Travaux Dirigés de Combinatoire n°1

Master d'informatique

—Première année—

Python et Sage

`Sage` est un logiciel open-source de calcul formel sous licence GPL. Il combine la puissance de nombreux programmes open-source en une interface commune basée sur Python.
Un système de calcul formel est capable d'effectuer à la fois des calculs numériques sur des entiers, des fractions, et des flottants en précision arbitraire, mais également sur des objets formels tels que les polynômes, les fractions rationnelles et les fonctions usuelles (trigonométriques, exponentielles, *etc.*..)
`Sage` couvre une vaste gamme de mathématiques, dont l'algèbre, l'analyse, la théorie des nombres, la cryptographie, l'analyse numérique, la combinatoire, la théorie des graphes, l'algèbre linéaire formelle, etc ...

Généralités

- On peut utiliser `Sage` soit en ligne de commande (via l'interface `ipython`), soit par l'interface web du notebook.
- La commande «`<nom>?`» permet d'avoir de l'aide sur nom. «`<nom>??`» permet de lire également le code source.
- On trouvera la documentation de python en ligne sur `http://docs.python.org/`.

Type de données Python/Sage

- En python, le **typage est dynamique** ; il n'y a pas de déclaration de variable. La fonction `type` retourne le type d'un objet ;
- Toute valeur est une **instance d'une classe** ; Il n'y a pas de différences entre classes et types. La fonction `isinstance(f, t)` retourne `True` si la valeur de l'expression `f` est une instance de la classe `t`.
- L'opérateur `=` est l'affectation à une variable, l'opérateur d'égalité s'écrit `==`.
- Les **types standards** sont `bool, int, list, tuple, set, dict, str`.
- Le type `bool` (**Booléens**) à deux valeurs : `True` et `False`.
- À l'inverse de Python, `Sage` réalise une **arithmétique exacte** sur les entiers. Les entiers exacts de `Sage` sont de type `Integer`.
- Une **liste** est une structure de données qui regroupe des expressions. L'instruction `range` permet de créer des listes d'entiers.

- ☞ On peut aussi crée des listes **en extension** :

  `[<expr> for <nom> in <iterable> (if <condition>) ]`

  par exemple : `[i^2 for i in range(10) if i % 2 == 0]`.
- ☞ Une **tuple** est une structure très similaire à une liste, on le crée avec des parenthèses. Le tuple vide est obtenu par `()` ou par le constructeur `tuple()`. S'il n'y a qu'un seul élément, on écrit `(a,)`. Un tuple n'est pas modifiable mais est en revanche hashable. On peut aussi créer des tuples en extension.
- ☞ Un **ensemble** est une structure de données qui regroupe des expressions sans duplication ni ordre. On la crée à partir d'une liste ou d'un tuple par le constructeur `set`. L'ensemble vide est noté `set()`. Les éléments d'un ensemble doivent être hashables.
- ☞ Un **dictionnaire** est une table d'association, qui a une clef associe une valeur. Les clés doivent être hashables. On crée un dictionnaire par le constructeur `dict` ou en extension par

  `{cle1 : valeur1, cle2 : valeur2 ...}`
- ☞ Les guillemets simples `' '` ou doubles `" "` délimitent les **chaîne de caractères**. On concatène les chaînes de caractères par `+`.
- ☞ Pour une liste un tuple ou un dictionnaire l'**opérateur d'indexation** est noté `l[i]`. Il peut aussi prendre des intervalles `l[:]`, `l[:b]`, `l[a:]` ou `l[a:b]`. Les indices négatifs sont comptés à partir de la fin.
- ☞ La fonction `len` permet d'obtenir le nombre d'élements d'une liste, d'un tuple ou d'un dictionnaire.

Structure de contrôle Python :

En python, il n'y a pas de mot clé de début et de fin de bloc, tout est fait par l'indentation. La plupart du temps un bloc est introduit par deux points «:». Python dispose des structures de contrôle suivantes :

- ☞ Instruction conditionnelle :

```
if <condition>:
    <suite_d'instructions>
[elif <condition>:
     <suite_d'instructions>]*
[else:
     <suite_d'instructions>]
```

- ☞ Dans les expressions exclusivement, on peut aussi écrire :

  `<valeur> if <cond> else <valeur>`
- ☞ Instructions de boucle :

```
for <nom> in <iterable>:
     <suite_d'instructions>
[else:
     <suite_d'instructions>]
while <condition>:
     <suite_d'instructions>
[else:
     <suite_d'instructions>]
```

- ☞ Le bloc `else:` est éxécuté à la fin de la boucle si la sortie de boucle se fait normalement (ni par `break`, ni par une exception).
- ☞ Dans une boucle on peut forcer le passage à l'étape suivante par l'instruction `continue`.
- ☞ On peut sortir prématurément de la boucle avec l'instruction `break`. Dans le cas le bloc `else` n'est pas exécuté.

