TP7 : Manipulations de listes

Démarrer le logiciel Spyder puis, dans le menu « Fichier » créer un nouveau fichier puis le sauvegarder avec un nom sous la forme TP7_votrenom. Dans la suite, on sauvegardera régulièrement son travail.

I Développements limités

Exercice 1 *xlim,ylim*

Charger les modules `numpy` et `matplotlib.pypl` sous les noms `np` et `pypl` puis tester le code suivant :

```
def graph():
     abscisse = np.linspace(0, 2*pi, 30)
     ordonnee = [cos(x) for x in abscisse]
     pypl.xlim(-1,5)
     pypl.ylim(-4,8)
     return pypl.plot(abscisse,ordonnee)
```

Expliquer le fonctionnement de `xlim` et `ylim`.

Exercice 2 *Polynômes et graphiques*

1. Écrire une fonction Python `evalf_poly(a,b,c,x)` qui retourne, pour des flottants a, b, c, x :
$$P(x) = ax^2 + bx + c.$$
2. Écrire une fonction Python `graph_poly(n)` qui trace dans une fenêtre où les abscisses varient de -10 à 10 et les ordonnées de -10 à 10 les graphiques des fonction
$$P_k : x \mapsto \frac{1}{k}x^2 + 2x + 1$$
pour k variant de 1 à n.
3. Écrire une fonction Python `evaluation(L,x)` qui à partir d'une liste $[a_0, a_1, \ldots, a_n]$ renvoie
$$P(x) = a_0 + a_1 x + \cdots + a_n x^n.$$

Exercice 3 *Approximation de la fonction* sin

1. Rappeler le développement limité de la fonction sin à l'ordre n en 0.
2. Écrire une fonction Python `coeff_DL(n)` qui renvoie une liste avec les coefficients du développement limité de la fonction sin à l'ordre n en 0. Par exemple, pour $n = 3$, cette fonction retourne
`[1,0,-0.166666667]`
car les premiers coefficients du développements limité de sin son 1, 0 et $-\frac{1}{3!} \simeq -0.16666667$. On pourra commencer par programmer, à nouveau, la fonction `facto`.
3. Écrire une fonction Python `approx(n)` qui trace sur un même graphique la fonction sin et la partie régulière du développement limité de sin à l'ordre n dans une fenêtre où les abscisses varient de -10 à 10 et les ordonnées de -10 à 10. Testez la fonction avec $n = 1, ..., 20$.

II Polynômes

Exercice 4 *Représentation des polynômes*

En Python, comme précédemment, on peut représenter un polynôme par une liste. Par exemple : `[1,2,4]` représente le polynôme

$$1 + 2X + 4X^2.$$

1. Que représente la liste `[1]*8` comme polynôme ?
2. Écrire une fonction Python qui, pour une liste `L` retourne la même liste mais en éliminant tous les 0 à droite « en trop ». À quoi sert cette fonction ?
3. Écrire une fonction `deg(L)` qui pour une liste `L` renvoie le degré du polynôme représenté.
4. Écrire une fonction `somme(L1,L2)` qui renvoie une liste représentant la somme des polynômes représentés par `L1` et `L2`.
5. Écrire une fonction `ajout_zero(L1,L2)` qui renvoie une liste de deux listes de longueur `len(L1) + len(L2)` en complétant `L1` et `L2` par des 0.
6. Écrire une fonction `produit(L1,L2)` qui renvoie une liste représentant le produit des polynômes représentés par `L1` et `L2`.

Exercice 5 *Algorithme de Horner*

Pour évaluer un polynôme en un réel x, par exemple $1 + 2X + 4X^2 + 2X^3$ on peut procéder de deux façons :

$$1 + 2x + 4x^2 + 2x^3 = 1 + 2 \times x + 4 \times x \times x + 2 \times x \times x \times x$$

ou de la façon suivante :

$$1 + 2x + 4x^2 + 2x^3 = 1 + x \times \big(2 + x \times (4 + 3 \times x)\big).$$

Cette seconde façon de procéder s'appelle l'algorithme de Horner.

1. Sur l'exemple, compter le nombre d'opérations effectuées et comparer l'efficacité des deux méthodes.
2. Calculer, pour un polynôme de degré n, les complexité $C_1(n)$ et $C_2(n)$ de chaque méthode et confirmer la question précédente.
3. Écrire une fonction `horner(L,x)` qui donne le même résultat que `evaluation(L,x)` mais avec l'algorithme de Horner.
4. Importer le package `time`.
5. Écrire une fonction `graph(n)` qui compare les temps de calculs de `evaluation([1]*n,2)` et de `horner([1]*n,2)` en fonction de n.

