Sage チュートリアル

リリース *10.5*

The Sage Group

2025 年 02 月 12 日

目 次

Sage は，代数学，幾何学，数論，暗号理論，数値解析，および関連諸分野の研究と教育を支援する，フリーなオープンソース数学ソフトウェアである．Sage の開発モデルとテクノロジーに共通する著しい特徴は，公開，共有，協調と協働の原則の徹底的な遵守である．我々の目的は言わば実用車の制作であって，車輪を再発明することではない．総合目標としているのは，Maple，Mathematica，Magma，MATLAB に代りうるフリーかつオープンソース化された実用システムの開発である．

Sage がどんなものか，短時間で知りたければ，まずこのチュートリアルを読んでみていただきたい．HTML 版と PDF 版のどちらを読んでもいいし，Sage ノートブックを経由することもできる (チュートリアル内容を対話的に実行するには，ノートブックで `Help`， 続けて `Tutorial` をクリックする).

第1章　はじめに

このチュートリアルは，３〜４時間あればじゅうぶん読み通すことができるはずだ．HTML 版と PDF 版のどちらを読んでもいいし、Sage ノートブックを経由することもできる (チュートリアル内容を Sage から対話的に実行するには，ノートブックで Help, 続けて Tutorial をクリックする).

Sage のかなりの部分が Python を使って実装されているものの，このチュートリアルを読むについては Python の予備知識はいらない．いずれは Python を勉強したくなるはずだが (とても面白い言語だ)，そんな場合のためには Python ビギナーズガイド [PyB] にリストがある優れた教材がフリーでたくさん用意されている．とにかく手っ取り早く Sage を試してみたいだけなら、このチュートリアルがよい出発点になる．例えばこんな具合だ:

```
sage: 2 + 2
4
sage: factor(-2007)
-1 * 3^2 * 223

sage: A = matrix(4,4, range(16)); A
[ 0  1  2  3]
[ 4  5  6  7]
[ 8  9 10 11]
[12 13 14 15]

sage: factor(A.charpoly())
x^2 * (x^2 - 30*x - 80)

sage: m = matrix(ZZ,2, range(4))
sage: m[0,0] = m[0,0] - 3
sage: m
[-3  1]
[ 2  3]

sage: E = EllipticCurve([1,2,3,4,5]);
sage: E
Elliptic Curve defined by y^2 + x*y + 3*y = x^3 + 2*x^2 + 4*x + 5
over Rational Field
sage: E.anlist(10)
[0, 1, 1, 0, -1, -3, 0, -1, -3, -3, -3]
sage: E.rank()
1
```

(次のページに続く)

(前のページからの続き)

```
sage: k = 1/(sqrt(3)*I + 3/4 + sqrt(73)*5/9); k
36/(20*sqrt(73) + 36*I*sqrt(3) + 27)
sage: N(k)
0.165495678130644 - 0.0521492082074256*I
sage: N(k,30)       # 精度は 30 ビット
0.16549568 - 0.052149208*I
sage: latex(k)
\frac{36}{20 \, \sqrt{73} + 36 i \, \sqrt{3} + 27}
```

1.1 インストール

まだ Sage をコンピュータにインストールしていないけれども何かコマンドを実行してはみたいというなら， http://sagecell.sagemath.org 上でオンライン実行してみる手がある．

Sage を自分のコンピュータへインストールする手順については，本家 Sage ウェブページ [SA] のドキュメンテーション部にある "Sage Installation Guide"を見てほしい．ここではいくつかコメントしておくだけにしよう．

1. Sage のダウンロード用ファイルは「バッテリー込み」である．つまり、Sage は Python, IPython, PARI, GAP, Singular, Maxima, NTL, GMP などを援用して動作するが，これらは全て Sage の配布ファイルに含まれているので別途インストールする必要はないということだ．ただし、Macaulay や KASH など一部の機能を利用するには関連するオプショナルな Sage パッケージをインストールするか，少なくとも使うコンピュータに関連プログラム群がインストール済みでなくてはならない (利用できるオプショナル・パッケージの一覧を見るには, `sage -optional` を実行するか，あるいは Sage ウェブサイトの "Download"ページを見るとよい).

2. コンパイル済みのバイナリ版 Sage(Sage サイトにある) は，ソースコードより簡単かつ速やかにインストールすることができる．ファイルを入手したら展開して `sage` コマンドを実行するだけで出来上がりだ．

3. SageTeX パッケージを使いたいのならば (SageTeX は Sage の処理結果を LaTeX 文書に埋め込み可能にしてくれる)，使用すべき TeX ディストリビューションを SageTeX に教えてやる必要がある．設定法については， Sage installation guide 中の "Make SageTeX known to TeX" を参照してほしい (ローカルシステム上の ここ にもインストールガイドがある)．手順はごく簡単で，環境変数を一つ設定するか，あるいは TeX 配下のディレクトリにファイルを 1 個コピーしてやるだけである．

SageTeX の利用に関する解説は`$SAGE_ROOT/venv/share/texmf/tex/latex/sagetex/` にある．`$SAGE_ROOT` は Sage がインストールされているディレクトリで，例えば `/opt/sage-9.6` などとなっているはずだ．

1.2 Sage の使いかた

Sage を使うには以下のようなやり方がある.

- **ノートブック グラフィカル インターフェイス:** `sage -n jupyter` を実行する. Jupyter documentation on-line を読む.
- **対話的コマンドライン:** 対話型シェル 節を参照.
- **プログラム作成:** Sage 上でインタープリタおよびコンパイラを経由してプログラムを書く (*Sage* ファイルの読み込みと結合 節と 実行形式の作成 節を参照). さらに
- **スクリプト作成:** Sage ライブラリを利用するスタンドアロン Python スクリプトを書く (スタンドアロン *Python/Sage* スクリプト 節を参照).

1.3 Sage の長期目標

- **有用性**: Sage が想定しているユーザは，数学を学ぶ学生 (高校生から大学学部生まで) と教師、そして数学の専門家である．代数、幾何、数論、解析学、数値解析などの数学諸分野には，種々の概念や量が現われてくる．Sage の狙いは、ユーザが数学上の概念や諸量の性質を探ったり，それらの働きを体験する手助けになるようなソフトウェアを提供することである．Sage を使えば，各種の数学的な実験を容易に対話的に実行することができる.
- **高速性:** 動作が高速である．Sage は GMP, PARI, GAP, NTL など高度に最適化された完成度の高いソフトウェアを援用しており，多くの場合きわめて高速に演算が実行される.
- **フリーかつオープンソース:** ソースコードは自由に入手可能で，可読性が高くなければならない．そうすればユーザは Sage が行なう処理の詳細を理解することができるし，拡張も容易になる．数学者であれば，定理を深く理解するために証明をていねいに読むか，少なくとも証明の流れ程度は追っておくはずである．計算システムのユーザも同じことで，演算処理がどのように実行されるのかソースコードを読んで把握できるようであってほしい．論文発表する仕事の計算に Sage を使っておけば，論文の読者も確実に Sage とその全ソースコードを自由に利用できることになる．Sage では，仕事に使ったバージョンを保存しておいて再配布することすら許されているのだ.
- **コンパイルが容易:** Sage は，Linux， OSX あるいは Windows のユーザがソースコードから容易にコンパイル・ビルドできるようでなくてはならない．これによりユーザは Sage システムを柔軟に修正することができる.
- **協調性:** Sage は，PARI， GAP， Singular， Maxima， KASH， Magma， Maple，さらに Mathematica など多くのコンピュータ代数システムとの頑健なインターフェイスを提供する．Sage の狙いは、既存の数学ソフトウェアとの統合と拡張である.
- **豊富な関連文書:** チュートリアル，プログラミングガイド，レファレンスマニュアル，ハウツー類が揃っている．これには多数の具体例と数学的背景知識の解説も含まれる.
- **拡張性:** 新しいデータ型をゼロから定義したり，既存のデータ型を利用して作り出すことができる．さまざまな言語で書いたプログラムをシステムに組み込んで利用することも可能だ.
- **ユーザーフレンドリー**: ユーザは使用するオブジェクトにどんな属性や機能が組込まれているかを簡単に把握し，さらに関連文書やソースコードなども容易に閲覧できなくてはならない．高度のユーザーサポートも提供される.

第2章　Sage観光ツアー

この節では，Sage を使えばどんなことできるのか，その一端をご覧にいれる．「…を作るにはどうやるの？」という質問全般に答える "Sage Constructions"ドキュメントには，さらに豊富な実例が用意されている．加えて "Sage Reference Manual"では，数千の具体例が見つかるはずだ．Sage ノートブックの Help リンクをクリックすれば、このツアーの内容を対話的に確認することもできる．

(このチュートリアルを Sage ノートブックから閲覧している場合，入力セルの内容を実行するには shift-enter と押せばよい． shift-enter を押して実行する前に入力セルの中身を変更することもできる．Mac マシンでは， shift-enter ではなく shift-return を押すことになるかもしれない．)

2.1 代入，等式と算術演算

一部に些細な例外はあるが，Sage はプログラミング言語 Python に拠っているので，Python の入門書があれば Sage を学ぶ助けになるはずだ．

Sage では代入に記号 = ，比較演算には == ， <= ， >= ， < と > を用いる．

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

基本的な演算操作は全て可能だ:

```
sage: 2**3    #  ** は「べき乗」の意味
8
sage: 2^3     #  ^ は**と同じ「べき乗」の意味 (Python では通用しない)
8
sage: 10 % 3 #  整数については % は mod，つまり余りを与える
1
sage: 10/4
5/2
```

(次のページに続く)

(前のページからの続き)

```
sage: 10//4   # 整数に対して // は商を返す
2
sage: 4 * (10 // 4) + 10 % 4 == 10
True
sage: 3^2*4 + 2%5
38
```

3^2*4 + 2%5 のような式の値は，演算子が適用される順序に依存する．付録にある 算術二項演算子の優先順位 の表を見ると演算子の適用順序が判る．

Sage では，一般によく使われる数学関数も豊富に用意されている．ここでは，ほんの一部しか例を示すことができないが:

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

いちばん最後の例のように，数式の中には近似値ではなく「厳密」な値を返してくるものもある．近似値が欲しいときは関数 n あるいはメソッド n を使う (両者とも同じ説明的な名称 numerical_approx を持つが，関数 N は n と同じものだ)．双方の精度ビット数を指定する引数 prec と有効十進桁数を指定する引数 digits はオプションで，精度 53 ビットがデフォルト値になる．

```
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 のデータはダイナミックに型付けされ，変数を通して参照される値にはデータの型情報が付随している．いずれにせよ、Python の変数はそのスコープ内ではいかなる型の変数でも保持することができる:

```
sage: a = 5   # a は整数
sage: type(a)
<class 'sage.rings.integer.Integer'>
sage: a = 5/3  # a は有理数になった
sage: type(a)
```

(次のページに続く)

(前のページからの続き)

```
<class 'sage.rings.rational.Rational'>
sage: a = 'hello'  # ここで a は文字列
sage: type(a)
<... 'str'>
```

プログラミング言語 C では変数がスタティックに型付けされるから振舞いはかなり異なっていて，整数 (int) 型として宣言された変数は同じスコープ内では整数しか保持できない.

2.2 ヘルプの利用

Sage には充実したドキュメントが組込まれていて，(例えば) 関数や定数の名前に続けて疑問符 ? を入力するだけで解説を呼び出すことができる:

```
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)
```

(次のページに続く)

(前のページからの続き)

```
        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)
        0.69314718055994530941723212145817656807   # 同じく 64 ビット
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]
```

(次のページに続く)

(前のページからの続き)

```
        [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]
```

さらに Sage では，関数名の最初の何文字かを入力して TAB キーを打つと候補が表示される，タブ補完機能を使うことができる．例えば ta と入力して TAB キーを押せば、Sage は tachyon，tan，tanh,taylor を表示してくる．この機能を使えば関数やオブジェクトなどの名称を容易に探しあてることができるはずだ．

2.3 関数, インデントおよび数え上げ

Sage で新しい関数を定義するには， def 命令を使い、変数名を並べた後にコロン : を付ける．以下に例を示そう:

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

注意 : チュートリアルをどの形式で閲覧しているかにもよるが，上のコード例の 2 行目には四つのドット .... が見えているはずだ．この四点ドットは入力しないこと．四点ドットは，コードがインデントされていることを示しているだけだからだ．そうした場面では，常に構文ブロックの末尾で一度 Return/Enter を押して空行を挿入し，関数定義を終了してやらねばならない．

引数の型を指定していないことに注意．複数個の引数を指定し，その各々にデフォルト値を割り当てることもできる．例えば、以下の関数では引数 divisor の値が指定されない場合， divisor=2 がデフォルト値になる:

```
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
```

関数を呼び出すときには，特定の引数へ明示的に値を代入することもできる．引数への明示的な代入を行なう場合，関数に渡す引数の順序は任意になる:

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

Python の構文ブロックは，他の多くの言語のように中括弧や begin-end で括ることによって示されるわけではない．代りに、Python では構文構造に正確に対応したインデンテーション (字下げ) によってブロックを示す．次の例は， return ステートメントが関数内の他のコードと同じようにインデントされていないために，文法エラーになっている:

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

しかし正しくインデントし直せば，この関数はきちんと動くようになる:

```
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]
```

行末にセミコロンは必要ない．ほとんどの場合，行末は改行記号によって示される．しかし，1 行に複数のステートメントをセミコロンで区切って書き込むこともできる:

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

1 行の内容を複数行に分けて書きたければ，各行末にバックスラッシュをつければよい:

```
sage: (2 +
....:    3)
5
```

Sage では，一定範囲の整数の数え上げによって反復を制御する．例えば，以下のコードの 1 行目は C++や Java における for(i=0; i<3; i++) と全く同じ意味になる:

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

(次のページに続く)

(前のページからの続き)

```
0
1
2
```

次の例の最初の行は， `for(i=2;i<5;i++)` に対応している．

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

range の三つ目の引数は増分値を与えるので, 次のコードは `for(i=1;i<6;i+=2)` と同じ意味になる．

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

Sage で計算した値を見映えよく表形式に並べて表示したくなることもあるだろう．そんなとき役立つのが文字列フォーマットだ．以下では，各列幅がきっかり 6 文字分の表を作り，整数とその 2 乗、3 乗の値を並べてみる．

```
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
```

Sage における最も基本的なデータ構造はリストで，名前の示すとおり任意のオブジェクトの並びのことである．上で使った range も、整数のリストを生成している：

```
sage: list(range(2,10))
[2, 3, 4, 5, 6, 7, 8, 9]
```

もう少し複雑なリストの例として:

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

多くのプログラミング言語と同じように，リスト添字は 0 から始まる．

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

v の長さを取得するには len(v) を使い， v の末尾に新しいオブジェクトを追加するには v.append(obj) ，そして v の i 番目の要素を削除するためには del v[i] とする:

```
sage: len(v)
4
sage: v.append(1.5)
sage: v
[1, 'hello', 2/3, sin(x^3), 1.500000000000000]
sage: del v[1]
sage: v
[1, 2/3, sin(x^3), 1.500000000000000]
```

もう一つの重要なデータ構造がディクショナリ (連想配列とも言う) である．ディクショナリの振舞いはリストに似ているが，異なるのはその添字付けに基本的にいかなるオブジェクトでも使うことができる点だ (ただし添字は不変性オブジェクトでなければならない).

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

クラスを使えば自分で新しいデータ型を定義することも可能だ．クラスによる数学オブジェクトのカプセル化は，Sage プログラムを見通しよく構成するための強力な方法である．以下では， n までの正の偶数のリストを表すクラスを定義してみよう．定義には組み込み型 list を使っている．

```
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."
```

オブジェクトの生成時には初期化のために __init__ メソッドが呼ばれ， __repr__ メソッドはオブジェクトを印字する．__init__ メソッドの 2 行目ではリストコンストラクタを使った．クラス Evens のオブジェクトを生成するには以下のようにする:

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

e の印字には，我々が定義した __repr__ メソッドが使われている．オブジェクトに含まれる偶数のリス

トを表示するには， list 関数を使う:

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

n 属性にアクセスし、 e をリストのように扱うこともできる．

```
sage: e.n
10
sage: e[2]
6
```

2.4 代数と微積分の基本

Sage では，初等的な代数と微積分に関連した多様な演算を実行することができる．例として，方程式の解を求める，微分や積分を計算する，ラプラス変換の実行などがあげられる．Sage Constructions には，さらに多様な具体例が盛られている．

2.4.1 方程式を解く

方程式を解析的に解く

solve 関数を使って方程式の解を求めることができる．これを使うには、まず変数を定義し、ついで対象とする方程式 (または方程式系) と解くべき変数を solve の引数として指定する:

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

解くべき変数を変更して，解を他の変数で表わすこともできる:

```
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)]
```

多変数の方程式を解くことも可能だ:

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

次の Jason Grout による例題では、Sage を使って連立非線形方程式を解く．まず，この連立方程式の解を記号的に求めてみよう:

```
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]]
```

解の数値近似を求めるには，やり方を変えて:

```
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]]
```

n 関数は解の数値的近似値を表示する. n の引数は数値精度を表わすビット数を指定している.

方程式を数値的に解く

目的の方程式 (または方程式系) に対し solve では厳密解を求めることができないというのは珍しいことではない．そうした場合には find_root を使って数値解を求めることができる．例えば，以下に示す方程式については solve は何も役に立つことを教えてくれない.

```
sage: theta = var('theta')
sage: solve(cos(theta)==sin(theta), theta)
[sin(theta) == cos(theta)]
```

しかし代りに find_root を使えば， $0 < \phi < \pi/2$ の範囲で上の方程式の数値解を求めることができる.

```
sage: phi = var('phi')
sage: find_root(cos(phi)==sin(phi),0,pi/2)
0.785398163397448...
```

2.4.2 微分，積分，その他

Sage で多様な関数の微分と積分を計算することができる．例えば $\sin(u)$ を u で微分するには，以下のようにする:

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

$\sin(x^2)$ の 4 次微分を計算するには:

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

$x^2 + 17y^2$ の x と y それぞれによる偏微分を計算するには:

```
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
```

次は不定積分と定積分だ． $\int x\sin(x^2)\,dx$ と $\int_0^1 \frac{x}{x^2+1}\,dx$ を計算してみよう．

```
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)
```

$\frac{1}{x^2-1}$ の部分分数展開を求めるには:

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

2.4.3 微分方程式を解く

Sage を使って常微分方程式を研究することもできる． $x' + x - 1 = 0$ を解くには:

```
sage: t = var('t')            # 変数 t を定義
sage: x = function('x')(t)     # x を t の関数とする
sage: DE = diff(x, t) + x - 1
sage: desolve(DE, [x,t])
(_C + e^t)*e^(-t)
```

ここで Sage は Maxima [Max] とインターフェイスしているので，その出力もこれまで見てきた Sage の出力とは若干違っている．上の結果は，上の微分方程式の一般解が $x(t) = e^{-t}(e^t + c)$ であることを示している．

ラプラス変換を実行することができる．$t^2e^t - \sin(t)$ のラプラス変換は以下のような手順を踏む:

```
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
```

もう少し手間のかかる問題を考えてみよう．左端が壁に固定された連成バネ各々の、平衡位置からの変位

```
|------\/\/\/\/\---|mass1|----\/\/\/\/\/----|mass2|
         spring1                spring2
```

は、連立 2 階微分方程式

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

でモデル化される．ここで m_i はおもり i の質量， x_i はそのおもり i の平衡位置からの変位，そして k_i はバネ i のバネ定数である．

例題: 上の問題で各パラメータの値を $m_1 = 2, m_2 = 1, k_1 = 4, k_2 = 2, x_1(0) = 3, x_1'(0) = 0, x_2(0) = 3, x_2'(0) = 0$ と置き，Sage を使って解いてみよう．

解法: まず 1 番目の方程式をラプラス変換する (記号は $x = x_1, y = x_2$ に変える):

```
sage: de1 = maxima("2*diff(x(t),t, 2) + 6*x(t) - 2*y(t)")
sage: lde1 = de1.laplace("t","s"); lde1.sage()
2*s^2*laplace(x(t), t, s) - 2*s*x(0) + 6*laplace(x(t), t, s) - 2*laplace(y(t), t, s) -
 2*D[0](x)(0)
```

この出力は読みにくいけれども，意味しているのは

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

ということだ (ここでは小文字名の関数 $x(t)$ のラプラス変換が大文字名の関数 $X(s)$ となっている)．2 番目の方程式もラプラス変換してやると:

```
sage: t,s = SR.var('t,s')
sage: x = function('x')
sage: y = function('y')
sage: f = 2*x(t).diff(t,2) + 6*x(t) - 2*y(t)
sage: f.laplace(t,s)
2*s^2*laplace(x(t), t, s) - 2*s*x(0) + 6*laplace(x(t), t, s) - 2*laplace(y(t), t, s) -
 2*D[0](x)(0)
```

意味するところは

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

初期条件 $x(0), x'(0), y(0)$ ，および $y'(0)$ を代入して得られる 2 つの方程式を X と Y について解く:

```
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)]]
```

この解の逆ラプラス変換を行なうと:

```
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)
```

というわけで，求めていた解は

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

これを媒介変数プロットするには

```
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)
```

各成分ごとにプロットするには

```
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)
```

プロットについては **プロットする** 節の，もう少し詳しい説明を見てほしい．微分方程式については [NagleEtAl2004] の 5.5 節にもっと詳しい解説がある．

2.4.4 オイラーによる連立微分方程式の解法

次の例では，1 階および 2 階微分方程式に対するオイラーの解法を具体的に解説する．手始めに 1 階微分方程式に対する解法の基本的アイデアを復習しておこう．初期値問題が

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

のような形式で与えられており， $b > a$ を満足する $x = b$ における解の近似値を求めたいものとする．

微分係数の定義から

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

ここで $h > 0$ は与えるべき小さな量である．この近似式と先の微分方程式を組み合わせると $f(x, y(x)) \approx \frac{y(x+h)-y(x)}{h}$ が得られる．これを $y(x+h)$ について解くと:

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

(他にうまい呼び方も思いつかないので) $h \cdot f(x, y(x))$ を "補正項" と呼び， $y(x)$ を y の "更新前項 (old)", $y(x+h)$ を y の "更新後項 (new)"と呼ぶことにすると，上の近似式を

$$y_{new} \approx y_{old} + h \cdot f(x, y_{old}).$$

と表わすことができる．

ここで a から b までの区間を n ステップに分割すると $h = \frac{b-a}{n}$ と書けるから，ここまでの作業から得られた情報を整理して以下の表のようにまとめることができる．

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

我々の目標は，この表の空欄を上から一行づつ全て埋めていき，最終的に $y(b)$ のオイラー法による近似である???に到達することである．

連立微分方程式に対する解法もアイデアは似ている．

例題: $z'' + tz' + z = 0, z(0) = 1, z'(0) = 0$ を満足する $t = 1$ における $z(t)$ を，4 ステップのオイラー法を使って数値的に近似してみよう．

ここでは問題の 2 階常微分方程式を ($x = z, y = z'$ として) 二つの 1 階微分方程式に分解してからオイラー法を適用することになる。

