COURS DE NSI (Numérique et Sciences Informatiques)

Classe : terminale	LYCÉE INTERNATIONAL COURS LUMIÈRE
séquence 2	**Récursivité**
Mr BANANKO K.	

C’est quoi la La récursivité ?

La récursivité est un concept fondamental de la programmation qui fait référence à la capacité d'une fonction ou d'un algorithme à s'appeler elle-même pour résoudre un problème de manière itérative. En d'autres termes, une fonction récursive est une fonction qui s'auto-appelle pour résoudre un problème de manière répétée jusqu'à ce qu'une condition de base soit atteinte pour mettre fin à la récursion.

Les fonctions récursives sont généralement composées de deux parties principales :

1. **Le cas de base** : Il s'agit de la condition qui détermine quand la récursion doit s'arrêter. C'est la base de la récursion. Une fois que le cas de base est atteint, la fonction récursive cesse de s'appeler elle-même.
2. **Le cas récursif** : Il s'agit de la partie de la fonction qui s'appelle elle-même avec une version simplifiée du problème. Cette étape est cruciale pour progresser vers le cas de base.

Nous verrons en détail ces notions dans ce cours

Activité 1

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
```

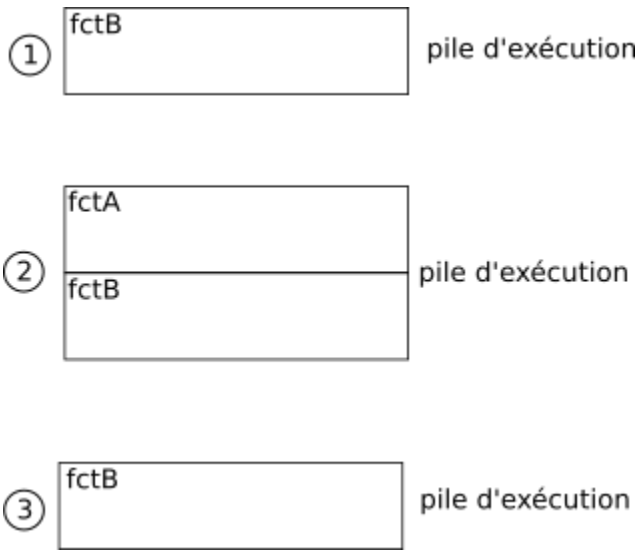
Dans l'exemple ci-dessus, nous avons une fonction (fctB) qui appelle une autre fonction (fctA).

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 :

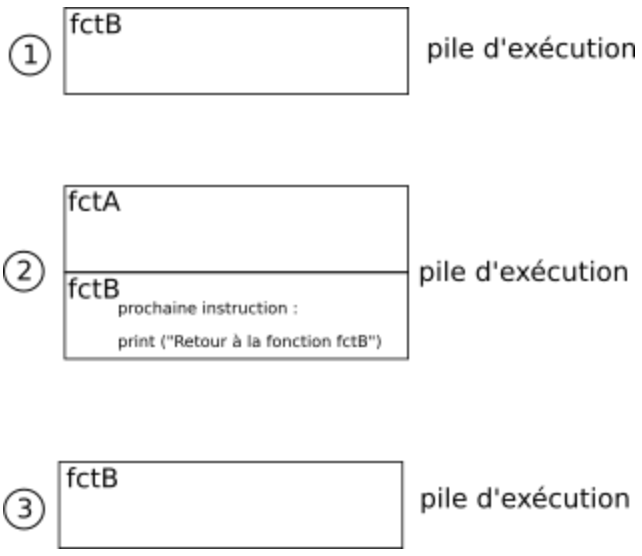


Nous pouvons "**découper**" l'exécution de ce programme en 3 parties :

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) :



Évidemment l'explication donnée ci-dessus est quelque peu simpliste : c'est l'adresse mémoire de la prochaine instruction machine à exécuter qui est conservée dans 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").

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

Activité 2

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
```

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 :

