CHAPITRE 2

Visite guidée

Cette partie présente une sélection des possibilités actuellement offertes par Sage. Pour plus d'exemples, on se reportera à *Sage Constructions* (« Construction d'objets en Sage »), dont le but est de répondre à la question récurrente du type « Comment faire pour construire . . . ? ».

On consultera aussi le *Sage Reference Manual* (« Manuel de référence pour Sage »), qui contient des milliers d'exemples supplémentaires. Notez également que vous pouvez explorer interactivement ce tutoriel — ou sa version en anglais — en cliquant sur `Help` à partir du notebook de Sage.

(Si vous lisez ce tutoriel à partir du *notebook* de Sage, appuyez sur `maj-enter` pour évaluer le contenu d'une cellule. Vous pouvez même éditer le contenu avant d'appuyer sur `maj-entrée`. Sur certains Macs, il vous faudra peut-être appuyer sur `maj-return` plutôt que `maj-entrée`).

2.1 Affectation, égalité et arithmétique

A quelques exceptions mineures près, Sage utilise le langage de programmation Python, si bien que la plupart des ouvrages d'introduction à Python vous aideront à apprendre Sage.

Sage utilise = pour les affectations. Il utilise ==, <=, >=, < et > pour les comparaisons.

```
sage: a = 5
sage: a
5
sage: 2 == 2
True
sage: 2 == 3
False
sage: 2 < 3
True
sage: a == 5
True
```

Sage fournit toutes les opérations mathématiques de base :

```
sage: 2**3    #  ** désigne l'exponentiation
8
sage: 2^3     #  ^ est un synonyme de ** (contrairement à Python)
8
sage: 10 % 3  #  pour des arguments entiers, % signifie mod, i.e., le reste dans la␣
↪division euclidienne
1
sage: 10/4
5/2
sage: 10//4   #  pour des arguments entiers, // renvoie le quotient dans la division␣
↪euclidienne
2
sage: 4 * (10 // 4) + 10 % 4 == 10
True
sage: 3^2*4 + 2%5
38
```

Le calcul d'une expression telle que `3^2*4 + 2%5` dépend de l'ordre dans lequel les opérations sont effectuées ; ceci est expliqué dans l'annexe *Priorité des opérateurs arithmétiques binaires* *Priorité des opérateurs arithmétiques binaires*.

Sage fournit également un grand nombre de fonctions mathématiques usuelles ; en voici quelques exemples choisis :

```
sage: sqrt(3.4)
1.84390889145858
sage: sin(5.135)
-0.912021158525540
sage: sin(pi/3)
1/2*sqrt(3)
```

Comme le montre le dernier de ces exemples, certaines expressions mathématiques renvoient des valeurs "exactes" plutôt que des approximations numériques. Pour obtenir une approximation numérique, on utilise au choix la fonction `n` ou la méthode `n` (chacun de ces noms possède le nom plus long `numerical_approx`, la fonction `N` est identique à `n`). Celles-ci acceptent, en argument optionnel, `prec`, qui indique le nombre de bits de précisions requis, et `digits`, qui indique le nombre de décimales demandées ; par défaut, il y a 53 bits de précision.

```
sage: exp(2)
e^2
sage: n(exp(2))
7.38905609893065
sage: sqrt(pi).numerical_approx()
1.77245385090552
sage: sin(10).n(digits=5)
-0.54402
sage: N(sin(10),digits=10)
-0.5440211109
sage: numerical_approx(pi, prec=200)
3.1415926535897932384626433832795028841971693993751058209749
```

Python est doté d'un typage dynamique. Ainsi la valeur à laquelle fait référence une variable est toujours associée à un type donné, mais une variable donnée peut contenir des valeurs de plusieurs types Python au sein d'une même portée :

```
sage: a = 5   # a est un entier
sage: type(a)
<type 'sage.rings.integer.Integer'>
sage: a = 5/3  # a est maintenant un rationnel...
sage: type(a)
<type 'sage.rings.rational.Rational'>
```

```
sage: a = 'hello'  # ...et maintenant une chaîne
sage: type(a)
<... 'str'>
```

Le langage de programmation C, qui est statiquement typé, est bien différent : une fois déclarée de type int, une variable ne peut contenir que des int au sein de sa portée.

2.2 Obtenir de l'aide

Sage est doté d'une importante documentation intégrée, accessible en tapant (par exemple) le nom d'une fonction ou d'une constante suivi d'un point d'interrogation :

```
sage: tan?
Type:        <class 'sage.calculus.calculus.Function_tan'>
Definition:  tan( [noargspec] )
Docstring:

    The tangent function

    EXAMPLES:
        sage: tan(pi)
        0
        sage: tan(3.1415)
        -0.0000926535900581913
        sage: tan(3.1415/4)
        0.999953674278156
        sage: tan(pi/4)
        1
        sage: tan(1/2)
        tan(1/2)
        sage: RR(tan(1/2))
        0.546302489843790
sage: log2?
Type:        <class 'sage.functions.constants.Log2'>
Definition:  log2( [noargspec] )
Docstring:

    The natural logarithm of the real number 2.

    EXAMPLES:
        sage: log2
        log2
        sage: float(log2)
        0.69314718055994529
        sage: RR(log2)
        0.693147180559945
        sage: R = RealField(200); R
        Real Field with 200 bits of precision
        sage: R(log2)
        0.69314718055994530941723212145817656807550013436025525412068
        sage: l = (1-log2)/(1+log2); l
        (1 - log(2))/(log(2) + 1)
        sage: R(l)
        0.18123221829928249948761381864650311423330609774776013488056
        sage: maxima(log2)
        log(2)
```

```
        sage: maxima(log2).float()
        .6931471805599453
        sage: gp(log2)
        0.69314718055994530941723212145             # 32-bit
        0.69314718055994530941723212145817656807    # 64-bit
sage: sudoku?
File:        sage/local/lib/python2.5/site-packages/sage/games/sudoku.py
Type:        <... 'function'>
Definition:  sudoku(A)
Docstring:

    Solve the 9x9 Sudoku puzzle defined by the matrix A.

    EXAMPLE:
        sage: A = matrix(ZZ,9,[5,0,0, 0,8,0, 0,4,9, 0,0,0, 5,0,0,
    0,3,0, 0,6,7, 3,0,0, 0,0,1, 1,5,0, 0,0,0, 0,0,0, 0,0,0, 2,0,8, 0,0,0,
    0,0,0, 0,0,0, 0,1,8, 7,0,0, 0,0,4, 1,5,0,   0,3,0, 0,0,2,
    0,0,0, 4,9,0, 0,5,0, 0,0,3])
        sage: A
        [5 0 0 0 8 0 0 4 9]
        [0 0 0 5 0 0 0 3 0]
        [0 6 7 3 0 0 0 0 1]
        [1 5 0 0 0 0 0 0 0]
        [0 0 0 2 0 8 0 0 0]
        [0 0 0 0 0 0 0 1 8]
        [7 0 0 0 0 4 1 5 0]
        [0 3 0 0 0 2 0 0 0]
        [4 9 0 0 5 0 0 0 3]
        sage: sudoku(A)
        [5 1 3 6 8 7 2 4 9]
        [8 4 9 5 2 1 6 3 7]
        [2 6 7 3 4 9 5 8 1]
        [1 5 8 4 6 3 9 7 2]
        [9 7 4 2 1 8 3 6 5]
        [3 2 6 7 9 5 4 1 8]
        [7 8 2 9 3 4 1 5 6]
        [6 3 5 1 7 2 8 9 4]
        [4 9 1 8 5 6 7 2 3]
```

Sage dispose aussi de la complétion de ligne de commande, accessible en tapant les quelques premières lettres du nom d'une fonction puis en appuyant sur la touche tabulation. Ainsi, si vous tapez `ta` suivi de `TAB`, Sage affichera `tachyon, tan, tanh, taylor`. C'est une façon commode de voir quels noms de fonctions et d'autres structures sont disponibles en Sage.

2.3 Fonctions, indentation et itération

Les définitions de fonctions en Sage sont introduites par la commande `def`, et la liste des noms des paramètres est suivie de deux points, comme dans :

```
sage: def is_even(n):
....:     return n%2 == 0
sage: is_even(2)
True
sage: is_even(3)
False
```

Remarque : suivant la version du *notebook* que vous utilisez, il est possible que vous voyez trois points `....:` au début de la deuxième ligne de l'exemple. Ne les entrez pas, ils servent uniquement à signaler que le code est indenté.

Les types des paramètres ne sont pas spécifiés dans la définition de la fonction. Il peut y avoir plusieurs paramètres, chacun accompagné optionnellement d'une valeur par défaut. Par exemple, si la valeur de `divisor` n'est pas donnée lors d'un appel à la fonction ci-dessous, la valeur par défaut `divisor=2` est utilisée.

```
sage: def is_divisible_by(number, divisor=2):
....:     return number%divisor == 0
sage: is_divisible_by(6,2)
True
sage: is_divisible_by(6)
True
sage: is_divisible_by(6, 5)
False
```

Il est possible de spécifier un ou plusieurs des paramètres par leur nom lors de l'appel de la fonction ; dans ce cas, les paramètres nommés peuvent apparaître dans n'importe quel ordre :

```
sage: is_divisible_by(6, divisor=5)
False
sage: is_divisible_by(divisor=2, number=6)
True
```

En Python, contrairement à de nombreux autres langages, les blocs de code ne sont pas délimités par des accolades ou des mots-clés de début et de fin de bloc. Au lieu de cela, la structure des blocs est donnée par l'indentation, qui doit être la même dans tout le bloc. Par exemple, le code suivant déclenche une erreur de syntaxe parce que l'instruction `return` n'est pas au même niveau d'indentation que les lignes précédentes.

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

Une fois l'indentation corrigée, l'exemple fonctionne :

```
sage: def even(n):
....:     v = []
....:     for i in range(3,n):
....:         if i % 2 == 0:
....:             v.append(i)
....:     return v
sage: even(10)
[4, 6, 8]
```

Il n'y a pas besoin de placer des points-virgules en fin de ligne ; une instruction est en général terminée par un passage à la ligne. En revanche, il est possible de placer plusieurs instructions sur la même ligne en les séparant par des points-virgules :

```
sage: a = 5; b = a + 3; c = b^2; c
64
```

Pour continuer une instruction sur la ligne suivante, placez une barre oblique inverse en fin de ligne :

```
sage: 2 + \
....:     3
5
```

Pour compter en Sage, utilisez une boucle dont la variable d'itération parcourt une séquence d'entiers. Par exemple, la première ligne ci-dessous a exactement le même effet que `for(i=0; i<3; i++)` en C++ ou en Java :

```
sage: for i in range(3):
....:     print(i)
0
1
2
```

La première ligne ci-dessous correspond à `for(i=2;i<5;i++)`.

```
sage: for i in range(2,5):
....:     print(i)
2
3
4
```

Le troisième paramètre contrôle le pas de l'itération. Ainsi, ce qui suit est équivalent à `for(i=1;i<6;i+=2)`.

```
sage: for i in range(1,6,2):
....:     print(i)
1
3
5
```

Vous souhaiterez peut-être regrouper dans un joli tableau les résultats numériques que vous aurez calculés avec Sage. Une façon de faire commode utilise les chaînes de format. Ici, nous affichons une table des carrés et des cubes en trois colonnes, chacune d'une largeur de six caractères.

```
sage: for i in range(5):
....:     print('%6s %6s %6s' % (i, i^2, i^3))
     0      0      0
     1      1      1
     2      4      8
     3      9     27
     4     16     64
```

La structure de données de base de Sage est la liste, qui est — comme son nom l'indique — une liste d'objets arbitraires. Par exemple, la commande `range` que nous avons utilisée plus haut crée en fait une liste (en python 2) :

```
sage: range(2,10)    # optional - python2
[2, 3, 4, 5, 6, 7, 8, 9]
sage: list(range(2,10))    # optional - python3
[2, 3, 4, 5, 6, 7, 8, 9]
```

Voici un exemple plus compliqué de liste :

```
sage: v = [1, "hello", 2/3, sin(x^3)]
sage: v
[1, 'hello', 2/3, sin(x^3)]
```

Comme dans de nombreux langages de programmation, les listes sont indexées à partir de 0.

```
sage: v[0]
1
sage: v[3]
sin(x^3)
```

La fonction `len(v)` donne la longueur de `v`... :`v.append(obj)` ajoute un nouvel objet à la fin de `v` ; et `del v[i]` supprime l'élément d'indice `i` de `v`.

```
sage: len(v)
4
sage: v.append(1.5)
sage: v
[1, 'hello', 2/3, sin(x^3), 1.50000000000000]
sage: del v[1]
sage: v
[1, 2/3, sin(x^3), 1.50000000000000]
```

Une autre structure de données importante est le dictionnaire (ou tableau associatif). Un dictionnaire fonctionne comme une liste, à ceci près que les indices peuvent être presque n'importe quels objets (les objets mutables sont interdits) :

```
sage: d = {'hi':-2,  3/8:pi,   e:pi}
sage: d['hi']
-2
sage: d[e]
pi
```

Vous pouvez définir de nouveaux types de données en utilisant les classes. Encapsuler les objets mathématiques dans des classes représente une technique puissante qui peut vous aider à simplifier et organiser vos programmes Sage. Dans l'exemple suivant, nous définissons une classe qui représente la liste des entiers impairs strictement positifs jusqu'à n. Cette classe dérive du type interne `list`.

```
sage: class Evens(list):
....:     def __init__(self, n):
....:         self.n = n
....:         list.__init__(self, range(2, n+1, 2))
....:     def __repr__(self):
....:         return "Even positive numbers up to n."
```

La méthode `__init__` est appelée à la création de l'objet pour l'initialiser ; la méthode `__repr__` affiche l'objet. À la seconde ligne de la méthode `__init__`, nous appelons le constructeur de la classe `list`. Pour créer un objet de classe `Evens`, nous procédons ensuite comme suit :

```
sage: e = Evens(10)
sage: e
Even positive numbers up to n.
```

Notez que `e` s'affiche en utilisant la méthode `__repr__` que nous avons définie plus haut. Pour voir la liste de nombres sous-jacente, on utilise la fonction `list` :

```
sage: list(e)
[2, 4, 6, 8, 10]
```

Il est aussi possible d'accéder à l'attribut `n`, ou encore d'utiliser `e` en tant que liste.

```
sage: e.n
10
```

```
sage: e[2]
6
```

2.4 Algèbre de base et calcul infinitésimal

Sage peut accomplir divers calculs d'algèbre et d'analyse de base : par exemple, trouver les solutions d'équations, dériver, intégrer, calculer des transformées de Laplace. Voir la documentation Sage Constructions pour plus d'exemples.