III Algorithmes de tri

Exercice 6 *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)` qui renvoie une liste de taille n d'entiers naturels inférieurs ou égaux à 10000.

Exercice 7 *Tri fusion*

À partir de deux listes triées, on peut facilement construire une liste triée comportant les éléments issus de ces deux listes : on parle de fusion des listes. Le principe de l'algorithme de tri fusion repose sur cette observation : le plus petit élément de la liste à construire est soit le plus petit élément de la première liste, soit le plus petit élément de la deuxième liste. Ainsi, on peut construire la liste élément par élément en retirant tantôt le premier élément de la première liste, tantôt le premier élément de la deuxième liste (en fait, le plus petit des deux, à supposer qu'aucune des deux listes ne soit vide, sinon la réponse est immédiate). Ainsi, on peut en séparant récursivement une liste en deux listes de taille « égale » successivement trier complètement une liste.

1. Écrire une fonction Python `fusion(L1,L2)` qui, à partir de deux listes **triées** `L1` et `L2` renvoie une liste triée comportant tous les éléments de `L1` et `L2`. Par exemple

```
> > > fusion([1,2,4,8,9],[1,5,12])
[1,1,2,4,5,8,9,12]
```

2. Écrire une fonction `tri_fusion(L)` qui trie la liste `L` qui sépare à chaque étape la liste en deux listes dont la taille a été au moins divisée par 2 (utiliser du slicing). On donne quelques éléments du code :

```
def tri_fusion(L):
    if len(L)<=1:

    else:
        L1,L2=L[:len(L)//2+1],...
        return fusion(...)
```

Exercice 8 *Comparaisons de complexités*

1. Recopier la fonction `tri_insertion` du cours.
2. Représenter sur un même graphique, le temps d'exécution des deux fonctions précédentes avec la liste `liste_alea(n)` en fonction de n pour $n = 0, ..., 100$ voire 1000 (vous serez limité par le caractère récursif de certains fonctions).
3. Expliquer le graphe précédent.

Exercice 9 *Tri fusion en itératif*

Écrire une fonction `tri_fusion` en utilisant uniquement des méthodes itératives.

IV Calendrier

Exercice 10 *Années bissextiles*

On rappelle qu'une année est bissextile (comporte 366 jours au lieu de 365) lorsque

1. l'année est divisible par 4 et non divisible par 100 ;
2. l'année est divisible par 400.

Écrire une fonction Python `bissextile(n)` qui renvoie `True` si l'année est bissextile et `False` sinon.

Exercice 11 *Premier janvier et jour*

1. Sachant que le premier janvier 2013 est tombé un dimanche, écrire une fonction `jour(n)` retournant le jour dans la semaine où tombe le premier janvier d'une année. On prendra comme convention : 1 pour lundi, 2 pour mardi... jusqu'à 7 pour dimanche. Vérifier avec les exemples suivants : le premier janvier de 2042 tombe un dimanche et le premier janvier 2000 est tombé un samedi.
2. Écrire une fonction prenant en entrée une date (triplet d'entiers jour/mois/année) et retournant le jour dans la semaine correspondant à cette date.

Exercice 12 *Age*

Écrire une fonction Python qui prend en entrée deux liste à trois éléments de la forme `[annee,mois,jour]` (une pour la date du jour et une pour la date de naissance) et qui renvoie l'âge d'une personne.

V Compléments

Exercice 13 *Famille de polynômes*

On pose $P_0 = 2$, $P_1 = X$ et pour tout $n \in \mathbb{N}$,

$$P_{n+2}(X) = XP_{n+1}(X) - P_n(X).$$

1. Écrire une fonction Python `liste_coeff(n)` qui renvoie la liste des coefficients de P_n.
2. Représenter, sur un même graphique $P_0, P_1, \ldots, P_{10}$.
3. Montrer que pour tout $n \in \mathbb{N}$ et $\theta \in \mathbb{R}$,

$$P_n(2\cos\theta) = 2\cos(n\theta).$$

4. Déterminer les racines de P_n (on confirmera avec le graphique précédent).

Exercice 14 *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 :