
Sage Developer's Guide

Release 4.8

The Sage Development Team

January 20, 2012

CONTENTS

1	Walking Through the Development Process	3
1.1	Modifying Sage source code	3
1.2	Creating a sandbox	4
1.3	Reviewing a patch	5
1.4	Creating a change	6
1.5	Submitting a change	6
1.6	Updating a change	7
1.7	Being more efficient: Mercurial queues	8
1.8	Starting with Mercurial queues	8
1.9	Reviewing patches with queues	9
1.10	Creating your own patch with queues	10
1.11	Upgrading Sage with queues present	10
1.12	The Big Picture for Mercurial queues	11
1.13	More on queues	12
1.14	Cherry picking	12
1.15	More about Mercurial	12
2	Writing Code for Sage	13
2.1	Conventions for Coding in Sage	13
2.2	Coding in Python for Sage	27
2.3	Coding in Cython	34
2.4	Coding using external libraries and interfaces	40
2.5	Doctesting the Sage Library	51
2.6	The Sage Manuals	61
3	Disseminating Code for Sage	65
3.1	Inclusion Procedure for New Packages	65
3.2	Producing Patches with Mercurial	66
3.3	Producing New Sage Packages	68
3.4	Patching a Sage Package	73
3.5	The Sage Trac Server: Submitting Patches and Packages	76
4	Indices and tables	83
	Index	85

Sage is a free mathematics software system. It is implemented using Python, Cython, and C++, together with various packages such as GAP, GSL, Matplotlib, Maxima, MWRANK, NetworkX, NTL, Numpy, PARI, Singular and many specialized systems and libraries. Sage is free and open source, and is available under the terms of the GNU Public License. Some parts are available under compatible licenses.

Everybody who uses Sage is encouraged to contribute something back to Sage at some point. Implement a new function, add examples to the documentation, find bugs and typos, fix a bug, create a new class, create a fast new C library, etc. This book is a guide on how to contribute to Sage.

This document describes how to write programs using Sage, how to modify and extend the core Sage libraries, and how to modify Sage's documentation. It also discusses how to share your new and modified code with other Sage users.

This work is licensed under a [Creative Commons Attribution-Share Alike 3.0 License](#).

WALKING THROUGH THE DEVELOPMENT PROCESS

This section presents a general process for working on Sage. We will walk through each step of the process, explaining specific steps and commands to help you get started on developing Sage. Along the way, other sources of information will be presented where you could find more detailed information on some particular issue. If you are a beginner to Sage development, this introductory guide is here to help you become familiar with the Sage development process.

1.1 Modifying Sage source code

The name `SAGE_ROOT` refers to the directory where Sage is installed on your system, e.g. `/sage/sage-4.1.2/`. Some people install Sage under their home directory, in which case `SAGE_ROOT` would be something like `/home/username/sage-4.3.1`. Software packages that are shipped with Sage would then be located under `/home/username/sage-4.3.1`. In other words, `SAGE_ROOT` is the top-level directory of your Sage installation.

The majority of Sage code for development work is under `SAGE_ROOT/devel`, where the symbolic link `SAGE_ROOT/devel/sage` points to the current branch that you are using. The default branch is the branch named `sage-main` and is also referred to as the main branch. By default, the symbolic link `SAGE_ROOT/devel/sage` points to `SAGE_ROOT/devel/sage-main`. Under each branch is a directory named `sage`, which contains code for interfacing with third-party packages and code that comprises the Sage library. You can think of the directory `SAGE_ROOT/devel/sage-main/sage` (or `SAGE_ROOT/devel/sage/sage`) as where the Sage library resides.

Suppose you have modified some part of the Sage library, e.g. add new code or delete some code. How do you get Sage to know about your changes? You need to rebuild the Sage library so that it is updated with your changes. Navigate to `SAGE_ROOT` and rebuild the library as follows:

```
./sage -b main
```

If `SAGE_ROOT/devel/sage` points to the main branch, you could use the above command or more simply the following command:

```
./sage -b
```

The switch `-b` is for (re)building the Sage library and the argument `main` is the branch you want to rebuild. In this case, you want to rebuild the Sage library as contained in the main branch, i.e. under `SAGE_ROOT/devel/sage-main`. In fact, the command `./sage -b` rebuilds whatever branch that the symbolic link `SAGE_ROOT/devel/sage` points to. Any of the above two commands does not rebuild everything in the Sage library, but only those files in the library that your changes affect. During the rebuild process, the affected source files are copied elsewhere and compiled, Cython files get converted to C code and compiled, and so on.

This build process will copy the necessary files to `SAGE_ROOT/devel/sage/build` and `SAGE_ROOT/local/lib/python/site-packages`. It is very important that you **do not change files** in these directories directly! If you do that, then those files would get overwritten the next time you run `sage -b`.

You may want to create a totally distinct installation of Sage compiled from source, where you will work so as to not impact a version of Sage that you actually use. This could be true even if you make clones (or sandboxes) as described in the next section.

1.2 Creating a sandbox

With `SAGE_ROOT` as the default directory, you can create a new branch (or clone) named `test` with the command:

```
./sage -clone test
```

Note that a general way to create a clone is the syntax `./sage -clone <branch-name>`, where you should replace `<branch-name>` with the name you want to name your new clone. Creating a clone may take a couple of minutes to conclude. The result is a whole new subdirectory `SAGE_ROOT/devel/sage-test` where the name `sage-` is automatically prepended to `test`, and the symbolic link `SAGE_ROOT/devel/sage` will now point to this new subdirectory. Running Sage, or rebuilding, will use this version of the source.

Now you can safely experiment with Sage code as much as you would like. A cloned version of the Sage library can be found under

```
SAGE_ROOT/devel/sage-test/sage
```

The Sage library consists of Python files (with the file extension `.py`) or Cython files (with extension `.pyx`), organized into subdirectories that mirror the module hierarchy reflected in the help files, Python types, etc. (With the `sage` symbolic link, you will now see why sometimes you see directories that look like `../sage/sage/...`)

Apart from the directory that holds the Sage library, another special directory is

```
SAGE_ROOT/devel/sage-test/doc/en/reference
```

You can think of that directory as containing “table of contents” files for documentation, each such file having the extension `.rst` to indicate that they follow the ReST format for documenting source code. Most documentation comes from the docstrings in the source files. When you build the HTML or PDF versions of the reference manual, the generated output lands in subdirectories of `SAGE_ROOT/devel/sage-test/doc/output`.

To make another clone of the main branch, you need to first switch back to the untouched code in the main branch. From `SAGE_ROOT`, type

```
./sage -b main
```

to switch back to the original version, i.e. the main branch. Now make a new clone as shown above.

If a clone, say `test`, is a mess and has nothing of value to you, switch back to the main branch, then delete the directory `SAGE_ROOT/devel/sage-test` and everything below it. To know which branch (or clone) you are using, issue the command

```
./sage -branch
```

This will report the current branch that the symbolic link `SAGE_ROOT/devel/sage` points to. Such information is also reported by the command `./sage` when the current branch is not the main branch.

There are many more arguments you can pass to Sage. For a list of basic arguments, execute

```
./sage -help
```

The command

```
./sage -advanced
```

will report a list of advanced arguments in addition to the list of basic arguments as output by `./sage -help`.

1.3 Reviewing a patch

An essential part of improving Sage is reviewing the patches that other developers have written; before any patch is accepted into Sage, it needs the green light from another developer. This process catches many small bugs and omissions that could otherwise easily sneak into such a large project, even if it does slow down the development process.

This section goes through the process of downloading and applying patches. For detailed information on the review process, see the section *Reviewing patches*. William Stein also wrote this [blog entry](#) on his patch reviewing workflow.

When viewing a Trac ticket, you will see available patches listed near the top in the Attachments section. Clicking on the file name will show you a *diff* comparison view that is useful for previewing changes; red shading is deletions, green shading is additions. Once you've decided to review the patch, download the patch file from the Trac server using the little download icon next to the file name.

To apply a patch to the code in your sandbox (see *Creating a sandbox* for information on creating a sandbox), follow these steps:

1. Run Sage: from `SAGE_ROOT`, type `./sage`.
2. Apply the patch: at the Sage command line, type:


```
hg_sage.apply("<full-path-and-filename.patch>")
```
3. Quit Sage: use the command `exit`.
4. Rebuild Sage: use the command `./sage -b` to rebuild the affected files in the Sage library.

In step 2, you are using Sage's simplified interface to the *Mercurial* revision control system. This command will add the patch as a new "changeset" and "commit" the changes. At the Sage command line, you can run `hg_sage.log()` to see before/after changes to the Sage library. In step 4, you should only see a few files copied, modified, etc. Unaffected files should not be part of this step. Look for compilation errors in this output and modify your changes as appropriate. Avoid producing patches that result in compilation errors or errors in building the documentation. (You want a working Sage installation, right?) See also the section *Reviewing patches with queues* for another way to apply patches with *Mercurial*.

To actually test out a patch, do the following. Even if you're new to Sage development and tentative about reviewing, reporting the success or failure of these steps on the ticket Trac page will be quite helpful:

1. Experiment with the functionality proposed by the patch. Verify results are correct by hand computations, test bad input, outrageous situations, etc.
2. Run tests on the affected files. From `SAGE_ROOT`, issue the command `./sage -t devel/sage-test/path-to-directory-or-file` to run doctests on the affected file(s). Failures should be reported on the ticket and are reason to move the ticket to "needs work". (See *Doctesting the Sage Library* for more information.)
3. Test the entire Sage library, including `#long` doctests. From `SAGE_ROOT`, issue the command `./sage --testall --long`. (See *Parallel testing the whole Sage library* for information about testing in parallel.) This will take a while to complete. No, it is not optional. It is entirely possible that changes to one part of Sage break something in an entirely different part of Sage; patches which introduce new doctest failures cannot be included in new releases.

4. Ensure that the documentation builds. From `SAGE_ROOT`, run `./sage -docbuild reference html`, which will build the HTML version of the documentation. Check the “look” of affected files in the output directory for the documentation (see above).
5. Check for full doctest coverage. From `SAGE_ROOT`, run `./sage -coverage <file>` which will provide a complete report. Less than 100% coverage is another reason to return a patch to “needs work” status.
6. Look at *Reviewing patches* for more guidelines on reviewing.

Once you've tested the patch, either by hand or with doctests (or both!), report any failures on the Trac page for the ticket. Make suggestions about simplifying the code or fixing typos you noticed. Mark it as “needs work” if there is anything to do. Otherwise, mark it as “positive review”, and mention in a comment all the things you checked. If you don't feel experienced enough for that, add a comment on the Trac page explaining what you have checked, what the results were, and that you think someone more experienced should take a look.

Tickets relating to doctests or the Sage documentation may be a good place to get started reviewing. A list of such tickets with patches ready for review can be found using the following Trac [custom query](#). Alternatively, you can search for tickets needing review in a component whose mathematics you know well, or for tickets needing review which have priority “minor” or “trivial”.

1.4 Creating a change

To make a change to Sage (fix a bug, add new functionality), proceed as follows:

1. Make a fresh clone, as discussed in *Creating a sandbox*.
2. Apply any precursor patches not in your current version, as demonstrated in *Reviewing a patch*.
3. Edit source files (see *Modifying Sage source code* for location), test building Sage, test functionality, and so on.
4. Once you have something you like, do everything suggested for reviewing a patch. It is a waste of time for a reviewer to start on reviewing a patch and find that tests fail, documentation was not tested, etc. It would save any reviewer a lot of time if your patches have been fully tested before you submit them for review. Everybody makes mistakes, everybody has bugs they did not anticipate, and everybody writes code that can be improved—that is why there are code reviews. But do not cut corners.

1.5 Submitting a change

Here is how to prepare a patch with your changes:

1. Register for a Trac account at the URL http://trac.sagemath.org/sage_trac/register. If you have problems with the registration process, please refer to the page <http://www.sagemath.org/contact.html> for the relevant person to contact about your registration issues. Most people use some variant of their real name, especially if they already have a reputation within mathematics. Edit the main Trac page where there is a list of developers and add yourself with a link to your web page. Make sure to sort your Trac username alphabetically.
2. If it does not already exist, make a Trac ticket for your changes. Provide a one-line summary and then a description of the problem. Include a link to a sage-devel discussion if appropriate. Choose a component, if this is a defect or enhancement, set your real name in the author field. It works well if you have your Trac settings such that you get an email every time the ticket changes. Make a note of the ticket number.
3. Create a `.hgrc` Mercurial configuration file in your home directory. Specify your name and email address here, so it will identify you as the author of a patch, in the form “ Bill Smith <bsmith@bigu.edu>“. Here is a template for your `.hgrc` file:

```
[ui]
username = Carl Friedrich Gauss <cfgauss@uni-goettingen.de>
```

```
[extensions]
# Enable the Mercurial queue extension.
hgext.mq =
```

The Mercurial project website <http://mercurial.selenic.com> contains many tutorials on using Mercurial.

4. If necessary, first switch to the branch holding your changes. From the Sage command line interface, run `hg_sage.status()`. The output will be a list of modified files, preceded by a capital M. Check that this is what you expect. For explanation of other letters, see the Mercurial documentation on the `hg status` command.
5. From the Sage command line, run `hg_sage.diff()`. This will show you the changes you have made. A plus sign is new code being added, a minus sign is code being deleted. This should look like the changes you have made.
6. Now run `hg_sage.commit()` from the Sage command line. This will package your changes as a single Mercurial “changeset”, allowing others (reviewers, release manager) to add your changes to their versions of Sage. An editor window will pop up (set your favorite editor in the `.hgrc` file mentioned above) where you should enter a *one-line* message describing the patch. This message is known as the commit message for your patch. You are encouraged to write commit messages of the form `Trac XXXX: <description-goes-here>` using the Trac ticket number and then have a concise description, e.g. “fix echelon form error” or “add echelon form over finite fields.” Some people also write commit messages in the form `#xxxx: <description-goes-here>`, which is also acceptable. A key information to provide in a commit message is the ticket number.
7. Run the command `hg_sage.log()` from the Sage command line. The first entry should be your changeset. Note the changeset number, which is probably 5 decimal digits.
8. Next, issue the command

```
hg_sage.export(<changeset-number>, "/path-to-somewhere/trac_XXXX_short_descriptor.patch")
```

where `short_descriptor` is really short, like `echelon_form_fix` or at most `finite_field_echelon_form`.

9. You can preview your patch using a “diff viewer”. Some people use `kompare` on Linux, others use `kdiff3`.
10. Upload your patch to the Trac server.
11. Feel free to CC another developer (use their Trac username from the list on the main page) if you think they might be able to review your change. If somebody else originated, or commented on the Trac ticket, they will be notified of your change if they have set Trac to email them of any changes.

1.6 Updating a change

Your first patch would likely have a review that suggests changes. Here is one way to update your patch. (There is probably a better way, but the following steps should be easy to follow.)

1. Make a new fresh clone. Read *Creating a sandbox* to be sure you clone the right stuff (i.e. do not clone the branch you changed). We will call this clone `test2` here.
2. Apply your patch, but not with `hg_sage.apply()`. You want to make the changes without doing a commit. (There is a switch that will prevent a commit, but by doing this, you will see how to do this at the system level.)

First make `SAGE_ROOT/devel/sage-test2/` your working directory. Then at the system command line, run:

```
patch -p1 </path-to-somewhere/trac_XXXX_short_descriptor.patch
```

which will be like you just edited the source files with all the changes from your original patch. Now you can edit to reflect a reviewer's suggestions and prepare a new patch.

3. When you upload to Trac, you can replace the file with one of the same name. The comments will include an indication of when the upload happened, so nobody will be confused about when the replacement happened.

1.7 Being more efficient: Mercurial queues

If you are new to Sage development, the material above is sufficient to allow you to participate and contribute. So on a first exposure, right here is a good place to stop reading and start developing. However, soon you will have a submitted patch that needs repeated revisions, or you will find yourself in the middle of creating a patch and also desiring to review a patch, or you are creating a new patch and uncover a separate bug that you want to isolate and fix independently. When you reach this point come back here and read the remainder about Mercurial queues.

Making a new clone for every review and for each revision to a patch is a method that is easy to understand. But it soon feels rather inefficient. Mercurial queues use Sage's Mercurial repository for tracking, collecting and organizing changes to Sage but are much more flexible and fit better with the way a typical Sage developer needs to work. You will find them much more natural and useful than repeatedly making clones and adjusting patches in response to reviews. And you don't need to learn anything about Mercurial itself before you can begin.

In a nutshell, Mercurial queues are two stacks of patches (a "patch" here means "a named collection of changes"). One stack is a sequence of patches applied in the order in the stack. The other stack is a set of patches that are "out of the way", but still arranged in some order. Simple commands then allow you to move a patch off the top of one stack to the top of the other (in either direction).

We will describe first how to get started, then three typical activities will show how to use the basic commands. We then finish with a "big picture" view, which you can read first if your mind works that way.

1.8 Starting with Mercurial queues

You might find it most convenient to install a fresh copy of Sage as your "development" version. Instead of a version number for the `SAGE_ROOT` directory you might name it something like `dev`. Since you will likely keep it constantly upgraded, a version number will not make much sense. Mine lives at `/sage/dev` with other copies right alongside in places like `/sage/sage-4.3.1/`.

You also need to install a copy of Mercurial on your system, since now you will not be using Sage's simplified interface. So use a package manager, or whatever, to install a recent version. Alternatively, you could also use the version of Mercurial that is shipped with Sage. To do so, from within your terminal window, create an alias similar to the following:

```
alias 'hg'='/path/to/SAGE_ROOT/sage -hg'
```

Afterwards, any time you execute the command `hg` in your terminal, this would invoke the version of Mercurial that is installed with your local Sage installation. However, once you quit or close your terminal, the alias `hg` would be lost and would no longer point to the Mercurial installation that comes with Sage.

You first need to "turn on" support for queues. It is all built-in by default but not visible. Edit your `~/.hgrc` file (in your home directory, where your user info is) and add the following stanza if it is not present:

```
[extensions]
# Enable the Mercurial queue extension.
hgext.mq =
```

The main Mercurial repository for the Sage library lives in `SAGE_ROOT/devel/sage` so you will interact with the repository using this as the working directory. This means to actually run Sage, or rebuild it, you will use commands such as `../../sage -b`. (The next step, initialization, has been deprecated as of version 1.5.) One time only, for each repository, you need to initialize it for use with queues, so do the following:

```
cd SAGE_ROOT/devel/sage
hg qinit
```

HG is the symbol for the chemical element mercury, so `hg` is the executable. All the commands specific to queues begin with the letter “q”. That’s all the setup, you are ready to go. The command `hg help mq` will give a summary of queue commands. A command like `hg help qpop` will give documentation for using the `qpop` command.

1.9 Reviewing patches with queues

The two stacks used by queues are called “applied” and “unapplied”. The names do a good job of describing the status of the patches in each. Download a patch from Trac as described above (*Reviewing a patch*) in the usual way. Then execute

```
hg qimport <path-and-filename.patch>
```

This will add the patch to the top of your unapplied stack. Use `hg qunapplied` to verify that the patch is in this stack. Presumably you want to have the changes in this patch applied to your Sage library, so use the simple command `hg qpush` to accomplish this. Now issue the command `hg qapplied` to see the patch now present in the applied stack. You can now rebuild Sage, run the modified version, run tests, build documentation and so on, as described above.

Let’s suppose the patch you were reviewing was so bad Sage wouldn’t even build due to compiler errors. So you have the time to review something else. Let’s move the first patch out of the way. The command `hg qpop` will move the top patch in the applied queue over to the top of the unapplied queue, so you would be able to apply and review other patches. Use `hg qapplied` and `hg qunapplied` to verify this movement. Now download a new patch, `hg qimport` it, and `hg qpush` to apply it.

Suppose this second patch turned out to be too far beyond your expertise in a certain area of mathematics or programming. Pop it off the applied stack with `hg qpop` so it is now at the top of the unapplied stack, sitting on top of the the un-compilable patch (you haven’t forgotten that one, have you?). Use

```
hg qdelete <patch-name>
```

to totally get rid of it. Bye-bye. In the meantime, the author of the first patch found the single little error that prevented the patch from compiling and has posted a very small patch to make the correction. First, apply the original patch again with `hg qpush`, then download the small patch with the fix, use `hg qimport` to get it onto the unapplied stack, then finally `hg qpush` to apply it on top of the buggy patch. Now you should be able to compile, experiment and test as usual with both patches applied.

So we see you can use `hg qimport` and `hg qdelete` to move patches in and out, `hg qpop` and `hg qpush` to move patches between applied and unapplied states (stacks). Keep track of where you are with liberal use of `hg qapplied` and `hg qunapplied`.

You may be wondering what to do if your patches in the stacks end up “out of order.” We’ll cover that in a bit.

1.10 Creating your own patch with queues

Let's suppose you are ready to make some changes to the Sage library of your own. Put anything in the applied stack that you need to build on, get everything else out of the way on the unapplied stack (see *Reviewing patches with queues* for techniques). Issue

```
hg qnew <descriptive-name>
```

I am always in such a rush, I often forget this step. If you are like me, then

```
hg qnew -f <descriptive-name>
```

will capture your changes made so far and give you a patch to work with. (In newer versions of Mercurial, the `-f` flag has been deprecated.) The “descriptive-name” can be anything you like, nobody else ever has to see it. Use a Trac ticket number or whatever you please. Edit, build, test, create documentation, knock yourself out. At any time, run `hg qdiff` to see your changes.

Once satisfied with your work, use `hg qrefresh` to save your changes into the patch. Even better is to use the `-e` or `-m` switches to allow you to edit (or specify) a summary line for the patch. This was described above as the commit message. The use of `-m` is illustrated below. To create a patch file in the proper format for submission to Trac, you need a generic Mercurial command. Your patch is at the “tip” of the Mercurial repository and you want to export it, with redirection to a file.

```
hg qrefresh -m "Trac 1234: modified matrix memory management mostly"  
hg export tip > ~/sage-patches/trac_1234_matrix_memory.patch
```

Now upload this to the ticket in the usual way. Note the message in the `-m` switch is what others will see as a description of your patch, not the name you used in `hg qnew` initially. *Do not use* `hg qfinish` when you think a patch is done, despite the pleasing sounding name. It will finalize your patch, add it into the main repository, remove it from your queues, and generally make it much harder to get back to with subsequent edits based on reviewer comments. Your work is not gone, but it will take a few steps to get it out as a patch and back into the queues. You might want to read up on the command `hg qimport -r` as a possible way to undo an accidental commit.

Of course, the minute you upload, you get a better idea about a key step in your algorithm. Simple—edit some more, then `hg qrefresh` (the message stays put, so you don't have to redo it), and `hg export tip > <filename>`. You can use a new filename, or recycle the previous one. Trac will let you add a new file, or replace the existing one with a file having the same name.