2.4.1 Résolution d'équations

La fonction `solve` résout des équations. Pour l'utiliser, il convient de spécifier d'abord les variables. Les arguments de `solve` sont alors une équation (ou un système d'équations) suivie des variables à résoudre.

```
sage: x = var('x')
sage: solve(x^2 + 3*x + 2, x)
[x == -2, x == -1]
```

On peut résoudre une équation en une variable en fonction des autres :

```
sage: x, b, c = var('x b c')
sage: solve([x^2 + b*x + c == 0],x)
[x == -1/2*b - 1/2*sqrt(b^2 - 4*c), x == -1/2*b + 1/2*sqrt(b^2 - 4*c)]
```

On peut également résoudre un système à plusieurs variables :

```
sage: x, y = var('x, y')
sage: solve([x+y==6, x-y==4], x, y)
[[x == 5, y == 1]]
```

L'exemple suivant, qui utilise Sage pour la résolution d'un système d'équations non-linéaires, a été proposé par Jason Grout. D'abord, on résout le système de façon symbolique :

```
sage: var('x y p q')
(x, y, p, q)
sage: eq1 = p+q==9
sage: eq2 = q*y+p*x==-6
sage: eq3 = q*y^2+p*x^2==24
sage: solve([eq1,eq2,eq3,p==1],p,q,x,y)
[[p == 1, q == 8, x == -4/3*sqrt(10) - 2/3, y == 1/6*sqrt(10) - 2/3], [p == 1, q == 8,
↪ x == 4/3*sqrt(10) - 2/3, y == -1/6*sqrt(10) - 2/3]]
```

Pour une résolution numérique, on peut utiliser à la place :

```
sage: solns = solve([eq1,eq2,eq3,p==1],p,q,x,y, solution_dict=True)
sage: [[s[p].n(30), s[q].n(30), s[x].n(30), s[y].n(30)] for s in solns]
[[1.0000000, 8.0000000, -4.8830369, -0.13962039],
 [1.0000000, 8.0000000, 3.5497035, -1.1937129]]
```

(La fonction `n` affiche une approximation numérique ; son argument indique le nombre de bits de précision.)

2.4.2 Dérivation, intégration, etc.

Sage est capable de dériver et d'intégrer de nombreuses fonctions. Par exemple, pour dériver $\sin(u)$ par rapport à u, on procède comme suit :

```
sage: u = var('u')
sage: diff(sin(u), u)
cos(u)
```

Pour calculer la dérivée quatrième de $\sin(x^2)$:

```
sage: diff(sin(x^2), x, 4)
16*x^4*sin(x^2) - 48*x^2*cos(x^2) - 12*sin(x^2)
```

Pour calculer la dérivée partielle de $x^2 + 17y^2$ par rapport à x et y respectivement :

```
sage: x, y = var('x,y')
sage: f = x^2 + 17*y^2
sage: f.diff(x)
2*x
sage: f.diff(y)
34*y
```

Passons aux primitives et intégrales. Pour calculer $\int x\sin(x^2)\,dx$ et $\int_0^1 \frac{x}{x^2+1}\,dx$

```
sage: integral(x*sin(x^2), x)
-1/2*cos(x^2)
sage: integral(x/(x^2+1), x, 0, 1)
1/2*log(2)
```

Pour calculer la décomposition en éléments simples de $\frac{1}{x^2-1}$:

```
sage: f = 1/((1+x)*(x-1))
sage: f.partial_fraction(x)
-1/2/(x + 1) + 1/2/(x - 1)
```

2.4.3 Résolution des équations différentielles

On peut utiliser Sage pour étudier les équations différentielles ordinaires. Pour résoudre l'équation $x' + x - 1 = 0$:

```
sage: t = var('t')     # on définit une variable t
sage: function('x')(t)    # on déclare x fonction de cette variable
x(t)
sage: DE = lambda y: diff(y,t) + y - 1
sage: desolve(DE(x(t)), [x(t),t])
(_C + e^t)*e^(-t)
```

Ceci utilise l'interface de Sage vers Maxima *[Max]*, aussi il se peut que la sortie diffère un peu des sorties habituelles de Sage. Dans notre cas, le résultat indique que la solution générale à l'équation différentielle est $x(t) = e^{-t}(e^t + C)$.

Il est aussi possible de calculer des transformées de Laplace. La transformée de Laplace de $t^2e^t - \sin(t)$ s'obtient comme suit :

```
sage: s = var("s")
sage: t = var("t")
sage: f = t^2*exp(t) - sin(t)
```

```
sage: f.laplace(t,s)
-1/(s^2 + 1) + 2/(s - 1)^3
```

Voici un exemple plus élaboré. L'élongation à partir du point d'équilibre de ressorts couplés attachés à gauche à un mur

```
|------\/\/\/\/\---|masse1|----\/\/\/\/\/----|masse2|
        ressort1                 ressort2
```

est modélisée par le système d'équations différentielles d'ordre 2

$$m_1 x_1'' + (k_1 + k_2)x_1 - k_2 x_2 = 0 m_2 x_2'' + k_2(x_2 - x_1) = 0,$$

où m_i est la masse de l'objet i, x_i est l'élongation à partir du point d'équilibre de la masse i, et k_i est la constante de raideur du ressort i.

Exemple : Utiliser Sage pour résoudre le problème ci-dessus avec $m_1 = 2$, $m_2 = 1$, $k_1 = 4$, $k_2 = 2$, $x_1(0) = 3$, $x_1'(0) = 0$, $x_2(0) = 3$, $x_2'(0) = 0$.

Solution : Considérons la transformée de Laplace de la première équation (avec les notations $x = x_1$, $y = x_2$) :

```
sage: de1 = maxima("2*diff(x(t),t, 2) + 6*x(t) - 2*y(t)")
sage: lde1 = de1.laplace("t","s"); lde1
2*((-%at('diff(x(t),t,1),t=0))+s^2*'laplace(x(t),t,s)-x(0)*s)-2*'laplace(y(t),t,s)+6*
↪'laplace(x(t),t,s)
```

La réponse n'est pas très lisible, mais elle signifie que

$$-2x'(0) + 2s^2 \cdot X(s) - 2sx(0) - 2Y(s) + 6X(s) = 0$$

(où la transformée de Laplace d'une fonction notée par une lettre minuscule telle que $x(t)$ est désignée par la majuscule correspondante $X(s)$). Considérons la transformée de Laplace de la seconde équation :

```
sage: de2 = maxima("diff(y(t),t, 2) + 2*y(t) - 2*x(t)")
sage: lde2 = de2.laplace("t","s"); lde2
(-%at('diff(y(t),t,1),t=0))+s^2*'laplace(y(t),t,s)+2*'laplace(y(t),t,s)-2*
↪'laplace(x(t),t,s)-y(0)*s
```

Ceci signifie

$$-Y'(0) + s^2 Y(s) + 2Y(s) - 2X(s) - sy(0) = 0.$$

Injectons les conditions initiales pour $x(0)$, $x'(0)$, $y(0)$ et $y'(0)$ et résolvons les deux équations qui en résultent :

```
sage: var('s X Y')
(s, X, Y)
sage: eqns = [(2*s^2+6)*X-2*Y == 6*s, -2*X +(s^2+2)*Y == 3*s]
sage: solve(eqns, X,Y)
[[X == 3*(s^3 + 3*s)/(s^4 + 5*s^2 + 4),
  Y == 3*(s^3 + 5*s)/(s^4 + 5*s^2 + 4)]]
```

À présent, prenons la transformée de Laplace inverse pour obtenir la réponse :

```
sage: var('s t')
(s, t)
sage: inverse_laplace((3*s^3 + 9*s)/(s^4 + 5*s^2 + 4),s,t)
cos(2*t) + 2*cos(t)
sage: inverse_laplace((3*s^3 + 15*s)/(s^4 + 5*s^2 + 4),s,t)
-cos(2*t) + 4*cos(t)
```

Par conséquent, la solution est

$$x_1(t) = \cos(2t) + 2\cos(t), \quad x_2(t) = 4\cos(t) - \cos(2t).$$

On peut en tracer le graphe paramétrique en utilisant

```
sage: t = var('t')
sage: P = parametric_plot((cos(2*t) + 2*cos(t), 4*cos(t) - cos(2*t) ),
....:     (t, 0, 2*pi), rgbcolor=hue(0.9))
sage: show(P)
```

Les coordonnées individuelles peuvent être tracées en utilisant

```
sage: t = var('t')
sage: p1 = plot(cos(2*t) + 2*cos(t), (t, 0, 2*pi), rgbcolor=hue(0.3))
sage: p2 = plot(4*cos(t) - cos(2*t), (t, 0, 2*pi), rgbcolor=hue(0.6))
sage: show(p1 + p2)
```

Les fonctions de tracé de graphes sont décrites dans la section *Graphiques* de ce tutoriel. On pourra aussi consulter *[NagleEtAl2004]*, §5.5 pour plus d'informations sur les équations différentielles.

2.4.4 Méthode d'Euler pour les systèmes d'équations différentielles

Dans l'exemple suivant, nous illustrons la méthode d'Euler pour des équations différentielles ordinaires d'ordre un et deux. Rappelons d'abord le principe de la méthode pour les équations du premier ordre. Etant donné un problème donné avec une valeur initiale sous la forme

$$y' = f(x, y), \quad y(a) = c,$$

nous cherchons une valeur approchée de la solution au point $x = b$ avec $b > a$.

Rappelons que par définition de la dérivée

$$y'(x) \approx \frac{y(x+h) - y(x)}{h},$$

où $h > 0$ est fixé et petit. Ceci, combiné à l'équation différentielle, donne $f(x, y(x)) \approx \frac{y(x+h)-y(x)}{h}$. Aussi $y(x+h)$ s'écrit :

$$y(x+h) \approx y(x) + h \cdot f(x, y(x)).$$

Si nous notons $h \cdot f(x, y(x))$ le « terme de correction » (faute d'un terme plus approprié), et si nous appelons $y(x)$ « l'ancienne valeur de y » et $y(x+h)$ la « nouvelle valeur de y », cette approximation se réécrit

$$y_{nouveau} \approx y_{ancien} + h \cdot f(x, y_{ancien}).$$

Divisions l'intervalle entre a et b en n pas, si bien que $h = \frac{b-a}{n}$. Nous pouvons alors remplir un tableau avec les informations utilisées dans la méthode.

x	y	$h \cdot f(x, y)$
a	c	$h \cdot f(a, c)$
$a + h$	$c + h \cdot f(a, c)$	$\ldots$
$a + 2h$	$\ldots$	
$\ldots$		
$b = a + nh$	???	$\ldots$

Le but est est de remplir tous les trous du tableau, ligne après ligne, jusqu'à atteindre le coefficient « ??? », qui est l'approximation de $y(b)$ au sens de la méthode d'Euler.

L'idée est la même pour les systèmes d'équations différentielles.

Exemple : Rechercher une approximation numérique de $z(t)$ en $t = 1$ en utilisant 4 étapes de la méthode d'Euler, où $z'' + tz' + z = 0$, $z(0) = 1$, $z'(0) = 0$.

Il nous faut réduire l'équation différentielle d'ordre 2 à un système de deux équations différentielles d'ordre 1 (en posant $x = z$, $y = z'$) et appliquer la méthode d'Euler :

```
sage: t,x,y = PolynomialRing(RealField(10),3,"txy").gens()
sage: f = y; g = -x - y * t
sage: eulers_method_2x2(f,g, 0, 1, 0, 1/4, 1)
      t                x             h*f(t,x,y)                y        h*g(t,x,y)
      0                1                   0.00                0            -0.25
    1/4              1.0                 -0.062            -0.25            -0.23
    1/2             0.94                  -0.12            -0.48            -0.17
    3/4             0.82                  -0.16            -0.66           -0.081
      1             0.65                  -0.18            -0.74            0.022
```

On en déduit $z(1) \approx 0.65$.

