Sage FAQ

Release 9.3

The Sage Development Team

May 10, 2021

CONTENTS

CHAPTER

ONE

FAQ: GENERAL

1.1 Why does this project exist?

The stated mission of Sage is to be viable free open source alternative to Magma, Maple, Mathematica, and Matlab. Sage's predecessors, known as HECKE and Manin, came about because William Stein needed to write them as part of his research in number theory. Started by William in 2005 during his time at Harvard University, Sage combines best-of-breed free open source mathematics software, packaging and unifying them through a common interface. Since then Sage has become something used not just by researchers in number theory, but throughout the mathematical sciences.

Sage builds upon and extends functionalities of many underlying packages. Even from early on, when Sage was primarily used for number theory, this included Givaro, MPIR, NTL, Pari/GP, and many others too numerous to list here. Students, teachers, professors, researchers throughout the world use Sage because they require a comprehensive free open source mathematics package that offers symbolic and numerical computation. Most of the time, people are happy with what Sage has to offer.

As is common throughout the free open source software (FOSS) world, many people often identify cases where Sage lacks certain mathematics functionalities that they require. And so they delve into the underlying source code that comprises Sage in order to extend it for their purposes, or expose functionalities of underlying packages shipped with Sage in order to use their favourite mathematics software packages from within Sage. The Sage-Combinat team is comprised of researchers in algebraic combinatorics. The team's stated mission is to improve Sage as an extensible toolbox for computer exploration in algebraic combinatorics, and foster code sharing between researchers in this area.

For detailed information about why Sage exists, see William's personal mathematics software biography.

1.2 What does "Sage" mean and how do you pronounce it?

In the first few years of Sage's existence, the project was called "SAGE". This acronym stood for "Software for Algebra and Geometry Experimentation". Starting around 2007 and early 2008, the name "Sage" was widely adopted. Think of "Sage" as a name for a free open source mathematics software project, just as "Python" is a name for a free open source general purpose programming language. Whenever possible, please use the name "Sage" instead of "SAGE" to avoid confusing the Sage project with a computer project called SAGE. You pronounce "Sage" similar to how you would pronounce "sage" which refers to a wise person, or "sage" which refers to a plant. Some people pronounce "Sage" as "sarge", similar to how you would pronounce Debian Sarge.

However you pronounce "Sage", please do not confuse the Sage project with an accounting software by the same name.

1.3 Who is behind this project?

Sage is a volunteer based project. Its success is due to the voluntary effort of a large international team of students, teachers, professors, researchers, software engineers, and people working in diverse areas of mathematics, science, engineering, software development, and all levels of education. The development of Sage has benefited from the financial support of numerous institutions, and the previous and ongoing work of many authors of included components.

A list of (some) direct contributors can be found on the Sage Development Map and the history of changes can be found in the changelogs. Refer to the acknowledgment page of the Sage website for an up-to-date list of financial and infrastructure supporters, mirror network hosting providers, and indirect contributors.

1.4 Why is Sage free/open source?

A standard rule in the mathematics community is that everything is laid open for inspection. The Sage project believes that not doing the same for mathematics software is at best a gesture of impoliteness and rudeness, and at worst a violation against standard scientific practices. An underlying philosophical principle of Sage is to apply the system of open exchange and peer review that characterizes scientific communication to the development of mathematics software. Neither the Sage project nor the Sage Development Team make any claims to being the original proponents of this principle.

The development model of Sage is largely inspired by the free software movement as spearheaded by the Free Software Foundation, and by the open source movement. One source of inspiration from within the mathematics community is Joachim Neubüser as expressed in the paper

- J. Neubüser. An invitation to computational group theory. In C. M. Campbell, T. C. Hurley, E. F. Robertson, S. J. Tobin, and J. J. Ward, editors, *Groups '93 Galway/St. Andrews, Volume 2*, volume 212 of London Mathematical Society Lecture Note Series, pages 457–475. Cambridge University Press, 1995.

and in particular the following quotation from his paper:

```
You can read Sylow's Theorem and its proof in Huppert's book in
the library without even buying the book and then you can use
Sylow's Theorem for the rest of your life free of charge,
but...for many computer algebra systems license fees have to be
paid regularly for the total time of their use. In order to
protect what you pay for, you do not get the source, but only an
executable, i.e. a black box. You can press buttons and you get
answers in the same way as you get the bright pictures from your
television set but you cannot control how they were made in either
case.

With this situation two of the most basic rules of conduct in
mathematics are violated: In mathematics information is passed on
free of charge and everything is laid open for checking. Not
applying these rules to computer algebra systems that are made for
mathematical research...means moving in a most undesirable
direction. Most important: Can we expect somebody to believe a
result of a program that he is not allowed to see? Moreover: Do we
really want to charge colleagues in Moldava several years of their
salary for a computer algebra system?
```

