Sage

Christoph Koutschan

Research Institute for Symbolic Computation
Johannes Kepler Universität Linz, Austria

Computer Algebra Systems
8.11.2010

Sage

- general purpose CAS
- freely available
- open source under the GNU General Public License
- slogan: "open source alternative to Magma, Maple, Mathematica, and MATLAB"

History

William Stein speaking at ISSAC'2008 in Hagenberg. Photo: Christoph Koutschan

- founder and lead developer: William Stein
- first release in 2005
- many developers around the world
- release early, release often
- http://www.sagemath.org
- http://www.sagenb.com

Sage Philosophy

- **Useful:** Sage's intended audience is mathematics students, teachers, and research mathematicians. The aim is to provide software that can be used to explore and experiment with mathematical constructions in algebra, geometry, number theory, calculus, numerical computation, etc.
- **Efficient:** Be fast. Sage uses highly-optimized mature software like GMP, PARI, GAP, and NTL, and so is very fast at certain operations.
- **Free and open source:** The source code must be freely available and readable, so users can understand what the system is really doing and more easily extend it. Just as mathematicians gain a deeper understanding of a theorem by carefully reading or at least skimming the proof, people who do computations should be able to understand how the calculations work by reading documented source code.

Sage Philosophy (2)

- **Cooperation:** Provide robust interfaces to most other computer algebra systems, including PARI, GAP, Singular, Maxima, KASH, Magma, Maple, and Mathematica. Sage is meant to unify and extend existing math software.
- **Python:** Sage is based on the programming languages Python and Cython.

Python / Cython

- developed in the late 1980s
- interpreted high-level programming language
- design philosophy emphasizes code readability (indentation!)
- object-oriented, but also other programming styles are supported
- dynamically typed

Cython is an extension language for writing C and C++ modules for the Python runtime:

- call C/C++ functions from Cython
- strongly typed

Why Python?

- excellent support for documentation of functions and packages in the source code
- many packages available: numerical analysis and linear algebra, 2D and 3D visualization, networking
- Python is easy to compile from source on most platforms
- Python has a sophisticated system of exception handling
- debugger with extensive stack trace
- saving objects to disk files is well-supported in Python
- well thought out and robust memory manager and garbage collector
- existing language

Pre-Parser

Python syntax:

- `^` denotes XOR: `8^2` gives `10`
- `**` denotes exponentiation: `3**4` gives `81`
- `/` denotes the integer quotient: `7/3` gives `2`

Sage pre-parses all input such that

- `8^2` is interpreted as a power
- `7/3` is recognized as a rational number
- etc.

Quick start into Sage

Sage worksheets are displayed in notebook style in your web browser:

- upload worksheets from files
- download a worksheet to a file

Quick start into Sage

Sage worksheets are displayed in notebook style in your web browser:

- upload worksheets from files
- download a worksheet to a file

Observations:

- results of assignments are not displayed
- symbolic variables (except x) have to be defined:
  `y = var('y')`
- many commands allow standard syntax for input instead of object-oriented syntax:
  `2.sqrt().n()` can be inputted as `n(sqrt(2))`
- list indexing is 0-based
- every object has a type

Some issues

`i` and `I` are used to denote the imaginary unit, `n` is used for numerical approximation. These symbols are not protected!

Example:

```
i = 1;
reset('i')
n(pi)
n = 1;
n(pi)
```

$\longrightarrow$ use `CC(re,im)` to create complex numbers!

$\longrightarrow$ use `numerical_approx` instead of `n`!

Types

Strange experiences with types:

```
type(add)

type(sin)

type(exp)

type(ln)

type(log)

type(bessel_J)
```

Getting help

There are many ways to get help:

- Tab completion:
  - `f`+⟨Tab⟩ displays possible completions for `f`
  - `f.`+⟨Tab⟩ displays methods that are available for the object `f`
- `f?` displays help
- `f??` displays sourcecode
- Reference manual, Tutorial

Examples:

```
a = 2
a. ⟨Tab⟩
factor?
factor??
parent?
```

Functions

There are several ways to define functions.

1. Define a Python function:
   `def f(x):  return x^2`
2. Define a "callable symbolic expression":
   `g(x) = x^2`
3. Define unspecified functions:
   `h = function('h', x)`

Cases (2) and (3) describe functions in the mathematical sense (try, e.g., `g.diff()` and `h.diff()`). The variant (3) can be used for inputting differential equations (try, e.g., `desolve(diff(h,x)+h-1, [h,x])`). Functions of the form (1) can be plotted, but not differentiated or integrated.
Compare also `type(f)`, `type(g)`, and `type(h)`.

Plotting

Syntax is very simple:

```
plot(f, (-2,2))
circle((0,0), 1, rgbcolor=(1,1,0))
```

Plots can be added:

```
p1 = plot(f, (-2,2))
p2 = circle((0,0), 1, rgbcolor=(1,1,0)) show(p1+p2)
```

Programming in Sage

- Python syntax
- indentation matters!
- `def` ⟨function_name⟩`(`⟨arg1⟩, ⟨arg2⟩, . . . `):`
- no semicolons necessary for line ends
- the usual program control commands:
  `if (`⟨condition⟩`):` . . . `else/elif`
  `while (`⟨condition⟩`):`
  `for` ⟨variable⟩ `in` ⟨range⟩`:`
  etc.