On peut également tracer le graphe des points (x, y) pour obtenir une image approchée de la courbe. La fonction `eulers_method_2x2_plot` réalise cela ; pour l'utiliser, il faut définir les fonctions f et g qui prennent un argument à trois coordonnées : (t, x, y).

```
sage: f = lambda z: z[2]         # f(t,x,y) = y
sage: g = lambda z: -sin(z[1])   # g(t,x,y) = -sin(x)
sage: P = eulers_method_2x2_plot(f,g, 0.0, 0.75, 0.0, 0.1, 1.0)
```

Arrivé à ce point, `P` conserve en mémoire deux graphiques : `P[0]`, le graphe de x en fonction de t, et `P[1]`, le graphique de y par rapport à t. On peut tracer les deux graphiques simultanément par :

```
sage: show(P[0] + P[1])
```

(Pour plus d'information sur le tracé de graphiques, voir *Graphiques*.)

2.4.5 Fonctions spéciales

Plusieurs familles de polynômes orthogonaux et fonctions spéciales sont implémentées via PARI *[GAP]* et Maxima *[Max]*. Ces fonctions sont documentées dans les sections correspondantes (*Orthogonal polynomials* et *Special functions*, respectively) du manuel de référence de Sage (*Sage reference manual*).

```
sage: x = polygen(QQ, 'x')
sage: chebyshev_U(2,x)
4*x^2 - 1
sage: bessel_I(1,1).n(250)
0.56515910399248502720769602760986330732889962162109200948029448947925564096
sage: bessel_I(1,1).n()
0.565159103992485
sage: bessel_I(2,1.1).n()
0.167089499251049
```

Pour l'instant, ces fonctions n'ont été adaptées à Sage que pour une utilisation numérique. Pour faire du calcul formel, il faut utiliser l'interface Maxima directement, comme le présente l'exemple suivant :

```
sage: maxima.eval("f:bessel_y(v, w)")
'bessel_y(v,w)'
sage: maxima.eval("diff(f,w)")
'(bessel_y(v-1,w)-bessel_y(v+1,w))/2'
```

2.5 Graphiques

Sage peut produire des graphiques en deux ou trois dimensions.

2.5.1 Graphiques en deux dimensions

En deux dimensions, Sage est capable de tracer des cercles, des droites, des polygones, des graphes de fonctions en coordonnées cartésiennes, des graphes en coordonnées polaires, des lignes de niveau et des représentations de champs de vecteurs. Nous présentons quelques exemples de ces objets ici. Pour plus d'exemples de graphiques avec Sage, on consultera *Résolution des équations différentielles*, *Maxima* et aussi la documentation Sage Constructions

La commande suivante produit un cercle jaune de rayon 1 centré à l'origine :

```
sage: circle((0,0), 1, rgbcolor=(1,1,0))
Graphics object consisting of 1 graphics primitive
```

Il est également possible de produire un disque plein :

```
sage: circle((0,0), 1, rgbcolor=(1,1,0), fill=True)
Graphics object consisting of 1 graphics primitive
```

Il est aussi possible de créer un cercle en l'affectant à une variable ; ceci ne provoque pas son affichage.

```
sage: c = circle((0,0), 1, rgbcolor=(1,1,0))
```

Pour l'afficher, on utilise `c.show()` ou `show(c)`, comme suit :

```
sage: c.show()
```

Alternativement, l'évaluation de `c.save('filename.png')` enregistre le graphique dans le fichier spécifié.

Toutefois, ces « cercles » ressemblent plus à des ellipses qu'à des cercles, puisque les axes possèdent des échelles différentes. On peut arranger ceci :

```
sage: c.show(aspect_ratio=1)
```

La commande `show(c, aspect_ratio=1)` produit le même résultat. On peut enregistrer l'image avec cette option par la commande `c.save('filename.png', aspect_ratio=1)`.

Il est très facile de tracer le graphique de fonctions de base :

```
sage: plot(cos, (-5,5))
Graphics object consisting of 1 graphics primitive
```

En spécifiant un nom de variable, on peut aussi créer des graphes paramétriques :

```
sage: x = var('x')
sage: parametric_plot((cos(x),sin(x)^3),(x,0,2*pi),rgbcolor=hue(0.6))
Graphics object consisting of 1 graphics primitive
```

Différents graphiques peuvent se combiner sur une même image :

```
sage: x = var('x')
sage: p1 = parametric_plot((cos(x),sin(x)),(x,0,2*pi),rgbcolor=hue(0.2))
sage: p2 = parametric_plot((cos(x),sin(x)^2),(x,0,2*pi),rgbcolor=hue(0.4))
sage: p3 = parametric_plot((cos(x),sin(x)^3),(x,0,2*pi),rgbcolor=hue(0.6))
sage: show(p1+p2+p3, axes=false)
```

Une manière commode de tracer des formes pleines est de préparer une liste de points (`L` dans l'exemple ci-dessous) puis d'utiliser la commande `polygon` pour tracer la forme pleine dont le bord est formé par ces points. Par example, voici un deltoïde vert :

```
sage: L = [[-1+cos(pi*i/100)*(1+cos(pi*i/100)),
....:      2*sin(pi*i/100)*(1-cos(pi*i/100))] for i in range(200)]
sage: polygon(L, rgbcolor=(1/8,3/4,1/2))
Graphics object consisting of 1 graphics primitive
```

Pour visualiser le graphique en masquant les axes, tapez `show(p, axes=false)`.

On peut ajouter un texte à un graphique :

```
sage: L = [[6*cos(pi*i/100)+5*cos((6/2)*pi*i/100),
....:      6*sin(pi*i/100)-5*sin((6/2)*pi*i/100)] for i in range(200)]
sage: p = polygon(L, rgbcolor=(1/8,1/4,1/2))
sage: t = text("hypotrochoid", (5,4), rgbcolor=(1,0,0))
sage: show(p+t)
```

En cours d'analyse, les professeurs font souvent le dessin suivant au tableau : non pas une mais plusieurs branches de la fonction arcsin, autrement dit, le graphe d'équation $y = \sin(x)$ pour x entre -2π et 2π, renversé par symétrie par rapport à la première bissectrice des axes. La commande Sage suivante réalise cela :

```
sage: v = [(sin(x),x) for x in srange(-2*float(pi),2*float(pi),0.1)]
sage: line(v)
Graphics object consisting of 1 graphics primitive
```

Comme les valeurs prises par la fonction tangente ne sont pas bornées, pour utiliser la même astuce pour représenter la fonction arctangente, il faut préciser les bornes de la coordonnée x :

```
sage: v = [(tan(x),x) for x in srange(-2*float(pi),2*float(pi),0.01)]
sage: show(line(v), xmin=-20, xmax=20)
```

Sage sait aussi tracer des graphiques en coordonnées polaires, des lignes de niveau et (pour certains types de fonctions) des champs de vecteurs. Voici un exemple de lignes de niveau :

```
sage: f = lambda x,y: cos(x*y)
sage: contour_plot(f, (-4, 4), (-4, 4))
Graphics object consisting of 1 graphics primitive
```

2.5.2 Graphiques en trois dimensions

Sage produit des graphes en trois dimensions en utilisant le package open source appelé *[Jmol]*. En voici quelques exemples :

Le parapluie de Whitney tracé en jaune http://en.wikipedia.org/wiki/Whitney_umbrella :

```
sage: u, v = var('u,v')
sage: fx = u*v
sage: fy = u
sage: fz = v^2
sage: parametric_plot3d([fx, fy, fz], (u, -1, 1), (v, -1, 1),
....:   frame=False, color="yellow")
Graphics3d Object
```

Une fois évaluée la commande `parametric_plot3d`, qui affiche le graphique, il est possible de cliquer et de le tirer pour faire pivoter la figure.

Le bonnet croisé (cf. http://en.wikipedia.org/wiki/Cross-cap ou http://www.mathcurve.com/surfaces/bonnetcroise/bonnetcroise.shtml) :

```
sage: u, v = var('u,v')
sage: fx = (1+cos(v))*cos(u)
sage: fy = (1+cos(v))*sin(u)
sage: fz = -tanh((2/3)*(u-pi))*sin(v)
sage: parametric_plot3d([fx, fy, fz], (u, 0, 2*pi), (v, 0, 2*pi),
....:   frame=False, color="red")
Graphics3d Object
```

Un tore tordu :

```
sage: u, v = var('u,v')
sage: fx = (3+sin(v)+cos(u))*cos(2*v)
sage: fy = (3+sin(v)+cos(u))*sin(2*v)
sage: fz = sin(u)+2*cos(v)
sage: parametric_plot3d([fx, fy, fz], (u, 0, 2*pi), (v, 0, 2*pi),
....:   frame=False, color="red")
Graphics3d Object
```

2.6 Problèmes fréquents concernant les fonctions

La définition de fonctions, par exemple pour calculer leurs dérivées ou tracer leurs courbes représentatives, donne lieu à un certain nombre de confusions. Le but de cette section est de clarifier quelques points à l'origine de ces confusions.

Il y a plusieurs façons de définir un objet que l'on peut légitimement appeler « fonction ».

1. Définir une fonction Python, comme expliqué dans la section *Fonctions, indentation et itération*. Les fonctions Python peuvent être utilisées pour tracer des courbes, mais pas dérivées ou intégrées symboliquement :

```
sage: def f(z): return z^2
sage: type(f)
<... 'function'>
sage: f(3)
9
sage: plot(f, 0, 2)
Graphics object consisting of 1 graphics primitive
```

Remarquez la syntaxe de la dernière ligne. Écrire plutôt `plot(f(z), 0, 2)` provoquerait une erreur : en effet, le `z` qui apparaît dans la définition de `f` est une variable muette qui n'a pas de sens en dehors de la définition. Un simple `f(z)` déclenche la même erreur. En l'occurrence, faire de `z` une variable symbolique comme dans l'exemple ci-dessous fonctionne, mais cette façon de faire soulève d'autres problèmes (voir le point 4 ci-dessous), et il vaut mieux s'abstenir de l'utiliser.

```
sage: var('z')    # on définit z comme variable symbolique
z
sage: f(z)
z^2
sage: plot(f(z), 0, 2)
Graphics object consisting of 1 graphics primitive
```

L'appel de fonction `f(z)` renvoie ici l'expression symbolique `z^2`, qui est alors utilisée par la fonction `plot`.

2. Définir une expression symbolique fonctionnelle (« appelable »). Une telle expression représente une fonction dont on peut tracer le graphe, et que l'on peut aussi dériver ou intégrer symboliquement

```
sage: g(x) = x^2
sage: g         # g envoie x sur x^2
x |--> x^2
sage: g(3)
9
sage: Dg = g.derivative(); Dg
x |--> 2*x
sage: Dg(3)
6
sage: type(g)
<type 'sage.symbolic.expression.Expression'>
sage: plot(g, 0, 2)
Graphics object consisting of 1 graphics primitive
```

Notez que, si `g` est une expression symbolique fonctionnelle (`x |--> x^2`), l'objet `g(x)` (`x^2`) est d'une nature un peu différente. Les expressions comme `g(x)` peuvent aussi être tracées, dérivées, intégrées, etc., avec cependant quelques difficultés illustrées dans le point 5 ci-dessous.

```
sage: g(x)
x^2
sage: type(g(x))
<type 'sage.symbolic.expression.Expression'>
sage: g(x).derivative()
2*x
sage: plot(g(x), 0, 2)
Graphics object consisting of 1 graphics primitive
```

3. Utiliser une fonction usuelle prédéfinie de Sage. Celles-ci peuvent servir à tracer des courbes, et, indirectement, être dérivées ou intégrées

```
sage: type(sin)
<class 'sage.functions.trig.Function_sin'>
sage: plot(sin, 0, 2)
Graphics object consisting of 1 graphics primitive
sage: type(sin(x))
<type 'sage.symbolic.expression.Expression'>
sage: plot(sin(x), 0, 2)
Graphics object consisting of 1 graphics primitive
```

Il n'est pas possible de dériver la fonction `sin` tout court pour obtenir `cos`

```
sage: f = sin
sage: f.derivative()
Traceback (most recent call last):
...
AttributeError: ...
```

Une possibilité est de remplacer `f = sin` par `f = sin(x)`, mais il est généralement préférable de définir une expression symbolique fonctionnelle `f(x) = sin(x)`

```
sage: S(x) = sin(x)
sage: S.derivative()
x |--> cos(x)
```

Examinons maintenant quelques problèmes fréquents.

4. Évaluation accidentelle

```
sage: def h(x):
....:     if x < 2:
....:         return 0
....:     else:
....:         return x-2
```

Problème : `plot(h(x), 0, 4)` trace la droite $y = x - 2$, et non pas la fonction affine par morceaux définie par `h`. Pourquoi ? Lors de l'exécution, `plot(h(x), 0, 4)` évalue d'abord `h(x)` : la fonction Python `h` est appelée avec le paramètre `x`, et la condition `x < 2` est donc évaluée.

```
sage: type(x < 2)
<type 'sage.symbolic.expression.Expression'>
```

Or, l'évaluation d'une inégalité symbolique renvoie `False` quand la condition n'est pas clairement vraie. Ainsi, `h(x)` s'évalue en `x - 2`, et c'est cette expression-là qui est finalement tracée.

