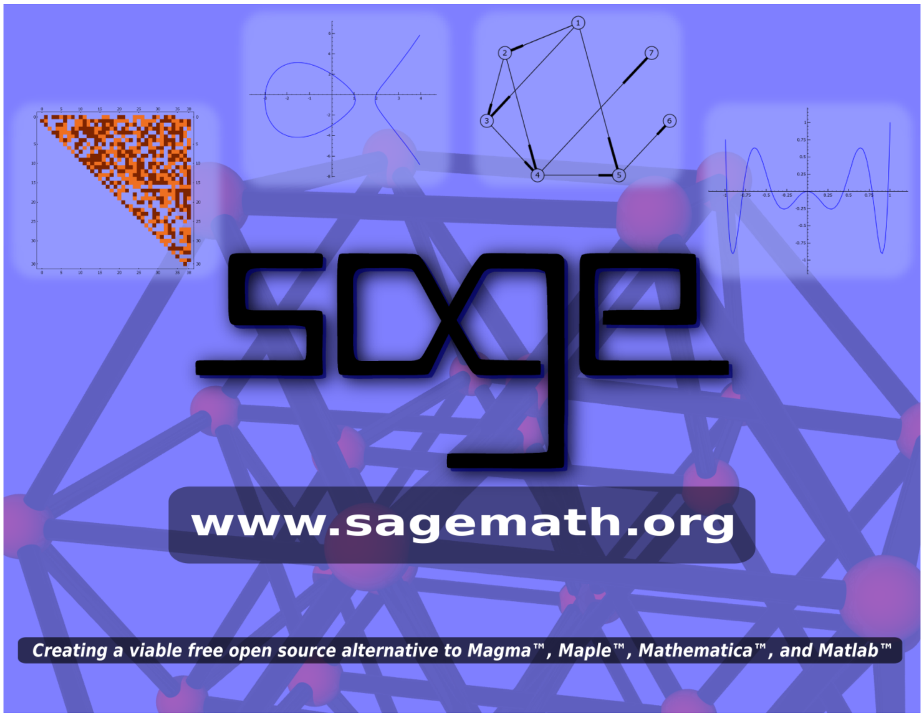
An Introduction to
Algebraic, Scientific, and Statistical Computing:
an Open Source Approach Using Sage

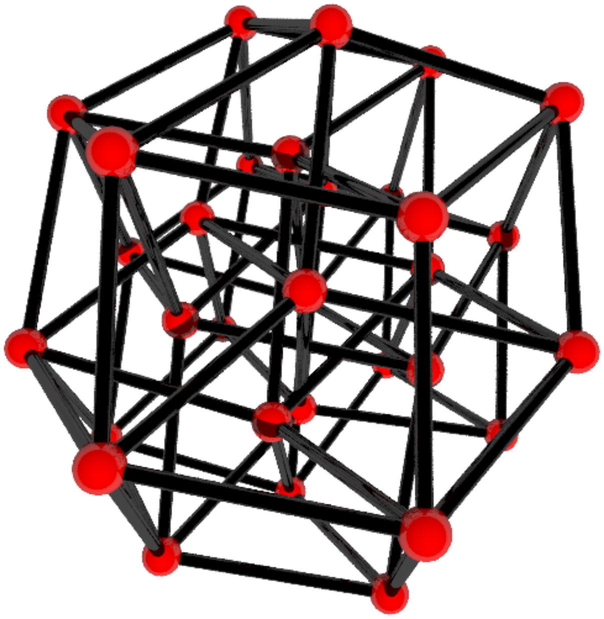

sage
www.sagemath.org
Creating a viable free open source alternative to Magma™, Maple™, Mathematica™, and Matlab™

William A. Stein

April 14, 2008

Abstract

This is an undergraduate textbook about algebraic, scientific, and statistical computing using the free open source mathematical software system Sage.



Contents

Preface

This is an undergraduate textbook about Sage, which is a free open source computational environment for the mathematical sciences. Sage includes optimized full-featured implementations of algorithms for computations in pure mathematics, applied mathematics, and statistics.

I started the Sage project in late 2004 as a project to provide a viable open source free alternative to the Magma computer algebra system [BCP97]. My main motivation for this was frustration with not being allowed to easily change or understand the internals of Magma, worry about the longterm future of Magma, and concern that students and researchers in number theory could not easily use the Magma-based tools that I had spent six hard years developing. I started Sage as a new project instead of switching from Magma to an existing open source system, since the only free open source software for number theory is PARI [ABC$^+$], whose functionality was far behind that of Magma in several areas of interest to me (exact linear algebra and arithmetic of algebraic curves). Since PARI development moves slowly (or I am a very impatient person!), I didn't think it was likely this would change in the near future. PARI is a superb program – it just doesn't meet my needs, and has design constraints that make it impossible to modify so that it does so.

The Sage mathematical software system takes a fresh approach to mathematical software development and architecture. For instance, one major distinction between Sage and older systems is that Sage uses a standard language. Maple, Mathematica, Magma, Matlab, PARI, Gap, etc., all use their own special purposes language written just for mathematics. In sharp contrast, one works with Sage using Python, which is one of the world's most popular *general purpose* scripting languages. This has some drawbacks, e.g., some mathematical expressions can be more difficult to express in Python than in Mathematica (say), but the overall pros greatly outweigh the cons. By using Python, one can use almost *anything* ever written in Python directly in Sage. And there is much useful Python code out there that adddresses a wide range of application areas[1]:

- "Python is fast enough for our site and allows us to *produce maintainable features in record times*, with a minimum of developers," said Cuong Do, Software Architect, **YouTube.com.**

[1] These quotes came from the Python website.

- “Google has made no secret of the fact they use Python a lot for a number of internal projects. Even knowing that, once *I was an employee, I was amazed at how much Python code there actually is in the Google source code system.*”, said Guido van Rossum, **Google**, creator of Python.

- “Python plays a key role in our production pipeline. Without it a project the size of *Star Wars: Episode II* would have been very difficult to pull off. From crowd rendering to batch processing to compositing, *Python binds all things together,*” said Tommy Burnette, Senior Technical Director, **Industrial Light & Magic.**

Instead of writing many of the core libraries from scratch like Maple, Mathematica, Magma, Gap, PARI, Singular, and Matlab did, in Sage I assembled together the best open source software out there, and built on it, always making certain that the complete system was easily buildable from source on a reasonable range of computers. I was able to do this to a large extent with Sage because of fortuitous timing: the components were out there and mature, their code is stable, and their copyright licences are clear and compatible (none of this was the case when the afformentioned math software was started). Of course there are drawbacks to this approach. Some of the upstream libraries can be difficult to understand, are written in a range of languages, and have different conventions than Sage. By strongly encouraging good relations between the Sage project and the projects that create many of the components of Sage, we turn these weakness into strengths.

A wide and vibrant community of developers and users have become involved with Sage. Due to the broad interests of this large community of developers, Sage has grown into a project with the following specific goal:

> **Mission Statement:** Provide a viable free open source alternative to Magma, Maple, Mathematica, and Matlab.

Among many other things, this mission statement implies that in order to succeed Sage should have a graphical user interface, 2D and 3D graphics, support for statistical and numerical computation, and much more.

Sage is starting to be recognized worldwide as a useful tool in mathematics education and research. Sage won first prize in the Scientific Category of the 2007 Tropheés du Libre.

The Sage development model has also matured. There are now regular releases of Sage about once every two weeks, and all code that goes into Sage is peer reviewed.

Sage has also received generous financial support from mathematics institutions (including the Clay Mathematics Institute, MSRI, PIMS, and IPAM), universities such as University of Washington and University of Bristol, and from the National Science Foundation, the Department of Defense, and Microsoft Corporation.

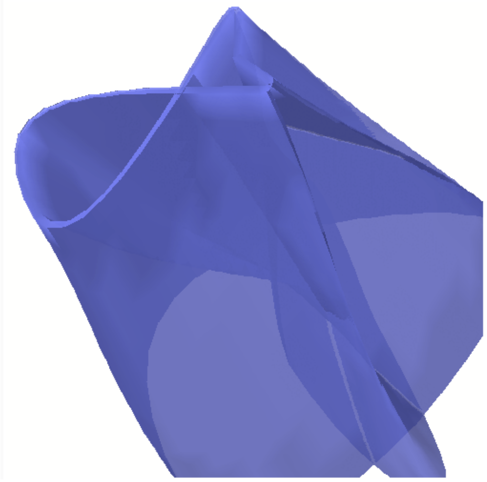
Exercise 0.1.

1. Make a list of 5 free open source programs you have used.
2. For each program, write down a corresponding commercial program, if it exists.
3. List some of the pros *and cons* from your perspective of the free program versus the commercial version.

Acknowledgements: Many people gave me feedback on this book including Fernando Perez, Timothy Clemans, Hector Villafuerte, Georg Muntingh, Robert Bradshaw...

Chapter 1

Computing with Sage



1.1 Installing and Using Sage

1.1.1 Installing Sage

You are strongly encouraged to follow along with all examples in this book. Get Sage up and running now! Try out everything, modify things, experiment, look under the hood, and do all the exercises!

The website http://sagemath.org/lists.html lists the two main Sage mailing list: sage-support and sage-devel. The first is for all questions about using and installing Sage, and for bug reports. The second is for discussion related directly to modifying and improving Sage. Join these lists; note that you will likely want to select *daily digest* for message deliver, so that you receive one email a day instead of 50!

If you find a bug in Sage, *we want to know about it!* Send an email to sage-support describing the bug in as much detail as you can.

The exact details for installing Sage will change over time. The following briefly describes the situation in April 2008; for more details see the Sage installation guide.

You install Sage on any computer either by extracting a pre-compiled binary or building from source.

To install a binary, download the binary (either a .tar.gz file, a .dmg file, or a .7z file) from http://sagemath.org, extract the file, and following the directions in the README.txt file. In most cases, you can simply put the extracted `sage-x.y.z` directory somewhere, and run Sage from there.

Installing Sage on Microsoft Windows currently involves installing the VMware player program, then running a virtual machine. This will likely change significantly within a year.

Currently you can only build Sage from source on Linux and OS X (or on Windows in a Linux virtual machine). To build from source, which will take over an hour, first download `sage-x.y.z.tar` from http://sagemath.org/download.html. Once you have downloaded Sage, extract the "tarball" as follows:

```
$ tar xvf sage-x.y.z.tar
```

Then run the `make` command in the `sage-x.y.z` directory:

```
$ cd sage-x.y.z
$ make
... 137600 lines of output ...
real    102m57.600s
user    71m10.115s
sys     14m56.116s
To install gap, gp, singular, etc., scripts
in a standard bin directory, start sage and
type something like
   sage: install_scripts('/usr/local/bin')
at the SAGE command prompt.

SAGE build/upgrade complete!
```

The above should work if you have gcc, g++, make, and a few other prerequisites installed (Sage includes almost all depedencies), and are using a supported architecture and operating system. An advantage of building from source is that if you can build from source, then you can change absolutely any part of Sage and use the modified version. If you run into trouble, email `sage-support`.