Similar sentiments were also expressed by Andrei Okounkov as can be found in

- V. Muñoz and U. Persson. Interviews with three Fields medalists. *Notices of the American Mathematical Society*, 54(3):405–410, 2007.

in particular the following quotation:

```
Computers are no more a threat to mathematicians than food
processors are a threat to cooks. As mathematics gets more and
more complex while the pace of our lives accelerates, we must
delegate as much as we can to machines. And I mean both numeric
and symbolic work. Some people can manage without dishwashers, but
I think proofs come out a lot cleaner when routine work is
automated.

This brings up many issues. I am not an expert, but I think we
need a symbolic standard to make computer manipulations easier to
document and verify. And with all due respect to the free market,
perhaps we should not be dependent on commercial software here. An
open-source project could, perhaps, find better answers to the
obvious problems such as availability, bugs, backward
compatibility, platform independence, standard libraries, etc. One
can learn from the success of TeX and more specialized software
like Macaulay2. I do hope that funding agencies are looking into
this.
```

1.5 Why did you write Sage from scratch, instead of using other existing software and/or libraries?

Sage was not written from scratch. Most of its underlying mathematics functionalities are made possible through FOSS projects such as

- ATLAS — Automatically Tuned Linear Algebra Software.
- BLAS — Basic Linear Algebra Subprograms.
- ECL — Embeddable Common-Lisp system
- FLINT — C library for doing number theory.
- GAP — a system for computational discrete algebra, with particular emphasis on computational group theory.
- GMP — GNU Multiple Precision Arithmetic Library.
- Maxima — system for symbolic and numerical computation.
- mpmath — a pure-Python library for multiprecision floating-point arithmetic.
- NumPy and SciPy — numerical linear algebra and other numerical computing capabilities for Python.
- OpenBLAS — an optimized BLAS library.
- Pari/GP — a computer algebra system for fast computations in number theory.
- Pynac — a modified version of GiNaC that replaces the dependency on CLN by Python.
- R — a language and environment for statistical computing and graphics.
- And many more too numerous to list here.

An up-to-date list can be found in the section External Packages in the Sage Reference Manual.

The principal programming languages of Sage are Python and Cython. Python is the primary programming and interfacing language, while Cython is the primary language for optimizing critical functionalities and interfacing with C libraries and C extensions for Python. Sage integrates over 90 FOSS packages into a common interface. On top of these packages is the Sage library, which consists of over 700,000 lines of new Python and Cython code. See openhub.net for source code analysis of the latest stable Sage release.

1.6 How do I get help?

For support about usage of Sage, there are two options:

- The question-and-answer website ask.sagemath.org
- The email list sage-support

For support about development of Sage, there is an email list sage-devel

See http://www.sagemath.org/help.html for a listing of other resources.

1.7 Wouldn't it be way better if Sage did not ship as a gigantic bundle?

The SageMath distribution continues to vendor versions of required software packages ("SPKGs") that work well together.

However, in order to reduce compilation times and the size of the Sage installation, a development effort ongoing since the 8.x release series has made it possible to use many system packages provided by the OS distribution (or by the Homebrew or conda-forge distributions) instead of building SageMath's own copies.

This so-called "spkg-configure" mechanism runs at the beginning of a build from source, during the `./configure` phase.

To ensure that SageMath builds and runs correctly on a wide variety of systems, we use automated testing. See the chapter Portability testing in the Developer's Guide for details.

1.8 With so many bugs in Sage and hundreds of open tickets, why don't you produce a stabilization release?

Any software package contains bug. With something as complex as Sage, neither the Sage community nor the Sage Development Team make any claims that Sage is free of bugs. To do so would be an act of dishonesty.

A Sage release cycle usually lasts for a few months, with several betas appearing at a 2-3 week intervals. Each release cycle is usually chaired by a single release manager who looks after the Sage merge tree for the duration of the release cycle. During that time, the release manager often needs to devote the equivalent of full-time work to quality management and actively interacts with an international community of Sage users, developers, and potential contributors.

There have been a number of cases where two Sage contributors paired up to be the release managers for a Sage release cycle. However, it is often the case that few people have the equivalent of 3 weeks' worth of free time to devote to release management. If you want to help out with release management, please subscribe to the sage-release mailing list.

Since the beginning of the Sage project, Sage contributors have tried to listen and think about what would increase the chances that serious potential contributors would actually contribute. What encourages one contributor can discourage another, so tradeoffs need to be made. To decide that a stabilization release would merge patches with bug fixes, and only fix bugs, would likely discourage someone from contributing when they have been told in advance that their positively reviewed patches will not be merged.

The Sage community believes in the principle of "release early, release often". How the Sage project is organized and run differ greatly from that of a commercial software company. Contributors are all volunteers and this changes the dynamic of the project dramatically from what it would be if Sage were a commercial development effort with all developers being full-time employees.

