TP 4 : Structures répétitives et conditionnelles (2)

Dans ce TP, on reverra les structures itératives et conditionnelles dans des cadres différents.

Exercice 1 *Avant de commencer (pour rappel)*

Démarrer le logiciel Spyder puis, dans le menu « Fichier » créer un nouveau fichier puis le sauvegarder avec un nom sous la forme TP4_votrenom. Dans la suite, on sauvegardera régulièrement son travail.

I PGCD

Exercice 2 *Manipulation d'ensembles*

Dans cet exercice, on travaillera directement dans la console.

1. Définir deux listes `L1 = [4,5,8,9,7]` et `L2 = [4,1,9,5]` puis taper `S1 = set(L1)` et `S2 = set(L2)`. Quel est le type de `S1` et `S2` ?
2. Tester les commandes
   (a) `S1.union(S2)` ;
   (b) `S1.intersection(S2)` ;
   (c) `list(S1.difference(S2))`.
   Expliquer chacune de ces commandes.

Exercice 3 *PGCD*

1. Créer une fonction secondaire `diviseurs(n)` qui, pour un entier n donné, renvoie la liste de ses diviseurs positifs. Cette fonction a déjà été faite en cours.
2. À l'aide de l'exercice précédent et de la question précédente, créer une fonction `diviseurs_communs(a,b)` qui renvoie la liste des diviseurs communs à a et b où a, b sont des entiers naturels.
3. Créer finalement une fonction `PGCD(a,b)` qui renvoie le plus grand diviseur commun à a et b.
4. Modifier la fonction précédente pour créer une documentation pour la fonction `PGCD`.
5. *Question facultative :* Créer une fonction `PPCM(a,b)` qui renvoie le plus petit multiple commun à a et b.

II Des suites

Exercice 4 *Série divergente*

1. Créer une fonction Python `S(n)` qui retourne $S_n = \sum_{k=1}^{n} \frac{1}{k}$.
2. Créer une fonction Python `plus_petit_élément(M)` qui retourne le plus petit entier n tel que $S_n > M$. Tester la fonction avec quelques valeurs de $M \leq 10$.
3. Ajouter dans la fonction `plus_petit_élément` une limitation du nombre d'itération à 10000. Dans le cas contraire, la fonction affichera le message `'nombre d iterations trop grand'`.
4. Créer une fonction `repres(n)` qui représente graphiquement les termes de la suite $\big(S_n/\ln(n)\big)$. On pourra utiliser l'option `'o'` dans les options de `pypl.plot`
5. Tester la fonction précédente pour $n = 1000$. Expliquer le graphique obtenu.

Exercice 5 *Suite de Syracuse*

On définit la suite de Syracuse associée à l'entier N par :

$$u_0 = N \text{ et } \forall n \in \mathbb{N},\ u_{n+1} = \begin{cases} \frac{u_n}{2} & \text{si } u_n \text{ est pair} \\ \frac{3u_n+1}{2} & \text{sinon} \end{cases}$$

1. Écrire une fonction `syracuse(n,N)` qui pour un couple (N, n) donné, renvoie une liste dans contenant les valeurs $u_0, \ldots, u_n$ avec $u_0 = N$.
2. Calculer `syracuse(15,50)` et `syracuse(127,50)`.
3. On appelle temps de vol le plus petit indice p tel que $u_p = 1$. Écrire une fonction `syracuse2(N)` qui pour un entier N donné qui renvoie le temps de vol obtenu.
4. Déterminer le temps de vol pour $N = 15$ et $N = 222$.

Exercice 6 *Évaluation du temps de calcul*

1. Importer le package `time`.
2. Pour déterminer le temps de calcul nécessaire à l'exécution de la fonction `facto`, on peut écrire

```
def facto(n):
    T1 = time.time()
    P=1
    for i in range(1,n+1):
         P = P*i
    T2 = time.time()
    return T2 - T1
```

   À noter qu'ici, on ne retourne pas le résultat de la fonction mais seulement le temps de calcul. Représenter le temps de calcul sur un graphique pour la fonction `facto` pour $n = 0, \ldots, 1000$.
3. Représenter le temps de calcul sur un graphique pour la fonction `plus_petit_element` pour M allant de 1 à 9 et 100 points.