```
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
```

したがって， $z(1) \approx 0.65$ が判る．

点 (x, y) をプロットすれば、その曲線としての概形を見ることができる．それには関数 `eulers_method_2x2_plot` を使うが，その前に三つの成分 $(t,\ x,\ y)$ からなる引数を持つ関数 f と g を定義しておかなければならない．

```
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)
```

この時点で， `P` は 2 系列のプロットを保持していることになる．x と t のプロットである `P[0]` ， および y と t のプロットである `P[1]` である．これら二つをプロットするには、次のようにする:

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

(プロットの詳細については プロットする 節を参照．)

2.4.5 特殊関数

数種類の直交多項式と特殊関数が，PARI [GAP] および Maxima [Max] を援用して実装されている．詳細については Sage レファレンスマニュアルの“Orthogonal polynomials"(直交多項式) と“Special functions"(特殊関数) を参照してほしい．

```
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
```

ここで注意したいのは，Sage ではこれらの関数群が専ら数値計算に便利なようにラップ (wrap) されている点だ．記号処理をする場合には，以下の例のように Maxima インターフェイスをじかに呼び出してほしい．

```
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 プロットする

Sage を使って 2 次元および 3 次元のグラフを作成することができる．

2.5.1 2 次元プロット

Sage の 2 次元プロット機能を使うと，円，直線，多辺形の描画はもちろん，直交座標系における関数のプロット，極座標プロット、等高線プロット，ベクトル場プロットを行うことができる．以下では、その具体例を見ていくことにしよう．さらに Sage によるプロットの具体例を見たければ， 微分方程式を解く 節と *Maxima* 節，および Sage Constructions を参照してほしい．

次のコマンドは原点を中心とした半径 1 の黄色い円を描く:

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

円を塗りつぶすこともできる：

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

円を生成して，それを変数に収めておくこともできる．ただし，それだけでは描画は実行されない．

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

描画するには，以下のように `c.show()` または `show(c)` などとする:

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

かわりに `c.save('filename.png')` などとすれば，プロット `c` は画像としてファイルに保存される．

ところで「円」がどちらかと言えば「楕円」に見えるのは，軸が縦横で異なってスケールされるからだ．これを直すには:

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

同じことはコマンド `show(c, aspect_ratio=1)` としても可能だし，画像ファイルとして保存したければ `c.save('filename.png', aspect_ratio=1)` と実行してもよい．

初等関数のプロットも簡単だ:

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

変数を特定しておけば，媒介変数プロットも可能になる:

```
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
```

プロットにおける座標軸の交点は，それがグラフの描画範囲にない限り表示されないことに注意しておいてほしい．描画範囲として十分大きな値を指定するために，科学記法 (指数記法) を用いる必要があるかもしれない．

```
sage: plot(x^2,(x,300,500))
Graphics object consisting of 1 graphics primitive
```

複数のプロットをまとめて描画するには，それらを足し合わせればよい:

```
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)
```

塗りつぶし図形を生成するには，まず図形の頂点からなるリストを生成し (以下の例の L)，次に polygon コマンドで頂点をつないで閉領域を作るとよい．ここでは，例として緑色の三角形を描画してみる:

```
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: p = polygon(L, rgbcolor=(1/8,3/4,1/2))
sage: p
Graphics object consisting of 1 graphics primitive
```

show(p, axes=false) と入力して座標軸なしの図を表示してみよう．

図形プロットに文字列を加えることができる:

```
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)
```

次に出てくる arcsin のグラフは、微積分の教師が黒板にたびたび描くものだ．主値だけではなく複数の分岐を含めてプロットするには， x が -2π から 2π までの $y = \sin(x)$ のグラフを傾き 45 度の線について反転させて描いてやればよい．これを Sage で実行するには，以下のコマンドを使う:

```
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
```

正接 (tan) 関数は sin 関数よりも値域が広いので，arcsin と同じ作戦で逆正接関数のグラフを描くには x -軸の最大値と最小値を調節してやる必要がある:

```
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 では、(対象となる関数は限られるが) 極座標プロット、等高線プロット、ベクトル場プロットも可能だ．ここでは，例として等高線プロットを見ておこう:

```
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 3 次元プロット

Sage では 3 次元プロットも作成することができる．ノートブック上でも REPL(コマンドライン) 上でも，3 次元プロットの表示はデフォルトでオープンソースパッケージ [ThreeJS] によって行なわれる．Jmol ではマウスによる描画の回転と拡大縮小が可能だ．

plot3d を使って $f(x,y) = z$ 形式の関数をプロットしてみよう:

```
sage: x, y = var('x,y')
sage: plot3d(x^2 + y^2, (x,-2,2), (y,-2,2))
Graphics3d Object
```

代りに parametric_plot3d を使い，x, y, z 各々が 1 あるいは 2 個のパラメター (いわゆる媒介変数，記号 u や v などが使われることが多い) で決定されるパラメトリック曲面として描画することもできる．上の関数を媒介変数表示してプロットするには:

```
sage: u, v = var('u, v')
sage: f_x(u, v) = u
sage: f_y(u, v) = v
sage: f_z(u, v) = u^2 + v^2
sage: parametric_plot3d([f_x, f_y, f_z], (u, -2, 2), (v, -2, 2))
Graphics3d Object
```

Sage で 3 次元曲面プロットを行うための第三の方法が implicit_plot3d の使用で，これは (空間内の点の集合を定義する) $f(x,y,z) = 0$ を満足する関数の等高線を描画する．ここでは古典的な表式を使って球面を作画してみよう:

```
sage: x, y, z = var('x, y, z')
sage: implicit_plot3d(x^2 + y^2 + z^2 - 4, (x,-2, 2), (y,-2, 2), (z,-2, 2))
Graphics3d Object
```

以下で，さらにいくつかの 3 次元プロットを示しておこう:

ホィットニーの傘:

```
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
```

クロスキャップ (十字帽):

```
sage: u, v = var('u,v')
sage: fx = (1+cos(v))*cos(u)
sage: fy = (1+cos(v))*sin(u)
```

(次のページに続く)

(前のページからの続き)

```
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
```

ねじれトーラス (twisted torus):

```
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
```

レムニスケート (連珠形, lemniscate):

```
sage: x, y, z = var('x,y,z')
sage: f(x, y, z) = 4*x^2 * (x^2 + y^2 + z^2 + z) + y^2 * (y^2 + z^2 - 1)
sage: implicit_plot3d(f, (x, -0.5, 0.5), (y, -1, 1), (z, -1, 1))
Graphics3d Object
```

2.6 関数まわりの注意点

関数の定義については紛らわしい側面があって，微積分やプロットなどを行なう際に問題になることがある．この節で，関連する諸問題について検討してみたい．

Sage で「関数」と呼ばれるべきものを定義する方法は何通りもある:

1. 関数, インデントおよび数え上げ 節で解説されている方法で，Python 関数を定義する．こうして定義された関数はプロット可能だが，微分積分演算はできない．

```
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
```

最終行の書法に注目していただきたい．これを `plot(f(z), 0, 2)` としていたら，エラーになっていたはずである．`z` は `f` 定義におけるダミー変数であって，定義ブロックの外では未定義になるからだ．むろん `f(z)` のみを実行してもエラーになる．以下のようにすると切り抜けられるが，どんな場合でも通用するとは限らないので要注意だ (下の第 4 項を参照).

```
sage: var('z')   # z を変数として定義
z
sage: f(z)
z^2
sage: plot(f(z), 0, 2)
Graphics object consisting of 1 graphics primitive
```

こうすると f(z) はシンボリック表現になる．シンボリック表現については，次の項目で解説する．

2. 「呼び出し可能シンボリック表現」(callable symbolic expression) を定義する．これはプロットおよび微分積分演算が可能である．

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

g は呼び出し可能シンボリック表現だが， g(x) の方はこれに関係はあっても異なる種類のオブジェクトである．やはりプロットと微積分などが可能なのだが，違っている点もあるので注意を要する．以下の第 5 項で具体的に説明する．

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

3. Sage で定義済みの「初等関数」(calculus function) を使う．これらはプロット可能で，ちょっと工夫すると微分積分もできるようになる．

```
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))
```

(次のページに続く)

(前のページからの続き)

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

そのままでは sin は微分演算を受けつけない．少なくとも cos にはならない．

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

sin そのままではなく f = sin(x) とすると微積分を受けつけるようになるが， もっと手堅いのは f(x) = sin(x) として呼び出し可能シンボリック表現を定義することである．

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

まだ注意を要する点が残っているので，説明しておこう:

4. 意図しない評価が起きることがある．

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

ここで plot(h(x), 0, 4) を実行すると，プロットされるのは $y = x - 2$ で，複数行にわたって定義しておいた h ではない．原因を考えてみよう．コマンド plot(h(x), 0, 4) が実行されると，まず h(x) が評価されるが， これは x が関数 h(x) に突っ込まれ x<2 が評価されることを意味する．

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

シンボリック式が評価される際， h の定義の場合と同じように，その式が明らかに真でないかぎり戻り値は偽になる．したがって h(x) は x-2 と評価され，プロットされるのも x-2 になるわけである．

解決策はというと， plot(h(x), 0, 4) ではなく

```
sage: plot(h, 0, 4)
Graphics object consisting of 1 graphics primitive
```

を実行せよ，ということになる．

5. 意図せず関数が定数になってしまう．

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

問題は，例えば g(3) などと実行するとエラーになって， "ValueError: the number of arguments must be less than or equal to 0."と文句をつけてくることだ．

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

g は関数ではなく定数になっているので，変数を持たないから何も値を受けつけない．

解決策は何通りかある．

- f を最初にシンボリック表式として定義しておく．

```
sage: f(x) = x        #  'f = x'とはしない
sage: g = f.derivative()
sage: g
x |--> 1
sage: g(3)
1
sage: type(g)
<class 'sage.symbolic.expression.Expression'>
```

- または f の定義は元のまま g をシンボリック表式として定義する．

```
sage: f = x
sage: g(x) = f.derivative()  # 'g = f.derivative()'とするかわり
sage: g
x |--> 1
sage: g(3)
1
sage: type(g)
<class 'sage.symbolic.expression.Expression'>
```

- または f と g の定義は元のまま，代入すべき変数を特定する．

```
sage: f = x
sage: g = f.derivative()
sage: g
1
sage: g(x=3)    # たんに'g(3)'とはしない
1
```

おしまいになったが， `f = x` と `f(x) = x` 各々に対する微分の相違点を示す方法がまだあった．

```
sage: f(x) = x
sage: g = f.derivative()
sage: g.variables()  # g に属する変数は?
()
sage: g.arguments()  # g に値を送り込むための引数は?
(x,)
sage: f = x
sage: h = f.derivative()
sage: h.variables()
()
sage: h.arguments()
()
```

ここの例から判るように， `h(3)` がエラーになるのは，そもそも `h` が引数を受けつけないためである．

2.7 基本的な環

行列やベクトル，あるいは多項式を定義する際，定義の土台となる「環」を指定することが有利に働くだけではなく，不可欠ですらある場合がある．**環** (ring) とは，和と積が質よくふるまうことが保証されている数学的構造物のことだ．これまで聞いたことがなかったとしても，以下にあげる四つの広く使われている環についてだけは知っておくべきだろう:

- 整数 $\{..., -1, 0, 1, 2, ...\}$, Sage では `ZZ` で呼ぶ．
- 有理数 -- つまり整数による分数あるいは比 -- `QQ` で呼ぶ．
- 実数， `RR` で呼ぶ．
- 複素数， `CC` で呼ぶ．

これらの環の違いについて意識しておきたいのは，例えば同じ多項式であってもそれがどの環の上で定義されるかによって異なった扱いがなされることがあるからだ．具体例をあげると，多項式 $x^2 - 2$ は二つの根 $\pm\sqrt{2}$ を持つ．両方とも有理数ではないから，もし有理係数の多項式として扱うのならばこの多項式は因数分解できない．しかし，実係数の多項式として扱うのなら分解可能だ．とすれば，求める情報を確実に得られる環を指定しておきたくなるのも当然だろう．以下の二つのコマンドは，有理係数と実係数の多項式の集合を定義する．集合はそれぞれ `ratpoly` と `realpoly` と命名されているが，大切なのはその名前ではない．むしろ注意すべきは，各々の集合で使われる **変数名** を定義しているのが文字列 `.<t>` と `.<z>` であることである．

```
sage: ratpoly.<t> = PolynomialRing(QQ)
sage: realpoly.<z> = PolynomialRing(RR)
```

ここで $x^2 - 2$ の因数分解の様子を見てみよう:

```
sage: factor(t^2-2)
t^2 - 2
```

(次のページに続く)

(前のページからの続き)

```
sage: factor(z^2-2)
(z - 1.41421356237310) * (z + 1.41421356237310)
```

以上と同様の注意点は，行列に対しても通用する．既約行階段形式は元の行列がどんな環の上で定義されているかに依存するし，固有値と固有ベクトルも同様である．多項式の構成法については **多項式** 節に，行列に関しては **線形代数** 節にもっと詳しい解説がある．

記号 `I` は -1 の平方根を表わし， `i` は `I` と同義である．言うまでもなく，これは有理数ではない．

```
sage: i  # -1 の平方根
I
sage: i in QQ
False
```

上のコードは，変数 `i` が反復制御に使われるなどして，違った値が割り当てられていたら予期した結果にならないことがある．そんな場合には，次のように入力すると元の虚数単位 `i` として復活する:

```
sage: reset('i')
```

複素数の定義には一つ微妙な点がある．すでに述べたように記号 `i` は -1 の平方根を表わすが，これはあくまでも代数的な数字で -1 の **形式的** な平方根である．一方， `CC(i)` または `CC.0` を実行すると戻ってくるのは -1 の **複素** 平方根である．異なる種類の数同士の演算は，いわゆる「型強制」(coercion) によって可能になる．これについては **ペアレント，型変換および型強制** 節を参照.

```
sage: i = CC(i)        # 複素浮動小数点数
sage: i == CC.0
True
sage: a, b = 4/3, 2/3
sage: z = a + b*i
sage: z
1.33333333333333 + 0.666666666666667*I
sage: z.imag()         # 虚部
0.666666666666667
sage: z.real() == a    # 比較の前に自動的に型変換される
True
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        # 加算の前に自動型変換
0.766666666666667
sage: 0.1 + 2/3        # Sage の型変換規則は対称的である
```

(次のページに続く)

(前のページからの続き)

```
0.766666666666667
```

もう少し Sage で基本となる環の実例をあげておこう．先に示したように，有理数の環は QQ あるいは RationalField() で参照することができる (**体** *(field)* とは積演算が可換で，非零元が常に逆元をもつ環のことである．したがって有理数は体を構成するが整数は体にならない).

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

有理数でもある小数は有理数に「型強制」することができるから，例えば小数 1.2 は QQ に属すると見なされる (**ペアレント，型変換および型強制** 節を参照)．しかし， π と $\sqrt{2}$ は有理数にはならない:

```
sage: 1.2 in QQ
True
sage: pi in QQ
False
sage: pi in RR
True
sage: sqrt(2) in QQ
False
sage: sqrt(2) in CC
True
```

より高度な数学で利用するため，Sage には有限体，p-進整数，代数環，多項式環，そして行列環などが用意されている．以下ではその中のいくつかを構成してみよう．

```
sage: GF(3)
Finite Field of size 3
sage: GF(27, 'a')  # 素数体でなければ生成元の命名が必要
Finite Field in a of size 3^3
sage: Zp(5)
5-adic Ring with capped relative precision 20
sage: sqrt(3) in QQbar # QQ の代数的閉包 (拡大)
True
```

2.8 線形代数

Sage には線形代数で常用されるツールが揃っていて，例えば行列の特性多項式の計算、階段形式、跡 (トレース)、各種の分解などの操作が可能である．

行列の生成と積演算の手順は，簡単かつ自然なものだ:

```
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]
```

Sage では，行列 A の核空間は「左」核空間，すなわち $wA = 0$ を満足するベクトル w が張る空間をさす．

行列方程式もメソッド `solve_right` を使って簡単に解くことができる．`A.solve_right(Y)` と実行すれば， $AX = Y$ を満たす行列 (またはベクトル) X が得られる．

```
sage: Y = vector([0, -4, -1])
sage: X = A.solve_right(Y)
sage: X
(-2, 1, 0)
sage: A * X   # 解のチェック...
(0, -4, -1)
```

解がない場合は，Sage はエラーを返してくる:

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

同様にして， $XA = Y$ を満足する X を求めるには `A.solve_left(Y)` とすればよい．

Sage は固有値と固有ベクトルの計算もしてくれる:

```
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)
```

(次のページに続く)

(前のページからの続き)

```
], 1), (-2, [
(1, -1)
], 1)]
```

(eigenvectors_left の出力は，三つ組タプル (固有値, 固有ベクトル, 多重度) のリストになっている．)

QQ または RR 上の固有値と固有ベクトルは Maxima を使って計算することもできる (後半の *Maxima* 節を参照).

基本的な環 節で述べたように，行列の性質の中には，その行列がどんな環の上で定義されているかに影響を受けるものがある．以下では， matrix コマンドの最初の引数を使って，Sage に生成すべき行列が整数の行列 (ZZ の場合) なのか，有理数の行列 (QQ) なのか，あるいは実数の行列 (RR) なのかを指定している．

```
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]
```

浮動小数点型の実数または複素数上で定義された行列の固有値と固有ベクトルを計算するためには，対象とする行列を，それぞれ RDF (Real Double Field) または CDF (Complex Double Field) 上で定義しておかなければならない．もし環を指定しないまま行列に浮動小数点型の実数あるいは複素数が使われる場合，その行列はデフォルトでそれぞれ RR あるいは CC 体上で定義される．この場合，以下の演算があらゆる状況で実行可能になるとは限らない.

```
sage: ARDF = matrix(RDF, [[1.2, 2], [2, 3]])
sage: ARDF.eigenvalues()   # abs tol 1e-10
[-0.09317121994613098, 4.293171219946131]
sage: ACDF = matrix(CDF, [[1.2, I], [2, 3]])
sage: ACDF.eigenvectors_right()   # abs tol 1e-10
[(0.881845698329 - 0.820914065343*I, [(0.750560818381, -0.616145932705 + 0.
↪238794153033*I)], 1),
(3.31815430167 + 0.820914065343*I, [(0.145594698293 + 0.37566908585*I, 0.
↪915245825866)], 1)]
```

2.8.1 行列の空間

有理数型の要素からなる 3×3 行列の空間 $\mathrm{Mat}_{3\times 3}(\mathbf{Q})$ を生成してみよう:

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

(3 行 4 列の行列空間を生成したければ `MatrixSpace(QQ,3,4)` とする．列数を省略するとデフォルトで行数に合わせられるから， `MatrixSpace(QQ,3)` は `MatrixSpace(QQ,3,3)` と同じ意味になる．) 行列の空間は標準基底を備えており:

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

`M` の元の一つとして行列を生成してみよう．

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

ついで，その既約階段形式と核を計算する．

```
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]
```

次に，有限体上で定義された行列による計算を実行してみる．

```
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]
```

(次のページに続く)

(前のページからの続き)

```
[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)]
```

上に現れた行ベクトル系 (rows) によって張られる $\mathbf{F}_2$ の部分空間を作成する.

```
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]
```

Sage は S の基底として， S の生成元行列の既約階段形式の非ゼロ行を使用している.

2.8.2 疎行列の線形代数

Sage では PID(単項イデアル整域) 上の疎行列に関する線形代数を扱うことができる.

