Récursivité

Les **fonctions récursives** sont des fonctions s'appellent elles-mêmes dans leur définition. Comme l'usage de boucles, la récursivité de fonction permet d'effectuer un nombre d'opérations non connu à l'avance car déterminé par les entrées du programme ; ces deux procédés permettent aussi d'écrire des programmes qui ne se terminent pas.

Les **structures de données récursives** qui contiennent une référence à elles-mêmes dans leur définition. On retrouve ce concept dans la définition d'arbres, de graphes ou de listes chaînées.

Ce chapitre ce concentre sur les fonctions récursives, les structures de données récursives seront abordées dans d'autres chapitres.

L' exemple de fonction récursive suivant est la fonction mathématique factorielle, noté `n!`, qui pour un entier `n` donné calcule le produit de tout les entiers de `1` à `n` : $n! = 1 \times 2 \times 3 \times ... \times n$.

```
# Version récursive

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

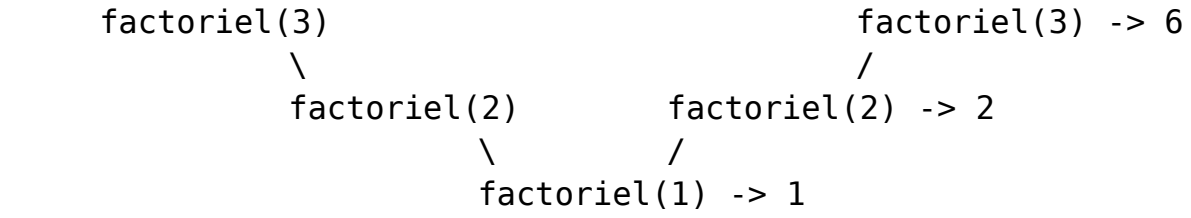
```
# Version itérative (boucle)

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

Tout programme récursif peut être transformé en un programme impératif et réciproquement.

Une fonction récursive inclut toujours une structure conditionnelle afin que les appels récursif puisse s'arrêter. On y définit le ou les cas simples pour lesquels sont connues les valeurs de cette fonction et pour lesquels les appels récursifs ne sont pas nécessaire.

```
def fonction_recursive(...):
    if ... :
        return ...      # cas simple, pas de récursion
    else :
        return ... fonction_recursive(...)  # appel récursif
```

On appelle **arbre d'appels** la représentation des différents appels de fonctions lors de l'exécution d'un programme. L'arbre d'appels suivant représente l'appel à `factoriel(3)`.



L'arbre d'appels d'une fonction récursive possède toujours une phase descendante suivie d'une phase ascendante.
Le site suivant permet la visualisation d'arbre d'appels https://pythontutor.com/render.html#mode=display .

Exercice 1 Predict, Run, Investigate

Sans exécuter, qu'affiche les codes suivants ? Vérifier en exécutant le code.

```
Def myst1(n):
    if n == 0:
        return 0
    else:
        return myst1(n-1) + 2

print(myst1(3))
```

```
Def myst2(n):
    if n == 0:
        return 1
    else:
        return myst2(n-1) * 2

print(myst2(4))
```

Exercice 2 Modify

1) Modifier la fonction `myst1` de l'exercice précédent pour écrire la fonction `somme(n)` qui calcule la somme des entiers de `1` à `n`. `assert somme(10) == 55`

2) Modifier la fonction `myst1` de l'exercice précédent pour écrire la fonction `produit(a, b)` qui calcule le produit de `a` par `b`. `assert produit(123, 456) == 123*456`

3) Modifier la fonction `myst2` de l'exercice précédent pour écrire la fonction `puissance(n, exp)` qui calcule `n` à la puissance `exp`. `assert puissance(123, 4) == 123**4`

Exercice 3 : Make

1) Suites

a) Écrire la fonction récursive `arithmetique(n)` qui calcule u_n avec $u_{n+1}=u_n+5$ et $u_0=2$.

b) Écrire la fonction récursive `geometrique(n)` qui calcule u_n avec $u_{n+1}=u_n\times 6$ et $u_0=\frac{1}{3}$.

c) Écrire la fonction récursive `arithmetico_geometrique(n)` qui calcule u_n avec $u_{n+1}=3u_n+2$ et $u_0=\frac{4}{3}$.