▸ **Exercice 1.** Pour chacun des ensemble suivants, calculer la liste des éléments et la somme de deux façons quand c'est possible : avec une boucle puis en extension

1. n premiers termes de la série harmonique

$$\sum_{i=1}^{n} \frac{1}{i} ;$$

2. les entiers impair entre 1 et n ;
3. les n premiers entiers impairs.
4. les entiers entre 1 et n et qui ne sont divisibles ni par 2 ni par 3 ni par 5 ;
5. les n premiers entiers qui ne sont divisibles ni par 2 ni par 3 ni par 5.

▸ **Exercice 2.** Afficher les 50 premiers nombres premiers. Quel est leur produit ? Comment retrouver la décomposition en facteurs premiers du nombre ainsi obtenu ?

▸ **Exercice 3. (Les fonctions)**. Une fonction se définit par :

```
def <nom>(<arguments>):
   <suite d'instructions>
```

On renvoi un résultat avec l'instruction `return`. Dans le cas d'une fonction simple on peut écrire une fonction anonyme par (il n'y a pas de return dans ce cas) :

```
lambda <arguments>: <expression>
```

Note : les fonctions sont des objets comme les autres, on peut les stocker dans des variables et les retourner (programmation fonctionnelle).

1. Écrire une fonction `est_pair(n)` qui retourne `True` si `n` est pair et `False` sinon.
2. Écrire une fonction `un_sur_deux(l)` qui retourne une liste contenant un élément sur deux dans la liste `l`.
3. Écrire une fonction calculant le ième nombre de Fibonacci. Comment améliorer les performances ?

Une fonction peut prendre un nombre variable d'arguments : la déclaration

```
def ma_fonction(*args):
```

crée une fonction dont les paramètres sont placés dans un tuple nommé `args`.

4. Écrire une fonction qui prend en paramètre un nombre variable de nombres et qui retourne leur somme.

▸ **Exercice 4. (Programmation objet en Python/Sage).**

On déclare une classe par la syntaxe :

```
class <nom_de_la_classe>(<super classes séparées par ,>):
    def <nom de methode>(self, <parametres>):
        <code le la méthode>
    <variable_de_classe> = valeur
```

Les attributs de l'instance `self` sont rangés dans `self.<nom_de_l_attribut>`. Il n'y a pas de mécanisme de protection (`public` ou `private` de C++ ou Java. Par convention les attributs privés commence par un souligné.

Voici un exemple de classe :

```
class Eleve(SageObject):
    def __init__(self, nom, notes):
        assert(isinstance(nom, str))
        assert(isinstance(notes, tuple))
        self._nom  = nom
        self._notes = notes

    def _repr_(self):
        return "Eleve "+ self._nom

    def moyenne(self):
        return float(sum(self._notes)/len(self._notes))
```

On remarque que contrairement à C++ ou à Java l'objet courant d'une méthode est nommé explicitement (par convention `self`). La méthode `__init__` est le constructeur de la classe. On crée un objet par :

```
toto = Eleve("Toto", (1,3,2))
```

La méthode `Sage _repr_` permet d'afficher un objet. On appelle une méthode avec le point, comme dans `toto.moyenne()`.

On code une série $F(x) = \sum f_i x^i$ par un objet contenant une fonction qui prend en paramètre un entier n et qui retourne le coefficient de X^n.

1. Écrire une classe `PowerSeries` dont le constructeur prend en paramètre une fonction.
2. Écrire une méthode `_repr_` qui affiche l'objet. L'ordre de développement sera stocké dans une variable de classe `Order`.
3. Écrire une méthode `mult_coeff(self, co)` qui prend en paramètre une constante `co` et qui retourne la fonction associée à la série coF ;
4. Écrire une méthode `__add__(self, other)` pour ajouter deux séries.
5. Écrire une méthode `__mult__(self, other)` pour multiplier deux séries.
6. Écrire une méthode `__pow__(self, n)` qui retourne la puissance nième de `self`.