1.9 How can I download the Sage documentation to read it offline?

To download the Sage standard documentation in HTML or PDF formats, visit the Help and Support page on the Sage website. Each release of Sage comes with the full documentation that makes up the Sage standard documentation. If you have downloaded a binary Sage release, the HTML version of the corresponding documentation comes pre-built and can be found under the directory `SAGE_ROOT/local/share/doc/sage/html/`. During the compilation of Sage from source, the HTML version of the documentation is also built in the process. To build the HTML version of the documentation, issue the following command from `SAGE_ROOT`:

```
$ ./sage --docbuild --no-pdf-links all html
```

Building the PDF version requires that your system has a working LaTeX installation. To build the PDF version of the documentation, issue the following command from `SAGE_ROOT`:

```
$ ./sage --docbuild all pdf
```

For more command line options, refer to the output of any of the following commands:

```
$ ./sage --help
$ ./sage --advanced
```

CHAPTER

TWO

FAQ: USING SAGE

2.1 How do I get started?

You can try out Sage without downloading anything:

- **CoCalc™:** Go to https://cocalc.com and set up a free account.

  If you log in, you will gain access to the latest version of Sage and to many other programs.

  Note that this website is an independent commercial service.

- **Sage cell:** A "one-off" version of Sage, available for doing one computation at a time. https://sagecell.sagemath.org/

To download a **pre-built binary** Sage distribution, visit http://sagemath.org/download.html and click on the link for the binary for your operating system.

The **source code** of Sage is also available for you to download and use. Go to http://www.sagemath.org/download-source.html to download the tar archive for any release of Sage.

The Sage Jupyter notebook runs within a web browser. To start the notebook, issue the following command in a terminal, if `sage` is in your `PATH`

```
$ sage -notebook
```

2.2 What are the prerequisites for installing a copy of Sage on my computer?

Most of the dependencies of Sage are shipped with Sage itself. In most cases, you can download a pre-built binary and use that without installing any dependencies. If you use Windows, you will need to install VirtualBox, which can be downloaded from the page https://www.virtualbox.org/wiki/Downloads. After installing VirtualBox, you need to download a VirtualBox distribution of Sage available at http://www.sagemath.org/download-windows.html. Ensure you follow the instructions at that page, then start the Sage virtual machine using the VirtualBox software.

You can get the complete source for Sage to compile it on your own Linux or Mac OS X system. Sage lives in an isolated directory and does not interfere with your surrounding system. It ships together with everything necessary to develop Sage, the source code, all its dependencies and the complete changelog. On Linux systems like Debian/Ubuntu, you may have to install the `build essential` package and the `m4` macro processor. Your system needs to have a working C compiler if you want to compile Sage from source. On Debian/Ubuntu, you can install these prerequisites as follows:

```
$ sudo apt-get install build-essential m4
```

If you have a multi-core system, you can opt for a parallel build of Sage. The command

```
$ export MAKE='make -j8'
```

will enable 8 threads for parts of the build that support parallelism. Change the number 8 as appropriate to suit the number of cores on your system. Some Sage installations may have OpenMP-enabled BLAS (and other) libraries. The amount of OpenMP parallelism is controlled by the environment variable OMP_NUM_THREADS; however, it is known to not play well with Python parallelism, and you might want to

```
$ export OMP_NUM_THREADS=1
```

in case of crashes or hangs.

More details may be found in Installation Manual.

2.3 How to get Sage's Python to recognize my system's Tcl/Tk install?

It may be that you have Tcl/Tk installed and that your system's Python recognizes it but Sage's Python does not. To fix that, install the tcl/tk development library. On Ubuntu, this is the command

```
$ sudo apt-get install tk8.5-dev
```

or something along that line. Next, reinstall Sage's Python:

```
$ sage -f python3
```

This will pick up the tcl/tk library automatically. After successfully reinstalling Sage's Python, from within the Sage command line interface, issue these commands:

```
import _tkinter
import Tkinter
```

If they do not raise an `ImportError` then it worked.

2.4 How do I import Sage into a Python script?

You can import Sage as a library in a Python script. One caveat is that you need to run that Python script using the version of Python that is bundled with Sage (Sage 9.2 ships with Python 3.7.x). To import Sage, put the following in your Python script:

```
from sage.all import *
```

When you want to run your script, you need to invoke Sage with the option `-python` which would run your script using the version of Python that comes with Sage. For example, if Sage is in your `PATH` variable then you can do this:

```
$ sage -python /path/to/my/script.py
```

Another way is to write a Sage script and run that script using Sage itself. A Sage script has the file extension `.sage` and is more or less a Python script but uses Sage-specific functions and commands. You can then run that Sage script like so:

```
$ sage /path/to/my/script.sage
```

This will take care of loading the necessary environment variables and default imports for you.

2.5 How can I reload a Python script in a Sage session?

You can load a Python script in a Sage session with the command **load**. For example, we could use Sage to import a file called simple.py with:

```
load("simple.py")
```

and repeat this command every time that we change the file simple.py. However, if we type:

```
attach("simple.py")
```

every change applied to the file simple.py will be automatically updated in Sage.