Suppose a reviewer suggests some changes. You can just keep editing the same patch, or you could `hg qnew` a second patch on top of the old one. It would depend on circumstances, there are situations where either approach would make sense.

Suppose it takes a while for a reviewer to look at your patch. Move it off into the unapplied stack with `hg qpop` and then begin a new project with `hg qnew <another-name>`. Or leave your patch in the applied queue and start something new that relies on your first set of changes (again using `hg qnew`).

So the sequence `hg qnew`, `hg qrefresh`, `hg export tip >` will create a new patch and allow you to easily amend or extend it, or totally move it “out of the way” to do other things.

1.11 Upgrading Sage with queues present

When it is time to upgrade Sage to the latest release, you need to return your development version back to a virgin state. Use `hg qrefresh` on whatever patch you are currently creating (if any). Then pop everything off the applied stack with `hg qpop -a`, where the switch `-a` means “all.” There you are, back to a known good state. Now use the standard commands to upgrade Sage:

```
cd SAGE_ROOT
./sage -upgrade
```

Sometimes for intermediate releases you will need a URL as an argument to the `-upgrade` switch. Check the Sage discussion groups, where these locations are typically announced. Now you can `hg qpush` to put all your patches back onto the applied stack in the same order. Realize, however, that the upgrade may have changed some of the source code where your patches have changes. Certainly, if you have patches you reviewed positively, those exact changes may already be present (so at least `hg qdelete` those patches before pushing everything back on).

It is also possible to use the Mercurial extension “rebase” to manage patches through an upgrade process. First you must enable the extension by editing your `.hgrc` file to include the following:

```
[extensions]
rebase=
```

Now, instead of using `hg qpush` to forcibly reapply the old patches to the new Sage version, you can do the following.

1. Do `hg update -C -r old.version.number`, where `old.version.number` is the old Sage version number you upgraded from, in order to get your working directory back to the good state of the old Sage version;
2. `hg qpush -a` to reapply all your patches to the old Sage files;
3. `hg heads` to find out the exact revision number you want to rebase your patch on;
4. `hg rebase -d other.head.rev.number`, where `other.head.rev.number` is the number you just looked up.

Now your patches should be properly rebased on the new version of Sage. It will sometimes happen that the “rebase” extension can’t quite figure out how to rebase some changes. In that case, Mercurial will automatically open a 3-way merge editor to enlist your help in resolving the problem. You can configure which program this will be by editing your `.hgrc` file.

1.12 The Big Picture for Mercurial queues

At some time when you have a few patches applied, and a few unapplied, run

```
cd SAGE_ROOT/devel/sage
hg qapplied
hg qunapplied
cat .hg/patches/series
```

The output of the two `hg` commands should together look just like the output of the `cat` command. The `.hg/patches/series` file has all of the names of your patches in some order, and you can imagine a separator that splits the list into the applied portion at the start of the file and the unapplied portion at the end of the file. (You can also use the command `hg qseries` to see a list of all your patches.) The top of each stack is on either side of the separator. (So the order of each stack runs in opposite directions in this file.)

The command `hg qpush` moves the separator toward the end of the file, while `hg qpop` moves the separator toward the start of the file. Furthermore, `hg qnew` inserts a new patch on the side of the separator toward the start of the file, while `hg qimport` adds an existing patch on the side of the separator toward the end of the file. Finally, `hg qdelete` totally removes a name from the series file.

So what if you want to rearrange the order of your patches (in either stack)? Make sure to `hg qpop` until all the affected patches are in the unapplied stack. Open `.hg/patches/series` with a text editor and rearrange the lines below the imaginary separator. Save the series file and confirm the new ordering with `hg qapplied` and `hg qunapplied`. Then `hg qpush` repeatedly to get to where you want to be.

1.13 More on queues

So with careful management of your queues and regular upgrades, you can contribute to Sage easily, review others' patches, work on several projects simultaneously, and so on, all with just a single copy of Sage devoted to development.

If you know how “regular” Mercurial functions (and even if you don't) you can look at the main Mercurial repository (with `hg log | more`) and see how queues “insert” your applied patches near the tip of the repository, all “behind the scenes.”

There are lots more you can do with queues, but you should understand enough now to experiment safely. The following URLs contain introductory tutorials on using Mercurial queues:

- <http://mercurial.selenic.com/wiki/MqExtension>
- <http://wiki.sagemath.org/MercurialQueues>
- https://developer.mozilla.org/en/Mercurial_Queues

1.14 Cherry picking

The “record” extensions allow you to selectively pick (record) portions of a patch to group together. (Also known as “cherry picking.”) So you can round up related bits and pieces of a patch if that makes sense in the context of your work. To enable this feature, just edit your `.hgrc` file to include

```
[extensions]
hgext.record=
```

Use the command `hg record` with “regular” Mercurial and `hg qrecord` if you are working with Mercurial queues. The use of the two is slightly different. We will illustrate the use of `qrecord`. The command

```
hg qrecord another-patch
```

creates a new, empty patch at the top of the applied stack. It then begins to interactively examine your changes at the granularity of a patch “hunk.” You can then choose to include each “hunk” of changes into this new patch or not. Then you can work with this patch as before with `hg qrefresh`, `hg qpop`, etc.

For more on `record`, `qrecord` and `crecord`, see

- <http://mercurial.selenic.com/wiki/RecordExtension>

1.15 More about Mercurial

The online book [Mercurial: The Definitive Guide](#) by Bryan O'Sullivan contains numerous examples on using Mercurial. See especially Chapters 12 and 13 for explanation on how to effectively use Mercurial queues.

WRITING CODE FOR SAGE

If there is something you would like to implement and make available in Sage, you have a wide range of options:

1. Implement it as Sage scripts.
2. Implement it as Python scripts that use the Sage library.
3. Implement it in C/C++ and make the result accessible to Sage using Cython.
4. Implement it using Cython.
5. Implement it using one or more of the following: Flint, FpLLL, GAP, GSL, IML, LinBox, M4RI, Matplotlib, Maxima, MWRank, ECLib, NetworkX, NTL, Numpy, PARI/GP, PolyBoRi, R, Scipy, Singular, Sympy or any of the other libraries included with Sage ¹.
6. Or any combination of the above.

If you have Magma, Maple or Mathematica and do not mind restricting who can use your code, you could also implement parts of your program in one of these systems and make it available in Sage.

Flint, FpLLL, GAP, GSL, IML, LinBox, M4RI, Matplotlib, Maxima, MWRank, ECLib, NetworkX, NTL, Numpy, PARI/GP, PolyBoRi, R, Scipy, Singular, Sympy are all included with all distributions of Sage. GAP, Singular, and PARI are very mature, and each implements a great amount of functionality, though in different domains. GAP addresses group theory well, Singular attacks polynomial computation, and PARI contains sophisticated, optimized number theory algorithms. Notably absent from this triad is a good system for exact linear algebra (something Magma does extremely well), but this gap is being filled by code written for Sage or covered by specialized C/C++ libraries like LinBox, IML and M4RI.

Sage is not just about gathering together functionality. It is about providing a clear, systematic and consistent way to access a large number of algorithms, in a coherent framework that makes sense mathematically. In the design of Sage, the semantics of objects, the definitions, etc., are informed by how the corresponding objects are used in everyday mathematics.

This document was authored by William Stein, David Joyner, John Palmieri and others with the editorial help of Iftikhar Burhanuddin and Martin Albrecht.

Contents:

2.1 Conventions for Coding in Sage

To meet the goal of making Sage easy to read, maintain, and improve, all Python/Cython code that is included with Sage should adhere to the style conventions discussed in this chapter.

¹ See <http://www.sagemath.org/links-components.html> for a full list of packages shipped with every copy of Sage

2.1.1 Python coding conventions

Follow the standard Python formatting rules when writing code for Sage, as explained at the following URLs:

- <http://www.python.org/dev/peps/pep-0008>
- <http://www.python.org/dev/peps/pep-0257>

In particular,

- Use 4 spaces for indentation levels. Do not use tabs as they can result in indentation confusion. Most editors have a feature that will insert 4 spaces when the tab key is hit. Also, many editors will automatically search/replace leading tabs with 4 spaces.
- Use all lowercase function names with words separated by underscores. For example, you are encouraged to write Python functions using the naming convention

```
def set_some_value()
```

instead of the CamelCase convention

```
def SetSomeValue()
```

- Use CamelCase for class names and major functions that create objects, e.g. `PolynomialRing`.

Note, however, that some functions do have uppercase letters where it makes sense. For instance, the function for lattice reduction by the LLL algorithm is called `Matrix_integer_dense.LLL`.

2.1.2 File and directory names

Python Sage library code uses the following conventions. Directory names may be plural (e.g. `rings`) and file names are almost always singular (e.g. `polynomial_ring.py`). Note that the file `polynomial_ring.py` might still contain definitions of several different types of polynomial rings.

2.1.3 An example is worth a thousand words

For all of the conventions discussed here, you can find many examples in the Sage library. Browsing through the code is helpful, but so is searching: the functions `search_src`, `search_def`, and `search_doc` are worth knowing about. Briefly, from the “sage:” prompt, `search_src(string)` searches Sage library code for the string `string`. The command `search_def(string)` does a similar search, but restricted to function definitions, while `search_doc(string)` searches the Sage documentation. See their docstrings for more information and more options.

2.1.4 Headings of Sage library code files

The top of each Sage code file should follow this format:

```
r"""
<Very short 1-line summary>

<Paragraph description>
...

AUTHORS:

- YOUR NAME (2005-01-03): initial version
```

```
- person (date in ISO year-month-day format): short desc
...
- person (date in ISO year-month-day format): short desc
...
```

Lots and lots of examples.

```
"""
```

```
#*****
#      Copyright (C) 2010 YOUR NAME <your email>
#
#  Distributed under the terms of the GNU General Public License (GPL)
#  as published by the Free Software Foundation; either version 2 of
#  the License, or (at your option) any later version.
#      http://www.gnu.org/licenses/
#*****
```

The following is the top of the file `SAGE_ROOT/devel/sage/sage/rings/integer.pyx`, which contains the implementation for \mathbb{Z} .

```
r"""
```

```
Elements of the ring '\ZZ' of integers
```

```
AUTHORS:
```

- William Stein (2005): initial version
- Gonzalo Tornaria (2006-03-02): vastly improved python/GMP conversion; hashing
- Didier Deshommes (2006-03-06): numerous examples and docstrings
- William Stein (2006-03-31): changes to reflect GMP bug fixes
- William Stein (2006-04-14): added GMP factorial method (since it's now very fast).
- David Harvey (2006-09-15): added `nth_root`, `exact_log`
- David Harvey (2006-09-16): attempt to optimise Integer constructor
- Rishikesh (2007-02-25): changed `quo_rem` so that the rem is positive
- David Harvey, Martin Albrecht, Robert Bradshaw (2007-03-01): optimized Integer constructor and pool
- Pablo De Napoli (2007-04-01): `multiplicative_order` should return `+infinity` for non zero numbers
- Robert Bradshaw (2007-04-12): `is_perfect_power`, Jacobi symbol (with Kronecker extension). Convert some methods to use GMP directly rather than `pari`, `Integer()`, `PY_NEW(Integer)`
- David Roe (2007-03-21): sped up valuation and `is_square`, added `val_unit`, `is_power`, `is_power_of` and `divide_knowing_divisible_by`

- Robert Bradshaw (2008-03-26): gamma function, multifactorials
- Robert Bradshaw (2008-10-02): bounded squarefree part

EXAMPLES:

Add 2 integers::

```
sage: a = Integer(3) ; b = Integer(4)
sage: a + b == 7
True
```

Add an integer and a real number::

```
sage: a + 4.0
7.000000000000000
```

Add an integer and a rational number::

```
sage: a + Rational(2)/5
17/5
```

Add an integer and a complex number::

```
sage: b = ComplexField().0 + 1.5
sage: loads((a+b).dumps()) == a+b
True
```

```
sage: z = 32
sage: -z
-32
sage: z = 0; -z
0
sage: z = -0; -z
0
sage: z = -1; -z
1
```

Multiplication::

```
sage: a = Integer(3) ; b = Integer(4)
sage: a * b == 12
True
sage: loads((a * 4.0).dumps()) == a*b
True
sage: a * Rational(2)/5
6/5
```

::

```
sage: list([2,3]) * 4
[2, 3, 2, 3, 2, 3, 2, 3]
```

::

```
sage: 'sage'*Integer(3)
'sagesagesage'
```

COERCIONS: Returns version of this integer in the multi-precision floating real field R.

```
::
```

```
sage: n = 9390823
sage: RR = RealField(200)
sage: RR(n)
9.3908230000000000000000000000000000000000000000000000000000000e6
```

```
"""
```

```
#*****
#      Copyright (C) 2004,2006 William Stein <wstein@gmail.com>
#      Copyright (C) 2006 Gonzalo Tornaria <tornaria@math.utexas.edu>
#      Copyright (C) 2006 Didier Deshommes <dfdeshom@gmail.com>
#      Copyright (C) 2007 David Harvey <dmharvey@math.harvard.edu>
#      Copyright (C) 2007 Martin Albrecht <malb@informatik.uni-bremen.de>
#      Copyright (C) 2007,2008 Robert Bradshaw <robertwb@math.washington.edu>
#      Copyright (C) 2007 David Roe <roed314@gmail.com>
#
# Distributed under the terms of the GNU General Public License (GPL)
# as published by the Free Software Foundation; either version 2 of
# the License, or (at your option) any later version.
#      http://www.gnu.org/licenses/
#*****
```

All code included with Sage must be licensed under the GPLv2+ or a less restrictive license (e.g. the BSD license). It is very important that you include your name in the AUTHORS log so that everybody who submits code to Sage receives proper credit². If ever you feel you are not receiving proper credit for anything you submit to Sage, please let the development team know!

2.1.5 Documentation strings

Docstring markup with ReST and Sphinx

Every function must have a docstring that includes the following information. Source files in the Sage library contain numerous examples on how to format your documentation, so you could use them as a guide.

- A one-sentence description of the function, followed by a blank line.
- An INPUT and an OUTPUT block for input and output arguments (see below for format). The type names should be descriptive, but do not have to represent the exact Sage/Python types. For example, use “integer” for anything that behaves like an integer; you do not have to put a precise type name such as `int`. The INPUT block describes the expected input to your function or method, while the OUTPUT block describes the expected output of the function/method. If appropriate, you need to describe any default values for the input arguments. For example:

```
INPUT:

- ``p`` -- (default: 2) a positive prime integer.

OUTPUT:

A 5-tuple consisting of integers in this order:
```

² See <http://www.sagemath.org/development-map.html>

1. the smallest primitive root modulo p
2. the smallest prime primitive root modulo p
3. the largest primitive root modulo p
4. the largest prime primitive root modulo p
5. total number of prime primitive roots modulo p

Some people prefer to format their OUTPUT section as a block by using a dash. That is acceptable as well:

OUTPUT:

```
- The plaintext resulting from decrypting the ciphertext ``C``  
  using the Blum-Goldwasser decryption algorithm.
```

- Instead of INPUT and OUTPUT blocks, you can include descriptions of the arguments and output using Sphinx/ReST markup, as described in <http://sphinx.pocoo.org/markup/desc.html#info-field-lists>. See below for an example.
- An EXAMPLES block for examples. This is not optional. These examples are used for automatic testing before each release and new functions without these doctests will not be accepted for inclusion with Sage.
- An ALGORITHM block (optional) which indicates what software and/or what algorithm is used. For example ALGORITHM: Uses Pari. Here's a longer example that describes an algorithm used. Note that it also cites the reference where this algorithm can be found:

ALGORITHM:

The following algorithm is adapted from page 89 of [Nat2000]_.

Let ' p ' be an odd (positive) prime and let ' g ' be a generator modulo ' p '. Then ' g^k ' is a generator modulo ' p ' if and only if ' $\gcd(k, p-1) = 1$ '. Since ' p ' is an odd prime and positive, then ' $p - 1$ ' is even so that any even integer between 1 and ' $p - 1$ ', inclusive, is not relatively prime to ' $p - 1$ '. We have now narrowed our search to all odd integers ' k ' between 1 and ' $p - 1$ ', inclusive.

So now start with a generator ' g ' modulo an odd (positive) prime ' p '. For any odd integer ' k ' between 1 and ' $p - 1$ ', inclusive, ' g^k ' is a generator modulo ' p ' if and only if ' $\gcd(k, p-1) = 1$ '.

REFERENCES:

```
.. [Nat2000] M.B. Nathanson. Elementary Methods in Number Theory.  
   Springer, 2000.
```

You can also number the steps in your algorithm using the hash-dot symbol. This way, the actual numbering of the steps are automatically taken care of when you build the documentation:

ALGORITHM:

The Blum-Goldwasser decryption algorithm is described in Algorithm 8.56, page 309 of [MenezesEtAl1996]_. The algorithm works as follows:

- #. Let ' C ' be the ciphertext ' $C = (c_1, c_2, \dots, c_t, x_{t+1})$ '. Then ' t ' is the number of ciphertext sub-blocks and ' h ' is the length of each binary string sub-block ' c_i '.
- #. Let ' (p, q, a, b) ' be the private key whose corresponding public key is ' $n = pq$ '. Note that ' $\gcd(p, q) = ap + bq = 1$ '.
- #. Compute ' $d_1 = ((p + 1) / 4)^{t+1} \bmod (p - 1)$ '.

```

#. Compute `d_2 = ((q + 1) / 4)^{t+1} \bmod{(q - 1)}`.
#. Let `u = x_{t+1}^{d_1} \bmod p`.
#. Let `v = x_{t+1}^{d_2} \bmod q`.
#. Compute `x_0 = vap + ubq \bmod n`.
#. For `i` from 1 to `t`, do:

    #. Compute `x_i = x_{t-1}^2 \bmod n`.
    #. Let `p_i` be the `h` least significant bits of `x_i`.
    #. Compute `m_i = p_i \oplus c_i`.

#. The plaintext is `m = m_1 m_2 \cdots m_t`.

```

- A NOTES block for special notes (optional). Include information such as purpose etc. A NOTES block should start with `.. NOTE::`. You can also use the lower-case version `.. note::`, but do not mix lower-case with upper-case. However, you are encouraged to use the upper-case version `.. NOTE::`. If you want to put anything within the NOTES block, you should indent it at least 4 spaces (no tabs). Here's an example of a NOTES block:

```

.. NOTE::

    You should note that this sentence is indented at least 4
    spaces. Avoid tab characters as much as possible when
    writing code or editing the Sage documentation. You should
    follow Python conventions by using spaces only.

```

- A WARNING block for critical information about your code. For example, the WARNING block might include information about when or under which conditions your code might break, or information that the user should be particularly aware of. A WARNING block should start with `.. WARNING::`. It can also be the lower-case form `.. warning::`. However, you are encouraged to use the upper-case form `.. WARNING::`. Here's an example of a WARNING block:

```

.. WARNING::

    Whenever you edit the Sage documentation, make sure that
    the edited version still builds. That is, you need to ensure
    that you can still build the HTML and PDF versions of the
    updated documentation. If the edited documentation fails to
    build, it is very likely that you would be requested to
    change your patch.

```

- A TODO block for room for improvements. The TODO block might contains disabled doctests to demonstrate the desired feature. A TODO block should start with `.. TODO::`. It can also be the lower-case form `.. todo::`. However, you are encouraged to use the upper-case form `.. TODO::`. Here's an example of a TODO block:

```

.. TODO::

    Improve further function ``have_fresh_beers`` using algorithm
    ``buy_a_better_fridge``:

    sage: have_fresh_beers('Bière de l'Yvette') # todo: not implemented
    Enjoy !

```

- A REFERENCES block to list books or papers (optional). This block serves a similar purpose to a list of references in a research paper, or a bibliography in a monograph. If your method, function or class uses an algorithm that can be found in a standard reference, you should list that reference under this block. The Sphinx/ReST markup for citations is described at <http://sphinx.pocoo.org/rest.html#citations>. See below for an example.
- An AUTHORS block (optional, but encouraged for important functions, so users can see from the docstring

who wrote it and therefore whom to contact if they have questions).

Use the following template when documenting functions. Note the indentation:

```
def point(self, x=1, y=2):
    r"""
    This function returns the point  $(x^5, y)$ .

    INPUT:

    - ``x`` - integer (default: 1) the description of the
      argument x goes here. If it contains multiple lines, all
      the lines after the first need to be indented.

    - ``y`` - integer (default: 2) the ...

    OUTPUT:

    integer -- the ...

    EXAMPLES:

    This example illustrates ...

    ::

        sage: A = ModuliSpace()
        sage: A.point(2,3)
        xxx

    We now ...

    ::

        sage: B = A.point(5,6)
        sage: xxx

    It is an error to ...::

        sage: C = A.point('x',7)
        Traceback (most recent call last):
        ...
        TypeError: unable to convert x (=r) to an integer

    NOTES:

    This function uses the algorithm of [BCDT]_ to determine
    whether an elliptic curve E over Q is modular.

    ...

    REFERENCES:

    .. [BCDT] Breuil, Conrad, Diamond, Taylor, "Modularity ...."

    AUTHORS:

    - William Stein (2005-01-03)
```

```
- First_name Last_name (yyyy-mm-dd)
"""
<body of the function>
```

If you used Sphinx/ReST markup for the arguments, the beginning of the docstring would look like this:

```
def point(self, x=1, y=2):
    r"""
    This function returns the point `(x^5,y)`.

    :param x: the description of the argument x goes here.
        If it contains multiple lines, all the lines after the
        first need to be indented.

    :type x: integer; default 1

    :param y: the ...

    :type y: integer; default 2

    :returns: the ...

    :rtype: integer, the return type
```

You are strongly encouraged to:

- Use nice LaTeX formatting everywhere. If you use backslashes, either use double backslashes or place an “r” right before the first triple opening quote. For example,

```
def cos(x):
    """
    Returns  $\cos(x)$ .
    """

def sin(x):
    r"""
    Returns  $\sin(x)$ .
    """
```

You can also use the MATH block to format complicated mathematical expressions:

```
.. MATH::

    \sum_{i=1}^{\infty} (a_1 a_2 \cdots a_i)^{1/i}
    \leq
    e \sum_{i=1}^{\infty} a_i
```

Note: In ReST documentation, you use backticks ‘ to mark LaTeX code to be typeset. In Sage docstrings, unofficially you may use dollar signs instead – “unofficially” means that it ought to work, but might be a little buggy. Thus $x^2 + y^2 = 1$ and $\$x^2 + y^2 = 1\$$ should produce identical output, typeset in math mode.

