**Prise en main du logiciel Sage**

Sage est un logiciel libre de calcul mathématique qui s'appuie sur le langage Python. Le but de ce TP est de se familiariser avec Sage. Nous allons lancer Sage avec un notebook Jupyter. Pour cela vous pouvez ou bien cliquer sur l'icône correspondante, ou bien lancer un terminal et taper la commande `sage -notebook=ipython`.

Il est indispensable de savoir utiliser l'aide de Sage. Si l'on connaît le nom d'une commande, on obtient l'aide de Sage sur celle-ci en faisant suivre cette commande d'un point d'interrogation. Par exemple : `expand?` pour l'aide sur `expand`. La touche tabulation à la suite d'un début de commande montre les commandes commençant par ce début. On peut également faire une recherche dans la documentation ou les tutoriels de Sage, accessibles dans le menu `Help`. (Il est par contre interdit, aux contrôles continus/examen, et par extension dans tout le cours, de chercher de l'aide grâce à Google ou généralement sur Internet).

Le symbole `#` marque le reste de la ligne comme commentaire.

- Appeler l'aide de la fonction `plot` et trouver l'exemple de tracé de la fonction partie entière (`floor` en anglais).
- Tester cet exemple.
- Les commandes qui permettent de tester une propriété commencent souvent par les lettres `is`. Trouver le nom de la commande qui teste si un nombre est premier.

## 1 Symboles, expressions et variables

Sage peut en premier lieu être vu comme une calculatrice. Il utilise les symboles `+`, `-`, `*`, `/` pour les opérations usuelles $+$, $-$, $\times$, $/$. La puissance se note `^` ou `**`, le reste dans la division euclidienne se note `%`. Sage connaît un certain nombre de fonctions prédéfinies (exp, ln, cos, ...) et de constantes (e, $\pi$, i, $\infty$, ...).

Sage est un outil de *calcul formel* : ses possibilités vont donc bien au-delà de la manipulation de nombres. Plusieurs fonctions Sage permettent en particulier de calculer des dérivées (`diff`), des intégrales ou primitives (`integrate`), des sommes ou séries (`sum`), faisant intervenir des variables symboliques (au sens mathématique du terme). Pour prévenir Sage que, disons la lettre $t$, va être utilisée comme variable symbolique, on tape `var('t')` (notez que le symbole $x$ est par défaut une variable symbolique). Il ne faut pas confondre ces symboles avec des variables (au sens informatique du terme) qui sont des objets auxquels on a donné un nom à l'aide de l'opérateur `=`. Ce dernier opérateur ne doit pas non plus être confondu avec l'opérateur de comparaison `==`.

- Essayer, dans l'ordre, les commandes suivantes et expliquer les résultats obtenus.

```
a = 10
a + 1
a == 3
```

- Tester les autres opérateurs de comparaison `==, >, <, <=, >=, !=`.

D'autres manipulations courantes d'expressions possibles avec Sage sont le développement, la factorisation, la substitution de variables, la simplification (`expand, factor, subs, simplify`).

- Calculer la dérivée de quelques fonctions usuelles.
- Calculer des primitives de quelques fonctions usuelles.
- Factoriser $4x + 4y + 2x^2y + 9x^2 + 2x^3 + 9xy$ en utilisant `factor`.
- Afficher 200 décimales de $\pi$ (voir l'aide sur `N` ou `numerical_approx`).
- Calculer $\sum_{n\geqslant 1} \frac{1}{n^2}$.

## 2 Listes et ensembles finis

En Python, une suite finie d'objets s'appelle une *liste*. Par exemple `s = [4, 7, 3, 3, 6]`. La commande `s[i]` appelle le $i$-ième élément de la liste `s`, sachant que le premier est numéroté `0`. Ainsi, dans l'exemple, `s[1]` est 7. La longueur de `s` s'obtient par `len(s)`. Pour obtenir la liste des entiers entre 1 et $n$ on utilise `[1 .. n]`. Pour obtenir la liste des nombres premiers inférieurs à $n$, on utilise `prime_range(n)`. Pour construire une liste obtenue en faisant varier un paramètre, l'opérateur `for` est très utile. On dit alors que la liste est définie «par compréhension». Pour tester une conjecture sur une liste d'exemples, on utilise souvent les functions `any` et `all`.

- Essayer `[2^ i for i in [1 .. 10]]`.
- Essayer `[i+1 for i in [1 .. 10] if is_even(i)]`.
- Essayer `all([is_odd(i+1) for i in [1 .. 10] if is_even(i)])`.
- Essayer `any([is_even(i) for i in prime_range(100)])`.
- Après avoir défini une liste `s`, expérimenter les commandes `s + s`, `4*s` et enfin `s.append(pi)` suivi de l'affichage de `s`.