Solution : Il ne faut pas utiliser `plot(h(x), 0, 4)`, mais plutôt

```
sage: def h(x):
....:     if x < 2:
....:         return 0
....:     else:
....:         return x-2
sage: plot(h, 0, 4)
Graphics object consisting of 1 graphics primitive
```

5. Constante plutôt que fonction

```
sage: f = x
sage: g = f.derivative()
sage: g
1
```

Problème : `g(3)` déclenche une erreur avec le message « ValueError : the number of arguments must be less than or equal to 0 ».

```
sage: type(f)
<type 'sage.symbolic.expression.Expression'>
sage: type(g)
<type 'sage.symbolic.expression.Expression'>
```

En effet, `g` n'est pas une fonction, mais une constante, sans variable en laquelle on peut l'évaluer.

Solution : il y a plusieurs possibilités.

— Définir `f` comme une expression symbolique fonctionnelle

```
sage: f(x) = x          # au lieu de 'f = x'
sage: g = f.derivative()
```

```
sage: g
x |--> 1
sage: g(3)
1
sage: type(g)
<type 'sage.symbolic.expression.Expression'>
```

— Ou, sans changer la définition de `f`, définir `g` comme une expression symbolique fonctionnelle

```
sage: f = x
sage: g(x) = f.derivative()  # au lieu de 'g = f.derivative()'
sage: g
x |--> 1
sage: g(3)
1
sage: type(g)
<type 'sage.symbolic.expression.Expression'>
```

— Ou encore, avec `f` et `g` définies comme dans l'exemple de départ, donner explicitement la variable à remplacer par sa valeur

```
sage: f = x
sage: g = f.derivative()
sage: g
1
sage: g(x=3)     # au lieu de  'g(3)'
1
```

Nous terminons en mettant encore une fois en évidence la différence entre les dérivées des expressions `f` définies par `f = x` et par `f(x) = x`

```
sage: f(x) = x
sage: g = f.derivative()
sage: g.variables()  # variables apparaissant dans g
()
sage: g.arguments()  # paramètres auxquels on peut donner une valeur dans g
(x,)
sage: f = x
sage: h = f.derivative()
sage: h.variables()
()
sage: h.arguments()
()
```

Comme l'illustre cet exemple, `h` n'accepte pas de paramètres. C'est pour cela que `h(3)` déclenche une erreur.

2.7 Anneaux de base

Nous illustrons la prise en main de quelques anneaux de base avec Sage. Par exemple, `RationalField()` ou `QQ` désigneront dans ce qui suit au corps des nombres rationnels :

```
sage: RationalField()
Rational Field
sage: QQ
Rational Field
sage: 1/2 in QQ
True
```

Le nombre décimal `1.2` est considéré comme un élément de `QQ`, puisqu'il existe une application de coercition entre les réels et les rationnels :

```
sage: 1.2 in QQ
True
```

Néanmoins, il n'y a pas d'application de coercition entre le corps fini à 3 éléments et les rationnels :

```
sage: c = GF(3)(1)    # c est l'élément 1 du corps fini à 3 éléments
sage: c in QQ
False
```

De même, bien entendu, la constante symbolique π n'appartient pas aux rationnels :

```
sage: pi in QQ
False
```

Le symbole `I` représente la racine carrée de -1 ; `i` est synonyme de `I`. Bien entendu, `I` n'appartient pas aux rationnels :

```
sage: i  # i^2 = -1
I
sage: i in QQ
False
```

À ce propos, d'autres anneaux sont prédéfinis en Sage : l'anneau des entiers relatifs `ZZ`, celui des nombres réels `RR` et celui des nombres complexes `CC`. Les anneaux de polynômes sont décrits dans *Polynômes*.

Passons maintenant à quelques éléments d'arithmétique.

```
sage: a, b = 4/3, 2/3
sage: a + b
2
sage: 2*b == a
True
sage: parent(2/3)
Rational Field
sage: parent(4/2)
Rational Field
sage: 2/3 + 0.1       # coercition automatique avant addition
0.766666666666667
sage: 0.1 + 2/3       # les règles de coercition sont symétriques en SAGE
0.766666666666667
```

Il y a une subtilité dans la définition des nombres complexes. Comme mentionné ci-dessus, le symbole `i` représente une racine carrée de -1, mais il s'agit d'une racine carrée *formelle* de -1. L'appel `CC(i)` renvoie la racine carrée complexe de -1.

```
sage: i = CC(i)       # nombre complexe en virgule flottante
sage: z = a + b*i
sage: z
1.33333333333333 + 0.666666666666667*I
sage: z.imag()        # partie imaginaire
0.666666666666667
sage: z.real() == a   # coercition automatique avant comparaison
True
sage: QQ(11.1)
111/10
```

2.8 Polynômes

Dans cette partie, nous expliquons comment créer et utiliser des polynômes avec Sage.

2.8.1 Polynômes univariés

Il existe trois façons de créer des anneaux de polynômes.

```
sage: R = PolynomialRing(QQ, 't')
sage: R
Univariate Polynomial Ring in t over Rational Field
```

Ceci crée un anneau de polynômes et indique à Sage d'utiliser (la chaîne de caractère) "t" comme indéterminée lors de l'affichage à l'écran. Toutefois, ceci ne définit pas le symbole `t` pour son utilisation dans Sage. Aussi, il n'est pas possible de l'utiliser pour saisir un polynôme (comme $t^2 + 1$) qui appartient à `R`.

Une deuxième manière de procéder est

```
sage: S = QQ['t']
sage: S == R
True
```

Ceci a les mêmes effets en ce qui concerne `t`.

Une troisième manière de procéder, très pratique, consiste à entrer

```
sage: R.<t> = PolynomialRing(QQ)
```

ou

```
sage: R.<t> = QQ['t']
```

ou même

```
sage: R.<t> = QQ[]
```

L'effet secondaire de ces dernières instructions est de définir la variable `t` comme l'indéterminée de l'anneau de polynômes. Ceci permet de construire très aisément des éléments de `R`, comme décrit ci-après. (Noter que cette troisième manière est très semblable à la notation par constructeur de Magma et que, de même que dans Magma, ceci peut servir pour une très large classe d'objets.)

```
sage: poly = (t+1) * (t+2); poly
t^2 + 3*t + 2
sage: poly in R
True
```

Quelle que soit la méthode utilisée pour définir l'anneau de polynômes, on récupère l'indéterminée comme le 0-ième générateur :

```
sage: R = PolynomialRing(QQ, 't')
sage: t = R.0
sage: t in R
True
```

Notez que les nombres complexes peuvent être construits de façon similaire : les nombres complexes peuvent être vus comme engendrés sur les réels par le symbole `i`. Aussi, on dispose de :

```
sage: CC
Complex Field with 53 bits of precision
sage: CC.0  # 0ième générateur CC
1.00000000000000*I
```

Pour un anneau de polynômes, on peut obtenir à la fois l'anneau et son générateur ou juste le générateur au moment de la création de l'anneau comme suit :

```
sage: R, t = QQ['t'].objgen()
sage: t    = QQ['t'].gen()
sage: R, t = objgen(QQ['t'])
sage: t    = gen(QQ['t'])
```

Finalement, on peut faire de l'arithmétique dans $\mathbf{Q}[t]$.

```
sage: R, t = QQ['t'].objgen()
sage: f = 2*t^7 + 3*t^2 - 15/19
sage: f^2
4*t^14 + 12*t^9 - 60/19*t^7 + 9*t^4 - 90/19*t^2 + 225/361
sage: cyclo = R.cyclotomic_polynomial(7); cyclo
t^6 + t^5 + t^4 + t^3 + t^2 + t + 1
sage: g = 7 * cyclo * t^5 * (t^5 + 10*t + 2)
sage: g
7*t^16 + 7*t^15 + 7*t^14 + 7*t^13 + 77*t^12 + 91*t^11 + 91*t^10 + 84*t^9
       + 84*t^8 + 84*t^7 + 84*t^6 + 14*t^5
sage: F = factor(g); F
(7) * t^5 * (t^5 + 10*t + 2) * (t^6 + t^5 + t^4 + t^3 + t^2 + t + 1)
sage: F.unit()
7
sage: list(F)
[(t, 5), (t^5 + 10*t + 2, 1), (t^6 + t^5 + t^4 + t^3 + t^2 + t + 1, 1)]
```

On remarquera que la factorisation prend correctement en compte le coefficient dominant, et ne l'oublie pas dans le résultat.

S'il arrive que vous utilisiez intensivement, par exemple, la fonction `R.cyclotomic_polynomial` dans un projet de recherche quelconque, en plus de citer Sage, vous devriez chercher à quel composant Sage fait appel pour calculer en réalité ce polynôme cyclotomique et citer ce composant. Dans ce cas particulier, en tapant `R.cyclotomic_polynomial??` pour voir le code source, vous verriez rapidement une ligne telle que `f = pari.polcyclo(n)` ce qui signifie que PARI est utilisé pour le calcul du polynôme cyclotomique. Pensez à citer PARI dans votre travail.

La division d'un polynôme par un autre produit un élément du corps des fractions, que Sage crée automatiquement.

```
sage: x = QQ['x'].0
sage: f = x^3 + 1; g = x^2 - 17
sage: h = f/g;  h
(x^3 + 1)/(x^2 - 17)
sage: h.parent()
Fraction Field of Univariate Polynomial Ring in x over Rational Field
```

En utilisant des séries de Laurent, on peut calculer des développements en série dans le corps des fractions de `QQ[x]` :

```
sage: R.<x> = LaurentSeriesRing(QQ); R
Laurent Series Ring in x over Rational Field
sage: 1/(1-x) + O(x^10)
1 + x + x^2 + x^3 + x^4 + x^5 + x^6 + x^7 + x^8 + x^9 + O(x^10)
```

Si l'on nomme les variables différemment, on obtient un anneau de polynômes univariés différent.

```
sage: R.<x> = PolynomialRing(QQ)
sage: S.<y> = PolynomialRing(QQ)
sage: x == y
False
sage: R == S
False
sage: R(y)
x
sage: R(y^2 - 17)
x^2 - 17
```

L'anneau est déterminé par sa variable. Notez que créer un autre anneau avec la même variable `x` ne renvoie pas de nouvel anneau.

```
sage: R = PolynomialRing(QQ, "x")
sage: T = PolynomialRing(QQ, "x")
sage: R == T
True
sage: R is T
True
sage: R.0 == T.0
True
```

Sage permet aussi de travailler dans des anneaux de séries formelles et de séries de Laurent sur un anneau de base quelconque. Dans l'exemple suivant, nous créons un élément de $\mathbf{F}_7[[T]]$ et effectuons une division pour obtenir un élément de $\mathbf{F}_7((T))$.

```
sage: R.<T> = PowerSeriesRing(GF(7)); R
Power Series Ring in T over Finite Field of size 7
sage: f = T  + 3*T^2 + T^3 + O(T^4)
sage: f^3
T^3 + 2*T^4 + 2*T^5 + O(T^6)
sage: 1/f
T^-1 + 4 + T + O(T^2)
sage: parent(1/f)
Laurent Series Ring in T over Finite Field of size 7
```

On peut aussi créer des anneaux de séries formelles en utilisant des doubles crochets :

```
sage: GF(7)[['T']]
Power Series Ring in T over Finite Field of size 7
```

2.8.2 Polynômes multivariés

Pour travailler avec des polynômes à plusieurs variables, on commence par déclarer l'anneau des polynômes et les variables, de l'une des deux manières suivantes.

```
sage: R = PolynomialRing(GF(5),3,"z") # here, 3 = number of variables
sage: R
Multivariate Polynomial Ring in z0, z1, z2 over Finite Field of size 5
```

De même que pour les polynômes à une seule variable, les variantes suivantes sont autorisées :

```
sage: GF(5)['z0, z1, z2']
Multivariate Polynomial Ring in z0, z1, z2 over Finite Field of size 5
sage: R.<z0,z1,z2> = GF(5)[]; R
Multivariate Polynomial Ring in z0, z1, z2 over Finite Field of size 5
```

Si l'on désire de simples lettres comme noms de variables, on peut utiliser les raccourcis suivants :

```
sage: PolynomialRing(GF(5), 3, 'xyz')
Multivariate Polynomial Ring in x, y, z over Finite Field of size 5
```

A présent, passons aux questions arithmétiques.

```
sage: z = GF(5)['z0, z1, z2'].gens()
sage: z
(z0, z1, z2)
sage: (z[0]+z[1]+z[2])^2
z0^2 + 2*z0*z1 + z1^2 + 2*z0*z2 + 2*z1*z2 + z2^2
```

On peut aussi utiliser des notations plus mathématiques pour construire un anneau de polynômes.

```
sage: R = GF(5)['x,y,z']
sage: x,y,z = R.gens()
sage: QQ['x']
Univariate Polynomial Ring in x over Rational Field
sage: QQ['x,y'].gens()
(x, y)
sage: QQ['x'].objgens()
(Univariate Polynomial Ring in x over Rational Field, (x,))
```

Sous Sage, les polynômes multivariés sont implémentés en représentation « distributive » (par opposition à récursive), à l'aide de dictionnaires Python. Sage a souvent recours à Singular *[Si]*, par exemple, pour le calcul de pgcd ou de bases de Gröbner d'idéaux.

```
sage: R, (x, y) = PolynomialRing(RationalField(), 2, 'xy').objgens()
sage: f = (x^3 + 2*y^2*x)^2
sage: g = x^2*y^2
sage: f.gcd(g)
x^2
```