LaTeX style: typeset standard rings and fields like the integers and the real numbers using the locally-defined macro `\Bold`, as in `\Bold{Z}` for the integers. This macro is defined to be ordinary bold-face `\mathbf` by default, but users can switch to blackboard-bold `\mathbb` and back on-the-fly by using `latex.blackboard_bold(True)` and `latex.blackboard_bold(False)`.

The docstring will be available interactively (for the “def point...” example above, by typing “point?” at the “sage:” prompt) and also in the reference manual. When viewed interactively, LaTeX code has the backslashes

stripped from it, so “`\cos`” will appear as “`cos`”.

Because of the dual role of the docstring, you need to strike a balance between readability (for interactive help) and using perfect LaTeX code (for the reference manual). For instance, instead of using “`\frac{a}{b}`”, use “`a/b`” or maybe “`a b^{-1}`”. Also keep in mind that some users of Sage are not familiar with LaTeX; this is another reason to avoid complicated LaTeX expressions in docstrings, if at all possible: “`\frac{a}{b}`” will be obscure to someone who doesn't know any LaTeX.

Finally, a few non-standard LaTeX macros are available to help achieve this balance, including “`\ZZ`”, “`\RR`”, “`\CC`”, and “`\QQ`”. These are names of Sage rings, and they are typeset using a single boldface character; they allow the use of “`\ZZ`” in a docstring, for example, which will appear interactively as “`ZZ`” while being typeset as “`\Bold{Z}`” in the reference manual. Other examples are “`\GF`” and “`\Zmod`”, each of which takes an argument: “`\GF{q}`” is typeset as “`\Bold{F}_{q}`” and “`\Zmod{n}`” is typeset as “`\Bold{Z}/n\Bold{Z}`”. See the file `$SAGE_ROOT/devel/sage/sage/misc/latex_macros.py` for a full list and for details about how to add more macros.

-
- Liberally describe what the examples do. Note that there must be a blank line after the example code and before the explanatory text for the next example (indentation is not enough).
 - Illustrate any exceptions raised by the function with examples, as given above. (It is an error to ...; In particular, use ...)
 - Include many examples. These are automatically tested on a regular basis, and are crucial for the quality and adaptability of Sage. Without such examples, small changes to one part of Sage that break something else might not go seen until much later when someone uses the system, which is unacceptable. Note that new functions without doctests will not be accepted for inclusion in Sage.

Warning: Functions whose names start with an underscore do not currently appear in the reference manual, so avoid putting crucial documentation in their docstrings. In particular, if you are defining a class, you might put a long informative docstring after the class definition, not for the `__init__` method. For example, from the file `SAGE_ROOT/devel/sage/sage/crypto/classical.py`:

```
class HillCryptosystem(SymmetricKeyCryptosystem):
    """
    Create a Hill cryptosystem defined by the 'm' x 'm' matrix space
    over '\mathbf{Z} / N \mathbf{Z}', where 'N' is the alphabet size of
    the string monoid 'S'.

    INPUT:

    - 'S' - a string monoid over some alphabet

    - 'm' - integer '> 0'; the block length of matrices that specify
      block permutations

    OUTPUT:

    - A Hill cryptosystem of block length 'm' over the alphabet 'S'.

    EXAMPLES::

    sage: S = AlphabeticStrings()
    sage: E = HillCryptosystem(S, 3)
    sage: E
    Hill cryptosystem on Free alphabetic string monoid on A-Z of block length 3
    """
```

and so on, while the `__init__` method starts like this:

```
def __init__(self, S, m):
    """
    See 'HillCryptosystem' for full documentation.

    EXAMPLES::
    ...
    """
```

Note also that the first docstring is printed if users type “HillCryptosystem?” at the “sage:” prompt. (Before Sage 3.4, the reference manual used to include methods starting with underscores, so you will probably find many examples in the code which don’t follow this advice...)

Automatic testing

The code in the examples should pass automatic testing. This means that if the above code is in the file `f.py` (or `f.sage`), then `sage -t f.py` should not give any error messages. Testing occurs with full Sage preparing of input within the standard Sage shell environment, as described in *Sage preparing*. **Important:** The file `f.py` is not imported when running tests unless you have arranged that it be imported into your Sage environment, i.e. unless its functions are available when you start Sage using the `sage` command. For example, the function `cdd_convert` in the file `SAGE_ROOT/devel/sage/sage/geometry/polyhedra.py` includes an `EXAMPLES` block containing the following:

```
sage: from sage.geometry.polyhedra import cdd_convert
sage: cdd_convert(' 1 1 0 0')
```

```
[1, 1, 0, 0]
```

Sage does not know about the function `cdd_convert` by default, so it needs to be imported before it is tested. Hence the first line in the example.

Further conventions for automated testing of examples

The Python script `SAGE_LOCAL/bin/sage-doctest` implements documentation testing in Sage (see *Automated testing* for more details). When writing documentation, keep the following points in mind:

- All input is prepared before being passed to Python, e.g. `2/3` is replaced by `Integer(2)/Integer(3)`, which evaluates to `2/3` as a rational instead of the Python `int` `0`. For more information on preparsing, see *Sage preparsing*.
- If a test outputs to a file, the file should be in a temporary directory. For example (taken from the file `SAGE_ROOT/devel/sage/sage/plot/plot.py`):

```
sage: fig.savefig(os.path.join(SAGE_TMP, 'test.png'))
```

Here `fig.savefig` is the function doing the saving, `SAGE_TMP` is a temporary directory—this variable will always be defined properly during automated testing—and `os.path.join` is the preferred way to construct a path from a directory and a file. It works more generally than a Unix-flavored construction like `SAGE_TMP + '/test.png'`. If you want to use `SAGE_TMP` in Sage code, not just in a doctest, then you need to import it. Search the Sage code for examples.

- If a test line contains the text `random`, it is executed by `sage-doctest` but `sage-doctest` does not check that the output agrees with the output in the documentation string. For example, the docstring for the `__hash__` method for `CombinatorialObject` in `SAGE_ROOT/devel/sage/sage/combinat/combinat.py` includes the lines

```
sage: hash(c) #random
1335416675971793195
sage: c._hash #random
1335416675971793195
```

However, most functions generating pseudorandom output do not need this tag since the doctesting framework guarantees the state of the pseudorandom number generators (PRNGs) used in Sage for a given doctest. See *Randomized testing* for details on this framework.

- If a line contains the text `long time` then that line is not tested unless the `-long` option is given, e.g. `sage -t -long f.py`. Use this to include examples that take more than about a second to run. These will not be run regularly during Sage development, but will get run before major releases. No example should take more than about 30 seconds.

For instance, here is part of the docstring from the `regulator` method for rational elliptic curves, from the file `SAGE_ROOT/devel/sage/sage/schemes/elliptic_curves/ell_rational.py`:

```
sage: E = EllipticCurve([0, 0, 1, -1, 0])
sage: E.regulator() # long time (1 second)
0.0511114082399688
```

- If a line contains `tol` or `tolerance`, numerical results are only verified to the given tolerance. This may be prefixed by `abs[olute]` or `rel[ative]` to specify whether to measure absolute or relative error; this defaults to relative error except when the expected value is exactly zero:

```
sage: RDF(pi) # abs tol 1e-5
3.14159
sage: [10^n for n in [0.0 .. 4]] # rel tol 2e-4
[0.9999, 10.001, 100.01, 999.9, 10001]
```

This can be useful when the exact output is subject to rounding error and/or processor floating point arithmetic variation. Here are some more examples.

A singular value decomposition of a matrix will produce two unitary matrices. Over the reals, this means the inverse of the matrix is equal to its transpose. We test this result by applying the norm to a matrix difference. The result will usually be a “small” number, distinct from zero.

```
sage: A = matrix(RDF, 8, range(64))
sage: U, S, V = A.SVD()
sage: (U.transpose()*U-identity_matrix(8)).norm()      # abs tol 1e-10
0.0
```

The 8-th cyclotomic field is generated by the complex number $e^{\frac{i\pi}{4}}$. Here we compute a numerical approximation. The value provided in the source of the doctest ($0.707106781186548 + 0.707106781186547*I$), and the value computed by the tested instance of Sage ($N(\text{zeta}8)$) are subtracted from each other and fed into the `abs()` function for comparison to the tolerance. So the only prerequisite for using this feature is that the `abs()` function may be applied. Of course, for a relative tolerance, division must also be possible.

```
sage: K = CyclotomicField(8)
sage: g = K.gen(0); g
zeta8
sage: N(zeta8)                                # absolute tolerance 1e-15
0.707106781186548 + 0.707106781186547*I
```

A relative tolerance on a root of a polynomial. Notice that the root should normally print as $1e+16$, or something similar. However, the tolerance testing causes the doctest framework to use the output in a *computation*, so any valid text representation of the predicted value may be used.

```
sage: y = polygen(RDF, 'y')
sage: p = (y - 10^16)*(y-10^(-13))*(y-2); p
y^3 + (-1e+16)*y^2 + (2e+16)*y - 2000.0
sage: p.roots(multiplicities=False)[2]        # relative tol 1e-10
10^16
```

- If a line contains `todo: not implemented`, it is never tested. It is good to include lines like this to make clear what we want Sage to eventually implement:

```
sage: factor(x*y - x*z)      # todo: not implemented
```

It is also immediately clear to the user that the indicated example does not currently work.

- If a line contains the text `optional`, it is not tested unless either the `--optional` flag or the `--only-optional` flag is passed to `sage -t`. Mark a doctest as `optional` if it requires optional packages; even better, mark it as `optional - PKG_NAME` if it requires the package `PKG_NAME`. Running `sage -t --optional f.py` executes all doctests, including those marked as `optional`. Running `sage -t --only-optional=sloane_database f.py` runs only those doctests marked as `# optional - sloane_database`. For example, the file `SAGE_ROOT/devel/sage/sage/databases/sloane.py` contains the lines

```
sage: sloane_sequence(60843)      # optional - internet
```

and

```
sage: SloaneEncyclopedia[60843]  # optional - sloane_database
```

The first of these just needs internet access, while the second requires that the “`sloane_database`” package be installed. Calling `sage -t --optional` on this file runs both of these tests, while calling `sage -t --only-optional=internet` on it will only run the first test. A test requiring several packages would

be marked `optional - pkg1 pkg2` and executed by `sage -t --only-optional=pkg1,pkg2 f.py`.

Note: Any text after `optional` is interpreted as a package name. Therefore if the doctest is marked `optional: requires chomp`, then `requires` is viewed as a package name, so the test would only be run by either `sage -t --optional f.py` or `sage -t --only-optional=requires,chomp f.py`.

- If the entire documentation string contains all three words `optional`, `package`, and `installed`, then the entire documentation string is not executed unless the `--optional` flag is passed to `sage -t`. This is useful for a long sequence of examples that all require that an optional package be installed.

Using `search_src` from the Sage prompt (or `grep`), one can easily find the aforementioned keywords. In the case of `todo: not implemented`, one can use the results of such a search to direct further development on Sage.

2.1.6 Automated testing

This section describes Sage's automated testing of test files of the following types: `.py`, `.pyx`, `.sage`, `.rst`. Briefly, use `sage -t <file>` to test that the examples in `<file>` behave exactly as claimed. See the following subsections for more details. See also *Documentation strings* for a discussion on how to include examples in documentation strings and what conventions to follow. The chapter *Doctesting the Sage Library* contains a tutorial on doctesting modules in the Sage library.

Testing `.py`, `.pyx` and `.sage` files

Run `sage -t <filename.py>` to test all code examples in `filename.py`. Similar remarks apply to `.sage` and `.pyx` files.

```
sage -t [--verbose] [--optional] [files and directories ... ]
```

When you run `sage -t <filename.py>`, Sage makes a copy of `<filename.py>` with all the `sage` prompts replaced by `>>>`, then uses the standard Python doctest framework to test the documentation. More precisely, the Python script `SAGE_LOCAL/bin/sage-doctest` implements documentation testing. It does the following when asked to test a file `foo.py` or `foo.sage`.

1. Creates the directory `.doctest` if it does not exist and the file `.doctest/foo.py`.
2. The file `.doctest/foo.py` contains functions for each docstring in `foo.py`, but with all Sage preparsing applied and with `from sage.all import *` at the top. The function documentation is thus standard Python with `>>>` prompts.
3. The script `SAGE_LOCAL/bin/sage-doctest` then runs Sage's Python interpreter on `.doctest/foo.py`.

Your file passes these tests if the code in it will run when entered at the `sage:` prompt with no special imports. Thus users are guaranteed to be able to exactly copy code out of the examples you write for the documentation and have them work.

Testing ReST documentation

Run `sage -t <filename.rst>` to test the examples in verbatim environments in ReST documentation. Sage creates a file `.doctest_filename.py` and tests it just as for `.py`, `.pyx` and `.sage` files.

Of course in ReST files, one often inserts explanatory texts between different verbatim environments. To link together verbatim environments, use the `.. link` comment. For example:

```
::
```

```
    sage: a = 1
```

Next we add 1 to ```a```.

```
.. link
```

```
::
```

```
    sage: 1 + a
    2
```

If you want to link all the verbatim environments together, you can put `.. linkall` anywhere in the file, on a line by itself. (For clarity, it might be best to put it near the top of the file.) Then `sage -t` will act as if there were a `.. link` before each verbatim environment. The file `SAGE_ROOT/devel/sage/doc/en/tutorial/interfaces.rst` contains a `.. linkall` directive, for example.

You can also put `.. skip` right before a verbatim environment to have that example skipped when testing the file. This goes in the same place as the `.. link` in the previous example.

See the files in `SAGE_ROOT/devel/sage/doc/en/tutorial/` for many examples of how to include automated testing in ReST documentation for Sage.

2.1.7 Randomized testing

In addition to all the examples in your docstrings, which serve as both demonstrations and tests of your code, you should consider creating a test suite. Think of this as a program that will run for a while and “tries” to crash your code using randomly generated input. Your test code should define a class `Test` with a `random()` method that runs random tests. These are all assembled together later, and each test is run for a certain amount of time on a regular basis.

For example, see the file `SAGE_ROOT/devel/sage/sage/modular/modsym/tests.py`.

2.2 Coding in Python for Sage

This chapter discusses some issues with, and advice for, coding in Sage.

2.2.1 Design

If you are planning to develop some new code for Sage, design is important. So think about what your program will do and how that fits into the structure of Sage. In particular, much of Sage is implemented in the object-oriented language Python, and there is a hierarchy of classes that organize code and functionality. For example, if you implement elements of a ring, your class should derive from `sage.structure.element.RingElement`, rather than starting from scratch. Try to figure out how your code should fit in with other Sage code, and design it accordingly.

2.2.2 Special Sage functions

Functions with leading and trailing double underscores `__XXX__` are all predefined by Python. Functions with leading and trailing single underscores `_XXX_` are defined for Sage. Functions with a single leading underscore are meant to

be semi-private, and those with a double leading underscore are considered really private. Users can create functions with leading and trailing underscores.

Just as Python has many standard special methods for objects, Sage also has special methods. They are typically of the form `__XXX__`. (In a few cases, the trailing underscore is not included, but this will be changed so that the trailing underscore is always included.) This section describes these special methods.

All objects in Sage should derive from the Cython extension class `SageObject`:

```
from sage.ext.sage_object import SageObject

class MyClass(SageObject, ...):
    ...
```

or from some other already existing Sage class:

```
from sage.rings.ring import Algebra

class MyFavoriteAlgebra(Algebra):
    ...
```

You should implement the `__latex__` and `__repr__` method for every object. The other methods depend on the nature of the object.

LaTeX representation

Every object `x` in Sage should support the command `latex(x)`, so that any Sage object can be easily and accurately displayed via LaTeX. Here is how to make a class (and therefore its instances) support the command `latex`.

1. Define a method `__latex__(self)` that returns a LaTeX representation of your object. It should be something that can be typeset correctly within math mode. Do not include opening and closing `$`'s.
2. Often objects are built up out of other Sage objects, and these components should be typeset using the `latex` function. For example, if `c` is a coefficient of your object, and you want to typeset `c` using LaTeX, use `latex(c)` instead of `c.__latex__()`, since `c` might not have a `__latex__` method, and `latex(c)` knows how to deal with this.
3. Do not forget to include a docstring and an example that illustrates LaTeX generation for your object.
4. You can use any macros included in `amsmath`, `amssymb`, or `amsfonts`, or the ones defined in `SAGE_ROOT/doc/commontex/macros.tex`.

An example template for a `__latex__` method follows:

```
class X:
    ...
    def __latex__(self):
        r"""
        Returns the LaTeX representation of X.

        EXAMPLES::

            sage: a = X(1,2)
            sage: latex(a)
            '\\frac{1}{2}'
        """
        return '\\frac{%s}{%s}'%(latex(self.numer), latex(self.denom))
```

As shown in the example, `latex(a)` will produce LaTeX code representing the object `a`. Calling `view(a)` will display the typeset version of this.

Print representation

The standard Python printing method is `__repr__(self)`. In Sage, that is for objects that derive from `SageObject` (which is everything in Sage), instead define `_repr_(self)`. This is preferable because if you only define `_repr_(self)` and not `__repr__(self)`, then users can rename your object to print however they like. Also, some objects should print differently depending on the context.

Here is an example of the `_latex_` and `_repr_` functions for the `Pi` class. It is from the file `SAGE_ROOT/devel/sage/sage/functions/constants.py`:

```
class Pi (Constant):
    """
    The ratio of a circle's circumference to its diameter.

    EXAMPLES:
    sage: pi
    pi
    sage: float(pi)
    3.1415926535897931
    """
    ...
    def _repr_(self):
        return "pi"

    def _latex_(self):
        return "\\pi"
```

Matrix or vector from object

Provide a `_matrix_` method for an object that can be coerced to a matrix over a ring R . Then the Sage function `matrix` will work for this object.

The following is from `SAGE_ROOT/devel/sage/sage/graphs/graph.py`:

```
class GenericGraph (SageObject):
    ...
    def _matrix_(self, R=None):
        if R is None:
            return self.am()
        else:
            return self.am().change_ring(R)

    def adjacency_matrix(self, sparse=None, boundary_first=False):
        ...
```

Similarly, provide a `_vector_` method for an object that can be coerced to a vector over a ring R . Then the Sage function `vector` will work for this object.

The following is from the file `SAGE_ROOT/sage/sage/modules/free_module_element.pyx`:

```
cdef class FreeModuleElement (element_Vector): # abstract base class
    ...
    def _vector_(self, R):
        return self.change_ring(R)
```

2.2.3 Sage preparising

The following files are relevant to preparising in Sage:

1. SAGE_ROOT/local/bin/sage-sage
2. SAGE_ROOT/local/bin/sage-preparse
3. SAGE_ROOT/devel/sage/sage/misc/preparser.py

In particular, the file `preparser.py` contains the Sage preparser code. The following are some notes from it:

- In Sage, methods can be called on integer and real literals. Note that in pure Python this would be a syntax error. For example:

```
sage: 16.sqrt()
4
sage: 87.factor()
3 * 29
```

- Raw literals are not preparsed, which can be useful from an efficiency point of view. Just like Python ints are denoted by an L, in Sage raw integer and floating literals are followed by an “r” (or “R”) for raw, meaning not preparsed. For example:

```
sage: a = 393939r
sage: a
393939
sage: type(a)
<type 'int'>
sage: b = 393939
sage: type(b)
<type 'sage.rings.integer.Integer'>
sage: a == b
True
```

- Raw literals can be very useful in certain cases. For instance, Python integers can be more efficient than Sage integers when they are very small. Large Sage integers are much more efficient than Python integers since they are implemented using the GMP C library.

Consult the file `preparser.py` for more details about Sage preparising, more examples involving raw literals, etc.

When a file `foo.sage` is loaded in a Sage session, a preparsed version of `foo.sage` is created and named `foo.py`. The beginning of `foo.py` states:

```
This file was *autogenerated* from the file foo.sage.
```

2.2.4 The Sage coercion model

The primary goal of coercion is to be able to transparently do arithmetic, comparisons, etc. between elements of distinct sets. For example, when one writes $1 + 1/2$, one wants to perform arithmetic on the operands as rational numbers, despite the left term being an integer. This makes sense given the obvious and natural inclusion of the integers into the rational numbers. The goal of the coercion system is to facilitate this (and more complicated arithmetic) without having to explicitly map everything over into the same domain, and at the same time being strict enough to not resolve ambiguity or accept nonsense.

The coercion model for Sage is described in detail, with examples, in the Coercion section of the Sage Reference Manual.

2.2.5 Mutability

Parent structures (e.g. rings, fields, matrix spaces, etc.) should be immutable and globally unique whenever possible. Immutability means, among other things, that properties like generator labels and default coercion precision cannot be changed.

Global uniqueness while not wasting memory is best implemented using the standard Python weakref module, a factory function, and module scope variable.

Certain objects, e.g. matrices, may start out mutable and become immutable later. See the file `SAGE_ROOT/devel/sage/sage/structure/mutability.py`.

2.2.6 The `__hash__` special method

Here is the definition of `__hash__` from the Python reference manual.

Called by built-in function `hash()` and for operations on members of hashed collections including set, frozenset, and dict. `__hash__()` should return an integer. The only required property is that objects which compare equal have the same hash value; it is advised to somehow mix together (e.g. using exclusive or) the hash values for the components of the object that also play a part in comparison of objects. If a class does not define a `__cmp__()` method it should not define a `__hash__()` operation either; if it defines `__cmp__()` or `__eq__()` but not `__hash__()`, its instances will not be usable as dictionary keys. If a class defines mutable objects and implements a `__cmp__()` or `__eq__()` method, it should not implement `__hash__()`, since the dictionary implementation requires that a key's hash value is immutable (if the object's hash value changes, it will be in the wrong hash bucket).

Notice the phrase, "The only required property is that objects which compare equal have the same hash value." This is an assumption made by the Python language, which in Sage we simply cannot make (!), and violating it has consequences. Fortunately, the consequences are pretty clearly defined and reasonably easy to understand, so if you know about them they do not cause you trouble. The following example illustrates them pretty well:

```
sage: v = [Mod(2,7)]
sage: 9 in v
True
sage: v = set([Mod(2,7)])
sage: 9 in v
False
sage: 2 in v
True
sage: w = {Mod(2,7): 'a'}
sage: w[2]
'a'
sage: w[9]
Traceback (most recent call last):
...
KeyError: 9
```

Here is another example:

```
sage: R = RealField(10000)
sage: a = R(1) + R(10)^-100
sage: a == RDF(1) # because the a gets coerced down to RDF
True
```

but `hash(a)` should not equal `hash(1)`.

