TP1

1 Sage Math

Les travaux pratiques seront effectués en utilisant Sage math.

Le logiciel Sage math `http://www.sagemath.org/fr` est un logiciel de libre de mathématiques. Il se présente comme une alternative libre à plusieurs logiciels mathématiques tels que Maple, Matlab, Mathematica...

Le logiciel est codé en python et il est possible de créer de nouvelles classes et de nouvelles fonctions pour sage en python. De fait les fonctions que vous écrirez seront en python.

L'avantage de sage pour les TPs de graphes est que sage dispose de plusieurs classes qui permettent de représenter des graphes et de les manipuler simplement. Il est également assez aisée d'écrire des fonctions qui vont manipuler ces graphes.

Les classes qui nous intéressent sont `Graph` qui permet de représenter un graphe en mémoire et de le manipuler. La classe `DiGraph` qui permet de représenter un graphe orienté et de le manipuler avec des fonctions propres au graphes orientés. Ainsi que les classes `graphs` et `digraphs` qui contiennent des méthodes pour créer (construire) certaines familles de graphes. Comme par exemple `graphs.CompleteGraph(10)` qui créera une clique à 10 sommets...

La syntaxe est la même que pour python, tout ce qui apparaît après `#` est un commentaire. Vous pouvez obtenir de l'aide sur une fonction en utilisant `help(le_nom_de_la_fonction)`. La complétion est également présente. Exemple tapez `Graph.` puis tabulation, et sage vous affiche toutes les fonctions disponibles pour la classe `Graph`.

Exemple :

```
sage:g=Graph(15)
sage:# g crée un graphe avec 15 sommets

sage:g=graphs.CompleteGraph(15)
sage:# g est maintenant une clique à 15 sommets

sage:g.show()
sage:# permet d'afficher le graphe g (le dessin n'est pas focément optimal)
sage:# dans un programme à part.

sage:g.show3d()
sage:# permet d'afficher le graphe en 3 dimensions
sage:# dans un programme à part.

sage:g.vertices()
sage:# renvoie la liste des sommets de g

sage:g.edges()
sage:#renvoie la liste des arêtes de g
sage: g.neighbors(g.vertices()[1])
#renvoie la liste des voisins du sommet g.vertices()[1]

sage: g=g.complement()
# calcule le graphe complémentaire de g
```

```
sage: g.show()
g.add_edges([[10,11],[11,12]])
# ajoute au graphe les arêtes {10,11} et {11,12}

sage: g.delete_edges([[10,11],[11,12]])
# supprime les arêtes que l'on a ajouté.

g.degree()
#renvoie la liste des  degrés du graphe
g.degree(g.vertices()[0])
#renvoie le nombre de voisin du sommet en première position dans la liste

sage: h=g # copie le graphe g dans h
sage: h=h.complement() # h est maintenant une clique.

sage: g.add_edges([ [i,(i+1)%g.num_verts()] for i in range(g.num_verts())])
# ajoute les arêtes {i,(i+1)%n} pour tout i entre [0 et n)
# g.num_verts() représente le nombre de sommet de g
sage: g.add_edges([ [i,(i+2)%g.num_verts()] for i in range(g.num_verts())])
#...

sage: cycle=Graph()
sage: cycle.add_edges([[i,(i+1)%6]  for i in range(6)])
sage: chemin= Graph()
sage: chemin.add_edges([[i,i+1]  for i in range(5)])

sage: cylindre= Graph.cartesian_product(cycle, chemin)
sage: cylindre.show()
sage: cylindre.relabel()
sage: cylindre.show()


sage: attach ~/mon_fichier.sage #permet de charger un fichier contenant
#des instructions sage (python  le fichier peut contenier des nouvelles
# fonctions que l'on peut  ensuite utiliser sur les graphes.
```

2 Exercices

Exercice 1

Écrivez une fonction `est_eulerien()` qui prend pour paramètre un graphe et qui renvoie `True` si le graphe est Eulérien `False` sinon.

Exercice 2

Implémentez la fonction parcours en largeur vu en cours.

Exercice 3

- Implémentez la fonction parcours en profondeur vu en cours
- Implémentez l'algorithme qui calcule le tri topologique d'un graphe orienté ou trouve un circuit.
- Implémentez l'algorithme qui calcule les composantes fortement connexes.
- Implémentez de manière itérative la fonction parcours en profondeur...

3 Python

Python est un langage objet à typage implicite (i.e. on ne précise pas le type dans les paramètre des fonctions). Python dispose nativement de plusieurs structures de données telles que les listes/tableaux,

les tuples et les dictionnaire (structure (clé,valeur) où la clé est unique).

