TNSI	Langages & programmation Récursivité	Cours - Récursivité

Objectifs :

- ⇨ Savoir ce qu'est la récursivité en programmation
- ⇨ Programmer un exemple simple de fonction récursive
- ⇨ Connaitre les avantages et les inconvénients de la récursivité

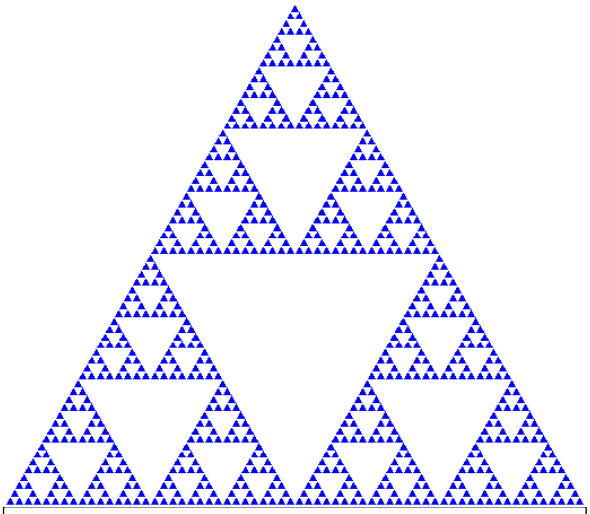
I - Qu'est-ce que la récursivité ?

Définition : "Une procédure récursive est une procédure récursive."

Dit comme cela, ce n'est pas forcément explicite et pourtant nous verrons que cela définit assez bien la récursivité. Prenons donc une définition plus compréhensible :

Définition 2 : Une procédure récursive est une procédure qui s'appelle elle-même.

D'une manière générale, la récursivité consiste pour un objet à être défini par lui-même comme dans l'exemple du triangle de Sierpiński ci-contre.



Le triangle de Sierpiński est défini par récursivité

La récursivité est un domaine très intéressant de l'informatique, un peu abstrait, mais très élégant ; elle permet de résoudre certains problèmes d'une manière très rapide, alors que si on devait les résoudre de manière itérative (classique, avec des boucles), il nous faudrait beaucoup plus de temps et de structures de données intermédiaires.

Dans ce qui suit, nous allons utiliser le terme "procédure" pour désigner aussi bien une procédure qu'une fonction.

Une procédure récursive comporte un appel à elle-même, alors qu'une procédure non récursive ne comporte que des appels à d'autres procédures.

Toute procédure récursive comporte une instruction (ou un bloc d'instructions) nommée « point terminal » ou « point d'appui » ou « **point d'arrêt** » ou encore « cas de base », qui indique que le reste des instructions ne doit plus être exécuté.
En l'absence de ce point d'arrêt, la procédure récursive s'appellerait elle-même à l'infini et ne reviendrait jamais au programme appelant.

II - Un exemple pour comprendre

On souhaite écrire une procédure qui affiche un compte à rebours à partir d'un nombre donné en argument. Voyons comment l'écrire de façon itérative, puis de façon récursive :

Façon itérative

```
1 def CompteARebours_iteratif(n):
2     for i in range(n, 0, -1):
3         print(i, "... ",end="")
4     print(" partez !")
```

Façon récursive

```
1 def CompteARebours_recursif(n):
2     if n == 0:
3         print(" partez !")
4     else:
5         print(n, "... ", end="")
6         CompteARebours_recursif(n-1)
```

Voyons ce qui se passe lorsque l'on appelle la procédure `CompteARebours_recursif` avec comme argument le nombre 4 :

```
CompteARebours(n=4)
if 4 == 0:  […]
else :
      print(4, "... ", end="")
      CompteARebours(4-1) ;
      CompteARebours(n=3)
      if 3==0: […]
      else:
            print(3, "... ", end="")
            CompteARebours(3-1) ;
            CompteARebours(n=2)
            if 2==0: […]
            else:
                  print(2, "... ", end="")
                  CompteARebours(2-1) ;
                  CompteARebours(n=1)
                  if 1==0: […]
                  else:
                        print(1, "... ", end="")
                        CompteARebours(1-1) ;
                        CompteARebours(n=0)
                        if 0==0:   # Ici on a atteint la condition d'arrêt
                              print(" partez !")
                        else: […]
```

Dans cet exemple simple, l'avantage de la récursivité n'est pas flagrant : Le programme récursif n'est ni plus simple ni plus court. On pourrait même montrer qu'il n'est pas plus rapide.
En fait la récursivité n'est pas particulièrement adaptée à ce type de problème car il n'est pas intrinsèquement récursif, c'est-à-dire qu'il ne peut pas se décomposer naturellement comme étant une sous-tâche de lui-même.
Voyons maintenant un problème intrinsèquement récursif pour lequel la récursivité va apporter un gain réel.

III – La fonction factorielle : le classique de la récursivité

En mathématiques, la factorielle d'un entier naturel n est le produit des nombres entiers strictement positifs inférieurs ou égaux à n.
Cette opération est notée avec un point d'exclamation : n!
ce qui se lit soit « factorielle de n », soit « factorielle n » soit « n factorielle ».