Créons ensuite l'idéal (f, g) engendré par f et g, en multipliant simplement `(f,g)` par `R` (nous pourrions aussi bien écrire `ideal([f,g])` ou `ideal(f,g)`).

```
sage: I = (f, g)*R; I
Ideal (x^6 + 4*x^4*y^2 + 4*x^2*y^4, x^2*y^2) of Multivariate Polynomial
Ring in x, y over Rational Field
sage: B = I.groebner_basis(); B
[x^6, x^2*y^2]
sage: x^2 in I
False
```

En passant, la base de Gröbner ci-dessus n'est pas une liste mais une suite non mutable. Ceci signifie qu'elle possède un univers, un parent, et qu'elle ne peut pas être modifiée (ce qui est une bonne chose puisque changer la base perturberait d'autres routines qui utilisent la base de Gröbner).

```
sage: B.universe()
Multivariate Polynomial Ring in x, y over Rational Field
sage: B[1] = x
```

```
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
```

Un peu (comprenez : pas assez à notre goût) d'algèbre commutative est disponible en Sage. Ces routines font appel à Singular. Par exemple, il est possible de calculer la décomposition en facteurs premiers et les idéaux premiers associés de I :

```
sage: I.primary_decomposition()
[Ideal (x^2) of Multivariate Polynomial Ring in x, y over Rational Field,
 Ideal (y^2, x^6) of Multivariate Polynomial Ring in x, y over Rational Field]
sage: I.associated_primes()
[Ideal (x) of Multivariate Polynomial Ring in x, y over Rational Field,
 Ideal (y, x) of Multivariate Polynomial Ring in x, y over Rational Field]
```

2.9 Parents, conversions, coercitions

Cette section peut paraître plus technique que celles qui précèdent, mais nous pensons qu'il est important de comprendre ce que sont les parents et les coercitions pour utiliser comme il faut les structures algébriques fournies par Sage.

Nous allons voir ici ce que ces notions signifient, mais pas comment les mettre en œuvre pour implémenter une nouvelle structure algébrique. Un tutoriel thématique couvrant ce point est disponible ici.

2.9.1 Éléments

Une première approximation en Python de la notion mathématique d'anneau pourrait consister à définir une classe pour les éléments `X` de l'anneau concerné, de fournir les méthodes « double-underscore » nécessaires pour donner un sens aux opérations de l'anneau, par exemple `__add__`, `__sub__` et `__mul__`, et naturellement de s'assurer qu'elles respectent les axiomes de la structure d'anneau.

Python étant un language (dynamiquement) fortement typé, on pourrait s'attendre à devoir implémenter une classe pour chaque anneau. Après tout, Python définit bien un type `<int>` pour les entiers, un type `<float>` pour les réels, et ainsi de suite. Mais cette approche ne peut pas fonctionner : il y a une infinité d'anneaux différents, et l'on ne peut pas implémenter une infinité de classes !

Une autre idée est de créer une hiérarchie de classes destinées à implémenter les éléments des structures algébriques usuelles : éléments de groupes, d'anneaux, d'algèbres à division, d'anneaux commutatifs, de corps, d'algèbres, etc.

Mais cela signifie que des éléments d'anneaux franchement différents peuvent avoir le même type.

```
sage: P.<x,y> = GF(3)[]
sage: Q.<a,b> = GF(4,'z')[]
sage: type(x)==type(a)
True
```

On pourrait aussi vouloir avoir des classes Python différentes pour fournir plusieurs implémentations d'une même structure mathématique (matrices denses contre matrices creuses par exemple).

```
sage: P.<a> = PolynomialRing(ZZ)
sage: Q.<b> = PolynomialRing(ZZ, sparse=True)
sage: R.<c> = PolynomialRing(ZZ, implementation='NTL')
sage: type(a); type(b); type(c)
<type 'sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_
↪flint'>
```

```
<class 'sage.rings.polynomial.polynomial_ring.PolynomialRing_integral_domain_with_
↪category.element_class'>
<type 'sage.rings.polynomial.polynomial_integer_dense_ntl.Polynomial_integer_dense_ntl
↪'>
```

Deux problèmes se posent alors. D'une part, si deux éléments sont instances de la même classe, on s'attend à ce que leur méthode `__add__` soit capable de les additionner, alors que ce n'est pas ce que l'on souhaite si les éléments appartiennent en fait à des anneaux différents. D'autre part, si l'on a deux éléments qui appartiennent à des implémentations différentes d'un même anneau, on veut pouvoir les ajouter, et ce n'est pas immédiats s'ils ne sont pas instances de la même classe.

La solution à ces difficultés est fournie par le mécanisme de *coercition* décrit ci-dessous.

Mais avant tout, il est essentiel que chaque élément « sache » de quoi il est élément. Cette information est donnée par la méthode `parent()`.

```
sage: a.parent(); b.parent(); c.parent()
Univariate Polynomial Ring in a over Integer Ring
Sparse Univariate Polynomial Ring in b over Integer Ring
Univariate Polynomial Ring in c over Integer Ring (using NTL)
```

2.9.2 Parents et catégories

En plus d'une hiérarchie de classes destinée à implémenter les éléments de structures algébriques, Sage fournit une hiérarchie similaire pour les structures elles-mêmes. Ces structures s'appellent en Sage des *parents*, et leurs classes dérivent d'une même classe de base. Celle-ci a des sous-classes « ensemble », « anneau », « corps », et ainsi de suite, dont la hiérarchie correspond à peu près à celle des concepts mathématiques qu'elles décrivent :

```
sage: isinstance(QQ,Field)
True
sage: isinstance(QQ, Ring)
True
sage: isinstance(ZZ,Field)
False
sage: isinstance(ZZ, Ring)
True
```

Or en algèbre, on regroupe les objets qui partagent le même genre de structure algébrique en ce que l'on appelle des *catégories*. Il y a donc un parallèle approximatif entre la hiérarchie des classes de Sage et la hiérarchie des catégories. Mais cette correspondance n'est pas parfaite, et Sage implémente par ailleurs les catégories en tant que telles :

```
sage: Rings()
Category of rings
sage: ZZ.category()
Join of Category of euclidean domains
    and Category of infinite enumerated sets
    and Category of metric spaces
sage: ZZ.category().is_subcategory(Rings())
True
sage: ZZ in Rings()
True
sage: ZZ in Fields()
False
sage: QQ in Fields()
True
```

Tandis que la hiérarchie des classes est déterminée avant tout par des considérations de programmation, l'infrastructure des catégories cherche plutôt à respecter la structure mathématique. Elle permet de munir les objets d'une catégorie de méthodes et de tests génériques, qui ne dépendent pas de l'implémentation particulière d'un objet donné de la catégorie.

Les parents en tant qu'objets Python doivent être uniques. Ainsi, lorsqu'un anneau de polynômes sur un anneau donné et avec une liste donnée de générateurs est construit, il est conservé en cache et réutilisé par la suite :

```
sage: RR['x','y'] is RR['x','y']
True
```

2.9.3 Types et parents

Le type `RingElement` ne correspond pas parfaitement à la notion mathématique d'élément d'anneau. Par exemple, bien que les matrices carrées appartiennent à un anneau, elles ne sont pas de type `RingElement` :

```
sage: M = Matrix(ZZ,2,2); M
[0 0]
[0 0]
sage: isinstance(M, RingElement)
False
```

Si les *parents* sont censés être uniques, des *éléments* égaux d'un parent ne sont pas nécessairement identiques. Le comportement de Sage diffère ici de celui de Python pour certains entiers (pas tous) :

```
sage: int(1) is int(1) # Python int
True
sage: int(-15) is int(-15)
False
sage: 1 is 1           # Sage Integer
False
```

Il faut bien comprendre que les éléments d'anneaux différents ne se distinguent généralement pas par leur type, mais par leur parent :

```
sage: a = GF(2)(1)
sage: b = GF(5)(1)
sage: type(a) is type(b)
True
sage: parent(a)
Finite Field of size 2
sage: parent(b)
Finite Field of size 5
```

Ainsi, **le parent d'un élément est plus important que son type** du point de vue algébrique.

2.9.4 Conversion et coercition

Il est parfois possible de convertir un élément d'un certain parent en élément d'un autre parent. Une telle conversion peut être explicite ou implicite. Les conversions implicites sont appelées *coercitions*.

Le lecteur aura peut-être rencontré les notions de *conversion de type* et de *coercition de type* dans le contexte du langage C par exemple. En Sage, il existe aussi des notions de conversion et de coercition, mais elles s'appliquent aux *parents* et non aux types. Attention donc à ne pas confondre les conversions en Sage avec les conversions de type du C !

Nous nous limitons ici à une brève présentation, et renvoyons le lecteur à la section du manuel de référence consacrée aux coercitions ainsi qu'au tutoriel spécifique pour plus de détails.

On peut adopter deux positions extrêmes sur les opérations arithmétiques entre éléments d'anneaux *différents* :

— les anneaux différents sont des mondes indépendants, et l'addition ou la multiplication entre éléments d'anneaux différents n'ont aucun sens ; même `1 + 1/2` n'a pas de sens puisque le premier terme est un entier et le second un rationnel ;

ou

— si un élément `r1` d'un anneau `R1` peut, d'une manière ou d'une autre, s'interpréter comme élément d'un autre anneau `R2`, alors toutes les opérations arithmétiques entre `r1` et un élément quelconque de `R2` sont permises. En particulier, les éléments neutres de la multiplication dans les corps et anneaux doivent tous être égaux entre eux.

Sage adopte un compromis. Si `P1` et `P2` sont des parents et si `p1` est un élément de `P1`, l'utilisateur peut demander explicitement comment `P1` s'interprète dans `P2`. Cela n'a pas forcément de sens dans tous les cas, et l'interprétation peut n'être définie que pour certains éléments de `P1` ; c'est à l'utilisateur de s'assurer que la conversion a un sens. Cela s'appelle une **conversion** :

```
sage: a = GF(2)(1)
sage: b = GF(5)(1)
sage: GF(5)(a) == b
True
sage: GF(2)(b) == a
True
```

Cependant, une conversion *implicite* (c'est-à-dire automatique) n'est possible que si elle peut se faire *systématiquement* et de manière *cohérente*. Il faut ici absolument faire preuve de rigueur.

Une telle conversion implicite s'appelle une **coercition**. Si une coercition est définie entre deux parents, elle doit coïncider avec la conversion. De plus, les coercitions doivent obéir aux deux conditions suivantes :

1. Une coercition de `P1` dans `P2` doit être un morphisme (par exemple un morphisme d'anneaux). Elle doit être définie pour *tous* les éléments de `P1`, et préserver la structure algébrique de celui-ci.
2. Le choix des applications de coercition doit être fait de manière cohérente. Si `P3` est un troisième parent, la composée de la coercition choisie de `P1` dans `P2` et de celle de `P2` dans `P3` doit être la coercition de `P1` dans `P3`. En particulier, s'il existe des coercitions de `P1` dans `P2` et de `P2` dans `P1`, leur composée doit être l'identité sur `P1`.

Ainsi, bien qu'il soit possible de convertir tout élément de `GF(2)` en un élément de `GF(5)`, la conversion ne peut être une coercition, puisque il n'existe pas de morphisme d'anneaux de `GF(2)` dans `GF(5)`.

Le second point — la cohérence des choix — est un peu plus compliqué à expliquer. Illustrons-le sur l'exemple des anneaux de polynômes multivariés. Dans les applications, il s'avère utile que les coercitions respectent les noms des variables. Nous avons donc :

```
sage: R1.<x,y> = ZZ[]
sage: R2 = ZZ['y','x']
sage: R2.has_coerce_map_from(R1)
True
sage: R2(x)
x
sage: R2(y)
y
```

En l'absence d'un morphisme d'anneau qui préserve les noms de variable, la coercition entre anneaux de polynômes multivariés n'est pas définie. Il peut tout de même exister une conversion qui envoie les variables d'un anneau sur celle de l'autre en fonction de leur position dans la liste des générateurs :

```
sage: R3 = ZZ['z','x']
sage: R3.has_coerce_map_from(R1)
False
sage: R3(x)
z
sage: R3(y)
x
```

Mais une telle conversion ne répond pas aux critères pour être une coercition : en effet, en composant l'application de `ZZ['x','y']` dans `ZZ['y','x']` avec celle qui préserve les positions de `ZZ['y','x']` dans `ZZ['a','b']`, nous obtiendrions une application qui ne préserve ni les noms ni les positions, ce qui viole la règle de cohérence.