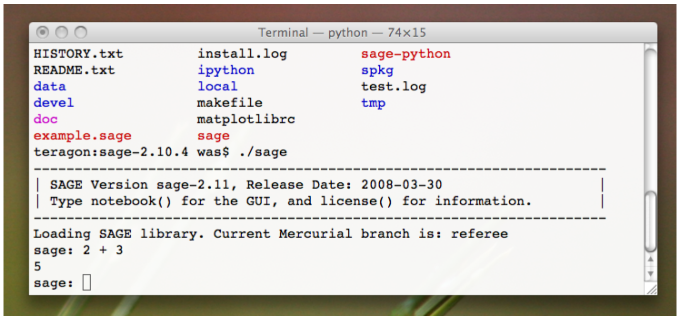
Exercise 1.1. Install Sage on a computer.

1.1.2 The command line and the notebook

The command line and the Sage notebook provide two complementary ways for you to interactively work with Sage. The command line provides a simple and powerful way to type Sage commands. The notebook provides a modern AJAX web-browser based graphical interface to Sage. Both the notebook and command line have advantages and disadvantages, and you will want to become familiar with each. For example, the notebook is better for graphics, whereas the command line is better for debugging and profiling code (see Section **??**).

Technical Note 1.2. The Sage command line is a customized version of the incredible IPython interactive shell by Fernando Perez et al. [PG07]. The Sage notebook is a Python/Javascript AJAX application built by the author and T. Boothby, T. Clemans, A. Clemesha, B. Moretti, Y. Qiang and D. Raymer; it uses Twisted [Twi], Pexpect [Pex], and jQuery [jQu].

In Linux or OS X, start the *command line* by typing `./sage` in the directory where you installed Sage, or just type `sage` if the Sage install directory `sage-x.y.z` is in your PATH. In Windows, after starting the Sage vmware image, type `sage` at the login prompt.



```
HISTORY.txt             install.log             sage-python
README.txt              ipython                 spkg
data                    local                   test.log
devel                   makefile                tmp
doc                     matplotlibrc
example.sage            sage
teragon:sage-2.10.4 was$ ./sage
----------------------------------------------------------------------
| SAGE Version sage-2.11, Release Date: 2008-03-30                   |
| Type notebook() for the GUI, and license() for information.        |
----------------------------------------------------------------------
Loading SAGE library. Current Mercurial branch is: referee
sage: 2 + 3
5
sage: 
```

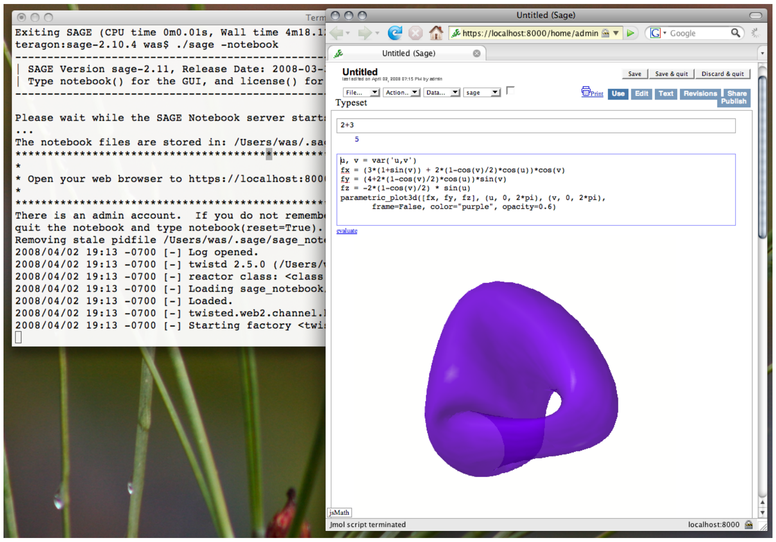
In Linux or OS X, start the *notebook* by typing `./sage -notebook` in the directory where you installed Sage:

```
teragon:sage-2.10.4 was$ ./sage -notebook
----------------------------------------------------------------------
| SAGE Version 2.10.4, Release Date: 2008-03-16                      |
| Type notebook() for the GUI, and license() for information.        |
----------------------------------------------------------------------

Please wait while the SAGE Notebook server starts...
...
```

If you are on a single-user system, you can alternatively type `./sage -inotebook`, which is less secure, but sometimes more convenient since it only uses http. In particular, any *other* user on your system could also connect to your notebook and delete your files – this isn't a problem if there are no other users logged into your system.

In Windows, type `notebook` at the login prompt, then use your web browser to navigate to the URL that is displayed. You can also use Sage without installing it on your computer by signing up for an account at https://sagenb.org.



```
Exiting SAGE (CPU time 0m0.01s, Wall time 4m18.1
teragon:sage-2.10.4 was$ ./sage -notebook
------------------------------------------------
| SAGE Version sage-2.11, Release Date: 2008-03-
| Type notebook() for the GUI, and license() for
------------------------------------------------

Please wait while the SAGE Notebook server start
...
The notebook files are stored in: /Users/was/.sa
************************************************
*
* Open your web browser to https://localhost:800
*
************************************************
There is an admin account.  If you do not rememb
quit the notebook and type notebook(reset=True).
Removing stale pidfile /Users/was/.sage/sage_not
2008/04/02 19:13 -0700 [-] Log opened.
2008/04/02 19:13 -0700 [-] twistd 2.5.0 (/Users/
2008/04/02 19:13 -0700 [-] reactor class: <class
2008/04/02 19:13 -0700 [-] Loading sage_notebook
2008/04/02 19:13 -0700 [-] Loaded.
2008/04/02 19:13 -0700 [-] twisted.web2.channel.
2008/04/02 19:13 -0700 [-] Starting factory <twi
```

Untitled

last edited on April 02, 2008 07:15 PM by admin

```
2+3
```

```
5
```

```
u, v = var('u,v')
fx = (3*(1+sin(v)) + 2*(1-cos(v)/2)*cos(u))*cos(v)
fy = (4+2*(1-cos(v)/2)*cos(u))*sin(v)
fz = -2*(1-cos(v)/2) * sin(u)
parametric_plot3d([fx, fy, fz], (u, 0, 2*pi), (v, 0, 2*pi),
        frame=False, color="purple", opacity=0.6)
```

Exercise 1.3.

The point of this exercise is to try out both the Sage command line *and* the Sage notebook.

1. Using the Sage command line, compute $123 + 456$.

2. Using the Sage notebook, compute $456 + 789$.

Tab completion and help are incredibly useful features of both the *command line* and *notebook*, and both work in almost the same way in both. This is useful in two ways. First, if you type the first few letters of a command, then press the tab key, you'll see all commands that begin with those first few letters.

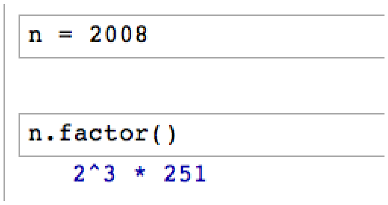
The second way in which tab completion is useful is that it shows you most of the things you can do with your object. For example, if n is an integer and you type `n.[tab key]`, you'll see a list of all the functions that you can call on n. For example, the code `n=2008` in Sage sets the *variable* n equal to the integer 2008:

```
    sage: n = 2008
```

Then type `n.fa[tab key]` (press the tab key after typing `n.fa`), you'll see that `n.factor` is a command associated to n. Type `n.factor()` to factor n:

```
sage: n = 2008
sage: n.factor()
2^3 * 251
```

Here we are showing all input and output as if they were typed at the command line. If you're using the notebook press shift-enter after typing $n = 2008$ into an input cell. After computing the factorization you will see something like this:



```
n = 2008
```

```
n.factor()
   2^3 * 251
```

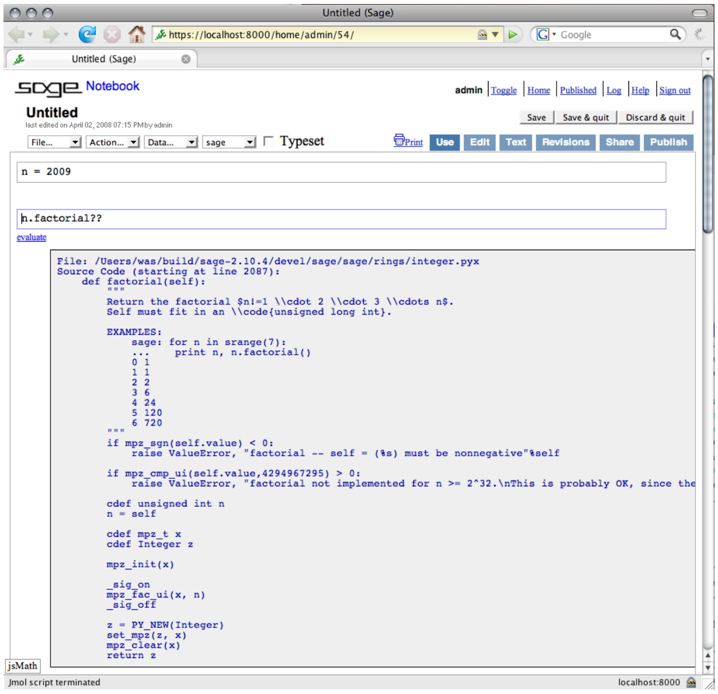
Exercise 1.4. Use tab completion to determine how to compute the *factorial* of 100.

In the *command line* every command you type is recorded in the history. Use the up arrow to scroll through previous commands; this history even works if you quit Sage and restart. Likewise, in the *notebook* previous commands are visibly recorded in input cells, and you can click or use the arrow keys to move to a previous cell and press shift-enter to evaluate it again.

Exercise 1.5.

1. Quit the Sage command line, restart Sage, and press the up arrow until you see `n = 2008`. Change 2008 to 2009 and press enter. Then factor the result, again using the up arrow to select `n.factor()`.

