

## FEUILLE D'EXERCICES n° 1

### Travail sur machine : initiation à sage

#### 1. INTRODUCTION

Sage est un logiciel libre de calcul, accessible sur <http://www.sagemath.org>. Il utilise le langage python. Il existe trois façons de l'utiliser.

- En utilisant une interface graphique (le notebook ou l'accès internet).
- Par la ligne de commande interactive, en appelant éventuellement des programmes écrits sur un éditeur de texte. Pour cela, il suffit de taper la commande `sage` sur une fenêtre terminal.
- En écrivant des scripts python indépendants qui font appel à la bibliothèque `sage`.

Vous pourrez utiliser au choix la ligne de commande interactive ou bien le notebook.

#### 2. QUELQUES COMMANDES UTILES

- Pour ouvrir `sage` : `sage` sur une fenêtre terminal.
  - Pour ouvrir le notebook : `notebook()`. Une fenêtre `sage` s'ouvre alors sur votre navigateur web.
  - Pour quitter `sage` : `ctrl d`
  - Pour interrompre un calcul : `ctrl c`
  - Dernier résultat : `_`
  - Complétion : `Tab`
  - Commentaire : `#` pour une ligne, `''' blablabla '''` pour un bloc.
  - Informations sur une fonction : `Nom_de_fonction?` (par exemple `factor?`).
- Pour quitter l'aide ainsi activée, taper `q`.

Les instructions suivantes concernent l'utilisation de `sage` via la ligne de commande sur une fenêtre "terminal".

- À l'extérieur de la session `sage`, nous écrirons des programmes dans différents fichiers, munis de l'extension `.sage`. Il faudra alors pouvoir se servir de ces fichiers dans notre session `sage`. Pour lire un tel fichier `toto.sage`, on dispose de la commande `load("toto.sage")`.

- Pour attacher un fichier `toto.sage` : `attach("toto.sage")`. Grâce à cette commande, à chaque fois que l'on tape `entrée`, le fichier est relu. C'est utile si l'on a entre-temps modifié ce programme.

- Pour écrire ces fichiers `.sage`, on utilise un éditeur au choix. L'un des choix pourra être l'éditeur `emacs`. Pour lancer `emacs`, valider `emacs &` sur votre fenêtre

`terminal` (le `&` étant là pour garder la main sur le terminal). Pour cette utilisation de `emacs`, il est commode de passer en mode `python`. Pour cela, il y a deux possibilités. La première : exécuter la commande `escape x python-mode` sur la fenêtre `emacs`. Il faudra alors exécuter cette commande à chaque fois. La seconde : ajouter la ligne suivante dans votre fichier `.emacs` :

```
(setq auto-mode-alist (cons '("\\.sage$" . python-mode) auto-mode-alist))
```

cette manipulation (à faire une fois pour toutes) permet de passer automatiquement en mode `python` lors de l'édition d'un fichier muni de l'extension `.sage`.

### 3. ACCÈS VIA INTERNET

Il pourra vous être utile de pouvoir accéder à `sage` via internet, par exemple pour travailler chez vous. Pour cela, aller à la page `cocalc.com` sur un navigateur.

### 4. OPÉRATIONS DE BASE, TESTS ET AFFECTATIONS

Expérimenter les commandes suivantes (on tapera la touche `entrée` entre chaque commande).

```
1+1
a=8
a
a==5
a==8
parent(a)
type(a)
a=4/3
parent(a)
2<=3
1==1;1!=1
```

(`sage` utilise `=` pour les affectations, et `==`, `<=`, `>=`, `<`, `>`, `!=` pour les comparaisons).

```
2**5 (exponentiation)
2^5 (autre écriture pour l'exponentiation)
10%3
10/3
10//3
4 * (10 // 4) + 10 % 4 == 10
floor(2.4);ceil(2.4);round(2.4);round(2.5)
Taper
a=9
```

et valider, puis taper

```
a.
```

et actionner la touche `Tab`. Alors, la liste des fonctions pouvant s'appliquer à `a` apparaît. Ici, `a` est un entier. Par exemple, dans la liste, on lit `a.bits`. Si vous voulez savoir à quoi correspond cette commande vous validez

`a.bits?`

Plus généralement, pour tout objet préalablement défini (par exemple un entier, un polynôme, un anneau, un corps, etc), on peut utiliser la complétion `Tab` de cette manière. Cela peut être utile lorsque l'on cherche le nom d'une fonction.

Attention, pour appliquer une fonction disons `fonc` sur  $a$  la syntaxe est parfois `fonc(a)` et parfois `a.fonc()`, il faut tester.