2) Suite de Syracuse

La suite de Syracuse d'un nombre entier $N > 0$ est définie par récurrence, de la manière suivante :

- $u_0 = N$
- et pour tout entier naturel n : $u_{n+1} = \begin{cases} \frac{u_n}{2} & \text{si } u_n \text{ est pair,} \\ 3u_n + 1 & \text{si } u_n \text{ est impair.} \end{cases}$

La conjecture affirme que pour tout entier $N > 0$, il existe un indice n tel que $u_n = 1$.

Suite de Syracuse pour *N* = 15

u_0	u_1	u_2	u_3	u_4	u_5	u_6	u_7	u_8	u_9	u_{10}	u_{11}	u_{12}	u_{13}	u_{14}	u_{15}	u_{16}	u_{17}	u_{18}	u_{19}	u_{20}	
15	46	23	70	35	106	53	**160**	80	40	**20**	**10**	5	16	8	4	2	**1**	4	2	**1**	…

Écrire la fonction récursive `Syracuse(n)` affiche les termes de la suite de Syracuse jusqu'à l'obtention d'un 1.

3) Exponentiation rapide

L'exponentiation rapide est un algorithme utilisé pour calculer rapidement de grandes puissances entières.

Soit n un entier strictement supérieur à 1, supposons que l'on sache calculer, pour chaque réel x, toutes les puissances x^k de x, pour tout k, tel que $1 \leq k < n$.

- Si n est pair alors $x^n = (x^2)^{n/2}$. Il suffit alors de calculer $y^{n/2}$ pour $y = x^2$.
- Si n est impair et $n > 1$, alors $x^n = x(x^2)^{(n-1)/2}$. Il suffit de calculer $y^{(n-1)/2}$ pour $y = x^2$ et de multiplier le résultat par x.

Cette remarque nous amène à l'algorithme récursif suivant qui calcule x^n pour un entier strictement positif n :

$$\text{puissance}(x, n) = \begin{cases} x, & \text{si } n = 1 \\ \text{puissance}(x^2, n/2), & \text{si } n \text{ est pair} \\ x \times \text{puissance}(x^2, (n-1)/2), & \text{si } n \text{ est impair} \end{cases}$$

En comparant à la méthode ordinaire qui consiste à multiplier x par lui-même $n - 1$ fois, cet algorithme nécessite de l'ordre de $O(\log n)$ multiplications et ainsi accélère le calcul de x^n de façon spectaculaire pour les grands entiers.

Écrire la fonction récursive `exp(x, n)` qui calcule x^n en utilisant la méthode de l'exponentiation rapide.

4) Génération de liste

Écrire la fonction récursive `gen_liste(n)` prenant en paramètre un entier positif `n` qui génère la liste `[n,n-1, . . .,1,0]`.

5) Récursivité croisée

La définition précédente des algorithmes récursifs ne couvre pas les cas des algorithmes mutuellement récursifs. L'exemple classique suivant permet de savoir si un nombre est pair ou impaire. La fonction pair appelle la fonction impaire et vice versa.
Complétez le code suivant et testez le.

```
def pair(n):
    if n == 0:
        return ...
    else:
        return impair(...)

def impair(n):
    if n == 0:
        return ...
    else:
        return ...
```

6) Récursivité multiple – Suite de Fibonacci

1 - Écrire la fonction récursive `Fibonacci(n)` qui calcule F_n le n-ième terme de la suite de Fibonacci définie par $F_{n+2}=F_{n+1}+F_n$ avec $F_1=F_0=1$
2 - Dessiner l'arbre d'appels de `Fibonacci(5)`. Que constatez vous ?