2. In the Sage notebook click and change `n = 2008` to `n = 2009`, then press shift-enter twice to see how 2009 factors.



```
n = 2009
```

```
n.factorial??
```

```
File: /Users/was/build/sage-2.10.4/devel/sage/sage/rings/integer.pyx
Source Code (starting at line 2087):
    def factorial(self):
        """
        Return the factorial $n!=1 \\cdot 2 \\cdot 3 \\cdots n$.
        Self must fit in an \\code{unsigned long int}.

        EXAMPLES:
            sage: for n in srange(7):
            ...       print n, n.factorial()
            0 1
            1 1
            2 2
            3 6
            4 24
            5 120
            6 720
        """
        if mpz_sgn(self.value) < 0:
            raise ValueError, "factorial -- self = (%s) must be nonnegative"%self

        if mpz_cmp_ui(self.value,4294967295) > 0:
            raise ValueError, "factorial not implemented for n >= 2^32.\nThis is probably OK, since the

        cdef unsigned int n
        n = self

        cdef mpz_t x
        cdef Integer z

        mpz_init(x)

        _sig_on
        mpz_fac_ui(x, n)
        _sig_off

        z = PY_NEW(Integer)
        set_mpz(z, x)
        mpz_clear(x)
        return z
```

If `n` is any object in Sage (even the function `n.factorial`), type `?` after it and press enter to see a description of the command along with examples; this works the same on the command line and in the notebook. In the notebook, you can also type `n.factorial(` and press the tab key for popup help (yes, that is an *open parenthesis* instead of a `?`, as if you are about to call the function). If you put two question marks instead of one you'll see the help *and source code* of the function or object, i.e., the computer code that defines that function or object.

There are several ways to time how long it takes for something in Sage to run. If the command is just one line, put the word `time` at the beginning of the line, e.g.,

```
sage: time n = factorial(10^5)
CPU time: 0.10 s,  Wall time: 0.10 s
```

Exercise 1.6. Make up a line of input to Sage that takes at least ten seconds to evaluate.

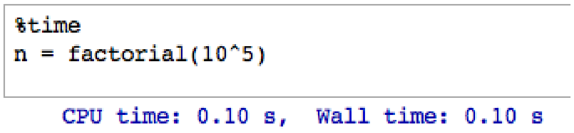
You can time execution of all the code in a *notebook* cell by putting `%time` at the beginning of the cell. For example:

```
{{{
%time
n = factorial(10^5)
///
CPU time: 0.10 s,  Wall time: 0.10 s
}}}
```

Above we have used the following notation:

```
{{{
INPUT
///
OUTPUT
}}}
```

Thus the above looks like the following in the notebook:



```
%time
n = factorial(10^5)
```

```
CPU time: 0.10 s,  Wall time: 0.10 s
```

You can also time execution of a block of code by typing `t = cputime()` before the block, then after the block typing `cputime(t)`. This will output the number of CPU seconds that elapsed during the computation. For the physical amount of time that actually elapsed on your "wall" clock, type instead `t = walltime()` and `walltime(t)`. The increased flexibility of using `cputime` and `walltime` can be useful, especially when profiling complicated programs.

Sage consists of dozens of components, and sometimes calls out to external programs. The `cputime` command only records the CPU time spent by the master Sage process, and ignores all time spent by programs that Sage calls out to. Thus to get a good sense of how long some Sage code really takes to run, it is best to use `walltime` and take the minimum over multiple runs. However, another issue to watch out for is that some Sage commands remember values they have computed, so all calls after the first one may be much faster, which will completely throw of timings.

Once you start writing complicated Sage programs, especially when using the command line, you'll want to place code in an external file and edit it with a standard code editor (use the special Python mode if your editor has one).

This works very well in Linux and OS X, where you put the code in the file of your choice and type

```
sage: load filename.sage
```

to execute all the code in `filename.sage`. Under Windows, the situation is currently more complicated – you either have to configure VMware shared folders, or regularly upload the file to the Sage notebook using `Data --> Upload or Create File`. Another separate option[1] is to use the Windows program WinSCP [**?**]. Using WinSCP you can login to the VMware machine (use login name 'login' and password 'sage'). Then you can select edit from the context menu and edit files. WinSCP takes care of automatically uploading and downloading the modified version of the file whenever you change it. If you call the file `filename.sage`, you would type the following to load the file you're editing:

```
sage: load /home/login/filename.sage
```

For OS X or Linux users, if you're constantly editing `filename.sage`, and find yourself regularly typing `load filename.sage` into Sage, you should instead *attach* `filename.sage` by typing

```
sage: attach filename.sage
```

This works exactly like `load filename.sage`, except that if `filename.sage` is changed and you execute a new command, Sage reloads `filename.sage` before executing the command. Try it; you'll like it.

Exercise 1.7. Create a file `hi.sage` that contains the line `print "hello"`. Load that file into Sage using the load command, and see that hello is printed out. If you're using OS X or Linux, attach `hi.sage`, then change "hello" to something else, then type something at the Sage prompt – you should see something besides "hello" printed out.

1.1.3 Loading and saving variables and sessions

As mentioned above, in Sage you create a new variable by assigning to it. For example, typing `n = 2008` creates a new variable n and assigns the value 2008 to it. Most (but not all![2]) individual variables, even incredibly complicated ones, can be easily saved or loaded to disk using the save and load functions. This can be extremely useful if it takes a long time to compute a variable, and you want to save it to use it again later.

First we discuss using save and load from the command line, then from the notebook. In the following *command-line session*, we set n to 2008, then save n

[1] suggested by Lars Fischer; the author has not yet tried this!!

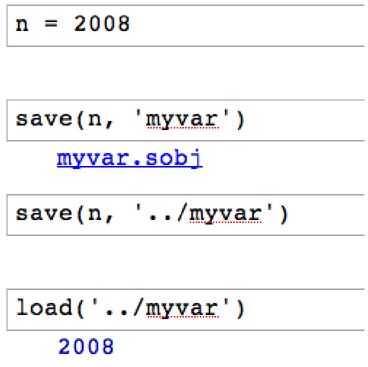
[2] See the Python documentation on the `pickle` module.

to the variable `myvar.sobj`, quit sage, restart, and reload the value of n from that file.

```
teragon:~ was$ sage
sage: n = 2008
sage: save(n, 'myvar')
sage: quit
Exiting SAGE (CPU time 0m0.02s, Wall time 0m41.28s).
teragon:~ was$ ls -l myvar.sobj
-rw-r--r--  1 was  was  48 Mar 26 22:40 myvar.sobj
teragon:~ was$ sage
sage: load('myvar.sobj')
2008
```

Note that `.sobj` is added to the end of the filename if it isn't given. Also, all saved objects are compressed to save disk space.

Using save and load *from the notebook* is a bit different, since every notebook cell is executed in a different subdirectory. In the following example, in the first cell the value of n is saved in the file `myvar.sobj` in the current cell directory. In general, in the notebook every file created in the current (cell) directory is either displayed embedded in the output or a link to it is created – you could right click and download `myvar.sobj` to your hard drive if you wanted. In the second example we save the value of n to the file `myvar.sobj` in the directory above the current cell directory. This directory is fixed independent of the cell we're working in, so we can load `myvar.sobj` in another cell, which we do.



```
n = 2008

save(n, 'myvar')
    myvar.sobj
save(n, '../myvar')

load('../myvar')
    2008
```

Not only can you load and save most objects, but you can also save all the objects defined in the current session. WARNING: *Some objects, e.g., references to files, can't be saved at all, and they won't be saved when you save a session.* In this example we define two variables $n = 2008$ and $m = -2/3$, save the session, then restart, load the session and see that n and m are again defined. p

```
teragon:~ was$ sage
sage: n = 2008
sage: m = -2/3
sage: save_session(’session’)
sage: quit
Exiting SAGE (CPU time 0m0.08s, Wall time 9m11.00s).
teragon:~ was$ sage
losage: load_session(’session’)
sage: n
2008
sage: m
-2/3
```

Think of `save_session` as being sort of like “saving your game” in a video game.

Exercise 1.8. (Command line only.) A useful feature of session loading and saving is that you can save two separate sessions and load them at the same time. Instead of the second loaded session overwriting the first completely, the two are merged. Try this out – define several variables in two separate sessions, save both sessions. Quit, load one session, then load the other and see that the sessions are merged.

Behind the scenes session loading and saving works essentially by calling save on every currently defined variable – the objects that can’t be saved just don’t get saved.

1.2 Python: the Language of Sage

Python has excellent support for basic data structures, including lists, tuples, strings, dictionaries, sets, and user-defined classes. Functions, control flow, and error handling are also all well supported. In this section we just give a very brief overview with exercises of the Python language, which should be enough to get you going. There are many excellent books about the basics of Python; my favorites are the official Python Tutorial, Dive Into Python, and Python in a Nutshell, and I strongly encourage you to look at them. In fact, the examples below very closely follow those in the Python Tutorial.

1.2.1 Lists, Tuples, Strings, Dictionaries and Sets

The five basic data structures you will use constantly in Python programming are lists, tuples, strings, dictionaries, and sets.

Lists

Lists and tuples are the two main sequence types, and both allow you to store and work with arbitrary finite sequences of Python objects. Both are indexed *starting at 0*. The main difference between lists and tuples is that lists are mutable, whereas tuples are immutable; this means that you can easily change lists, which has some subtle pros and cons as we will see below.

You can create a list by explicitly listing its elements, by adding, multiplying, appending to and “slicing” existing lists:

```
sage: a = [’spam’, ’eggs’, 100, 1234]; a
[’spam’, ’eggs’, 100, 1234]
sage: a[0]
’spam’
sage: a[3]
1234
sage: a[-2]
100
sage: a[1:-1]                          # list slicing
[’eggs’, 100]
sage: a[:2] + [’bacon’, 2*2]
[’spam’, ’eggs’, ’bacon’, 4]
sage: 3*a[:3] + [’Boo!’]
[’spam’, ’eggs’, 100, ’spam’, ’eggs’, 100, ’spam’, ’eggs’, 100, ’Boo!’]
sage: a.append(314); a
[’spam’, ’eggs’, 100, 1234, 314]
```

Absolutely anything can go in a list, and you can change the entries:

```
sage: b = [[1,2], 1.234, 3/4, vector([1/3,2,3]), [[1]]]; b
[[1, 2], 1.23400000000000, 3/4, (1/3, 2, 3), [[1]]]
```

Assignment to lists is also quite flexible. For example, if you use a negative index the index counts down from the right. You can also use the slicing notation, e.g., `a[1:3]`, to change whole sections of a list at once.

```
sage: b[0] = 5; b
[5, 1.23400000000000, 3/4, (1/3, 2, 3), [[1]]]
sage: b[-1] = 'hello'; b
[5, 1.23400000000000, 3/4, (1/3, 2, 3), 'hello']
sage: b[:2]; b
[5, 1.23400000000000]
sage: b[:2] = [1,2,3,4]; b
[1, 2, 3, 4, 3/4, (1/3, 2, 3), 'hello']
sage: b[1:4] = []; b
[5, 'hello']
sage: len(b)        # the length of b
2
```

There are many list methods such as append, remove, sort, index, etc. Make a list `v` then type `v.[tab key]` to see a list of those methods.