## 5. LISTES, ENSEMBLES

1. Expérimenter les commandes suivantes (ne pas hésiter à évaluer  $S$  après chaque commande pour vérifier son effet).

```
S=[1,2,3,12];S
len(S)
S[0]
S[3]
S[-1]
S[1]=6
S.append(10)
S.extend([18,19])
[1,2,3]+[2,4,5,6] (c'est une autre manière de concaténer).
S[2:4]
S[2:]
S[:-1]
S=range(1,10)
S=[0..10]
S=[1,5,..33]
29 in S
S[1]=50
S.sort()
S=[i^2 for i in [1..10]]
S=[i^2+1 for i in [2,4..,10] if is_prime(i^2+1)]
S=set([1,1,2,2,5,5,4,4]) (c'est un ensemble. Il n'y a pas de répétition et les entrées sont triées).
```

Chercher des commandes de sage pour calculer  $S \cup T$ ,  $S \cap T$  et  $S \setminus T$ , où  $T = \{1, 5, 6, 7\}$ .

**Remarque.** La commande `set('***')` donne l'ensemble défini par `'***'`. De même, si on fait `list(S)`, on obtient la liste définie par l'ensemble  $S$ , c'est-à-dire ici la liste  $[1, 2, 5, 4]$ . Plus généralement, si  $A$  est un objet, défini comme un ensemble, ou un ensemble muni d'une structure, comme un anneau, un corps, etc. et si  $a$  est un objet susceptible de définir un élément de  $A$ , alors la commande `A(a)` définit cet objet. Nous verrons d'autres exemples de ce principe plus tard.

Exécuter encore :

```
sum(S)
```

La fonction `sum` additionne donc les éléments d'une liste, ou d'un ensemble. Elle peut être utilisée autrement.

```
sum(k^2,k,0,4)
```

Il y a un problème : il faut déclarer la variable  $k$ . Dans Sage, il faut toujours déclarer les objets sur lesquels on travaille. Faire plutôt

```
k=var('k');sum(k^2,k,0,4)
```

2. En utilisant `sum`, retrouver l'expression de la somme des carrés des entiers compris entre 1 et  $n$  (pour obtenir la forme factorisée de cette expression, utiliser la fonction `factor`).

3. Construire la liste  $L = [1, 2, 3, 4, 5, 6]$ . Quels sont les éléments  $L[3]$ ,  $L[0]$ ,  $L[-1]$ ,  $L[-3]$ ? Comment obtenir la liste  $[2, 3, 4]$  directement à partir de la liste  $L$ ?

4. La commande `map` permet d'appliquer une fonction à tous les éléments d'une liste, d'un ensemble, d'un  $n$ -uplet ... Exécuter les commandes suivantes.

```
f(x)=x^3
map(f,L)
list(map(f,L))
```

## 6. BOUCLES, FONCTIONS

La syntaxe est celle de Python. La structure des fonctions est déterminée par l'indentation.

Pour créer une fonction `f` dont les indéterminées sont `a`, `b`, `c` :

```
def f(a,b,c):
    Instruction 1
    Instruction 2
    :
    return "résultat"
```

1. Expérimenter la fonction suivante, d'abord en suivant la ligne de commande, puis en l'écrivant sur un fichier à part `pair.sage`, à attacher grâce à la commande `attach`. Il faut veiller à bien respecter l'indentation.

```
def pair(n):
    v = []
    for i in range(3,n):
        if i % 2 == 0:
            v.append(i)
    return v
```

2. Pour les boucles, on utilise la syntaxe suivante (la structure est toujours déterminée par l'indentation).

```
if :
if condition:
    instruction 1
    instruction 2
```

```

:
elif condition 2:
    instruction 1
    instruction 2
:
elif condition 3:
    instruction 1
    instruction 2
:
else:
    instruction 1
    instruction 2
:

```

```

while :
while condition:
    instruction 1
    instruction 2
:

```

```

for :
for i in L:
    instruction 1
    instruction 2
:

```

Ici, L peut être une liste, ou bien un  $n$ -uplet, ou bien un ensemble, ou une autre structure pouvant être énumérée.

Expérimenter par exemple :

```

k=GF(3)
for i in VectorSpace(k,2):
    print i

```

La commande `GF(3)` définit le corps  $\mathbb{F}_3$ . Nous verrons plus tard ce type de commande.

