Sage Developer's Guide

Release 6.9

The Sage Development Team

October 13, 2015

Everybody who uses Sage is encouraged to contribute something back to Sage at some point. You could:

- Add examples to the documentation
- Find bugs or typos
- Fix a bug
- Implement a new function
- Contribute a useful tutorial for a mathematical topic
- Translate an existing document to a new language
- Create a new class, create a fast new C library, etc.

This document tells you what you need to know to do all the above, from reporting bugs to modifying and extending Sage and its documentation. We also discuss how to share your new and modified code with other Sage users around the globe.

Here are brief overviews of each part; for more details, see the extended table of contents below. No matter where you start, good luck and welcome to Sage development!

- **Trac server:** all changes go through the the Sage Trac server at some point. It contains bug reports, upgrade requests, changes in progress, and those already part of Sage today. *Click here* for more information.

  Importantly, you will need to *create a trac account* in order to contribute.

- **Source code:** You need your own copy of Sage's source code to change it. Go there to get it and for instructions to build it.

  If you have never worked on software before, pay close attention to the prerequisites to compile on your system.

- **Conventions:** read our *conventions and guidelines* for code and documentation.

  For everything related to manuals, tutorials, and languages, *click here*.

- **Git (revision control):** To share changes with the Sage community, you will need to learn about revision control; we use the software Git for this purpose.
  - *Here is* an overview of our development flow.
  - *Unfamiliar with Git or revision control?*
  - *How to install it?*
  - *How to configure it for use with Trac?*

CHAPTER

ONE

GIT FOR SAGE DEVELOPMENT

1.1 First Steps with Git

Sage uses git for version control.

1.1.1 Setting Up Git

To work on the Sage source code, you need

- a working git installation, see *Installing Git*. Sage actually comes with git, see below. However, it is recommended that you have a system-wide install if only to save you some typing.
- configure git to use your name and email address for commits, see *Your Name and Email*. The Sage development scripts will prompt you if you don't. But, especially if you use git for other projects in the future as well, you really should configure git.

The *Tips and References* chapter contains further information about git that might be useful to some but are not required.

Installing Git

First, try `git` on the command line. Most distributions will have it installed by default if other development tools are installed. If that fails, use the following to install git:

Debian / Ubuntu `sudo apt-get install git-core`

Fedora `sudo yum install git-core`

Windows Download and install msysGit

OS X Use the git OSX installer. If you have an older Mac, be sure to get the correct version. (Alternately you may get it from the Command Line Tools or even simply by attempting to use `git` and then following instructions.)

Finally, Sage includes git. Obviously there is a chicken-and-egg problem to checkout the Sage source code from its git repository, but one can always download a Sage source tarball or binary distribution. You can then run git via the `sage -git` command line switch. So, for example, `git help` becomes `sage -git help` and so on. Note that the examples in the developer guide will assume that you have a system-wide git installation.

Some further resources for installation help are:

- Chapter 2 of the git book
- The git homepage for the most recent information.
- Github install help pages

Your Name and Email

The commit message of any change contains your name and email address to acknowledge your contribution and to have a point of contact if there are questions in the future; Filling it in is required if you want to share your changes. The simplest way to do this is from the command line:

```
[user@localhost ~] git config --global user.name "Your Name"
[user@localhost ~] git config --global user.email you@yourdomain.example.com
```

This will write the settings into your *git configuration file* with your name and email:

```
[user]
    name = Your Name
    email = you@yourdomain.example.com
```

Of course you'll need to replace `Your Name` and `you@yourdomain.example.com` with your actual name and email address.

1.1.2 Sage Development Process

This section is a concise overview of the Sage development process. In it, we will see how to make changes to the Sage source code and record them in the git revision control system.

In the following section on *Collaborative Development with Git-Trac* we will look at communicating these changes back to the Sage project. We also have a handy one-page "cheat sheet" of commonly used git commands that you can print out and leave on your desk. We have some *recommended references and tutorials* as well.

You can alternatively fork and create a pull request at github which will automatically fetch your code and open a ticket on our trac server.

Configuring Git

One way or another, `git` is what Sage uses for tracking changes. So first, open a shell (for instance, Terminal on Mac) and check that `git` works:

```
[user@localhost]$ git
usage: git [--version] [--help] [-C <path>] [-c name=value]
...
The most commonly used git commands are:
   add        Add file contents to the index
...
   tag        Create, list, delete or verify a tag object signed with GPG

'git help -a' and 'git help -g' lists available subcommands and some
concept guides. See 'git help <command>' or 'git help <concept>'
to read about a specific subcommand or concept.
```

Don't worry about the giant list of subcommands. You really only need a handful for effective development, and we will walk you through them in this guide. If you got a "command not found" error, then you don't have git installed. Now is the time to install it; see *Setting Up Git* for instructions.

Because we also track who does changes in Sage with git, you must tell git how you want to be known. This only needs to be done once:

```
[user@localhost]$ git config --global user.name "Your Name"
[user@localhost]$ git config --global user.email you@yourdomain.example.com
```

If you have multiple accounts / computers use the same name on each of them. This name/email combination ends up in commits, so do it now before you forget!

Obtaining the Sage Source Code

Obviously one needs the Sage source code to develop. You can use your local installation of Sage, or (to start without Sage) download it from github which is a public read-only mirror (=faster) of our internal git repository:

```
[user@localhost]$ git clone git://github.com/sagemath/sage.git
Cloning into 'sage'...
[...]
Checking connectivity... done.
```

This creates a directory named `sage` containing the sources for the current stable and development releases of Sage. You will need to compile Sage in order to use it (if you cloned, you will need to remain on the internet for it to download various packages of Sage).

(For the experts, note that the repository at git.sagemath.org is where development actually takes place .)

Branching Out

In order to start modifying Sage, we want to make a *branch* of Sage. A branch is a copy (except that it doesn't take up twice the space) of the Sage source code where you can store your modifications to the Sage source code and which you can upload to trac tickets.

It is easy to create a new branch, just check out (switch to) the branch from where you want to start (that is, `master`) and use the `git branch` command:

```
[user@localhost sage]$ git checkout master
[user@localhost sage]$ git branch last_twin_prime
[user@localhost sage]$ git checkout last_twin_prime
```

You can list all branches using:

```
[user@localhost]$ git branch
  master
* last_twin_prime
```

The asterisk shows you which branch you are on. Without an argument, the `git branch` command just displays a list of all local branches with the current one marked by an asterisk. Also note that `git branch` creates a new branch, but does not switch to it. For this, you have to use `git checkout`:

```
[user@localhost sage]$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'github/master'.
[user@localhost sage]$ git branch
* master
  last_twin_prime
[user@localhost sage]$ git checkout last_twin_prime
Switched to branch 'last_twin_prime'
```

Note that, unless you explicitly upload ("push") a branch to remote git repository, the local branch will only be on your computer and not visible to anyone else.

To avoid typing the new branch name twice you can use the shortcut `git checkout -b my_new_branch` to create and switch to the new branch in one command.

The History

It is always a good idea to check that you are making your edits on the version that you think you are on. The first one shows you the topmost commit in detail, including its changes to the sources:

```
[user@localhost sage]$ git show
```

To dig deeper, you can inspect the log:

```
[user@localhost sage]$ git log
```

By default, this lists all commits in reverse chronological order.