2.6 Can I use SageMath with Python 3.x?

Since release 9.0 from January 2020, SageMath is running on top of Python 3.

2.7 I downloaded a Sage binary and it crashes on startup with "Illegal instruction". What can I do?

One way to fix this is to build Sage entirely from source. Another option is to fix your Sage installation by rebuilding MPIR and ATLAS by typing the following from the `SAGE_ROOT` of your Sage installation directory and wait about 15 to 20 minutes

```
$ rm spkg/installed/mpir* spkg/installed/atlas*
$ make
```

It is possible that the binaries have been built for a newer architecture than what you have. Nobody has yet figured out how to build Sage in such a way that MPIR and ATLAS work on all hardware. This will eventually get fixed. Any help is appreciated.

2.8 I used XXX to install Sage X.Y and that version is giving lots of errors. What can I do?

The version of Sage, i.e. Sage version X.Y, that is available on your XXX system through its package manager, is very old. No one has yet found time to update the XXX version of Sage. Any help is greatly appreciated. You should download the latest version of Sage from the download page. If you would like to help with updating the XXX version of Sage, please email the sage-devel mailing list.

2.9 Should I use the official version or development version?

You are encouraged to use the latest official version of Sage. Development versions are frequently announced on the sage-devel and sage-release mailing lists. An easy way of helping out with Sage development is to download the latest development release, compile it on your system, run all doctests, and report any compilation errors or doctest failures.

2.10 Is Sage difficult to learn?

Basic features of Sage should be as easy to learn as learning the basics of Python. Numerous tutorials are available online to help you learn Sage. To get the most out of Sage, you are encouraged to learn some features of the Python programming language. Here is an incomplete list of resources on Python. Further resources can be found by a web search.

- Building Skills in Python by Steven F. Lott
- Dive into Python by Mark Pilgrim
- How to Think Like a Computer Scientist by Jeffrey Elkner, Allen B. Downey, and Chris Meyers
- Official Python Tutorial
- Python home page and the Python standard documentation

2.11 Can I do X in Sage?

You are encouraged to use Sage's tab autocompletion. Just type a few characters, hit the tab key, and see if the command you want appears in the list of tab autocompletion. If you have a command called `mycmd`, then type `mycmd.` and hit the tab key to get a list of functionalities that are supported by that command. To read the documentation of `mycmd`, type `mycmd?` and press the enter key to read the documentation for that command. Similarly, type `mycmd??` and hit the enter key to get the source code of that command. You are also encouraged to search through the source code and documentation of the Sage library. To search through the source code of the Sage library, use the command `search_src("<search-keyword>")` where you should replace `<search-keyword>` with the key words you are looking for. Also, you can search through the documentation of the Sage library using the command `search_doc("<search-keyword>")`.

2.12 What exactly does Sage do when I type "0.6**2"?

When you type "0.6**2" in Python, it returns something like 0.35999999999999999. But when you do the same in Sage it returns 0.360000000000000. To understand why Python behaves as it does, see the Python Tutorial, especially the chapter "Floating Point Arithmetic: Issues and Limitations". What Sage does is "preparse" the input and transforms it like this:

```
sage: preparse("0.6**2")
"RealNumber('0.6')**Integer(2)"
```

So what is *actually* run is:

```
RealNumber('0.6')**Integer(2)
```

The Sage developers (in fact, Carl Witty) decided that Sage floating point numbers should by default print only the known correct decimal digits, when possible, thus skirting the problem that Python has. This decision has its pros and

cons. Note that `RealNumber` and `Integer` are Sage specific, so you would not be able to just type the above into Python and expect it to work without first an import statement such as:

```
from sage.all import RealNumber, Integer, preparse
```

2.13 Why is Sage's command history different from Magma's?

Using Sage, you are missing a feature of the Magma command line interface. In Magma, if you enter a line found in history using up arrow key and then press down arrow key, then the next line in history is fetched. This feature allows you to fetch as many successive lines in history as you like. However, Sage does not have a similar feature. The IPython command prompt uses the readline library (via pyreadline), which evidently does not support this feature. Magma has its own custom "readline-like" library, which does support this feature. (Since so many people have requested this feature, if anybody can figure out how to implement it, then such an implementation would certainly be welcome!)