III Recherche dans une liste par dichotomie

Exercice 7 *Recherche dans une liste de façon linéaire*

1. Chercher dans l'aide l'utilité de la fonction `randint`.
2. Avec la question précédente, créer une fonction Python `liste_alea(n,m)` qui renvoie une liste de taille n d'entiers naturels inférieurs ou égaux à m.
3. Écrire une fonction Python `presence(k,L)` qui retourne `True` si k est dans la liste L et `False` sinon.
4. Écrire une fonction Python `positions(k,L)` qui pour une liste d'entiers renvoie les positions de k dans cette liste. De façon plus précise, cette fonction doit retourner :

```
>>> positions(42, [12, 17, 42, 5])
[2]
>>> positions(24, [12, 17, 42, 5])
[ ]
>>> positions(42, [42, 12, 17, 42, 5])
[0, 3]
```

Exercice 8 *Recherche par dichotomie*

On souhaite réécrire la fonction `presence` lorsque la liste est triée.

1. Lorsque `L` est une liste, quelle est l'action de `L.sort()` ?
2. Avec la question précédente, créer une fonction Python `liste_alea_triee(n,m)` qui renvoie une liste de taille n d'entiers inférieurs ou égaux à m **triés dans l'ordre croissant**.
3. Écrire une fonction Python `presence_dicho(k,L)` qui retourne `True` si k est dans la liste L **triée dans l'ordre croissant** et `False` sinon. On utilisera la stratégie suivante :

(a) on compare k avec l'élément au milieu de la liste (si la longueur de la liste est paire, on choisi un des éléments centraux) ;

(b) si k est strictement plus grand que cet élément, on sait qu'il est dans la partie « droite » de la liste et dans le cas contraire, celui-ci est dans la partie « gauche » de la liste.

On conseille au fur et à mesure de l'exécution de la fonction de conserver une liste `[a,b]` où $[\![a, b]\!]$ donne un encadrement de l'endroit où de trouve potentiellement l'élément cherché. On pourra utiliser `(b+a)//2` pour avoir le « milieu » de la liste.

IV Compléments

Exercice 9 *Complexité des méthodes précédentes*

Comparer les temps de calcul des fonctions de `presence` et `presence_dicho` pour des listes triées de taille n de plus en plus grande. Quelle est la méthode la plus efficace ?

Exercice 10 *Flocon de Von Koch*

On introduit d'abord un outil mathématique. On appelle similitudes directe du plan complexe les composées d'homothéties, de rotations et de translations. Si M est un point du plan d'affixe z, son image par une similitude a pour affixe

$$f(z) = az + b$$

où $a, b \in \mathbb{C}$ avec $a \neq 0$ caractérisent la similitude de la façon suivante :

1. si $a = 1$, la fonction f représente la translation de vecteur d'affixe b ;
2. si $a \notin \{0, 1\}$, en posant $a = \rho e^{i\theta}$ et en notant ω l'unique complexe qui vérifie $f(\omega) = \omega$), la similitude représentée par f est la composée d'une rotation de centre Ω d'affixe ω et d'angle θ et d'une homothétie de rapport ρ.

On note :

- f_1 la similitude de centre O de coordonnées $(0, 0)$, de rapport $\frac{1}{3}$ et d'angle 0 ;
- f_2 la similitude qui transforme O en B de coordonnées $\left(\frac{1}{3}, 0\right)$ et A de coordonnées $(1, 0)$ en C de coordonnées $\left(\frac{1}{2}, \frac{1}{2\sqrt{3}}\right)$;
- f_3 la similitude qui transforme O en C et A en D de coordonnées $\left(\frac{2}{3}, 0\right)$;
- f_1 la similitude de centre A de coordonnées $(0, 0)$, de rapport $\frac{1}{3}$ et d'angle 0.

1. Calculer les fonctions f_1, f_2, f_3 et f_4.
2. Charger le module `cmath` et définir les fonctions précédentes en Python.
3. Définir une fonction `successeur(L)` où, pour une liste de nombre complexe `L = [a1, ...,an]` renvoie `[f_1(a1), ...,f_1(an),f_2(a1), ...,f_2(an),f_3(a1), ...,f_3(an),f_4(a1), ...,f_4(an)]`.
4. Créer une fonction Python `iteration(n)` qui, à partir de la liste `[0,1]` effectue n fois l'étape précédente.
5. Écrire une fonction Python `fractale(n)` qui représente une ligne polygonale qui joint les points de la liste précédente.
6. Écrire finalement une fonction Python `flocon(n)` qui permet d'obtenir la figure ci-contre :