You can also use the beautiful and powerful *list comprehension* construction to make new lists from old lists. This is similar to "set builder notation" in mathematics:

```
sage: vec = [2, 4, 6]
sage: [3*x for x in vec]
[6, 12, 18]
sage: [3*x for x in vec if x > 3]
[12, 18]
sage: [3*x for x in vec if x < 2]
[]
sage: [[x,x^2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
sage: vec1 = [2, 4, 6]
sage: vec2 = [4, 3, -9]
sage: [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
sage: [x+y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
sage: [vec1[i]*vec2[i] for i in range(len(vec1))]
[8, 12, -54]
```

Finally use the `del` statement to delete an element of a list or to delete a whole chunk of a list:

```
sage: a = [-1, 1, 66.25, 333, 333, 1234.5]; a
[-1, 1, 66.2500000000000, 333, 333, 1234.50000000000]
sage: del a[0]
sage: a
[1, 66.2500000000000, 333, 333, 1234.50000000000]
sage: del a[2:4]
sage: a
[1, 66.2500000000000, 1234.50000000000]
sage: del a[:]
sage: a
[]
```

Note in particular the command `del a[2:4]` which deletes the entries of a in positions 2 and 3. These are exactly the entries you see if you type `a[2:4]`.

At this point a *major warning* is in order. In Python, variables store a reference to an object. Thus we have the following:

```
sage: v = [1]
sage: w = [v,v,v]
sage: w
[[1], [1], [1]]
sage: v.append(-4)
sage: w
[[1, -4], [1, -4], [1, -4]]
```

Whoa!? What just happened there? We first made a list v with the single element 1. Then we made a list w with three *references* to that first list v. Finally, we appended -4 to v, which changed v to a list with two elements $1, -4$. Finally, when we print w we see this new 2-element v printed 3 times. OK? The entries of w above are all the same list. We confirm this using the Python `is` operator, which returns `True` only when two variables reference exactly the same object:

```
sage: w[0] is w[1]
True
sage: w[1] is w[2]
True
```

To make each entry of the list w distinct, make a list of copies of v:

```
sage: v = [1]
sage: w = [copy(v) for _ in range(3)]; w
[[1], [1], [1]]
sage: w[0].append(-4)
sage: w
[[1, -4], [1], [1]]
```

Tuples

As mentioned above, tuples are sequence types like lists, except the objects in the tuple can't be deleted or replaced once the tuple has been created.

```
sage: t = (12345, 54321, 'hello!')
sage: t[0]
12345
sage: t
(12345, 54321, 'hello!')
sage: type(t)
<type 'tuple'>
sage: t[0] = 5   # tuples are immutable
Traceback (most recent call last):
...
TypeError: 'tuple' object does not support item assignment
sage: # Tuples may be nested:
sage: u = (t, (1, 2, 3, 4, 5))
sage: u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

There is also an eloquent tuple unpacking system, which makes it simple to assign several variables to the entries of a tuple.

```
sage: t = (12345, 54321, 'hello!'); t
(12345, 54321, 'hello!')
sage: # tuple UNPACKING
sage: x, y, z = t
sage: print x, y, z
12345 54321 hello!
```

Tuple unpacking also supports having functions to have multiple return variables, without running into the numerous painful gotcha's that other languages (e.g., Matlab and Magma) suffer from having multiple return variables.

```
sage: def foo(a,b):
...       return a+b, a-b
sage: c,d = foo(2,3)    # multiple return values!
sage: print c, d
5 -1
```

Strings

Python can manipulate strings, and uses similar slice notation to lists and tuples. In fact, a string behaves very much like a tuple of characters.

In the following examples we illustrate some of the ways you can construct a string. Notice that you can use single or double quotes, that you can use a backslash to include arbitrary quotes in a string, and that strings can span multiple lines.

```
sage: 'sage math'
'sage math'
sage: 'doesn\'t'
'doesn't'
sage: "doesn't"
"doesn't"
sage: '"Yes," he said.'
'"Yes," he said.'
sage: "\"Yes,\" he said."   # NOTE: this is broken in sage-2.11
'"Yes," he said.'
```

Use triple quotes to create a string the spans several lines:

```
sage: s = """
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
"""
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
```

You can index, slice and add strings in exactly the same way as you do with lists and tuples:

```
sage: word = "Sage"
sage: word + word
’SageSage’
sage: word*5
’SageSageSageSageSage’
sage: len(word*5)
20
sage: word[1]
’a’
sage: word[-1]
’e’
sage: word[0:2]
’Sa’
sage: word[2:]
’ge’
```

Dictionaries

A *Python dictionary* is an unordered set of `key:value` pairs, where the keys are unique. A pair of braces {} creates an empty dictionary; placing a comma-separated list of `key:value` pairs initializes the dictionary.

The following examples illustrate how to create a dictionary, get access to entries, get a list of the keys and values, etc.

```
sage: d = {’sage’:’math’, 1:[1,2,3]}; d
{1: [1, 2, 3], ’sage’: ’math’}
sage: d[’sage’]
’math’
sage: d[1]
[1, 2, 3]
sage: d.keys()
[1, ’sage’]
sage: d.values()
[[1, 2, 3], ’math’]
sage: d.has_key(’sage’)
True
sage: ’sage’ in d
True
```

You can delete entries from the dictionary using the `del` keyword.

```
sage: del d[1]
sage: d
{’sage’: ’math’}
```

You can also create a dictionary by typing `dict(v)` where `v` is a list of pairs:

```
sage: dict( [(1, [1,2,3]), (’sage’, ’math’)])
{1: [1, 2, 3], ’sage’: ’math’}
sage: dict( [(x, x^2) for x in [1..5]] )
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

You can also make a dictionary from a “generator expression” (we have not discussed these yet).

```
sage: dict( (x, x^2) for x in [1..5] )
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Dictionary keys must be hashable

At this point another *major warning* is in order. The keys `k` of a dictionary must be *hashable*, which means that calling `hash(k)` doesn’t result in an error. Some Python objects are hashable and some are not. Usually objects that can’t be changed are hashable, whereas objects that can be changed are not hashable, since the hash of the object would change, which would totally devastate most algorithms that use hashes. In particular, numbers and strings are hashable, as are tuples of hashable objects, but lists are never hashable.

We hash the string ’sage’, which works since one cannot change strings.

```
sage: hash(’sage’)
-596024308
```

The list `v = [1,2]` is not hashable, since `v` can be changed by deleting, appending, or modifying an entry. Because `[1,2]` is not hashable it can’t be used as a key for a dictionary.

```
sage: hash([1,2])
Traceback (most recent call last):
...
TypeError: list objects are unhashable
sage: d = {[1,2]: 5}
Traceback (most recent call last):
...
TypeError: list objects are unhashable
```

However the tuple `(1,2)` is hashable and can hence be used as a dictionary key.

```
sage: hash( (1,2) )
1299869600
sage: d = {(1,2): 5}
```

Sets

Python has a set datatype, which behaves much like the keys of a dictionary. A *set* is an unordered collection of unique hashable (see Section 1.2.1) objects. Sets are incredibly useful when you want to quickly eliminate duplicates, do set theoretic operations (union, intersection, etc.), and tell whether or not an objects belongs to some collection.

You create sets *from the other Python data structures* such as lists, tuples, and strings. For example:

```
sage: set( (1,2,1,5,1,1) )
set([1, 2, 5])
sage: a = set(’abracadabra’); b = set(’alacazam’)
sage: print a
set([’a’, ’r’, ’b’, ’c’, ’d’])
sage: print b
set([’a’, ’c’, ’z’, ’m’, ’l’])
```

There are also many handy operations on sets.

```
sage: a - b    # letters in a but not in b
set(['r', 'b', 'd'])
sage: a | b    # letters in either a or b
set(['a', 'c', 'b', 'd', 'm', 'l', 'r', 'z'])
sage: a & b    # letters in both a and b
set(['a', 'c'])
```

If you have a big list `v` and want to repeatedly check whether various elements `x` are in `v`, you *could* write `x in v`. This would work. Unfortunately, it would be really slow, since every command `x in v` requires *linearly* searching through for x. A much better option is to create `w = set(v)` and type `x in w`, which is very fast.

```
sage: v = range(10^6)
sage: time 10^5 in v
True
CPU time: 0.16 s,  Wall time: 0.18 s
sage: time w = set(v)
CPU time: 0.12 s,  Wall time: 0.12 s
sage: time 10^5 in w
True
CPU time: 0.00 s,  Wall time: 0.00 s
```

1.2.2 Control Flow

[[This section is not finished.]]

1.2.3 Errors Handling

[[This section is not finished.]]

- Loops, functions, control statements; putting code in files and modules (import/load/attach)

```
math 480 -- april 7, 2008
system:sage

<font color="purple">
<h1> Errors and Exceptions: Recovering from problems gracefully </h1>
</font>

<img src="http://www.abc.net.au/reslib/200705/r146804_516761.jpg" width=200

<h2>Reference: <a target="_new" href="http://docs.python.org/tut/node10.htm

<h3>Syntax Errors</h3>

{{{
ljsad 9as oha ivhxzv @#!#^!#QASDkias fiaosdj f
///
Syntax Error:
    ljsad 9as oha ivhxzv @#!#^!#QASDkias fiaosdj f
}}}

{{{
while True print ’Hello world’
///
Syntax Error:
    while True print ’Hello world’
}}}

<h3>Exceptions</h3>

Python has excellent very fast support for exception handling, which makes

Here is an example. The <i>last line</i> of the error message indicates wha

{{{
10 * (1/0)
///

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/was/.sage/sage_notebook/worksheets/admin/61/code/302.py", li
    exec compile(ur’Integer(10) * (Integer(1)/Integer(0))’ + ’\n’, ’’, ’sin
  File "/Users/was/build/sage-3.0.alpha1/local/lib/python2.5/site-packages/

  File "element.pyx", line 1480, in sage.structure.element.RingElement.__di
  File "coerce.pxi", line 138, in sage.structure.element._div_c
  File "integer.pyx", line 1095, in sage.rings.integer.Integer._div_c_impl
  File "integer_ring.pyx", line 204, in sage.rings.integer_ring.IntegerRing_
ZeroDivisionError: Rational division by zero
}}}

This page <a href="http://docs.python.org/lib/module-exceptions.html">http:
<ol>
<li> TypeError
<li> ZeroDivisionError
<li> ArithmeticError
```

1.2.4 Classes

[[This section is not finished.]]

Class corresponding to the mathematical objects you are working with, e.g., a Matrix class for matrices, a DifferentialEquations class for differential equations, etc. This works very very nicely for expressing mathematics, and is much different and conceptually superior to what you get in Mathematica and Matlab. Magma has classes, but users can't define their own.

The Python class construction allows you to define your own new data types. It is modeled on C++ classes, though Python classes are simpler and easier to use. They support both single and multiple inheritance and one can derive from builtin classes.

Reference: Chapter 9 of the Python Tutorial

You can define any new class you want very easily at any point, even inside the body of a function, etc. It's very nice. Here are some examples.