Unfortunately, in Sage we simply cannot require

```
(#) "a == b ==> hash(a) == hash(b) "
```

because serious mathematics is simply too complicated for this rule. For example, the equalities $z == \text{Mod}(z, 2)$ and $z == \text{Mod}(z, 3)$ would force `hash()` to be constant on the integers.

The only way we could “fix” this problem for good would be to abandon using the `==` operator for “Sage equality”, and implement Sage equality as a new method attached to each object. Then we could follow Python rules for `==` and our rules for everything else, and all Sage code would become completely unreadable (and for that matter unwritable). So we just have to live with it.

So what is done in Sage is to attempt to satisfy (#) when it is reasonably easy to do so, but use judgment and not go overboard. For example,

```
sage: hash(Mod(2, 7))
2
```

The output 2 is better than some random hash that also involves the moduli, but it is of course not right from the Python point of view, since $9 == \text{Mod}(2, 7)$.

The goal is to make a hash function that is fast, but within reason respects any obvious natural inclusions and coercions.

2.2.7 Exceptions

Please avoid code like this:

```
try:
    some_code()
except:
    more_code() # bad
```

Instead, catch specific exceptions. For example,

```
try:
    return self.__coordinate_ring
except (AttributeError, OtherExceptions), msg:
    more_code_to_compute_something() # Good
```

Note that the syntax in `except` is to list all the exceptions that are caught as a tuple, followed by an error message.

If you do not have any exceptions explicitly listed (as a tuple), your code will catch absolutely anything, including `ctrl-C`, typos in the code, and alarms, and this will lead to confusion. Also, this might catch real errors which should be propagated to the user.

2.2.8 Importing

We mention two issues with importing: circular imports and importing large third-party modules.

First, you must avoid circular imports. For example, suppose that the file `SAGE_ROOT/devel/sage/sage/algebras/steenrod_algebra.py` started with a line

```
from sage.sage.algebras.steenrod_algebra_bases import *
```

and that the file `SAGE_ROOT/devel/sage/sage/algebras/steenrod_algebra_bases.py` started with a line

```
from sage.sage.algebras.steenrod_algebra import SteenrodAlgebra
```

This sets up a loop: loading one of these files requires the other, which then requires the first, etc.

With this set-up, running Sage will produce an error:

```
Exception exceptions.ImportError: 'cannot import name SteenrodAlgebra'
in 'sage.rings.polynomial.polynomial_element.
Polynomial_generic_dense.__normalize' ignored
-----
ImportError                                Traceback (most recent call last)

...
ImportError: cannot import name SteenrodAlgebra
```

Instead, you might replace the `import *` line at the top of the file by more specific imports where they are needed in the code. For example, the `basis` method for the class `SteenrodAlgebra` might look like this (omitting the documentation string):

```
def basis(self, n):
    from steenrod_algebra_bases import steenrod_algebra_basis
    return steenrod_algebra_basis(n, basis=self._basis_name, p=self.prime)
```

Second, do not import at the top level of your module a third-party module that will take a long time to initialize (e.g. `matplotlib`). As above, you might instead import specific components of the module when they are needed, rather than at the top level of your file.

It is important to try to make `from sage.all import *` as fast as possible, since this is what dominates the Sage startup time, and controlling the top-level imports helps to do this.

2.2.9 Editing existing files

There are several copies of Sage library files, and it can be confusing for beginners to know which one to modify. In the directory `SAGE_ROOT/devel/sage`, there is a subdirectory `build` which contains copies of Python files and their byte-compiled versions, along with compiled version of Cython files. These are the files that Sage actually uses, but *you never need to touch these*. Instead, always work with files in the directory `SAGE_ROOT/devel/sage/sage`. For example, if you want to add a new method for simplicial complexes, then edit the file `SAGE_ROOT/devel/sage/sage/homology/simplicial_complex.py`. Save your changes, and then type `sage -b` to incorporate those changes. This automatically copies the appropriate files into the appropriate places under `SAGE_ROOT/devel/sage/build`.

You should also read *Producing Patches with Mercurial* for information about how to create a copy of the Sage library and make your changes there, so that first, it is easy to undo your changes, and second, it is easy to produce a “patch” file so you can share your changes with other people.

2.2.10 Creating a new directory

If you want to create a new directory in the Sage library `SAGE_ROOT/devel/sage/sage` (say, `measure_theory`), that directory should contain an empty file `__init__.py` in addition to whatever files you want to add (say, `borel_measure.py` and `banach_tarski.py`), and also a file `all.py` listing imports from that directory. The file `all.py` might look like this:

```
from borel_measure import BorelMeasure

from banach_tarski import BanachTarskiParadox
```

Then in the file `SAGE_ROOT/devel/sage/sage/all.py`, add a line

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

Finally, add the directory name (“measure_theory”) to the `packages` list in the `Distutils` section of the file `SAGE_ROOT/devel/sage/setup.py`: add a line

```
'sage.measure_theory',
```

between

```
'sage.matrix',
```

and

```
'sage.media',
```

As noted above, you should also read *Producing Patches with Mercurial* for information about how to do this in a copy of the Sage library and how to disseminate your changes.

2.2.11 Using optional packages

If a function requires an optional package, that function should fail gracefully—perhaps using a `try-except` block—when the optional package is not available, and should give a hint about how to install it. For example, typing `sage -optional` gives a list of all optional packages, so it might suggest to the user that they type that. The command `optional_packages()` from within Sage also returns this list.

2.3 Coding in Cython

This chapter discusses Cython, which is a compiled language based on Python. The major advantage it has over Python is that code can be much faster (sometimes orders of magnitude).

Cython also allows Sage to interface with C and C++, as well as other languages. See the Python documentation at <http://www.python.org/doc/> for more details. In particular, the section “Extending and Embedding the Python Interpreter”, available at <http://docs.python.org/ext/ext.html>, describes how to write C or C++ modules for use in Python.

Cython is a compiled version of Python. It is based on Pyrex (<http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/>). To a large degree, Cython has changed based on what Sage’s developers needed; Cython has been developed in concert with Sage. However, it is an independent project now, which is used beyond the scope of Sage.

As such, it is a young, but developing language, with young, but developing documentation. See its web page, <http://www.cython.org/>, for the most up-to-date information.

Python is an interpreted language and has no declared data types for variables. These features make it easy to write and debug, but Python code can sometimes be slow. Cython code can look a lot like Python, but it gets translated into C code (often very efficient C code) and then compiled. Thus it offers a language which is familiar to Python developers, but with the potential for much greater speed.

2.3.1 Writing Cython code in Sage

There are several ways to create and build Cython code in Sage.

1. In the Sage Notebook, begin any cell with `%cython`. When you evaluate that cell,
 - (a) It is saved to a file.
 - (b) Cython is run on it with all the standard Sage libraries automatically linked if necessary.

- (c) The resulting `.so` file is then loaded into your running instance of Sage.
 - (d) The functionality defined in that cell is now available for you to use in the notebook. Also, the output cell has a link to the C program that was compiled to create the `.so` file.
2. Create an `.spyx` file and attach or load it from the command line. This is similar to creating a `%cython` cell in the notebook but works completely from the command line (and not from the notebook).
 3. Create a `.pyx` file and add it to the Sage library.
 - (a) First, add a listing for the Cython extension to the variable `ext_modules` in the file `SAGE_ROOT/devel/sage/module_list.py`. See the `distutils.extension.Extension` class for more information on creating a new Cython extension.
 - (b) Then, if you created a new directory for your `.pyx` file, add the directory name to the `packages` list in the file `SAGE_ROOT/devel/sage/setup.py`. (See also the section on “Creating a new directory” in *Coding in Python for Sage*.)
 - (c) Run `sage -b` to rebuild Sage.

For example, the file `SAGE_ROOT/devel/sage/sage/graphs/chrompoly.pyx` has the lines

```
Extension('sage.graphs.chrompoly',
          sources = ['sage/graphs/chrompoly.pyx']),
```

in `module_list.py`. In addition, `sage.graphs` is included in the `packages` list under the `Distutils` section of `setup.py` since `chrompoly.pyx` is contained in the directory `sage/graphs`.

2.3.2 Special pragmas

If Cython code is either attached or loaded as a `.spyx` file or loaded from the notebook as a `%cython` block, the following pragmas are available:

- `clang` — may be either `c` or `c++` indicating whether a C or C++ compiler should be used.
- `clib` — additional libraries to be linked in, the space separated list is split and passed to `distutils`.
- `cinclude` — additional directories to search for header files. The space separated list is split and passed to `distutils`.
- `cfile` — additional C or C++ files to be compiled
- `cargs` — additional parameters passed to the compiler

For example:

```
#clang C++
#clib givaro
#cinclude /usr/local/include/
#cargs -gdb
#cfile foo.c
```

2.3.3 Attaching or loading .spyx files

The easiest way to try out Cython without having to learn anything about `distutils`, etc., is to create a file with the extension `spyx`, which stands for “Sage Pyrex”:

1. Create a file `power2.spyx`.
2. Put the following in it:

```
def is2pow(n):
    while n != 0 and n%2 == 0:
        n = n >> 1
    return n == 1
```

3. Start the Sage command line interpreter and load the `spyx` file (this will fail if you do not have a C compiler installed).

```
sage: load "power2.spyx"
Compiling power2.spyx...
sage: is2pow(12)
False
```

Note that you can change `power2.spyx`, then load it again and it will be recompiled on the fly. You can also attach `power2.spyx` so it is reloaded whenever you make changes:

```
sage: attach "power2.spyx"
```

Cython is used for its speed. Here is a timed test on a 2.6 GHz Opteron:

```
sage: %time [n for n in range(10^5) if is2pow(n)]
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536]
CPU times: user 0.60 s, sys: 0.00 s, total: 0.60 s
Wall time: 0.60 s
```

Now, the code in the file `power2.spyx` is valid Python, and if we copy this to a file `powerslow.py` and load that, we get the following:

```
sage: load "powerslow.py"
sage: %time [n for n in range(10^5) if is2pow(n)]
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536]
CPU times: user 1.01 s, sys: 0.04 s, total: 1.05 s
Wall time: 1.05 s
```

By the way, we could gain even a little more speed with the Cython version with a type declaration, by changing `def is2pow(n):` to `def is2pow(unsigned int n):`.

2.3.4 Interrupt and signal handling

(This section was written by Jeroen Demeyer.)

When writing Cython code for Sage, special care must be taken to ensure the code can be interrupted with `CTRL-C`. Since Cython optimizes for speed, Cython normally does not check for interrupts. For example, code like the following cannot be interrupted:

```
sage: cython('while True: pass') # DON'T DO THIS
```

While this is running, pressing `CTRL-C` has no effect. The only way out is to kill the Sage process. On certain systems, you can still quit Sage by typing `CTRL-\` (sending a Quit signal) instead of `CTRL-C`.

Using `sig_on()` and `sig_off()`

To enable interrupt handling, use the `sig_on()` and `sig_off()` functions. You should put `sig_on()` *before* and `sig_off()` *after* any Cython code which could potentially take a long time. These two *must always* be called in **pairs**, i.e. every `sig_on()` must be matched by a closing `sig_off()`.

In practice your function will probably look like:

```
def sig_example():
    # (some harmless initialization)
    sig_on()
    # (a long computation here, potentially calling a C library)
    sig_off()
    # (some harmless post-processing)
    return something
```

You can put `sig_on()` and `sig_off()` in all kinds of Cython functions: `def`, `cdef` or `cpdef`. You cannot put them in pure Python code (i.e. files with extension `.py`).

It is possible to put `sig_on()` and `sig_off()` in different functions, provided that `sig_off()` is called before the function which calls `sig_on()` returns. The following code is *invalid*:

```
# INVALID code because we return from function foo()
# without calling sig_off() first.
cdef foo():
    sig_on()

def f1():
    foo()
    sig_off()
```

But the following is valid:

```
cdef int foo():
    sig_off()
    return 2+2

def f1():
    sig_on()
    return foo()
```

For clarity however, it is best to avoid this. One good example where the above makes sense is the `new_gen()` function in *The PARI C library interface*.

A common mistake is to put `sig_off()` towards the end of a function (before the `return`) when the function has multiple `return` statements. So make sure there is a `sig_off()` before *every* `return` (and also before every `raise`).

Warning: The code inside `sig_on()` should be pure C or Cython code. If you call Python code, an interrupt is likely to mess up Python.

Also, when an interrupt occurs inside `sig_on()`, code execution immediately stops without cleaning up. For example, any memory allocated inside `sig_on()` is lost. See *Advanced functions* for ways to deal with this.

When the user presses CTRL-C inside `sig_on()`, execution will jump back to `sig_on()` (the first one if there is a stack) and `sig_on()` will raise `KeyboardInterrupt`. These can be handled just like other Python exceptions:

```
def catch_interrupts():
    try:
        sig_on() # This MUST be inside the try
        # (some long computation)
        sig_off()
    except KeyboardInterrupt:
        # (handle interrupt)
```

Certain C libraries in Sage are written in a way that they will raise Python exceptions: NTL and PARI are examples of this. NTL can raise `RuntimeError` and PARI can raise `PariError`. Since these use the `sig_on()` mechanism, these exceptions can be caught just like the `KeyboardInterrupt` in the example above.

It is possible to stack `sig_on()` and `sig_off()`. If you do this, the effect is exactly the same as if only the outer `sig_on()/sig_off()` was there. The inner ones will just change a reference counter and otherwise do nothing. Make sure that the number of `sig_on()` calls equal the number of `sig_off()` calls:

```
def stack_sig_on():
    sig_on()
    sig_on()
    sig_on()
    # (some code)
    sig_off()
    sig_off()
    sig_off()
```

Extra care must be taken with exceptions raised inside `sig_on()`. The problem is that, if you do not do anything special, the `sig_off()` will never be called if there is an exception. If you need to *raise* an exception yourself, call a `sig_off()` before it:

```
def raising_an_exception():
    sig_on()
    # (some long computation)
    if (something_failed):
        sig_off()
        raise RuntimeError, "something failed"
    # (some more computation)
    sig_off()
    return something
```

Alternatively, you can use `try/finally` which will also catch exceptions raised by subroutines inside the `try`:

```
def try_finally_example():
    sig_on()
    try:
        # (some long computation, potentially raising exceptions)
    finally:
        sig_off()
    return something
```

Other signals

Apart from handling interrupts, `sig_on()` provides more general signal handling. Indeed, if the code inside `sig_on()` would generate a segmentation fault or call the C function `abort()` (or more generally, raise any of `SIGSEGV`, `SIGILL`, `SIGABRT`, `SIGFPE`, `SIGBUS`), this is caught by the interrupt framework and a `RuntimeError` is raised:

```
cdef extern from 'stdlib.h':
    void abort()
```

```
def abort_example():
    sig_on()
    abort()
    sig_off()
```

```
sage: abort_example()
Traceback (most recent call last):
...
RuntimeError: Aborted
```

This exception can then be caught as explained above. This means that `abort()` can be used as an alternative to exceptions within `sig_on()/sig_off()`. A segmentation fault unguarded by `sig_on()` would simply terminate Sage.

Instead of `sig_on()`, there is also a function `sig_str(s)`, which takes a C string `s` as argument. It behaves the same as `sig_on()`, except that the string `s` will be used as a string for the exception. `sig_str(s)` should still be closed by `sig_off()`. Example Cython code:

```
cdef extern from 'stdlib.h':
    void abort()

def abort_example_with_sig_str():
    sig_str("custom error message")
    abort()
    sig_off()
```

Executing this gives:

```
sage: abort_example_with_sig_str()
Traceback (most recent call last):
...
RuntimeError: custom error message
```

With regard to ordinary interrupts (i.e. `SIGINT`), `sig_str(s)` behaves the same as `sig_on()`: a simple `KeyboardInterrupt` is raised.

Advanced functions

There are several more specialized functions for dealing with interrupts. The function `sig_check()` behaves exactly as `sig_on()`; `sig_off()` (except that `sig_check()` is faster since it does not involve a `setjmp()` call).

`sig_check()` can be used to check for pending interrupts. If an interrupt happens outside of a `sig_on()/sig_off()` block, it will be caught by the next `sig_check()` or `sig_on()`.

The typical use case for `sig_check()` is within tight loops doing complicated stuff (mixed Python and Cython code, potentially raising exceptions). It gives more control, because a `KeyboardInterrupt` can *only* be raised during `sig_check()`:

```
def sig_check_example():
    for x in foo:
        # (one loop iteration which does not take a long time)
        sig_check()
```

As mentioned above, `sig_on()` makes no attempt to clean anything up (restore state or freeing memory) when an interrupt occurs. In fact, it would be impossible for `sig_on()` to do that. If you want to add some cleanup code, use `sig_on_no_except()` for this. This function behaves *exactly* like `sig_on()`, except that any exception raised (either `KeyboardInterrupt` or `RuntimeError`) is not yet passed to Python. Essentially, the exception is there, but we prevent Cython from looking for the exception. Then `cython_check_exception()` can be used to make Cython look for the exception.

Normally, `sig_on_no_except()` returns 1. If a signal was caught and an exception raised, `sig_on_no_except()` instead returns 0. The following example shows how to use `sig_on_no_except()`:

```
def no_except_example():
    if not sig_on_no_except():
        # (clean up messed up internal state)

        # Make Cython realize that there is an exception.
        # It will look like the exception was actually raised
```

```
# by cython_check_exception().
cython_check_exception()
# (some long computation, messing up internal state of objects)
sig_off()
```

There is also a function `sig_str_no_except(s)` which is analogous to `sig_str(s)`.

Note: See the file `SAGE_ROOT/devel/sage/sage/tests/interrupt.pyx` for more examples of how to use the various `sig_*()` functions.

Testing interrupts

When writing *Documentation strings*, one sometimes wants to check that certain code can be interrupted in a clean way. In the module `sage.tests.interrupt`, there is a function `interrupt_after_delay(ms_delay = 500)` which can be used to test interrupts. That function simulates a CTRL-C (by sending SIGINT) after `ms_delay` milliseconds.

The following is an example of a doctest demonstrating that the function `factor()` can be interrupted:

```
sage: import sage.tests.interrupt
sage: try:
...     sage.tests.interrupt.interrupt_after_delay()
...     factor(10^1000 + 3)
... except KeyboardInterrupt:
...     print "ok!"
ok!
```

2.4 Coding using external libraries and interfaces

When writing code for Sage, use Python for the basic structure and interface. For speed, efficiency, or convenience, you can implement parts of the code using any of the following languages: *Cython*, C/C++, Fortran 95, GAP, Common Lisp, Singular, and PARI/GP. You can also use all C/C++ libraries included with Sage³. (And if you are okay with your code depending on optional Sage packages, you can use Octave, or even Magma, Mathematica, or Maple.)

In this chapter, we discuss interfaces between Sage and *PARI*, *GAP* and *Singular*.

2.4.1 The PARI C library interface

(This chapter was written by Martin Albrecht.)

Here is a step-by-step guide to adding new PARI functions to Sage. We use the Frobenius form of a matrix as an example.

Some heavy lifting for matrices over integers is implemented using the PARI library. To compute the Frobenius form in PARI, the `matfrobenius` function is used.

There are two ways to interact with the PARI library from Sage. The `gp` interface uses the `gp` interpreter. The PARI interface uses direct calls to the PARI C functions—this is the preferred way as it is much faster. Thus this section focuses on using PARI.

We will add a new method to the `gen` class. This is the abstract representation of all PARI library objects. That means that once we add a method to this class, every PARI object, whether it is a number, polynomial or matrix, will have

³ See <http://www.sagemath.org/links-components.html> for a list

our new method. So you can do `pari(1).matfrobenius()`, but since PARI wants to apply `matfrobenius` to matrices, not numbers, you will receive a `PariError` in this case.

The `gen` class is defined in `SAGE_ROOT/devel/sage/sage/libs/pari/gen.pyx`, and this is where we add the method `matfrobenius`:

```
def matfrobenius(self, flag=0):
    """
    M.matfrobenius(flag=0): Return the Frobenius form of the square
    matrix M. If flag is 1, return only the elementary divisors (a list
    of polynomials). If flag is 2, return a two-components vector [F,B]
    where F is the Frobenius form and B is the basis change so that
    `M=B^{-1} F B`.

    EXAMPLES::

        sage: a = pari('[1,2;3,4]')
        sage: a.matfrobenius()
        [0, 2; 1, 5]
        sage: a.matfrobenius(flag=1)
        [x^2 - 5*x - 2]
        sage: a.matfrobenius(2)
        [[0, 2; 1, 5], [1, -1/3; 0, 1/3]]
    """
    sig_on()
    return self.new_gen(matfrobenius(self.g, flag, 0))
```

Note the use of the `sig_on()` statement.

The `matfrobenius` call is just a call to the PARI C library function `matfrobenius` with the appropriate parameters.

The `self.new_gen(GEN x)` call constructs a new Sage `gen` object from a given PARI `GEN` where the PARI `GEN` is stored as the `.g` attribute. Apart from this, `self.new_gen()` calls a closing `sig_off()` macro and also clears the PARI stack so it is very convenient to use in a return statement as illustrated above. So after `self.new_gen()`, all PARI `GEN`'s which are not converted to Sage `gen`'s are gone. There is also `self.new_gen_noclear(GEN x)` which does the same as `self.new_gen(GEN x)` except that it does *not* call `sig_off()` nor clear the PARI stack.

The information about which function to call and how to call it can be retrieved from the PARI user's manual (note: Sage includes the development version of PARI, so check that version of the user's manual). Looking for `matfrobenius` you can find:

The library syntax is `GEN matfrobenius(GEN M, long flag, long v = -1)`, where `v` is a variable number.

In case you are familiar with `gp`, please note that the PARI C function may have a name that is different from the corresponding `gp` function (for example, see `mathnf`), so always check the manual.

We can also add a `frobenius(flag)` method to the `matrix_integer` class where we call the `matfrobenius()` method on the PARI object associated to the matrix after doing some sanity checking. Then we convert output from PARI to Sage objects:

```
def frobenius(self, flag=0, var='x'):
    """
    Return the Frobenius form (rational canonical form) of this
    matrix.

    INPUT:
```

- `flag` -- 0 (default), 1 or 2 as follows:
 - `0` -- (default) return the Frobenius form of this matrix.
 - `1` -- return only the elementary divisor polynomials, as polynomials in `var`.
 - `2` -- return a two-components vector $[F, B]$ where F is the Frobenius form and B is the basis change so that $M = B^{-1}FB$.
- `var` -- a string (default: `'x'`)

ALGORITHM: uses PARI's `matfrobenius()`

EXAMPLES::

```
sage: A = MatrixSpace(ZZ, 3) (range(9))
sage: A.frobenius(0)
[ 0  0  0]
[ 1  0 18]
[ 0  1 12]
sage: A.frobenius(1)
[x^3 - 12*x^2 - 18*x]
sage: A.frobenius(1, var='y')
[y^3 - 12*y^2 - 18*y]
"""
if not self.is_square():
    raise ArithmeticError("frobenius matrix of non-square matrix not defined.")

v = self._pari_().matfrobenius(flag)
if flag==0:
    return self.matrix_space()(v.python())
elif flag==1:
    r = PolynomialRing(self.base_ring(), names=var)
    retr = []
    for f in v:
        retr.append(eval(str(f).replace("^", "**"), {'x':r.gen()}), r.gens_dict()))
    return retr
elif flag==2:
    F = matrix_space.MatrixSpace(QQ, self.nrows())(v[0].python())
    B = matrix_space.MatrixSpace(QQ, self.nrows())(v[1].python())
    return F, B
```

2.4.2 GAP

(The first version of this chapter was written by David Joyner.)