3.1 Structure de données

3.1.1 variables simple

```
>>> i=0 # i est donc un entier
>>> j=1.25 #j est donc un flottant
>>> s="bonjour" # s est un chaine de caractères
>>> i+j
1.25
>>> 3/2
1
>>> #renvoie 1 car 2 et 3 sont des entiers c'est donc la division entière qui est utilisée
>>> 3/2.0 # on force 2 comme étant un flottant avec 2.0 division flottante utilisée
1.5
>>> i=2
>>> j/i
0.625
>>> Comme j est flottant c'est la division flottante qui est utilisée...
```

3.1.2 Manipulation des listes

```
>>> L = []
>>> L=range(10) #crée une liste de 10 éléments partant de 0 jusqu'à 9
>>> print L # affiche la liste
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> SL= L[2:4] # crée une sous liste de L en prenant les éléments L[2], L[3] et L[4]
>>> print SL
[2, 3, 4]
>>> M = range(5,15) # liste de 10 éléments qui commence à 5
>>> print M
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
>>> N=range(5,15,3)  #Liste qui commence à 5 jusqu'à 15 (exclu) avec un incrément de 3
>>> print N
[5, 8, 11, 14]
>>> L.append("bonjour") #ajout "bonjour" à la fin de la liste.
>>> print L
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 'bonjour']
>>> L.insert(0,"au revoir") #Insère au début de la liste
>>> print L
['au revoir', 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 'bonjour']
>>> L.reverse() # renverse la liste
>>> print L
['bonjour', 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 'au revoir']
>>> L.pop() # renvoie le dernier élément de L et le supprime de L
'au revoir'
```

3.1.3 tuples

Un tuple est une structure de type liste de taille fixe, et où la valeur des éléments est fixée à la création :

```
>>>t=(1,2,3)
>>>print t[0]
1
>>>t[0]=10 #on peut essayer de modifier le premier élément du tuple t, on peut essayer...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

3.1.4 Dictionnaires

Un dictionnaire, aussi appelé liste de hachage, est un tableau où les index ne sont pas forcément des entiers positifs.

```
>>> dico = {} # déclaration d'un dictionnaire vide
>>> dico = {('a', 26), ("bonjour",'b')} # chaque élément est un couple (tuple à 2 éléments)
# où la première partie est la clé et la seconde la valeur.
>>>for i,j in dico:
>>>...     print i #i est la clé
>>>...  print j #j est la valeur
```

3.2 Instructions en python

3.2.1 L'instruction `while`

Exemple d'exécution dans la console python :

```
>>> i=0
>>> while i<10:
...     print i
...     i+=1
...
0
1
2
3
4
5
6
7
8
9
```

Exemple de boucle avec un `if` imbriqué dans le `while`. Le `:` à la fin de la ligne du `while` ou du `if` indique que ce qui vient après fait partie du bloc d'instruction. À condition que l'indentation des instructions soient correctes.

```
while i < 10:
    print i
    i+=1
    if i%2 ==0:
        i+=1
```

La condition dans le `while` peut être exprimé avec les opérateurs de logique `and`, `or`, `==`, `!=`...

3.2.2 L'instruction `for`

L'instruction `for` est un peu différente de la version habituellement rencontrée dans autres langages de programmation. En effet en python `for` balaie un intervalle de valeurs. Cet intervalle de valeur est défini à l'aide de la fonction `range` (voir plus haut) ou `xrange`. La fonction `for` fonctionne également comme un *for each*.

```
 >>> L=[1,10,100,1000,10000]
>>> for i in L:
...     print i
...
1
10
```

```
100
1000
10000
```

3.3 `if, else ...`

Python propose les structures conditionnelles classiques :

```
>>> i=23
>>> if i % 2 == 0:
...     print "ok"
... elif i%5 == 0: # correspond au else if
...     print "ok"
... else:
...     print "D'oh"
```

3.4 Fonctions

Il est également possible de définir des fonctions avec le mot clé `def` suivi du nom de la fonction et entre parenthèses les arguments.

```
def Iteration(x):
    for i in xrange(x):
        print i

    print
    j=0
    while j < x:
        print j
        j+=3

def Liste(x):
    L = []
    for i in xrange(x+1):
        L.append(i)
    return L

def DelListe(L):
    for i in range(len(L)):

        if (L[i] % 2 == 0):
            print L[i]
            del L[i] #del supprime ici l'élément i de la liste
            #(la liste possède un élément de moins).

    return L

Iteration(10)

print Liste(10)
print
print DelListe(Liste(10))
```