Lorsqu'une coercition est définie, elle est souvent utilisée pour comparer des éléments d'anneaux différents ou pour effectuer des opérations arithmétiques. Cela est commode, mais il faut être prudent en étendant la relation d'égalité == au-delà des frontières d'un parent donné. Par exemple, si == est bien censé être une relation d'équivalence entre éléments d'"*un* anneau, il n'en va pas forcément de même quand on compare des éléments d'anneaux différents. Ainsi, les éléments `1` de `ZZ` et d'un corps fini sont considérés comme égaux, puisqu'il existe une coercition canonique des entiers dans tout corps fini. En revanche, il n'y a en général pas de coercition entre deux corps finis quelconques. On a donc

```
sage: GF(5)(1) == 1
True
sage: 1 == GF(2)(1)
True
sage: GF(5)(1) == GF(2)(1)
False
sage: GF(5)(1) != GF(2)(1)
True
```

De même, on a

```
sage: R3(R1.1) == R3.1
True
sage: R1.1 == R3.1
False
sage: R1.1 != R3.1
True
```

Une autre conséquence de la condition de cohérence est que les coercitions ne sont possibles que des anneaux exacts (comme les rationnels `QQ`) vers les anneaux inexacts (comme les réels à précision donnée `RR`), jamais l'inverse. En effet, pour qu'une conversion de `RR` dans `QQ` puisse être une coercition, il faudrait que la composée de la coercition de `QQ` dans `RR` et de cette conversion soit l'identité sur `QQ`, ce qui n'est pas possible puisque des rationnels distincts peuvent très bien être envoyés sur le même élément de `RR` :

```
sage: RR(1/10^200+1/10^100) == RR(1/10^100)
True
sage: 1/10^200+1/10^100 == 1/10^100
False
```

Lorsque l'on compare des éléments de deux parents `P1` et `P2`, il peut arriver qu'il n'existe pas de coercition entre `P1` et `P2`, mais qu'il y ait un choix canonique de parent `P3` tel que `P1` et `P2` admettent tous deux des coercitions dans `P3`. Dans ce cas aussi, la coercition a lieu. Un exemple typique de ce mécanisme est l'addition d'un rationnel et d'un polynôme à coefficients entiers, qui produit un polynôme à coefficients rationnels :

```
sage: P1.<x> = ZZ[]
sage: p = 2*x+3
sage: q = 1/2
```

```
sage: parent(p)
Univariate Polynomial Ring in x over Integer Ring
sage: parent(p+q)
Univariate Polynomial Ring in x over Rational Field
```

Notons qu'en principe, on aurait très bien pu choisir pour `P3` le corps des fractions de `ZZ['x']`. Cependant, Sage tente de choisir un parent commun *canonique* aussi naturel que possible (ici `QQ['x']`). Afin que cela fonctionne de façon fiable, Sage ne se contente *pas* de prendre n'importe lequel lorsque plusieurs candidats semblent aussi naturels les uns que les autres. La manière dont le choix est fait est décrite dans le tutoriel spécifique déjà mentionné.

Dans l'exemple suivant, il n'y a pas de coercition vers un parent commun :

```
sage: R.<x> = QQ[]
sage: S.<y> = QQ[]
sage: x+y
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for +: 'Univariate Polynomial Ring in x over␣
↪Rational Field' and 'Univariate Polynomial Ring in y over Rational Field'
```

En effet, Sage refuse de choisir entre les candidats `QQ['x']['y']`, `QQ['y']['x']`, `QQ['x','y']` et `QQ['y','x']`, car ces quatre structures deux à deux distinctes semblent toutes des parents communs naturels, et aucun choix canonique ne s'impose.

2.10 Algèbre linéaire

Sage fournit les constructions standards d'algèbre linéaire, par exemple le polynôme caractéristique, la forme échelonnée, la trace, diverses décompositions, etc. d'une matrice.

La création de matrices et la multiplication matricielle sont très faciles et naturelles :

```
sage: A = Matrix([[1,2,3],[3,2,1],[1,1,1]])
sage: w = vector([1,1,-4])
sage: w*A
(0, 0, 0)
sage: A*w
(-9, 1, -2)
sage: kernel(A)
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[ 1  1 -4]
```

Notez bien qu'avec Sage, le noyau d'une matrice A est le « noyau à gauche », c'est-à-dire l'espace des vecteurs w tels que $wA = 0$.

La résolution d'équations matricielles est facile et se fait avec la méthode `solve_right`. L'évaluation de `A.solve_right(Y)` renvoie une matrice (ou un vecteur) X tel que $AX = Y$:

```
sage: Y = vector([0, -4, -1])
sage: X = A.solve_right(Y)
sage: X
(-2, 1, 0)
sage: A * X   # vérifions la réponse...
(0, -4, -1)
```

Un antislash (contre-oblique) \ peut être employé à la place de `solve_right` : il suffit d'écrire `A \ Y` au lieu de `A.solve_right(Y)`.

```
sage: A \ Y
(-2, 1, 0)
```

S'il n'y a aucune solution, Sage renvoie une erreur :

```
sage: A.solve_right(w)
Traceback (most recent call last):
...
ValueError: matrix equation has no solutions
```

De même, il faut utiliser `A.solve_left(Y)` pour résoudre en X l'équation $XA = Y$.

Sage sait aussi calculer les valeurs propres et vecteurs propres :

```
sage: A = matrix([[0, 4], [-1, 0]])
sage: A.eigenvalues ()
[-2*I, 2*I]
sage: B = matrix([[1, 3], [3, 1]])
sage: B.eigenvectors_left()
[(4, [
(1, 1)
], 1), (-2, [
(1, -1)
], 1)]
```

(La sortie de `eigenvectors_left` est une liste de triplets (valeur propre, vecteur propre, multiplicité).) Sur `QQ` et `RR`, on peut aussi utiliser Maxima (voir la section *Maxima* ci-dessous).

Comme signalé en *Anneaux de base*, l'anneau sur lequel une matrice est définie a une influence sur les propriétés de la matrice. Dans l'exemple suivant, le premier argument de la commande `matrix` indique à Sage s'il faut traiter la matrice comme une matrice d'entier (`ZZ`), de rationnels (`QQ`) ou de réels (`RR`) :

```
sage: AZ = matrix(ZZ, [[2,0], [0,1]])
sage: AQ = matrix(QQ, [[2,0], [0,1]])
sage: AR = matrix(RR, [[2,0], [0,1]])
sage: AZ.echelon_form()
[2 0]
[0 1]
sage: AQ.echelon_form()
[1 0]
[0 1]
sage: AR.echelon_form()
[ 1.00000000000000 0.000000000000000]
[0.000000000000000  1.00000000000000]
```

Pour le calcul de valeurs propres et vecteurs propres sur les nombres à virgule flottante réels ou complexes, la matrice doit être respectivement à coefficients dans `RDF` (*Real Double Field*, nombres réels à précision machine) ou `CDF` (*Complex Double Field*). Lorsque l'on définit une matrice avec des coefficients flottants sans spécifier explicitement l'anneau de base, ce ne sont pas `RDF` ou `CDF` qui sont utilisés par défaut, mais `RR` et `CC`, sur lesquels ces calculs ne sont pas implémentés dans tous les cas :

```
sage: ARDF = matrix(RDF, [[1.2, 2], [2, 3]])
sage: ARDF.eigenvalues() # rel tol 8e-16
[-0.09317121994613098, 4.293171219946131]
sage: ACDF = matrix(CDF, [[1.2, I], [2, 3]])
```

```
sage: ACDF.eigenvectors_right()  # rel tol 3e-15
[(0.8818456983293743 - 0.8209140653434135*I, [(0.7505608183809549, -0.616145932704589␣
↪+ 0.2387941530333261*I)], 1),
(3.3181543016706256 + 0.8209140653434133*I, [(0.14559469829270957 + 0.
↪3756690858502104*I, 0.9152458258662108)], 1)]
```

2.10.1 Espaces de matrices

Créons l'espace $\mathrm{Mat}_{3\times 3}(\mathbf{Q})$:

```
sage: M = MatrixSpace(QQ,3)
sage: M
Full MatrixSpace of 3 by 3 dense matrices over Rational Field
```

(Pour indiquer l'espace des matrices 3 par 4, il faudrait utiliser `MatrixSpace(QQ,3,4)`. Si le nombre de colonnes est omis, il est égal par défaut au nombre de lignes. Ainsi `MatrixSpace(QQ,3)` est un synonyme de `MatrixSpace(QQ,3,3)`). L'espace des matrices est muni de sa base canonique :

```
sage: B = M.basis()
sage: len(B)
9
sage: B[0,1]
[0 1 0]
[0 0 0]
[0 0 0]
```

Nous créons une matrice comme un élément de `M`.

```
sage: A = M(range(9)); A
[0 1 2]
[3 4 5]
[6 7 8]
```

Puis, nous calculons sa forme échelonnée en ligne et son noyau.

```
sage: A.echelon_form()
[ 1  0 -1]
[ 0  1  2]
[ 0  0  0]
sage: A.kernel()
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[ 1 -2  1]
```

Puis nous illustrons les possibilités de calcul de matrices définies sur des corps finis :

```
sage: M = MatrixSpace(GF(2),4,8)
sage: A = M([1,1,0,0, 1,1,1,1, 0,1,0,0, 1,0,1,1,
....:        0,0,1,0, 1,1,0,1, 0,0,1,1, 1,1,1,0])
sage: A
[1 1 0 0 1 1 1 1]
[0 1 0 0 1 0 1 1]
[0 0 1 0 1 1 0 1]
[0 0 1 1 1 1 1 0]
sage: rows = A.rows()
sage: A.columns()
```

```
[(1, 0, 0, 0), (1, 1, 0, 0), (0, 0, 1, 1), (0, 0, 0, 1),
 (1, 1, 1, 1), (1, 0, 1, 1), (1, 1, 0, 1), (1, 1, 1, 0)]
sage: rows
[(1, 1, 0, 0, 1, 1, 1, 1), (0, 1, 0, 0, 1, 0, 1, 1),
 (0, 0, 1, 0, 1, 1, 0, 1), (0, 0, 1, 1, 1, 1, 1, 0)]
```

Nous créons le sous-espace engendré sur $\mathbf{F}_2$ par les vecteurs lignes ci-dessus.

```
sage: V = VectorSpace(GF(2),8)
sage: S = V.subspace(rows)
sage: S
Vector space of degree 8 and dimension 4 over Finite Field of size 2
Basis matrix:
[1 0 0 0 0 1 0 0]
[0 1 0 0 1 0 1 1]
[0 0 1 0 1 1 0 1]
[0 0 0 1 0 0 1 1]
sage: A.echelon_form()
[1 0 0 0 0 1 0 0]
[0 1 0 0 1 0 1 1]
[0 0 1 0 1 1 0 1]
[0 0 0 1 0 0 1 1]
```

La base de S utilisée par Sage est obtenue à partir des lignes non nulles de la matrice des générateurs de S réduite sous forme échelonnée en lignes.

2.10.2 Algèbre linéaire creuse

Sage permet de travailler avec des matrices creuses sur des anneaux principaux.

```
sage: M = MatrixSpace(QQ, 100, sparse=True)
sage: A = M.random_element(density = 0.05)
sage: E = A.echelon_form()
```

L'algorithme multi-modulaire présent dans Sage fonctionne bien pour les matrices carrées (mais moins pour les autres) :

```
sage: M = MatrixSpace(QQ, 50, 100, sparse=True)
sage: A = M.random_element(density = 0.05)
sage: E = A.echelon_form()
sage: M = MatrixSpace(GF(2), 20, 40, sparse=True)
sage: A = M.random_element()
sage: E = A.echelon_form()
```

Notez que Python distingue les majuscules des minuscules :

```
sage: M = MatrixSpace(QQ, 10,10, Sparse=True)
Traceback (most recent call last):
...
TypeError: __classcall__() got an unexpected keyword argument 'Sparse'
```

2.11 Groupes finis, groupes abéliens

Sage permet de faire des calculs avec des groupes de permutation, des groupes classiques finis (tels que $SU(n, q)$), des groupes finis de matrices (avec vos propres générateurs) et des groupes abéliens (même infinis). La plupart de ces fonctionnalités est implémentée par une interface vers GAP.

