'enchainent sans rien pour mettre un terme à cet enchainement, la taille de la pile d'exécution augmente sans cesse (aucune fonction ne termine son exécution, nous n'avons pas de "dépilement" juste des "empilements"). Le système interrompt le programme en générant une erreur quand la pile d'exécution dépasse une certaine taille.

Quand on écrit une fonction récursive, il est donc nécessaire de bien penser à mettre en place une structure qui à un moment ou à un autre mettra fin à ces appels récursifs.

Dans le cas de fonctions récursives, il est, comme pour n'importe quelle fonction, possible d'utiliser des paramètres :

Essayez de prévoir le résultat de l'exécution du programme ci-dessous. Vérifiez votre hypothèse en exécutant le programme.

def fonct(n):

if n>0:

fonct(n-1)

print(n)

fonct(3)

**III/ Exemple de fonction récursive (**Tri par insertion)

III-1/ **Principe et implémentation**

Tri par insertion version itérative:

Autre exemple

def tri\_ins(t):

for k in range(1,len(t)):

temp=t[k]

j=k

while j>0 and temp<t[j-1]:

t[j]=t[j-1]

j-=1

t[j]=temp

return t

def tri\_insertion(T):

n = len(T)

for i in range(1,n):

x = T[i]

p = i

while p > 0 and T[p-1]> x:

T[p]=T[p-1]

p=p-1

T[p]=x

Return T

#test

T= array([2,4,5,1,0])

print(T)

print(tri\_insertion(T))

Tri par insertion version récursif

def insert\_elt(x,T):

Autre exempledef insere(t,j):

if j>0 and t[j]<t[j-1]:

t[j-1],t[j]=t[j],t[j-1]

insere(t,j-1)

def tri\_ins(t,j=1):

if j<len(t):

insere(t,j)

tri\_ins(t,j+1)

if j == (len(t)-1):

print(t)

# Test

t=[12,5,-12,7,6]

print("before:",t)

T=tri\_ins(t,j=1)

print("after:",t)

n=len(T)

p=n-1

e=T(p)

while p>0 and T[p-1]>x:

T[p]=T[p-1]

p=p-1

T[p]=x

T=np.append(T,e)

return T

def tri\_insertion(T,i):

n=len(T)

if n==0 or i==n:

return T

else:

R=insert\_elt(T[i],T[:i])

T=np.append(R,T[i+1:]

i+=1

return tri\_insertion(T,i)

# Test

T=np.array([12,5,-12,7,6])

Print("before:",T)

T=tri\_insertion(T,1)

Print("after:",T)