TP-4 : Récursivité

1 Dessin du flocon de Von Koch

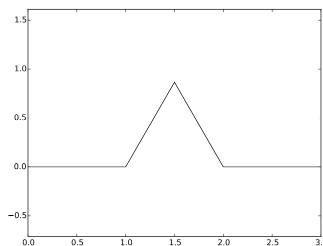
1.1 Le type complex en Python

Pour travailler en Python avec les nombres complexes, on importe le module `cmath`. En entrant dans la console les instructions suivantes, on découvre les principales fonctions qui nous intéressent.

```
import cmath as cp
z = complex(1,2) # on rentre le couple (partie réelle, partie imaginaire)
z                # représentation à l'écran
z.real           # pour récupérer la partie réelle
z.imag           # pour récupérer la partie imaginaire
type(z)
(1+1j).conjugate() # le conjugué
(1+2j)+(2-3j)      # l'addition
(1+2j)*(2-3j)      # le produit
(1+2j)/(2-3j)      # le quotient
(1+2j)**7, (2+1j)**-5 # puissances
abs(3-4j)             # le module
cp.polar(1-1j)         # le module et un argument
cp.rect(1,2*cp.pi/3)     # le complexe de module 1 et d'argument 2*pi/3
```

1.2 Un motif

On sait que dans le plan complexe l'image d'un point M d'affixe z par une rotation de centre Ω d'affixe ω et d'angle θ est le point M' d'affixe z' telle que $z' - \omega = \mathrm{e}^{\mathrm{i}\,\theta}\,(z - \omega)$, c'est-à-dire $z' = \omega + \mathrm{e}^{\mathrm{i}\,\theta}\,(z - \omega)$. Ici on part d'un segment $[A, B]$ donné par les affixes (complexes) a et b de A et B, on divise ce segment en trois segments de même longueur et on remplace le segment du milieu par les deux autres côtés d'un triangle équilatéral construit sur ce segment, pour obtenir le motif suivant (pour $a = 0$ et $b = 3$)



On utilise le module `pyplot` de la bibliothèque `matplotlib` et on précise (rappelle) que le tracé d'un segment d'extrémités $A1 = (x1, y1)$ et $A2 = (x2, y2)$ s'obtient par

`plt.plot([x1,x2], [y1,y2],'k')`, avec `'k'` pour noir (black)

et le tracé d'une ligne brisée $A1A2...AN$, avec $Ai = (xi, yi)$, s'obtient avec la commande

`plt.plot([x1,x2,...,xN],[y1,y2,...,yN],'r')` (en rouge).

On crée d'abord une fonction qui réalise une rotation plane connaissant son centre et son angle. Avec un angle de $\frac{\pi}{3}$, A et B étant donnés, on construit ainsi un point C tel que ABC soit un triangle équilatéral qui tourne dans le sens trigonométrique positif.

```
import matplotlib.pyplot as plt
def rot(centre, angle, z):
    a = cp.rect(1, angle)
    return centre + a * (z-centre)
def motif(a,b): # a et b sont les affixes complexes des extrémités
    L = [a, b]
    L.insert(1,a+(b-a)/3)    # le premier point intermédiaire
    L.insert(2,a+2*(b-a)/3) # le deuxième point intermédiaire
    L.insert(2,rot(L[1], cp.pi/3,L[2] )) # le troisième sommet du triangle
    X = [z.real for z in L] # la liste des abscisses des cinq points
    Y = [z.imag for z in L] # la liste des ordonnées
    plt.figure()
    plt.axis('equal') # pour avoir un repère orthonormé
    plt.plot(X, Y, 'k')
    plt.show()
```

On peut voir et comprendre avec `motif(0,3)`, `motif(-1+2j, 3-3j)`, `motif(3-3j,-1+2j)`

1.3 Récursivité du motif

La figure actuelle est une ligne brisée constituée de quatre segments et sur chacun de ces quatre segments on peut recommencer la manipulation précédente et ainsi de suite, autant de fois qu'on le veut, sur chacun des segments ainsi construits au fur et à mesure. On pourrait faire cela de manière itérative mais on va procéder ici de manière récursive. On crée tout d'abord une fonction `tfVK` qui crée le motif sur tout segment (sans le représenter à l'écran, il s'agit juste de construire la liste des points), puis la fonction récursive `VKrec` qui construit la ligne brisée voulue et enfin la fonction dessin qui donne la représentation graphique de cette ligne brisée.

```
def tfVK(a,b): # transformation de Von Kock
    c = a + (b-a)/3
    d = a + 2*(b-a)/3
    e = rot(c, cp.pi/3, d)
    return a, c, e, d, b
def VKrec(L, n): # fonction récursive
    if n == 0:
        return L
    L1 = [L[0]] # après un return le else est redondant
    for i in range(1, len(L)):
        a, b = L[i-1], L[i]
        u, v, w, x, y = tfVK(a, b)
        L1 += [v, w, x, y]
    return VKrec(L1, n-1)
def dessin(L, n):
    T = VKrec(L,n)
    X = [z.real for z in T]
    Y = [z.imag for z in T]
    plt.figure()
    plt.axis('equal')
    plt.plot(X, Y, 'r')
    plt.show()
```

On pourra tester avec `dessin([0,3],0)`, `dessin([0,3],1)`, `dessin([0,3],2)` et `dessin([0,3],4)`

1.4 Un flocon

On peut maintenant faire le flocon de Von Koch à l'ordre n que l'on veut en partant d'un triangle équilatéral, mais il faut alors tourner de $-\frac{\pi}{3}$ (vers l'extérieur du triangle équilatéral, dans le sens trigonométrique négatif).

```
def tfVK(a,b):
    c = a + (b-a)/3
    d = a + 2*(b-a)/3
    e = rot(c, -cp.pi/3, d) # on tourne vers l'extérieur
    return a, c, e, d, b
def flocon(a, b, n):
    L = [a, b, rot(a, cp.pi/3, b), a]
    dessin(L,n)
```

que l'on pourra tester avec `flocon(0,3,5)`, `flocon(5+3j, -1+1j, 4)`.