```
{{{
class NaturalNumber:
    pass
///
}}}

{{{
# A class itself is a Python object, just like anything else

print NaturalNumber
///

__main__.NaturalNumber
}}}

{{{
type(NaturalNumber)
///

<type ’classobj’>
}}}

{{{
v = [NaturalNumber, int, Integer]; v
///

[<class __main__.NaturalNumber at 0x836fd50>, <type ’int’>, <type ’sage.rin
}}}

{{{
# This is how to make instances of a class
n = NaturalNumber()
///
}}}

{{{
n
///

<__main__.NaturalNumber instance at 0x82f98c8>
}}}

{{{
type(n)
///

<type ’instance’>
}}}

{{{
# The above class is very boring.  Let’s add printing capabilities and a va
class NaturalNumber:
    def __init__(self, n):
        if not is_Integer(n):
            raise TypeError
```

1.3 Cython: Compiled Python

1.3.1 The Cython project

1.3.2 How to use Cython: command line, notebook

when to use; when good / when bad

1.4 Optimizing Sage Code

Once your code is working correctly (see Section 1.5 below) *and* you have lots of examples (hopefully formated as doctests) that illustrate this, you should next search for serious inefficiencies in your code, which could render your code nearly useless even if it works. These will typically result from a bad choice of algorithms, using a function in Sage that itself hasn't been optimized (but should someday be!), or using a non-optimal data type or data structure that works but does so much more slowly than a different choice. In Sections 1.4.1 and 1.4.2 ways to track down such issues.

We present this section before Section 1.5 since many of the fundamental techniques and ideas are similar but more fun. Also, code that is too slow to be useful might as well be considered to be buggy.

1.4.1 Example: Optimizing a Function Using A Range of Manual Techniques

In the following example we implement a naive function to compute the last d digits of a power n^e of an integer n. We then time various parts of the implementation, and replace bits of code by better code, eventually obtaining a speedup by a factor of $20,000$!

```
{{{
def last_digits1(n, e, d):
    """
    Return the last d digits of n^e.
    NOTE: First naive version.
    """
    return str(n^e)[-d:]
}}}
```

```
{{{
time last_digits1(3,10^6,4)
///
'0001'
CPU time: 0.43 s,  Wall time: 0.43 s
}}}
```

Next we rewrite the above function to break up the one lineer into separate statements, which makes timing each of them easier.

```
{{{
def last_digits2(n, e, d):
    time m = n^e
    time s = str(m)
    time t = s[-d:]
    return t
}}}
```

Now we rerun the rewritten function. The three timing outputs correspond in order to the time commands above:

```
{{{
time last_digits2(3,10^6,4)
///
Time: CPU 0.04 s, Wall: 0.04 s
Time: CPU 0.39 s, Wall: 0.39 s
Time: CPU 0.00 s, Wall: 0.00 s
'0001'
CPU time: 0.43 s,  Wall time: 0.43 s
}}}
```

From the above, we see that the string conversion takes by far the most time. The following version of the code avoids that string conversion:

```
{{{
def last_digits3(n, e, d):
    time m = n^e
    time s = m%(10^d)
    time t = str(s)
    time t = '0'*(d-len(t)) + t
    return t
}}}
```

```
{{{
time last_digits3(3,10^6,4)
///
Time: CPU 0.04 s, Wall: 0.04 s
Time: CPU 0.00 s, Wall: 0.00 s
Time: CPU 0.00 s, Wall: 0.00 s
Time: CPU 0.00 s, Wall: 0.00 s
’0001’
CPU time: 0.04 s,  Wall time: 0.04 s
}}}
```

The code runs so quickly now that we switch to a bigger benchmark.

```
{{{
time last_digits3(3,10^7,4)
///
Time: CPU 0.53 s, Wall: 0.54 s
Time: CPU 0.01 s, Wall: 0.01 s
Time: CPU 0.00 s, Wall: 0.00 s
Time: CPU 0.00 s, Wall: 0.00 s
’0001’
CPU time: 0.53 s,  Wall time: 0.54 s
}}}
```

From the above we see that the initial powering operation $m = n^e$ takes most of the time (“it dominates”). We rewrite the function below to use a data type (and consequently an algorithm) that speeds up the powering computation.

```
{{{
def last_digits4(n, e, d):
    time m = Integers(10^d)(n)
    time s = m ^ e
    time t = str(s)
    time t = ’0’*(d-len(t)) + t
    return t
}}}
```

Now we time it:

```
{{{
time last_digits4(3,10^7,4)
///
Time: CPU 0.00 s, Wall: 0.00 s
Time: CPU 0.00 s, Wall: 0.00 s
Time: CPU 0.00 s, Wall: 0.00 s
Time: CPU 0.00 s, Wall: 0.00 s
'0001'
CPU time: 0.00 s,  Wall time: 0.00 s
}}}
```

Victory!? No. At this point maybe we just need a bigger example.

```
{{{
time last_digits4(389,10^50,10)
///
Time: CPU 0.00 s, Wall: 0.00 s
Time: CPU 0.00 s, Wall: 0.00 s
Time: CPU 0.00 s, Wall: 0.00 s
Time: CPU 0.00 s, Wall: 0.00 s
'0000000001'
CPU time: 0.00 s,  Wall time: 0.00 s
}}}
```

The above timing information is not very useful. Instead we next time the computation many times using the `timeit` function. For this, we better get rid of the print statements, which will throw off the timing.

```
{{{
def last_digits5(n, e, d):
    m = Integers(10^d)(n)
    s = m ^ e
    t = str(s)
    t = '0'*(d-len(t)) + t
    return t
}}}
```

Using `timeit` we get the result of running the above function on given input hundreds of times, and taking the best wall time.

```
{{{
a = 389; b = 10^50; d = 10
timeit('last_digits5(a,b,d)')          # input must be in quotes
///
625 loops, best of 3: 69.7 s per loop
}}}
```

That seems really really fast. Can we hope to do much better? We try switching to using Cython so that our code gets compiled to C, and the C optimizer might be able to speed it up. **WARNING:** There is (currently) no preparsing in Cython or `%cython` blocks, so it is critical to use `**` instead of ^ below!

```
{{{
%cython
def last_digits6(n, e, d):
    m = Integers(10**d)(n)
    s = m ** e
    t = str(s)
    t = '0'*(d-len(t)) + t
    return t
}}}
```

Now we time this:

```
{{{
time last_digits6(389,10^50,10)
///
'0000000001'
CPU time: 0.01 s,  Wall time: 0.08 s
}}}
{{{
a = 389; b = 10^50; d = 10
timeit('last_digits6(a,b,d)')
///
625 loops, best of 3: 66.4 s per loop
}}}
```

Going from 69.7 to 66.4 micro seconds isn't much of a speed up, and may just be timing variation. Now imagine that we are getting desperate. Let's say we really *really* care about the speed of this particular function. We are willing to spend hours (or even *days*) looking at manuals and source code. We are willing to put up with the possibility of segfaults, etc. We are willing to learn a little about how to use the GMP C library (http://gmplib.org/). We are also willing to sacrifice readability.

By reading the source code for each of the underlying powering operations used in last_digits6, which we find in the files

```
    SAGE_ROOT/devel/sage/sage/ring/integer.pyx
    SAGE_ROOT/devel/sage/sage/ring/integer.pyx
```

along with some reading of the GMP manual, we translate last_digits6 almost line-for-line into the following code. The key difference is that we avoid the overhead of creating Sage (or Python) integers when possible. This overhead matters at this level of optimization.

```
{{{
%cython
from sage.rings.integer cimport Integer
def last_digits7(Integer n, Integer e, unsigned int d):
    cdef mpz_t ten, tenpow
    mpz_init_set_ui(ten, 10)      # ten = 10
    mpz_init(tenpow)
    mpz_pow_ui(tenpow, ten, d)    # tenpow = 10^d
    cdef Integer s = Integer()
    mpz_powm(s.value, n.value, e.value, tenpow)   # m = n^d % tenpow
    mpz_clear(ten); mpz_clear(tenpow)   # avoid memory leaks!
    t = str(s)
    t = ’0’*(d-len(t)) + t
    return t
}}}
```

Does it work? Yes.

```
{{{
last_digits7(389,10^50,10)
///
’0000000001’
}}}
```

How fast is it?

```
{{{
a = 389; b = 10^50; d = 10
timeit(’last_digits7(a,b,d)’)
///
625 loops, best of 3: 30.4 s per loop
}}}
```

Wow, it’s over twice as fast as `last_digits6`! What should we optimize next? The pure Python lines

```
    t = str(s)
    t = ’0’*(d-len(t)) + t
```

look like really good candidates. Are they? To find out we simply skip those two lines to obtain a *lower bound* on how much time we can possibly save by optimizing them further. Note that the resulting function `last_digits8` does not output the right thing; timing it just gives a useful lower bound.

```
{{{
%cython
from sage.rings.integer cimport Integer
def last_digits8(Integer n, Integer e, unsigned int d):
    cdef mpz_t ten, tenpow
    mpz_init_set_ui(ten, 10)      # ten = 10
    mpz_init(tenpow)
    mpz_pow_ui(tenpow, ten, d)    # tenpow = 10^d
    cdef Integer s = Integer()
    mpz_powm(s.value, n.value, e.value, tenpow)  # m = n^d % tenpow
    mpz_clear(ten); mpz_clear(tenpow)  # avoid memory leaks!
    return s
}}}
{{{
a = 389; b = 10^50; d = 10
timeit(’last_digits8(a,b,d)’)
///
625 loops, best of 3: 25 s per loop
}}}
```

Timing this function we see that the time is *not* dominated by the string stuff at the end, since we only get a very modest speedup. Is the time dominated by the Python function call overhead?

```
{{{
%cython
from sage.rings.integer cimport Integer
def last_digits9(Integer n, Integer e, unsigned int d):
    return n
}}}
{{{
a = 389; b = 10^50; d = 10
timeit('last_digits9(a,b,d)')
///
625 loops, best of 3: 306 ns per loop
}}}
```

No, just the function call overhead can't possibly be at fault. Is it the `Integer` creation code? We test this possibility by replacing the `Integer s` by a GMP `mpz_t` called `s`.