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

Sage で使われている多重モジュラーアルゴリズムは，正方行列ではうまく働く (非正方行列ではいまひとつである):

```
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()
```

Python では，大文字小文字が区別されることに注意:

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

2.9 多項式

この節では，Sage 上で多項式を生成し利用する方法について解説する．

2.9.1 1 変数多項式

多項式環を生成するには，三通りのやり方がある．

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

このやり方では，多項式環を生成し，画面表示時の変数名として (文字列) t を割り当てている．ただし，これは Sage 内で使うために記号 t を定義しているわけではないから，(t^2+1 のようにして) R 上の多項式を入力するときには使えないことを注意しておく．

これに代わるやり方として

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

があるが， 記号 t については最初の方法と同じ問題が残る．

第三の，とても便利な方法が

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

とするか

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

あるいはまた

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

とすることである．

最後の方法には，多項式の変数として変数 t が定義される余禄がついてくるので，以下のようにして簡単に R の元を構成することができる．(第三の方法が，Magma におけるコンストラクタ記法によく似ている

ことに注意．Magma と同じように，Sage でも多様なオブジェクトでコンストラクタ記法を使うことができる．)

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

どの方法で多項式環を定義していても，その変数を 0 番目の生成元として取り出すことができる:

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

Sage では，複素数も多項式環と似た構成法で生成されている．すなわち，実数体上に記号 i を生成元として構成されているのが複素数であると見なすことができるのだ．それを示すのが:

```
sage: CC
Complex Field with 53 bits of precision
sage: CC.0  # CC の 0 番目の生成元
1.00000000000000*I
```

多項式環を生成するときは，以下のように環と生成元の両方を同時に作るか，あるいは生成元のみを作るか選ぶことができる:

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

最後に， $\mathbf{Q}[t]$ 上の演算を試してみよう．

```
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
```

(次のページに続く)

(前のページからの続き)

```
sage: list(F)
[(t, 5), (t^5 + 10*t + 2, 1), (t^6 + t^5 + t^4 + t^3 + t^2 + t + 1, 1)]
```

因数分解の際には，定数 (倍) 部がきちんと分離して記録されていることに注目．

仕事中に，例えば R.cyclotomic_polynomial 関数を頻繁に使う必要があったとしよう．研究発表の際には，Sage 本体だけではなく，円周等分多項式の実質的な計算に使われたコンポーネントを突きとめて引用するのが筋だ．この場合、 R.cyclotomic_polynomial?? と入力してソースコードを表示すると，すぐに f = pari.polcyclo(n) という行が見つかるはずだ．これで円周等分多項式の計算に PARI が使われていることが判ったのだから，発表の際には PARI も引用することができる．

多項式同士で割算すると，結果は (Sage が自動的に生成する) 有理数体の元になる．

```
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
```

QQ[x] の有理数体上でローラン級数による級数展開を計算することができる:

```
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)
```

異なる変数名を割り当てて生成した 1 変数多項式環は，ぞれぞれ異なる環と見なされる．

```
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
```

環は変数名によって識別される．同じ変数 x を使うと，異なる環をもう 1 つ作ったつもりでいても，そうはならないことに注意してほしい．

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

(次のページに続く)

(前のページからの続き)

```
sage: R is T
True
sage: R.0 == T.0
True
```

Sage では，任意の基底環上で巾級数環およびローラン級数環を扱うことができる．以下の例では，$\mathbf{F}_7[[T]]$ の元を生成し，ついでその逆数をとって $\mathbf{F}_7((T))$ の元を作っている．

```
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
```

巾級数環を生成するには，二重括弧を使う省略記法を用いることもできる:

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

2.9.2 多変数多項式

複数個の変数を含む多項式を扱うには，まず多項式環と変数を宣言する．

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

1 変数の多項式を定義したときと同じように，他の方法もある:

```
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
```

さらに，変数名を 1 文字にしたければ，以下のような略記法を使えばよい:

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

ここで，ちょっと計算してみよう．

```
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
```

多項式環を生成するには，もっと数学寄りの記号法を使うこともできる．

```
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,))
```

Sage の多変数多項式は，多項式に対する分配表現 (distributive representation) と Pyhon のディクショナリを使って実装されている．gcd やイデアルのグレブナー基底の計算には Singular [Si] を経由している部分がある．

```
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
```

次に， f と g から生成されるイデアル (f, g) を求めてみる．これには `(f,g)` に `R` を掛けてやるだけでよい (`ideal([f,g])` あるいは `ideal(f,g)` としても同じだ).

```
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
```

ちなみに，上のグレブナー基底はリストではなく不変性シーケンスとして与えられている．これは，基底がユニバースまたは親クラスとなっているため変更できないことを意味している (グレブナー基底が変えられてしまうとその基底系に依存するルーチン群も働かなくなるから当然のことだ).

```
sage: B.parent()
<class 'sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic'>
sage: B.universe()
Multivariate Polynomial Ring in x, y over Rational Field
```

(次のページに続く)

(前のページからの続き)

```
sage: B[1] = x
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
```

(種類はまだ十分ではないものの) 可換代数の中には Sigular 経由で Sage 上に実装されているものがある．例えば，I: の準素分解および随伴素イデアルを求めることができる:

```
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]
```

2.10 ペアレント，型変換および型強制

この節の内容はこれまでと比べるとテクニカルな感じがするかもしれない．しかし，ペアレントと型強制の意味について理解しておかないと，Sage における環その他の代数構造を有効かつ効率的に利用することができないのである．

以下で試みるのは概念の解説であって，それをどうやって実現するかまでは示すことはできない．実装法に関するチュートリアルは Sage thematic tutorial にある．

2.10.1 元

ある環を Python を使って実装する場合，その第一歩は目的の環の元 `x` を表すクラスを作り，`__add__`，`__sub__`，`__mul__` のようなダブルアンダースコア メソッド (フックメソッド) によって環の公理を保証する演算を実現することだ．

Python は (ダイナミックではあっても) 強い型付けがなされる言語なので，最初のうちは環それぞれを一つの Python クラスで実装すべきだろうと思うかもしれない．なんと言っても，Python は整数については `<int>`，実数については `<float>` といった具合に型を一つづつ備えているわけだし．しかし，このやり方はすぐに行き詰まらないわけにはいかない．環の種類が無限だからと言って，対応するクラスを無限に実装することはできないからだ．

代りに，群，環，斜体 (加除環)，可換環，体，代数系などという順で，普遍的な代数構造の実装を目的とするクラスの階層を作りあげようとする人もいるかもしれない．

しかし，これは明確に異なる環に属する元が同じ型を持ちうることを意味する．

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

一方，数学的には同等の構造物に対して，(例えば密行列と疎行列に対するように) 別々の Python クラスが異なった形で実装されることもありうる．

```
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)
<class 'sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_
↪flint'>
<class 'sage.rings.polynomial.polynomial_ring.PolynomialRing_integral_domain_with_
↪category.element_class'>
<class 'sage.rings.polynomial.polynomial_integer_dense_ntl.Polynomial_integer_dense_
↪ntl'>
```

以上から，解決すべき問題は二系統あることが分る．ある二つの元が同じ Python クラス由来のインスタンスであるとすると，付随する __add__ メソッドによる加算が可能になっているはずだ．しかし，これら二つが数学的には非常に異なる環に属しているのならば，加算は不能にしておきたい．一方，数学的に同一の環に属している元に対しては，異なる実装に由来していてもそれらの元の加算は可能であるべきだろう．異なる Python クラスに由来する限り，これは簡単に実現できることではない．

これらの問題に対する解は「型強制」(coercion) と呼ばれており，以降で解説する．

しかし，まず肝心なのは，全ての元が自分の帰属先を知っていることだ．これを可能にするのが parent() メソッドである:

```
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)
```

2.10.2 ペアレントとカテゴリー

Python が代数構造の元に対応するクラス階層を備えているように，Sage もそれらの元を含む代数構造に対応するクラス群を提供している．Sage では元が属する構造物のことを「ペアレント構造」(parent structure) と呼び，基底となるクラスを持つ．そうしたクラス群は，おおむね数学的概念に沿った形で，集合，環，体などといった順の階層を形成している．

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

代数学では，同じ種類の代数構造を共有する物を，いわゆる「圏」(category) と呼ばれるものに集約して扱う．Sage のクラス階層と圏の階層構造にはそれなりに類似が見られないでもない．しかし，Python クラスについては圏との類似はあまり強調すべきものでもなさそうだ．いずれにせよ，数学的な意味における圏は Sage でも実装されている:

```
sage: Rings()
Category of rings
sage: ZZ.category()
Join of Category of Dedekind domains
    and Category of euclidean domains
    and Category of noetherian rings
    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
```

Sage におけるクラス階層は具体的実装に焦点が当てられている一方，Sage の圏フレームワークではより数学的な構造が重視されている．圏に対する個々の実装からは独立した，包括的なメソッドとテストを構成することが可能である．

Sage におけるペアレント構造は，Python オブジェクトとして唯一のものであると仮定されている．例えば，いったんある基底環上の多項式環が生成元と共に作成されると，その結果は実記憶上に配置される:

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

2.10.3 型とペアレント

`RingElement` 型は数学的概念としての「環の元」に完璧に対応しているわけではない．例えば，正方行列は一つの環に属していると見なしうるにもかかわらず， `RingElement` のインスタンスにはならない:

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

ペアレント が唯一のものであるとしても，同じ Sage のペアレントに由来する対等な **元** までが同一になるとは限らない．この辺りは Python の (全てではないにしても) 整数の振舞いとは違っている．

```
sage: int(1) is int(1) # Python の int 型
True
sage: int(-15) is int(-15)
False
sage: 1 is 1           # Sage の整数
False
```

重要なのは，異なる環に由来する元は，一般にその型ではなくペアレントによって判別されることである:

```
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
```

とういうわけで，代数学的な立場からすると **元のペアレントはその型より重要である** ことになる．

2.10.4 型変換と型強制

場合によっては，あるペアレント構造に由来する元を，異なるペアレント構造の元へ変換することができる．そうした変換は明示的に，あるいは暗黙的に行なうことが可能で，後者を **型強制** (coercion) と呼ぶ．

読者は，例えば C 言語における **型変換** (type conversion) と **型強制** (type coercion) の概念をご存知かもしれない．Sage にも **型変換** と **型強制** の考えは取り込まれている．しかし，Sage では主たる対象が型ではなくペアレントになっているので，C の型変換と Sage における変換を混同しないよう注意していただきたい．

以下の説明はかなり簡略化されているので，詳しい解説と実装情報については Sage レファレンスマニュアルの型強制に関する節と thematic tutorial を参照されたい．

異なる 環に属する元同士の演算実行については，両極をなす二つの立場がある:

- 異なる環はそれぞれが異なる世界を形作っており，何であれ異なる環由来の元同士で和や積を作ることは意味をなさない．`1` は整数であるのに `1/2` が有理数なのだから， `1 + 1/2` ですら意味をもちえない．

という立場もあるし

- 環 `R1` の元 `r1` が何とか他の環 `R2` の元と見なしうるなら， `r1` と `R2` の任意の元に対する全ての算術演算が許される．単位元は全ての体と多くの環に存在し，全て等価と見なしうる．

と考える立場もありうる．

Sage が宗とするのは歩み寄りだ．`P1` と `P2` がペアレント構造で `p1` が `P1` の元であるとき， `p1` が `P2` に帰属するとする解釈をユーザが明示的に求めることがあるかもしれない．この解釈があらゆる状況で有意であるとは限らないし， `P1` の全ての元に対して適用可能とも言えない．その解釈が意味を持つかどうかはユーザの判断にかかっているのである．我々はこうした解釈の要求を， **変換** (conversion) と呼ぶことにする:

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

しかし， **暗黙的** (自動的) 変換については，変換が **全面的** かつ **無矛盾** に行ないうる場合にのみ実行される．こちらで重視されているのは数学的な厳密さである．

そうした暗黙的変換は **型強制** (coercion) と呼ばれる．型強制が定義できるのならば，結果は型変換と一致しなければならない．型強制の定義に際して満足されるべき条件は二つある:

1. P1 から P2 への型強制は構造保存写像 (すなわち環準同形写像) になっていなければならない．P1 の要素が P2 に写像されるだけでは不十分で，その写像は P1 の代数構造を反映している必要がある．
2. 型強制は無矛盾に構成されなければならない．P3 を 3 つ目のペアレント構造として， P1 から P2 への型強制と P2 から P3 への型強制を合成すると， P1 から P3 への型強制に一致しなければならない．特に P1 から P2 へと P2 から P1 への型強制が存在する場合，この 2 つの変換を合成すると P1 への恒等写像にならねばならない．

したがって， GF(2) の全ての元は GF(5) 上へ変換可能であるにも関わらず，型強制は成立しない．GF(2) と GF(5) の間には環準同形写像が存在しないからである．

二つ目の条件 --- 無矛盾性 --- については，いくぶん説明が難しいところがある．多変数多項式環を例にとって説明してみたい．実用上，変数名を維持しない型強制はまず使いものにならないはずだ．であれば:

```
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
```

変数名を維持する環準同形写像が定義できなければ，型強制も成立しない．しかし，対象とする環の生成元を生成元リスト上の順序に応じて写像してやれば，型変換の方はまだ定義の可能性が残る:

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

ところが, そうした順序依存の変換は型強制としては満足すべきものにならない．ZZ['x','y'] から ZZ['y','x'] への変数名維持写像と ZZ['y','x'] から ZZ['a','b'] への順序依存写像を合成すると，結果は変数

名も順序も保存しない写像となって無矛盾性が破れてしまうからである．

型強制が成立するなら，異なる環に由来する元同士の比較や算術演算の際に利用されるはずである．これはたしかに便利なのだが，ペアレントの違いを越えた == 型関係の適用には無理が生じがちなことには注意を要する．== は **同一の** 環上の元同士の等価関係を表わすが，これは **異なる** 環の元が関わると必ずしも有効なわけではない．例えば， ZZ 上の 1 と，何か有限体上にあるとした 1 は等価であると見なすことができる．というのは，整数から任意の有限体へは型強制が成り立つからだ．しかし，一般には二つの異なる有限体環の間に型強制は成立しない．以下を見ていただきたい:

```
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
```

同様にして

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

さらに無矛盾性の条件から帰結するのは，厳密な環 (例えば有理数 QQ) から厳密ではない環 (例えば有限精度の実数 RR) への型強制は成立するが，逆方向は成立しないことである．QQ から RR への型強制と RR から QQ への変換を合成すると QQ 上の恒等写像になるはずだが，これは不可能である．と言うのは，有理数の中には，以下で示すように RR 上で問題なく扱えるものがあるからだ:

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

型強制が成立しない環 P1 と P2 の二つのペアレント由来の元を比較するとき，基準となるペアレント P3 が選択できて P1 と P2 を P3 へ型強制できる場合がある．そうした状況では型強制がうまく成立するはずだ．典型的な例は有理数と整数係数の多項式の和の計算で，結果は有理係数の多項式になる．

```
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
```

この結果は，原則的には ZZ['x'] の有理数体上でも成立する．しかし，Sage は最も自然に見える **正準** な共通のペアレントを選択しようとする (ここでは QQ['x'])．共通のペアレント候補が複数あってどれも同じく有望そうな場合，Sage は中の一つをランダムに選択するということは **しない**．これは再現性の高い結果を求めるためで，選択の手段については thematic tutorial に解説がある．

以下に示すのは，共通のペアレントへの型強制が成立しない例である:

```
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'
```

だめな理由は，Sage が有望そうな候補 QQ['x']['y']，QQ['y']['x']，QQ['x','y'] あるいは QQ['y','x'] のどれも選択できないことである．と言うのも，これら 4 つの相異なる構造はどれも共通なペアレントとして相応しく，基準となるべき選択肢にならないからだ．

2.11 有限群，アーベル群

Sage では，置換群，有限古典群 (例えば $SU(n,q)$)，有限行列群 (生成元を指定して生成)，そしてアーベル群 (無限次も可) などの演算が可能である．これらの機能の大半は，GAP とのインターフェイスを経由して実現されている．

まず，例として置換群を生成してみよう．それには，以下のようにして生成元のリストを指定してやればよい．

```
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()           # 結果は変化しがち
[Subgroup generated by [(3,4), (1,2,3)(4,5)] of (Permutation Group with generators␣
↪[(3,4), (1,2,3)(4,5)]),
 Subgroup generated by [...] of (Permutation Group with generators [(3,4), (1,2,3)(4,
↪5)])]
sage: G.center()
Subgroup generated by [()] of (Permutation Group with generators [(3,4), (1,2,3)(4,
↪5)])
sage: G.random_element()           # random 出力は変化する
(1,5,3)(2,4)
```

(次のページに続く)

(前のページからの続き)

```
sage: print(latex(G))
\langle (3,4), (1,2,3)(4,5) \rangle
```

Sage を使えば (LaTeX 形式で) 指標表を作ることもできる:

```
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 は有限体上の古典群と行列群も扱うことができる:

```
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()             # random 元をランダムに出力
[5 5 5 1]
[0 2 6 3]
[5 0 1 0]
[4 6 3 4]
sage: G.order()
276595200
```

(無限次および有限次の) アーベル群を使う演算も可能だ:

```
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
```

(次のページに続く)

(前のページからの続き)

```
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 数論

Sage は数論関連の多彩な機能を備えている．例えば，以下のようにして $\mathbf{Z}/N\mathbf{Z}$ 上の演算を実行することができる:

```
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 は数論では標準となっている関数群を装備している．例えば

```
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
```

という具合で，申し分なしだ．

Sage の sigma(n,k) 関数は， n の商の k 乗の和を計算する:

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

以下では，拡張ユークリッド互除法，オイラーの ϕ -関数，そして中国剰余定理を見てみよう:

```
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
```

次に， $3n+1$ 問題をちょっと調べてみる．

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

最後に，中国剰余定理を確かめてみよう．

```
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)]
```

(次のページに続く)

(前のページからの続き)

```
[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 p -進数

Sage には p -進数体も組込まれている．ただし，いったん生成された p -進数体については，後でその精度を変更することはできないことを注意しておく．

```
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)
```

p -進数体あるいは QQ 以外の数体上に整数環を実装するために多大の労力が投入されてきている．興味ある読者は Google グループ sage-support で専門家に詳細を聞いてみてほしい．

NumberField クラスには，すでに多くの関連メソッドが実装されている．

```
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()
Galois group 3T2 (S3) with order 6 of 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
```

(次のページに続く)

(前のページからの続き)

```
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 より進んだ数学

2.13.1 代数幾何

Sage では，任意の代数多様体を定義することができるが，その非自明な機能は $\mathbf{Q}$ 上の環あるいは有限体でしか使えない場合がある．例として，2 本のアフィン平面曲線の和を取り，ついで元の曲線を和の既約成分として分離してみよう．

```
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
]
```

以上の 2 本の曲線の交わりを取れば，全ての交点を求めてその既約成分を計算することもできる．

```
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,
```

(次のページに続く)

(前のページからの続き)

```
  2*y^2 + 4*y + 3
]
```

というわけで，点 $(1,0)$ および $(0,1)$ が双方の曲線上にあるのはすぐ見てとることができるし，y 成分が $2y^2+4y+3=0$ を満足する (2 次の) 点についても同じことだ．

Sage では，3 次元射影空間における捻れ 3 次曲線のトーリック・イデアルを計算することができる:

```
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],
 [-b*c + a*d, -c^2 + b*d, b^2 - a*c],
 [-c^3 + a*d^2, -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^3 - a*d^2],
 [-b*c + a*d, -b^2 + a*c, c^2 - b*d],
 [-b^3 + a^2*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^3 - a^2*d],
 [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 楕円曲線

Sage の楕円曲線部門には PARI の楕円曲線機能の大部分が取り込まれており，Cremona の管理するオンラインデータベースに接続することもできる (これにはデータベースパッケージを追加する必要がある)．さらに、Second-descent によって楕円曲線の完全 Mordell-Weil 群を計算する mwrank の機能が使えるし，SEA アルゴリズムの実行や同種写像全ての計算なども可能だ．$\mathbf{Q}$ 上の曲線群を扱うためのコードは大幅に更新され，Denis Simon による代数的降下法ソフトウェアも取り込まれている．

楕円曲線を生成するコマンド `EllipticCurve` には，さまざまな書法がある:

- EllipticCurve([a_1, a_2, a_3, a_4, a_6]): 楕円曲線

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

  を生成する．ただし a_i は a_1 のペアレントクラスに合わせて型強制される．全ての a_i がペアレント $\mathbf{Z}$ を持つ場合， a_i は $\mathbf{Q}$ に型強制される．

- EllipticCurve([a_4, a_6]): $a_1 = a_2 = a_3 = 0$ となる以外は上と同じ．

- EllipticCurve(ラベル): Cremona の (新しい) 分類ラベルを指定して，Cremona データベースに登録された楕円曲線を生成する．ラベルは `"11a"` や `"37b2"` といった文字列で，(以前のラベルと混同しないように) 小文字でなければならない．

- EllipticCurve(j): j -不変量 j を持つ楕円曲線を生成する.
- EllipticCurve(R,[a_1, a_2, a_3, a_4, a_6]): 最初と同じように a_i を指定して環 R 上の楕円曲線を生成する.

以上の各コンストラクタを実際に動かしてみよう:

```
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
```

点 $(0,0)$ は、 $y^2 + y = x^3 - x$ で定義される楕円曲線 E 上にある. Sage を使ってこの点を生成するには, `E([0,0])` と入力する. Sage は, そうした楕円曲線上に点を付け加えていくことができる (楕円曲線は, 無限遠点が零元、同一曲線上の 3 点を加えると 0 となる加法群としての構造を備えている):

```
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
```

複素数体上の楕円曲線は, j -不変量によって記述される. Sage では, j -不変量を以下のようにして計算する:

```
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
```

(次のページに続く)

(前のページからの続き)

```
sage: E.j_invariant()
110592/37
```

E と同じ j -不変量を指定して楕円曲線を作っても，それが E と同型になるとは限らない．次の例でも，2 つの曲線は導手 (conductor) が異なるため同型にならない．

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

しかし， F を 2 で捻ったツイスト (twist) は同型の曲線になる．

```
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
```

楕円曲線に随伴する L -級数，あるいはモジュラー形式 $\sum_{n=0}^{\infty} a_n q^n$ の係数 a_n を求めることもできる．計算には PARI の C-ライブラリを援用している:

```
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)
```

a_n を $n \leq 10^5$ の全てについて計算しても 1 秒ほどしかかからない:

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

楕円曲線を，対応する Cremona の分類ラベルを指定して生成する方法もある．そうすると，目的の楕円曲線がその階数，玉河数，単数基準 (regulator) などの情報と共にプレロードされる:

```
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
```

(次のページに続く)

(前のページからの続き)

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

Cremona のデータベースへ直接にアクセスすることも可能だ.

```
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]}
```

この方法でデータベースから引き出されるデータは，むろん EllipticCurve 型のオブジェクトにはならない．複数のフィールドから構成されたデータベースのレコードであるにすぎない．デフォルトで Sage に付属しているのは，導手が ≤ 10000 の楕円曲線の情報要約からなる，Cremona のデータベースの小型版である．オプションで大型版のデータベースも用意されていて，こちらは導手が 120000 までの全ての楕円曲線群の詳細情報を含む (2005 年 10 月時点)．さらに、Sage 用の大規模版データベースパッケージ (2GB) では，Stein-Watkins データベース上の数千万種の楕円曲線を利用することができる.