- If you find your branch to be in the wrong place, see the *Reset and Recovery* section.
- Many programs are available to help you visualize the history tree better. `tig` is a very nice text-mode such tool.

Editing the Source Code

Once you have your own branch, feel free to make any changes as you like. *Subsequent chapters* of this developer guide explain how your code should look like to fit into Sage, and how we ensure high code quality throughout.

Status is probably the most important git command. It tells you which files changed, and how to continue with recording the changes:

```
[user@localhost sage]$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   some_file.py
    modified:   src/sage/primes/all.py

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    src/sage/primes/last_pair.py

no changes added to commit (use "git add" and/or "git commit -a")
```

To dig deeper into what was changed in the files you can use:

```
[user@localhost sage]$ git diff some_file.py
```

to show you the differences.

Rebuilding Sage

Once you have made any changes you of course want to build Sage and try out your edits. As long as you only modified the Sage library (that is, Python and Cython files under `src/sage/...`) you just have to run:

```
[user@localhost sage]$ ./sage -br
```

to rebuild the Sage library and then start Sage. This should be quite fast. If you made changes to third-party packages then you have to run:

```
[user@localhost sage]$ make
```

as if you were installing Sage from scratch. However, simply running `make` will only recompile packages that were changed, so it shoud be much faster than compiling Sage the first time. Rarely there are conflicts with other packages, or with the already-installed older version of the package that you changed, in that case you do have to recompile everything using:

```
[user@localhost sage]$ make distclean && make
```

Also, don't forget to run the tests (see *Running Sage's doctests*) and build the documentation (see *The Sage Manuals*).

Commits (Snapshots)

Whenever you have reached your goal, a milestone towards it, or just feel like you got some work done you should *commit* your changes. A commit is just a snapshot of the state of all files in the *repository* (the program you are working on).

Unlike with some other revision control programs, in git you first need to *stage* the changed files, which tells git which files you want to be part of the next commit:

```
[user@localhost sage]$ git status
# On branch my_branch
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       src/sage/primes/last_pair.py
nothing added to commit but untracked files present (use "git add" to track)

[user@localhost sage]$ git add src/sage/primes/last_pair.py
[user@localhost sage]$ git status
# On branch my_branch
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   src/sage/primes/last_pair.py
#
```

Once you are satisfied with the list of staged files, you create a new snapshot with the `git commit` command:

```
[user@localhost sage]$ git commit
... editor opens ...
[my_branch 31331f7] Added the very important foobar text file
 1 file changed, 1 insertion(+)
  create mode 100644 foobar.txt
```

This will open an editor for you to write your commit message. The commit message should generally have a one-line description, followed by an empty line, followed by further explanatory text:

```
Added the last twin prime

This is an example commit message. You see there is a one-line
summary followed by more detailed description, if necessary.
```

You can then continue working towards your next milestone, make another commit, repeat until finished. As long as you do not `git checkout` another branch, all commits that you make will be part of the branch that you created.

1.2 The git-trac command

Putting your local changes on a Trac ticket.

1.2.1 Collaborative Development with Git-Trac

Sometimes you will only want to work on local changes to Sage, for your own private needs. However, typically it is beneficial to share code and ideas with others; the manner in which the Sage project does this (as well as fixing bugs and upgrading components) is in a very collaborative and public setting on the Sage Trac server (the Sage bug and enhancement tracker).

One can use `git` *the hard way* for this, but this section explains how to use the helper `git trac` command, which simplifies many of the most common actions in collaboration on Sage. Some of the *tutorials* we suggest may be helpful in navigating what they are for.

Most of the commands in the following section will not work unless you have an account on Trac. If you want to contribute to Sage, it is a good idea to get an account now (see *Obtaining an Account*).

Installing the Git-Trac Command

Git is a separate project from trac, and the two do not know how to talk to each other. To simplify the development, we have a special `git trac` subcommand for the git suite. Note that this really is only to simplify interaction with our trac issue management, you can perform every development task with just git and a web browser. See *Git the Hard Way* instead if you prefer to do everything by hand:

```
[user@localhost]$ git clone https://github.com/sagemath/git-trac-command.git
Cloning into 'git-trac-command'...
[...]
Checking connectivity... done.
[user@localhost]$ source git-trac-command/enable.sh
Prepending the git-trac command to your search PATH
```

This creates a directory `git-trac-command`.

Sourcing the `enable.sh` script in there is just a quick and dirty way to enable it temporarily. For a more permanent installation on your system later, make sure to put the `git-trac` command in your `PATH`. Assuming that `~/bin` is already in your `PATH`, you can do this by symlinking:

```
[user@localhost]$ echo $PATH
/home/user/bin:/usr/local/bin:/usr/bin:/bin:/usr/local/sbin:/usr/sbin
[user@localhost]$ cd git-trac-command
[user@localhost git-trac-command]$ ln -s `pwd`/git-trac ~/bin/
```

See the git-trac README for more details.

Git and Trac Configuration

Note:

- trac uses username/password for authentication.

- Our git repository server uses SSH public key authentication for write access.

You need to set up both authentication mechanisms to be able to upload your changes with "git trac". For read-only access neither authentication mechanism is needed. To set up `git trac`, first go to the Sage directory and tell `git trac` about your trac account:

```
[user@localhost sage]$ git trac config --user USERNAME --pass 'PASSWORD'
Trac xmlrpc URL:
    http://trac.sagemath.org/xmlrpc (anonymous)
    http://trac.sagemath.org/login/xmlrpc (authenticated)
    realm sage.math.washington.edu
Username: USERNAME
Password: PASSWORD
Retrieving SSH keys...
    1024 ab:1b:7c:c9:9b:48:fe:dd:59:56:1e:9d:a4:a6:51:9d  My SSH Key
```

where you have to replace USERNAME with your trac user name and PASSWORD with your trac password. If you don't have a trac account, use `git trac config` without any arguments. The single quotes in `'PASSWORD'` escape special characters that you might have in your password. The password is stored in plain-text in `.git/config`, so make sure that it is not readable by other users on your system. For example, by running `chmod 0600 .git/config` if your home directory is not already private.

If there is no SSH key listed then you haven't uploaded your SSH public key to the trac server. You should do that now following the instructions to *Manually Linking your Public Key to your Trac Account*, if you want to upload any changes.

Note: The `git trac config` command will automatically add a `trac` remote git repository to your list of remotes if necessary.

If you followed the above instructions then you will have two remote repositories set up:

```
[user@localhost sage]$ git remote -v
origin      git://github.com/sagemath/sage.git (fetch)
origin      git://github.com/sagemath/sage.git (push)
trac        git://trac.sagemath.org/sage.git (fetch)
trac        git@trac.sagemath.org:sage.git (push)
```

The `git@...` part of the push url means that write access is secured with SSH keys, which you must have set up as in *Manually Linking your Public Key to your Trac Account*. Read-only access happens through the fetch url and does not require SSH.

Finally, if you do not want to use the `git trac` subcommand at all then you can set up the remote by hand as described in the section on *The Trac Server*.

Trac Tickets and Git Branches

Now let's start adding code to Sage!