```
{{{
%cython
from sage.rings.integer cimport Integer
def last_digits10(Integer n, Integer e, unsigned int d):
    cdef mpz_t ten, tenpow
    mpz_init_set_ui(ten, 10)     # ten = 10
    mpz_init(tenpow)
    mpz_pow_ui(tenpow, ten, d)   # tenpow = 10^d
    cdef mpz_t s
    mpz_init(s)
    mpz_powm(s, n.value, e.value, tenpow)  # m = n^d % tenpow
    return 0
}}}

{{{
a = 389; b = 10^50; d = 10
timeit('last_digits10(a,b,d)')
///
625 loops, best of 3: 24.9 s per loop
}}}
```

This has almost no effect on the timing. The last obvious thing to try is to see if removing the `mpz_powm` function makes a difference.

```
{{{
%cython
from sage.rings.integer cimport Integer
def last_digits11(Integer n, Integer e, unsigned int d):
    """
    Return the last d digits of n^e.
    """
    cdef mpz_t ten, tenpow
    mpz_init_set_ui(ten, 10)      # ten = 10
    mpz_init(tenpow)
    mpz_pow_ui(tenpow, ten, d)    # tenpow = 10^d
    cdef mpz_t s
    mpz_init(s)
    #mpz_powm(s, n.value, e.value, tenpow)  # m = n^d % tenpow
    return 0
}}}
```

```
{{{/
a = 389; b = 10^50; d = 10
timeit(’last_digits11(a,b,d)’)
//
625 loops, best of 3: 843 ns per loop
}}}
```

Thus when we remove the `mpz_powm` statement above the timing goes down to almost nothing, so *that one single call* to the GMP C Library dominates the runtime. Thus our function is probably reasonably optimized at this point (though of course one could do a little better with more work).

How did we do?

```
{{{
a = 389; b = 10^5; d = 10
timeit(’last_digits7(a,b,d)’)
///
625 loops, best of 3: 8.69 s per loop
}}}
```

```
{{{
a = 389; b = 10^5; d = 10
timneit(’last_digits1(a,b,d)’)
///
5 loops, best of 3: 183 ms per loop
}}}
```

The speedup from the first version to the last version is thus by a factor of over 20,000:

```
{{{
183/(8.69/1000)
///
21058.6881472957
}}}
```

The main observations to take away from the above example are that the first nice snippet of code you think of to solve a problem may be pretty but *very slow*, and to make code fast you should systematically and *logically reason* about what could possibly slow the code down then speed up what is slowing the code down the most, if possible.

1.4.2 Optimizing Using the Python `profile` Module

You probably got the feeling in Section 1.4.1 that some of what we were doing there regarding timing could be automated. In fact, Python includes a builtin profiler, which given an arbitrary expression or statement will run it and create a report about how much time was spent in each Python function, and how many times that function was called. The advantage of using profile is that it nicely automates much of what we did at the beginning above; the disadvantage for us is that it's not so useful with Cython code. In constrast, using print statements and `cputime` combined with good strategy works in the same way in Python and Cython or whatever else (but can be much more tedious and biased).

Let's try Python's profile module out on the first one-line version of the last digits function. Recall that the code we will profile is the following:

```
{{{
def last_digits1(n, e, d):
    return str(n^e)[-d:]
}}}
```

Now we import and run the profiler:

```
{{{
import profile
profile.run(preparse('last_digits1(3,10^6,4)'))
///
         4 function calls in 0.426 CPU seconds
   Ordered by: standard name
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.426    0.426    0.426    0.426 174.py:6(last_digits1)
        1    0.000    0.000    0.000    0.000 :0(setprofile)
        1    0.000    0.000    0.426    0.426 <string>:1(<module>)
        1    0.000    0.000    0.426    0.426 profile:0(last_digits1(Intege
        0    0.000             0.000          profile:0(profiler)
}}}
```

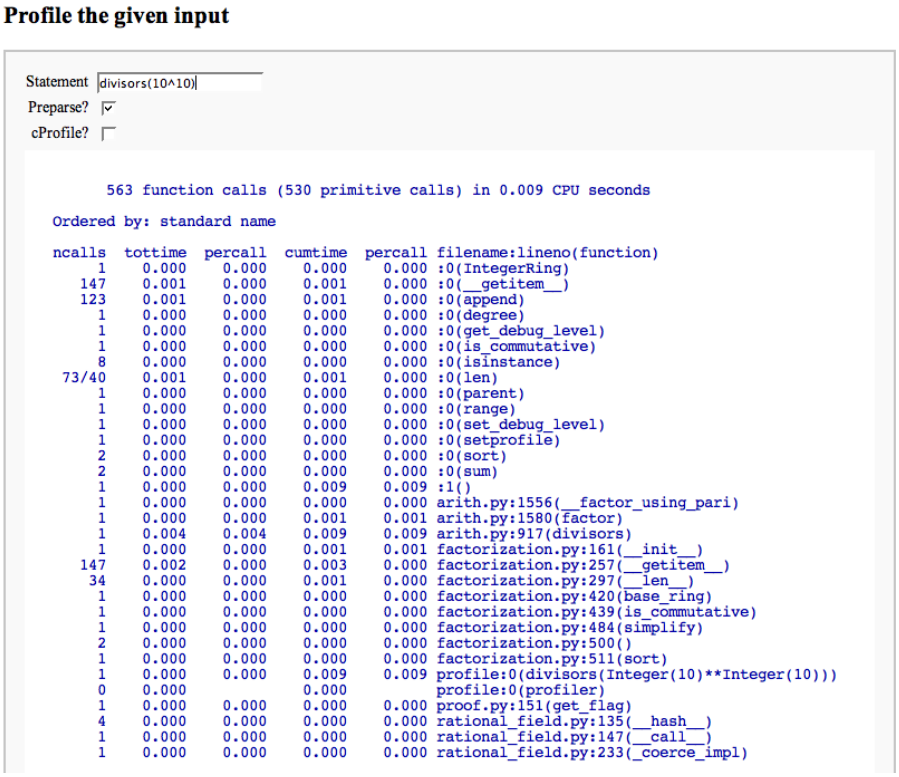
Notice above that we import the profile module then call the function `profile.run` on the preparsed version of the input line. If we didn't preparse the input line, then `10^6` would be interpreted as pure Python interprets it, namely as exclusive or of the Python ints 10 and 6. Also, the inputs to `last_digits1` would all be Python `int`s instead of Sage `Integer`s.

Type `profile.run?` for more help on how the profiler works. For far more information, see the section *The Python Profiler* in the Python Library Reference (see http://docs.python.org/lib/profile.html).

Note that profiling code can take a lot longer than running the code not under profile. There is also a module `cProfile` that is supposed to work exactly like `profile`, but is much faster. However, `cProfile` is new and not so well tested.

The following Sage interact allows you to profile a line of code easily in the Sage notebook.

```
html('<h2>Profile the given input</h2>')
import cProfile; import profile
@interact
def _(cmd = ("Statement", 'divisors(10^10)'),
      do_preparse=("Preparse?", True),
      cprof =("cProfile?", False)):
    if do_preparse: cmd = preparse(cmd)
    print "<html>"  # a trick to avoid word wrap
    if cprof:
        cProfile.run(cmd)
    else:
        profile.run(cmd)
    print "</html>"
```



Profile the given input

Statement divisors(10^10)

Preparse? ☑

cProfile? □

```
         563 function calls (530 primitive calls) in 0.009 CPU seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 :0(IntegerRing)
      147    0.001    0.000    0.001    0.000 :0(__getitem__)
      123    0.001    0.000    0.001    0.000 :0(append)
        1    0.000    0.000    0.000    0.000 :0(degree)
        1    0.000    0.000    0.000    0.000 :0(get_debug_level)
        1    0.000    0.000    0.000    0.000 :0(is_commutative)
        8    0.000    0.000    0.000    0.000 :0(isinstance)
    73/40    0.001    0.000    0.001    0.000 :0(len)
        1    0.000    0.000    0.000    0.000 :0(parent)
        1    0.000    0.000    0.000    0.000 :0(range)
        1    0.000    0.000    0.000    0.000 :0(set_debug_level)
        1    0.000    0.000    0.000    0.000 :0(setprofile)
        2    0.000    0.000    0.000    0.000 :0(sort)
        2    0.000    0.000    0.000    0.000 :0(sum)
        1    0.000    0.000    0.009    0.009 :1()
        1    0.000    0.000    0.000    0.000 arith.py:1556(__factor_using_pari)
        1    0.000    0.000    0.001    0.001 arith.py:1580(factor)
        1    0.004    0.004    0.009    0.009 arith.py:917(divisors)
        1    0.000    0.000    0.001    0.001 factorization.py:161(__init__)
      147    0.002    0.000    0.003    0.000 factorization.py:257(__getitem__)
       34    0.000    0.000    0.001    0.000 factorization.py:297(__len__)
        1    0.000    0.000    0.000    0.000 factorization.py:420(base_ring)
        1    0.000    0.000    0.000    0.000 factorization.py:439(is_commutative)
        1    0.000    0.000    0.000    0.000 factorization.py:484(simplify)
        2    0.000    0.000    0.000    0.000 factorization.py:500()
        1    0.000    0.000    0.000    0.000 factorization.py:511(sort)
        1    0.000    0.000    0.009    0.009 profile:0(divisors(Integer(10)**Integer(10)))
        0    0.000             0.000          profile:0(profiler)
        1    0.000    0.000    0.000    0.000 proof.py:151(get_flag)
        4    0.000    0.000    0.000    0.000 rational_field.py:135(__hash__)
        1    0.000    0.000    0.000    0.000 rational_field.py:147(__call__)
        1    0.000    0.000    0.000    0.000 rational_field.py:233(_coerce_impl)
```

1.5 Debugging Sage Code

There are many tools, some simple and some highly sophisticated, for debugging Sage programs. In practice, many skilled programmers use a combination of reasoning *very systematically* about their code, print statements, a debugger such as pdb or gdb, and occassionaly a sophisticated instrumentation system like Valgrind. The following is a typical quote from one of the main Cython developers:

> "For debugging, I mostly use print and unit tests for Python and print, tests and valgrind for Cython, so I can't really comment on the debugging environments." – Stefan Behnel (Cython developer), 2008-04-11

By far the most important tool in debugging mathematical programs is your ability to reason deductively. Always be careful about any deductions you make based on tools that you use.

In this section, we will only discuss print statements, pdb, and gdb below. If you want to debug more subtle problems you'll need to use the amazing Valgrind (see `http://valgrind.org/`), an instrumentation framework that can automatically detect many memory management and threading bugs, and profile your programs in detail. Note that Valgrind is currently only available on Linux.

Section 1.5.1 will hopefully increase your general skepticism toward the correctness of mathematical software. Section 1.5.2 explains various techniques for using print statements in Sage code to obtain information about what is happening. In Section 1.5.3 we dive into `pdb`, which is a nice interactive *command-line* debugger, and in Section 1.5.4 we discuss using `gdb` with Sage, which is an interactive *command-line* debugger for tracing down issues with Cython code (or more general C/C++ code). It is critical when searching for a bug in code to systematically use the tools you know well, so Section **??** describes strategies for narrowing down the source of a problem through a combination of logical deduction and data gathering via print statements and more sophisticated debugging tools.

1.5.1 Be Skeptical!

The best perspective to take when writing or using code is to assume that it has bugs. Be skeptical! If you read source code of Sage or its components you'll sometimes see things that will make you worry. This is a good thing. For instance, this occurs in a comment in the Sage code by Jonathan Bober for computing the number of partitions of an integer (the `number_of_partitions` command):