2.14 I have type issues using SciPy, cvxopt or NumPy from Sage.

You are using SciPy or cvxopt or NumPy from Sage and you get type errors, e.g.

```
TypeError: function not supported for these types, and can't coerce safely to␣
↪supported types.
```

When you type in numbers into Sage, the pre-processor converts them to a base ring, which you can see by doing:

```
sage: preparse("stats.uniform(0,15).ppf([0.5,0.7])")
"stats.uniform(Integer(0),Integer(15)).ppf([RealNumber('0.5'),RealNumber('0.7')])"
```

Unfortunately, NumPy support of these advanced Sage types like `Integer` or `RealNumber` is not yet at 100%. As a solution, redefine `RealNumber` and/or `Integer` to change the behavior of the Sage preparser, so decimal literals are floats instead of Sage arbitrary precision real numbers, and integer literals are Python ints. For example:

```
sage: RealNumber = float; Integer = int
sage: from scipy import stats
sage: stats.ttest_ind(list([1,2,3,4,5]),list([2,3,4,5,.6]))
Ttest_indResult(statistic=0.0767529..., pvalue=0.940704...)
sage: stats.uniform(0,15).ppf([0.5,0.7])
array([  7.5,  10.5])
```

Alternatively, be explicit about data types, e.g.

```
sage: from scipy import stats
sage: stats.uniform(int(0),int(15)).ppf([float(0.5),float(0.7)])
array([  7.5,  10.5])
```

As a third alternative, use the raw suffix:

```
sage: from scipy import stats
sage: stats.uniform(0r,15r).ppf([0.5r,0.7r])
array([  7.5,  10.5])
```

You can also disable the preprocessor in your code via `preparser(False)`. You can start IPython alone from the command line `sage -ipython` which does not pre-load anything Sage-specific. Or switch the Notebook language to "Python".

2.15 How do I save an object so I don't have to compute it each time I open a worksheet?

The `save` and `load` commands will save and load an object, respectively.

2.16 Does Sage contain a function similar to Mathematica's ToCharacterCode[]?

You might want to convert ASCII characters such as "Big Mac" to ASCII numerals for further processing. In Sage and Python, you can use `ord`, e.g.

```
sage: list(map(ord, "abcde"))
[97, 98, 99, 100, 101]
sage: list(map(ord, "Big Mac"))
[66, 105, 103, 32, 77, 97, 99]
```

2.17 How can I write multiplication implicitly as in Mathematica?

Sage has a function that enables this:

```
sage: implicit_multiplication(True)
sage: x 2 x  # not tested
2*x^2
sage: implicit_multiplication(False)
```

This is preparsed by Sage into Python code. It may not work in a complicated situation. To see what the preparser does:

```
sage: implicit_multiplication(True)
sage: preparse("2 x")
'Integer(2)*x'
sage: implicit_multiplication(False)
sage: preparse("2 x")
'Integer(2) x'
```

See https://wiki.sagemath.org/sage_mathematica for more information about Mathematica vs. SageMath.

2.18 Can I make Sage automatically execute commands on startup?

Yes, just make a file `$HOME/.sage/init.sage` and it will be executed any time you start Sage. This assumes that the Sage environment variable `DOT_SAGE` points to the hidden directory `$HOME/.sage`, which by default is the case.

2.19 When I compile Sage my computer beeps and shuts down or hangs.

Compiling Sage is quite taxing on the CPU. The above behavior usually indicates that your computer has overheated. In many cases this can be fixed by cleaning the CPU fan and assuring proper ventilation of the system. Please ask your system administrator or a professional to do this in case you have never done this. Such hardware maintenance, if not performed by a skilled professional, you can potentially damage your system.

For Linux users, if you suspect that the compilation fails because of a resource issue, a fix might be to edit your `/etc/inittab` so that Linux boots into run level 3. The file `/etc/inittab` usually contains something similar to the following snippet:

```
#   0 - halt (Do NOT set initdefault to this)
#   1 - Single user mode
#   2 - Multiuser, without NFS (The same as 3, if you do not have
#   networking)
#   3 - Full multiuser mode
#   4 - unused
#   5 - X11
#   6 - reboot (Do NOT set initdefault to this)
#
id:5:initdefault:
```

which directs your Linux distribution to boot into a graphical login screen. Comment out the line `id:5:initdefault:` and add the line `id:3:initdefault:`, so that you now have something like:

```
#   0 - halt (Do NOT set initdefault to this)
#   1 - Single user mode
#   2 - Multiuser, without NFS (The same as 3, if you do not have
#   networking)
#   3 - Full multiuser mode
#   4 - unused
#   5 - X11
#   6 - reboot (Do NOT set initdefault to this)
#
# id:5:initdefault:
id:3:initdefault:
```

Now if you reboot your system, you will be greeted with a text based login screen. This allows you to log into your system with a text based session from within a virtual terminal. A text based session usually does not consume as much system resources as would be the case with a graphical session. Then build your Sage source distribution from within your text based session. You need to make sure that you can first restore your graphical session, before you attempt to log into a text based session.

2.20 When I start Sage, SELinux complains that "/path/to/libpari-gmp.so.2" requires text-relocation. How can I fix it?

The problem can be fixed by running the following command:

```
$ chcon -t textrel_shlib_t /path/to/libpari-gmp.so.2
```

2.21 Upgrading Sage went fine, but now the banner still shows the old version. How can I fix this?

The banner is stored and not computed at every new start of Sage. If it has not been updated, this should not prevent Sage to run correctly. Type `banner()` in a Sage session to check the real version. If you want the correct banner, you need to build Sage again by typing `make build` in a terminal.