Create a Ticket

Suppose you have written an algorithm for calculating the last twin prime, and want to add it to Sage. You would first open a ticket for that:

```
[user@localhost sage]$ git trac create 'Last Twin Prime'
Remote branch: u/user/last_twin_prime
Newly-created ticket number: 12345
Ticket URL: http://trac.sagemath.org/12345
Local branch: t/12345/last_twin_prime
```

This will create a new trac ticket titled "Last Twin Prime" with a *remote branch* `u/user/last_twin_prime` attached to it. The remote branch name is automatically derived from the ticket title; If you don't like this then you can use the `-b` switch to specify it explicitly. See `git trac create -h` for details. This new branch is automatically checked out for you with the *local branch* name `t/12345/last_twin_prime`.

Note: Only some trac fields are filled in automatically. See *The Ticket Fields* for what trac fields are available and how we use them.

Check out an Existing Ticket

Alternatively, you can use the web interface to the Sage trac development server to open a new ticket. Just log in and click on "Create Ticket".

Or maybe somebody else already opened a ticket. Then, to get a suitable local branch to make your edits, you would just run:

```
[user@localhost sage]$ git trac checkout 12345
Loading ticket #12345...
Checking out Trac #13744 remote branch u/user/last_twin_prime -> local branch t/12345/last_twin_prime
```

The `git trac checkout` command downloads an existing branch (as specified in the "Branch:" field on the trac ticket) or creates a new one if there is none yet. Just like the create command, you can specify the remote branch name explicitly using the `-b` switch if you want.

Note on Branch Names

The "Branch:" field of a trac ticket (see *The Ticket Fields*) indicates the git branch containing its code. Our git server implements the following access restrictions for **remote branch names**:

- You can read/write/create a branch named `u/your_username/whatever_you_like`. Everybody else can read.
- Everybody can read/write/create a branch named `public/whatever_you_like`.

Depending on your style of collaboration, you can use one or the other. The `git trac` subcommands defaults to the former.

As a convention, the `git trac` subcommand uses **local branch names** of the form `t/12345/description`, where the number is the trac ticket number. The script uses this number to figure out the ticket from the local branch name. You can rename the local branches if you want, but if they don't contain the ticket number then you will have to specify the ticket number manually when you are uploading your changes.

Making Changes

Once you have checked out a ticket, edit the appropriate files and commit your changes to the branch as described in *Editing the Source Code* and *Commits (Snapshots)*.

Uploading Changes to Trac

Automatic Push

At some point, you may wish to share your changes with the rest of us: maybe it is ready for review, or maybe you are collaborating with someone and want to share your changes "up until now". This is simply done by:

```
[user@localhost sage]$ git trac push
Pushing to Trac #12345...
Guessed remote branch: u/user/last_twin_prime

To git@trac.sagemath.org:sage.git
 * [new branch]      HEAD -> u/user/last_twin_prime

Changing the trac "Branch:" field...
```

This uploads your changes to a remote branch on the Sage git server. The `git trac` command uses the following logic to find out the remote branch name:

- By default, the remote branch name will be whatever is already on the trac ticket.
- If there is no remote branch yet, the branch will be called `u/user/description` (`u/user/last_twin_prime` in the example).
- You can use the `--branch` option to specify the remote branch name explicitly, but it needs to follow the naming convention from *Note on Branch Names* for you to have write permission.

Specifying the Ticket Number

You can upload any local branch to an existing ticket, whether or not you created the local branch with `git trac`. This works exactly like in the case where you started with a ticket, except that you have to specify the ticket number (since there is no way to tell which ticket you have in mind). That is:

```
[user@localhost sage]$ git trac push TICKETNUM
```

where you have to replace `TICKETNUM` with the number of the trac ticket.

Finishing It Up

It is common to go through a few iterations of commits before you upload, and you will probably also have pushed your changes a few times before your changes are ready for review.

Once you are happy with the changes you uploaded, they must be reviewed by somebody else before they can be included in the next version of Sage. To mark your ticket as ready for review, you should set it to `needs_review` on the trac server. Also, add yourself as the (or one of the) author(s) for that ticket by inserting the following as the first line:

```
Authors: Your Real Name
```

Downloading Changes from Trac

If somebody else worked on a ticket, or if you just switched computers, you'll want to get the latest version of the branch from a ticket into your local branch. This is done with:

```
[user@localhost sage]$ git trac pull
```

Technically, this does a *merge* (just like the standard `git pull`) command. See *Merging and Rebasing* for more background information.

Merging

As soon as you are working on a bigger project that spans multiple tickets you will want to base your work on branches that have not been merged into Sage yet. This is natural in collaborative development, and in fact you are very much encouraged to split your work into logically different parts. Ideally, each part that is useful on its own and can be reviewed independently should be a different ticket instead of a huge patch bomb.

For this purpose, you can incorporate branches from other tickets (or just other local branches) into your current branch. This is called merging, and all it does is include commits from other branches into your current branch. In particular, this is done when a new Sage release is made: the finished tickets are merged with the Sage master and the result is the next Sage version. Git is smart enough to not merge commits twice. In particular, it is possible to merge two branches, one of which had already merged the other branch. The syntax for merging is easy:

```
[user@localhost sage]$ git merge other_branch
```

This creates a new "merge" commit, joining your current branch and `other_branch`.

Warning: You should avoid merging branches both ways. Once A merged B and B merged A, there is no way to distinguish commits that were originally made in A or B. Effectively, merging both ways combines the branches and makes individual review impossible.
In practice, you should only merge when one of the following holds:

- Either two tickets conflict, then you have to merge one into the other in order to resolve the merge conflict.
- Or you definitely need a feature that has been developed as part of another branch.

A special case of merging is merging in the `master` branch. This brings your local branch up to date with the newest Sage version. The above warning against unnecessary merges still applies, though. Try to do all of your development with the Sage version that you originally started with. The only reason for merging in the master branch is if you need a new feature or if your branch conflicts.

Collaboration and conflict resolution

Exchanging Branches

It is very easy to collaborate by just going through the above steps any number of times. For example, Alice starts a ticket and adds some initial code:

```
[alice@laptop sage]$ git trac create "A and B Ticket"
... EDIT EDIT ...
[alice@laptop sage]$ git add .
[alice@laptop sage]$ git commit
[alice@laptop sage]$ git trac push
```

The trac ticket now has "Branch:" set to `u/alice/a_and_b_ticket`. Bob downloads the branch and works some more on it:

```
[bob@home sage]$ git trac checkout TICKET_NUMBER
... EDIT EDIT ...
[bob@home sage]$ git add .
```

```
[bob@home sage]$ git commit
[bob@home sage]$ git trac push
```

The trac ticket now has "Branch:" set to `u/bob/a_and_b_ticket`, since Bob cannot write to `u/alice/...`. Now the two authors just pull/push in their collaboration:

```
[alice@laptop sage]$ git trac pull
... EDIT EDIT ...
[alice@laptop sage]$ git add .
[alice@laptop sage]$ git commit
[alice@laptop sage]$ git trac push

[bob@home sage]$ git trac pull
... EDIT EDIT ...
[bob@home sage]$ git add .
[bob@home sage]$ git commit
[bob@home sage]$ git trac push
```

Alice and Bob need not alternate, they can also add further commits on top of their own remote branch. As long as their changes do not conflict (edit the same lines simultaneously), this is fine.