```
// Extensive trial and error has found 3 to be the smallest
// value that doesn't seem to produce any wrong answers.
// Thus, to be safe, we use 5 extra bits.
```

See http://www.sagemath.org/hg/sage-main/file/cc1e12a492fc/sage/combinat/partitions_c.cc for the exact version of source file mentioned above.

Incidentally, Bober's code for computing the number of partitions of an integer is currently faster than anything else in the world, and it is not known to have any bugs.

A healthy amount of skepticism and worry is a good to cultivate when doing computational mathematics. Never listen to anybody who suggests you do otherwise. Similar issues occur in closed source systems such as Mathematica, but you don't get to see them – that doesn't make the chance of bugs less likely. In fact, the Mathematica documentation says the following:

> But just as long proofs will inevitably contain errors that go undetected for many years, so also a complex software system such as Mathematica will contain errors that go undetected even after millions of people have used it.

There can be bugs in the implementation of basic arithmetic, even at the hardware level! For example, on some Pentium processors in the 1990s we had:

$$4195835.0/3145727.0 = 1.333739068902037589$$

Yikes, since already the fourth digit after the decimal place is wrong!

```
sage: 4195835.0/3145727.0
1.33382044913624
```

See http://www.trnicely.net/pentbug/ for the full story, which we quote from below:

> "I was pursuing a research project in an area of pure mathematics known as computational number theory. Specifically, I have written a code which enumerates the primes [...]. Some of these results have been published in journal papers; many others are being published and updated at my Web site, http://www.trnicely.net. [...]
>
> Simultaneously with the calculation of the unknown quantities, a number of checks are maintained by calculating previously published values (such as $\pi(x)$, the number of primes $\leq x$). [...]
>
> Calculations began in early 1993. On 13 June 1994, the outputs of several runs were assembled, and I found that the computed value for $\pi(2 \cdot 10^{13})$ disagreed with the published value. This led to a long

search for *logic errors and sources of reduced precision in my source code* (some 3000 lines in all). In the process, I found that the Borland C++ 4.02 compiler was producing erroneous code when compiled in 32-bit mode with certain optimizations (-Op -Om -Og) enabled. For some time I believed this to be the source of my woes.

After eliminating this source of error, and rewriting the code to convert certain floating-point calculations from double precision to long double precision, I put the revised code into use on 10 September. To my dismay, I soon discovered (on 4 October) that I was now encountering a new error, a discrepancy in the long double (sum of the) floating-point reciprocals returned by the x87 FPU. The results for the first trillion, as computed on the Pentium-60, differed from the results obtained on a 486DX-33 by an amount orders of magnitude in excess of that expected from rounding or truncation error accumulation (the floating-point and ultraprecision sums also differed, but by an amount less than the expected floating-point noise). Through trial and error and finally a binary search, the discrepancy was isolated to the twin-prime pair (824633702441, 824633702443), which was producing incorrect floating-point reciprocals (the ultraprecision reciprocals were also in error, by a lesser amount, evidently due to the incidental dependency on floating-point arithmetic in Lenstra's original integer arithmetic code).

My first conjecture was that the error was again an artifact of the Borland compiler, but even completely disabling optimization failed to eliminate the problem. [...]

The final pieces of the puzzle fell into place during the week of 16–22 October. On 17 October I gained access to a second Pentium, which had a motherboard from a different manufacturer. The error was present in this machine as well. During 17–19 October, I reproduced the error in a code written in Power Basic, eliminating the C compiler as a cause. I reproduced the error in a Quattro Pro spreadsheet, and also verified that the error disappeared when the FPU was locked out in real-mode DOS (this is difficult to do in Windows code or 32-bit code, which I was using for my main application). On 21 October, I ran the test code on a 486DX2-66 with a PCI bus; when no error appeared, I felt that the PCI bus had been eliminated as a cause. On 22 October, I tested the code on still a third Pentium on display at Staples, a local office supply store; this Packard-Bell machine also produced the error. I was now certain that the error was in the FPU of the Pentium chip.

[...] In the absence of any meaningful response from Intel or Micron, on 30 October I sent e-mail to a number of individuals and organizations whom I felt would have access to many other Pentium systems, and asked them to check for the problem. I believe you are aware of events from that point on. [Lots of publicity and articles in the popular press.]

So be skeptical when you do mathematical computations!

Always think about what mathematical computations tell you, and try to find ways to *double and triple check* results that matter.

It almost goes without saying, but beautifully written, well-documented code that has been around for a long time and used a great deal is generally much less likely to be seriously buggy than newly written code. A great example of such high quality – by aging – code is NTL (`http://www.shoup.net/ntl/`). Quality code that has been used for years, and thus likely has few bugs is like gold – treat it that way; don't just toss it out and assume that something new that you sit down and write is likely to be better. That said, if you persist you can and will write beautiful code that equals or surpasses anything ever done before. When you do this, please consider contributing you code to the Sage project.

There are several good reasons to write new code. One excellent reason is that you simply want to understand an algorithm well, perhaps one you're learning about in a course, a book, a paper, or that you just designed. Implementing an algorithm correctly forces you to understand every detail, which can provide new insight into the algorithm. If you're implementing a nontrivial algorithm that is described in a book or paper, the chances are high that it is wrong in some subtle way (e.g., a typo in a formula, a fundamental mistake in the algorithm, whatever) – so you will definitely learn something, and possibly improve the mathematical literature while you're at it. See, for example, the article *Beautiful Tests* by Alberto Savoia in *Beautiful Code: Leading Programmers Explain How They Think* by O'Reilly `http://www.oreilly.com/catalog/9780596510046/` for an example of a subtle bug in a famous and "proven correct" implementation of binary search.

Another great reason to write new code is to implement an algorithm that isn't available in Sage. When you do this, make sure to be skeptical about the correctness of your code; always test it extensively, document it, etc., just to increase the chance that it might always work correctly. And if there is any way to independently verify correctness of the output of code, attempt to implement this too. If the algorithm is available in other mathematical software such as Maple or Mathematica, and you have that program, use the interfaces described in Section 1.7 to write code that automatically tests the output of your implementation against the output of the implementation in that other system. Include such test code in your final product. Get other people to read your code. Get your code into Sage.

A final good reason to write new code is that people really want to use it, and they will greatly appreciate you writing that code.

"If you implement this I will be forever in your debt."
– Nick Alexander, on sage-devel,

April 14, 2008.

1.5.2 Using Print Statements

Yes, you *can* use print statements for debugging your code. There is no shame in this! Especially when using Python where you do not have to recompile every time, this can be a particularly useful technique.

Here are some simple but powerful techniques for using print statements when debugging code:

1. Put `print 0`, `print 1`, `print 2`, etc., at various points in your code. This will show you were something crashes or some other weird behavior happens. Sprinkle in more print statements until you narrow down exactly where the problem occurs.

2. Print the values of variables at key spots in your code.

3. Print other state information about Sage at key spots in your code, e.g., `cputime`, `walltime`, `get_memory_usage`, etc.

The main key to using the above is to think deductively and carefully about what you are doing, and hopefully isolate the problem. Also, with experience you'll recognize which problems are best tracked down using print statements, and which are not.

1.5.3 Debugging Python using pdb

Sage includes a very nice interactive debugger that allows you to basically interactively insert print statements into your running program, and move up and down the execution stack and run bits of code.

WARNING: *Unfortunately, nothing in this section works in the Sage notebook (yet!).*

It is easy to use the Python debugger from the Sage *command line*. Just turn it on by typing `%pdb`:

```
sage: %pdb
Automatic pdb calling has been turned ON
```

Note that typing `%pdb` again toggles pdb calling off.

To use `%pdb` just run some code that produces a traceback. You'll get dumped into the running Sage session at exactly the point where the exception was raised:

```
sage: plot(sin(x), 0, 'hello')
---------------------------------------------------------------------
<type 'exceptions.ValueError'>   Traceback (most recent call last)
...
<type 'exceptions.ValueError'>: invalid literal for float(): hello
> /Users/was/build/sage-3.0.alpha1/local/lib/python2.5/site-packages
                      /sage/plot/plot.py(4320)var_and_list_of_values()
   4319     a = float(a)
-> 4320     b = float(b)
   4321     if plot_points == 2:

ipdb> print b
hello
```

In fact, you can execute *any* Python code at the ipdb prompt, which is pretty amazing.

```
ipdb> type(b)
<type 'str'>


ipdb> ?

Documented commands (type help <topic>):
========================================
EOF    break  commands   debug    h
l      pdef   quit       tbreak   whatis
a      bt     condition  disable  help
list   pdoc   r          u        where
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

Miscellaneous help topics:
==========================
exec  pdb

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

You should try this out and type help to get help on each of the above commands. You can move up the execution stack, inspect code, execute any line of Python code, etc.

Technical Note 1.9. The `%pdb` command is part of the IPython interactive shell (see Note 1.2). It takes the `pdb` module that is included standard in Python and adds:

- proper handling of input history when entering/leaving pdb
- tab completion
- more introspection commands like pinfo (equivalent of '?' while in pdb), pdoc, etc.
- more complete (and syntax highlighted) tracebacks and code listings.

Setting a break point: If you have some code in a file and would like to drop into the debugger at a given point, put the following code at that point in the file:

```
import pdb; pdb.set_trace()
```

This will pop you into the debugger at that point in the file. By the way, as mentioned above, `pdb` is a standard Python module, which you can read more about here `http://docs.python.org/lib/module-pdb.html`.

There is also a trace facility available from the Sage command line. If you want to step through each line of execution of an expression type, e.g.,

```
sage: trace("factor(100)")              # note the quotes!
```

then at the pdb prompt type `s` (or `step`), then press return over and over to step through every line of *interpreted pure Python code* that is called in the course of the above computation. (All compiled and Cython code is skipped!) Type `?` at any time for help on how to use the debugger (e.g., `l` lists 11 lines around the current line; `bt` gives a back trace, etc.).