2.22 How do I run sage in daemon mode, i.e. as a service?

There are several possibilities. Use `screen`, `nohup` or `disown`.

2.23 The show command for plotting 3-D objects does not work.

The default live 3-D plotting for Sage 6.4+ uses Jmol/JSmol for viewing. From the command line the Jmol Java application is used, and for in browser viewing either pure javascript or a Java applet is used. By default in browsers pure javascript is used to avoid the problems with some browsers that do not support java applet plugins (namely Chrome). On each browser worksheet there is a checkbox which must be checked before a 3-D plot is generated if the user wants to use the Java applet (the applet is a little faster with complex plots).

The most likely reason for a malfunction is that you do not have a Java Run Time Environment (JRE) installed or you have one older than version 1.7. If things work from the command line another possibility is that your browser does not have the proper plugin to support Java applets (at present, 2014, plugins do not work with most versions of Chrome). Make sure you have installed either the IcedTea browser plugin (for linux see your package manager), see: IcedTea, or the Oracle Java plugin see: Java.

If you are using a Sage server over the web and even javascript rendering does not work, you may have a problem with your browser's javascript engine or have it turned off.

2.24 May I use Sage tools in a commercial environment?

Yes! Absolutely! Basically the *only* constraint is that if you make changes to Sage itself and redistribute this changed version of Sage publicly, then you must make these changes available to us so that we can put them into the standard version of Sage (if we want). Otherwise, you are free to use as many copies of Sage as you want completely for free to make money, etc. without paying any license fees at all.

2.25 I want to write some Cython code that uses finite field arithmetic but "cimport sage.rings.finite_field_givaro" fails. What can I do?

You need to give hints to Sage so that it uses C++ (both Givaro and NTL are C++ libraries), and it also needs the GMP and STDC C++ libraries. Here is a small example:

```
# These comments are hints to Cython about the compiler and
# libraries needed for the Givaro library:
#
# distutils: language = c++
```

(continues on next page)

(continued from previous page)

```
# distutils: libraries = givaro gmpxx gmp m
cimport sage.rings.finite_field_givaro
# Construct a finite field of order 11.
cdef sage.rings.finite_field_givaro.FiniteField_givaro K
K = sage.rings.finite_field_givaro.FiniteField_givaro(11)
print("K is a {}".format(type(K)))
print("K cardinality = {}".format(K.cardinality()))
# Construct two values in the field:
cdef sage.rings.finite_field_givaro.FiniteField_givaroElement x
cdef sage.rings.finite_field_givaro.FiniteField_givaroElement y
x = K(3)
y = K(6)
print("x is a {}".format(type(x)))
print("x = {}".format(x))
print("y = {}".format(y))
print("x has multiplicative order = {}".format(x.multiplicative_order()))
print("y has multiplicative order = {}".format(y.multiplicative_order()))
print("x*y = {}".format(x * y))
# Show that x behaves like a finite field element:
for i in range(1, x.multiplicative_order() + 1):
    print("{} {}".format(i, x**i))
assert x*(1/x) == K.one()
```

To find out more, type

```
sage.rings.finite_field_givaro.FiniteField_givaro.
```

at the Sage prompt and hit tab, then use `??` to get more information on each function. For example:

```
sage.rings.finite_field_givaro.FiniteField_givaro.one??
```

tells you more about the multiplicative unit element in the finite field.

2.26 I'm getting weird build failures on Mac OS X. How do I fix this?

Search the build log (install.log) to see if you are getting the following log message:

```
fork: Resource temporarily unavailable.
```

If so, try the following. Create (or edit) `/etc/launchd.conf` and include the following:

```
limit maxproc 512 2048
```

then reboot. See this page for more details.

2.27 How do I plot the cube root (or other odd roots) for negative input?

This is one of the most frequently asked questions. There are several methods mentioned in the plot documentation, but this one is easiest:

```
sage: plot(real_nth_root(x, 3), (x, -1, 1))
Graphics object consisting of 1 graphics primitive
```

On the other hand, note that the straightforward

```
sage: plot(x^(1/3), (x, -1, 1))  # not tested
```

produces the expected plot only for positive x. The *reason* is that Sage returns complex numbers for odd roots of negative numbers when numerically approximated, which is a standard convention.

```
sage: numerical_approx( (-1)^(1/3) )
0.500000000000000 + 0.866025403784439*I
```

2.28 How do I use the bitwise XOR operator in Sage?

The exclusive or operator in Sage is ^^. This also works for the inplace operator ^^=:

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

If you define two variables and then evaluate as follows:

```
sage: a = 5; b = 8
sage: a.__xor__(b), 13
(13, 13)
```

You can also do

```
sage: (5).__xor__(8)
13
```

The parentheses are necessary so that Sage does not think you have a real number. There are several ways to define a function:

```
sage: xor = lambda x, y: x.__xor__(y)
sage: xor(3, 8)
11
```

Another option, which sneaks around the Sage preparser, is

```
sage: def xor(a, b):
....:     return eval("%s^%s" % (a, b))
sage: xor(3, 8)
11
```

You can also turn off the Sage preparser with `preparser(False)`, then `^` will work just like in Python. You can later turn on the preparser with `preparser(True)`. That only works in command line Sage. In a notebook, switch to Python mode.