Wrapping a GAP function in Sage is a matter of writing a program in Python that uses the pexpect interface to pipe various commands to GAP and read back the input into Sage. This is sometimes easy, sometimes hard.

For example, suppose we want to make a wrapper for the computation of the Cartan matrix of a simple Lie algebra. The Cartan matrix of G_2 is available in GAP using the commands

```
gap> L:= SimpleLieAlgebra( "G", 2, Rationals );
<Lie algebra of dimension 14 over Rationals>
```

```
gap> R:= RootSystem( L );
<root system of rank 2>
gap> CartanMatrix( R );
```

(Incidentally, most of the GAP Lie algebra implementation was written by Thomas Breuer, Willem de Graaf and Craig Struble.)

In Sage, one can access these commands by typing

```
sage: L = gap.SimpleLieAlgebra('G', 2, 'Rationals'); L
Algebra( Rationals, [ v.1, v.2, v.3, v.4, v.5, v.6, v.7, v.8, v.9, v.10,
  v.11, v.12, v.13, v.14 ] )
sage: R = L.RootSystem(); R
<root system of rank 2>
sage: R.CartanMatrix()
[ [ 2, -1 ], [ -3, 2 ] ]
```

Note the 'G' which is evaluated in GAP as the string "G".

The purpose of this section is to use this example to show how one might write a Python/Sage program whose input is, say, ('G', 2) and whose output is the matrix above (but as a Sage Matrix—see the code in the directory SAGE_ROOT/devel/sage/sage/matrix/ and the corresponding parts of the Sage reference manual).

First, the input must be converted into strings consisting of legal GAP commands. Then the GAP output, which is also a string, must be parsed and converted if possible to a corresponding Sage/Python object.

```
def cartan_matrix(type, rank):
    """
    Return the Cartan matrix of given Chevalley type and rank.

    INPUT:
        type -- a Chevalley letter name, as a string, for
              a family type of simple Lie algebras
        rank -- an integer (legal for that type).

    EXAMPLES:
        sage: cartan_matrix("A",5)
        [ 2 -1  0  0  0]
        [-1  2 -1  0  0]
        [ 0 -1  2 -1  0]
        [ 0  0 -1  2 -1]
        [ 0  0  0 -1  2]
        sage: cartan_matrix("G",2)
        [ 2 -1]
        [-3  2]
    """

    L = gap.SimpleLieAlgebra("%s"%type, rank, 'Rationals')
    R = L.RootSystem()
    sM = R.CartanMatrix()
    ans = eval(str(sM))
    MS = MatrixSpace(QQ, rank)
    return MS(ans)
```

The output `ans` is a Python list. The last two lines convert that list to an instance of the Sage class `Matrix`.

Alternatively, one could replace the first line of the above function with this:

```
L = gap.new('SimpleLieAlgebra("%s", %s, Rationals);'%(type, rank))
```

Defining “easy” and “hard” is subjective, but here is one definition. Wrapping a GAP function is “easy” if there is already a corresponding class in Python or Sage for the output data type of the GAP function you are trying to wrap. For example, wrapping any GUAVA (GAP’s error-correcting codes package) function is “easy” since error-correcting codes are vector spaces over finite fields and GUAVA functions return one of the following data types:

- vectors over finite fields,
- polynomials over finite fields,
- matrices over finite fields,
- permutation groups or their elements,
- integers.

Sage already has classes for each of these.

A “hard” example is left as an exercise! Here are a few ideas.

- Write a wrapper for GAP’s `FreeLieAlgebra` function (or, more generally, all the finitely presented Lie algebra functions in GAP). This would require creating new Python objects.
- Write a wrapper for GAP’s `FreeGroup` function (or, more generally, all the finitely presented groups functions in GAP). This would require writing some new Python objects.
- Write a wrapper for GAP’s character tables. Though this could be done without creating new Python objects, to make the most use of these tables, it probably would be best to have new Python objects for this.

2.4.3 Singular

(The first version of this chapter was written by David Joyner.)

Using Singular functions from Sage is not much different conceptually from using GAP functions from Sage. As with GAP, this can range from easy to hard, depending on how much of the data structure of the output of the Singular function is already present in Sage.

First, some terminology. For us, a *curve* X over a finite field F is an equation of the form $f(x, y) = 0$, where $f \in F[x, y]$ is a polynomial. It may or may not be singular. A *place of degree d* is a Galois orbit of d points in $X(E)$, where E/F is of degree d . For example, a place of degree 1 is also a place of degree 3, but a place of degree 2 is not since no degree 3 extension of F contains a degree 2 extension. Places of degree 1 are also called F -rational points.

As an example of the Sage/Singular interface, we will explain how to wrap Singular’s `NSplaces`, which computes places on a curve over a finite field. (The command `closed_points` also does this in some cases.) This is “easy” since no new Python classes are needed in Sage to carry this out.

Here is an example on how to use this command in Singular:

```
A Computer Algebra System for Polynomial Computations / version 3-0-0
                                                    0<
    by: G.-M. Greuel, G. Pfister, H. Schoenemann \ May 2005
FB Mathematik der Universitaet, D-67653 Kaiserslautern \
> LIB "brnoeth.lib";
[...]
```

```
> ring s=5, (x,y), lp;
> poly f=y^2-x^9-x;
> list X1=Adj_div(f);
Computing affine singular points ...
Computing all points at infinity ...
Computing affine singular places ...
Computing singular places at infinity ...
Computing non-singular places at infinity ...
```

Adjunction divisor computed successfully

The genus of the curve is 4

```
> list X2=NSplaces(1,X1);
```

Computing non-singular affine places of degree 1 ...

```
> list X3=extcurve(1,X2);
```

Total number of rational places : 6

```
> def R=X3[1][5];
```

```
> setring R;
```

```
> POINTS;
```

```
[1]:
```

```
  [1]:
    0
```

```
  [2]:
    1
```

```
  [3]:
    0
```

```
[2]:
```

```
  [1]:
   -2
```

```
  [2]:
    1
```

```
  [3]:
    1
```

```
[3]:
```

```
  [1]:
   -2
```

```
  [2]:
    1
```

```
  [3]:
    1
```

```
[4]:
```

```
  [1]:
   -2
```

```
  [2]:
   -1
```

```
  [3]:
    1
```

```
[5]:
```

```
  [1]:
    2
```

```
  [2]:
   -2
```

```
  [3]:
    1
```

```
[6]:
```

```
  [1]:
    0
```

```
  [2]:
    0
```

```
  [3]:
    1
```

Here is another way of doing this same calculation in the Sage interface to Singular:

```
sage: singular.LIB("brnoeth.lib")
sage: singular.ring(5, '(x,y)', 'lp')
// characteristic : 5
// number of vars : 2
//      block 1 : ordering lp
//              : names   x y
//      block 2 : ordering C
sage: f = singular('y^2-x^9-x')
sage: print singular.eval("list X1=Adj_div(%s);"%f.name())
Computing affine singular points ...
Computing all points at infinity ...
Computing affine singular places ...
Computing singular places at infinity ...
Computing non-singular places at infinity ...
Adjunction divisor computed successfully
```

The genus of the curve is 4

```
sage: print singular.eval("list X2=NSplaces(1,X1);")
Computing non-singular affine places of degree 1 ...
sage: print singular.eval("list X3=extcurve(1,X2);")
```

Total number of rational places : 6

```
sage: singular.eval("def R=X3[1][5];")
'def R=X3[1][5];'
sage: singular.eval("setring R;")
'setring R;'
sage: L = singular.eval("POINTS;")
```

```
sage: print L
```

```
[1]:
  [1]:
    0
  [2]:
    1
  [3]:
    0
[2]:
  [1]:
    0 # 32-bit
   -2 # 64-bit
  [2]:
    0 # 32-bit
   -1 # 64-bit
  [3]:
    1
...
```

From looking at the output, notice that our wrapper function will need to parse the string represented by L above, so let us write a separate function to do just that. This requires figuring out how to determine where the coordinates of the points are placed in the string L . Python has some very useful string manipulation commands to do just that.

```
def points_parser(string_points, F):
    """
    This function will parse a string of points
    of  $X$  over a finite field  $F$  returned by Singular's NSplaces
    command into a Python list of points with entries from  $F$ .
    """
```

```

EXAMPLES:
    sage: F = GF(5)
    sage: points_parser(L,F)
    ((0, 1, 0), (3, 4, 1), (0, 0, 1), (2, 3, 1), (3, 1, 1), (2, 2, 1))
    """
Pts=[]
n=len(L)
#print n
#start block to compute a pt
L1=L
while len(L1)>32:
    idx=L1.index(" ")
    pt=[]
    ## start block1 for compute pt
    idx=L1.index(" ")
    idx2=L1[idx:].index("\n")
    L2=L1[idx:idx+idx2]
    #print L2
    pt.append(F(eval(L2)))
    # end block1 to compute pt
    L1=L1[idx+8:] # repeat block 2 more times
    #print len(L1)
    ## start block2 for compute pt
    idx=L1.index(" ")
    idx2=L1[idx:].index("\n")
    L2=L1[idx:idx+idx2]
    pt.append(F(eval(L2)))
    # end block2 to compute pt
    L1=L1[idx+8:] # repeat block 1 more time
    ## start block3 for compute pt
    idx=L1.index(" ")
    if "\n" in L1[idx:]:
        idx2=L1[idx:].index("\n")
    else:
        idx2=len(L1[idx:])
    L2=L1[idx:idx+idx2]
    pt.append(F(eval(L2)))
    #print pt
    # end block3 to compute pt
    #end block to compute a pt
    Pts.append(tuple(pt)) # repeat until no more pts
    L1=L1[idx+8:] # repeat block 2 more times
return tuple(Pts)

```

Now it is an easy matter to put these ingredients together into a Sage function which takes as input a triple (f, F, d) : a polynomial f in $F[x, y]$ defining $X : f(x, y) = 0$ (note that the variables x, y must be used), a finite field F of prime order, and the degree d . The output is the number of places in X of degree $d = 1$ over F . At the moment, there is no “translation” between elements of $GF(p^d)$ in Singular and Sage unless $d = 1$. So, for this reason, we restrict ourselves to points of degree one.

```

def places_on_curve(f,F):
    """
    INPUT:
        f -- element of F[x,y], defining X: f(x,y)=0
        F -- a finite field of *prime order*

    OUTPUT:
        integer -- the number of places in X of degree d=1 over F
    """

```

EXAMPLES:

```
sage: F=GF(5)
sage: R=MPolynomialRing(F,2,names=["x","y"])
sage: x,y=R.gens()
sage: f=y^2-x^9-x
sage: places_on_curve(f,F)
((0, 1, 0), (3, 4, 1), (0, 0, 1), (2, 3, 1), (3, 1, 1), (2, 2, 1))
"""
d = 1
p = F.characteristic()
singular.eval('LIB "brnoeth.lib";')
singular.eval("ring s="+str(p)+", (x,y), lp;")
singular.eval("poly f="+str(f))
singular.eval("list X1=Adj_div(f);")
singular.eval("list X2=NSplaces("+str(d)+",X1);")
singular.eval("list X3=extcurve("+str(d)+",X2);")
singular.eval("def R=X3[1][5];")
singular.eval("setring R;")
L = singular.eval("POINTS;")
return points_parser(L,F)
```

Note that the ordering returned by this Sage function is exactly the same as the ordering in the Singular variable POINTS.

One more example (in addition to the one in the docstring):

```
sage: F = GF(2)
sage: R = MPolynomialRing(F,2,names = ["x","y"])
sage: x,y = R.gens()
sage: f = x^3*y+y^3+x
sage: places_on_curve(f,F)
((0, 1, 0), (1, 0, 0), (0, 0, 1))
```

2.4.4 Singular: Another approach

There is also a more Python-like interface to Singular. Using this, the code is much simpler, as illustrated below. First, we demonstrate computing the places on a curve in a particular case.

```
sage: singular.lib('brnoeth.lib')
sage: R = singular.ring(5, '(x,y)', 'lp')
sage: f = singular.new('y^2 - x^9 - x')
sage: X1 = f.Adj_div()
sage: X2 = singular.NSplaces(1, X1)
sage: X3 = singular.extcurve(1, X2)
sage: R = X3[1][5]
sage: singular.set_ring(R)
sage: L = singular.new('POINTS')

sage: [(L[i][1], L[i][2], L[i][3]) for i in range(1,7)]
[(0, 1, 0), (2, 2, 1), (0, 0, 1), (-2, -1, 1), (-2, 1, 1), (2, -2, 1)] # 32-bit
[(0, 1, 0), (-2, 1, 1), (-2, -1, 1), (2, 2, 1), (0, 0, 1), (2, -2, 1)] # 64-bit
```

Next, we implement the general function (for brevity we omit the docstring, which is the same as above). Note that the `point_parser` function is not required.

```
def places_on_curve(f,F):
    p = F.characteristic()
```

```

if F.degree() > 1:
    raise NotImplementedError
singular.lib('brnoeth.lib')
R = singular.ring(5, '(x,y)', 'lp')
f = singular.new('y^2 - x^9 - x')
X1 = f.Adj_div()
X2 = singular.NSplaces(1, X1)
X3 = singular.extcurve(1, X2)
R = X3[1][5]
singular.setring(R)
L = singular.new('POINTS')
return [(int(L[i][1]), int(L[i][2]), int(L[i][3])) \
        for i in range(1,int(L.size()+1))]

```

This code is much shorter, nice, and more readable. However, it depends on certain functions, e.g. `singular.setring` having been implemented in the Sage/Singular interface, whereas the code in the previous section used only the barest minimum of that interface.

2.4.5 Creating a new pseudo-tty interface

You can create Sage pseudo-tty interfaces that allow Sage to work with almost any command line program, and which do not require any modification or extensions to that program. They are also surprisingly fast and flexible (given how they work!), because all I/O is buffered, and because interaction between Sage and the command line program can be non-blocking (asynchronous). A pseudo-tty Sage interface is asynchronous because it derives from the Sage class `Expect`, which handles the communication between Sage and the external process.

For example, here is part of the file `SAGE_ROOT/devel/sage/sage/interfaces/octave.py`, which defines an interface between Sage and Octave, an open source program for doing numerical computations, among other things.

```

import os
from expect import Expect, ExpectElement

class Octave(Expect):
    ...

```

The first two lines import the library `os`, which contains operating system routines, and also the class `Expect`, which is the basic class for interfaces. The third line defines the class `Octave`; it derives from `Expect` as well. After this comes a docstring, which we omit here (see the file for details). Next comes:

```

def __init__(self, maxread=100, script_subdirectory="", logfile=None,
             server=None, server_tmpdir=None):
    Expect.__init__(self,
                    name = 'octave',
                    prompt = '>',
                    command = "octave --no-line-editing --silent",
                    maxread = maxread,
                    server = server,
                    server_tmpdir = server_tmpdir,
                    script_subdirectory = script_subdirectory,
                    restart_on_ctrlc = False,
                    verbose_start = False,
                    logfile = logfile,
                    eval_using_file_cutoff=100)

```

This uses the class `Expect` to set up the Octave interface.

```

def set(self, var, value):
    """
    Set the variable var to the given value.
    """
    cmd = '%s=%s;'%(var,value)
    out = self.eval(cmd)
    if out.find("error") != -1:
        raise TypeError, "Error executing code in Octave\nCODE:\n\t%s\nOctave ERROR:\n\t:%s"%(cmd, out)

def get(self, var):
    """
    Get the value of the variable var.
    """
    s = self.eval('%s'%var)
    i = s.find('=')
    return s[i+1:]

def console(self):
    octave_console()

```

These let users type `octave.set('x', 3)`, after which `octave.get('x')` returns `' 3'`. Running `octave.console()` dumps the user into an Octave interactive shell.

```

def solve_linear_system(self, A, b):
    """
    Use octave to compute a solution x to A*x = b, as a list.

    INPUT:
        A -- mxn matrix A with entries in QQ or RR
        b -- m-vector b entries in QQ or RR (resp)

    OUTPUT:
        An list x (if it exists) which solves M*x = b

    EXAMPLES:
        sage: M33 = MatrixSpace(QQ,3,3)
        sage: A   = M33([1,2,3,4,5,6,7,8,0])
        sage: V3  = VectorSpace(QQ,3)
        sage: b   = V3([1,2,3])
        sage: octave.solve_linear_system(A,b) # requires optional octave
        [-0.333332999999999999, 0.666667000000000001, -3.5236600000000002e-18]

    AUTHOR: David Joyner and William Stein
    """
    m = A.nrows()
    n = A.ncols()
    if m != len(b):
        raise ValueError, "dimensions of A and b must be compatible"
    from sage.matrix.all import MatrixSpace
    from sage.rings.all import QQ
    MS = MatrixSpace(QQ,m,1)
    b = MS(list(b)) # converted b to a "column vector"
    sA = self.sage2octave_matrix_string(A)
    sb = self.sage2octave_matrix_string(b)
    self.eval("a = " + sA )
    self.eval("b = " + sb )
    soln = octave.eval("c = a \\ b")
    soln = soln.replace("\n\n ", "[")

```

```

soln = soln.replace("\n\n", "]")
soln = soln.replace("\n", ", ")
sol  = soln[3:]
return eval(sol)

```

This code defines the method `solve_linear_system`, which works as documented.

These are only excerpts from `octave.py`; check that file for more definitions and examples. Look at other files in the directory `SAGE_ROOT/devel/sage/sage/interfaces/` for examples of interfaces to other software packages.

2.5 Doctesting the Sage Library

Doctesting a function ensures that the function performs as claimed by its documentation. Testing can be performed using one thread or multiple threads. After compiling a source version of Sage, doctesting can be run on the whole Sage library, on all modules under a given directory, or on a specified module only. For the purposes of this chapter, suppose we have compiled Sage 4.8 from source and the top level Sage directory is

```

[jdemeyer@sage sage-4.8]$ pwd
/scratch/jdemeyer/build/sage-4.8

```

See the section *Automated testing* for information on Sage's automated testing process. The general syntax for doctesting is as follows. To doctest a module in the library of a version of Sage, use this syntax:

```

/path/to/sage-x.y.z/sage -t [--long] /path/to/sage-x.y.z/path/to/module.py[x]

```

where `--long` is an optional argument. The version of `sage` used must match the version of Sage containing the module we want to doctest. A Sage module can be either a Python script (with the file extension `".py"`) or it can be a Cython script, in which case it has the file extension `".pyx"`.

2.5.1 Testing a module

Say we want to run all tests in the `sudoku` module `sage/games/sudoku.py`. In a terminal window, first we `cd` to the top level Sage directory of our local Sage installation. Now we can start doctesting as demonstrated in the following terminal session:

```

[jdemeyer@sage sage-4.8]$ ./sage -t devel/sage-main/sage/games/sudoku.py
sage -t "devel/sage-main/sage/games/sudoku.py"
[7.3 s]

```

```

-----
All tests passed!
Total time for all tests: 7.3 seconds

```

The numbers output by the test show that testing the `sudoku` module takes about six seconds, while testing all specified modules took the same amount of time. In this case, we only tested one module so it is not surprising that the total testing time is approximately the same as the time required to test only that one module. Notice that the syntax is

```

[jdemeyer@sage sage-4.8]$ ./sage -t devel/sage-main/sage/games/sudoku.py
sage -t "devel/sage-main/sage/games/sudoku.py"
[7.3 s]

```

```

-----
All tests passed!
Total time for all tests: 7.3 seconds

```

```
[jdemeyer@sage sage-4.8]$ ./sage -t "devel/sage-main/sage/games/sudoku.py"
sage -t "devel/sage-main/sage/games/sudoku.py"
[7.5 s]
```

```
-----
All tests passed!
Total time for all tests: 7.6 seconds
```

but not

```
[jdemeyer@sage sage-4.8]$ ./sage -t sage/games/sudoku.py
ERROR: File ./sage/games/sudoku.py is missing
exit code: 1
```

```
-----
The following tests failed:
```

```
./sage/games/sudoku.py
Total time for all tests: 0.0 seconds
[jdemeyer@sage sage-4.8]$ ./sage -t "sage/games/sudoku.py"
ERROR: File ./sage/games/sudoku.py is missing
```

```
-----
The following tests failed:
```

```
./sage/games/sudoku.py # File not found
Total time for all tests: 0.0 seconds
```

We can also first `cd` to the directory containing the module `sudoku.py` and doctest that module as follows:

```
[jdemeyer@sage sage-4.8]$ cd devel/sage-main/sage/games/
[jdemeyer@sage games]$ ls
__init__.py  hexad.py      sudoku.py      sudoku_backtrack.pyx
all.py       quantumino.py sudoku_backtrack.c
[jdemeyer@sage games]$ ../../../../sage -t sudoku.py
sage -t "devel/sage-main/sage/games/sudoku.py"
[7.1 s]
```

```
-----
All tests passed!
Total time for all tests: 7.1 seconds
```

In all of the above terminal sessions, we used a local installation of Sage to test its own modules. Even if we have a system-wide Sage installation, using that version to doctest the modules of a local installation is a recipe for confusion.