1.5.4 Advanced: Debugging Using gdb

You cannot use pdb to walk through or inspect Cython code, or code defined in external C/C++ libraries. The GNU debugger (gdb) can be used for this. The main (only) way I use gdb is to figure out precisely where a segmentation fault occurs. For example, consider the following block of Cython code:

```
cdef int bad():
    # <int*> casts 0 to a pointer
    cdef int* pointer = <int*>0
    cdef int n
    n = 10
    # [0] is pointer dereferencing in cython, since
    # *pointer is ambiguous and hard to parse.
    n = pointer[0]
    return n

def foo():
    return bad()
```

If we define the above code in the notebook (by putting `%cython` at the beginning of a cell and try it, we get a segmentation fault:

```
{{{
%cython
cdef int bad():
    cdef int* pointer = <int*>0  # <int*> casts 0 to a pointer
    cdef int n
    n = 10
    n = pointer[0]   # [0] is pointer dereferencing in cython
    return n
def foo():
    return bad()
}}}

{{{
foo()
///
------------------------------------------------------------
Unhandled SIGBUS: A bus error occured in SAGE.
This probably occured because a *compiled* component
of SAGE has a bug in it (typically accessing invalid memory)
or is not properly wrapped with _sig_on, _sig_off.
You might want to run SAGE under gdb with ’sage -gdb’
to debug this.  SAGE will now terminate (sorry).
------------------------------------------------------------
}}}
```

How can we automatically track this problem down? We could sprinkle print statements around as in Section 1.5.2, and we *would* find the problem.

Alternatively, on the *command line* we can use gdb to track down the problem.

Put the Cython code (without bad.spyx. Then type sage -gdb and load bad.spyx:

```
----------------------------------------------------------------------
| SAGE Version 3.0.alpha1, Release Date: 2008-04-04                  |
| Type notebook() for the GUI, and license() for information.        |
----------------------------------------------------------------------
sage: load "bad.spyx"
Compiling bad.spyx...
Reading symbols for shared libraries . done
sage: foo()
Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_PROTECTION_FAILURE at address: 0x00000000
__pyx_pf_58_Users_was_edu_2007_2008_sage_lectures_20080414_bad_spyx_0_foo
(__pyx_self=0x0, unused=0x0) at
_Users_was_edu_2007_2008_sage_lectures_20080414_bad_spyx_0.c:138
138   __pyx_1 = PyInt_FromLong(__pyx_f_58_Users_was_edu_2007_2008_\
sage_lectures_20080414_bad_spyx_0_bad()); if (unlikely(!__pyx_1))
{__pyx_filename = __pyx_f[0]; __pyx_lineno = 15; goto __pyx_L1;}
(gdb) bt
#0  __pyx_pf_58_Users_was_edu_2007_2008_sage_lectures_20080414_bad_spyx_0_f
(__pyx_self=0x0, unused=0x0) at
_Users_was_edu_2007_2008_sage_lectures_20080414_bad_spyx_0.c:138
...
#18 0x000c34a3 in PyRun_SimpleFileExFlags ()
#19 0x000d1e92 in Py_Main ()
#20 0x0000210c in _start ()
#21 0x00002039 in start ()
```

The above is potentially useful. It's even more useful if you track down the autogenerated C file that the traceback refers to, which is stored in

```
$HOME/.sage/temp/`hostname`/pid/spyx/exacty_path_to_spyx
```

where pid is the current pid, e.g., as output by os.getpid() when you start Sage. (Note that the above directory will be deleted when you quite Sage.)

Remark 1.10. A good project would be to somehow improve how easy it is to track down the point where a segfault or other error occurs in Cython code. I.e., make the above few paragraphs more user friendly.

Also, %cython in the notebook is implemented by simply calling `cython.eval(...)` on a string. It would likely be very easy to make a debug variant of this command that literally inserts Python print statements before every single line (say printing the raw text of that line and values of variables(?)), so one could see where/when a bug occurs and watch Cython code execute. This could be quite useful.

Another idea would be to make it easy to run blocks of code under gdb from the notebook, and if something goes wrong display the output of gdb. This code block would run as a separate python process.

1.6 Source Control Management

1.6.1 Distributed versus Centralized

1.6.2 Mercurial

1.7 Using Mathematica, Matlab, Maple, etc., from Sage

A distinctive feature of Sage is that Sage supports using Maple, Mathematica, Octave, Matlab, and many other programs from Sage, assuming you have the relevant program (there are some caveats). This makes it much easier to combine the functionality of these systems with each other and Sage.

Before discussing interfaces in more detail, we make a few operating system dependent remarks.

what works on all os's; in particular gap, singular, gp, maxima, always there.

what works on linux

on os x

on windows

----

example of using gp to do something.

example of using mathematica

example of using maple

example of using matlab/octave

----

eval versus call.

Discussion of what goes on behind the scenes. Files used for large inputs -- named pipes for small.

Warning-- multiple processes; complicated; can get parallel, which is harder to think about...

Chapter 2

Algebraic Computing

```
00:00 < ondrej> I got it
00:00 < gfurnish|away> in symbolics
00:00 < gfurnish|away> I don’t think sympy handles this well
00:00 < ondrej> well, I think you can do the same with regular classes
00:01 < ondrej> sympy doesn’t handle Z/6
00:01 < gfurnish|away> not as elegantly
00:01 < gfurnish|away> Come to the algebraic side :)
00:01 < ondrej> but you need to have the code to support that anyway
00:01 < ondrej> :)
00:01 < wstein|afk> sage does; but not symbolically....
00:01 < wstein|afk> It will be exciting to do it symbolically!
00:01 < wstein|afk> I don’t know any system that does that.
00:01 < wstein|afk> Maybe axiom...
00:01 < gfurnish|away> see?  The feature gets the number theorists happy.
00:01 < ondrej> me neither
00:02  * ondrej knows Sage people love rings. But he thought that’s because they
00:03 < ondrej> anyway, I should have asked about that at SD8, it’s difficult fo
                to work.
00:03 < gfurnish|away> Well you know how crazy theoretical physicists have gotte
00:03 < gfurnish|away> Number theory and algebraic geometry are starting to beco
                       :)
00:05 < ondrej> yeah I know. I am looking at it just from a purely technical (in
00:06 < ondrej> I’ll just study Sage sources more and I get the idea
00:07 < wstein|afk> Do you mean "using rings" instead of "classes"?
00:07 < wstein|afk> I.e., "what is the point"?
00:07 < gfurnish|away> I’m actually curious where the idea came from.. it was ma
00:08 < wstein|afk> For *me* it was Magma.  100%.
00:08 < wstein|afk> I had never heard of anything like that before Magma.
00:08 < wstein|afk> Or rather -- it’s what mathematicians do theoretically.
00:09 < wstein|afk> The bold thing in Magma is to actually program it on a compu
00:09 < wstein|afk> It’s where the name of Magma comes from.
00:09 < mhansen> It seems like the only natural thing to do if you want to work
00:09 < wstein|afk> So yes,   think it is solidly John Cannon’s very good idea.
00:09 < wstein|afk> It seems very natural in retrospect.
00:09 < wstein|afk> But in 1998 it shocked me.
00:10 < wstein|afk> And it is *very* different than Maxima, Maple, Mathematica,
00:10 < wstein|afk> Even GAP.
00:10 < wstein|afk> The thing is that from some points of view it’s impossible t
00:10 < wstein|afk> with the infinite objects of mathematics.
00:10 < wstein|afk> That’s a very natural limiting way to think about things.
```

Algebraic computing is concerned with computing with the algebraic objects of mathematics, such as arbitrary precision integers and rational numbers, groups, rings, fields, vector spaces and matrices, and other objects. The tools of algebraic computing support research and education in pure mathematics, underly the design of error correcting codes and cryptographical systems, and play a role in scientific and statistical computing.

2.1 Groups, Rings and Fields

2.1.1 Groups

2.1.2 Rings

2.1.3 Fields

2.2 Number Theory

2.2.1 Prime numbers and integer factorization

2.2.2 Elliptic curves

2.2.3 Public-key cryptography: Diffie-Hellman, RSA, and Elliptic curve

2.3 Linear Algebra

2.3.1 Matrix arithmetic and echelon form

Matrix multiplication using a numerical BLAS (in both mod p and over ZZ cases)

2.3.2 Vector spaces and free modules

2.3.3 Solving linear systems

Applicatin: computing determinants over ZZ

2.4 Systems of polynomial equations

2.5 Graph Theory

2.5.1 Creating graphs and plotting them

2.5.2 Computing automorphisms and isomorphisms

2.5.3 The genus and other invariants

Chapter 3

Scientific Computing

Scientific computing is concerned with constructing
mathematical models and using numerical techniques to
solve scientific, social, and engineering problems.

3.1 Floating Point Numbers

3.1.1 Machine precision floating point numbers

3.1.2 Arbitrary precision floating point numbers

3.2 Interval arithmetic

3.3 Root Finding and Optimization

3.3.1 Single variable: max, min, roots, rational root isolation

3.3.2 Multivariable: local max, min, roots

3.4 NumericalSolution of Linear Systems

3.4.1 Solving linear systems using LU factorization

3.4.2 Solving linear systems iteratively

3.4.3 Eigenvalues and eigenvectors

3.5 Symbolic Calculus

3.5.1 Symbolic Differentiation and integration

3.5.2 Symbolic Limits and Taylor series

3.5.3 Numerical Integration

Chapter 4

Statistical Computing

4.1 Introduction to R and Scipy.stats

4.1.1 The R System for Statistical Computing

4.1.2 The Scipy.stats Python Library

4.2 Descriptive Statistics

4.2.1 Mean, standard deviation, etc.

4.3 Inferential Statistics

4.3.1 Simple Inference

4.3.2 Conditional Inference

4.4 Regression

4.4.1 Linear regression

4.4.2 Logistic regression

Bibliography

[ABC+] B. Allombert, K. Belabas, H. Cohen, X. Roblot, and I. Zakharevitch, PARI/GP, http://pari.math.u-bordeaux.fr/.

[BCP97] W. Bosma, J. Cannon, and C. Playoust, *The Magma algebra system. I. The user language*, J. Symbolic Comput. **24** (1997), no. 3--4, 235--265, Computational algebra and number theory (London, 1993). MR 1 484 478

[jQu] jQuery, *A new type of javascript library*, http://www.4dsolutions.net/ocn/overcome.html.

[Pex] Pexpect, *A pure python expect-like module*, http://www.noah.org/wiki/Pexpect#Download_and_Installation.

[PG07] Fernando. Pérez and Brian E. Granger, *IPython: a System for Interactive Scientific Computing*, Comput. Sci. Eng. **9** (2007), no. 3, 21--29, University of Colorado APPM Preprint #549.

[Twi] Twisted, *A framework for networked applications*, http://twistedmatrix.com.