Par exemple, pour créer un groupe de permutation, il suffit de donner une liste de générateurs, comme dans l'exemple suivant.

```
sage: G = PermutationGroup(['(1,2,3)(4,5)', '(3,4)'])
sage: G
Permutation Group with generators [(3,4), (1,2,3)(4,5)]
sage: G.order()
120
sage: G.is_abelian()
False
sage: G.derived_series()           # sortie plus ou moins aléatoire (random)
[Permutation Group with generators [(1,2,3)(4,5), (3,4)],
 Permutation Group with generators [(1,5)(3,4), (1,5)(2,4), (1,3,5)]]
sage: G.center()
Subgroup of (Permutation Group with generators [(3,4), (1,2,3)(4,5)]) generated by␣
↪[()]
sage: G.random_element()           # sortie aléatoire (random)
(1,5,3)(2,4)
sage: print(latex(G))
\langle (3,4), (1,2,3)(4,5) \rangle
```

On peut obtenir la table des caractères (au format LaTeX) à partir de Sage :

```
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3)]])
sage: latex(G.character_table())
\left(\begin{array}{rrrr}
1 & 1 & 1 & 1 \\
1 & -\zeta_{3} - 1 & \zeta_{3} & 1 \\
1 & \zeta_{3} & -\zeta_{3} - 1 & 1 \\
3 & 0 & 0 & -1
\end{array}\right)
```

Sage inclut aussi les groupes classiques ou matriciels définis sur des corps finis :

```
sage: MS = MatrixSpace(GF(7), 2)
sage: gens = [MS([[1,0],[-1,1]]),MS([[1,1],[0,1]])]
sage: G = MatrixGroup(gens)
sage: G.conjugacy_classes_representatives()
(
[1 0]  [0 6]  [0 4]  [6 0]  [0 6]  [0 4]  [0 6]  [0 6]  [0 6]  [4 0]
[0 1], [1 5], [5 5], [0 6], [1 2], [5 2], [1 0], [1 4], [1 3], [0 2],

[5 0]
[0 3]
)
sage: G = Sp(4,GF(7))
sage: G
Symplectic Group of degree 4 over Finite Field of size 7
sage: G.random_element()  # élément du groupe tiré au hasard (random)
[5 5 5 1]
[0 2 6 3]
[5 0 1 0]
```

```
[4 6 3 4]
sage: G.order()
276595200
```

On peut aussi effectuer des calculs dans des groupes abéliens (infinis ou finis) :

```
sage: F = AbelianGroup(5, [5,5,7,8,9], names='abcde')
sage: (a, b, c, d, e) = F.gens()
sage: d * b**2 * c**3
b^2*c^3*d
sage: F = AbelianGroup(3,[2]*3); F
Multiplicative Abelian group isomorphic to C2 x C2 x C2
sage: H = AbelianGroup([2,3], names="xy"); H
Multiplicative Abelian group isomorphic to C2 x C3
sage: AbelianGroup(5)
Multiplicative Abelian group isomorphic to Z x Z x Z x Z x Z
sage: AbelianGroup(5).order()
+Infinity
```

2.12 Théorie des nombres

Sage possède des fonctionnalités étendues de théorie des nombres. Par exemple, on peut faire de l'arithmétique dans $\mathbf{Z}/N\mathbf{Z}$ comme suit :

```
sage: R = IntegerModRing(97)
sage: a = R(2) / R(3)
sage: a
33
sage: a.rational_reconstruction()
2/3
sage: b = R(47)
sage: b^20052005
50
sage: b.modulus()
97
sage: b.is_square()
True
```

Sage contient les fonctions standards de théorie des nombres. Par exemple,

```
sage: gcd(515,2005)
5
sage: factor(2005)
5 * 401
sage: c = factorial(25); c
15511210043330985984000000
sage: [valuation(c,p) for p in prime_range(2,23)]
[22, 10, 6, 3, 2, 1, 1, 1]
sage: next_prime(2005)
2011
sage: previous_prime(2005)
2003
sage: divisors(28); sum(divisors(28)); 2*28
[1, 2, 4, 7, 14, 28]
```

```
56
56
```

Voilà qui est parfait !

La fonction `sigma(n,k)` de Sage additionne les k-ièmes puissances des diviseurs de `n` :

```
sage: sigma(28,0); sigma(28,1); sigma(28,2)
6
56
1050
```

Nous illustrons à présent l'algorithme d'Euclide de recherche d'une relation de Bézout, l'indicatrice d'Euler ϕ et le théorème des restes chinois :

```
sage: d,u,v = xgcd(12,15)
sage: d == u*12 + v*15
True
sage: n = 2005
sage: inverse_mod(3,n)
1337
sage: 3 * 1337
4011
sage: prime_divisors(n)
[5, 401]
sage: phi = n*prod([1 - 1/p for p in prime_divisors(n)]); phi
1600
sage: euler_phi(n)
1600
sage: prime_to_m_part(n, 5)
401
```

Voici une petite expérience concernant la conjecture de Syracuse :

```
sage: n = 2005
sage: for i in range(1000):
....:     n = 3*odd_part(n) + 1
....:     if odd_part(n)==1:
....:         print(i)
....:         break
38
```

Et finalement un exemple d'utilisation du théorème chinois :

```
sage: x = crt(2, 1, 3, 5); x
11
sage: x % 3  # x mod 3 = 2
2
sage: x % 5  # x mod 5 = 1
1
sage: [binomial(13,m) for m in range(14)]
[1, 13, 78, 286, 715, 1287, 1716, 1716, 1287, 715, 286, 78, 13, 1]
sage: [binomial(13,m)%2 for m in range(14)]
[1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1]
sage: [kronecker(m,13) for m in range(1,13)]
[1, -1, 1, 1, -1, -1, -1, -1, 1, 1, -1, 1]
sage: n = 10000; sum([moebius(m) for m in range(1,n)])
-23
```

```
sage: Partitions(4).list()
[[4], [3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
```

2.12.1 Nombres p-adiques

Le corps des nombres p-adiques est implémenté en Sage. Notez qu'une fois qu'un corps p-adique est créé, il n'est plus possible d'en changer la précision.

```
sage: K = Qp(11); K
11-adic Field with capped relative precision 20
sage: a = K(211/17); a
4 + 4*11 + 11^2 + 7*11^3 + 9*11^5 + 5*11^6 + 4*11^7 + 8*11^8 + 7*11^9
  + 9*11^10 + 3*11^11 + 10*11^12 + 11^13 + 5*11^14 + 6*11^15 + 2*11^16
  + 3*11^17 + 11^18 + 7*11^19 + O(11^20)
sage: b = K(3211/11^2); b
10*11^-2 + 5*11^-1 + 4 + 2*11 + O(11^18)
```

Beaucoup de travail a été accompli afin d'implémenter l'anneau des entiers dans des corps p-adiques ou des corps de nombres distincts de $\mathbf{Q}$. Le lecteur intéressé est invité à poser ses questions aux experts sur le groupe Google `sage-support` pour plus de détails.

Un certain nombre de méthodes associées sont d'ores et déjà implémentées dans la classe NumberField.

```
sage: R.<x> = PolynomialRing(QQ)
sage: K = NumberField(x^3 + x^2 - 2*x + 8, 'a')
sage: K.integral_basis()
[1, 1/2*a^2 + 1/2*a, a^2]
```

```
sage: K.galois_group(type="pari")
Galois group PARI group [6, -1, 2, "S3"] of degree 3 of the Number Field
in a with defining polynomial x^3 + x^2 - 2*x + 8
```

```
sage: K.polynomial_quotient_ring()
Univariate Quotient Polynomial Ring in a over Rational Field with modulus
x^3 + x^2 - 2*x + 8
sage: K.units()
(3*a^2 + 13*a + 13,)
sage: K.discriminant()
-503
sage: K.class_group()
Class group of order 1 of Number Field in a with
defining polynomial x^3 + x^2 - 2*x + 8
sage: K.class_number()
1
```

2.13 Quelques mathématiques plus avancées

2.13.1 Géométrie algébrique

Il est possible de définir des variétés algébriques arbitraires avec Sage, mais les fonctionnalités non triviales sont parfois limitées aux anneaux sur $\mathbf{Q}$ ou sur les corps finis. Calculons par exemple la réunion de deux courbes planes affines, puis récupérons chaque courbe en tant que composante irréductible de la réunion.

```
sage: x, y = AffineSpace(2, QQ, 'xy').gens()
sage: C2 = Curve(x^2 + y^2 - 1)
sage: C3 = Curve(x^3 + y^3 - 1)
sage: D = C2 + C3
sage: D
Affine Plane Curve over Rational Field defined by
   x^5 + x^3*y^2 + x^2*y^3 + y^5 - x^3 - y^3 - x^2 - y^2 + 1
sage: D.irreducible_components()
[
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  x^2 + y^2 - 1,
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  x^3 + y^3 - 1
]
```

Nous pouvons également trouver tous les points d'intersection des deux courbes en les intersectant et en calculant les composantes irréductibles.

```
sage: V = C2.intersection(C3)
sage: V.irreducible_components()
[
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  y,
  x - 1,
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  y - 1,
  x,
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  x + y + 2,
  2*y^2 + 4*y + 3
]
```

Ainsi, par exemple, $(1, 0)$ et $(0, 1)$ appartiennent aux deux courbes (ce dont on pouvait directement s'apercevoir) ; il en va de même des points (quadratiques), dont la coordonnée en y satisfait à l'équation $2y^2 + 4y + 3 = 0$.

Sage peut calculer l'idéal torique de la cubique gauche dans l'espace projectif de dimension 3.

```
sage: R.<a,b,c,d> = PolynomialRing(QQ, 4)
sage: I = ideal(b^2-a*c, c^2-b*d, a*d-b*c)
sage: F = I.groebner_fan(); F
Groebner fan of the ideal:
Ideal (b^2 - a*c, c^2 - b*d, -b*c + a*d) of Multivariate Polynomial Ring
in a, b, c, d over Rational Field
sage: F.reduced_groebner_bases ()
[[-c^2 + b*d, -b*c + a*d, -b^2 + a*c],
 [-c^2 + b*d, b^2 - a*c, -b*c + a*d],
 [-c^2 + b*d, b*c - a*d, b^2 - a*c, -c^3 + a*d^2],
 [c^3 - a*d^2, -c^2 + b*d, b*c - a*d, b^2 - a*c],
 [c^2 - b*d, -b*c + a*d, -b^2 + a*c],
 [c^2 - b*d, b*c - a*d, -b^2 + a*c, -b^3 + a^2*d],
 [c^2 - b*d, b*c - a*d, b^3 - a^2*d, -b^2 + a*c],
 [c^2 - b*d, b*c - a*d, b^2 - a*c]]
sage: F.polyhedralfan()
Polyhedral fan in 4 dimensions of dimension 4
```

2.13.2 Courbes elliptiques

Les fonctionnalités relatives aux courbes elliptiques comprennent la plupart des fonctionnalités de PARI, l'accès aux données des tables en ligne de Cremona (ceci requiert le chargement d'une base de donnée optionnelle), les fonctionnalités de mwrank, c'est-à-dire la 2-descente avec calcul du groupe de Mordell-Weil complet, l'algorithme SEA, le calcul de toutes les isogénies, beaucoup de nouveau code pour les courbes sur $\mathbf{Q}$ et une partie du code de descente algébrique de Denis Simon.

La commande `EllipticCurve` permet de créer une courbe elliptique avec beaucoup de souplesse :

— EllipticCurve([a_1, a_2, a_3, a_4, a_6]) : renvoie la courbe elliptique

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6,$$

où les a_i"s sont convertis par coercition dans le parent de a_1. Si tous les a_i ont pour parent $\mathbf{Z}$, ils sont convertis par coercition dans $\mathbf{Q}$.

— EllipticCurve([a_4, a_6]) : idem avec $a_1 = a_2 = a_3 = 0$.

— EllipticCurve(label) : Renvoie la courbe elliptique sur $\mathbf{Q}$ de la base de données de Cremona selon son nom dans la (nouvelle !) nomenclature de Cremona. Les courbes sont étiquetées par une chaîne de caractère telle que `"11a"` ou `"37b2"`. La lettre doit être en minuscule (pour faire la différence avec l'ancienne nomenclature).

— EllipticCurve(j) : renvoie une courbe elliptique de j-invariant j.

— EllipticCurve(R, [a_1, a_2, a_3, a_4, a_6]) : Crée la courbe elliptique sur l'anneau R donnée par les coefficients a_i comme ci-dessus.

Illustrons chacune de ces constructions :

```
sage: EllipticCurve([0,0,1,-1,0])
Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field

sage: EllipticCurve([GF(5)(0),0,1,-1,0])
Elliptic Curve defined by y^2 + y = x^3 + 4*x over Finite Field of size 5

sage: EllipticCurve([1,2])
Elliptic Curve defined by y^2  = x^3 + x + 2 over Rational Field

sage: EllipticCurve('37a')
Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field

sage: EllipticCurve_from_j(1)
Elliptic Curve defined by y^2 + x*y = x^3 + 36*x + 3455 over Rational Field

sage: EllipticCurve(GF(5), [0,0,1,-1,0])
Elliptic Curve defined by y^2 + y = x^3 + 4*x over Finite Field of size 5
```

Le couple $(0, 0)$ est un point de la courbe elliptique E définie par $y^2 + y = x^3 - x$. Pour créer ce point avec Sage, il convient de taper `E([0,0])`. Sage peut additionner des points sur une telle courbe elliptique (rappelons qu'une courbe elliptique possède une structure de groupe additif où le point à l'infini représente l'élément neutre et où trois points alignés de la courbe sont de somme nulle) :

```
sage: E = EllipticCurve([0,0,1,-1,0])
sage: E
Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field
sage: P = E([0,0])
sage: P + P
(1 : 0 : 1)
sage: 10*P
(161/16 : -2065/64 : 1)
sage: 20*P
(683916417/264517696 : -18784454671297/4302115807744 : 1)
```

```
sage: E.conductor()
37
```

Les courbes elliptiques sur les nombres complexes sont paramétrées par leur j-invariant. Sage calcule le j-invariant comme suit :

```
sage: E = EllipticCurve([0,0,0,-4,2]); E
Elliptic Curve defined by y^2 = x^3 - 4*x + 2 over Rational Field
sage: E.conductor()
2368
sage: E.j_invariant()
110592/37
```

Si l'on fabrique une courbe avec le même j-invariant que celui de E, elle n'est pas nécessairement isomorphe à E. Dans l'exemple suivant, les courbes ne sont pas isomorphes parce que leur conducteur est différent.

```
sage: F = EllipticCurve_from_j(110592/37)
sage: F.conductor()
37
```

Toutefois, le twist de F par 2 donne une courbe isomorphe.

```
sage: G = F.quadratic_twist(2); G
Elliptic Curve defined by y^2 = x^3 - 4*x + 2 over Rational Field
sage: G.conductor()
2368
sage: G.j_invariant()
110592/37
```