Conflict Resolution

Merge conflicts happen if there are overlapping edits, and they are an unavoidable consequence of distributed development. Fortunately, resolving them is common and easy with git. As a hypothetical example, consider the following code snippet:

```
def fibonacci(i):
    """
    Return the 'i'-th Fibonacci number
    """
    return fibonacci(i-1) * fibonacci(i-2)
```

This is clearly wrong; Two developers, namely Alice and Bob, decide to fix it. First, in a cabin in the woods far away from any internet connection, Alice corrects the seed values:

```
def fibonacci(i):
    """
    Return the 'i'-th Fibonacci number
    """
    if i > 1:
        return fibonacci(i-1) * fibonacci(i-2)
    return [0, 1][i]
```

and turns those changes into a new commit:

```
[alice@laptop sage]$ git add fibonacci.py
[alice@laptop sage]$ git commit -m 'return correct seed values'
```

However, not having an internet connection, she cannot immediately send her changes to the trac server. Meanwhile, Bob changes the multiplication to an addition since that is the correct recursion formula:

```
def fibonacci(i):
    """
    Return the 'i'-th Fibonacci number
```

```
    """
    return fibonacci(i-1) + fibonacci(i-2)
```

and immediately uploads his change:

```
[bob@home sage]$ git add fibonacci.py
[bob@home sage]$ git commit -m 'corrected recursion formula, must be + instead of *'
[bob@home sage]$ git trac push
```

Eventually, Alice returns to civilization. In her mailbox, she finds a trac notification email that Bob has uploaded further changes to their joint project. Hence, she starts out by getting his changes into her own local branch:

```
[alice@laptop sage]$ git trac pull
...
CONFLICT (content): Merge conflict in fibonacci.py
Automatic merge failed; fix conflicts and then commit the result.
```

The file now looks like this:

```
def fibonacci(i):
    """
    Return the `i`-th Fibonacci number
    """
<<<<<<< HEAD
    if i > 1:
        return fibonacci(i-1) * fibonacci(i-2)
    return i
=======
    return fibonacci(i-1) + fibonacci(i-2)
>>>>>>> 41675dfaedbfb89dcff0a47e520be4aa2b6c5d1b
```

The conflict is shown between the conflict markers `<<<<<<<` and `>>>>>>>`. The first half (up to the `=======` marker) is Alice's current version, the second half is Bob's version. The 40-digit hex number after the second conflict marker is the SHA1 hash of the most recent common parent of both.

It is now Alice's job to resolve the conflict by reconciling their changes, for example by editing the file. Her result is:

```
def fibonacci(i):
    """
    Return the `i`-th Fibonacci number
    """
    if i > 1:
        return fibonacci(i-1) + fibonacci(i-2)
    return [0, 1][i]
```

And then upload both her original change *and* her merge commit to trac:

```
[alice@laptop sage]$ git add fibonacci.py
[alice@laptop sage]$ git commit -m "merged Bob's changes with mine"
```

The resulting commit graph now has a loop:

```
[alice@laptop sage]$ git log --graph --oneline
*   6316447 merged Bob's changes with mine
|\
| * 41675df corrected recursion formula, must be + instead of *
* | 14ae1d3 return correct seed values
```

```
|/
* 14afe53 initial commit
```

If Bob decides to do further work on the ticket then he will have to pull Alice's changes. However, this time there is no conflict on his end: git downloads both Alice's conflicting commit and her resolution.

Reviewing

This section gives an example how to review using the `sage` command. For an explanation of what should be checked by the reviewer, see *The reviewer's check list*.

If you go to the web interface to the Sage trac development server then you can click on the "Branch:" field and see the code that is added by combining all commits of the ticket. This is what needs to be reviewed.

The `git trac` command gives you two commands that might be handy (replace `12345` with the actual ticket number) if you do not want to use the web interface:

- `git trac print 12345` displays the trac ticket directly in your terminal.
- `git trac review 12345` downloads the branch from the ticket and shows you what is being added, analogous to clicking on the "Branch:" field.

1.3 Git Tricks & Tips

When `git trac` is not enough.

1.3.1 Git the Hard Way

If you have no git experience, we recommend you to read the *Collaborative Development with Git-Trac* chapter instead. The `git-trac` simplifies the interaction with our git and trac servers.

If you want to contribute using git only, you are at the right place. This chapter will tell you how to do so, assuming some basic familiarity with git. In particular, you should have read *Sage Development Process* first.

We assume that you have a copy of the Sage git repository, for example by running:

```
[user@localhost ~]$ git clone git://github.com/sagemath/sage.git
[user@localhost ~]$ cd sage
[user@localhost sage]$ make
```

The Trac Server

The Sage trac server also holds a copy of the Sage repository, it is served via the ssh and git protocols. To add it as a remote repository to your local git repository, use these commands:

```
[user@localhost sage]$ git remote add trac git://trac.sagemath.org/sage.git -t master
[user@localhost sage]$ git remote set-url --push trac git@trac.sagemath.org:sage.git
[user@localhost sage]$ git remote -v
origin      git://github.com/sagemath/sage.git (fetch)
origin      git://github.com/sagemath/sage.git (push)
trac        git://trac.sagemath.org/sage.git (fetch)
trac        git@trac.sagemath.org:sage.git (push)
```

Instead of `trac` you can use any local name you want, of course. It is perfectly fine to have multiple remote repositories for git, think of them as bookmarks. You can then use `git pull` to get changes and `git push` to upload your local changes using:

```
[user@localhost sage]$ git <push|pull> trac [ARGS]
```

Note: In the command above we set up the remote to only track the `master` branch on the trac server (the `-t master` option). This avoids clutter by not automatically downloading all branches ever created. But it also means that you will not fetch everything that is on trac by default, and you need to explicitly tell git which branch you want to get from trac. See the *Checking Out Tickets* section for examples.

We set up the remote here to perform read-only operations (fetch) using the git protocol and write operations (push) using the ssh protocol (specified by the `git@` part). To use the ssh protocol you need to have a trac account and to set up your ssh public key as described in *Manually Linking your Public Key to your Trac Account*. Authentication is necessary if you want to upload anything to ensure that it really is from you.

If you want to use ssh only, use these commands:

```
[user@localhost sage]$ git remote add trac git@trac.sagemath.org:sage.git -t master
[user@localhost sage]$ git remote -v
origin      git://github.com/sagemath/sage.git (fetch)
origin      git://github.com/sagemath/sage.git (push)
trac        git@trac.sagemath.org:sage.git (fetch)
trac        git@trac.sagemath.org:sage.git (push)
```

Checking Out Tickets

Trac tickets that are finished or in the process of being worked on can have a git branch attached to them. This is the "Branch:" field in the ticket description. The branch name is generally of the form `u/user/description`, where `user` is the name of the user who made the branch and `description` is some free-form short description (and can include further slashes).