2.29 With objects a and b and a function f, I accidentally typed f(a) = b instead of f(a) == b. This returned a TypeError (as expected), but also deleted the object a. Why?

It is because of how functions are defined in Sage with the `f(x) = expr` notation using the preparser. Also notice that if you make this mistake inside of an `if` statement, you will get a `SyntaxError` before anything else goes wrong. So in this case, there is no problem.

2.30 How do I use a different browser with the Sage notebook?

You will need to do this from the command line. Just run a command like this.

- Linux (assuming you have Sage in `/usr/bin`):

```
$ env BROWSER=opera /usr/bin/sage --notebook
```

- Mac (assuming you are in the directory of your downloaded Sage). With the Jupyter notebook:

```
$ BROWSER='open -a Firefox %s' ./sage --notebook jupyter
$ BROWSER='open -a Google\ Chrome %s' ./sage --notebook jupyter
```

2.31 Where is the source code for `<function>`?

Functions and classes written in Python or Cython are in general accessible on the IPython command line with the `??` shortcut:

```
sage: plot??                                # not tested
Signature: plot(*args, **kwds)
Source:
...
```

Objects that are built into Python or IPython are compiled and will not show, however. There are many functions in Sage implemented as symbolic functions, i.e., they can be used unevaluated as part of symbolic expressions. Their source code may also not be accessible from the command line, especially with elementary functions, because they are coded in C++ for efficiency reasons.

CHAPTER

THREE

FAQ: CONTRIBUTING TO SAGE

3.1 How can I start contributing to Sage?

The first step is to use Sage and encourage your friends to use Sage. If you find bugs or confusing documentation along the way, please report your problems!

Two popular ways to contribute to Sage are to write code and to create documentation or tutorials. Some steps in each direction are described below.

3.2 I want to contribute code to Sage. How do I get started?

Take a look at the official development guide for Sage. At a minimum, the first chapter in that guide is required reading for any Sage developer. Also pay special attention to the trac guidelines. You can also join the sage-devel mailing list or hang around on the `#sage-devel` IRC channel on freenode. While you are getting to know the community, grab a copy of the Sage source and familiarize yourself with the git version control system.

The best way to become familiar with the Sage development process is to choose a ticket from the trac server and review the proposed changes contained in that ticket. If you want to implement something, it is a good practice to discuss your ideas on the `sage-devel` mailing list first, so that other developers have a chance to comment on your ideas/proposals. They are pretty open to new ideas, too, as all mathematicians should be.

Sage's main programming language is Python. Some parts of Sage may be written in other languages, especially the components that do the heavy number crunching, but most native functionality is done using Python, including "glue code". One of the good aspects of Python that Sage inherits is that working code is considered more valuable than merely fast code. Fast code is valuable, but clean, readable code is important. In the mathematics community, inaccurate results are unacceptable. Correctness comes before optimization. In the following paper

- D. Knuth. Structured Programming with go to Statements. *ACM Journal Computing Surveys*, 6(4), 1974.

Don Knuth observes that: "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil."

If you do not know Python, you should start learning that language. A good place to start is the Python Official Tutorial and other documents in the Python standard documentation. Another good place to take a look at is Dive Into Python by Mark Pilgrim, which may be pretty helpful on some specific topics such as test-driven development. The book Building Skills in Python by Steven F. Lott is suitable for anyone who is already comfortable with programming.

If you want, you can try to learn Python by using Sage. However, it is helpful to know what is pure Python and when Sage is doing its "magic". There are many things that work in Python but not in Sage, and vice versa. Furthermore, even when the syntax is identical, many programming concepts are explained more thoroughly in Python-centered resources than in Sage-centered resources; in the latter, mathematics is usually the priority.

3.3 I am not a programmer. Is there another way I can help out?

Yes. As with any free open source software project, there are numerous ways in which you could help out within the Sage community, and programming is only one of many ways to contribute.

Many people like writing technical tutorials. One of the joys of doing so is that you also learn something new in the process. At the same time, you communicate your knowledge to beginners, a skill which is useful in fields other than technical writing itself. A main point about technical writing is that you communicate a technical subject to beginners, so keep technical jargon to a minimum. Darrell Anderson has written some tips on technical writing, which we highly recommend.

For the graphic designers or the artistically creative, you can help out with improving the design of the Sage website.

If you can speak, read, and write in another (natural) language, there are many ways in which your contribution would be very valuable to the whole Sage community. Say you know Italian. Then you can write a Sage tutorial in Italian, or help out with translating the official Sage tutorial to Italian.