2.13.3 ディリクレ指標

ディリクレ指標とは，環 R に対する準同型写像 $(\mathbf{Z}/N\mathbf{Z})^* \to R^*$ を， $\gcd(N,x) > 1$ なる整数 x を 0 と置くことによって写像 $\mathbf{Z} \to R$ へ拡張したものである.

```
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
```

ディリクレ群を作成したので、次にその元を一つ取って演算に使ってみよう.

```
sage: G = DirichletGroup(21)
sage: chi = G.1; chi
Dirichlet character modulo 21 of conductor 7 mapping 8 |--> 1, 10 |--> zeta6
```

(次のページに続く)

(前のページからの続き)

```
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
```

この指標に対してガロワ群 $\mathrm{Gal}(\mathbf{Q}(\zeta_N)/\mathbf{Q})$ がどう振る舞うか計算したり，法 (modulus) の因数分解に相当する直積分解を実行することも可能だ.

```
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
]
```

次に，mod 20，ただし値が $\mathbf{Q}(i)$ 上に収まるディリクレ指標の群を作成する:

```
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
```

ついで， G の不変量をいくつか計算してみよう:

```
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)
```

(次のページに続く)

(前のページからの続き)

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

以下の例では、数体上でディリクレ指標を生成する．1 の累乗根については、 DirichletGroup の 3 番目の引数として明示的に指定している．

```
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]
```

ここで NumberField(x^4 + 1, 'a') と指定したのは，Sage に記号 a を使って K の内容 (a で生成される数体上の多項式 $x^4 + 1$) を表示させるためである．その時点で記号名 a はいったん未定義になるが、 a = K.0 (a = K.gen() としても同じ) が実行されると記号 a は多項式 $x^4 + 1$ の根を表すようになる．

2.13.4 モジュラー形式

Sage を使ってモジュラー空間の次元，モジュラー・シンボルの空間，Hecke 演算子、素因数分解などを含むモジュラー形式に関連した計算を実行することができる．

モジュラー形式が張る空間の次元を求める関数が数種類用意されている．例えば

```
sage: from sage.modular.dims import dimension_cusp_forms
sage: dimension_cusp_forms(Gamma0(11),2)
1
sage: dimension_cusp_forms(Gamma0(1),12)
1
sage: dimension_cusp_forms(Gamma1(389),2)
6112
```

次に、レベル 1 ，ウェイト 12 のモジュラー・シンボル空間上で Hecke 演算子を計算してみよう．

```
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
```

$\Gamma_0(N)$ と $\Gamma_1(N)$ の空間を生成することもできる.

```
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 over Rational Field
```

特性多項式と q -展開を計算してみよう.

```
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)
]
```

モジュラー・シンボルの空間を，指標を指定して生成することも可能だ.

```
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)
]
```

以下の例では，モジュラー形式によって張られる空間に対する Hecke 演算子の作用を，Sage でどうやって計算するかを示す.

```
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()
```

(次のページに続く)

(前のページからの続き)

```
((1,0), (1,8), (1,9))
sage: M.sign()
0
```

T_p は通常の Hecke 演算子 (p は素数) を表す．Hecke 演算子 T_2 ， T_3 ， T_5 はモジュラー・シンボル空間にどんな作用を及ぼすのだろうか？

```
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]
```

第3章　対話型シェル

このチュートリアルの大部分は，読者が sage コマンドによって Sage インタプリタを起動しているものと前提している．コマンド sage は改造版 IPython シェルを起動し，大量の関数やクラス群をインポートしてコマンドプロンプトから利用可能にする．$SAGE_ROOT/ipythonrc ファイルを編集すれば，さらなるシェル環境のカスタマイズも可能だ．Sage を起動すると，すぐに次のような画面が現れる:

```
┌──────────────────────────────────────────────────────────────────────────────────────────────────────┐
│ SageMath version 9.7, Release Date: 2022-01-10                                                       │
│ Using Python 3.10.4. Type "help()" for help.                                                         │
└──────────────────────────────────────────────────────────────────────────────────────────────────────┘


sage:
```

Sage を終了するには，Ctrl-D と押すか， コマンド quit あるいは exit を入力する．

```
sage: quit
Exiting Sage (CPU time 0m0.00s, Wall time 0m0.89s)
```

"Wall time"は，CPU タイムではなく外界の実経過時間を示している．CPU タイムは GAP や Singular などのサブプロセスの消費時間までは勘定に入れてくれないから，実経過時間も計算時間の見積りに必要だ．

(ターミナルから kill -9 を入力して Sage プロセスを停止するのは止めたほうがいい．kill -9 では Maple などの子プロセスが停止しなかったり， $HOME/.sage/tmp 内の一時ファイルが消去されずに終わるなどの恐れがある．)

3.1 Sage セッション

セッション とは，Sage の起動から終了までの間に行なわれた一連の入出力の総体のことをいう．Sage は，Sage に対する入力の全てを IPython 経由で記録している．事実，(ノートブック経由ではなく) 対話型シェルを使って Sage を動かしているのならば，好きな時に %history (または %hist) と入力して，それまでの全入力履歴を見ることができる．IPython についてもっと知りたければ、Sage プロンプトで ? と入力すれば，"IPython offers numbered prompts ... with input and output caching. All input is saved and can be retrieved as variables (besides the usual arrow key recall). The following GLOBAL variables always exist (so don't overwrite them!)" などと詳しい情報を表示させることができる:

```
  _:  前回の入力を呼び出す (対話型シェルとノートブックの両方で通用する)
 __: 前々回の入力を呼び出す (対話型シェルのみで通用)
_oh : 全ての入力をリストする (対話型シェルのみで通用)
```

ここで例を見てみよう:

```
sage: factor(100)
 _1 = 2^2 * 5^2
sage: kronecker_symbol(3,5)
 _2 = -1
sage: %hist   # これが使えるのは対話型シェル上のみ. ノートブックではだめ.
1: factor(100)
2: kronecker_symbol(3,5)
3: %hist
sage: _oh
 _4 = {1: 2^2 * 5^2, 2: -1}
sage: _i1
 _5 = 'factor(ZZ(100))\n'
sage: eval(_i1)
 _6 = 2^2 * 5^2
sage: %hist
1: factor(100)
2: kronecker_symbol(3,5)
3: %hist
4: _oh
5: _i1
6: eval(_i1)
7: %hist
```

以降, このチュートリアル, それに他の Sage ドキュメンテーションでも出力番号を省略する.

セッション中は, 一連の入力をマクロとして保存しておいて再利用することもできる.

```
sage: E = EllipticCurve([1,2,3,4,5])
sage: M = ModularSymbols(37)
sage: %hist
1: E = EllipticCurve([1,2,3,4,5])
2: M = ModularSymbols(37)
3: %hist
sage: %macro em 1-2
Macro `em` created. To execute, type its name (without quotes).
```

```
sage: E
Elliptic Curve defined by y^2 + x*y + 3*y = x^3 + 2*x^2 + 4*x + 5 over
Rational Field
sage: E = 5
```

(次のページに続く)

(前のページからの続き)

```
sage: M = None
sage: em
Executing Macro...
sage: E
Elliptic Curve defined by y^2 + x*y + 3*y = x^3 + 2*x^2 + 4*x + 5 over
Rational Field
```

対話型シェルを使っている間も，感嘆符！を前置すれば好きな UNIX シェルコマンドを実行することができる．例えば

```
sage: !ls
auto  example.sage glossary.tex  t  tmp  tut.log  tut.tex
```

のように，カレントディレクトリの内容を表示することができる．

シェル変数 PATH の先頭には Sage の bin ディレクトリが配置されているから， gp ， gap ， singular ， maxima などを実行すると，Sage に付属しているプログラムのバージョンを確認することができる．

```
sage: !gp
Reading GPRC: /etc/gprc ...Done.

                         GP/PARI CALCULATOR Version 2.2.11 (alpha)
                  i686 running linux (ix86/GMP-4.1.4 kernel) 32-bit version
...
sage: !singular
                     SINGULAR                                /  Development
 A Computer Algebra System for Polynomial Computations   /   version 3-0-1
                                                       0<
     by: G.-M. Greuel, G. Pfister, H. Schoenemann        \   October 2005
FB Mathematik der Universitaet, D-67653 Kaiserslautern    \
```

3.2 入出力のログをとる

Sage セッションのロギングと，セッションの保存 (セッション全体の保存と読み込み 節を参照) は同じことではない．入力のログをとるには， logstart コマンドを使う (オプションで出力のログも可能だ)．詳細については logstart? と入力してみてほしい．logstart を使えば，全ての入力と出力のログを残し，将来のセッション時に (そのログファイルをリロードしてやるだけで) 入力を再生することも可能になる．

```
was@form:~$ sage
 ┌────────────────────────────────────────────────────────────────────────────────────
─────────────────────────────────────┐
 │ SageMath version 9.7, Release Date: 2022-01-10                                     │
 │ Using Python 3.10.4. Type "help()" for help.                                       │
 └────────────────────────────────────────────────────────────────────────────────────
```

(次のページに続く)

(前のページからの続き)

```
──────────────────────────────────────────┘


sage: logstart setup
Activating auto-logging. Current session state plus future input saved.
Filename       : setup
Mode           : backup
Output logging : False
Timestamping   : False
State          : active
sage: E = EllipticCurve([1,2,3,4,5]).minimal_model()
sage: F = QQ^3
sage: x,y = QQ['x,y'].gens()
sage: G = E.gens()
sage:
Exiting Sage (CPU time 0m0.61s, Wall time 0m50.39s).
was@form:~$ sage
 ┌────────────────────────────────────────────────────────────────────────────────────
──────────────────────────────────────────┐
 │ SageMath version 9.7, Release Date: 2022-01-10                                     │
 │ Using Python 3.10.4. Type "help()" for help.                                       │
 └────────────────────────────────────────────────────────────────────────────────────
──────────────────────────────────────────┘


sage: load("setup")
Loading log file <setup> one line at a time...
Finished replaying log file <setup>
sage: E
Elliptic Curve defined by y^2 + x*y  = x^3 - x^2 + 4*x + 3 over Rational
Field
sage: x*y
x*y
sage: G
[(2 : 3 : 1)]
```

Sage を Linux KDE ターミナル konsole 上で使っているなら，以下の手順でセッションを保存することもできる．まず konsole 上で Sage を起動したら、 "settings"(日本語環境であれば『設定』) を選択し，次に "history"(『履歴』), "set unlimited"(『無制限にする』) の順に選択しておく．セッションを保存したくなった時点で， "edit"(『編集』) の中の "save history as..."(『履歴を名前を付けて保存』) を選択してセッションを保存するファイル名を入力してやればよい．いったんファイルとして保存してしまえば，好きなように xemacs などのエディタで読み込んだりプリントアウトしたりすることができる．

3.3 プロンプト記号はペースト時に無視される

Sage セッションあるいは Python の演算結果を読み込んで，Sage 上にコピーしたい場合がある．厄介なのは、そうした出力に >>> や sage: といったプロンプト記号が紛れ込んでいることだ．しかし実際には，プロンプト記号を含む実行例を Sage 上へ好きにコピー・ペーストしてやることができる．デフォルトで Sage パーサーはデータを Python に送る前に行頭の >>> や sage: プロンプト記号を除去してくれるからだ．例えば

```
sage: 2^10
1024
sage: sage: sage: 2^10
1024
sage: >>> 2^10
1024
```

3.4 計時コマンド

入力行の先頭に %time コマンドを入れておくと，出力までに要した時間を表示することができる．例として，べき乗計算を異なった方法で行なった場合の実行時間を比較してみよう．以下に示した実行時間の値は，動かしているコンピュータ本体や Sage のバージョンによって大きく異なる可能性が高い．まず、Python を直に動かしてみると:

```
sage: %time a = int(1938)^int(99484)
CPU times: user 0.66 s, sys: 0.00 s, total: 0.66 s
Wall time: 0.66
```

上の出力は，実行に計 0.66 秒かかり， "Wall time" つまりユーザーの実待ち時間もやはり 0.66 秒だったことを示している．コンピュータに他のプログラムから大きな負荷がかかっている場合， "Wall time"が CPU タイムよりかなり長くなることがある．

次に，同じべき乗計算を Sage 組み込みの Integer 型を使って実行した場合の時間を計ってみよう．Sage の Integer 型は，Cython 経由で GMP ライブラリを使って実装されている:

```
sage: %time a = 1938^99484
CPU times: user 0.04 s, sys: 0.00 s, total: 0.04 s
Wall time: 0.04
```

PARI の C-ライブラリを経由すると

```
sage: %time a = pari(1938)^pari(99484)
CPU times: user 0.05 s, sys: 0.00 s, total: 0.05 s
Wall time: 0.05
```

GMP の方が速いが，その差はわずかだ (Sage 用にビルドされた PARI は整数演算に GMP を使っているのだから，納得できる結果である).

次の例のように， cputime コマンドを使えば，一連のコマンドからなるコードブロックの実行時間を計ることもできる:

```
sage: t = cputime()
sage: a = int(1938)^int(99484)
sage: b = 1938^99484
sage: c = pari(1938)^pari(99484)
sage: cputime(t)                        # random 値には若干の幅がある.
0.64
```

```
sage: cputime?
 ...
    Return the time in CPU second since Sage started, or with optional
    argument t, return the time since time t.
    INPUT:
        t -- (optional) float, time in CPU seconds
    OUTPUT:
        float -- time in CPU seconds
```

walltime コマンドの動作は，計測するのが実経過時間である点以外は cputime コマンドと変わらない.

上で求めたべき乗を，Sage に取り込まれている各コンピュータ代数システムを使って計算することもできる．計算を実行するには，使いたいシステムの名前をコマンド名としてそのプログラムのサーバを呼び出す．いちばん肝心な計測値は，実経過時間 (wall time) だ．しかし，実経過時間と CPU タイムの値が大幅に食い違う場合は，解決すべきパフォーマンス上の問題点の存在を示している可能性がある.

```
sage: time 1938^99484;
CPU times: user 0.01 s, sys: 0.00 s, total: 0.01 s
Wall time: 0.01
sage: gp(0)
0
sage: time g = gp('1938^99484')
CPU times: user 0.00 s, sys: 0.00 s, total: 0.00 s
Wall time: 0.04
sage: maxima(0)
0
sage: time g = maxima('1938^99484')
CPU times: user 0.00 s, sys: 0.00 s, total: 0.00 s
Wall time: 0.30
sage: kash(0)
0
sage: time g = kash('1938^99484')
CPU times: user 0.00 s, sys: 0.00 s, total: 0.00 s
Wall time: 0.04
sage: mathematica(0)
        0
sage: time g = mathematica('1938^99484')
```

(次のページに続く)

(前のページからの続き)

```
CPU times: user 0.00 s, sys: 0.00 s, total: 0.00 s
Wall time: 0.03
sage: maple(0)
0
sage: time g = maple('1938^99484')
CPU times: user 0.00 s, sys: 0.00 s, total: 0.00 s
Wall time: 0.11
sage: libgap(0)
0
sage: time g = libgap.eval('1938^99484;')
CPU times: user 0.00 s, sys: 0.00 s, total: 0.00 s
Wall time: 1.02
```

以上のテスト計算で最も遅かったのは, GAP と Maxima である (実行結果はホスト sage.math.washington.edu 上のもの). 各システムとの pexpect インターフェイスにかかる負荷を考えると，上の一連の計測値を最速だった Sage の値と比較するのは公平を欠く面があるかもしれない.

3.5 IPython トリック

すでに述べたように，Sage はそのフロントエンドとして IPython を援用しており，ユーザは IPython のコマンドと独自機能を自由に利用することができる. その全貌については，ご自分で full IPython documentation を読んみてほしい. そのかわり，ここでは IPython の「マジックコマンド」と呼ばれる，お便利なトリックをいくつか紹介させていただこう:

- %edit (%ed や ed でもいい) を使ってエディタを起動すれば，複雑なコードの入力が楽になる. Sage の使用前に, 環境変数 EDITOR に好みのエディタ名を設定しておこう (export EDITOR=/usr/bin/emacs または export EDITOR=/usr/bin/vim とするか， .profile ファイルなどで同様の設定をする). すると Sage プロンプトで %edit を実行すれば設定したエディタが起動する. そのエディタで関数

  ```
  def some_function(n):
      return n**2 + 3*n + 2
  ```

  を定義し，保存したらエディタを終了する. 以降，このセッション中は some_function を利用できるようになる. 内容を編集したければ Sage プロンプトで %edit some_function と入力すればよい.

- 結果出力を他の用途のために編集したければ， %rep を実行する. すると直前に実行したコマンドの出力が編集できるように Sage プロンプト上に配置される. :

  ```
  sage: f(x) = cos(x)
  sage: f(x).derivative(x)
  -sin(x)
  ```

  この段階で Sage プロンプトから %rep を実行すると，新しい Sage プロンプトに続いて -sin(x) が現われる. カーソルは同じ行末にある.

IPython のクイック レファレンスガイドを見たければ， %quickref と入力する．執筆時点 (2011 年 4 月) では Sage は IPython のバージョン 0.9.1 を採用しており， documentation for its magic commands はオンラインで読むことができる．マジックコマンドの，ちょっと進んだ機能群については IPython の ここで文書化されているのが見つかるはずだ．

3.6 エラーと例外処理

処理中に何かまずいことが起きると，Python はふつう『例外』(exception) を発生し，その例外を引き起こした原因を教えてくれることもある．よくお目にかかることになるのは， NameError や ValueError といった名称の例外だ (Python ライブラリーリファレンス [PyLR] に例外名の包括的なリストがある)．実例を見てみよう:

```
sage: EllipticCurve([0,infinity])
Traceback (most recent call last):
...
SignError: cannot multiply infinity by zero
```

何が悪いか調べるには対話型デバッガが役立つこともある．デバッガを使うには， %pdb コマンドによって作動のオン/オフをトグルする (デフォルトはオフ)．作動後は、例外が発生するとデバッガが起動し，プロンプト ipdb> が表示される．このデバッガの中から，任意のローカル変数の状態を表示したり，実行スタックを上下して様子を調べることができる．

```
sage: %pdb
Automatic pdb calling has been turned ON
sage: EllipticCurve([1,infinity])
---------------------------------------------------------------------------
<class 'exceptions.TypeError'>             Traceback (most recent call last)
...

ipdb>
```

デバッガから実行できるコマンドの一覧を見るには， ipdb> プロンプト上で ? を入力する:

```
ipdb> ?

Documented commands (type help <topic>):
========================================
EOF    break  commands   debug    h       l     pdef   quit    tbreak
a      bt     condition  disable  help    list  pdoc   r       u
alias  c      cont       down     ignore  n     pinfo  return  unalias
args   cl     continue   enable   j       next  pp     s       up
b      clear  d          exit     jump    p     q      step    w
whatis where

Miscellaneous help topics:
```

(次のページに続く)

(前のページからの続き)

```
=========================
exec  pdb

Undocumented commands:
======================
retval  rv
```

Sage に戻るには，Ctrl-D か quit を入力する．

3.7 コマンド入力の遡行検索とタブ補完

遡行検索: コマンドの冒頭部を打ち込んでから Ctrl-p (または上向き矢印キー) を押すと，冒頭部が一致する過去の入力行を全て呼び出すことができる．この機能は，Sage をいったん終了し再起動してからでも有効である．Ctrl-r を入力すれば，入力ヒストリを逆方向に検索することも可能だ．この入力行の検索と再利用機能は全て readline パッケージを経由しており，ほとんどの Linux 系システム上で利用できるはずだ．

タブ補完機能を体験するため，まず 3 次元ベクトル空間 $V = \mathbf{Q}^3$ を生成しておく:

```
sage: V = VectorSpace(QQ,3)
sage: V
Vector space of dimension 3 over Rational Field
```

次のような，もっと簡潔な記号法を使ってもよい:

```
sage: V = QQ^3
```

タブ補完を使えば，簡単に V の全メンバ関数を一覧表示することができる．V. と入力し，ついで [tab] キーを押すだけだ:

```
sage: V.[tab key]
V._VectorSpace_generic__base_field
...
V.ambient_space
V.base_field
V.base_ring
V.basis
V.coordinates
...
V.zero_vector
```

関数名の出だし何文字かを打ってから [tab キー] を押せば，入力した文字で始まる名前の関数だけに候補を絞ることができる．

```
sage: V.i[tab key]
V.is_ambient  V.is_dense    V.is_full     V.is_sparse
```

特定の関数について調べたい場合もある．coordinates 関数を例にとると，そのヘルプを表示するには V.coordinates? と入力すればいいし，ソースコードを見るには V.coordinates?? を入力すればいい．詳細については次の節で解説する．

3.8 統合ヘルプシステム

Sage の特長の一つは，総合的なヘルプ機能の装備である．関数名に続けて?を入力すると、その関数のドキュメントを表示することができる．

```
sage: V = QQ^3
sage: V.coordinates?
Type:           instancemethod
Base Class:     <class 'instancemethod'>
String Form:    <bound method FreeModule_ambient_field.coordinates of Vector
space of dimension 3 over Rational Field>
Namespace:      Interactive
File:           /home/was/s/local/lib/python2.4/site-packages/sage/modules/f
ree_module.py
Definition:     V.coordinates(self, v)
Docstring:
    Write v in terms of the basis for self.

    Returns a list c such that if B is the basis for self, then

            sum c_i B_i = v.

    If v is not in self, raises an ArithmeticError exception.

    EXAMPLES:
        sage: M = FreeModule(IntegerRing(), 2); M0,M1=M.gens()
        sage: W = M.submodule([M0 + M1, M0 - 2*M1])
        sage: W.coordinates(2*M0-M1)
        [2, -1]
```

上で見たように，ヘルプ表示には，そのオブジェクトの型，定義されているファイル，現セッションにペーストすることができる使用例付きの解説が含まれる．使用例のほとんどは常に自動的なテストが行なわれていて，仕様どおりの正確な動作が確認されている．

もう一つの機能は，Sage のオープンソース精神をよく表すものだ．f が Python で書かれた関数であれば f?? と入力すると f を定義しているソースを表示することができるのだ．例えば

```
sage: V = QQ^3
sage: V.coordinates??
Type:           instancemethod
...
Source:
def coordinates(self, v):
        """
        Write $v$ in terms of the basis for self.
        ...
        """
        return self.coordinate_vector(v).list()
```

これを見ると，coordinates 関数は coordinate_vector 関数を呼び出して結果をリストに変換しているだけであることが判る．では coordinate_vector 関数が何をしているかと言うと:

```
sage: V = QQ^3
sage: V.coordinate_vector??
...
def coordinate_vector(self, v):
        ...
        return self.ambient_vector_space()(v)
```

coordinate_vector 関数は，入力を生成空間 (ambient space) に合わせて型変換するから，これは v の係数ベクトルが空間 V ではどう変換されるか計算していることと同じである．V は $\mathbf{Q}^3$ そのものだから，すでに同じ構造になっている．部分空間用に，上とは異なる coordinate_vector 関数も用意されている．部分空間を作って，どんな関数か見てみることにしよう:

```
sage: V = QQ^3; W = V.span_of_basis([V.0, V.1])
sage: W.coordinate_vector??
...
def coordinate_vector(self, v):
        """
         ...
        """
        # First find the coordinates of v wrt echelon basis.
        w = self.echelon_coordinate_vector(v)
        # Next use transformation matrix from echelon basis to
        # user basis.
        T = self.echelon_to_user_matrix()
        return T.linear_combination_of_rows(w)
```

(こうした実装の仕方は無駄が多いと思われる方は，どうか我々に連絡して線形代数周りの最適化に力を貸していただきたい．)

help(コマンド名) あるいは help(クラス名) と入力すれば，知りたいクラスの man ページ型ヘルプファイルを表示することもできる．

```
sage: help(VectorSpace)
Help on function VectorSpace in module sage.modules.free_module:

VectorSpace(K, dimension_or_basis_keys=None, sparse=False, inner_product_matrix=None,␣
↪*,
            with_basis='standard', dimension=None, basis_keys=None, **args)
EXAMPLES:

The base can be complicated, as long as it is a field.

::

    sage: V = VectorSpace(FractionField(PolynomialRing(ZZ,'x')),3)
    sage: V
    Vector space of dimension 3 over Fraction Field of Univariate Polynomial Ring in x
     over Integer Ring
    sage: V.basis()
    [
    (1, 0, 0),
    (0, 1, 0),
--More--
```

q と入力してヘルプを終えると，中断前のセッション画面がそのまま復帰する．セッションに干渉することがある function_name? と違って，ヘルプ表示はセッションの邪魔をしない．とりわけ便利なのは help(モジュール名) と入力することだ．例えばベクトル空間は sage.modules.free_module で定義されているから，そのモジュール全体に関するドキュメントを見たければ help(sage.modules.free_module) と実行すればよい．ヘルプを使ってドキュメントを閲覧している間は，/ と打てば語句検索 ができるし，? と打てば逆方向に検索することができる．

3.9 オブジェクトの保存と読み込み

行列や，あるいはもっと手間のかかる複雑なモジュラーシンボルの空間を扱っていて，後で利用するため結果を保存しておきたくなったとしよう．そんな場合にはどうすればよいだろうか．オブジェクトを保存するために各コンピュータ代数システムが提供している方法は，以下の通りである．

1. **セッションの保存:** セッション全体の保存と読み込みのみ可能 (GAP，Magma など).
2. **統合入出力:** 全オブジェクトの印字が再利用可能な形式で行なわれる (GAP と PARI).
3. **再実行**: インタープリタによるプログラムの再実行が容易にしてある (Singular，PARI).