If you want to work with the changes in that remote branch, you must make a local copy. In particular, git has no concept of directly working with the remote branch, the remotes are only bookmarks for things that you can get from/to the remote server. Hence, the first thing you should do is to get everything from the trac server's branch into your local repository. This is achieved by:

```
[user@localhost sage]$ git fetch trac u/user/description
remote: Counting objects: 62, done.
remote: Compressing objects: 100% (48/48), done.
remote: Total 48 (delta 42), reused 0 (delta 0)
Unpacking objects: 100% (48/48), done.
From trac.sagemath.org:sage
* [new branch]      u/user/description -> FETCH_HEAD
```

The `u/user/description` branch is now temporarily (until you fetch something else) stored in your local git database under the alias `FETCH_HEAD`. In the second step, we make it available as a new local branch and switch to it. Your local branch can have a different name, for example:

```
[user@localhost sage]$ git checkout -b my_branch FETCH_HEAD
Switched to a new branch 'my_branch'
```

creates a new branch in your local git repository named `my_branch` and modifies your local Sage filesystem tree to the state of the files in that ticket. You can now edit files and commit changes to your local branch.

Pushing Your Changes to a Ticket

To add your local branch to a trac ticket, you should first decide on a name on the Sage trac repository.

For read/write permissions on git branches, see *Note on Branch Names*

In order to avoid name clashes, you can use `u/your_username/a_description_of_your_branch` (the description can contain slashes, but no spaces). Then:

- **Fill** the `Branch` field of the trac ticket with that name.
- **Push** your branch to trac with either:

  ```
  [user@localhost sage]$ git push --set-upstream trac HEAD:u/user/description
  ```

  if you started the branch yourself and do not follow any other branch, or use:

  ```
  [user@localhost sage]$ git push trac HEAD:u/user/description
  ```

  if your branch already has an upstream branch.

Here, `HEAD` means that you are pushing the most recent commit (and, by extension, all of its parent commits) of the current local branch to the remote branch.

The `Branch` field on the trac ticket can appear in red/green. See *The Ticket Fields* to learn what it means.

Getting Changes

A common task during development is to synchronize your local copy of the branch with the branch on trac. In particular, assume you downloaded somebody else's branch made some suggestions for improvements on the trac ticket. Now the original author incorporated your suggestions into his branch, and you want to get the added changesets to complete your review. Assuming that you originally got your local branch as in *Checking Out Tickets*, you can just issue:

```
[user@localhost sage]$ git pull trac u/user/description
From trac.sagemath.org:sage
 * branch            u/user/description -> FETCH_HEAD
Updating 8237337..07152d8
Fast-forward
 src/sage/tests/cmdline.py      | 3 ++-
 1 file changed, 2 insertions(+), 1 deletions(-)
```

where now `user` is the other developer's trac username and `description` is some description that he chose. This command will download the changes from the originally-used remote branch and merge them into your local branch. If you haven't published your local commits yet then you can also rebase them via:

```
[user@localhost sage]$ git pull -r trac u/user/description
From trac.sagemath.org:sage
 * branch            u/user/description -> FETCH_HEAD
First, rewinding head to replay your work on top of it...
Applying: my local commit
```

See *Merging and Rebasing* section for an in-depth explanation of merge vs. rebase.

So far, we assumed that there are no conflicts. It is unavoidable in distributed development that, sometimes, the same location in a source source file is changed by more than one person. Reconciling these conflicting edits is explained in the *Conflict Resolution* section.

Updating Master

The `master` branch can be updated just like any other branch. However, your local copy of the master branch should stay **identical** to the trac master branch.

If you accidentally added commits to your local copy of `master`, you must delete them before updating the branch.

One way to ensure that you are notified of potential problems is to use `git pull --ff-only`, which will raise an error if a non-trivial merge would be required:

```
[user@localhost sage]$ git checkout master
[user@localhost sage]$ git pull --ff-only trac master
```

If this pull fails, then something is wrong with the local copy of the master branch. To switch to the correct Sage master branch, use:

```
[user@localhost sage]$ git checkout master
[user@localhost sage]$ git reset --hard trac/master
```

Merging and Rebasing

Sometimes, a new version of Sage is released while you work on a git branch.

Let us assume you started `my_branch` at commit `B`. After a while, your branch has advanced to commit `Z`, but you updated `master` (see *Updating Master*) and now your git history looks like this (see *The History*):

```
      X---Y---Z my_branch
     /
A---B---C---D master
```

How should you deal with such changes? In principle, there are two ways:

- **Rebase:** The first solution is to **replay** commits `X`, `Y`, `Z` atop of the new `master`. This is called **rebase**, and it rewrites your current branch:

  ```
  git checkout my_branch
  git rebase -i master
  ```

  In terms of the commit graph, this results in:

  ```
                X'--Y'--Z' my_branch
               /
  A---B---C---D master
  ```

  Note that this operation rewrites the history of `my_branch` (see *Rewriting History*). This can lead to problems if somebody began to write code atop of your commits `X`, `Y`, `Z`. It is safe otherwise.

  Alternatively, you can rebase `my_branch` while updating master at the same time (see *Getting Changes*):

  ```
  git checkout my_branch
  git pull -r master
  ```

- **Merging** your branch with `master` will create a new commit above the two of them:

  ```
  git checkout my_branch
  git merge master
  ```

  The result is the following commit graph:

```
      X---Y---Z---W my_branch
     /           /
A---B---C-------D master
```

- **Pros:** you did not rewrite history (see *Rewriting History*).The additional commit is then easily pushed to the git repository and distributed to your collaborators.
- **Cons:** it introduced an extra merge commit that would not be there had you used rebase.

Alternatively, you can merge `my_branch` while updating master at the same time (see *Getting Changes*):

```
git checkout my_branch
git pull master
```

In case of doubt use merge rather than rebase. There is less risk involved, and rebase in this case is only useful for branches with a very long history.

Finally, **do nothing unless necessary:** it is perfectly fine for your branch to be behind `master`. You can always merge/rebase if/when your branch's name appears in red on its trac page (see *The Ticket Fields*), or when you will really need a feature that is only available in the current master.

Merge Tools

Simple conflicts can be easily solved with git only (see *Conflict Resolution*)

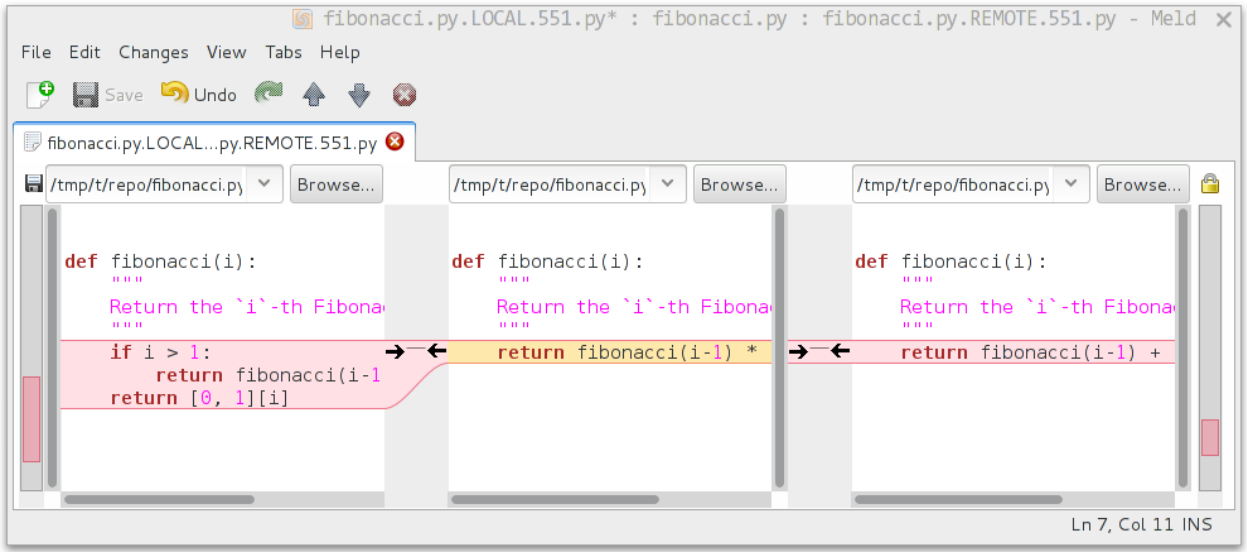
For more complicated ones, a range of specialized programs are available. Because the conflict marker includes the hash of the most recent common parent, you can use a three-way diff:

```
[alice@laptop]$ git mergetool

This message is displayed because 'merge.tool' is not configured.
See 'git mergetool --tool-help' or 'git help config' for more details.
'git mergetool' will now attempt to use one of the following tools:
meld opendiff kdiff3 [...] merge araxis bc3 codecompare emerge vimdiff
Merging:
fibonacci.py

Normal merge conflict for 'fibonacci.py':
  {local}: modified file
  {remote}: modified file
Hit return to start merge resolution tool (meld):
```