Les éléments d'une liste sont ordonnés et peuvent contenir des répétitions. On peut créer un ensemble fini non-ordonné (et sans répétition) à l'aide de la commande `set`.

- Essayer `set([4, 3, 3, 6])`.
- Essayer `set(a*b for a in [1 .. 3] for b in [2 .. 5])`.

En vous inspirant des exemples précédents et en essayant d'être efficace :

- déterminer s'il existe un nombre premier inférieur à 1000 congru à 1 modulo 85 ;
- déterminer la liste des entiers $n$ compris entre 2 et 100 tels que pour tout $p$ premier inférieur à 500 on ait $p \neq 1\,[n]$ ;
- établir la liste des nombres de facteurs premiers distincts des entiers entre 1000 et 1100.

## 3 Un peu de programmation

On peut créer de nouvelles fonctions (à ne pas confondre avec les expressions symboliques rencontrées jusqu'à présent) en utilisant la commande `def`. Par exemple, la définition suivante :

```
def diviseurs(n):
    return [k for k in [1 .. n-1] if n % k == 0]
```

définit une fonction renvoyant la liste des diviseurs d'un entier positif $n$. Notez les deux-points qui marquent le début du corps de la fonction et notez que toutes les lignes qui sont dans le corps de la fonction sont décalées d'un nombre strictement positif et constant d'espaces. C'est la mise en page (en fait l'indentation) qui sert à délimiter les blocs faisant partie d'une fonction ou d'une boucle ou étant exécutés sous condition. Le même principe s'applique pour les boucles définies par `for` ou `while`.

- Essayer d'appeler sur deux arguments entiers positifs :

```
def reste(a, b):
    while a >= b:
        a = a - b
    return a
```

L'instruction `if` sert à exécuter une portion de programme seulement si une certaine condition est vérifiée. Pour exprimer cette condition on peut utiliser les opérateurs de comparaison introduits ci-dessus, que l'on peut combiner logiquement en utilisant `or` et `and`.

```
def absolue(x):
    if (x >= 0):
        return x
    else:
        return -x
```

- Définir une fonction `sinc` qui calcule le sinus cardinal d'un réel ; la tracer.

Lorsque, comme les exemples ci-dessus, on veut simplement calculer une valeur dépendant de la validité d'une condition, il est possible d'utiliser l'opérateur de comparaison ternaire, par exemple `x if x >=0 else -x` . Dans ce cas il n'y a pas de ponctuation ni d'indentation, et la partie `else` est obligatoire. Ainsi le mot clef `if` apparaît dans trois contextes étroitement liés mais différents : le filtrage des listes et ensembles définis par compréhension, l'instruction `if`, et l'opérateur de comparaison ternaire. De même le mot clef `for` apparaît dans les définitions par compréhension mais aussi comme instruction de boucle, comme dans

```
def est_premier(n):
    for k in [2 .. floor(sqrt(n))]:
        if n % k == 0:
            return False
    return True
```

Dans ce cours, presque toutes les boucles pourront être avantageusement remplacées par le mécanisme de liste définie par compréhension. Par exemple la fonction précédente peut s'écrire `return all(n % k != 0 for k in [2 .. floor(sqrt(n))])`. Notons aussi que la fonction `is_prime` utilise un test de primalité beaucoup moins naïf...

- Implanter le calcul de la fonction factorielle $n \mapsto n! = 1 \times 2 \times 3 \times \cdots \times n$ :
  - en utilisant une boucle `for` ;
  - en utilisant un appel récursif.

  Tester vos fonctions à l'aide de la commande de Sage permettant de calculer des factorielles.

# 4 Quelques objets

Sage connaît déjà beaucoup d'objets mathématiques compliqués : par exemple, l'anneau $\mathbb{Q}[X, Y]$ des polynômes en deux variables à coefficients rationnels, le groupe linéaire $\mathrm{GL}(5, \mathbb{C})$ des matrices complexes inversibles de taille $5 \times 5$, le graphe complet $K_7$ sur 7 sommets, *etc.* sont des objets déjà définis dans Sage.

Si $n \geqslant 1$, on peut définir l'anneau $\mathbb{Z}/n\mathbb{Z}$ grâce à `Zmod(n)`. Ainsi, l'anneau `Z7` des entiers modulo 7 est défini par `Z7 = Zmod(7)`. La classe $b$ d'un entier $a$ modulo $n$ peut être représentée par `b=Zmod(n)(a)` (Sage «convertit» `a` en un élément `b` de `Zmod(n)`). L'anneau des entiers est lui noté `ZZ`, les rationnels sont notés `QQ`. Pour obtenir un représentant d'une classe $a \in \mathbf{Z}/7\mathbf{Z}$, on peut utiliser la commande `ZZ(a)`.