Python で動く Sage では，全てのオブジェクトのシリアル化 (直列化) という，他とは異なる方法が採用されている．つまりオブジェクトを，その原型を再現可能な形式で文字列に変換するのだ．これは PARI の統合入出力の考え方に近いが，オブジェクトを複雑な印字形式で画面出力してやる必要がないのが利点だ．さらに保存と読み込みは (ほとんどの場合) 完全に自動化されているから，新たにプログラムを書く必要もない．そうした機能は Python に最初から組込まれているものだからである．

ほぼ全ての Sage オブジェクト x は， コマンド save(x, ファイル名) (あるいは多くの場合 x.save(ファイル名)) を使えば圧縮形式でディスクに保存することができるようになっている．保存したオブジェクトを読み戻すには， load(ファイル名) を実行する．

```
sage: A = MatrixSpace(QQ,3)(range(9))^2
sage: A
[ 15  18  21]
[ 42  54  66]
[ 69  90 111]
sage: save(A, 'A')
```

ここでいったん Sage を終了してみよう．再起動後に A を読み込むには:

```
sage: A = load('A')
sage: A
[ 15  18  21]
[ 42  54  66]
[ 69  90 111]
```

楕円曲線のようなもっと複雑なオブジェクトに対しても，以上と同じやり方が通用する．メモリ上に配置されていたオブジェクト関連の全データは，そのオブジェクトと共に保存される．例えば

```
sage: E = EllipticCurve('11a')
sage: v = E.anlist(100000)                # ちょっと時間がかかる
sage: save(E, 'E')
sage: quit
```

こうして保存された E は，オブジェクト本体と一緒に a_n の冒頭 100000 個も保存するため，153K バイトの大きさになる．

```
~/tmp$ ls -l E.sobj
-rw-r--r--  1 was was 153500 2006-01-28 19:23 E.sobj
~/tmp$ sage [...]
sage: E = load('E')
sage: v = E.anlist(100000)                # すぐ終了
```

(Python 経由の保存と読み込みには， cPickle モジュールが使われている．実際，Sage オブジェクト x の保存は cPickle.dumps(x, 2) を実行して行なうことができる．引数 2 に注目．)

Sage で保存・読み込みできないのは，GAP， Singular， Maxima など外部コンピュータ代数システムで作成されたオブジェクトである． これらは読み込むことができても "invalid"(利用不能) な状態にあると認識される．GAP では，相当数のオブジェクトが再構成に使える印字形式を持つ一方，再構成できな場合も多いため印字形式からのオブジェクトの再構成は意図的に禁止されている．

```
sage: a = libgap(2)
sage: a.save('a')
sage: load('a')
Traceback (most recent call last):
```

(次のページに続く)

(前のページからの続き)

```
...
ValueError: The session in which this object was defined is no longer
running.
```

GP/PARI オブジェクトは，印字形式から十分に再構成可能なため，保存と読み込みも可能になっている．

```
sage: a = gp(2)
sage: a.save('a')
sage: load('a')
2
```

保存したオブジェクトは，異なるアーキテクチャ上の，異なるオペレーティングシステムで動く Sage へもロードすることができる．例えば，32 ビット OSX 上で保存した大規模行列を 64 ビット版 Linux の Sage へ読み込み，その階段形式を求めてから元の OS X 上へ戻すといったことも可能だ．さらに，オブジェクトの保存までに使ったのとは違うバージョンの Sage でオブジェクトを読み込むこともできる場合が多い．ただし，これは読み書きしたいオブジェクトに関わるコードがバージョン間で大きくは異ならないことが条件となる．オブジェクトの保存に際しては，その属性の全てがオブジェクトを定義している (ソースコードではなく) クラスと共に保存される．そのクラスが新バージョンの Sage に存在しない場合，配下のオブジェクトを新バージョンでは読み込むことはできない．しかし古いバージョンで読み込むことはできるはずだから，(`x.__dict__` で) オブジェクト `x` のディクショナリを生成して保存しておけば，それを新しいバージョンで読み込むことができることもある．

3.9.1 テキスト形式で保存する

オブジェクトを ASCII テキスト形式で保存しておくこともできる．手順は，ファイルを書込みモードで開いて，そこに保存すべきオブジェクトの文字列表現を書き込むだけのことだ (このやり方で複数個のオブジェクトを保存することができる)．オブジェクトの書込みを終えたら，ファイルをクローズすればよい．

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: f = (x+y)^7
sage: o = open('file.txt','w')
sage: o.write(str(f))
sage: o.close()
```

3.10 セッション全体の保存と読み込み

Sage は，セッション全体を保存し再ロードするための非常に柔軟な機能を備えている．

コマンド `save_session(セッション名)` は，現セッション中に定義された全ての変数を、コマンドで指定した セッション名 にディクショナリとして保存する．(保存を想定していない変数がある場合もまれに見られるが，そうした時はディクショナリに保存されずに終るだけだ．) 保存先は `.sobj` ファイルとなり，他の保存済みオブジェクトと全く同じように読み込むことができる．セッション中に保存したオブジェクトを再びロードすると，変数名をキー，オブジェクトを値とするディクショナリが生成されることになる．

実行中のセッションに セッション名 に定義された変数をロードするには，load_session(セッション名) コマンドを使う．このコマンドは現セッションとロードされる側のセッション内容を合併するのであって，現セッションで定義した変数が消去されるわけではない．

まず Sage を起動し，変数をいくつか定義しておく．

```
sage: E = EllipticCurve('11a')
sage: M = ModularSymbols(37)
sage: a = 389
sage: t = M.T(2003).matrix(); t.charpoly().factor()
 _4 = (x - 2004) * (x - 12)^2 * (x + 54)^2
```

次にこのセッションをファイルに保存し，先に定義した変数を残しておく．.sobj ファイルを確認すると，その大きさは 3K バイトほどとなっている．

```
sage: save_session('misc')
Saving a
Saving M
Saving t
Saving E
sage: quit
was@form:~/tmp$ ls -l misc.sobj
-rw-r--r--  1 was was 2979 2006-01-28 19:47 misc.sobj
```

仕上げに Sage を再起動し，変数をいくつか追加定義してから，先に保存したセッションを読み込んでみよう．

```
sage: b = 19
sage: load_session('misc')
Loading a
Loading M
Loading E
Loading t
```

保存しておいた変数が再び利用可能になる一方，上で追加した変数 b は上書きされていないことが分る．

```
sage: M
Full Modular Symbols space for Gamma_0(37) of weight 2 with sign 0
and dimension 5 over Rational Field
sage: E
Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20 over Rational
Field
sage: b
19
sage: a
389
```

第4章　インターフェイスについて

Sage の最大の特長は，多種多様なコンピュータ代数システムを，共通インターフェイスと本格的なプログラミング言語 Python を用いて一つ屋根の下にまとめ上げている点だ．これにより Sage では由来の異なるオブジェクト群を組み合わせた演算処理が可能になっている．

ある 1 つのインターフェイスに対し， `console` メソッドと `interact` メソッドは，全く違った機能を果たしている．GAP を例にとって具体的に見てみよう:

1. `gap.console()`: このメソッドは GAP のコンソールを起動し，命令実行の主体を GAP 上へ移す．Sage の役割は，Linux の bash シェルのような便利なプログラムランチャとしての役割に限定される．
2. `gap.interact()`: このメソッドは，Sage オブジェクト群を使って動作している GAP インスタンスと情報交換するための便利な径路を提供する．これを使うと，GAP セッション中に (対話型インタフェイスからでも)Sage オブジェクトをインポートすることができる．

4.1 GP/PARI

数論関係の演算処理を主目的とする PARI は，コンパクトで非常に練れたプログラムで，高度に最適化されている．Sage から PARI を使うには，大きく異なる 2 種類のインターフェイスを選ぶことができる:

- `gp` -- "**G** o **P** ARI" インタープリタ
- `pari` -- PARI C ライブラリ

以下の例では，同一の計算を二通りのやり方で実行している．一見同じように見えても実は出力内容は同一ではないし，画面の奥で実際に行なわれている処理過程は二つの方法で全くと言っていいほど異なっているのだ．

```
sage: gp('znprimroot(10007)')
Mod(5, 10007)
sage: pari('znprimroot(10007)')
Mod(5, 10007)
```

第一のやり方では，GP インタープリタが独立したサーバプロセスとして起動され，そのプロセスに文字列 `'znprimroot(10007)'` が送られる．GP は送られて来た文字列を評価し，結果を変数に格納する (変数は子 GP プロセス配下の，開放されないメモリ空間内に確保される)．その変数の値が画面表示されて仕上がりになる．第二のやり方では、独立したプロセスが起動されることはなく，文字列 `'znprimroot(10007)'` は PARI C ライブラリ関数群によって処理される．処理結果は Python の確保しているヒープ上に配置され，その変数が参照されなくなると使っていたヒープメモリは開放される．二つのやり方では、生成されるオブジェクトの型からして違っている:

```
sage: type(gp('znprimroot(10007)'))
<class 'sage.interfaces.gp.GpElement'>
sage: type(pari('znprimroot(10007)'))
<class 'cypari2.gen.Gen'>
```

では，どちらの方法を選ぶべきだろうか？答は目的による、としか言えない．GP インターフェイスは GP/PARI プログラムをそのまま動作させるのだから，GP/PARI のコマンド入力行から可能なことは全て出来る．複雑な PARI プログラムを読み込んで走らせたい場合などには向いているだろう．これと比較すると，(C ライブラリを経由する)PARI インターフェイスはかなり制限がきつい．まだライブラリのメンバ関数の全てが実装されているわけではないし，数値積分などを含むコードの多くが PARI インターフェイスからは使えない状態だ．一方，PARI インターフェイスは GP インターフェイスより大幅に高速で頑強でもある．

(入力行を評価中にメモリ不足に陥った場合，GP インターフェイスは特に警告することなく自動的にスタックサイズを 2 倍して評価を再試行する．必要とされるメモリ量を正しく見積っていなかったとしても処理が頓挫することはまずない．この有難い仕掛けは，標準の GP インタープリタには備えられていないようだ．PARI C ライブラリ インターフェイスについて言うと，こちらは生成したオジェクトを直ちにコピーして PARI スタックから送り出してしまうので，スタックが積み上がることはない．しかし，どんなオブジェクトも大きさが 100MB を越えてはならず，越えて生成された場合はスタックオーバーフローが起きる．このオブジェクトのコピーが，わずかながら実行効率上の不利を招いているのも確かである．)

まとめると，Sage は PARI C ライブラリを利用して GP/PARI インタープリタと同様の機能を提供しているが，優秀なメモリ管理とプログラミング言語 Python の援用という利点がある，ということになる．

ここで，Python のリストから PARI のリストを作ってみよう．

```
sage: v = pari([1,2,3,4,5])
sage: v
[1, 2, 3, 4, 5]
sage: type(v)
<class 'cypari2.gen.Gen'>
```

PARI のオブジェクトは全て `Gen` 型になる．各オブジェクトに埋め込まれている PARI 由来のデータ型を取得するには，メンバ関数 `type` を使う．

```
sage: v.type()
't_VEC'
```

PARI で楕円曲線を生成するには，`ellinit([1,2,3,4,5])` と入力する．Sage の方法も似ているが，`ellinit` を先の `t_VEC` v のような任意の PARI オブジェクトに対して呼び出すことができる点が違っている．

```
sage: e = v.ellinit()
sage: e.type()
't_VEC'
sage: pari(e)[:13]
[1, 2, 3, 4, 5, 9, 11, 29, 35, -183, -3429, -10351, 6128487/10351]
```

楕円曲線オブジェクトが生成できたので，これを使った計算を試みよう．

```
sage: e.elltors()
[1, [], []]
sage: e.ellglobalred()
[10351, [1, -1, 0, -1], 1, [11, 1; 941, 1], [[1, 5, 0, 1], [1, 5, 0, 1]]]
sage: f = e.ellchangecurve([1,-1,0,-1])
sage: f[:5]
[1, -1, 0, 4, 3]
```

4.2 GAP

Sage には GAP 4.4.10 が付属しており，群論を始めとする計算離散数学に対応している．

ここでは、例として GAP の IdGroup 関数を取り上げることにしよう．この機能を使うには群論関係の小規模なデータベースが必要だが，標準ではインストールされない．これは別途インストールしておかなければならないから，後で手順を説明する．

```
sage: G = gap('Group((1,2,3)(4,5), (3,4))')
sage: G
Group( [ (1,2,3)(4,5), (3,4) ] )
sage: G.Center()
Group( () )
sage: G.IdGroup()
[ 120, 34 ]
sage: G.Order()
120
```

上と同じ処理を，GAP インタ=フェイスを明示的には呼び出さずに Sage から実行するには:

```
sage: G = PermutationGroup([[(1,2,3),(4,5)],[(3,4)]])
sage: G.center()
Subgroup generated by [()] of (Permutation Group with generators [(3,4), (1,2,3)(4,
↪5)])
sage: G.group_id()
[120, 34]
sage: n = G.order(); n
120
```

(GAP の機能の一部は，二種類の標準外 Sage パッケージをインストールしないと使うことができない．コマンド行から sage -optional と入力すると，インストール可能なパッケージの一覧が表示される．その一覧に gap_packages-x.y.z といった項目があるから， sage -i gap_packages-x.y.z を実行してパッケージをインストールする．一部の GPL ではない GAP パッケージは，GAP ウェブサイト [GAPkg] からダウンロードし，$SAGE_ROOT/local/lib/gap-4.4.10/pkg で解凍してからインストールする必要がある．)

4.3 Singular

Singular は，グレブナー基底，多変数多項式の gcd，平面曲線の Rieman-Roch 空間に対する基底、因数分解などを始めとする各種処理のための，大規模で十分に枯れたライブラリを提供する．実例として，多変数多項式の因数分解を Sage から Singular へのインターフェイスを使って実行してみよう (.... は入力しないこと):

```
sage: R1 = singular.ring(0, '(x,y)', 'dp')
sage: R1
polynomial ring, over a field, global ordering
//   coefficients: QQ
//   number of vars : 2
//        block   1 : ordering dp
//                  : names    x y
//        block   2 : ordering C
sage: f = singular('9*y^8 - 9*x^2*y^7 - 18*x^3*y^6 - 18*x^5*y^6 +'
....: '9*x^6*y^4 + 18*x^7*y^5 + 36*x^8*y^4 + 9*x^10*y^4 - 18*x^11*y^2 -'
....: '9*x^12*y^3 - 18*x^13*y^2 + 9*x^16')
```

f を定義できたので，その内容を表示してから因数分解を試みる．

```
sage: f
9*x^16-18*x^13*y^2-9*x^12*y^3+9*x^10*y^4-18*x^11*y^2+36*x^8*y^4+18*x^7*y^5-18*x^5*y^
↪6+9*x^6*y^4-18*x^3*y^6-9*x^2*y^7+9*y^8
sage: f.parent()
Singular
sage: F = f.factorize(); F
[1]:
   _[1]=9
   _[2]=x^6-2*x^3*y^2-x^2*y^3+y^4
   _[3]=-x^5+y^2
[2]:
   1,1,2
sage: F[1][2]
x^6-2*x^3*y^2-x^2*y^3+y^4
```

GAP 節における GAP の実行例のように，Singular インターフェイスを直には使わず上の因数分解を行なうこともできる (Sage が実際の計算に裏で Singular インターフェイスを使っていることに変わりない)．以下の例でも， .... は入力しないこと:

```
sage: x, y = QQ['x, y'].gens()
sage: f = (9*y^8 - 9*x^2*y^7 - 18*x^3*y^6 - 18*x^5*y^6 + 9*x^6*y^4
....: + 18*x^7*y^5 + 36*x^8*y^4 + 9*x^10*y^4 - 18*x^11*y^2 - 9*x^12*y^3
....: - 18*x^13*y^2 + 9*x^16)
sage: factor(f)
(9) * (-x^5 + y^2)^2 * (x^6 - 2*x^3*y^2 - x^2*y^3 + y^4)
```

4.4 Maxima

Maxima は，LISP 言語の実装の一種と共に Sage に同梱されてくる．(Maxima が標準でプロットに用いる)gnuplot パッケージは，Sage でも非標準パッケージとして公開されている．Maxima が得意とするのは，記号処理である．Maxima は関数の微分と積分を記号的に実行し，1 階の常微分方程式 (ODE) と大半の線形 2 次 ODE を解くことができるし，任意次数の線形 ODE をラプラス変換することもできる．さらに Maxima には多様な特殊関数も組込まれており，gnuplot を介したプロット機能も備えている．(掃き出し法や固有値問題などに始まる) 行列操作や行列方程式の解法を実行し，多項式方程式を解くことも可能だ．

Sage/Maxima インターフェイスの使い方を例示するため，ここでは i, j 要素の値が i/j で与えられる行列を作成してみよう．ただし $i, j = 1, \ldots, 4$ とする．

```
sage: f = maxima.eval('ij_entry[i,j] := i/j')
sage: A = maxima('genmatrix(ij_entry,4,4)'); A
matrix([1,1/2,1/3,1/4],[2,1,2/3,1/2],[3,3/2,1,3/4],[4,2,4/3,1])
sage: A.determinant()
0
sage: A.echelon()
matrix([1,1/2,1/3,1/4],[0,0,0,0],[0,0,0,0],[0,0,0,0])
sage: A.eigenvalues()
[[0,4],[3,1]]
sage: A.eigenvectors().sage()
[[[0, 4], [3, 1]], [[1, 0, 0, -4], [0, 1, 0, -2], [0, 0, 1, -4/3]], [[1, 2, 3, 4]]]
```

使用例をもう一つ示す:

```
sage: A = maxima("matrix ([1, 0, 0], [1, -1, 0], [1, 3, -2])")
sage: eigA = A.eigenvectors()
sage: V = VectorSpace(QQ,3)
sage: eigA
[[[-2,-1,1],[1,1,1]],[[[0,0,1]],[[0,1,3]],[[1,1/2,5/6]]]]
sage: v1 = V(sage_eval(repr(eigA[1][0][0]))); lambda1 = eigA[0][0][0]
sage: v2 = V(sage_eval(repr(eigA[1][1][0]))); lambda2 = eigA[0][0][1]
sage: v3 = V(sage_eval(repr(eigA[1][2][0]))); lambda3 = eigA[0][0][2]

sage: M = MatrixSpace(QQ,3,3)
sage: AA = M([[1,0,0],[1, - 1,0],[1,3, - 2]])
sage: b1 = v1.base_ring()
sage: AA*v1 == b1(lambda1)*v1
True
sage: b2 = v2.base_ring()
sage: AA*v2 == b2(lambda2)*v2
True
sage: b3 = v3.base_ring()
sage: AA*v3 == b3(lambda3)*v3
True
```

最後に，Sage 経由で openmath を使ってプロットを行なう際の手順を紹介する．以下の例題の多くは，Maxima のレファレンスマニュアルのものを修正したものだ．

関数の 2 次元プロットをするには (... は入力しない)

```
sage: maxima.plot2d('[cos(7*x),cos(23*x)^4,sin(13*x)^3]','[x,0,1]',  # not tested
....:     '[plot_format,openmath]')
```

次の「ライブ」3 次元プロットは，マウスで動かすことができる (.... は入力しない):

```
sage: maxima.plot3d ("2^(-u^2 + v^2)", "[u, -3, 3]", "[v, -2, 2]",  # not tested
....:     '[plot_format, openmath]')
sage: maxima.plot3d("atan(-x^2 + y^3/4)", "[x, -4, 4]", "[y, -4, 4]",  # not tested
....:     "[grid, 50, 50]",'[plot_format, openmath]')
```

次に有名なメビウスの帯を 3 次元プロットしてみよう (.... は入力しない).