If you don't have a favourite merge tool we suggest you try meld (cross-platform). The result looks like the following screenshot.



The middle file is the most recent common parent; on the right is Bob's version and on the left is Alice's conflicting version. Clicking on the arrow moves the marked change to the file in the adjacent pane.

1.3.2 Tips and References

This chapter contains additional material about the git revision control system. It is not necessary if you stick with the Sage development scripts. See *Setting Up Git* for the minimal steps needed for Sage development.

Configuration Tips

Your personal git configurations are saved in the `~/.gitconfig` file in your home directory. Here is an example:

```
[user]
    name = Your Name
    email = you@yourdomain.example.com

[core]
    editor = emacs
```

You can edit this file directly or you can use git to make changes for you:

```
[user@localhost ~] git config --global user.name "Your Name"
[user@localhost ~] git config --global user.email you@yourdomain.example.com
[user@localhost ~] git config --global core.editor vim
```

Aliases

Aliases are personal shortcuts for git commands. For example, you might want to be able to shorten `git checkout` to `git co`. Or you may want to alias `git diff --color-words` (which gives a nicely formatted output of the diff) to `git wdiff`. You can do this with:

```
[user@localhost ~] git config --global alias.ci "commit -a"
[user@localhost ~] git config --global alias.co checkout
[user@localhost ~] git config --global alias.st "status -a"
[user@localhost ~] git config --global alias.stat "status -a"
```

```
[user@localhost ~] git config --global alias.br branch
[user@localhost ~] git config --global alias.wdiff "diff --color-words"
```

The above commands will create an `alias` section in your `.gitconfig` file with contents like this:

```
[alias]
    ci = commit -a
    co = checkout
    st = status -a
    stat = status -a
    br = branch
    wdiff = diff --color-words
```

Editor

To set the editor to use for editing commit messages, you can use:

```
[user@localhost ~] git config --global core.editor vim
```

or set the $EDITOR$ environment variable.

Merging

To enforce summaries when doing merges (`~/.gitconfig` file again):

```
[merge]
    log = true
```

Or from the command line:

```
[user@localhost ~] git config --global merge.log true
```

Fancy Log Output

Here is an alias to get a fancy log output; it should go in the `alias` section of your `.gitconfig` file:

```
lg = log --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)[%
```

Using this `lg` alias gives you the changelog with a colored ascii graph:

```
[user@localhost ~] git lg
* 6d8e1ee - (HEAD, origin/my-fancy-feature, my-fancy-feature) NF - a fancy file (45 minutes ago) [Mat
*   d304a73 - (origin/placeholder, placeholder) Merge pull request #48 from hhuuggoo/master (2 weeks
|\
| * 4aff2a8 - fixed bug 35, and added a test in test_bugfixes (2 weeks ago) [Hugo]
|/
* a7ff2e5 - Added notes on discussion/proposal made during Data Array Summit. (2 weeks ago) [Corran W
* 68f6752 - Initial implimentation of AxisIndexer - uses 'index_by' which needs to be changed to a ca
*   376adbd - Merge pull request #46 from terhorst/master (2 weeks ago) [Jonathan Terhorst]
|\
| * b605216 - updated joshu example to current api (3 weeks ago) [Jonathan Terhorst]
| * 2e991e8 - add testing for outer ufunc (3 weeks ago) [Jonathan Terhorst]
| * 7beda5a - prevent axis from throwing an exception if testing equality with non-axis object (3 wee
```

```
| * 65af65e - convert unit testing code to assertions (3 weeks ago) [Jonathan Terhorst]
| *   956fbab - Merge remote-tracking branch 'upstream/master' (3 weeks ago) [Jonathan Terhorst]
| |\
| |/
```

Tutorials and Summaries

There are many, many tutorials and command summaries available online.

Beginner

- Try Git is an entry-level tutorial you can do in your browser. If you are unfamiliar with revision control, you will want to pay close attention to the "Advice" section toward the bottom.
- Git magic is an extended introduction with intermediate detail.
- The git parable is an easy read explaining the concepts behind git.
- Git foundation expands on the git parable.
- Although it also contains more advanced material about branches and detached head and the like, the visual summaries of merging and branches in Learn Git Branching are really quite helpful.

Advanced

- Github help has an excellent series of how-to guides.
- The pro git book is a good in-depth book on git.
- Github Training has an excellent series of tutorials as well as videos and screencasts.
- The git tutorial.
- Git ready is a nice series of tutorials.
- Fernando Perez' git page contains many links and tips.
- A good but technical page on git concepts
- Git svn crash course: git for those of us used to subversion

Summaries/Cheat Sheets

- A git cheat sheet is a page giving summaries of common commands.
- The git user manual.

Git Best Practices

There are many ways of working with git; here are some posts on the rules of thumb that other projects have come up with:

- Linus Torvalds on git management
- Linus Torvalds on linux git workflow. Summary: use the git tools to make the history of your edits as clean as possible; merge from upstream edits as little as possible in branches where you are doing active development.

Manual Pages Online

You can get these on your own machine with (e.g) `git help push` or (same thing) `git push --help`, but, for convenience, here are the online manual pages for some common commands:

- git add
- git branch
- git checkout
- git clone
- git commit
- git config
- git diff
- git log
- git pull
- git push
- git remote
- git status

1.3.3 Advanced Git

This chapter covers some advanced uses of git that go beyond what is required to work with branches. These features can be used in Sage development, but are not really necessary to contribute to Sage. If you are just getting started with Sage development, you should read *Sage Development Process* instead. If you are new to git, please see *Git the Hard Way*.