Programming in Sage

A simple example (note the flexible use of arguments):

```
def is_divisible_by(number, divisor=2):
       return number%divisor == 0

is_divisible_by(6)

is_divisible_by(6, 5)

is_divisible_by(divisor=2, number=6)
```

Issues with functions

Try the following:

```
def f(x):
       if x<2:
           return 0
       else:
           return x-2
plot(f(x), 0, 4)
```

Issues with functions

Try the following:

```
def f(x):
        if x<2:
            return 0
        else:
            return x-2
plot(f(x), 0, 4)
```

Correct way to plot this function:

```
plot(f, 0, 4)
```

Cube root with bisections

```
def cuberoot1(c, prec):
    s = sign(c);
    c = abs(c);
    a = 0;
    b = max(c, 1);
    count = 0;
    while (b-a > prec):
        m = (a+b)/2;
        if (m^3 < c):
            a = m
        else:
            b = m
        count = count + 1
    print 'did %s loops'%(count)
    return n(s*m);
```

Cube root with Heron

```
def cuberoot2(c, prec):
    x = 1
    count = 0
    while (abs(x^3-c) > prec):
        x = (2*x^3+c)/(3*x^2)
        count = count + 1
    print 'did %s loops'%(count)
    return n(x);
```

Comparison

Compare

```
time cuberoot1(500, 10^(-6))
time cuberoot2(500, 10^(-6))
```

Comparison

Compare

```
time cuberoot1(500, 10^(-6))
time cuberoot2(500, 10^(-6))
```

Why is Heron much slower than bisections?
Count the number of loops:

- bisection method: 29 loops
- Heron: 13 loops

Comparison

Compare

```
time cuberoot1(500, 10^(-6))
time cuberoot2(500, 10^(-6))
```

Why is Heron much slower than bisections?
Count the number of loops:

- bisection method: 29 loops
- Heron: 13 loops

$\longrightarrow$ Keep control of your expressions!

Define new data types

Encapsulating mathematical objects with classes is a powerful technique that can help to simplify and organize your Sage programs.

Example: define a class that represents the list of even positive integers up to n:

```
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."
```

- the class Evens derives from the built-in type list
- the method `__init__` initializes the object when it is created
- the method `__repr__` prints the object out
- the `list` constructor method is called in line 4
- an object of the class Evens is created by `e = Evens(10)`

Basic Rings

When defining matrices, vectors, or polynomials, it is sometimes useful and sometimes necessary to specify the ring over which it is defined:

- `ZZ`: integers $\{\ldots, -2, 1, 0, 1, 2, \ldots\}$
- `QQ`: rational numbers
- `RR`: real numbers (floating point numbers)
- `CC`: complex numbers

Examples:

```
-4 in ZZ
pi in QQ
I+1 in RR
parent(4/3)
```

Polynomial Rings

Create a univariate polynomial ring:

1. `R = PolynomialRing(QQ, 't')`
2. `S = QQ['t']`
3. `T.<t> = PolynomialRing(QQ)` or `T.<t> = QQ['t']`

When using (1) and (2), the variable `t` will not be defined, in contrast to option (3).

Create multivariate polynomial rings

- `R = PolynomialRing(RealField(10), 3, "rst")`
- `R.<r,s,t> = PolynomialRing(QQ)`

Try also:

```
R.gens()
R.objgens()
```

Why Polynomial Rings?

Multiply two polynomials (in the ring):

```
ring = PolynomialRing(QQ, 'x')
p1 = ring.random_element(degree=200);
p2 = ring.random_element(degree=200);
time p3 = p1*p2
```

Now the same (with symbolic expressions):

```
v = var('v')
p1 = (p1 + v) - v
type(p1)
p2 = (p2 + v) - v
time p4 = expand(p1*p2)
expand(p3 - p4)
```

Multiplying polynomials is fast!

```
def time_polymult(deg):
    ring = PolynomialRing(ZZ, 'x')
    res = [];
    d = 1;
    while (d < deg):
        p1 = ring.random_element(degree=d)
        p2 = ring.random_element(degree=d)
        t = cputime()
        p1 = p1*p2
        res.append((d,cputime(t)))
        d = ceil(11/10*d)
    return res

timing = time_polymult(50000)
list_plot(timing)
```

Other Rings and Fields

Dividing two polynomials constructs an element of the fraction field (which Sage creates automatically):

```
x = QQ['x'].0
f = x^3 + 1
g = x^2 - 17
h = f/g; h
h.parent()
```

Now Laurent series: `R.<x> = LaurentSeriesRing(QQ)`

Sage also knows about finite fields, p-adic integers, the ring of algebraic numbers, etc.:

```
GF(3)
GF(27, 'a')
Zp(5)
sqrt(3) in QQbar
```

Ideals

Define two polynomials:

```
R = PolynomialRing(QQ, 2, "xy")
x,y = R.gens()
f = x^2+y^2+1; g = x^3+2*x*y-y^3
```

Next we create the ideal $\langle f, g\rangle$ generated by f and g:

```
id1 = (f,g)*R
id1
id1.groebner_basis()
```

$\longrightarrow$ These computations are executed in Singular.

Interface to Singular

Call Singular directly from Sage:

```
ring r1 = 0,(t,x,y,z),ls;
r1
ideal i1 = x-t2,y-t3,z;
```

Group Theory

```
G = PermutationGroup(['(1,2,3)(4,5)', '(3,4)'])
G.order()
G.random_element()
```

$\longrightarrow$ These computations are executed in GAP.

Number Theory

```
time
factor(109149643706541976490211967825493555058875303071)
```

```
time
pari('factor(10914964370654197649021196782549355505887530
```

```
time
mathematica('FactorInteger[1091496437065419764902119678254
```

$\longrightarrow$ These computations are executed in PARI.