```
sage: maxima.plot3d("[cos(x)*(3 + y*cos(x/2)), sin(x)*(3 + y*cos(x/2)), y*sin(x/2)]",
 # not tested
....:     "[x, -4, 4]", "[y, -4, 4]", '[plot_format, openmath]')
```

プロットの最後の例は，あの「クラインの壺」である (.... は入力しない):

```
sage: _ = maxima("expr_1: 5*cos(x)*(cos(x/2)*cos(y) + sin(x/2)*sin(2*y)+ 3.0) - 10.0")
sage: _ = maxima("expr_2: -5*sin(x)*(cos(x/2)*cos(y) + sin(x/2)*sin(2*y)+ 3.0)")
sage: _ = maxima("expr_3: 5*(-sin(x/2)*cos(y) + cos(x/2)*sin(2*y))")
sage: maxima.plot3d ("[expr_1, expr_2, expr_3]", "[x, -%pi, %pi]",  # not tested
....:     "[y, -%pi, %pi]", "['grid, 40, 40]", '[plot_format, openmath]')
```

第5章　Sage，LaTeXと仲間たち

著者: Rob Beezer (2010-05-23)

Sage と TeX 派生の組版システム LaTeX は，緊密な相乗的協働関係にある．ここでは，最も基本的なものから始めて，かなり特殊かつ難解なものに至るまでの両者の多様な相互関係を概観する．(このチュートリアルを初めて読む場合は，この節は飛ばしてかまわない)

5.1 概観

Sage は LaTeX を多種多様な形で利用している．利用のメカニズムを理解する早道は，まず三種類の代表的な利用法のあらましを把握しておくことだろう．

1. Sage では全オブジェクトに対して LaTeX 形式の印字表現が可能になっている．Sage オブジェクト `foo` の LaTeX 形式による印字表現を見たければ，Sage ノートブックまたは Sage コマンド入力行で `latex(foo)` と実行する．出力された文字列を TeX の数式モード (例えば一対の `$` で囲まれた領域) 内で処理すると十分に使える `foo` の表現が得られるはずだ．これについては以下で実例を見る．

   同じ手順によって Sage を LaTeX 文書の作成に活用することができる．目的のオブジェクトを Sage 上で作成あるいは計算しておいてから，そのオブジェクトを `latex()` 経由で表示した出力を LaTeX 文書にカット/ペーストすればよい．

2. ノートブックインターフェイスは，web ブラウザ上で数式を明瞭に表示するため MathJax を使用するよう設定されている．MathJax は JavaScript で書かれたオープンソースの数式表示エンジンで，現行のブラウザ全てで動作し，TeX 形式で表現された数式の全てではないにしても, その大部分を表示することができる．その目的は TeX で表わされた短い数式を正確に表示することであって，複雑な表や文書構造の表現と管理を支援するものではない．ノートブックにおける見たところ自動的な数式のレンダリング表示は，数式オブジェクトの `latex()` 出力を MathJax が動作可能な HTML 形式に変換して得られたものである．MathJax はそれ自身のスケーラブルフォントを利用しており，方程式など TeX 形式で表現されたテキストを静的なインライン画像に変換する方法より優れているといえる．

3. Sage コマンドラインあるいはノートブック上で MathJax では処理しきれないほど LaTeX コードが複雑化してしまった場合は，システム上で公開運用されている LaTeX によって処理することもできる．Sage には，Sage 自体をビルドして使用するために必要な物ほとんど全てが含まれているが，TeX はその例外となっている．場合によっては，TeX システムおよび関連の変換ユーティリティ群を別途インストールしなければ欲しい機能が完全には使えないこともあるかもしれない．

ここで `latex()` 関数の基本的な用法を試してみよう．

```
sage: var('z')
z
```

(次のページに続く)

(前のページからの続き)

```
sage: latex(z^12)
z^{12}
sage: latex(integrate(z^4, z))
\frac{1}{5} \, z^{5}
sage: latex('a string')
\text{\texttt{a{ }string}}
sage: latex(QQ)
\Bold{Q}
sage: latex(matrix(QQ, 2, 3, [[2,4,6],[-1,-1,-1]]))
\left(\begin{array}{rrr}
2 & 4 & 6 \\
-1 & -1 & -1
\end{array}\right)
```

ノートブック上では MathJax の基本機能は自動的に動作するが，ここでは `MathJax` クラスを使って MathJax の働きを少しばかり観察してみることにしよう．`MathJax` クラスの `eval` 関数は Sage オブジェクトを LaTeX 形式に変換し，さらに CSS の "math"クラスを呼び出す HTML にラップして MathJax を起動する．

```
sage: from sage.misc.html import MathJax
sage: mj = MathJax()
sage: var('z')
z
sage: mj(z^12)
<html>\[z^{12}\]</html>
sage: mj(QQ)
<html>\[\newcommand{\Bold}[1]{\mathbf{#1}}\Bold{Q}\]</html>
sage: mj(ZZ[x])
<html>\[\newcommand{\Bold}[1]{\mathbf{#1}}\Bold{Z}[x]\]</html>
sage: mj(integrate(z^4, z))
<html>\[\frac{1}{5} \, z^{5}\]</html>
```

5.2 基本的な使い方

上で概観したように，Sage の LaTeX 支援機能を利用する最も基本的な方法は `latex()` 関数を使って数学オブジェクトの LaTeX 形式による印字表現を生成することである．生成された LaTeX 表現は別仕立ての LaTeX 文書に貼り付けて利用すればよい．この方法はノートブックでも Sage コマンドラインでも同じように通用する．

もう一つの手軽な方法に `view()` コマンドの使用がある．Sage コマンドラインで `view(foo)` を実行すると `foo` の LaTeX 表現が得られるから，これを LaTeX 文書に取り込んで使っているシステムにインストールされた TeX で処理する．TeX の出力を適切なビューワーで表示して出来上がりである．使用する TeX のバージョンや，それに応じた出力とビューワーは選択することができる (*LaTeX 処理のカスタマイズ* 節を参照).

ノートブック上では, `view(foo)` コマンドは MathJax がワークシート上で LaTeX 表現を正しくレンダリングできるよう適切な HTML と CSS の組を生成する．ユーザーの目に映るのは Sage デフォルトの ASCII 文

字出力ではなく，きれいにフォーマットされた数式出力ということになる．ただし Sage の数学オブジェクト全てが MathJax で表示可能な LaTeX 表現をもつとは限らない．表示できない場合には MathJax ではなくシステム上の TeX を使って処理し，出力をワークシートに表示可能な画像へ変換することができる．これに必要な一連の処理をどうやってコントロールするかについては *LaTeX コード生成のカスタマイズ* 節で解説する．

ノートブックには TeX を利用するための機能があと二つある．一つはワークシートの最初のセルのすぐ上に並ぶ四つのドロップダウンボックスの右にある "Typeset"ボタンである．これをチェックしておくと，以降はセルの評価結果は MathJax で処理されて印刷品質で表示される．ただし，この機能は遡及的に作用するわけではなく，事前に評価済みのセルについては評価し直してやる必要がある．つまるところ "Typeset" ボタンにチェックを入れるのは，以降のセル評価出力を `view()` コマンドでラップして表示することを意味することになる．

ノートブックの TeX 支援機能の二つ目は，ワークシートの注釈に TeX 形式が利用可能な点である．ワークシート上でセルの間にカーソルが移動すると青色のバーが現れるから，そこで <shift-click> すると TinyMCE というちょっとしたワードプロセッサが起動する．これを使えばテキスト入力を行ない，かつ WSIWYG エディタ経由で HTML と CSS を作成して書式付けすることができる．つまりワークシート内で書式付きテキストを作成し注釈として付加することが可能なわけだ．一方，ダラー記号 (`$`) あるいはダラー記号 2 つ (`$$`) の間に挟まれた文字列は MathJax によってインラインあるいはディスプレイ数式として解釈される．

5.3 LaTeX コード生成のカスタマイズ

`latex()` コマンドによる LaTeX コードの生成をカスタマイズする方法は何通りも用意されている．ノートブックでも Sage コマンドラインでも，すでに `latex` という名前のオブジェクトが定義済みで，`latex.` (ピリオド `.` に 注意) と入力して `[Tab]` キーを押せばメソッドの一覧を表示することができる．

ここでは `latex.matrix_delimiters` メソッドに注目してみよう．このメソッドを使えば，行列を囲む記号を大丸かっこ，角かっこ，中かっこ，縦棒などに変更することができる．何か形式上の制限があるわけではないから，好きなように組み合わせてかまわない．LaTeX で使われるバックスラッシュには，Python の文字列内でエスケープするためもう 1 個のバックスラッシュを付ける必要があることに注意．

```
sage: A = matrix(ZZ, 2, 2, range(4))
sage: latex(A)
\left(\begin{array}{rr}
0 & 1 \\
2 & 3
\end{array}\right)
sage: latex.matrix_delimiters(left='[', right=']')
sage: latex(A)
\left[\begin{array}{rr}
0 & 1 \\
2 & 3
\end{array}\right]
sage: latex.matrix_delimiters(left='\\{', right='\\}')
sage: latex(A)
```

(次のページに続く)

(前のページからの続き)

```
\left\{\begin{array}{rr}
0 & 1 \\
2 & 3
\end{array}\right\}
```

latex.vector_delimiters メソッドも同様の機能をもつ.

(整数，有理数，実数など) 標準的な環や体をどんな書体で表示するかは latex.blackboard_bold メソッドによって制御することができる．デフォルトではボールド体で表示されるが，手書きの場合にやるように黒板ボールド体 (重ね打ち体) を使うこともできる．それには Sage のビルトインマクロ \Bold{} を再定義してやればよい.

```
sage: latex(QQ)
\Bold{Q}
sage: from sage.misc.html import MathJax
sage: mj=MathJax()
sage: mj(QQ)
<html>\[\newcommand{\Bold}[1]{\mathbf{#1}}\Bold{Q}\]</html>
sage: latex.blackboard_bold(True)
sage: mj(QQ)
<html>\[\newcommand{\Bold}[1]{\mathbb{#1}}\Bold{Q}\]</html>
sage: latex.blackboard_bold(False)
```

新しいマクロやパッケージなどを追加して，TeX の高い拡張性を利用することができる．まず，ノートブックで MathJax が短い TeX コードを解釈する際に使われる，自分用のマクロを追加してみよう.

```
sage: latex.extra_macros()
''
sage: latex.add_macro("\\newcommand{\\foo}{bar}")
sage: latex.extra_macros()
'\\newcommand{\\foo}{bar}'
sage: var('x y')
(x, y)
sage: latex(x+y)
x + y
sage: from sage.misc.html import MathJax
sage: mj=MathJax()
sage: mj(x+y)
<html>\[\newcommand{\foo}{bar}x + y\]</html>
```

以上のようなやり方で追加したマクロは，MathJax では対応しきれない大規模な処理が発生してシステム上の TeX が呼ばれるような場合にも使われる．自立した LaTeX 文書のプリアンブルを定義する latex_extra_preamble コマンドの使い方は以下で具体例を示す．これまで通り Python 文字列中ではバックスラッシュが二重になっていることに注意.

```
sage: latex.extra_macros('')
```

(次のページに続く)

(前のページからの続き)

```
sage: latex.extra_preamble('')
sage: from sage.misc.latex import latex_extra_preamble
sage: print(latex_extra_preamble())
\newcommand{\ZZ}{\Bold{Z}}
...
\newcommand{\Bold}[1]{\mathbf{#1}}
sage: latex.add_macro("\\newcommand{\\foo}{bar}")
sage: print(latex_extra_preamble())
\newcommand{\ZZ}{\Bold{Z}}
...
\newcommand{\Bold}[1]{\mathbf{#1}}
\newcommand{\foo}{bar}
```

長く複雑な LaTeX 表現を処理するために，LaTeX ファイルのプリアンブルでパッケージ類を付加してやることができる．`latex.add_to_preamble` コマンドを使えば好きなものをプリアンブルに取り込めるし，`latex.add_package_to_preamble_if_available` コマンドはプリアンブルへの取り込みを実行する前に，指定したパッケージが利用可能かどうかをチェックしてくれる．

以下の例ではプリアンブルで geometry パッケージを取り込み，ページ上で TeX に割り当てる領域 (実質的にはマージン) サイズを指定している．例によって Python 文字列のバックスラッシュは二重になっていることに注意．

```
sage: from sage.misc.latex import latex_extra_preamble
sage: latex.extra_macros('')
sage: latex.extra_preamble('')
sage: latex.add_to_preamble('\\usepackage{geometry}')
sage: latex.add_to_preamble('\\geometry{letterpaper,total={8in,10in}}')
sage: latex.extra_preamble()
'\\usepackage{geometry}\\geometry{letterpaper,total={8in,10in}}'
sage: print(latex_extra_preamble())
\usepackage{geometry}\geometry{letterpaper,total={8in,10in}}
\newcommand{\ZZ}{\Bold{Z}}
...
\newcommand{\Bold}[1]{\mathbf{#1}}
```

あるパッケージの存在確認をした上で取り込みを実行することもできる．例として，ここでは存在しないはずのパッケージのプリアンブルへの取り込みを試みてみよう．

```
sage: latex.extra_preamble('')
sage: latex.extra_preamble()
''
sage: latex.add_to_preamble('\\usepackage{foo-bar-unchecked}')
sage: latex.extra_preamble()
'\\usepackage{foo-bar-unchecked}'
sage: latex.add_package_to_preamble_if_available('foo-bar-checked')
sage: latex.extra_preamble()
```

(次のページに続く)

(前のページからの続き)

```
'\\usepackage{foo-bar-unchecked}'
```

5.4 LaTeX 処理のカスタマイズ

システムで公開運用されている TeX システムから好みの種類を指定して，出力形式を変更することも可能だ．さらに，ノートブックが MathJax(簡易 TeX 表現用) と TeX システム (複雑な LaTeX 表現用) を使い分ける仕方を制御することができる．

`latex.engine()` コマンドを使えば，複雑な LaTeX 表現に遭遇した場合，システム上で運用されている TeX の実行形式 `latex`, `pdflatex` または `xelatex` の内どれを使って処理するかを指定することができる．`view()` が sage コマンドラインから発行されると，実行形式 `latex` が選択されるから，出力は dvi ファイルとなり sage における表示にも (xdvi のような)dvi ビューワーが使われる．これに対し，TeX 実行形式として `pdflatex` が設定された状態で sage コマンドラインから `view()` を呼ぶと，出力は PDF ファイルとなって Sage が表示に使用するのはシステム上の PDF 表示ユーティリティ (acrobat，okular, evince など) となる．

ノートブックでは，簡単な TeX 表現だから MathJax で処理できるのか，あるいは複雑な LaTeX 表現のためシステム上の TeX を援用すべきなのか，判断の手掛りを与えてやる必要がある．手掛りとするのは文字列リストで，リストに含まれる文字列が処理対象の LaTeX 表現に含まれていたらノートブックは MathJax を飛ばして latex(もしくは `latex.engine()` コマンドで指定された実行形式) による処理を開始する．このリストは `latex.add_to_mathjax_avoid_list` および `latex.mathjax_avoid_list` コマンドによって指定される．

```
sage: # not tested
sage: latex.mathjax_avoid_list([])
sage: latex.mathjax_avoid_list()
[]
sage: latex.mathjax_avoid_list(['foo', 'bar'])
sage: latex.mathjax_avoid_list()
['foo', 'bar']
sage: latex.add_to_mathjax_avoid_list('tikzpicture')
sage: latex.mathjax_avoid_list()
['foo', 'bar', 'tikzpicture']
sage: latex.mathjax_avoid_list([])
sage: latex.mathjax_avoid_list()
[]
```

ノートブック上で `view()` コマンド，あるいは "Typeset"ボタンがチェックされた状態で LaTeX 表式が生成されたが， `latex.mathjax_avoid_list` によってシステム上の LaTeX が別途必要とされたとしよう．すると，その LaTeX 表式は (`latex.engine()` で設定した) 指定の TeX 実行形式によって処理される．しかし，Sage は (コマンドライン上のように) 外部ビューワーを起動して表示する代わりに，結果をセル出力としてきっちりトリミングした 1 個の画像に変換してからワークシートに挿入する．

変換が実際にどう実行されるかについては，いくつもの要因が影響している．しかし大勢は，TeX 実行形式として何が指定されているか，そして利用可能な変換ユーティリティは何かによって決まるようだ．`dvips`, `ps2pdf`, `dvipng` そして `ImageMagick` に含まれる `convert` の四種の優秀な変換ユーティリティがあれば，

あらゆる状況に対処できるだろう．目標はワークシートに挿入して表示可能な PNG ファイルを生成することで，LaTeX 表式から dvi 形式への LaTeX エンジンによる変換が成功していれば，dvipng が変換を仕上げてくれる．LaTeX 表式と LaTeX エンジンの生成する dvi 形式に dvipng が扱えない special 命令が入っている場合には，dvips でポストスクリプトファイルへ変換する．ポストスクリプトあるいは pdflatex エンジンによって出力された PDF ファイルは convert ユーティリティによって PNG 形式へ変換される．ここで紹介した二つの変換ユーティリティは have_dvipng() と have_convert() ルーチンを使って存在を確認することができる．

必要な変換プログラムがインストールされていれば変換は自動的に行われる．ない場合は，何が不足していて，どこからダウンロードすればよいかを告げるエラーメッセージが表示される．

どうすれば複雑な LaTeX 表式を処理できるのか，その具体例として次節 (具体例： *tkz-graph* による連結グラフの作成) では LaTeX の tkz-graph パッケージを使って高品位の連結グラフを作成する方法を解説する．他にも例が見たければ，パッケージ化済みのテストケースが用意されている．利用するには，以下で見るように sage.misc.latex.LatexExamples クラスのインスタンスである sage.misc.latex.latex_examples オブジェクトをインポートしなければならない．現在，このクラスには可換図，組合せ論グラフと結び目理論，および pstricks 関連の例題が含まれており，各々が xy, tkz-graph, xypic, pstricks パッケージの使用例になっている．インポートを終えたら， latex_examples をタブ補完してパッケージに含まれる実例を表示してみよう．例題各々で適正な表示に必要なパッケージや手続きが解説されている．(プリアンブルや latex エンジン類が全て上手く設定されていても) 実際に例題を表示するには view() を使わなくてはならない．

```
sage: from sage.misc.latex import latex_examples
sage: latex_examples.diagram()
LaTeX example for testing display of a commutative diagram produced
by xypic.

To use, try to view this object -- it will not work.  Now try
'latex.add_to_preamble("\\usepackage[matrix,arrow,curve,cmtip]{xy}")',
and try viewing again. You should get a picture (a part of the diagram arising
from a filtered chain complex).
```

5.5 具体例： tkz-graph による連結グラフの作成

tkz-graph パッケージを使って高品位の連結グラフ (以降はたんに「グラフ」と呼ぶ) を作成することができる．このパッケージは pgf ライブラリの tikz フロントエンド上に構築されている．したがってその全構成要素がシステム運用中の TeX で利用可能でなければならないが，TeX システムによっては必要なパッケージが全て最新になっているとは限らない．満足すべき結果を得るには，必要なパッケージの最新版をユーザの texmf 配下にインストールする必要が生ずるかもしれない．個人または公開利用のための TeX のインストールや管理運用についてはこのチュートリアルの範囲を越えるが，必要な情報は簡単に見つかるはずだ．必要なファイル類は *TeX* システムの完全な運用 節に挙げられている．

まずは土台とする LaTeX 文書のプリアンブルで必要なパッケージが付加されることを確認しておこう．グラフ画像は dvi ファイルを経由すると正しく生成されないので，latex エンジンとしては pdflatex プログラムを指定するのが一番だ．すると，Sage コマンドライン上で view(graphs.CompleteGraph(4)) の実行が可能になり，完結したグラフ K_4 の適切な PDF 画像が生成されるはずだ．

ノートブックでも同様の動作を再現するには, グラフを表わす LaTeX コードの MathJax による処理を mathjax

avoid list を使って抑止する必要がある．グラフは tikzpicture 環境で取り込まれるから，文字列 tikzpicture を mathjax avoid list に入れておくとよい．そうしてからワークシートで view(graphs.CompleteGraph(4)) を実行すると pdflatex が PDF ファイルを生成し，ついで convert ユーティリティが抽出した PNG 画像がワークシートの出力セルに挿入されることになる．ノートブック上でグラフを LaTeX にグラフを処理させるために必要なコマンド操作を以下に示す．

```
sage: from sage.graphs.graph_latex import setup_latex_preamble
sage: setup_latex_preamble()
sage: latex.extra_preamble() # random - システムで運用されている TeX に依存
'\\usepackage{tikz}\n\\usepackage{tkz-graph}\n\\usepackage{tkz-berge}\n'
sage: latex.engine('pdflatex')
sage: latex.add_to_mathjax_avoid_list('tikzpicture')  # not tested
sage: latex.mathjax_avoid_list()                      # not tested
['tikz', 'tikzpicture']
```

ここまで設定してから view(graphs.CompleteGraph(4)) のようなコマンドを実行すると，tkz-graph で表現されたグラフが pdflatex で処理されてノートブックに挿入される．LaTeX に tkz-graph 経由でグラフを描画させる際には多くのオプションが影響するが，詳細はこの節の範囲を越えている．興味があればレファレンスマニュアル "LaTeX Options for Graphs"の解説を見てほしい．

5.6 TeX システムの完全な運用

TeX を Sage に統合して運用する際, 高度な機能の多くはシステムに独立してインストールされた TeX がないと利用できない．Linux 系システムでは TeXlive を基にした基本 TeX パッケージを採用しているディストリビューションが多く, OSX では TeXshop, Windows では MikTeX などが使われている．convert ユーティリティは ImageMagick パッケージ (簡単にダウンロード可能) に含まれているし，dvipng, ps2pdf と dvips の三つのプログラムは TeX パッケージに同梱されているはずだ．また dvipng は http://sourceforge.net/projects/dvipng/ から， ps2pdf は Ghostscript の一部として入手することもできる．

連結グラフの作画には，PGF ライブラリの新しいバージョンに加えて tkz-graph.sty を https://www.ctan.org/pkg/tkz-graph から入手する必要があり，さらに tkz-arith.sty　とおそらく tkz-berge.sty も https://www.ctan.org/pkg/tkz-berge から入手する必要がある．

5.7 外部プログラム

TeX と Sage のさらなる統合運用に役立つプログラムが三つある．その一番目が sagetex で，この TeX マクロ集を使えば LaTeX 文書から Sage 上の多様なオブジェクトに対する演算や組み込みコマンド latex() によるフォーマットなどを実行することができる．LaTeX 文書のコンパイル処理過程で，Sage の演算や LaTeX によるフォーマット支援などの全ての機能も自動的に実行されるのである．sagetex を使えば，例えば数学試験作成において，問題の計算そのものを Sage に実行させて対応する解答を正確に維持管理することなどが可能になる．詳細は *SageTeX を使う* 節を参照してほしい．

第6章　プログラミング

6.1 Sage ファイルの読み込みと結合

ここでは，独立したファイルに保存したプログラムを Sage に読み込む方法を解説する．まず，ファイル example.sage に以下のプログラムを保存しておこう:

```
print("Hello World")
print(2^3)
```

example.sage ファイルを読み込んで実行するには， load コマンド を使う．

```
sage: load("example.sage")
Hello World
8
```

実行中のセッションに Sage ファイルを結合するには， attach コマンドが使える:

```
sage: attach("example.sage")
Hello World
8
```

こうして attach したファイル example.sage に変更を加えてから Sage で空行を入力すると (return を押す)，Sage は自動的に example.sage の内容を読み込んで実行する．

attach は結合したファイルに変更が生じると自動的に読込んで実行してくれるので，デバッグの時にはとりわけ便利なコマンドになる．これに対し load の方は，コマンド実行時に一度だけファイルを読込んで実行するだけだ．

example.sage を読込むと Sage は内容を Python プログラムへと変換し，これを Python インタプリタが実行する．この Python への変換は最小限に留められていて，整数リテラルを Integer() で，浮動小数点数リテラルを RealNumber() でラップし， ^ を ** で，また R.2 を R.gen(2) で置換するなどといった程度である．変換された example.sage のコードは， example.sage と同じディレクトリに example.sage.py という名前のファイルとして保存されている．この example.sage.py に入っているコードは:

```
print("Hello World")
print(Integer(2)**Integer(3))
```

たしかに整数リテラルはラップされ， ^ は ** に置換されている．(Python では ^ は「排他的論理和」， ** は「べき乗」を意味する．)

こうした前処理は sage/misc/interpreter.py として実装されている．

コードブロックが改行で開始されているなら，インデントされた複数行のコードを Sage にペーストすることができる (ファイルの読込みについてはブロック開始時の改行も不要だ)．しかし Sage 上にコードを取り込むには，そのコードをファイルに保存してから，先に説明したように attach するのが一番安全だ．

6.2 実行形式の作成

数学的演算処理では実行速度がとりわけ重要になる．Python は使い勝手のよい非常に高水準の言語ではあるが，計算の種類によってはスタティックに型付けされたコンパイラ言語で実行した方が処理速度が数桁も速くなる場合がある．Sage にも，もし完全に Python のみで実装していたら遅くて使いものにならなかったはずの機能がある．そうした状況を切り抜けるために，Sage は Cython と呼ばれるコンパイラ言語版 Python を利用している ([Cyt] と [Pyr])．Cython は，Python と C の両方に似ていて，リスト内包表記、条件制御、それに += などを含む Python の構文の大半を使うことができるし，Python で書いておいたモジュールをインポートすることも可能だ．加えて，C 言語の変数を自由に宣言し，かつ好きな C ライブラリをコード内から直接呼び出すことができる．Cython で書いたコードは C に変換され，C コンパイラでコンパイルされることになる．

Sage コードの実行形式を作成するには，ソースファイルの拡張子を (.sage ではなく) .spyx とする．コマンドラインインターフェイスを使っている場合，実行形式をインタプリタ用コードのときと全く同じやり方で attach あるいは load することができる (今のところ，ノートブックインターフェイスでは Cython で書いたコードの読込みと結合はできない)．実際のコンパイル処理は，ユーザーが特に何も指定しなくとも裏で実行されている．作成された実行形式の共有ライブラリは $HOME/.sage/temp/hostname/pid/spyx に格納されるが，Sage の終了時に消去される．

Sage は spyx ファイルに対しては前処理をしない．だから例えば spyx ファイル中の 1/3 は有理数 1/3 ではなく 0 と評価される．Sage ライブラリに含まれる関数 foo を spyx ファイルで使用したければ， sage.all をインポートしてから sage.all.foo として呼び出す必要がある．

```
import sage.all
def foo(n):
    return sage.all.factorial(n)
```

6.2.1 他ファイル中の C 関数を使う

別途 *.c ファイルで定義された C 関数を使うのも簡単だ．実例を見てみよう．まず，同じディレクトリにファイル test.c と test.spyx を作り，内容はそれぞれ以下の通りであるとする:

純粋に C で書かれたプログラムは test.c :

```
int add_one(int n) {
  return n + 1;
}
```

Cython プログラムが入っているのが test.spyx:

```
cdef extern from "test.c":
    int add_one(int n)

def test(n):
    return add_one(n)
```

すると，次の手順で C プログラムを取り込むことができる:

```
sage: attach("test.spyx")
Compiling (...)/test.spyx...
sage: test(10)
11
```

Cython ソースファイルから生成された C 言語コードをコンパイルするために，さらにライブラリ foo が必要な場合は，元の Cython ソースに clib foo という行を加える．同様に、他にも C ソースファイル bar が必要ならば Cython ソースに取り込みを宣言する行 cfile bar を加えてコンパイルすればよい．

6.3 スタンドアロン Python/Sage スクリプト

以下のスタンドアロン型 Sage スクリプトは，整数や多項式などを因数分解する:

```
#!/usr/bin/env sage

import sys

if len(sys.argv) != 2:
    print("Usage: %s <n>" % sys.argv[0])
    print("Outputs the prime factorization of n.")
    sys.exit(1)

print(factor(sage_eval(sys.argv[1])))
```

このスクリプトを実行するには， SAGE_ROOT を PATH に含めておかなければならない．スクリプト名を factor とすると，実行は以下のような具合になる:

```
bash $ ./factor 2006
2 * 17 * 59
```

6.4 データ型

Sage に現れるオブジェクトには，全て明確に定義されたデータ型が割り当てられている．Python は豊富な組み込み型を備えているが，それをさらに多彩に拡張しているのが Sage のライブラリだ．Python の組み込み型としては，string(文字列)，list(リスト)，タプル (tuple)，int(整数)，float(浮動小数点数) などがある．実際に型を表示してみると:

```
sage: s = "sage"; type(s)
<... 'str'>
sage: s = 'sage'; type(s)      # シングルあるいはダブル クォーテーションのどちらも使える
<... 'str'>
sage: s = [1,2,3,4]; type(s)
<... 'list'>
sage: s = (1,2,3,4); type(s)
<... 'tuple'>
sage: s = int(2006); type(s)
<... 'int'>
sage: s = float(2006); type(s)
<... 'float'>
```

Sage では，さらに多様な型が加わる．その一例がベクトル空間である:

```
sage: V = VectorSpace(QQ, 1000000); V
Vector space of dimension 1000000 over Rational Field
sage: type(V)
<class 'sage.modules.free_module.FreeModule_ambient_field_with_category'>
```

この V に適用できるのは，あらかじめ定められた特定の関数に限られる．他の数学ソフトウェアでは，そうした関数の呼び出しに「関数型記法」 foo(V,...) が用いられているようだ．これに対し，Sage では V の型 (クラス) に付属する関数群が定められていて，JAVA や C++に見られるようなオブジェクト指向型の構文 V.foo(...) で関数呼び出しが行なわれる．オブジェクト指向の世界では，関数が莫大な数になってもグローバルな名前空間を混乱なく運用することが可能になる．たとえ機能の異なる関数群が同じ foo という名前を持っていたとしても，型チェックや場合分け抜きで引数の型に応じた適切な関数が自動的に呼び出されるのである．さらに，ある関数の名前を他の意味で使い回しても関数そのものは使い続けることができる (例えば zeta を何かの変数名として使った後でも，リーマンのゼータ関数の 0.5 における値を求めるには s=.5; s.zeta() と入力すれば足りる).

```
sage: zeta = -1
sage: s=.5; s.zeta()
-1.46035450880959
```

ごく慣用化している場合については，通常の関数型記法を使うこともできる．これは便利であると同時に，数学表現にはそもそもオブジェクト指向型記法になじまないものもあるためだ．ここで少し例を見てみよう．

```
sage: n = 2; n.sqrt()
sqrt(2)
```

(次のページに続く)

(前のページからの続き)

```
sage: sqrt(2)
sqrt(2)
sage: V = VectorSpace(QQ,2)
sage: V.basis()
    [
    (1, 0),
    (0, 1)
    ]
sage: basis(V)
    [
    (1, 0),
    (0, 1)
    ]
sage: M = MatrixSpace(GF(7), 2); M
Full MatrixSpace of 2 by 2 dense matrices over Finite Field of size 7
sage: A = M([1,2,3,4]); A
[1 2]
[3 4]
sage: A.charpoly('x')
x^2 + 2*x + 5
sage: charpoly(A, 'x')
x^2 + 2*x + 5
```

A のメンバ関数を全て表示するには，タブ補完入力を利用すればよい．コマンド入力の遡行検索とタブ補完 節で説明したように，これは `A.` と入力してから，キーボードの `[tab]` キーを押すだけのことだ．

6.5 リスト，タプル，シーケンス

リスト型には，任意の型の要素を格納することができる．(大半のコンピュータ代数システムとは違って)C や C++などと同じように，Sage でもリストの要素番号は 0 から始まる:

```
sage: v = [2, 3, 5, 'x', SymmetricGroup(3)]; v
[2, 3, 5, 'x', Symmetric group of order 3! as a permutation group]
sage: type(v)
<... 'list'>
sage: v[0]
2
sage: v[2]
5
```

リストの要素番号は，Python の int 型でなくとも平気だ．Sage の Integer クラスが使えるのは言うまでもない (Rational クラスを含めて、 `__index__` メソッドが有効なクラスであれば何でも使える).

```
sage: v = [1,2,3]
sage: v[2]
3
sage: n = 2       # SAGE の整数
sage: v[n]        # 全く問題なし
3
sage: v[int(n)]   # これも大丈夫
3
```

range 関数は，Python の int 型からなるリストを生成する (Sage の Integer ではないことに注意):

```
sage: list(range(1, 15))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

この range が便利なのは，リスト内包表記を使ってリストを生成する場合だ:

```
sage: L = [factor(n) for n in range(1, 15)]
sage: L
[1, 2, 3, 2^2, 5, 2 * 3, 7, 2^3, 3^2, 2 * 5, 11, 2^2 * 3, 13, 2 * 7]
sage: L[12]
13
sage: type(L[12])
<class 'sage.structure.factorization_integer.IntegerFactorization'>
sage: [factor(n) for n in range(1, 15) if is_odd(n)]
[1, 3, 5, 7, 3^2, 11, 13]
```

以上のようなリスト内包表記を使ったリスト生成については， [PyT] に詳しい．

とりわけ使い勝手が良いのが，リストのスライシングだ．リスト L のスライシング L[m:n] は， m 番目の要素に始まり $n-1$ 番目の要素で終わる部分リストを返す．以下に例を示そう:

```
sage: L = [factor(n) for n in range(1, 20)]
sage: L[4:9]
[5, 2 * 3, 7, 2^3, 3^2]
sage: L[:4]
[1, 2, 3, 2^2]
sage: L[14:4]
[]
sage: L[14:]
[3 * 5, 2^4, 17, 2 * 3^2, 19]
```

タプルはリストに似ているが，これがいったん生成された後は変更できない不変性 (immutable) オブジェクトである点で異なる．

```
sage: v = (1,2,3,4); v
(1, 2, 3, 4)
sage: type(v)
```

(次のページに続く)

(前のページからの続き)

```
<... 'tuple'>
sage: v[1] = 5
Traceback (most recent call last):
...
TypeError: 'tuple' object does not support item assignment
```

Sage で使われる第三のリスト類似データ型が，シーケンスである．リストやタプルと違って，シーケンスは Python 本体の組み込み型ではない．デフォルトではシーケンス型は可変だが，以下の例で見るように Sequence クラスのメソッド set_immutable を使って不変性を与えることができる．あるシーケンスの全要素は，シーケンス・ユニバースと呼ばれる共通のペアレントを持つ．

```
sage: v = Sequence([1,2,3,4/5])
sage: v
[1, 2, 3, 4/5]
sage: type(v)
<class 'sage.structure.sequence.Sequence_generic'>
sage: type(v[1])
<class 'sage.rings.rational.Rational'>
sage: v.universe()
Rational Field
sage: v.is_immutable()
False
sage: v.set_immutable()
sage: v[0] = 3
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
```

シーケンスはリストから導き出すことができて，リストが使える文脈では常に利用することができる:

```
sage: v = Sequence([1,2,3,4/5])
sage: isinstance(v, list)
True
sage: list(v)
[1, 2, 3, 4/5]
sage: type(list(v))
<... 'list'>
```

不変性シーケンスの例としては，ベクトル空間の基底系があげられる．基底系そのものが変わっては困るから，これは当然のことだ．

```
sage: V = QQ^3; B = V.basis(); B
[
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
```

(次のページに続く)

(前のページからの続き)

```
]
sage: type(B)
<class 'sage.structure.sequence.Sequence_generic'>
sage: B[0] = B[1]
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
sage: B.universe()
Vector space of dimension 3 over Rational Field
```

6.6 ディクショナリ

ディクショナリ (「連想配列」と呼ばれる場合もある) とは，文字列、数値、タプルなどのハッシュ可能なオブジェクトから任意のオブジェクトへの写像のことである．(ハッシュ可能オブジェクトについての詳細は http://docs.python.org/tut/node7.html と http://docs.python.org/lib/typesmapping.html を参照．)

```
sage: d = {1:5, 'sage':17, ZZ:GF(7)}
sage: type(d)
<... 'dict'>
sage: list(d.keys())
[1, 'sage', Integer Ring]
sage: d['sage']
17
sage: d[ZZ]
Finite Field of size 7
sage: d[1]
5
```

三番目の例を見ると分るように，ディクショナリのインデックス (キー) として整数環のような複雑なオブジェクトでも使うことができる．

上の例のディクショナリは，同じデータを含むリストに直すことができる:

```
sage: list(d.items())
[(1, 5), ('sage', 17), (Integer Ring, Finite Field of size 7)]
```

ディクショナリに含まれるキーと値の対を反復に利用する場合に，よく使われるイディオムがある:

```
sage: d = {2:4, 3:9, 4:16}
sage: [a*b for a, b in d.items()]
[8, 27, 64]
```

最後の出力を見ると判るように，ディクショナリ内は整列されていない．

6.7 集合

Python には集合 (set) 型が組込まれている．集合型の主な利点としては，標準的な集合演算が可能になるだけではなく，ある要素が集合に属するかどうかを極めて高速に判定する機能を備えている点があげられる．

```
sage: X = set([1,19,'a']);   Y = set([1,1,1, 2/3])
sage: X  # random sort order
{1, 19, 'a'}
sage: X == set(['a', 1, 1, 19])
True
sage: Y
{2/3, 1}
sage: 'a' in X
True
sage: 'a' in Y
False
sage: X.intersection(Y)
{1}
```

さらに，Sage は (Python の組み込み集合型を使って実装されたものも含まれる) 独自の集合型を備えており，こちらには Sage に固有の付加機能がいくつか加えられている．この Sage 独自の集合型を生成するには， `Set(...)` を使う．例えば

```
sage: X = Set([1,19,'a']);   Y = Set([1,1,1, 2/3])
sage: X # random sort order
{'a', 1, 19}
sage: X == Set(['a', 1, 1, 19])
True
sage: Y
{1, 2/3}
sage: X.intersection(Y)
{1}
sage: print(latex(Y))
\left\{1, \frac{2}{3}\right\}
sage: Set(ZZ)
Set of elements of Integer Ring
```

6.8 イテレータ

イテレータ (iterator) は最近になって Python に加えられた機能で，数学指向のアプリケーション作成にはとりわけ便利なものだ．以下で実例を見ていくことにするが，使用法の詳細は [PyT] を見てほしい．まず 10000000 までの非負整数の平方に関するイテレータを作ってみよう．

```
sage: v = (n^2 for n in range(10000000))
sage: next(v)
```

(次のページに続く)

(前のページからの続き)

```
0
sage: next(v)
1
sage: next(v)
4
```

今度は，素数 p から $4p+1$ の形の素数に関するイテレータを作り，最初の数個を見てみることにする．

```
sage: w = (4*p + 1 for p in Primes() if is_prime(4*p+1))
sage: w         # 次の行の 0xb0853d6c はランダムに生成された 16 進数
<generator object <genexpr> at ...>
sage: next(w)
13
sage: next(w)
29
sage: next(w)
53
```

有限体，整数など，ある種の環にはイテレータが付随している:

```
sage: [x for x in GF(7)]
[0, 1, 2, 3, 4, 5, 6]
sage: W = ((x,y) for x in ZZ for y in ZZ)
sage: next(W)
(0, 0)
sage: next(W)
(0, 1)
sage: next(W)
(0, -1)
```

6.9 ループ，関数，制御文，比較

for ループの一般的な使用法については，これまでに何度も実例を見ている．Python では、 for ループ構文はインデントで分節されている．次のような具合だ:

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

for 文はコロン : で終っており，ループの本体すなわち print(i) がインデントされていることに注意 (GAP や Maple に見られる "do"や "od"はない)．Python では，このインデントが重要な役割を果たしている．以下の例のように，Sage では : に続けて enter キーを押すと自動的にインデントが挿入される．

```
sage: for i in range(5):
....:     print(i)  # ここでは [Enter] を 2 回押す
....:
0
1
2
3
4
```

代入には = 記号，比較には == 記号を使う:

```
sage: for i in range(15):
....:     if gcd(i,15) == 1:
....:         print(i)
....:
1
2
4
7
8
11
13
14
```

if， for および while 文のブロック構造が，インデントによって決まっているところに注目：

```
sage: def legendre(a,p):
....:     is_sqr_modp=-1
....:     for i in range(p):
....:         if a % p == i^2 % p:
....:             is_sqr_modp=1
....:     return is_sqr_modp

sage: legendre(2,7)
1
sage: legendre(3,7)
-1
```

むろん，上のコードの目的は Python/Sage によるプログラムの特徴を例示することであって，Legendre 記号の効率的な実装にはなっていない．Sage に付属している関数 kronecker は，PARI の C ライブラリを経由して Legendre 記号を効率良く計算する．

最後に注意したいのは ==， !=， <=， >=， >， < などを使った比較演算では，比較対象の量は可能ならば自動的に同じ型に変換されることだ:

```
sage: 2 < 3.1; 3.1 <= 1
True
False
sage: 2/3 < 3/2;   3/2 < 3/1
True
True
```

記号を含む不等号の判定には bool 関数を用いる:

```
sage: x < x + 1
x < x + 1
sage: bool(x < x + 1)
True
```

Sage における異種オブジェクト間の比較演算では，まず対象オブジェクトの共通ペアレント型への正準型強制 (変換) が試みられる (「型強制」(coercion) の詳細については **ペアレント，型変換および型強制** 節を参照)．比較演算は，この型強制が成功後，変換されたオブジェクトに対して実行される．変換が不可能であれば，その二つのオブジェクトは等しくないと判定されることになる．二つの変数が同一のオブジェクトを参照 (レファレンス) しているかどうかを調べるには、 `is` を使う．例えば Python の int 型 1 は唯一だが，Sage の Integer 型 1 は違う:

```
sage: 1 is 2/2
False
sage: 1 is 1
False
sage: 1 == 2/2
True
```

以下に示す例の，前半の二つの不等式は `False` になる．これは正準写像 $\mathbf{Q} \to \mathbf{F}_5$ が存在せず， $\mathbf{F}_5$ 上の 1 を $1 \in \mathbf{Q}$ と比較する基準がないためである．一方、後半の不等式は関係する正準写像 $\mathbf{Z} \to \mathbf{F}_5$ が存在するため `True` と判定される．比較する式の左辺右辺を入れ替えても結果は変わらない．

```
sage: GF(5)(1) == QQ(1); QQ(1) == GF(5)(1)
False
False
sage: GF(5)(1) == ZZ(1); ZZ(1) == GF(5)(1)
True
True
sage: ZZ(1) == QQ(1)
True
```

警告: Sage における比較演算は， $1 \in \mathbf{F}_5$ は $1 \in \mathbf{Q}$ と等しいとみなす Magma よりも制限がきつい．

```
sage: magma('GF(5)!1 eq Rationals()!1')  # optional - magma オプションで magma が必要
true
```

6.10 プロファイリング

著者: Martin Albrecht (malb@informatik.uni-bremen.de)

> 「早計な最適化は，あらゆる災厄の源である」 -- ドナルド・クヌース

コードのボトルネック，つまり処理時間の大半を費している部分を洗い出さなければならない場面がある．ボトルネックが判らなければどこを最適化すべきかの判定もできないからだ．コードのボトルネックを特定する作業のことをプロファイリングと呼ぶが，Python/Sage にはプロファイリングの手段が何通りも用意されている．

一番簡単な方法は、対話型シェルで prun コマンドを実行することだ．prun は，使われた関数と各関数の処理時間の一覧を表示してくれる．例として，有限体上の行列積演算 (バージョン 1.0 で、まだ低速だ) をプロファイルしてみよう．その手順は:

```
sage: k,a = GF(2**8, 'a').objgen()
sage: A = Matrix(k,10,10,[k.random_element() for _ in range(10*10)])
```

```
sage: %prun B = A*A
       32893 function calls in 1.100 CPU seconds

Ordered by: internal time

ncalls tottime percall cumtime percall filename:lineno(function)
 12127  0.160   0.000   0.160  0.000 :0(isinstance)
  2000  0.150   0.000   0.280  0.000 matrix.py:2235(__getitem__)
  1000  0.120   0.000   0.370  0.000 finite_field_element.py:392(__mul__)
  1903  0.120   0.000   0.200  0.000 finite_field_element.py:47(__init__)
  1900  0.090   0.000   0.220  0.000 finite_field_element.py:376(__compat)
   900  0.080   0.000   0.260  0.000 finite_field_element.py:380(__add__)
     1  0.070   0.070   1.100  1.100 matrix.py:864(__mul__)
  2105  0.070   0.000   0.070  0.000 matrix.py:282(ncols)
  ...
```

ここで ncalls は関数の呼出し回数， tottime はその関数が費した総時間 (配下の関数群の呼出しにかかった時間は除く)， percall は tottime を ncalls で割って得られる平均総消費時間， cumtime は関数本体とそれが呼出している配下の全関数双方の処理にかかった (つまり注目している関数の呼出しから開放までの) 全時間， percall は cumtime を呼出し回数で割って得られる関数の平均処理時間で，filename:lineno(function) は各関数の所在情報を示している．結局のところ，prun の表示一覧の上位にある関数ほど処理に要する負荷も大きいことが判る．これで，どこを最適化すべきか考えやすくなるはずだ．

これまでと同じように， prun? と入力するとプロファイラの使い方と表示情報の解釈について詳細を見ることができる．

プロファイル情報をオブジェクトとして保存しておいて，後で詳しく調べてもよい:

```
sage: %prun -r A*A
sage: stats = _
sage: stats?
```

注意: `stats = prun -r A\*A` と実行すると，文法エラーが表示される．これは prun が IPython のシェルコマンドであって，通常の関数ではないためである．

プロファイル情報をグラフィカルに表示したければ，hotshot プロファイラ， `hotshot2cachetree` スクリプトと kcachegrind プログラム (Unix 系のみ) などを使えばよい．上の例と同じ作業を hotshot プロファイラで実行すると:

```
sage: k,a = GF(2**8, 'a').objgen()
sage: A = Matrix(k,10,10,[k.random_element() for _ in range(10*10)])
sage: import hotshot
sage: filename = "pythongrind.prof"
sage: prof = hotshot.Profile(filename, lineevents=1)
```

```
sage: prof.run("A*A")
<hotshot.Profile instance at 0x414c11ec>
sage: prof.close()
```

得られた結果は，現ディレクトリのファイル `pythongrind.prof` に保存されている．これを cachegrind フォーマットに変換すれば，内容をビジュアル化して把握することができる．

システムのコマンドシェルに戻り

```
hotshot2calltree -o cachegrind.out.42 pythongrind.prof
```

として変換を実行すると，出力ファイル `cachegrind.out.42` 内の情報を `kcachegrind` コマンドを使って検討することができる．ファイルの慣例的名称 `cachegrind.out.XX` は残念ながら変更できない．

第7章　SageTeXを使う

SageTeX パッケージを使うと，Sage による処理結果を LaTeX 文書に埋め込むことができるようになる．利用するためには，まず SageTeX を「インストール」しておかなければならない (*TeX に SageTeX の存在を教える* 節を参照).