Detached Heads and Reviewing Tickets

Each commit is a snapshot of the Sage source tree at a certain point. So far, we always used commits organized in branches. But secretly the branch is just a shortcut for a particular commit, the head commit of the branch. But you can just go to a particular commit without a branch, this is called "detached head". If you have the commit already in your local history, you can directly check it out without requiring internet access:

```
[user@localhost sage]$ git checkout a63227d0636e29a8212c32eb9ca84e9588bbf80b
Note: checking out 'a63227d0636e29a8212c32eb9ca84e9588bbf80b'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b new_branch_name

HEAD is now at a63227d... Szekeres Snark Graph constructor
```

If it is not stored in your local git repository, you need to download it from the trac server first:

```
[user@localhost sage]$ git fetch trac a63227d0636e29a8212c32eb9ca84e9588bbf80b
From ssh://trac/sage
 * branch            a63227d0636e29a8212c32eb9ca84e9588bbf80b -> FETCH_HEAD
[user@localhost sage]$ git checkout FETCH_HEAD
HEAD is now at a63227d... Szekeres Snark Graph constructor
```

Either way, you end up with your current HEAD and working directory that is not associated to any local branch:

```
[user@localhost sage]$ git status
# HEAD detached at a63227d
nothing to commit, working directory clean
```

This is perfectly fine. You can switch to an existing branch (with the usual `git checkout my_branch`) and back to your detached head.

Detached heads can be used to your advantage when reviewing tickets. Just check out the commit (look at the "Commit:" field on the trac ticket) that you are reviewing as a detached head. Then you can look at the changes and run tests in the detached head. When you are finished with the review, you just abandon the detached head. That way you never create a new local branch, so you don't have to type `git branch -D my_branch` at the end to delete the local branch that you created only to review the ticket.

Reset and Recovery

Git makes it very hard to truly mess up. Here is a short way to get back onto your feet, no matter what. First, if you just want to go back to a working Sage installation you can always abandon your working branch by switching to your local copy of the `master` branch:

```
[user@localhost sage]$ git checkout master
```

As long as you did not make any changes to the `master` branch directly, this will give you back a working Sage.

If you want to keep your branch but go back to a previous commit you can use the *reset* command. For this, look up the commit in the log which is some 40-digit hexadecimal number (the SHA1 hash). Then use `git reset --hard` to revert your files back to the previous state:

```
[user@localhost sage]$ git log
...
commit eafaedad5b0ae2013f8ae1091d2f1df58b72bae3
Author: First Last <user@email.com>
Date:   Sat Jul 20 21:57:33 2013 -0400

    Commit message
...
[user@localhost sage]$ git reset --hard eafae
```

Warning: Any *uncommitted* changes will be lost!

You only need to type the first couple of hex digits, git will complain if this does not uniquely specify a commit. Also, there is the useful abbreviation `HEAD~` for the previous commit and `HEAD~n`, with some integer `n`, for the n-th previous commit.

Finally, perhaps the ultimate human error recovery tool is the reflog. This is a chronological history of git operations that you can undo if needed. For example, let us assume we messed up the *git reset* command and went back too far (say, 5 commits back). And, on top of that, deleted a file and committed that:

```
[user@localhost sage]$ git reset --hard HEAD~5
[user@localhost sage]$ git rm sage
[user@localhost sage]$ git commit -m "I shot myself into my foot"
```

Now we cannot just checkout the repository from before the reset, because it is no longer in the history. However, here is the reflog:

```
[user@localhost sage]$ git reflog
2eca2a2 HEAD@{0}: commit: I shot myself into my foot
b4d86b9 HEAD@{1}: reset: moving to HEAD~5
af353bb HEAD@{2}: checkout: moving from some_branch to master
1142feb HEAD@{3}: checkout: moving from other_branch to some_branch
...
```

The `HEAD@{n}` revisions are shortcuts for the history of git operations. Since we want to rewind to before the erroneous *git reset* command, we just have to reset back into the future:

```
[user@localhost sage]$ git reset --hard HEAD@{2}
```

Rewriting History

Git allows you to rewrite history, but be careful: the SHA1 hash of a commit includes the parent's hash. This means that the hash really depends on the entire content of the working directory; every source file is in exactly the same state as when the hash was computed. This also means that you can't change history without modifying the hash. If others branched off your code and then you rewrite history, then the others are thoroughly screwed. So, ideally, you would only rewrite history on branches that you have not yet pushed to trac.

As an advanced example, consider three commits A, B, C that were made on top of each other. For simplicity, we'll assume they just added a file named `file_A.py`, `file_B.py`, and `file_C.py`

```
[user@localhost]$ git log --oneline
9621dae added file C
7873447 added file B
bf817a5 added file A
5b5588e base commit
```

Now, let's assume that the commit B was really independent and ought to be on a separate ticket. So we want to move it to a new branch, which we'll call `second_branch`. First, branch off at the base commit before we added A:

```
[user@localhost]$ git checkout 5b5588e
Note: checking out '5b5588e'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b new_branch_name

HEAD is now at 5b5588e... base commit
[user@localhost]$ git checkout -b second_branch
Switched to a new branch 'second_branch'
[user@localhost]$ git branch
  first_branch
```

```
* second_branch
[user@localhost]$ git log --oneline
5b5588e base commit
```

Now, we make a copy of commit B in the current branch:

```
[user@localhost]$ git cherry-pick 7873447
[second_branch 758522b] added file B
 1 file changed, 1 insertion(+)
 create mode 100644 file_B.py
[user@localhost]$ git log --oneline
758522b added file B
5b5588e base commit
```

Note that this changes the SHA1 of the commit B, since its parent changed! Also, cherry-picking *copies* commits, it does not remove them from the source branch. So we now have to modify the first branch to exclude commit B, otherwise there will be two commits adding `file_B.py` and our two branches would conflict later when they are being merged into Sage. Hence, we first reset the first branch back to before B was added:

```
[user@localhost]$ git checkout first_branch
Switched to branch 'first_branch'
[user@localhost]$ git reset --hard bf817a5
HEAD is now at bf817a5 added file A
```

Now we still want commit C, so we cherry-pick it again. Note that this works even though commit C is, at this point, not included in any branch:

```
[user@localhost]$ git cherry-pick 9621dae
[first_branch 5844535] added file C
 1 file changed, 1 insertion(+)
 create mode 100644 file_C.py
[user@localhost]$ git log --oneline
5844535 added file C
bf817a5 added file A
5b5588e base commit
```

And, again, we note that the SHA1 of commit C changed because its parent changed. Voila, now you have two branches where the first contains commits A, C and the second contains commit B.

Interactively Rebasing

An alternative approach to *Rewriting History* is to use the interactive rebase feature. This will open an editor where you can modify the most recent commits. Again, this will naturally modify the hash of all changed commits and all of their children.