2.5.2 Troubleshooting

To doctest modules of a Sage installation, from a terminal window we first `cd` to the top level directory of that Sage installation, otherwise known as the `SAGE_ROOT` of that installation. When we run tests, we use that particular Sage installation via the syntax `./sage`; notice the “dot-forward-slash” at the front of `sage`. This is a precaution against confusion that can arise when our system has multiple Sage installations. For example, the following syntax is acceptable because we explicitly specify the Sage installation in the current `SAGE_ROOT`:

```
[jdemeyer@sage sage-4.8]$ ./sage -t devel/sage-main/sage/games/sudoku.py
./sage -t "devel/sage-main/sage/games/sudoku.py"
[6.9 s]
```

```
-----
All tests passed!
Total time for all tests: 6.9 seconds
[jdemeyer@sage sage-4.8]$ ./sage -t "devel/sage-main/sage/games/sudoku.py"
sage -t "devel/sage-main/sage/games/sudoku.py"
      [7.7 s]
```

```
-----
All tests passed!
Total time for all tests: 7.7 seconds
```

The following syntax is not recommended as we are using a system-wide Sage installation (if it exists):

```
[jdemeyer@sage sage-4.8]$ sage -t devel/sage-main/sage/games/sudoku.py
sage -t "devel/sage-main/sage/games/sudoku.py"
*****
File "/home/jdemeyer/sage/sage-4.8/devel/sage-main/sage/games/sudoku.py", line 515:
    sage: h.solve(algorithm='backtrack').next()
Exception raised:
Traceback (most recent call last):
  File "/usr/local/sage/local/bin/ncadoctest.py", line 1231, in run_one_test
    self.run_one_example(test, example, filename, compileflags)
  File "/usr/local/sage/local/bin/sagedoctest.py", line 38, in run_one_example
    OrigDocTestRunner.run_one_example(self, test, example, filename, compileflags)
  File "/usr/local/sage/local/bin/ncadoctest.py", line 1172, in run_one_example
    compileflags, 1) in test.globs
  File "<doctest __main__.example_13[4]>", line 1, in <module>
    h.solve(algorithm='backtrack').next()###line 515:
sage: h.solve(algorithm='backtrack').next()
  File "/home/jdemeyer/.sage/tmp/sudoku.py", line 607, in solve
    for soln in gen:
  File "/home/jdemeyer/.sage/tmp/sudoku.py", line 719, in backtrack
    from sudoku_backtrack import backtrack_all
ImportError: No module named sudoku_backtrack
*****
[...more errors...]
2 items had failures:
  4 of 15 in __main__.example_13
  2 of  8 in __main__.example_14
***Test Failed*** 6 failures.
For whitespace errors, see the file /home/jdemeyer/.sage//tmp/.doctest_sudoku.py
      [21.1 s]
```

The following tests failed:

```
      sage -t "devel/sage-main/sage/games/sudoku.py"
Total time for all tests: 21.3 seconds
```

In this case, we received an error because the system-wide Sage installation is a different (older) version than the one we are using for Sage development. Make sure you always test the files with the correct version of Sage.

2.5.3 Parallel testing many modules

So far we have used a single thread to doctest a module in the Sage library. There are hundreds, even thousands of modules in the Sage library. Testing them all using one thread would take a few hours. Depending on our hardware, this could take up to six hours or more. On a multi-core system, parallel doctesting can significantly reduce the testing

time. Unless we also want to use our computer while doctesting in parallel, we can choose to devote all the cores of our system for parallel testing.

Let us doctest all modules in a directory, first using a single thread and then using four threads. For this example, suppose we want to test all the modules under `sage/crypto/`. We can use a syntax similar to that shown above to achieve this:

```
[jdemeyer@sage sage-4.8]$ ./sage -t devel/sage-main/sage/crypto/
sage -t "devel/sage-main/sage/crypto/block_cipher/__init__.py"
[0.1 s]
sage -t "devel/sage-main/sage/crypto/block_cipher/miniaes.py"
[5.5 s]
sage -t "devel/sage-main/sage/crypto/block_cipher/all.py"
[0.1 s]
sage -t "devel/sage-main/sage/crypto/block_cipher/sdes.py"
[4.2 s]
sage -t "devel/sage-main/sage/crypto/__init__.py"
[0.1 s]
sage -t "devel/sage-main/sage/crypto/stream.py"
[3.7 s]
sage -t "devel/sage-main/sage/crypto/classical_cipher.py"
[5.1 s]
sage -t "devel/sage-main/sage/crypto/boolean_function.pyx"
[7.3 s]
sage -t "devel/sage-main/sage/crypto/lattice.py"
[3.7 s]
sage -t "devel/sage-main/sage/crypto/util.py"
[3.4 s]
sage -t "devel/sage-main/sage/crypto/cryptosystem.py"
[3.6 s]
sage -t "devel/sage-main/sage/crypto/all.py"
[0.1 s]
sage -t "devel/sage-main/sage/crypto/mq/__init__.py"
[0.1 s]
sage -t "devel/sage-main/sage/crypto/mq/sbox.py"
[4.4 s]
sage -t "devel/sage-main/sage/crypto/mq/mpolynomialssystem.py"
[12.8 s]
sage -t "devel/sage-main/sage/crypto/mq/sr.py"
[10.6 s]
sage -t "devel/sage-main/sage/crypto/mq/mpolynomialssystemgenerator.py"
[3.4 s]
sage -t "devel/sage-main/sage/crypto/cipher.py"
[3.4 s]
sage -t "devel/sage-main/sage/crypto/classical.py"
[13.8 s]
sage -t "devel/sage-main/sage/crypto/public_key/blum_goldwasser.py"
[3.5 s]
sage -t "devel/sage-main/sage/crypto/public_key/__init__.py"
[0.1 s]
sage -t "devel/sage-main/sage/crypto/public_key/all.py"
[0.1 s]
sage -t "devel/sage-main/sage/crypto/stream_cipher.py"
[3.4 s]
sage -t "devel/sage-main/sage/crypto/lfsr.py"
[3.5 s]
```

```
-----
All tests passed!
```

Total time for all tests: 96.1 seconds

Now we do the same thing, but this time we also use the optional argument `--long`:

```
[jdemeyer@sage sage-4.8]$ ./sage -t --long devel/sage-main/sage/crypto/
sage -t --long "devel/sage-main/sage/crypto/block_cipher/__init__.py"
    [0.1 s]
sage -t --long "devel/sage-main/sage/crypto/block_cipher/miniaes.py"
    [4.1 s]
sage -t --long "devel/sage-main/sage/crypto/block_cipher/all.py"
    [0.1 s]
sage -t --long "devel/sage-main/sage/crypto/block_cipher/sdes.py"
    [3.9 s]
sage -t --long "devel/sage-main/sage/crypto/__init__.py"
    [0.0 s]
sage -t --long "devel/sage-main/sage/crypto/stream.py"
    [3.3 s]
sage -t --long "devel/sage-main/sage/crypto/classical_cipher.py"
    [3.9 s]
sage -t --long "devel/sage-main/sage/crypto/boolean_function.py"
    [7.2 s]
sage -t --long "devel/sage-main/sage/crypto/lattice.py"
    [3.4 s]
sage -t --long "devel/sage-main/sage/crypto/util.py"
    [3.3 s]
sage -t --long "devel/sage-main/sage/crypto/cryptosystem.py"
    [3.4 s]
sage -t --long "devel/sage-main/sage/crypto/all.py"
    [0.1 s]
sage -t --long "devel/sage-main/sage/crypto/mq/__init__.py"
    [0.1 s]
sage -t --long "devel/sage-main/sage/crypto/mq/sbox.py"
    [3.5 s]
sage -t --long "devel/sage-main/sage/crypto/mq/mpolynomialssystem.py"
    [11.8 s]
sage -t --long "devel/sage-main/sage/crypto/mq/sr.py"
    [96.8 s]
sage -t --long "devel/sage-main/sage/crypto/mq/mpolynomialssystemgenerator.py"
    [2.9 s]
sage -t --long "devel/sage-main/sage/crypto/cipher.py"
    [3.2 s]
sage -t --long "devel/sage-main/sage/crypto/classical.py"
    [13.6 s]
sage -t --long "devel/sage-main/sage/crypto/public_key/blum_goldwasser.py"
    [3.2 s]
sage -t --long "devel/sage-main/sage/crypto/public_key/__init__.py"
    [0.1 s]
sage -t --long "devel/sage-main/sage/crypto/public_key/all.py"
    [0.1 s]
sage -t --long "devel/sage-main/sage/crypto/stream_cipher.py"
    [3.4 s]
sage -t --long "devel/sage-main/sage/crypto/lfsr.py"
    [3.0 s]
```

All tests passed!

Total time for all tests: 174.3 seconds

Notice the time difference between the first set of tests and the second set, which uses the optional argument `--long`.

Many tests in the Sage library are flagged with `# long time` because these are known to take a long time to run through. Without using the optional `--long` argument, the module `sage/crypto/mq/sr.py` took about ten seconds. With this optional argument, it required 97 seconds to run through all tests in that module. Here is a snippet of a function in the module `sage/crypto/mq/sr.py` with a doctest that has been flagged as taking a long time:

```
def test_consistency(max_n=2, **kwargs):
    r"""
    Test all combinations of ``r``, ``c``, ``e`` and ``n`` in ``(1,
    2)`` for consistency of random encryptions and their polynomial
    systems. ``GF{2}`` and ``GF{2^e}`` systems are tested. This test takes
    a while.

    INPUT:

    - ``max_n`` -- maximal number of rounds to consider (default: 2)
    - ``kwargs`` -- are passed to the SR constructor

    TESTS:

    The following test called with ``max_n`` = 2 requires a LOT of RAM
    (much more than 2GB). Since this might cause the doctest to fail
    on machines with "only" 2GB of RAM, we test ``max_n`` = 1, which
    has a more reasonable memory usage. ::

        sage: from sage.crypto.mq.sr import test_consistency
        sage: test_consistency(1) # long time (80s on sage.math, 2011)
        True
    """
```

Now we doctest the same directory in parallel using 4 threads:

```
[jdemeyer@sage sage-4.8]$ ./sage -tp 4 devel/sage-main/sage/crypto/
Global iterations: 1
File iterations: 1
Using cached timings to run longest doctests first.
Doctesting 24 files doing 4 jobs in parallel
sage -t devel/sage-main/sage/crypto/__init__.py
[0.1 s]
sage -t devel/sage-main/sage/crypto/lattice.py
[3.3 s]
sage -t devel/sage-main/sage/crypto/stream.py
[3.5 s]
sage -t devel/sage-main/sage/crypto/classical_cipher.py
[4.0 s]
sage -t devel/sage-main/sage/crypto/all.py
[0.1 s]
sage -t devel/sage-main/sage/crypto/util.py
[3.4 s]
sage -t devel/sage-main/sage/crypto/cryptosystem.py
[3.4 s]
sage -t devel/sage-main/sage/crypto/boolean_function.pyx
[6.9 s]
sage -t devel/sage-main/sage/crypto/cipher.py
[3.3 s]
sage -t devel/sage-main/sage/crypto/block_cipher/__init__.py
[0.1 s]
sage -t devel/sage-main/sage/crypto/lfsr.py
[3.3 s]
sage -t devel/sage-main/sage/crypto/stream_cipher.py
```

```

[3.4 s]
sage -t devel/sage-main/sage/crypto/block_cipher/all.py
[0.1 s]
sage -t devel/sage-main/sage/crypto/mq/__init__.py
[0.1 s]
sage -t devel/sage-main/sage/crypto/block_cipher/miniaes.py
[4.0 s]
sage -t devel/sage-main/sage/crypto/block_cipher/sdes.py
[3.6 s]
sage -t devel/sage-main/sage/crypto/mq/sbox.py
[4.0 s]
sage -t devel/sage-main/sage/crypto/mq/mpolynomialssystemgenerator.py
[3.2 s]
sage -t devel/sage-main/sage/crypto/public_key/blum_goldwasser.py
[3.4 s]
sage -t devel/sage-main/sage/crypto/public_key/__init__.py
[0.1 s]
sage -t devel/sage-main/sage/crypto/classical.py
[14.3 s]
sage -t devel/sage-main/sage/crypto/public_key/all.py
[0.1 s]
sage -t devel/sage-main/sage/crypto/mq/sr.py
[9.3 s]
sage -t devel/sage-main/sage/crypto/mq/mpolynomialssystem.py
[12.0 s]

```

```

-----
All tests passed!
Timings have been updated.
Total time for all tests: 23.7 seconds
[jdemeyer@sage sage-4.8]$ ./sage -tp 4 --long devel/sage-main/sage/crypto/
Global iterations: 1
File iterations: 1
Using long cached timings to run longest doctests first.
Doctesting 24 files doing 4 jobs in parallel
sage -t --long devel/sage-main/sage/crypto/__init__.py
[0.1 s]
sage -t --long devel/sage-main/sage/crypto/stream.py
[3.2 s]
sage -t --long devel/sage-main/sage/crypto/lattice.py
[3.3 s]
sage -t --long devel/sage-main/sage/crypto/classical_cipher.py
[4.1 s]
sage -t --long devel/sage-main/sage/crypto/all.py
[0.1 s]
sage -t --long devel/sage-main/sage/crypto/util.py
[3.1 s]
sage -t --long devel/sage-main/sage/crypto/cryptosystem.py
[3.3 s]
sage -t --long devel/sage-main/sage/crypto/boolean_function.pyx
[7.0 s]
sage -t --long devel/sage-main/sage/crypto/cipher.py
[3.2 s]
sage -t --long devel/sage-main/sage/crypto/block_cipher/__init__.py
[0.1 s]
sage -t --long devel/sage-main/sage/crypto/stream_cipher.py
[3.2 s]
sage -t --long devel/sage-main/sage/crypto/block_cipher/all.py

```

```
[0.1 s]
sage -t --long devel/sage-main/sage/crypto/lfsr.py
[3.4 s]
sage -t --long devel/sage-main/sage/crypto/mq/___init___.py
[0.1 s]
sage -t --long devel/sage-main/sage/crypto/block_cipher/miniaes.py
[4.2 s]
sage -t --long devel/sage-main/sage/crypto/block_cipher/sdes.py
[4.0 s]
sage -t --long devel/sage-main/sage/crypto/mq/sbox.py
[3.8 s]
sage -t --long devel/sage-main/sage/crypto/mq/mpolynomialssystemgenerator.py
[3.1 s]
sage -t --long devel/sage-main/sage/crypto/classical.py
[13.8 s]
sage -t --long devel/sage-main/sage/crypto/public_key/___init___.py
[0.0 s]
sage -t --long devel/sage-main/sage/crypto/public_key/all.py
[0.0 s]
sage -t --long devel/sage-main/sage/crypto/public_key/blum_goldwasser.py
[3.1 s]
sage -t --long devel/sage-main/sage/crypto/mq/mpolynomialssystem.py
[11.3 s]
sage -t --long devel/sage-main/sage/crypto/mq/sr.py
[95.4 s]
```

```
-----
All tests passed!
Total time for all tests: 109.4 seconds
```

As the number of threads increases, the total testing time decreases. To minimize confusion, it is also a good idea to explicitly specify the path name of the directory we want to doctest and not a symbolic link to that directory. In the above examples, the symbolic link `devel/sage` points to the directory `devel/sage-main`, but the actual path to the directory has been specified instead of its symbolic link.

2.5.4 Parallel testing the whole Sage library

The main Sage library resides in the directory `SAGE_ROOT/devel/sage-main/`. We can use the syntax described above to doctest the main library using multiple threads. When doing release management or patching the main Sage library, a release manager would parallel test the library using 10 threads with the following command:

```
[jdemeyer@sage sage-4.8]$ ./sage -tp 10 -long devel/sage-main/
```

Another way is run `make ptestlong`, which builds Sage (if necessary), builds the Sage documentation (if necessary), and then runs parallel doctests. This determines the number of threads by reading the environment variable `MAKE`: if it is set to `make -j12`, then use 12 threads. If `MAKE` is not set, then by default it uses the number of CPU cores (as determined by the Python function `multiprocessing.cpu_count()`) with a minimum of 2 and a maximum of 8.

In any case, this will test the Sage library with multiple threads:

```
[jdemeyer@sage sage-4.8]$ make ptestlong
```

Any of the following commands would also doctest the Sage library or one of its clones:

```
make test
make check
```

```
make testlong
make ptest
make ptestlong
```

In each case, testing is performed on the directory that is pointed to by the symbolic link `devel/sage`.

- `make test` and `make check` — These two commands run the same set of tests. First the Sage standard documentation is tested, i.e. the documentation that resides in
 - `SAGE_ROOT/devel/sage/doc/common`
 - `SAGE_ROOT/devel/sage/doc/en`
 - `SAGE_ROOT/devel/sage/doc/fr`

Finally, the commands `doctest` the Sage library. For more details on these command, see the files `SAGE_ROOT/Makefile` and `SAGE_ROOT/local/bin/sage-maketest`.

- `make testlong` — This command `doctests` the standard documentation:

- `SAGE_ROOT/devel/sage/doc/common`
- `SAGE_ROOT/devel/sage/doc/en`
- `SAGE_ROOT/devel/sage/doc/fr`

and then the Sage library. Doctesting is run with the optional argument `-long`. See the file `SAGE_ROOT/Makefile` for further details.

- `make ptest` — Similar to the commands `make test` and `make check`. However, doctesting is run with the number of threads as described above for `make ptestlong`.
- `make ptestlong` — Similar to the command `make ptest`, but using the optional argument `-long` for doctesting.

2.5.5 Beyond the Sage library

The doctesting scripts of a Sage installation currently have limited support for doctesting of modules outside of the Sage library for that version of Sage. We cannot use the doctesting scripts of Sage 4.1.1 to doctest modules in, say, Sage 4.1. Doing so would result in errors:

```
[mvngu@sage sage-4.1.1]$ ./sage -t ../sage-4.1/devel/sage-main/sage/games/sudoku.py
sage -t "../sage-4.1/devel/sage-main/sage/games/sudoku.py"
File "./sudoku.py", line 18
    from ../sage-4.1/devel/sage-main/sage/games/sudoku import *
    ^
SyntaxError: invalid syntax

[0.2 s]
exit code: 1024
```

The following tests failed:

```
    sage -t "../sage-4.1/devel/sage-main/sage/games/sudoku.py"
Total time for all tests: 0.2 seconds
```

However, suppose we have a Python script called `my_python_script.py` that uses the Sage library. Our Python script has the following content:

```
[mvngu@sage build]$ cat my_python_script.py
from sage.all_cmdline import * # import sage library

def square(n):
    """
    Return the square of n.

    EXAMPLES::

        sage: square(2)
        4
    """
    return n**2
```

We can use any version of Sage to doctest our Python script, so long as that version of Sage has features that are used in our script. For example, we can use both Sage 4.1.1 and 4.1 to doctest the above Python script:

```
[mvngu@sage build]$ sage-4.1/sage -t my_python_script.py
sage -t "my_python_script.py"
[1.3 s]
```

```
-----
All tests passed!
Total time for all tests: 1.3 seconds
[mvngu@sage build]$ sage-4.1.1/sage -t my_python_script.py
sage -t "my_python_script.py"
[1.4 s]
```

```
-----
All tests passed!
Total time for all tests: 1.4 seconds
```

Doctesting can also be performed on Sage scripts. Say we have a Sage script called `my_sage_script.sage` with the following content:

```
[mvngu@sage build]$ cat my_sage_script.sage
def cube(n):
    r"""
    Return the cube of n.

    EXAMPLES::

        sage: cube(2)
        8
    """
    return n**3
```

This must be converted to an equivalent Python script prior to doctesting. First, we use Sage to convert `my_sage_script.sage` to an equivalent Python script called `my_sage_script.py`:

```
[mvngu@sage build]$ sage-4.1.1/sage my_sage_script.sage
[mvngu@sage build]$ cat my_sage_script.py
# This file was *autogenerated* from the file my_sage_script.sage.
from sage.all_cmdline import * # import sage library
_sage_const_3 = Integer(3)
def cube(n):
    r"""
    Return the cube of n.
```

EXAMPLES::

```
sage: cube(2)
      8
"""
return n**_sage_const_3
```

Doctesting is then performed on that equivalent Python script:

```
[mvngu@sage build]$ sage-4.1.1/sage -t my_sage_script.py
sage -t "my_sage_script.py"
[1.5 s]
```

```
-----
All tests passed!
Total time for all tests: 1.5 seconds
```

2.6 The Sage Manuals

This chapter describes how to modify the Sage manuals. Sage's manuals are written in ReST, otherwise known as [reStructuredText](#). To edit them, you just need to edit the appropriate file. The documentation builder is called [Sphinx](#).

Here is a list of the Sage manuals and the corresponding files to edit:

- The Sage tutorial: `SAGE_ROOT/devel/sage/doc/en/tutorial`
- The Sage developer's guide: `SAGE_ROOT/devel/sage/doc/en/developer`
- Constructions in Sage: `SAGE_ROOT/devel/sage/doc/en/constructions`
- The Sage installation guide: `SAGE_ROOT/devel/sage/doc/en/installation`
- The Sage reference manual: some of this is contained in the file `SAGE_ROOT/devel/sage/doc/en/reference`, but most of it is automatically generated from the Sage source code.
- Additional, more specialized manuals can be found under `SAGE_ROOT/devel/sage/doc/en` as well.

Note: You can edit manuals that have been translated into another language by replacing the `en/` above with the appropriate two letter language code. For example, the French tutorial is located in `SAGE_ROOT/devel/sage/doc/fr/tutorial`

2.6.1 Editing the manuals

If, for example, you want to change the Sage tutorial, then you should start by modifying the files in `SAGE_ROOT/devel/sage/doc/en/tutorial/`. Then to build a PDF file with your changes, type:

```
sage --docbuild tutorial pdf
```

You will get a file `tutorial.pdf` in `SAGE_ROOT/devel/sage/doc/output/pdf/en/tutorial` which you should inspect. You can build the HTML version of the tutorial by typing:

```
sage --docbuild tutorial html
```

Once you have done this, you can access the new HTML version from the notebook interface to Sage by clicking the Help link, or you can open the file

`SAGE_ROOT/devel/sage/doc/output/html/en/tutorial/index.html` in your web browser. For more detailed information about building the documentation, see *Building the manuals*.

You should also run

```
sage -tp 1 SAGE_ROOT/devel/sage/doc/en/tutorial/
```

to test all of the examples in the tutorial, see *Automated testing* for more details.

Finally, you might want to share your changes with the Sage community. To do this, use Mercurial (see *Producing Patches with Mercurial*) to produce patch files, and submit them to the Sage trac server.

As noted above, the reference manual is mostly autogenerated from Sage source code. To build it, type:

```
sage -b <repo-name>
sage --docbuild reference <format>
```

where `<repo-name>` is the name of the repository you are using, and `<format>` is `html`, `pdf`, or any other supported format (as listed when you run `sage --docbuild --formats`).

2.6.2 Setting hyperlink to modules, classes, methods, etc.

For full documentation, refer to [inline markup](#) in the Sphinx documentation. Currently, there is no support for defining chapters and labels in the autogenerated documentation. However, it is possible to generate a link to the documentation for any module, class, method, function, etc. The syntax is

```
:role:`title <target>`
```

or

```
:role:`target`
```

where

- `role` is the kind of thing you want to link to (i.e. `mod` for module, `class` for classes, `meth` for methods, `func` for functions, etc);
- `target` is the Python name of the object (class, module, method, etc.) which carries the documentation to be linked to;
- `title` is the name of the link as shown in the browser.

