Introduction à la récursivité

Si on veut créer une fonction qui renvoie la factorielle de n, on peut procéder de façon itérative :

```
def Factorielle(n):
    p=1
    for i in range(1,n+1): #i va varier entre 1 et n
        p=p*i
    return p
```

Cependant, on peut aussi procéder par **récursivité**, en effet, on sait que $n! = n \times (n-1)!$. Donc pour calculer $n!$ il suffit de calculer $(n-1)!$ et de multiplier par n. Mais comment calculer $(n-1)!$ alors ? Et bien justement grâce à la fonction factorielle que l'on est précisément en train de créer :

```
def Factor(n):
    if n==0:
        return 1#ce que fait le programme si la condition
        #n==0 est vérifiée
    else:#ce que fait le programme si la condition n==0
        #n'est pas vérifiée
        return Factor(n-1)*n
```

Détaillons un peu le fonctionnement. Si on utilise la commande `Factor(5)` Que va-t'il se passer ?

- Le programme va donc calculer `Factor(4)*5`.
- Mais comme il ne connaît pas la valeur de `Factor(4)`, il va se mettre à la calculer. Que vaut `Factor(4)` ? De toute évidence, `Factor(3)*4`.
- Que vaut `Factor(3)` ? `Factor(2)*3`.
- Que vaut `Factor(2)` ? `Factor(1)*2`
- Que Vaut `Factor(1)` ? `Factor(0)*1`.
- Que vaut `Factor(0)` ? 1 (car on a indiqué la valeur de `0!` dans le `if n==0:`.

Ainsi, on a une pile de calculs à faire et chaque calcul dépend d'un autre calcul à effectuer. On va donc déplier la pile dans l'autre sens : cela permet de trouver que `Factor(1)` vaut 1, puis que `Factor(2)` vaut 2, puis que `Factor(3)` vaut 6 puis que `Factor(4)` vaut 24 puis que `Factor(5)` vaut 120.

On voit donc que Python est capable de gérer tout ça tout seul. Pour cela il suffit de deux choses : donner la première valeur de la factorielle et la relation de récurrence.

Définition de la récursivité

Ce principe où une fonction pour renvoyer un résultat a besoin de s'appeler elle-même avec d'autres paramètres s'appelle la **récursivité**.

Attention à ne pas oublier l'initialisation

Pour qu'une fonction **récursive** fonctionne correctement, il faut toujours penser à indiquer une ou plusieurs valeurs dont Python aura besoin à un moment. Sinon, la pile va devenir infini, en effet si on écrivait

```
def Factor(n):
  return Factor(n-1)*n
```

Alors, arrivé à $n = 0$, le programme se mettra en quête de calculer `Factor(-1)` puis `Factor(-2)` etc. créant une infinité de calculs.

Dans la suite de ce TP, il faudra faire la même chose, réfléchir à chaque fois à la relation de récurrence et donner une valeur d'initialisation.

Suites récurrentes en mathématiques

Exercice 1. Soit $u_0 = 5$ on pose pour tout $n \in \mathbb{N}$, $u_{n+1} = 1 - \dfrac{1}{2 + u_n}$.

1. Écrire une fonction **récursive** `U(n)` qui renvoie le terme u_n de la suite $(u_n)_n$.
2. Tester si `U(100)` vaut 0.6180339887498949
3. Tenter de calculer `U(10**4)`, que renvoie Python ?

La question précédente met en évidence la limite de la pile d'exécution pour un algorithme récursif, c'est-à-dire le nombre maximal d'appels récursifs autorisé par Python (ce nombre peut être modifié).

4. À l'aide des commandes suivantes, déterminer la limite de votre pile d'exécution.