On peut calculer les coefficients a_n de la série-L ou forme modulaire $\sum_{n=0}^{\infty} a_n q^n$ attachée à une courbe elliptique. Le calcul s'effectue en utilisant la bibliothèque PARI écrite en C :

```
sage: E = EllipticCurve([0,0,1,-1,0])
sage: E.anlist(30)
[0, 1, -2, -3, 2, -2, 6, -1, 0, 6, 4, -5, -6, -2, 2, 6, -4, 0, -12, 0, -4,
 3, 10, 2, 0, -1, 4, -9, -2, 6, -12]
sage: v = E.anlist(10000)
```

Il faut à peine quelques secondes pour calculer tous les coefficients a_n pour $n \leq 10^5$:

```
sage: %time v = E.anlist(100000)
CPU times: user 0.98 s, sys: 0.06 s, total: 1.04 s
Wall time: 1.06
```

Les courbes elliptiques peuvent être construites en utilisant leur nom dans la nomenclature de Cremona. Ceci charge par avance la courbe elliptique avec les informations la concernant, telles que son rang, son nombre de Tamagawa, son régulateur, etc.

```
sage: E = EllipticCurve("37b2")
sage: E
Elliptic Curve defined by y^2 + y = x^3 + x^2 - 1873*x - 31833 over Rational
Field
sage: E = EllipticCurve("389a")
sage: E
Elliptic Curve defined by y^2 + y = x^3 + x^2 - 2*x  over Rational Field
sage: E.rank()
2
```

```
sage: E = EllipticCurve("5077a")
sage: E.rank()
3
```

On peut aussi accéder à la base de données de Cremona directement.

```
sage: db = sage.databases.cremona.CremonaDatabase()
sage: db.curves(37)
{'a1': [[0, 0, 1, -1, 0], 1, 1], 'b1': [[0, 1, 1, -23, -50], 0, 3]}
sage: db.allcurves(37)
{'a1': [[0, 0, 1, -1, 0], 1, 1],
 'b1': [[0, 1, 1, -23, -50], 0, 3],
 'b2': [[0, 1, 1, -1873, -31833], 0, 1],
 'b3': [[0, 1, 1, -3, 1], 0, 3]}
```

Les objets extraits de la base de données ne sont pas de type `EllipticCurve`, mais de simples entrées de base de données formées de quelques champs. Par défaut, Sage est distribué avec une version réduite de la base de données de Cremona qui ne contient que des informations limitées sur les courbes elliptiques de conducteur ≤ 10000. Il existe également en option une version plus complète qui contient des données étendues portant sur toute les courbes de conducteur jusqu'à 120000 (à la date d'octobre 2005). Une autre - énorme (2GB) - base de données optionnelle, fournie dans un package séparé, contient des centaines de millions de courbes elliptiques de la bases de donnée de Stein-Watkins.

2.13.3 Caractères de Dirichlet

Un *caractère de Dirichlet* est une extension d'un homomorphisme $(\mathbf{Z}/N\mathbf{Z})^* \to R^*$, pour un certain anneau R, à l'application $\mathbf{Z} \to R$ obtenue en envoyant les entiers x tels que $\gcd(N, x) > 1$ vers 0.

```
sage: G = DirichletGroup(12)
sage: G.list()
[Dirichlet character modulo 12 of conductor 1 mapping 7 |--> 1, 5 |--> 1,
Dirichlet character modulo 12 of conductor 4 mapping 7 |--> -1, 5 |--> 1,
Dirichlet character modulo 12 of conductor 3 mapping 7 |--> 1, 5 |--> -1,
Dirichlet character modulo 12 of conductor 12 mapping 7 |--> -1, 5 |--> -1]
sage: G.gens()
(Dirichlet character modulo 12 of conductor 4 mapping 7 |--> -1, 5 |--> 1,
Dirichlet character modulo 12 of conductor 3 mapping 7 |--> 1, 5 |--> -1)
sage: len(G)
4
```

Une fois le groupe créé, on crée aussitôt un élément et on calcule avec lui.

```
sage: G = DirichletGroup(21)
sage: chi = G.1; chi
Dirichlet character modulo 21 of conductor 7 mapping 8 |--> 1, 10 |--> zeta6
sage: chi.values()
[0, 1, zeta6 - 1, 0, -zeta6, -zeta6 + 1, 0, 0, 1, 0, zeta6, -zeta6, 0, -1,
 0, 0, zeta6 - 1, zeta6, 0, -zeta6 + 1, -1]
sage: chi.conductor()
7
sage: chi.modulus()
21
sage: chi.order()
6
sage: chi(19)
-zeta6 + 1
```

```
sage: chi(40)
-zeta6 + 1
```

Il est possible aussi de calculer l'action d'un groupe de Galois $\mathrm{Gal}(\mathbf{Q}(\zeta_N)/\mathbf{Q})$ sur l'un de ces caractères, de même qu'une décomposition en produit direct correspondant à la factorisation du module.

```
sage: chi.galois_orbit()
[Dirichlet character modulo 21 of conductor 7 mapping 8 |--> 1, 10 |--> -zeta6 + 1,
 Dirichlet character modulo 21 of conductor 7 mapping 8 |--> 1, 10 |--> zeta6]

sage: go = G.galois_orbits()
sage: [len(orbit) for orbit in go]
[1, 2, 2, 1, 1, 2, 2, 1]

sage: G.decomposition()
[
Group of Dirichlet characters modulo 3 with values in Cyclotomic Field of order 6 and␣
↪degree 2,
Group of Dirichlet characters modulo 7 with values in Cyclotomic Field of order 6 and␣
↪degree 2
]
```

Construisons à present le groupe de caractères de Dirichlet modulo 20, mais à valeur dans $\mathbf{Q}(i)$:

```
sage: K.<i> = NumberField(x^2+1)
sage: G = DirichletGroup(20,K)
sage: G
Group of Dirichlet characters modulo 20 with values in Number Field in i with␣
↪defining polynomial x^2 + 1
```

Nous calculons ensuite différents invariants de `G` :

```
sage: G.gens()
(Dirichlet character modulo 20 of conductor 4 mapping 11 |--> -1, 17 |--> 1,
Dirichlet character modulo 20 of conductor 5 mapping 11 |--> 1, 17 |--> i)

sage: G.unit_gens()
(11, 17)
sage: G.zeta()
i
sage: G.zeta_order()
4
```

Dans cet exemple, nous créons un caractère de Dirichlet à valeurs dans un corps de nombres. Nous spécifions ci-dessous explicitement le choix de la racine de l'unité par le troisième argument de la fonction `DirichletGroup`.

```
sage: x = polygen(QQ, 'x')
sage: K = NumberField(x^4 + 1, 'a'); a = K.0
sage: b = K.gen(); a == b
True
sage: K
Number Field in a with defining polynomial x^4 + 1
sage: G = DirichletGroup(5, K, a); G
Group of Dirichlet characters modulo 5 with values in the group of order 8 generated␣
↪by a in Number Field in a with defining polynomial x^4 + 1
sage: chi = G.0; chi
Dirichlet character modulo 5 of conductor 5 mapping 2 |--> a^2
```

```
sage: [(chi^i)(2) for i in range(4)]
[1, a^2, -1, -a^2]
```

Ici, `NumberField(x^4 + 1, 'a')` indique à Sage d'utiliser le symbole « a » dans l'affichage de ce qu'est `K` (un corps de nombre en « a » défini par le polynôme $x^4 + 1$). Le nom « a » n'est pas déclaré à ce point. Une fois que `a = K.0` (ou de manière équivalente `a = K.gen()`) est évalué, le symbole « a » représente une racine du polynôme générateur $x^4 + 1$.

2.13.4 Formes modulaires

Sage peut accomplir des calculs relatifs aux formes modulaires, notamment des calculs de dimension, d'espace de symboles modulaires, d'opérateurs de Hecke et de décomposition.

Il y a plusieurs fonctions disponibles pour calculer la dimension d'espaces de formes modulaires. Par exemple,

```
sage: dimension_cusp_forms(Gamma0(11),2)
1
sage: dimension_cusp_forms(Gamma0(1),12)
1
sage: dimension_cusp_forms(Gamma1(389),2)
6112
```

Nous illustrons ci-dessous le calcul des opérateurs de Hecke sur un espace de symboles modulaires de niveau 1 et de poids 12.

```
sage: M = ModularSymbols(1,12)
sage: M.basis()
([X^8*Y^2,(0,0)], [X^9*Y,(0,0)], [X^10,(0,0)])
sage: t2 = M.T(2)
sage: t2
Hecke operator T_2 on Modular Symbols space of dimension 3 for Gamma_0(1)
of weight 12 with sign 0 over Rational Field
sage: t2.matrix()
[ -24    0    0]
[   0  -24    0]
[4860    0 2049]
sage: f = t2.charpoly('x'); f
x^3 - 2001*x^2 - 97776*x - 1180224
sage: factor(f)
(x - 2049) * (x + 24)^2
sage: M.T(11).charpoly('x').factor()
(x - 285311670612) * (x - 534612)^2
```

Nous pouvons aussi créer des espaces pour $\Gamma_0(N)$ et $\Gamma_1(N)$.

```
sage: ModularSymbols(11,2)
Modular Symbols space of dimension 3 for Gamma_0(11) of weight 2 with sign
 0 over Rational Field
sage: ModularSymbols(Gamma1(11),2)
Modular Symbols space of dimension 11 for Gamma_1(11) of weight 2 with
sign 0 and over Rational Field
```

Calculons quelques polynômes caractéristiques et développements en série de Fourier.

```
sage: M = ModularSymbols(Gamma1(11),2)
sage: M.T(2).charpoly('x')
```

```
x^11 - 8*x^10 + 20*x^9 + 10*x^8 - 145*x^7 + 229*x^6 + 58*x^5 - 360*x^4
     + 70*x^3 - 515*x^2 + 1804*x - 1452
sage: M.T(2).charpoly('x').factor()
(x - 3) * (x + 2)^2 * (x^4 - 7*x^3 + 19*x^2 - 23*x + 11)
        * (x^4 - 2*x^3 + 4*x^2 + 2*x + 11)
sage: S = M.cuspidal_submodule()
sage: S.T(2).matrix()
[-2  0]
[ 0 -2]
sage: S.q_expansion_basis(10)
[
    q - 2*q^2 - q^3 + 2*q^4 + q^5 + 2*q^6 - 2*q^7 - 2*q^9 + O(q^10)
]
```

On peut même calculer des espaces de formes modulaires avec caractères.

```
sage: G = DirichletGroup(13)
sage: e = G.0^2
sage: M = ModularSymbols(e,2); M
Modular Symbols space of dimension 4 and level 13, weight 2, character
[zeta6], sign 0, over Cyclotomic Field of order 6 and degree 2
sage: M.T(2).charpoly('x').factor()
(x - zeta6 - 2) * (x - 2*zeta6 - 1) * (x + zeta6 + 1)^2
sage: S = M.cuspidal_submodule(); S
Modular Symbols subspace of dimension 2 of Modular Symbols space of
dimension 4 and level 13, weight 2, character [zeta6], sign 0, over
Cyclotomic Field of order 6 and degree 2
sage: S.T(2).charpoly('x').factor()
(x + zeta6 + 1)^2
sage: S.q_expansion_basis(10)
[
q + (-zeta6 - 1)*q^2 + (2*zeta6 - 2)*q^3 + zeta6*q^4 + (-2*zeta6 + 1)*q^5
  + (-2*zeta6 + 4)*q^6 + (2*zeta6 - 1)*q^8 - zeta6*q^9 + O(q^10)
]
```

Voici un autre exemple montrant comment Sage peut calculer l'action d'un opérateur de Hecke sur un espace de formes modulaires.

```
sage: T = ModularForms(Gamma0(11),2)
sage: T
Modular Forms space of dimension 2 for Congruence Subgroup Gamma0(11) of
weight 2 over Rational Field
sage: T.degree()
2
sage: T.level()
11
sage: T.group()
Congruence Subgroup Gamma0(11)
sage: T.dimension()
2
sage: T.cuspidal_subspace()
Cuspidal subspace of dimension 1 of Modular Forms space of dimension 2 for
Congruence Subgroup Gamma0(11) of weight 2 over Rational Field
sage: T.eisenstein_subspace()
Eisenstein subspace of dimension 1 of Modular Forms space of dimension 2
for Congruence Subgroup Gamma0(11) of weight 2 over Rational Field
sage: M = ModularSymbols(11); M
Modular Symbols space of dimension 3 for Gamma_0(11) of weight 2 with sign
```

```
0 over Rational Field
sage: M.weight()
2
sage: M.basis()
((1,0), (1,8), (1,9))
sage: M.sign()
0
```

Notons T_p les opérateurs de Hecke usuels (p premier). Comment agissent les opérateurs de Hecke T_2, T_3, T_5 sur l'espace des symboles modulaires ?

```
sage: M.T(2).matrix()
[ 3  0 -1]
[ 0 -2  0]
[ 0  0 -2]
sage: M.T(3).matrix()
[ 4  0 -1]
[ 0 -1  0]
[ 0  0 -1]
sage: M.T(5).matrix()
[ 6  0 -1]
[ 0  1  0]
[ 0  0  1]
```