If you do not provide any title then `target` will be used. For example, to link to the `dyck_word` module, you would use `:mod:`sage.combinat.dyck_word`` or if you prefer `:mod:`Dyck words<sage.combinat.dyck_word>``. Note that, in the first case, the full qualified Python address is used which is usually too long. You can prefix it with a `"~"` to get only the final name. For example, in the huge link

```
:meth:`~sage.combinat.non_decreasing_parking_function.NonDecreasingParkingFunction.to_dyck_word`
```

only `".to_dyck_word()"` will appear. Note that the parentheses in the link are autogenerated.

Finally, local names are handled. That is, for example, in the definition of a class (and any of its members or methods), you can link to any member or method of the same class by simply giving the name of it prepended by a dot `"."`. You do not need to give its full address. For example:

```
:meth:`.to_dyck_word`
```

sets up a link to the `.to_dyck_word()` method of the current class if it exists. If not, the documentation builder searches by going up in the class/module hierarchy of Python until it finds an object with this name or reaches the top-level module without finding it. If the name cannot be found, the title is typeset in boldface without any link

produced and also without any error or warning. Note that without the prepended dot, the object is searched starting from the top-level to the innermost module or class.

2.6.3 Adding a new file

If you write a new file, say, `sage/combinat/family.py`, and you want your documentation to be added to the standard documentation, you have to add your file to the relevant `index.rst` file usually located in the tree:

```
SAGE_ROOT/devel/sage/doc/en/reference
```

For this example, you would need to add to the file

```
SAGE_ROOT/devel/sage/doc/en/reference/combinat/index.rst
```

the following line

```
Combinatorics
=====

.. toctree::
   :maxdepth: 2

   ../sage/combinat/combinat
   [...]
   ../sage/combinat/dyck_word
+  ../sage/combinat/family
   ../sage/combinat/finite_class
   [...]
```

2.6.4 Building the manuals

All of the Sage manuals are built using the `sage --docbuild` script. The content of the `sage --docbuild` script is defined in `SAGE_ROOT/devel/sage/doc/common/builder.py`. It is a thin wrapper around the `sphinx-build` script which does all of the real work. It is designed to be a replacement for the default Makefiles generated by the `sphinx-quickstart` script. The general form of the command is

```
sage --docbuild <document-name> <format>
```

as explained below.

For more information, there are two help commands which give plenty of documentation for the `sage --docbuild` script:

```
sage --docbuild --help
```

(or `-h`) gives a basic listing of options and further help commands, while:

```
sage --docbuild --help-all
```

(or `-H`) shows a somewhat more comprehensive help message.

Document names

The `<document-name>` has the form

lang/name

where `lang` is a two-letter language code, and `name` is the descriptive name of the document. If the language is not specified, then it defaults to English (`en`). The following two commands do the exact same thing:

```
sage --docbuild tutorial html
sage --docbuild en/tutorial html
```

To specify the French version of the tutorial, you would simply run:

```
sage --docbuild fr/tutorial html
```

Output formats

The Sage documentation build system currently supports all of the output formats that Sphinx does.

For more detailed information, see the documentation on builders at <http://sphinx.pocoo.org/builders.html>.

2.6.5 Syntax highlighting Cython code

If you need to put *Cython* code in a ReST file, you can either precede the code block by `.. code-block:: cython` instead of the usual `::` if you want to highlight one block of code in Cython, or you can use `.. highlight:: cython` for a whole file.

The following example was generated by `.. code-block:: cython`:

```
cdef extern from "descrobject.h":
    ctypedef struct PyMethodDef:
        void *ml_meth
    ctypedef struct PyMethodDescrObject:
        PyMethodDef *d_method
    void* PyCFunction_GET_FUNCTION(object)
    bint PyCFunction_Check(object)
```

DISSEMINATING CODE FOR SAGE

Whether you have developed some new code for Sage or just have a simple bug fix, you need to know how to communicate what you have done to other Sage users. This part of the guide discusses this issue. Here are some of the available avenues of communication and tools to aid in that communication:

- The Google group `sage-support`, at <http://groups.google.com/group/sage-support>. This group gives help and support to those who have problems with Sage (installation, syntax, etc). The IRC channel `#sage-devel` on freenode serves the same purpose.
- The Google group `sage-devel`, at <http://groups.google.com/group/sage-devel>. This mailing list is for Sage development and is about programming, design and technical issues. This is a great place to post questions about whether certain behavior is a bug, or whether a certain feature ought to be implemented (or how it ought to be implemented), or similar issues. Also implementation issues are discussed here and the general direction of the project. Development discussion also takes place on the IRC channel `#sage-devel` on freenode.
- Mercurial: this is the source control system that is included with Sage. Use this to produce patches for Sage. See the chapter *Walking Through the Development Process* and the section *Producing Patches with Mercurial* for tutorials on using Mercurial to produce and manage patches.
- The Sage trac server, at http://trac.sagemath.org/sage_trac/. this is where you should post bugs, patches for bugs, additions to the Sage library, etc. See *The Sage Trac Server: Submitting Patches and Packages* for more information.

Contents:

3.1 Inclusion Procedure for New Packages

For a package to become part of Sage's standard distribution, it must meet the following requirements:

- **License.** The license must be a GPL version 2+ compatible license.
- **Build Support.** The code **must** build on all the **fully supported platforms**.

A standard package should also work on all the platforms where Sage is **expected to work**, but since we don't fully support these platforms and often lack the resources to test on them, you are not expected to confirm your packages works on those platforms. However, if you can, it is better to do so. As noted [here](#), a failure of Sage to work on a platform where it is expected to work, will be considered a bug.

There is no need to worry too much about platforms where Sage will **probably not work** though if it's clear that there is significant effort taking place to port Sage to a platform, then you should aim to ensure your package does not cause unnecessary headaches to those working on the port.

If it's clear that a port is stagnant, with nobody working on it, then you can safely ignore it.

Remarks:

- Some Sage developers are willing to help you port to OS X, Solaris and Windows. But this is no guarantee and you or your project are expected to do the heavy lifting and also support those ports upstream if there is no Sage developer who is willing to share the burden.
- One of the best ways to ensure your code works on multiple platforms is to only use commands which are defined by [POSIX.1-2008](#) and only use options which are defined in the POSIX standard. For example, do not use the `-p` option to `uname` as the `'-p'` option is not defined by the POSIX standard, so is not portable. If you must use a non-POSIX command, or a option which is not defined by POSIX, then ensure the code only gets executed on the platform(s) where that command and/or option will be acceptable.
- **Quality.** The code should be “better” than any other available code (that passes the two above criteria), and the authors need to justify this. The comparison should be made to both Python and other software. Criteria in passing the quality test include:
 - Speed
 - Documentation
 - Usability
 - Memory leaks
 - Maintainable
 - Portability
 - Reasonable build time, size, dependencies
- **Previously an optional package.** Usually a new standard package must have spent some time as an optional package. However, sometimes this is not possible, if for example a new library is needed to permit an updated version of a standard package to function.
- **Refereeing.** The code must be refereed, as discussed in *The Sage Trac Server: Submitting Patches and Packages*.

3.2 Producing Patches with Mercurial

If you are editing or adding to Sage’s core library, you will probably want to share your changes with other users. Mercurial is the tool to do this. Mercurial is the source control system that is included with Sage. This chapter provides an overview of how to use Mercurial with Sage; see <http://www.selenic.com/mercurial/> for full documentation on Mercurial.

All of the Mercurial repositories related to Sage are included with Sage. Thus the complete change history and setup for doing development is available in your copy of Sage.

Before using Mercurial, make sure to define your username so the patches you make are identified as yours. Make a file `~/ .hgrc` in your home directory like this one:

```
[ui]
username = Euclid of Alexandria <euclid@alexandria.edu>
```

3.2.1 Quick Mercurial tutorial for Sage

To submit your changes to the Sage development team for refereeing (and inclusion into Sage if the referee’s report is positive), you should produce patch files using Mercurial. The simplest way is to run Mercurial from within Sage following the examples below (note: `Hg` is the chemical symbol for mercury).

- Type `hg_sage.status()` and `hg_sage.diff()` to see exactly what you have done. Use the command `q` to quit the diff.

- If you have added new files, not just edited existing ones, type `hg_sage.add([filenames])` to add those new files to your repository.

Warning: As noted in *Coding in Cython*, if you have added a Cython file, you also need to edit `SAGE_ROOT/devel/sage/module_list.py`. If you have added a new directory, you need to edit `SAGE_ROOT/devel/sage/setup.py`. If you have added something other than Python or Cython files, then you might need to add entries to the file `SAGE_ROOT/devel/sage/MANIFEST.in`: this records all of the files to include in distributions of the Sage library. Look at the file itself for examples, and see the Python documentation <http://docs.python.org/distutils/sourcedist.html#specifying-the-files-to-distribute> for all of the details.

- Commit your changes by typing `hg_sage.commit()` to commit the changes in files to the repository. If you want to commit only specific files, each file must be listed individually with full path names, e.g. `hg_sage.commit('sage/misc/misc.py sage/all.py')`. If no file names are given, all changed files are committed. First, the output of `hg diff` is displayed: look at it or just enter `q`. Then you are dumped into an editor to type a brief comment on the changes. The default editor is `vi`, so type `i` to insert, write a one line commit message of the form `trac xxxx: <your-commit-message-here>` where `xxxx` is the Sage development tracking system ticket number (see <http://trac.sagemath.org>). To quit the `vi` editor and save your commit message, hit `Escape` and type `:wq`. (In bash, to make `emacs` the default editor, type `export EDITOR=emacs`.)
- Now create a patch file using `hg_sage.export(...)`. This command needs a revision number (or list of revision numbers) as an argument; use `hg_sage.export('tip')` to use the most recent revision number or use `hg_sage.log()` to see all these numbers. An optional second argument to `hg_sage.export(...)` is a file name for the patch. The default is `(changeset_revision_number).patch`, which is written in what Sage considers the current directory (this can be found with the command `os.path.abspath('.')`).
- Then post your patch on the Sage Trac server: see *The Sage Trac Server: Submitting Patches and Packages*.

You can also run Mercurial directly from the command line using the command `sage -hg`. Or you can start a very nice web server that allows you to navigate your repository with a web browser, or pull patches from it remotely, by typing `hg_sage.serve()`. Then open your web browser and point it to <http://localhost:8000>, which is the default listening address for Mercurial.

Finally, if you want to apply a patch file (perhaps you have downloaded a patch from the Trac server for review), use the command `hg_sage.patch('filename')` (or `hg_sage.apply('filename')` for hg bundle files).

Before you modify Sage library files, you might want to create a copy of the Sage library in which to work. Do this by typing `sage -clone myver`, for example. Then Sage will use Mercurial to clone the current repository and call the result `myver`. The new repository is stored in `<SAGE_ROOT>/devel/sage-myver`, and when you clone, the symbolic link `sage --> sage-myver` is made.

(You can also do, e.g. `sage -clone -r 1250 oldver`, to get a clone of Sage as it was at revision 1250. Of course, dependency issues could make old versions not work (e.g. maybe an old Sage library would not compile with the latest Singular library, which is what is installed elsewhere in `SAGE_ROOT`). From within Sage, type `hg_sage.log()` to see the revision history. Note that if you clone an old version, all of the Cython code is rebuilt, since there is no easy way to know which files do and do not need rebuilding.)

Once you have copied the library to a new branch `myver` and edited some files there, you should build the Sage library to incorporate those changes. Type `sage -b myver`, or just `sage -b` if the branch `myver` is already the current branch, i.e. if `SAGE_ROOT/devel/sage` links to `SAGE_ROOT/devel/sage-myver`. You can also type `sage -br myver` to build the library and then to immediately run Sage.

3.2.2 Using Mercurial with other Sage repositories

Sage includes these Mercurial repositories:

- `SAGE_ROOT/devel/sage-*`: the Sage library source code.
- `SAGE_ROOT/data/extcode`: external system code, i.e. code included with Sage that is written for the systems with which Sage interfaces, e.g. GAP, PARI, etc.
- `SAGE_ROOT/local/bin`: Sage shell scripts.
- `SAGE_ROOT`: Sage root – text files in the main Sage directory and in `SAGE_ROOT/spkg`.

The previous section discussed using Mercurial with the Sage library, via the command `hg_sage`. There are corresponding commands for each of the repositories:

- use `hg_sage` for the Sage library
- use `hg_extcode` for the external system code
- use `hg_scripts` for the Sage shell scripts.
- use `hg_root` for the Sage root.

Since version 3.4, both the Sage library and documentation repositories are managed by the command `hg_sage`.

3.3 Producing New Sage Packages

If you are producing code to add new functionality to Sage, you might consider turning it into a package (an “spkg”) instead of a patch file. If your code is very large (for instance) and should be offered as an optional download, a package is the right choice. Similarly, if your code depends on some other optional component of Sage, you should produce a package. When in doubt, ask for advice on the `sage-devel` mailing list.

This chapter covers issues relevant to producing a package. The directory structure of a package is discussed along with scripts for installing a package and running the test suite (if any) contained in an upstream project's source distribution. For guidelines on patching an existing Sage package, see the chapter *Patching a Sage Package*.

3.3.1 Creating a new spkg

The abbreviation “spkg” stands for “Sage package”. The directory `SAGE_ROOT/spkg/standard` contains spkg's. In a source install, these are all Sage spkg files (actually `.tar` or `.tar.bz2` files), which are the source code that defines Sage. In a binary install, some of these may be small placeholder files to save space.

Sage packages are distributed as `.spkg` files, which are `.tar.bz2` files (or `.tar` files) but have the extension `.spkg` to discourage confusion. Although Sage packages are packed using tar and/or bzip2, note that `.spkg` files contain control information (installation scripts and metadata) that are necessary for building and installing them. For source distributions, when you compile Sage the file `SAGE_ROOT/Makefile` takes care of the unpacking, compilation, and installation of Sage packages for you. You can type

```
tar -jxvf mypackage-version.spkg
```

to extract an spkg and see what is inside. If you want to create a new Sage package, it is recommended that you start by examining some existing spkg's. In a source distribution of Sage, the standard spkg's can be found under `SAGE_ROOT/spkg/standard/`. The URL <http://www.sagemath.org/download-packages.html> lists standard spkg's available for download.

Naming your spkg

Each Sage spkg has a name of the following form:

BASENAME-VERSION.spkg

BASENAME is the name of the package; it may contain lower-case letters, numbers, and underscores, but no hyphens. VERSION is the version number; it should start with a number and may contain numbers, letters, dots, and hyphens; it may end in a string of the form “pNUM”, where “NUM” is a non-negative integer. If your spkg is a “vanilla” (unmodified) version of some piece of software, say version 5.3 of “my-python-package”, then BASENAME would be “my_python_package” – note the change from hyphens to underscores, because BASENAME should not contain any hyphens – and VERSION would be “5.3”. If you need to modify the software to use it with Sage (as described below and in the chapter *Patching a Sage Package*), then VERSION would be “5.3.p0”, the “p0” indicating a patch-level of 0. If someone adds more patches, later, this would become “p1”, then “p2”, etc.

The string VERSION must be present. If you are using a piece software with no obvious version number, use a date: you can see several such names among the standard Sage packages: <http://www.sagemath.org/packages/standard/>.

To give your spkg a name like this, create a directory called BASENAME-VERSION and put your files in that directory – the next section describes the directory structure.

Directory structure

Put your files in a directory with a name like mypackage-0.1, as described above. If you are porting another software package, then the directory should contain a subdirectory `src/`, containing an unaltered copy of the package. Every file not in `src/` should be under version control, i.e. checked into an hg repository.

More precisely, the directory should contain the following:

- `src/`: this directory contains vanilla upstream code, with a few exceptions, e.g. when the spkg shipped with Sage is in effect upstream, and development on that code base is happening in close coordination with Sage. See John Cremona's `eclib` spkg, for instance. The directory `src/` must not be under revision control.
- `.hg`, `.hgignore`, and `.hgtags`: The Sage project uses Mercurial for its revision control system (see *Producing Patches with Mercurial*). The hidden directory `.hg` is part of the standard Sage spkg layout. It contains the Mercurial repository for all files not in the `src/` directory.

The files `.hgignore` and `.hgtags` also belong to the Mercurial repository. The file `.hgtags` is optional, and is frequently omitted. You should make sure that the file `.hgignore` contains “`src/`”, since we are not tracking its content. Indeed, frequently this file contains only a single line,

```
src/
```

- `spkg-install`: this file contains the install script. See below for more information and a template.
- `SPKG.txt`: this file describes the spkg in wiki format. Each new revision needs an updated changelog entry or the spkg will get an automatic “needs work” at review time. See below for a template.
- `spkg-check`: this file runs the test suite. This is somewhat optional since not all spkg's have test suites. If possible, do create such a script since it helps isolate bugs in upstream packages
- `patches/`: this directory contains patches to source files in `src/`. Each file requiring changes (e.g. `foo.c`) must have a diff against the original file (e.g. `foo.c.patch`) for easy rebases against new upstream source releases. Patches to files in `src/` should be applied in `spkg-install`, and all patches must be documented in `SPKG.txt`, i.e. what they do, if they are platform specific, if they should be pushed upstream, etc. To ensure that all patched versions of upstream source files under `src/` are under revision control, the whole directory `patches/` must be under revision control.

Never apply patches to upstream source files under `src/` and then package up an spkg. Such a mixture of upstream source with Sage specific patched versions is a recipe for confusion. There must be a **clean separation** between the source provided by the upstream project and the patched versions that the Sage project generates based on top of the upstream source.

The file `spkg-install`

The script `spkg-install` is run during installation of the Sage package. In this script, you may make the following assumptions:

- The `PATH` has the locations of `sage` and `python` (from the Sage installation) at the front. Thus the command

```
python setup.py install
```

will run the correct version of Python with everything set up correctly. Also, running `gap` or `Singular`, for example, will run the correct version.
- The environment variable `SAGE_ROOT` points to the root directory of the Sage installation.
- The environment variable `SAGE_LOCAL` points to the `SAGE_ROOT/local` directory of the Sage installation.
- The environment variables `LD_LIBRARY_PATH` and `DYLD_LIBRARY_PATH` both have `SAGE_ROOT/local/lib` at the front.

The `spkg-install` script should copy your files to the appropriate place after doing any build that is necessary. Here is a template:

```
#!/usr/bin/env bash

if [[ -z "$SAGE_LOCAL" ]]; then
    echo "SAGE_LOCAL undefined ... exiting"
    echo "Maybe run 'sage --sh'?"
    exit 1
fi

cd src

# Apply patches. See SPKG.txt for information about what each patch
# does.
for patch in ../patches/*.patch; do
    patch -p1 <"$patch"
    if [ $? -ne 0 ]; then
        echo >&2 "Error applying '$patch'"
        exit 1
    fi
done

./configure --prefix="$SAGE_LOCAL"
if [ $? -ne 0 ]; then
    echo "Error configuring PACKAGE_NAME."
    exit 1
fi

$MAKE
if [ $? -ne 0 ]; then
    echo "Error building PACKAGE_NAME."
    exit 1
fi

$MAKE install
if [ $? -ne 0 ]; then
    echo "Error installing PACKAGE_NAME."
    exit 1
fi
```

```

if [ [ "$SAGE_SPKG_INSTALL_DOCS" = yes ] ] ; then
# Before trying to build the documentation, check if any
# needed programs are present. In the example below, we
# check for 'latex', but this will depend on the package.
# Some packages may need no extra tools installed, others
# may require some. We use 'command -v' for testing this,
# and not 'which' since 'which' is not portable, whereas
# 'command -v' is defined by POSIX.

# if [ `command -v latex` ] ; then
#   echo "Good, latex was found, so building the documentation"
# else
#   echo "Sorry, can't build the documentation for PACKAGE_NAME as latex is not installed"
#   exit 1
# fi

# make the documentation in a package-specific way
# for example, we might have
# cd doc
# $MAKE html

if [ $? -ne 0 ]; then
    echo "Error building PACKAGE_NAME docs."
    exit 1
fi
mkdir -p $SAGE_ROOT/local/share/doc/PACKAGE_NAME
# assuming the docs are in doc/*
cp -r doc/* $SAGE_ROOT/local/share/doc/PACKAGE_NAME/
fi

```

Note that the first line is `#!/usr/bin/env bash`; this is important for portability. Next, the script checks that `SAGE_LOCAL` is defined to make sure that the Sage environment has been set. After this, the script may simply run `cd src` and then call either `python setup.py install` or the autotools sequence `./configure && make && make install`, or something else along these lines.

Sometimes, though, it can be more complicated. For example, you might need to apply the patches from the `patches` directory in a particular order. Also, you should first build (e.g. with `python setup.py build`, exiting if there is an error), before installing (e.g. with `python setup.py install`). In this way, you would not overwrite a working older version with a non-working newer version of the spkg.

When copying documentation to `$SAGE_ROOT/local/share/doc/PACKAGE_NAME`, it may be necessary to check that only the actual documentation files intended for the user are copied. For example, if the documentation is built from `.tex` files, you may just need to copy the resulting pdf files, rather than copying the entire `doc` directory. When generating documentation using Sphinx, copying the `build/html` directory generally will copy just the actual output intended for the user.

The file `SPKG.txt`

The `SPKG.txt` file should follow this pattern:

```
= name of spkg =
```

```
== Description ==
```

```
Describe the package here.
```

```
== License ==
```

Describe the package's license here.

```
== SPKG Maintainers ==
```

Put a bulleted list of the maintainers of the SPKG here:

```
* Mary Smith
* Bill Jones
* Leonhard Euler
```

```
== Upstream Contact ==
```

Provide information for upstream contact.

```
== Dependencies ==
```

Put a bulleted list of dependencies here:

```
* python
* readline
```

```
== Special Update/Build Instructions ==
```

List patches that need to be applied and what they do

```
== Changelog ==
```

Provide a changelog of the spkg here, where the entries have this format:

```
=== mypackage-0.1.p0 (Mary Smith, 1 Jan 2012) ===
```

```
 * Patch src/configure so it builds on Solaris. See Sage trac #137.
```

```
=== mypackage-0.1 (Leonhard Euler, 17 September 1783) ===
```

```
 * Initial release. See Sage trac #007.
```

When the directory (say, `mypackage-0.1`) is ready, the command

```
sage --pkg mypackage-0.1
```

will create the file `mypackage-0.1.spkg`. As noted above, this creates a compressed tar file. Running `sage --pkg_nc mypackage-0.1` creates an uncompressed tar file.