7.1 具体例

ここでは, ごく簡単な例題を通して SageTeX の利用手順を紹介する．完全な解説ドキュメントと例題ファイルは，ディレクトリ `SAGE_ROOT/venv/share/doc/sagetex` に置いてある．`SAGE_ROOT/venv/share/texmf/tex/latex/sagetex` にある Python スクリプトは何か役に立つ場面があるはずだ．以上の `SAGE_ROOT` は，Sage をインストールしたディレクトリである．

SageTeX の動作を体験するために，まず SageTeX のインストール手続き (*TeX に SageTeX の存在を教える* 節) を実行し，以下のテキストを `st_example.tex` などという名前で保存しておいてほしい:

> **警告**
>
> このテキストを "live"ヘルプから表示すると命令未定義エラーになる．正常に表示するためには "static" ヘルプで表示すること．

```
\documentclass{article}
\usepackage{sagetex}

\begin{document}

Using Sage\TeX, one can use Sage to compute things and put them into
your \LaTeX{} document. For example, there are
$\sage{number_of_partitions(1269)}$ integer partitions of $1269$.
You don't need to compute the number yourself, or even cut and paste
it from somewhere.

Here's some Sage code:

\begin{sageblock}
    f(x) = exp(x) * sin(2*x)
\end{sageblock}
```

(次のページに続く)

(前のページからの続き)

```
The second derivative of $f$ is

\[
  \frac{\mathrm{d}^{2}}{\mathrm{d}x^{2}} \sage{f(x)} =
  \sage{diff(f, x, 2)(x)}.
\]

Here's a plot of $f$ from $-1$ to $1$:

\sageplot{plot(f, -1, 1)}

\end{document}
```

この st_example.tex をいつも通りに LaTeX で処理する．すると LaTeX は以下のような文句をつけてくるだろう:

```
Package sagetex Warning: Graphics file
sage-plots-for-st_example.tex/plot-0.eps on page 1 does not exist. Plot
command is on input line 25.

Package sagetex Warning: There were undefined Sage formulas and/or
plots. Run Sage on st_example.sagetex.sage, and then run LaTeX on
st_example.tex again.
```

注目してほしいのは, LaTeX が通常の処理で生成するファイル群に加えて，st_example.sage というファイルが出来ていることだ．これは st_example.tex の処理時に生成された Sage スクリプトで, 上で見た LaTeX 処理時のメッセージは，この st_example.sage を Sage で実行せよという内容である．その通り実行すると st_example.tex を再び LaTeX で処理せよと告げられるが, その前に新しいファイル st_example.sout が生成されていることに注意．このファイルには Sage の演算結果が LaTeX テキストに挿入して利用可能な形式で保存されている．プロット画像の EPS ファイルを含むディレクトリも新規作成されている．ここで LaTeX 処理を実行すると，Sage の演算結果とプロットの全てが LaTeX 文書に収められることになる．

上の処理に用いられた各マクロの内容はごく簡単に理解できる．sageblock 環境は Sage コードを入力通りに組版し，ユーザーが Sage を動かすとそのコードを実行する．\sage{foo} とすると，Sage 上で latex(foo) を実行したのと同じ結果が LaTeX 文書に挿入される．プロット命令はやや複雑だが，もっとも単純な場合である \sageplot{foo} は foo.save('filename.eps') を実行して得られた画像を文書へ挿入する役割を果たす.

要するに，必要な作業は以下の三段階になる:

- LaTeX で .tex ファイルを処理
- 生成された .sage ファイルを Sage で実行
- LaTeX で .tex ファイルを再処理

作業中に LaTeX 文書内の Sage コマンドを変更しない場合，Sage による処理は省略することができる．

SageTeX は到底以上で語り尽せるものでなく，Sage と LaTeX は共に複雑で強力なツールだ．SAGE_ROOT/

venv/share/doc/sagetex にある SageTeX のドキュメントを読むことを強くお勧めする.

7.2 TeX に SageTeX の存在を教える

Sage はおおむね自己完結的なシステムなのだが，正しく機能するために外部ツールの介入を要する部分があることも確かだ．SageTeX もそうした部分の一つである.

SageTeX パッケージを使えば Sage による演算やプロットを LaTeX 文書に埋め込むことが可能になる．SageTeX はデフォルトで Sage にインストールされるが，LaTeX 文書で利用する前に，運用している TeX システムへ SageTeX の存在を教えておかねばならない.

鍵になるのは，TeX が sagetex.sty を発見できるかどうかである．この sagetex.sty は，SAGE_ROOT を Sage がビルトあるいはインストールされたディレクトリとすると，SAGE_ROOT/venv/share/texmf/tex/latex/sagetex/ に置かれているはずだ．TeX が sagetex.sty を読めるようにしてやらなければ，SageTeX も動作できないのである．これを実現するには何通りかのやり方がある.

- 第一の，かつ一番簡単な方法は， sagetex.sty を作成すべき LaTeX 文書と同じディレクトリ内にコピーしておくことである．TeX は組版処理の際に現ディレクトリを必ずサーチするから，この方法は常に有効だ.

  ただし，このやり方には二つのちょっとした問題点がある．一つ目は，このやり方では使用しているシステムが重複した sagetex.sty だらけになってしまうこと．二つ目の，もっと厄介な問題は，この状態で Sage が更新されて SageTeX も新しいバージョンになった場合，SageTeX を構成する Python コードや LaTeX コードとの食い違いが生じて実行時にエラーが発生しかねない点である.

- 第二の方法は，環境変数 TEXINPUTS を利用することである．bash シェルを使っているなら

  ```
  export TEXINPUTS="SAGE_ROOT/venv/share/texmf//:"
  ```

  と実行すればよい．ただし SAGE_ROOT は Sage のインストール先ディレクトリである．上の実行例では，行末にスラッシュ 2 個とコロンを付け忘れないでいただきたい．実行後は，TeX と関連ツールが SageTeX スタイルファイルを見つけられるようになる．上のコマンド行を .bashrc に付加して保存しておけば設定を永続させることができる．bash 以外のシェルを使っている場合， TEXINPUTS 変数を設定するためのコマンドも異なる可能性がある．設定法については，自分の使っているシェルのドキュメントを参照のこと.

  この方法にも瑕はある．ユーザが TeXShop や Kile，あるいは Emacs/AucTeX などを使っている場合，必ずしも環境変数を認識してくれるとは限らないのである．これらのアプリケーションが常にシェル環境を通して LaTeX を起動するわけではないからだ.

  インストール済みの Sage を移動したり，新バージョンを旧版とは違う場所にインストールした場合，先に紹介したコマンドも新しい SAGE_ROOT を反映させるように変更する必要がある.

- TeX に sagetex.sty の在処を教える第三の (かつ最善の) 方法は，このスタイルファイルを自分のホームディレクトリのどこか都合のよい所にコピーしておくことだ．TeX ディストリビューションの多くは，パッケージを求めてホームディレクトリにある texmf ディレクトリを自動的に探索するようになっている．このディレクトリを正確に特定するには，コマンド

  ```
  kpsewhich -var-value=TEXMFHOME
  ```

を実行する．すると /home/drake/texmf や /Users/drake/Library/texmf などと表示されるはずだから， SAGE_ROOT/venv/share/texmf/ 内の tex/ ディレクトリをホームディレクトリの texmf にコピーするには

```
cp -R SAGE_ROOT/venv/share/texmf/tex TEXMFHOME
```

などとする．もちろん， SAGE_ROOT を実際に Sage をインストールしたディレクトリとするのはこれまでと同じことで， TEXMFHOME は上で見た kpsewhich コマンドの結果で置き換える．

Sage をアップグレードしたら SageTeX がうまく動かなくなったという場合は，上記の手順をもう一度繰り返すだけで SageTeX の Sage と TeX 関連部分が同期する．

- 複数ユーザに対応するシステムでは，以上の手続きを変更して sagetex.sty を公開運用中の TeX ディレクトリにコピーすればよい．おそらく一番賢いコピー先は TEXMFHOME ディレクトリではなく，コマンド

```
kpsewhich -var-value=TEXMFLOCAL
```

の実行結果に従うことだろう．出力は /usr/local/share/texmf のようになるはずで，上と同じように tex ディレクトリを TEXMFLOCAL ディレクトリ内にコピーする．ついで TeX のパッケージデータベースを更新しなければならないが，これは簡単で，ルート権限で

```
texhash TEXMFLOCAL
```

と実行すればよい．ただし TEXMFLOCAL を現実に合わせて変更するのは先と同じだ．これでシステムの全ユーザは SageTeX パッケージへアクセス可能になり，Sage が利用できれば SageTeX も使えるようになる．

警告

肝心なのは, LaTeX が組版処理時に使う sagetex.sty ファイルと, Sage が援用する SageTeX のバージョンが一致していることである．Sage を更新したら, あちこちに散らばった古いバージョンの sagetex.sty を面倒でも全て削除してやらなければいけない．

SageTeX 関連ファイルをホームディレクトリの texmf ディレクトリ内にコピーしてしまうこと (先に紹介した第三の方法) をお勧めするのは，この面倒があるからである．第三の方法にしておけば，Sage 更新後も SageTeX を正常に動作させるために必要な作業はディレクトリを一つコピーするだけになる．

7.2.1 SageTeX ドキュメント

厳密には Sage のインストール一式には含まれないものの，ここで SageTeX のドキュメントが SAGE_ROOT/venv/share/doc/sagetex/sagetex.pdf に配置されていることに触れておきたい．同じディレクトリには例題ファイルと, これを LaTeX と SageTeX によってすでに組版処理した結果も用意されている (example.tex と example.pdf を参照)．これらのファイルは SageTeX ページ からダンロードすることもできる．

7.2.2 SageTeX と TeXLive

混乱を招きかねない問題点の一つとして，人気ある TeX ディストリビューション TeXLive 2009 に SageTeX が含まれている現実があげられる．これは有り難い感じがするかもしれないが，SageTeX に関して重要なのは Sage と LaTeX の各要素が同期していることだ．Sage と SageTeX は共に頻繁にアップデートされるが TeXLive はそうではないから，その「同期」のところで問題が生じる．この文の執筆時点 (2013 年 3 月) では，多くの Linux ディストリビューションが新しい TeXLive リリースに移行しつつある．しかし 2009 リリースもしぶとく生き残っていて，実はこれが SageTeX に関するバグレポートの主要な発生源になっているのだ．

このため **強く推奨** させていただきたいのは，SageTeX の LaTeX 関連部分は以上で説明したやり方で常に Sage からインストールすることである．上記の手順に従えば，SageTeX の Sage および LaTeX 対応部分の互換性が保証されるから，動作も正常に保たれる．SageTeX の LaTeX 対応部分を TeXLive から援用することはサポート対象外になる．

第8章　あとがき

8.1 なぜ Pyhton なのか

8.1.1 Python の強み

Sage の実装には主要言語として Python([Py] を参照) が採用されている．高速性を要する部分のコードはコンパイラ言語で実装されているものの，Python には固有の利点がある:

- **オブジェクト保存機能** が充実している．Python は，(ほぼ) いかなるオブジェクトでもディスクファイルあるいはデータベースとして保存するための機能を豊富に備えている．
- 優秀な **ドキュメント作成支援機能** を備えている．関数やパッケージ群のソースコードに対しては，ドキュメントの抽出と全具体例の検証までが自動化されている．具体例の自動検証は常時行なわれており，記載通りの動作を保証している．
- **メモリ管理**: Python は，入念に設計された頑健なメモリ管理機能と，循環参照を間違いなく処理するガーベッジコレクタを備えており，異なるファイル内で定義された局所変数群も問題なく扱うことができる．
- Python で利用できる **豊富なパッケージ群** は，Sage ユーザにとっても非常に有益であることは間違いない．数値解析，線形代数，2 次元および 3 次元グラフィクス，ネットワーキング (例えば twisted を経由した分散処理とサーバー構築)，データベース対応など，幅広い分野のパッケージが用意されている．
- **高い移植性**: 一般的なプラットフォームでは，Python をソースコードからビルドするのは簡単で時間もかからない仕事である．
- **例外処理**: Python は，細部まで考え抜かれ洗練された例外処理システムを備えている．この例外処理システムを使えば，呼出し先のコードでエラーが発生した場合でもプログラム本体を破綻なく障害から復帰させることが可能だ．
- **デバッガ**: 何らかの理由でコードがうまく動かない場合，Python のデバッガを使って詳細なスタックトレースを実行したり関連変数全ての状態を調べることができるし，スタック内を上下動することも可能だ．
- **プロファイラ**: Python のプロファイラを使えば，実際にコードを動かして関数の呼出し回数と処理時間を調べることができる．
- **汎用言語**: 数学指向の **独自言語** を開発した Magma，Maple，Mathematica，Matlab，GP/PARI，GAP，Macaulay 2，Simath などとは違って，Sage は広く普及している汎用言語 Python を採用している．オープンソースの代表的成功事例である Python は，安定した開発体制下，熟練したソフトウェア技術者多数によって研究開発と最適化が進められている ([PyDev] を参照).

8.1.2 前処理パーサ: Sage と Python の相違点

Python の数学機能には混乱を招きがちな面があり，Sage には Python とは異なる振舞いを示す部分がある．

- **べき乗の記法:** ** と ^ の意味が異なる. Python では ^ は "xor"の意味で，べき乗を意味しない．したがって，Python では

```
>>> 2^8
10
>>> 3^2
1
>>> 3**2
9
```

この ^ の使い方は奇妙な感じがするし，「排他的論理和」機能をほとんど使わない純粋な数学研究では無駄でもある．Sage では，Python に送る前に全コマンド行に対して文字列中にない限り ^ を ** に置換する前処理をしている:

```
sage: 2^8
256
sage: 3^2
9
sage: "3^2"
'3^2'
```

Sage では，ビット単位の XOR 演算子は ^^ である．同じ記号は代入演算子 ^^= にも応用されている:

```
sage: 3^^2
1
sage: a = 2
sage: a ^^= 8
sage: a
10
```

- **整数の除算:** Python における式 `2/3` は，数学者が当然と思う値にならない．Python では， `m` と `n` が int 型であれば， `m/n` つまり `m` を `n` で割った商も int 型になる．`2/3=0` となるのはこのためだ．Python コミュニティでは，仕様を変更して `2/3` は浮動小数点数 `0.6666...` を返し， `2//3` は `0` を返すよう修正すべきという議論が続いている．

  この問題を解決するために，Sage インタプリタは整数リテラルを `Integer()` でラップし，その除算を有理数のコンストラクタとして機能させている．具体例を見てみよう:

```
sage: 2/3
2/3
sage: (2/3).parent()
Rational Field
sage: 2//3
0
```

- **長整数:** Python 本体は，C 言語由来の int 型だけではなく任意精度整数をサポートしている．Python の任意精度整数は GMP 提供のものと比べると著しく速度が劣り，通常の int と区別するために末尾に L を付けて出力される仕様になっている (この仕様はすぐには変更されそうにない)．Sage の任意精度整数は GMP C-ライブラリを使って実装されており，末尾の L なしで出力される．

Sage では，(一部の人たちが内輪でやっているように)Python インタプリタそのものを改造することはせず，Python 言語をそのままの形で使っている．代りに IPython に対する前処理パーサを開発して，IPython コマンドラインの振舞いを数学者に馴染めるようにしてある．これにより既存の Python プログラムは例外なく Sage で使えることになるが，Sage にインポートして使うパッケージを開発する場合には標準的な Python の書法に従わなければならない．

(インターネットで見つけた Python ライブラリをインストールする場合は，`python` コマンドではなく `sage -python` としてインストール手順に従えばよい．こうすると，`sage -python setup.py install` の意味になる．)

8.2 Sage プロジェクトを手助けするには？

Sage プロジェクトに助力いただけるのなら，たいへん有難い．実質的なコードの提供から Sage ドキュメンテーションの追加やバグ報告まで，どんな形であれ大歓迎である．

開発者向けの情報については、Sage の Web ページをご覧いただきたい．優先順位とカテゴリー順に整理された Sage 関連プロジェクトの長いリストが見つかるはずだ．開発に役立つ情報は Sage Developer's Guide にも載っているし，Google グループ `sage-devel` も役立つ．

8.3 Sage を引用するには

Sage を使って論文を書く場合は，Sage による計算が行なわれたことを明記するため，以下の一文を参考文献として引用していただきたい (Version 8.7 の部分は実際に使用したバージョン番号に修正してください．):

```
[Sage] Sage Mathematics Software (Version 8.7).
       The Sage Development Team, 2019, https://www.sagemath.org.
```

さらに，Sage を構成する PARI，GAP，Singular，Maxima などのシステムの内，どれを計算に利用したのかを特定してそのシステムも引用していただけるようお願いする．もし計算に使ったソフトウェアがどれなのか確信がもてない場合は，Google グループ `sage-devel` で気軽に尋ねてみよう．こうした点については，*1 変数多項式* 節に詳しい話がある．

このチュートリアルを最後まで読み終えた方は，どのくらい時間がかかったか Google グループ `sage-devel` で教えていただければ幸いである．

どうか Sage で楽しんでほしい．

第9章　付録

9.1 算術二項演算子の優先順位

式 `3^2*4 + 2%5` は，どのようにして評価されるのだろうか？その値 (38) を決定しているのが，『演算子優先順位表』である．以下に示す順位表は、G. Rossum と F. Drake による *Python Language Reference Manual* の §5.14 にある表を基にしたものだ．表中，下へ行くほど演算子の優先順位が高くなっている．

演算子	説明
or	論理和
and	論理積
not	論理否定
in, not in	包含テスト
is, is not	同一性テスト
>, <=, >, >=, ==, !=	比較
+, -	加算，減算
*, /, %	乗算，除算，剰余
**, ^	べき乗

したがって `3^2*4 + 2%5` の値を求めるに際して，Sage はこの式を `((3^2)*4) + (2%5)` のように括弧で区切ることになる．次に `3^2` の値 9 を計算し，ついで `(3^2)*4` と `2%5` 両方の値を求めてから，全てを足し合わせて出来上がりだ．

第10章　Bibliography

第11章　Indices and tables

- genindex
- modindex
- search

関連図書

[Cyt] Cython, http://www.cython.org

[GAP] The GAP Group, GAP - Groups, Algorithms, and Programming, Version 4.4; 2005, https://www.gap-system.org

[GAPkg] GAP Packages, https://www.gap-system.org/Packages/packages.html

[GP] PARI/GP, https://pari.math.u-bordeaux.fr/

[Mag] Magma, http://magma.maths.usyd.edu.au/magma/

[Max] Maxima, http://maxima.sf.net/

[NagleEtAl2004] Nagle, Saff, and Snider. *Fundamentals of Differential Equations*. 6th edition, Addison-Wesley, 2004.

[Py] The Python language http://www.python.org/ Reference Manual http://docs.python.org/ref/ref.html. 日本 Python ユーザ会のサイト http://www.python.jp/ で日本語版ドキュメントが公開されている．感謝.

[PyB] The Python Beginner's Guide, https://wiki.python.org/moin/BeginnersGuide

[PyDev] Python Developer's Guide https://docs.python.org/devguide/

[PyLR] Python Library Reference, https://docs.python.org/ja/3/library/index.html

[Pyr] Pyrex, http://www.cosc.canterbury.ac.nz/~greg/python/Pyrex/

[PyT] The Python Tutorial, https://docs.python.org/ja/3/tutorial/

[SA] Sage web site, https://www.sagemath.org/

[Si] G.-M. Greuel, G. Pfister, and H. Schönemann. Singular 3.0. A Computer Algebra System for Polynomial Computations. Center for Computer Algebra, University of Kaiserslautern (2005). https://www.singular.uni-kl.de

[SJ] William Stein, David Joyner, Sage: System for Algebra and Geometry Experimentation, Comm. Computer Algebra {39} (2005) 61-64.

[ThreeJS] three.js, http://threejs.org

索引

E

ン