Activité 3

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

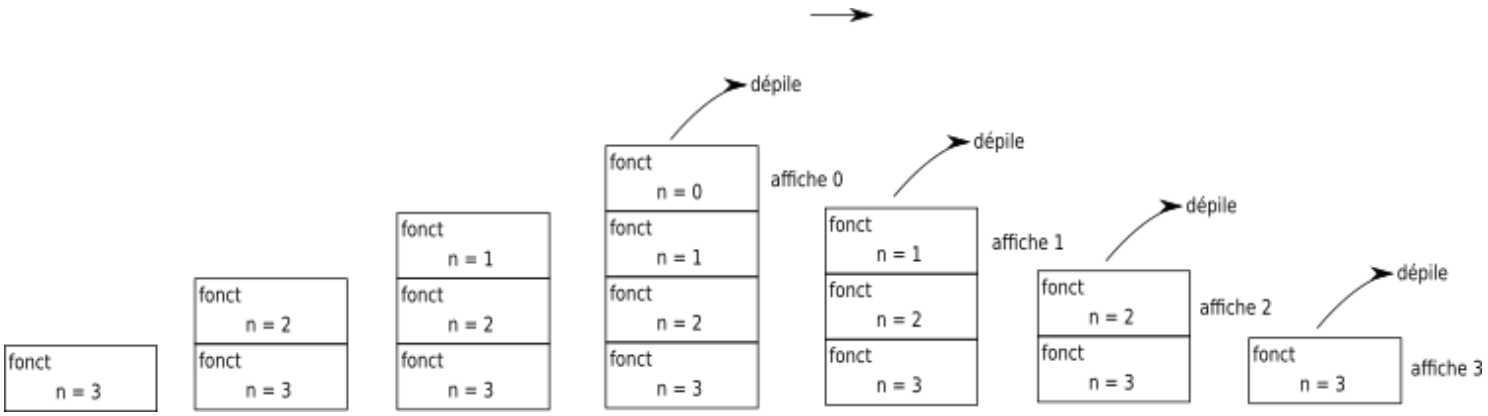
```
def fonct(n):
    if n>0:
        fonct(n-1)
    print(n)
fonct(3)
```

Essayons de comprendre en détail ce qui se passe dans le programme ci-dessus:

- 1er appel de la fonction fonct avec le paramètre n = 3 ; n > 0 donc appel de la fonction fonct avec le paramètre n = 2
- 2e appel de la fonction fonct avec le paramètre n = 2 ; n > 0 donc appel de la fonction fonct avec le paramètre n = 1
- 3e appel de la fonction fonct avec le paramètre n = 1 ; n > 0 donc appel de la fonction fonct avec le paramètre n = 0

- 4e appel de la fonction fonct avec le paramètre n = 0 ; n = 0 donc on exécute l'instruction print(n) => affichage : 0
- on "**dépile**" (3e appel, n = 1) : on exécute l'instruction print(n) => affichage : 1
- on "**dépile**" (2e appel, n = 2) : on exécute l'instruction print(n) => affichage : 2
- on "**dépile**" (1er appel, n = 3) : on exécute l'instruction print(n) => affichage : 3

Voici un schéma expliquant le processus en termes de pile d'exécution :



Il ne faut jamais perdre de vu qu'à chaque nouvel appel de la fonction fonct le paramètre n est différent.

Nous allons étudier le calcul de la factorielle grâce à une fonction récursive. D'après Wikipédia : "En mathématiques, la factorielle d'un entier naturel n est le produit des nombres entiers strictement positifs inférieurs ou égaux à n". Par exemple : la factorielle de 3 est : 3 x 2 x 1 = 6 ; la factorielle de 4 est 4 x 3 x 2 x 1 = 24 ; la factorielle de 5 est 5 x 4 x 3 x 2 x 1 = 120 ...

Si on note la factorielle de n par n!, on a :

- 0! = 1 (par définition)
- Pour tout entier n > 0, n! = n x (n – 1)!

Nous allons utiliser cette définition de la factorielle pour définir notre fonction récursive (nous allons utiliser le fait que la factorielle de n dépend de la factorielle de n-1 et que 0! = 1)

Activité 4

Analysez puis testez la faction fact à l'aide de la console Python :

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

Comme vous pouvez le constater, la fonction fact est structurée de la même manière que la définition mathématique vu ci-dessus :

- dans le cas où n = 0 la fonction renvoie 1 (0! = 1)
- dans le cas où n > 0 la fonction renvoie n*fact(n-1) (n! = n x (n – 1)!)

L'utilisation des fonctions récursives est souvent liée à la notion de récurrence en mathématiques :

En mathématiques une suite définie par récurrence est une suite définie par son premier terme et par une relation de récurrence, qui définit chaque terme à partir du précédent ou des précédents lorsqu'ils existent.

Prenons l'exemple de la suite de Fibonacci qui est définie par :

- $u_0 = 0$, $u_1 = 1$
- et par la relation de récurrence suivante avec n entier et n > 1 :

$$u_n = u_{n-1} + u_{n-2}$$

Ce qui nous donne pour les 6 premiers termes de la suite de Fibonacci :

- $u_0 = 0$
- $u_1 = 1$
- $u_2 = u_1 + u_0 = 1 + 0 = 1$
- $u_3 = u_2 + u_1 = 1 + 1 = 2$
- $u_4 = u_3 + u_2 = 2 + 1 = 3$

- $u_5 = u_4 + u_3 = 3 + 2 = 5$

Activité 5

En vous aidant de ce qui a été fait pour la fonction fact (voir le "À faire vous-même 4"), écrivez une fonction récursive fib qui donnera le n ième terme de la suite de Fibonacci. Cette fonction prendra en paramètre l'entier n.

Pour l'exemple suivant, nous allons utiliser le module Python Turtle. Ce module permet de dessiner très simplement.