7) Récursivité multiple – Triangle de pascal (coefficients binomiaux)

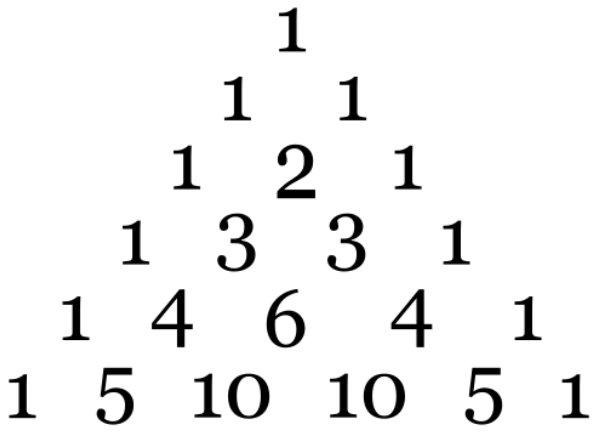
En mathématiques, le triangle de Pascal est une présentation des coefficients binomiaux dans un tableau triangulaire. Il a été nommé ainsi en l'honneur du mathématicien français Blaise Pascal.
La construction du triangle est régie par la relation de Pascal :

$$\binom{n}{k}=\binom{n-1}{k-1}+\binom{n-1}{k} \text{ avec } \binom{n}{0}=\binom{n}{n}=1$$

Les coefficients $\binom{n}{k}$ pour $0 \leq k \leq n$ figurent à la ligne d'indice n .

Le sommet de la pyramide correspond à $\binom{0}{0}$ et $\binom{5}{2}=\binom{5}{3}=10$.

$\binom{n}{0}$ et $\binom{n}{n}$ correspondent aux bords de la pyramide.



Écrire la fonction récursive `Pascal(n, k)` qui calcule la valeur du coefficient $\binom{n}{k}$.

8) Hors programme - Récursivité terminale

Une fonction à récursivité terminale est une fonction où l'appel récursif est la dernière instruction à être évaluée. Cette instruction est alors nécessairement « pure », c'est-à-dire qu'elle consiste en un simple appel à la fonction, et jamais à un calcul ou une composition. Par exemple, dans un langage de programmation fictif :

```
def récursionTerminale(n) :
     ...
     return récursionTerminale(n - 1)

def récursionNonTerminale(n) :
     ...
     return n + récursionNonTerminale(n - 1)
```

`récursionNonTerminale()` n'est pas une récursion terminale car sa dernière instruction est une composition faisant intervenir l'appel récursif. Dans le premier cas, aucune référence aux résultats précédents n'est à conserver en mémoire, tandis que dans le second cas, tous les résultats intermédiaires doivent l'être. Les algorithmes récursifs exprimés à l'aide de fonctions à récursion terminale profitent donc d'une optimisation de la pile d'exécution.

```
def factoriel_terminal(n, acc = 1):
     if n <= 1:
          return acc
     else:
          return factoriel_terminal(n - 1, acc * n)
```

Les différences entre la version récursive et la version récursive terminale :
- On utilise un paramètre `acc` (accumulateur) pour stoquer les résultats successifs des appels de fonction.
- La dernière instruction effectuée est toujours un simple appel de fonction.
- A la fin de la récursion, on retourne l'accumulateur.

Reprendre vos réponses de l'exercice 2 et réécrire ces fonctions avec de la récursivité terminal.

9) Bonus : Tribonacci, programmation dynamique, mémoïsation

On définit la suite de Tribonacci par la relation de récurrence : $u_0 = u_1 = 0$, $u_2 = 1$ et pour tout entier n :

$$u_{n+3} = u_{n+2} + u_{n+1} + u_n$$

1. Écrire une fonction récursive `tribonacci(n)` qui renvoie le n-ième terme de la suite de Tribonacci. Quel inconvénient y-a-t-il à procéder ainsi ?
2. Proposer une version itérative de la fonction précédente. `tribonacci_iter(n)`.
3. Proposer une version récursive efficace en utilisant un dictionnaire qui stocke les valeurs déjà calculées. On nommera la fonction `tribonacci_memo(n)`.