When your spkg is ready, you should post about it on `sage-devel`. If people there think it is a good idea, then post a link to the spkg on the Sage trac server (see *The Sage Trac Server: Submitting Patches and Packages*) so it can be refereed. Do not post the spkg itself to the trac server: you only need to provide a link to your spkg. If your spkg gets a positive review, it might be included into the core Sage library, or it might become an optional download from the Sage website, so anybody can automatically install it by typing `sage -i mypackage-version.spkg`.

Note: For any spkg:

- Make sure that the hg repository contains every file outside the `src` directory, and that these are all up-to-date and committed into the repository.
 - Include an `spkg-check` file if possible (see [trac ticket #299](#)).
-

Note:

- If your package depends on another package, say `boehm_gc`, then you should check that this other package has been installed. Your `spkg-install` script should check that it exists, with code like the following:

```
BOEHM_GC=`cd $SAGE_ROOT/spkg/standard/; ./newest_version boehm_gc`
if [ $? -ne 0 ]; then
    echo "Failed to find boehm_gc. Please install the boehm_gc spkg"
    exit 1
fi
```

- If your package is intended to be a standard Sage spkg, then you should make sure that any dependencies for your package are recorded in the makefile `SAGE_ROOT/spkg/standard/deps`. Also add lines for your package to the script `SAGE_ROOT/spkg/install`. For example, the relevant lines for the `readline` package are

```
READLINE=`$newest readline`
export READLINE
```

- *Caveat:* Do not just copy to e.g. `SAGE_ROOT/local/lib/gap*/` since that will copy your package to the `lib` directory of the old version of GAP if GAP is upgraded.
- External Magma code goes in `SAGE_ROOT/data/extcode/magma/user`, so if you want to redistribute Magma code with Sage as a package that Magma-enabled users can use, that is where you would put it. You would also want to have relevant Python code to make the Magma code easily usable.

3.3.2 Avoiding troubles

This section contains some guidelines on what an spkg must never do to a Sage installation. You are encouraged to produce an spkg that is as self-contained as possible.

1. An spkg must not modify an existing source file in the Sage library.
2. Do not allow an spkg to modify another spkg. One spkg can depend on other spkg – see above. You need to first test for the existence of the prerequisite spkg before installing an spkg that depends on it.

3.4 Patching a Sage Package

This chapter provides guidelines on patching an existing spkg. Also covered are steps for upgrading an upstream project's source distribution, as contained under the subdirectory `src/`, to the latest upstream release. For information on creating a new spkg, see the chapter *Producing New Sage Packages*.

3.4.1 Overview of patching spkg's

Make sure you are familiar with the structure and conventions relating to spkg's; see the chapter *Producing New Sage Packages* for details. Patching an spkg involves patching the installation script of the spkg and/or patching the upstream source code contained in the spkg. Say you want to patch the Matplotlib package `matplotlib-1.0.1.p0`. Note that “p0” denotes the patch level of the spkg, while “1.0.1” refers to the upstream version of Matplotlib as contained under `matplotlib-1.0.1.p0/src/`. The installation script of that spkg is

```
matplotlib-1.0.1.p0/spkg-install
```

In general, a script with the name `spkg-install` is an installation script for an `spkg`. To patch the installation script, use a text editor to edit that script. Then in the log file `SPKG.txt`, provide a high-level description of your changes. Once you are satisfied with your changes in the installation script and the log file `SPKG.txt`, use Mercurial to check in your changes and make sure to provide a meaningful commit message. See the section *Submitting a change* for guidelines relating to commit messages.

The directory `src/` contains the source code provided by the upstream project. For example, the source code of Matplotlib 1.0.1 is contained under

```
matplotlib-1.0.1.p0/src/
```

To patch the upstream source code, you should edit a copy of the relevant file – files in the `src/` directory should be untouched, “vanilla” versions of the source code. For example, you might copy the entire `src/` directory:

```
$ pwd
matplotlib-1.0.1.p0
$ cp -pR src src-patched
```

Then edit files in `src-patched/`. Once you are satisfied with your changes, generate a unified diff between the original file and the edited one, and save it in `patches/`:

```
$ diff -u src/configure src-patched/configure > patches/configure.patch
```

Save the unified diff to a file with the same name as the source file you patched, but using the file extension “.patch”. Note that the directory `src/` should not be under revision control, whereas `patches/` must be under revision control. The Mercurial configuration file `.hginore` should contain the following line:

```
src/
```

Ensure that the installation script `spkg-install` contains code to apply the patches to the relevant files under `src/`. For example, the file

```
matplotlib-1.0.1.p0/patches/finance.py.patch
```

is a patch for the file

```
matplotlib-1.0.1.p0/src/lib/matplotlib/finance.py
```

The installation script `matplotlib-1.0.1.p0/spkg-install` contains the following code to install the relevant patches:

```
cd src

# Apply patches. See SPKG.txt for information about what each patch
# does.
for patch in ../patches/*.patch; do
    patch -p1 <"$patch"
    if [ $? -ne 0 ]; then
        echo >&2 "Error applying '$patch'"
        exit 1
    fi
done
```

Of course, this could be modified if the order in which the patches are applied is important, or if some patches were platform-dependent. For example:

```
if [ "$UNAME" = "Darwin" ]; then
    for patch in ../patches/darwin/*.patch; do
        patch -p1 <"$patch"
        if [ $? -ne 0 ]; then
```

```

        echo >&2 "Error applying '$patch'"
        exit 1
    fi
done
fi

```

(The environment variable `UNAME` is defined by the script `sage-env`, and is available when `spkg-install` is run.)

Now provide a high-level explanation of your changes in `SPKG.txt`. Note the format of `SPKG.txt` – see the chapter *Producing New Sage Packages* for details. Once you are satisfied with your changes, use Mercurial to check in your changes with a meaningful commit message. Then use the command `hg tag` to tag the tip with the new version number (using “p1” instead of “p0”: we have made changes, so we need to update the patch level):

```
$ hg tag matplotlib-1.0.1.p1
```

Next, rename the directory `matplotlib-1.0.1.p0` to `matplotlib-1.0.1.p1` to match the new patch level. To produce the actual `spkg` file, change to the parent directory of `matplotlib-1.0.1.p1` and execute

```
$ /path/to/sage-x.y.z/sage --pkg matplotlib-1.0.1.p1
Creating Sage package matplotlib-1.0.1.p1
```

```
Created package matplotlib-1.0.1.p1.spkg.
```

```

NAME: matplotlib
VERSION: 1.0.1.p1
SIZE: 11.8M
HG REPO: Good
SPKG.txt: Good

```

`Spkg` files are either bziped tar files or just plain tar files; the command `sage --pkg ...` produces the bziped version. If your `spkg` contains mostly binary files which will not compress well, you can use `sage --pkg_nc ...` to produce an uncompressed version, i.e., a plain tar file:

```
$ sage --pkg_nc matplotlib-1.0.1.p0/
Creating Sage package matplotlib-1.0.1.p0/ with no compression
```

```
Created package matplotlib-1.0.1.p0.spkg.
```

```

NAME: matplotlib
VERSION: 1.0.1.p0
SIZE: 32.8M
HG REPO: Good
SPKG.txt: Good

```

Note that this is almost three times the size of the compressed version, so we should use the compressed version!

At this point, you might want to submit your patched `spkg` for review. So provide a URL to your `spkg` on the relevant trac ticket and/or in an email to the relevant mailing list. Usually, you should not upload your `spkg` itself to the relevant trac ticket – don’t post large binary files to the trac server.

3.4.2 Use patch for patching

The main message of this section is: use the GNU program `patch` to apply patches to files in `src/`. GNU `patch` is distributed with Sage, so if you are writing an `spkg` which is not part of the standard Sage distribution, you may use `patch` in the `spkg-install` script freely. If you are working on an `spkg` which is (or will be) a standard `spkg` in Sage, then you should make sure that `patch` is listed as a dependency for your `spkg` in the makefile `SAGE_ROOT/spkg/standard/deps`.

See the section *Overview of patching spkg's* for information about how to produce patch files in the directory `patches/`, and how to apply them in `spkg-install`.

3.4.3 Bumping up an spkg's version

If you want to bump up the version of an spkg, you need to follow some naming conventions. Use the name and version number as given by the upstream project, e.g. `matplotlib-1.0.1`. If the upstream package is taken from some revision other than a stable version, you need to append the date at which the revision is made, e.g. the Singular package `singular-3-1-0-4-20090818.p3.spkg` is made with the revision as of 2009-08-18. If you start afresh from an upstream release without any patches to its source code, the resulting spkg need not have any patch-level labels (appending `".p0"` is allowed, but is optional). For example, `sagenb-0.6.spkg` is taken from the upstream stable version `sagenb-0.6` without any patches applied to its source code. So you do not see any patch-level numbering such as `.p0` or `.p1`.

Say you start with `matplotlib-1.0.1.p0` and you want to replace Matplotlib 1.0.1 with version 1.0.2. This entails replacing the source code for Matplotlib 1.0.1 under `matplotlib-1.0.1.p0/src/` with the new source code. To start with, follow the naming conventions as described in the section *Overview of patching spkg's*. If necessary, remove any obsolete patches and create any new ones, placing them in the `patches/` directory. Modify the script `spkg-install` to take any changes to the patches into account; you might also have to deal with changes to how the new version of the source code builds. Then package your replacement spkg using the Sage command line options `--pkg` or `--pkg_nc` (or `tar` and `bzip2`).

To install your replacement spkg, you use

```
sage -f http://URL/to/package-x.y.z.spkg
```

or

```
sage -f /path/to/package-x.y.z.spkg
```

To compile Sage from source with the replacement (standard) spkg, untar a Sage source tarball, remove the existing spkg under `SAGE_ROOT/spkg/standard/`. In its place, put your replacement spkg. Then execute `make` from `SAGE_ROOT`.

3.5 The Sage Trac Server: Submitting Patches and Packages

What should you do with your Mercurial patches for Sage? You should post them on the Sage trac server.

The Sage trac server, located at http://trac.sagemath.org/sage_trac/, is where Sage bugs are listed and patched, new code is posted and reviewed, and ideas for extending and improving Sage are discussed. Thus if you find a bug in Sage, or if you have new code to submit, or if you have corrections for the documentation, you should post on the trac server.

Items on the server are called "tickets", and anyone may browse the tickets: just visit http://trac.sagemath.org/sage_trac/report. You need to open an account, though, if you want to comment on a ticket, submit a patch, or create a new ticket. See the [trac server](#) for more information about obtaining an account. This chapter contains various guidelines on using the trac server.

3.5.1 Reporting bugs

"The first step is admitting you have a problem."

If you think you have found a bug in Sage, you should first search through the following Google groups for postings related to your possible bug:

- `sage-devel`: <http://groups.google.com/group/sage-devel>
- `sage-support`: <http://groups.google.com/group/sage-support>

Maybe the problem you have encountered has already been discussed. You should also search the trac server to see if anyone else has opened a ticket about your bug.

If you do not find anything, and you are not sure that you have found a bug, ask about it on `sage-devel`. You might be asked to open a new ticket on the trac server. As mentioned above, you need an account to do this. To report a bug, login and click on the “New ticket” button. Type a meaningful one-liner in the “Short summary” box, with more information in the larger box below. You should include at least one explicit, reproducible example illustrating your bug (and/or the steps required to reproduce the buggy behavior). You should also include the version of Sage (and any relevant packages) you are using, and operating system information, being precise as possible (32-bit, 64-bit, ...).

Between the “Summary” and “Full description” boxes, there is a place to choose the “Type” of the ticket: “Defect”, “Enhancement”, or “Task”. Use your best judgment here; a bug should probably be reported as a “Defect”.

Choose a priority for your bug, keeping in mind that the “blocker” label should be used very sparingly. Also pick a component for your bug; this is sometimes straightforward. If your bug deals with Sage’s calculus implementation, choose “calculus”. If it is not obvious, do your best. Choose a milestone; if you are not sure what to choose, just choose the numbered version of Sage from the menu (“sage-4.3.3”, for example). Type in some helpful keywords. In the box labeled “Assign to”, type “somebody” if you are not sure what else to do.

Hit the “Preview” button to make sure everything looks okay, and then hit “Submit ticket”.

If you do not have an account on the trac system to report directly, you are still encouraged to report any possible bug to the `sage-devel` mailing list at `sage-devel@googlegroups.com`. The list is moderated for new users and requires subscription. In your bug report to `sage-devel`, make sure to include the following information:

- **operating system**: as precise as possible and architecture (32-bit, 64-bit, ...)
- affected version: the exact **version number** and the downloaded package (source, precompiled, virtual machine image, or an upgrade from a previous version (which one?))
- provide a **reproducible example** and/or define the steps to reproduce the erroneous behaviour.

Thank you in advance for reporting bugs to improve Sage in the future!

3.5.2 Guidelines for opening tickets

In addition to bug reports, you should also open a ticket if you have some new code which extends Sage’s capabilities. If you have a feature request, start a discussion on `sage-devel` first, and then if there seems to be general agreement that you have a good idea, open a ticket describing the idea.

When you consider opening a new ticket, please bear the following points in mind.

- Before opening a ticket, make sure that nobody else has opened a ticket about the same or closely related issue.
- It is much better to open several specific tickets than one that is very broad. Indeed, a single ticket which deals with lots of different issues can be quite problematic, and should be avoided.
- Be precise: If foo does not work on OS X but is fine on Linux, mention that in the title. Use the keyword option so that searches will pick up the issue.
- The problem described in the ticket must be solvable. For example, it would be silly to open a ticket whose purpose was “Make Sage the best mathematical software in the world”. There is no metric to measure this properly and it is highly subjective.
- If appropriate, provide URLs to background information or email threads relevant to the problem you are reporting.

3.5.3 Patching bugs/working on tickets

If you have code which fixes a bug or deals with some issue in Sage, here is what to do. First, use Mercurial to create a patch file. See *Walking Through the Development Process* for more information on using Mercurial to produce/manage patches. If the issue has been reported as a ticket on the trac server, attach your patch file to that ticket: go to the ticket, click on the “Attach File” button, and follow the directions. On the ticket page, you should add a comment explaining your patch. Some relevant information include:

- The version of Sage you used to create the patch. If the patch is based on Sage x.y.z, ensure you include such information.
- If the ticket has more than one patch, explicitly specify which ones are to be used. Are all of the patches to be applied? Or only a subset of the patches on the ticket?
- If more than one patch is to be applied, state the order in which those patches are to be applied.
- Does the ticket depend on another ticket? Sometimes, a ticket requires that the patches on another ticket be applied first. Be sure to include such information if relevant.
- It is best to supply information about ticket dependencies and patch order in a way that the Patch Buildbot can understand. This bot automatically applies patches from trac and tests them. See its wiki: <http://wiki.sagemath.org/buildbot>

If there is no trac ticket associated to this issue, create one (as explained in the previous sections) describing the issue and your solution, and attach your patch.

The following are some other relevant issues:

- Every bug fixed should result in a doctest.
- Cooperative debugging via IRC is faster by at least an order of magnitude. If you have not learned how to use IRC, please do so. If you have problems using IRC because of firewalls, but you do have an account on the machine `sage.math`, you can use `irssi` via ssh there. If you have a flaky connection, you can use it together with the program screen.
- This is not an issue with defects, but there are many enhancements possible for Sage and too few developers to implement all the good ideas. The trac server is useful for keeping ideas in a central place because in the Google groups they tend to get lost once they drop off the first page.
- If you are a developer, be nice and try to solve a stale/old ticket every once in a while.
- Some people regularly do triage. Triage in this context means that we look at new bugs and classify them according to our perceived priority. It is very likely that different people will see priorities of bugs very differently from us, so please let us know if you see a problem with specific tickets.
- **Patches Preferred:** Patches are easier to review, edit and can be merged without affecting the history. So we greatly prefer patches over Mercurial bundles. If you do have a large number of patches, a bundle can still be better than patches. One alternative to bundles is to use Mercurial queues to flatten the history. That might or might not be desirable. See *Walking Through the Development Process* for further information on using Mercurial queues to produce/manage patches.

3.5.4 Reviewing patches

All code that goes into Sage is peer-reviewed, to ensure that the conventions discussed in this manual are followed, to make sure that there are sufficient examples and doctests in the documentation, and to try to make sure that the code does, mathematically, what it is supposed to.

If someone (other than you) has posted a patch for a ticket on the trac server, you can review it! Look at the patch (by clicking on the file name in the list of attachments) to see if it makes sense. Download it (from the window displaying the patch, see the “Download” option at the bottom of the page). Apply it (using `hg_sage.patch('filename')`),

for example) to your copy of Sage, and build Sage with the new code by typing `sage -b`. See the walkthrough section *Reviewing a patch* for more details on downloading and applying patches.

Now ask yourself questions such as the following:

- Does the new source code make sense?
- When you run it in Sage, does it fix the problem reported on the ticket?
- Does it introduce any new problems?
- Is it documented sufficiently, including both explanation and doctests? This is **very** important: all code in Sage must have doctests, so even if the patch is for code which did not have a doctest before, the new version must include one. In particular, all new code must be **100% doctested**. Use the command `sage -coverage <files>` to see the coverage percentage of `<files>`.
- In particular, is there a doctest illustrating that the bug has been fixed? If a function used to give the wrong answer and this patch fixes that, then it should include a doctest illustrating its new success. That doctest should be marked with the ticket number as an in-line comment.
- Is the ticket number noted in the comment line near the top of the patch? Is the patch author noted in all the files which were edited?
- If the patch claims to speed up some computation, does the ticket contain code examples to illustrate the claim? The ticket should explain the speed efficiency before applying the patch. It should also explain the speed efficiency gained after applying the patch. In both the “before” and “after” explanation, there should be code samples to illustrate the claims. It is not sufficient to just mention that the patch results in a speed-up of up to x percent or y factor.
- Does the reference manual build without errors? You can test the reference manual using the command `sage -docbuild reference html` to build the HTML version. The PDF version of the reference manual must also build without errors. Use the command `sage -docbuild reference pdf` to test it out. The latter command requires that you have LaTeX installed on your system.
- Do all doctests pass without errors? This too is **very** important. It is extremely difficult to predict which components of Sage will be affected by a given patch (especially if you don't have working knowledge of the **entire** Sage library), so you should run tests on the whole library—including those flagged as `#long`—before giving a positive review. (For that matter, the patch writer should run these tests before uploading the patch.) You can test the Sage library with `make testlong` or `make ptestlong` (edit the number of threads in `$(SAGE_ROOT)/Makefile` before using `ptestlong`). See *Doctesting the Sage Library* for more information.
- Do the code and documentation follow conventions documented in the following sections?
 - *Conventions for Coding in Sage*
 - *Coding in Python for Sage*
 - *Coding in Cython*

If the answers to these and other such reasonable questions are yes, then you might want to give the patch a positive review. On the main ticket page, write a comment in the box explaining your review. If you don't feel experienced enough for this, make a comment explaining what you checked, and end by asking if someone more experienced will take a look. If you think there are issues with the patch, explain them in the comment box and change the status to “needs work”. Browse the tickets on the trac server to see how things are done.

3.5.5 Closing tickets

Closing tickets is not possible unless you have “TICKET_ADMIN” rights in Trac. This is because only the current Sage release manager should ever close tickets. If you feel strongly that a ticket should be closed or deleted, then change the status of the ticket to `needs review` and change the milestone to `sage-duplicate/invalid/wontfix`. You should also comment on the ticket, explaining why it should

be closed. A related issue is re-opening tickets. You should refrain from re-opening a ticket that is already closed. Instead ask the release manager what to do.

3.5.6 Reasons to invalidate tickets

One Issue Per Ticket: A ticket must cover only one issue and should not be a laundry list of unrelated issues. If a ticket covers more than one issue, we cannot close it and while some of the patches have been applied to a given release, the ticket would remain in limbo.

No Patch Bombs: Code that goes into Sage is peer-reviewed. If you show up with an 80,000 lines of code bundle that completely rips out a subsystem and replaces it with something else, you can imagine that the review process will be a little tedious. These huge patch bombs are problematic for several reasons and we prefer small, gradual changes that are easy to review and apply. This is not always possible (e.g. coercion rewrite), but it is still highly recommended that you avoid this style of development unless there is no way around it.

Sage Specific: Sage's philosophy is that we ship everything (or close to it) in one source tarball to make debugging possible. You can imagine the combinatorial explosion we would have to deal with if you replaced only ten components of Sage with external packages. Once you start replacing some of the more essential components of Sage that are commonly packaged (e.g. Pari, GAP, lisp, gmp), it is no longer a problem that belongs in our tracker. If your distribution's Pari package is buggy for example, file a bug report with them. We are usually willing and able to solve the problem, but there are no guarantees that we will help you out. Looking at the open number of tickets that are Sage specific, you hopefully will understand why.

No Support Discussions: The trac installation is not meant to be a system to track down problems when using Sage. Tickets should be clearly a bug and not "I tried to do X and I couldn't get it to work. How do I do this?" That is usually not a bug in Sage and it is likely that `sage-support` can answer that question for you. If it turns out that you did hit a bug, somebody will open a concise and to-the-point ticket.

Solution Must Be Achievable: Tickets must be achievable. Many times, tickets that fall into this category usually ran afoul to some of the other rules listed above. An example would be to "Make Sage the best CAS in the world". There is no metric to measure this properly and it is highly subjective.

3.5.7 Milestones vs. releases

Milestones are usually goals to be met while working toward a release. In Sage's trac, we use milestones instead of releases, but unless somebody volunteers to clean up all the old milestones, we will stick with the current model. It does not make a whole lot of difference if we use milestone instead of release.

Finely grained releases are good. Release early and often is the way to go, especially as more and more patches are coming in.

It is a good idea to make a big release and schedule at least one more bug fix release after that to sort out the inevitable "doctest X is broken on distribution Y and compiler Z" problem. Given the number of compilers and operating systems out there, one has to be realistic to expect problems. A compile farm would certainly help to catch issues early.

3.5.8 Assigning tickets

- Each ticket must have a milestone assigned. If you are unsure, assign it to the current milestone.
- If a ticket has a patch or spkg that is ready to be reviewed, assign it against the current milestone.
- Defect vs. enhancement vs. task: this can be tricky, but a defect should be something that leads to an exception or a mathematically wrong result.
- If you are unsure to whom to assign the ticket, assign it to "somebody" or "tba", which stands for "to be assigned".

- Certain categories have default people who get assigned all issues. For example, Jane Smith might be the default person who gets assigned all tickets relating to calculus. This means that Jane looks after tickets in that category, but not necessarily the person who is to fix all open tickets relating to calculus.
- If you have been assigned a ticket, you should either accept it or assign it back to “somebody” or “tba”. Many people do not accept pending tickets at the moment. You have accepted a ticket if your name has a star next to it.

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

INDEX

E

environment variable

MAKE, 58

UNAME, 75

M

MAKE, 58

U

UNAME, 75