Cela peut se traduire par la formule suivante :

$$n! = \prod_{k=1}^{n} k$$

Le tableau ci-contre donne l'exemple des premières factorielles.

n	n!
1	1
2	1x2 = 2
3	1x2x3 = 6
4	1x2x3x4 = 24
5	1x2x3x4x5 = 120
6	1x2x3x4x5x6 = 720
7	1x2x3x4x5x6x7 = 5 040
8	1x2x3x4x5x6x7x8 = 40 320
9	1x2x3x4x5x6x7x8x9 = 362 880
10	1x2x3x4x5x6x7x8x9x10 = 3628800
11	1x2x3x…x10x11 = 39916800
12	1x2x3x…x11x12 = 479001600
13	1x2x3x…x12x13 = 6227020800
14	1x2x3x…x13x14 = 87178291200
15	1x2x3x…x14x15 = 1307674368000
16	1x2x3x…x15x16 = 20922789888000
17	1x2x3x…x16x17 = 355687428096000
18	1x2x…x17x18= 6.402.373.705.728.000
19	1x2x…x19 = 121.645.100.408.832.000
20	1x2x…x20=2.43.2902.008.176.640.000

On voit bien ici la récursivité naturelle de la fonction : 9! peut se définir facilement comme 9! = 9 x 8! et 8! = 8 x 7!, 7! = 7 x 6!, …

D'une manière générale n! = n x (n-1)! : la factorielle d'un nombre peut se calculer à partir de la factorielle du nombre précédent. Il n'y a que 1! Qui ne se calcule pas à partir du précédent, mais dont on sait qu'elle vaut 1.

Q1 :
1) Ecrire un programme récursif permettant de calculer la factorielle d'un nombre entier positif quelconque (ne pas essayer avec de trop grandes valeurs). Essayer d'abord tout seul et si besoin aller voir un peu plus bas l'aide pour réaliser le programme.
2) Quel est le point d'arrêt de la fonction factorielle ?
3) Ecrire une version itérative de ce programme et comparer les deux versions.
4) Que se passe-t-il si on appelle la fonction récursive pour n = 3000 ?

Aide :
La fonction factorielle peut se résumer au problème suivant :
Calcul de n ! : si n = 1 alors n! = 1
sinon n! = n x (n-1) !

IV – A vous de jouer !

On souhaite réaliser une liste des matchs pour un tournoi de ping-pong. Chaque joueur doit jouer une fois et une seule contre chaque autre joueur. On fournit une liste des joueurs au programme et il doit afficher la liste de toutes les rencontres.
Exemple :
Entrée : joueurs = ["Anaïs", "Eric", "Miguel", "Linda"]
Sortie : Anaïs / Eric
Anaïs / Miguel
Anaïs / Linda
Eric / Miguel
Eric / Linda
Miguel / Linda

Q2 :
1) Expliquer en quoi le problème est récursif. Comment peut-on le décomposer ? Voir l'aide ci-après si besoin.
2) Ecrire un programme récursif qui réponde au problème.

On pourra utiliser à profit la méthode `copy()`. Cette méthode renvoie une copie de la liste (exemple : `maCopie = listeOriginale.copy()`). La méthode pop(n) peut également être utile car elle supprime l'élément de rang n de la liste :

```
maListe = ["abc", 25, 5.5, "def"]
maListe.pop(2)  # maListe vaut maintenant ["abc", 25, "def"]
maListe.pop(0)  # maListe vaut maintenant [25, "def"]
```

Aides :
Aide pour la question 1) :
On peut remarquer que si on connait la réponse pour une liste de n joueurs et qu'on rajoute un joueur, alors il suffira de le faire jouer contre chacun des n joueurs précédents. Ainsi la réponse au rang (n+1) dépend de la réponse au rang n, ce qui forme une récurrence.
Aide pour la question 2) :
Si on a une liste de n joueurs, on peut écrire tous les matchs du premier joueur contre les (n-1) suivants, puis s'il reste plus de 2 joueurs, retirer ce premier joueur de la liste et recommencer l'opération sur cette liste restreinte. On descend ainsi jusqu'à une liste de deux joueurs, pour laquelle la récursion s'arrête.

V - Conclusion

La récursivité permet de résoudre de façon simple certains problèmes qui peuvent s'exprimer naturellement par récursivité (comme les matchs de ping-pong ou la factorielle). Si ces problèmes peuvent être également programmés de manière itérative, la récursivité offre alors une simplification du problème qui facilite l'écriture, minimise les bugs et améliore la lisibilité (moins de lignes, plus clair).

⇨

⇨

Exemples de problèmes se prêtant bien à une résolution par la récursivité :
– Solveur de grille de sudoku.
– Afficher les étapes pour déplacer les anneaux dans le jeu des tours de Hanoï.
– Déterminer comment placer 8 reines sur un échiquier 8 sur 8 de façon à ce qu'aucune ne soit en prise avec une autre.
– Déterminer un parcours d'un cavalier sur un échiquier passant par chacune des cases de l'échiquier une unique fois.
– Algorithme du tri par fusion, algorithme du tri rapide.
– Représentation de fractales.