The above is a very short list. There are many, many more ways in which you can help out. Feel free to send an email to the sage-devel mailing list to ask about possible ways in which you could help out, or to suggest a project idea.

3.4 Where can I find resources on Python or Cython?

Here is an incomplete list of resources on Python and Cython. Further resources can be found by a web search.

General resources

- Cython
- pep8
- pydeps
- pycallgraph
- PyChecker
- PyFlakes
- Pylint
- Python home page and the Python standard documentation
- Snakefood
- Sphinx
- XDot

Tutorials and books

- Cython Tutorial by Stefan Behnel, Robert W. Bradshaw, and Dag Sverre Seljebotn
- Dive Into Python 3 by Mark Pilgrim
- Fast Numerical Computations with Cython by Dag Sverre Seljebotn
- Official Python Tutorial

Articles and HOWTOs

- decorator
- Functional Programming HOWTO by A. M. Kuchling

- Python Functional Programming for Mathematicians by Minh Van Nguyen
- Regular Expression HOWTO by A. M. Kuchling
- reStructuredText

3.5 Are there any coding conventions I need to follow?

You should follow the standard Python conventions as documented at **PEP 8** and **PEP 257**. Also consult the Sage Developer's Guide, especially the chapter Conventions for Coding in Sage.

3.6 I submitted a bug fix to the trac server several weeks ago. Why are you ignoring my branch?

We are not trying to ignore your branch. Most people who work on Sage do so in their free time. With hundreds of open tickets of varying degrees of impacts on the whole Sage community, people who work on tickets need to prioritize their time and work on those tickets that interest them. Sometimes you may be the only person who understands your branch. In that case, you are encouraged to take extra care to make it as easy as possible for anyone to review. Here are some tips on making your branch easy to review:

- Have you clearly described the problem your branch is trying to solve?
- Have you provided any background information relevant to the problem your patch is trying to solve? Such information include links to online resources and any relevant papers, books and reference materials.
- Have you clearly described how your branch solves the problem under consideration?
- Have you clearly described how to test the changes in your branch?
- Have you listed any tickets that your branch depends on?
- Is your branch based on a recent (preferably, the latest) Sage beta version?
- Does your branch follow relevant conventions as documented in the Developer's Guide?

If your branch stands no chance of being merged in the Sage source tree, we will not ignore your branch but simply close the relevant ticket with an explanation why we cannot include your changes.

3.7 When and how might I remind the Sage community of a branch I care about?

You are encouraged to take extra care in how you remind the Sage community of a branch/patch you want to get merged into the Sage source tree. There might be an upcoming bug squash sprint or an upcoming Sage Days workshop that relates to your patch. Monitor the relevant Sage mailing lists and respond politely to any relevant email threads, with clear explanation on why your patch is relevant.

3.8 I wrote some Sage code and I want it to be integrated into Sage. However, after renaming my file `a.sage` to `a.py`, I got syntax errors. Do I have to rewrite all my code in Python instead of Sage?

The basic answer is yes, but rewriting is a big word for what is really needed. There is little work to do since Sage mostly follows Python syntax. The two main differences are handling of integer (see also the afterword for more on the sage preparser), and the necessity to import what you need.

- **Handling of integers:** You need to take care of the following changes:
  - Notation for exponentiation: In Python `**` means exponentiation and `^` means "xor".
  - If you need to return an integer to the user, write `return Integer(1)` instead of `return 1`. In Python, 1 is a python `int`, and `Integer(1)` is a Sage/Gmp integer. In addition, `Integer` are much more powerful than `int`; for example, they know about being prime and factorization.
  - You should also notice that `2 / 3` no longer means `Integer(2) / Integer(3)` and returns `2/3`, but rather `int(2) / int(3)`, and therefore returns `0` due to integer division where it deregards any remainder. If you are dealing with `Integer` but you really need an integer division you can use `Integer(2) // Integer(3)`.
- **Importing stuff:** The second big change is the necessity to import everything what you need. More precisely, each time you use some Sage function, you need to import it at the beginning of the file. For example, if you want `PolynomialRing`, you need to write:

```
from sage.rings.polynomial.polynomial_ring_constructor import PolynomialRing
```

  You can use `import_statements` to get the exact necessary line:

```
sage: import_statements(PolynomialRing)
from sage.rings.polynomial.polynomial_ring_constructor import PolynomialRing
```

  If this fails, you can ask Sage where to find `PolynomialRing` using:

```
sage: PolynomialRing.__module__
'sage.rings.polynomial.polynomial_ring_constructor'
```

  This also corresponds to the path starting after `site-packages` given when you ask Sage for `PolynomialRing` help. For example, if you call `PolynomialRing?`, you get:

```
Type:     function
[...]
File:     /path_to_sage_root/sage/local/lib/python3.7/site-packages/sage/rings/
↪polynomial/polynomial_ring_constructor.py
[...]
```

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

P