Quelles valeurs raisonnables de n peut-on demander ? Expliquez !

2 Énumeration des permutations d'une liste

On prend une liste, par exemple `L=[1,2,3]`, et on voudrait donner la liste de toutes les « permutations » que l'on peut faire, c'est-à-dire obtenir, par exemple (avec mon classement personnel dans ce cas particulier) : `[[1, 2, 3], [2, 3, 1], [3, 1, 2], [1, 3, 2], [3, 2, 1], [2, 1, 3]]`.

On crée ici un algorithme récursif qui fait cela. Pour une liste qui ne contient qu'un élément, il n'y a rien à faire : si `L=[1]` alors on obtient `[[1]]` : tout les éléments sont fixes, nous tenons là le cas de base de notre récursivité. Pour créer la récursivité, on remarque que si on sait le faire quand on laisse fixe le premier, alors on saura le faire pour tous les autres, car il suffira d'échanger le `L[1]` avec le `L[i]` pour $i > 1$, puis de faire ce que l'on sait faire lorsque le premier est fixe et enfin de ré-échanger `L[1]` avec le `L[i]` pour remettre les choses en place. Or, si on laisse fixe le premier élément, alors permuter la liste revient à permuter la sous-liste `L[1:]`, de longueur `len(L)-1`, d'où l'amorce de la récursivité. Cette récursivité se terminera lorsque tous les points seront fixes, c'est-à-dire lorsqu'il suffit de rajouter à la liste des résultats déjà obtenus la liste (permutée) que l'on vient de créer avec les appels successifs.

Pour l'implémentation de cet algorithme en Python, on commence par créer une fonction récursive `permute(L, i, res):` qui permute les éléments de la liste `L` en laissant fixes les éléments d'indices strictement inférieurs à `i` et ajoute à la liste `res` des permutations déjà obtenues la nouvelle permutation ainsi créée. On remarquera que lorsque `i == len(L)-1`, on se contente d'ajouter à `res` la liste `L`.

Ensuite, dans une fonction `permutations(L):` on initialise la liste attendue des permutations à `pmt = []`, puis on lance `permute(L, 0, pmt)` et on retourne `pmt` en fin du processus récursif.

On pourra essayer d'implémenter tout cela tout seul, soit de la manière suggérée ci-dessus soit d'une autre manière mais alors il faudra expliquer comment on procède. À défaut, après avoir compris comment cela fonctionne, on pourra compléter le cadre suivant .

```
def permute(L, i, res):
    '''permute les éléments de la liste L en laissant fixes ceux d'indices
    strictement inférieurs à i et retourne les résultats dans la liste res'''
    if i == len(L)-1:
        res.append(list(L))
    for j in range(i, len(L)):
        L[j], L[i] = L[i], L[j]
            ...
        L[j], L[i] = L[i], L[j]
def permutations(L):
    pmt = []
    ...
    return pmt
```

Que l'on testera avec `permutations([1,2,3])` (on pourra comparer avec l'ordre des permutations donné ci-dessus), puis `permutations([1,2,3,4])`, `permutations([1])`, `permutations([])`, `permutations([1,2,1])`, `permutations(['a', 1, [0,1]])`.

3 Énumeration des sous-listes d'une liste

Écrire une fonction récursive qui pour une liste `L` retourne la liste de toutes les sous-listes de `L`, par exemple, pour `L=[1, 2, 3]` on retournera (pas forcément dans cet ordre)

`[[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]`

(on respecte l'ordre des éléments dans la liste initiale).

On pourra remarquer que dans la liste retournée ici, on commence d'abord par les quatre sous-listes qui ne contiennent pas le dernier élément le 3, puis ensuite on a les quatre premières (dans le même ordre) augmentées du 3. Dans le même ordre d'idée, les deux premières sont les sous-suites de $[1, 2]$ qui ne contiennent pas le dernier élément, le 2, et qu'ensuite on a les deux premières (dans le même ordre) augmentées du 2. Enfin la première est l'unique sous-liste qui ne contient aucun élément de la liste, même pas le premier, le 1, et que la deuxième sous-liste est la première augmentée du premier élément, le 1. Ceci peut permettre d'implémenter une récursivité, en partant de la fin de la liste et en prenant comme cas de base, le cas où la liste est vide auquel cas il n'y a qu'une seule sous-liste, c'est la liste elle-même.

On n'oubliera pas de tester avec des exemples pertinents et on pourra se poser les questions de la terminaison, de la conformité à la spécification et de la complexité (asymptotique).

Si après quelques (de gros) efforts on ne voit pas comment procéder, on pourra travailler sur ce qui suit : taper, faire tourner et comprendre et, éventuellement, améliorer, en tout cas se fabriquer sa propre version, celle que l'on comprend, que l'on maîtrise

```
def sous-listes(L, res):
    n = len(L)
    if n == 0:
        res.append([])
    else:
        ssL = L[:n-1]
        sous-listes(ssL, res)
        for i in range(len(res)):
            res.append(res[i] + [L[n-1]])

def ens-sous-listes(L):
    res = []
    sous-listes(L, res)
    return res
```