- Définir $a = 2\,[7]$ et calculer $a^{-1}$.
- Tester l'égalité de $2\,[7]$ et $2\,[9]$.
- Calculer $7^{5^{5^5}}$ $\left(= 7^{\left(5^{(5^5)}\right)}\right)$ modulo 13.
- Essayer de faire le même calcul en calculant d'abord l'entier $7^{5^{5^5}}$ puis en réduisant modulo 13 à l'aide l'opération `%` qui calcule le reste d'une division euclidienne (par exemple, `9 % 4` renvoie 1).
- Quel est l'effet de la commande `//` ?

Les objets Sage ont des fonctionnalités d'introspection et peuvent donc être interrogés sur leur classe, les données qu'ils renferment ou sur les méthodes qu'on peut leur appliquer. La classe d'un objet est obtenue grâce à la fonction `type`.

- Déterminer la classe de certains des objets rencontrés jusqu'ici.

Sage possède également la notion de parent qui donne l'ensemble auquel un objet est considéré appartenir.

- Tester la commande `parent` sur `5`, `5/2` et `x`.

## 5 Exponentiation rapide

Soit $E$ un ensemble, muni d'une loi de composition associative notée multiplicativement, et qui possède un élément neutre noté 1. Soit $x$ dans $E$. On définit la suite $u_n$ par

$$\begin{cases} u_0 = 1 \\ u_n = \left(u_{n/2}\right)^2 & \text{si } n \text{ est pair} \\ u_n = x u_{n-1} & \text{si } n \text{ est impair.} \end{cases}$$

- Montrer que pour tout $n \in \mathbb{N}$, on a $u_n = x^n$.

Pour tout $n \in \mathbb{N}$, on note $C(n)$ le nombre d'opérations effectuées dans $E$ pour calculer $u_n$.

- Montrer que $C(n) \leqslant 2\log_2(n)$ pour tout $n \geqslant 2$.
- Comparer avec le nombre d'opérations effectuées dans $E$ pour calculer naïvement $x^n$.
- En déduire un algorithme de calcul efficace de $x^n$ (on parle d'exponentiation rapide) et l'implanter dans Sage.

L'exponentiation rapide est omniprésente en calcul formel. Considérons par exemple la suite de Fibonacci $(F_n)$ définie par $F_0 = F_1 = 1$ et $F_{n+2} = F_{n+1} + F_n$ pour $n \geqslant 0$. On considère également la suite $(X_n)$ définie par $X_n = (F_n, F_{n+1})$.

- Déterminer une relation de récurrence vérifiée par $(X_n)$.
- Expliquer comment calculer efficacement le $n$-ième nombre de la suite de Fibonacci (sans calculer les termes précédents).
- Expérimenter avec Sage.

## 6 Classes et opérations

Dans ce cours, nous nous contenterons d'utiliser les classes d'objets fournies par Sage, mais comprendre le mécanisme de définition des classes et des opérations est utile. Le code suivant (disponible au format texte sur ecampus) définit une classe rudimentaire de matrices carrées de taille 2. Tous les noms entourés de traits de soulignement sont des noms conventionnels non-négociables. L'argument `self` désigne l'objet en cours de création ou sur lequel on agit directement.

```
class mat:
    def __init__(self, a, b, c, d):
        """
        Initialise un objet de classe mat
        """
        self.a = a
        self.b = b
        self.c = c
        self.d = d

    def __repr__(self):
        """
        Représente en chaîne de caractères un objet de classe mat
        """
        return " ".join(["matrice", latex(self.a), latex(self.b),
                         latex(self.c), latex(self.d)])

    def __eq__(self, other):
        """
        Teste l'égalité d'un objet de classe mat avec un autre
        """
        return [self.a, self.b, self.c, self.d] == [other.a, other.b, other.c, other.d]

    def __add__(self, other):
        """
        Renvoie la somme de cet objet de classe mat et de cet autre objet
        """
        return mat(self.a + other.a, self.b + other.b, self.c + other.c, self.d + other.d)

    def det(self):
        """
        Renvoie le déterminant d'un objet de classe mat
        """
        return self.a * self.d - self.b * self.c
```

- Essayer `A = mat(1, 2, 3, 4)`, `B = mat(2, 2, 2, 2)`, `A + B`, `A.det()`, `A == B`.
- Créer une classe d'objets `Zmod5` représentant les éléments de $\mathbb{Z}/5\mathbb{Z}$, de sorte qu'on puisse les additionner et les multiplier.
- Vérifier `Zmod5(3) + Zmod5(2) == Zmod5(0)`.