Références :
https://fr.wikipedia.org/wiki/Algorithme_r%C3%A9cursif
https://openclassrooms.com/courses/la-recursivite-1
Jeu utilisant la récursivité : https://www-verimag.imag.fr/~wack/CargoBot/

VI – D'autres exemples pour ceux qui ont du mal

1) Multiplication récursive

Soit le programme suivant :

```
1  def multiplie(a, b):
2      if a == 0:
3          return 0
4      else:
5          return b + multiplie(a-1, b)
6
7  a = 7
8  b = 18
9  print(a, "x", b, "=", multiplie(a,b))
```

Q3 :
1) Expliquer comment ce programme décompose l'opération de multiplication.
2) Où se trouve la condition d'arrêt de la récursivité ?

Réponses :
1) On décompose le produit a x *b sous la forme (b + b + b + b + ...)$_{a\ fois}$. Plus précisément, on effectue l'opération : a* x *b = b + (a-1)* x *b puis (a-1)* x *b = b + (a-2)* x *b , ... jusqu'à ce que le terme (a-n) soit nul auquel cas le calcul est terminé.*
2) La condition d'arrêt se trouve à la ligne 2, où on retourne une valeur (0) sans faire de récursion supplémentaire.

2) Puissance récursive

Q4 :
En vous inspirant de l'exemple précédant, écrire un programme récursif qui calcule a^b avec a et b deux entiers positifs.

Exercices - Récursivité

Exercice 1 : Suite de Syracuse

Soit u_n, la suite d'entiers définie par : $u_{n+1} = u_n / 2$ si u_n est pair,
$= 3 \times u_n + 1$ sinon

Avec u_0 un entier quelconque plus grand que 1.

Ecrire une fonction récursive `syracuse(u_n)` qui affiche les valeurs successives de la suite tant que u_n est plus grand que 1.
La conjecture de Syracuse affirme que, quelle que soit la valeur de u_0, il existe toujours un indice n dans la suite tel que $u_n = 1$. Cette conjecture défie toujours les mathématiciens.

`syracuse(31)` doit afficher 13 entiers avant d'arriver à 1. (ici on a donc pris $u_0 = 31$).

Exercice 2 : Comptage

Ecrire une fonction récursive `boucle(i, k)` qui renvoie le tableau des entiers naturels entre i et k. Par exemple `boucle(2,5)` doit renvoyer `[2, 3, 4, 5]`.

Rappel : On peut concaténer des listes en python avec l'opérateur « + » : `[0,1,2]+[3,4]` donne `[0,1,2,3,4]`.

Exercice 3 : Nombre de chiffres

Ecrire une fonction récursive `nombre_de_chiffres(n)` qui prend un entier n strictement positif et renvoie son nombre de chiffres. Par exemple `nombre_de_chiffres(32846)` doit renvoyer 5.

Exercice 4 : Recherche dichotomique

Ecrire une fonction récursive `recherche_dichotomique(tab, i, j, element)` qui prend en argument un tableau d'entiers triés `tab`, deux entiers `i` et `j` compris entre `0` et `len(tab)-1` et un entier `element`. La fonction renvoie l'indice de l'élément s'il se trouve dans le tableau entre les indices `i` et `j` ou `None` s'il n'est pas dans le tableau.

Exercice 5 : Suite de Fibonacci

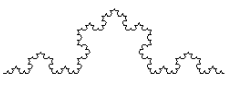
En mathématique la suite de Fibonacci (Mathématicien Italien) est une suite d'entiers définie par :

$$F_0 = 0,\ F_1 = 1 \text{ et } F_n = F_{n-1} + F_{n-2} \text{ pour } n \geq 2$$

1) Programmer une fonction récursive Fibo(n) qui renvoie le terme de rang n de la suite de Fibonacci.
2) Vérifier la correction de votre fonction avec les valeurs des premiers termes de la suite indiqués ci-contre.

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}	F_{16}
0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987

3) Essayez votre fonction sur des valeurs de n élevées (≥ 30). Quelle est d'après vous la classe de complexité de votre algorithme ? D'où vient cette complexité explosive ? Comment pourrait-on revenir à une complexité linéaire ?

Exercice 6 : Flocon de Koch 

Le flocon de Koch (du nom du mathématicien suédois Helge von Koch) est une figure qui s'obtient de manière récursive.
Le cas de base (ordre 0) est un segment de longueur *l*. L'ordre n s'obtient en divisant ce segment en 3 longueurs égales puis en un rectangle équilatéral dont la base est le morceau du milieu, en prenant soin de ne pas dessiner cette base.

Ordre n = 0 Ordre n = 1 Ordre n = 2

Ecrire avec le module `turtle` une fonction `koch(n, l)` qui dessine un flocon de Koch de profondeur `n` à partir d'un segment de longueur `l`.