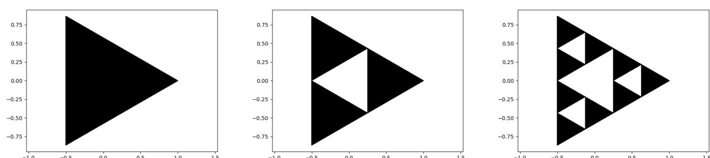
```python
import sys#À mettre en début de fichier
print(sys.getrecursionlimit())
```

Exercice 2.
1. Rappeler la formule du triangle de Pascal.
2. Grâce à cette formule, écrire une fonction **récursive** `Binomial(n,p)` qui renvoie $\binom{n}{p}$. Pour l'initialisation, traiter à part les cas $\binom{n}{0}$ et $\binom{n}{n}$.
3. Calculer à la main $\binom{10}{3}$ et comparer avec la fonction `Binomial`.
4. Cela est-il satisfaisant pour `Binomial(28,11)` ?

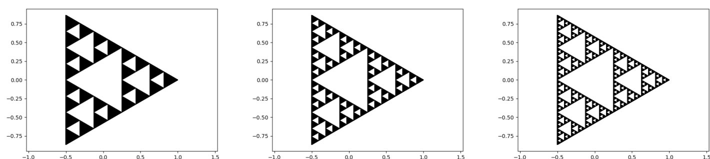
Récursivité en Python (diviser pour mieux régner)

Exercice 3 (⋆). Créer une fonction récursive `Maximum(L)` qui va renvoyer le maximum d'une liste `L` (non vide). Pour cela, traiter le cas à part où `L` a un élément. Puis sinon, créer une liste `M` qui est la liste `L` sans son dernier élément, ensuite, calculer le maximum de `M` puis comparer ce maximum au dernier élément retiré.

Exercice 4 (⋆⋆ 𝄞). Le but de cet exercice est de dessiner une fractale appelée fractale de Sierpinski. Pour cela, le code suivant nous permet d'afficher un triangle de sommets A, B et C avec une couleur de remplissage. Chaque point sera vu comme une liste de deux éléments : le premier son abscisse et le second son ordonnée.



(a) À l'étape 0 (b) À l'étape 1 (c) À l'étape 2



(d) À l'étape 3 (e) À l'étape 4 (f) À l'étape 5

```python
import matplotlib.pyplot as plt#Chargement de la
#bibliothèque pour les graphes, à mettre en début de fichier

def Triangle(A,B,C):
    plt.fill([A[0],B[0],C[0]],[A[1],B[1],C[1]],color="black")
    plt.axis('equal')
```

Grâce au code suivant, on va afficher le triangle à l'étape 0

```python
A=[1,0]#A, B et C sont les trois points d'un triangle équilatéral
B=[-0.5,3**(0.5)/2]#Notez que les affixes de ces points sont les
C=[-0.5,-3**(0.5)/2]#racines troisièmes de 1 (à réviser): 1,j,j*j
Triangle(A,B,C)
plt.show()
```

Écrire une fonction `Sierpinski(A,B,C,n)` **récursive** qui affiche le triangle de Sierpenski à l'étape n dans un triangle `ABC`. Pour cela, étant donné le triangle `ABC`, créer les milieux de ce triangle `I`, `J` et `K`. Remarquez qu'il faut que le petit triangle `IJK` soit blanc. De plus, il y a trois autres petits triangles, ces trois triangles doivent être coloriés grâce à la fonction Sierpinski à l'étape $n-1$.

Exercice 5 (𝄞⋆). Écrire une fonction **récursive** `EstDedansDichotomie(x,L)` qui renvoie `True` si l'élément `x` est dans la liste `L` que l'on suppose triée par ordre croissant. Pour cela, on coupe la liste en deux et on regarde dans quelle des deux sous-liste `x` devrait se trouver s'il était dans la liste. On traitera à part le cas d'une liste à un élément.

Exercice 6 (⋆⋆). Regardez le fichier `TP7Flocon` qui est sur CDP et programmez cette courbe de façon **récursive**. La création du gif est en bonus (cf. corrigé TP6).