**3.** On appelle triplet pythagoricien tout triplet  $(x, y, z)$  d'entiers naturels non nuls tels que

$$x^2 + y^2 = z^2.$$

a. Faire la liste de tous les triplets pythagoriciens dont toutes les composantes sont inférieurs ou égales à 50.

b. Déterminer les triplets pythagoriciens  $(a, b, c)$  tel que  $a + b + c = 1000$  (il n'y en a qu'un, à permutation près de  $a$  et  $b$ ).

4. Soit  $n$  un entier naturel. On note  $E(n)$  l'ensemble des entiers naturels compris entre 1 et  $n$ .

a. Écrire une fonction qui, étant donné un entier  $n$ , donne la liste des parties de  $E(n)$  à 2 éléments. Pour cela, on peut écrire la liste des  $[i, j]$  tels que  $\{i, j\} \subset E(n)$  et  $i < j$ .

b. Écrire une fonction qui, étant donnés  $n, k$  un entier tel que  $1 \leq k < n$  et la liste des parties de  $E(n)$  à  $k$  éléments (sous la forme  $[i_1, \dots, i_k]$  où  $i_1 < i_2 < \dots < i_k$ ), donne la liste des parties de  $E(n)$  à  $k + 1$  éléments.

c. Écrire une fonction qui, étant donnés  $n$  et  $k$ , donne la liste des parties de  $E(n)$  à  $k$  éléments.

## 7. OPÉRATIONS ÉLÉMENTAIRES SUR LES POLYNÔMES

Dans Sage, on peut définir un anneau de polynômes et utiliser un arsenal de fonctions spécifiques à ces polynômes.

Dans ce paragraphe, nous n'utiliserons pas ces fonctions. L'exercice consiste à programmer les opérations élémentaires sur les polynômes, lesquels seront représentés par des listes : la liste  $[p_0, \dots, p_{n-1}]$  représente le polynôme  $\sum_{i=0}^{n-1} p_i x^i$  (où les  $p_i$  appartiennent à un anneau  $R$ ).

On pourra tout de même utiliser les fonctions de sage sur les polynômes à titre de vérification. On se place dans le cas où l'anneau de base est  $R = \mathbb{Q}$ . Pour définir  $\mathbb{Q}[X]$  :

```
Qx.<x>=PolynomialRing(QQ)
```

Alors,  $\mathbb{Q}x$  est l'anneau  $\mathbb{Q}[x]$  (on aurait pu écrire un autre nom à la place de  $\mathbb{Q}x$ , comme  $A$  ou  $toto$ ). Si  $\mathbf{p} = [p_0, \dots, p_{n-1}]$ ,  $\mathbb{Q}x(\mathbf{p})$  est le polynôme  $\sum_{i=0}^{n-1} p_i x^i$  et si  $P$  est un polynôme,  $\text{list}(P)$  est la liste qui représente  $P$ . Ainsi, si  $\mathbf{p}$  et  $\mathbf{q}$  sont des listes, on pourra vérifier les résultats par les commandes suivantes.

```
list(Zx(p)+Qx(q))
list(Zx(p)*Qx(q))
map(list,Qx(p).quo_rem(Qx(q)))
```

`quo_rem` donne le quotient et de reste de la division euclidienne. `map` permet d'appliquer la commande `list` à chacun de ces deux polynômes (voir aussi le point 4 du paragraphe 5).

1. Les listes  $[1, 2]$  et  $[1, 2, 0]$  définissent le même polynôme. Écrire une fonction `Nettoie` qui enlève les 0 en fin de liste.

2. Écrire une fonction `Plus(P, Q)` qui additionne  $P$  et  $Q$ . Quelle est le nombre d'opérations dans  $R$  nécessaires ?

3. Écrire un algorithme `Decale(P, i)` qui multiplie  $P$  par  $x^i$ .

4. Écrire un algorithme `MultScalaire(P, a)` qui multiplie  $a$  par  $P$  (où  $a \in R$ ). Quelle est le nombre d'opérations dans  $R$  nécessaires ?

5. Écrire un algorithme `Fois(P,Q)` qui multiplie  $P$  par  $Q$  en utilisant une multiplication similaire à celle vue dans  $\mathbb{N}$ . Montrer que le nombre d'opérations dans  $R$  nécessaires pour cette multiplication est inférieure ou égale à  $2mn + m + n + 1$ .
6. Écrire un algorithme `Divise(P,Q)` (où  $Q \neq 0$ ) qui divise  $P$  par  $Q$  en utilisant une division similaire à celle vue dans  $\mathbb{N}$ . Donner l'ordre de grandeur du nombre d'opérations dans  $R$  nécessaires.