Activité 6

Étudiez le Wikibook consacré au module Turtle (wikibook Turtle) afin d'acquérir les bases de ce module.

Activité 7

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

```
import turtle as t
t.forward(100)
t.left(120)
t.forward(100)
t.left(120)
t.forward(100)
```

Nous allons maintenant étudier le flocon de Koch :

Activité 8

Visionnez la vidéo consacrée au flocon de Koch : https://youtu.be/PW_Pka9iBko?si=M3ARxTOMsMsRKNAZ

(Egalement sur Pronote)

Activité 9

Après avoir testé le programme ci-dessous, vous l'étudierez attentivement. Vous vous concentrerez notamment sur le rôle des paramètres taille et etape de la fonction flocon.

```
import turtle as t


def koch(longueur, n):
    if n == 0:
        t.forward(longueur)
    else:
        koch(longueur/3, n-1)
        t.left(60)
        koch(longueur/3, n-1)
        t.right(120)
        koch(longueur/3, n-1)
        t.left(60)
        koch(longueur/3, n-1)

def flocon(taille, etape):
    koch(taille, etape)
    t.right(120)
    koch(taille, etape)
```

```
    t.right(120)
    koch(taille, etape)


flocon(100, 3)
```

Maintenant, expliquons ce que fait ce code :

1. Le code commence par importer la bibliothèque **turtle** sous le pseudonyme **t**. La bibliothèque **turtle** permet de dessiner des formes graphiques en utilisant une tortue virtuelle.
2. La fonction **koch** est définie pour dessiner un segment de la fractale de Koch. Elle prend deux paramètres :
    - **longueur**: La longueur du segment à dessiner.
    - **n**: Le niveau de récursion, déterminant le nombre d'itérations pour créer le segment.
3. À l'intérieur de la fonction **koch**, il y a une structure conditionnelle :
    - Si **n** est égal à 0 (cas de base), la tortue avance de la longueur spécifiée, ce qui dessine un segment droit.
    - Si **n** n'est pas égal à 0 (cas récursif), la fonction **koch** est appelée quatre fois pour diviser le segment en quatre parties. Cela crée les côtés d'un triangle équilatéral avec un motif de Koch.
4. La fonction **flocon** est définie pour dessiner un flocon de Koch complet. Elle prend deux paramètres :
    - **taille**: La taille initiale de chaque côté du triangle équilatéral qui forme le flocon.
    - **etape**: Le niveau de récursion, qui détermine la complexité du flocon.
5. À l'intérieur de la fonction **flocon**, elle appelle la fonction **koch** trois fois pour dessiner les trois côtés du triangle équilatéral. Entre chaque appel, la tortue tourne de 120 degrés pour compléter le triangle.
6. Enfin, le code appelle la fonction **flocon** avec une taille de 100 et 3 étapes de récursion pour dessiner un flocon de Koch avec trois niveaux de détails.

Le code **flocon(100, 10)** ne fonctionne pas correctement pour plusieurs raisons:

1- **Profondeur de la récursion** :

Le flocon de Koch est généré à l'aide d'un algorithme récursif, et le paramètre **etape** dans la fonction flocon détermine le niveau de récursion.

Lorsque vous utilisez une valeur de 10 pour **etape**, cela entraîne un très grand nombre d'appels récursifs, ce qui peut provoquer une **RecursionError** ou ralentir considérablement le programme en raison du nombre énorme d'itérations récursives.

2- **Limitations de la bibliothèque Turtle Graphics** :

La bibliothèque **Turtle Graphics** en Python peut avoir des limitations en ce qui concerne la profondeur de la récursion et la précision du dessin à des échelles extrêmement petites. Dessiner un flocon de Koch avec **etape** réglé à 10 peut ne pas être pratique en raison de ces limitations.

*Une alternative serait de réduire la valeur du paramètre **etape** à une valeur plus petite pour créer un flocon de Koch avec moins d'itérations, ce qui devrait fonctionner sans problème.*

*Par exemple, vous pourriez essayer **flocon(100, 5)** ou **flocon(100, 6)** pour obtenir un flocon plus détaillé sans atteindre les limites de la profondeur de récursion.*

Ce qu'il faut savoir

- Une fonction récursive est une fonction qui s'appelle elle-même
- Quand on écrit une fonction récursive, il est nécessaire de bien penser à mettre en place une structure qui à un moment ou à un autre mettra fin à ces appels récursifs.
- L'utilisation des fonctions récursives est souvent liée à la notion de récurrence en mathématiques.

Ce qu'il faut savoir faire

- Analyser le fonctionnement d'un programme récursif (programme récursif = programme qui comporte au moins une fonction récursive).
- être capable d'écrire une fonction récursive simple

EXERCICES DU BAC

- ***Sujet 1 2021 Exercice 2***
- ***Sujet 5 2021 Exercice 4***
- ***Sujet 1 2022 Exercice 1***
- ***Sujet 7 2022 Exercice 1***