Now we start by making an identical branch to the first branch:

```
[user@localhost]$ git log --oneline
9621dae added file C
7873447 added file B
bf817a5 added file A
5b5588e base commit
[user@localhost]$ git checkout -b second_branch
Switched to a new branch 'second_branch'
[user@localhost]$ git rebase -i HEAD~3
```

This will open an editor with the last 3 (corresponding to `HEAD~3`) commits and instuctions for how to modify them:

```
pick bf817a5 added file A
pick 7873447 added file B
pick 9621dae added file C

# Rebase 5b5588e..9621dae onto 5b5588e
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

To only use commit B, we delete the first and third line. Then save and quit your editor, and your branch now consists only of the B commit.

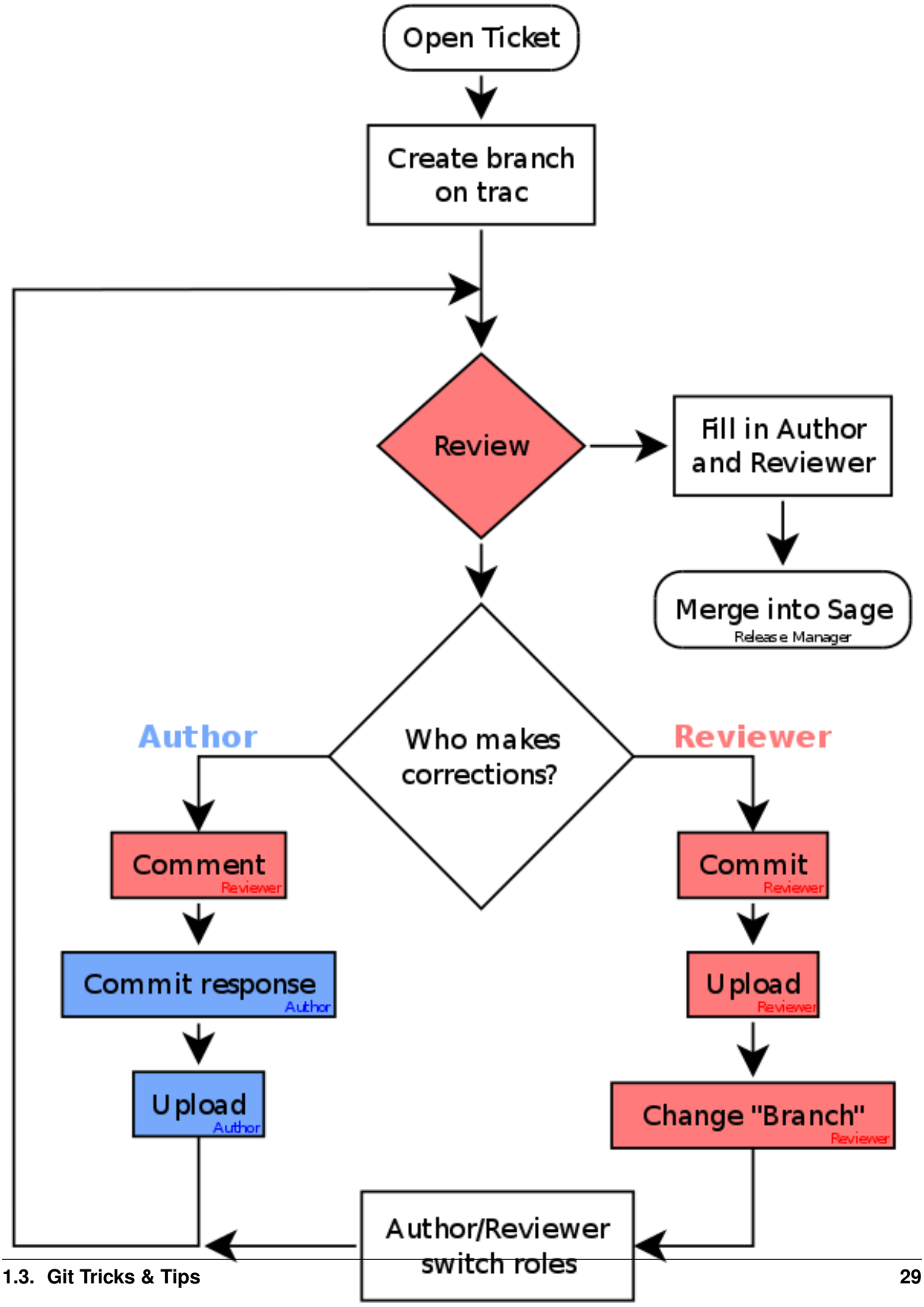
You still have to delete the B commit from the first branch, so you would go back (`git checkout first_branch`) and then run the same `git rebase -i` command and delete the B commit.

1.3.4 Distributed Development

Git is a tool to exchange commits (organized into branches) with other developers. As a distributed revision control system, it does not have the notion of a central server. The Sage trac server is just one of many possible remote repositories from your point of view. This lets you use and experiment with different ways to interact with other developers. In this chapter, we describe some common ways to develop for Sage.

For simplicity, let us assume two developers (Alice and Bob) are collaborating on a ticket. The first step of opening the ticket is always the same, and could be performed by either Alice or Bob or a third person.

Simple Workflow



1. Alice creates a *new local branch* and *commits* changes to the Sage sources.
2. Alice *uploads her branch* to the trac server. This fills in the "Branch:" field with her remote branch name `u/alice/description`.
3. Bob *downloads Alice's branch*, looks through the source, and leaves a comment on the ticket about a mistake in Alice's code.
4. Alice fixes the bug on top of her current branch, and uploads the updated branch.
5. Bob *retrieves Alice's updates* and reviews the changes.
6. Once Bob is satisfied, he sets the ticket to positive review. The "Author:" field is set to Alice's full name, and the "Reviewer:" field is set to Bob's full name.

Alternatively, Bob might want to make some changes himself. Then, instead, we would have

3. Bob *downloads Alice's branch*, makes changes, and *commits* them to his local branch.
4. Bob *uploads his branch* to the trac server. This fills in the "Branch:" field with his remote branch name `u/bob/description`.
5. Alice *downloads Bob's branch* and reviews his changes.
6. Once Alice is satisfied, she sets the ticket to positive review. If both contributions are of comparable size, then the "Author:" and "Reviewer:" fields are set to both Alice's and Bob's full name.

Public Repository

In addition to the user branches (`u/<user>/<description>` on the Sage trac server with `<user>` replaced by your trac user name) that only you can write to, you can also create a public branch that everybody with a trac account can write to. These start with `public/` plus some description. To avoid branch name collisions it is a good idea to include your trac user name in the branch name, so it is recommended that you use `public/<user>/<description>` as the branch name. Now all ticket authors push to the same remote branch.

1. Alice creates a *new local branch* and *commits* some changes to the Sage library.
2. Alice *uploads her branch* as a public branch to the trac server. This fills in the "Branch:" field with her remote branch name `public/alice/description`.
3. Bob *downloads Alice's branch* and makes changes to his local copy.
4. Bob *commits* changes to his local branch of the Sage sources.
5. Bob uploads his changes to the joint remote repository:

   ```
   [bob@localhost sage]$ git push trac local_branch:public/alice/description
   ```

6. Alice *retrieves Bob's updates*, makes more changes, commits, and pushes them to trac.
7. Charly reviews the final version, and then sets the ticket to positive review. The "Author:" field is set to Alice's and Bob's full name, and the "Reviewer:" field is set to Charly's full name.

GitHub

Yet another possible workflow is to use GitHub (or any other third-party git repository) to collaboratively edit your new branch, and only push the result to trac once you and your ticket co-authors are satisfied.

Fork

The first step is to create your own fork of the Sage repository; simply click "Fork" on the Sage GitHub repository. Then add it as one of the remotes to your local Sage repository. In the following, we will use the label "github" for this remote repository, though you are of course free to use a different one:

```
$ git remote add github git@github.com:github_user_name/sage.git
$ git remote -v
github      git@github.com:github_user_name/sage.git (fetch)
github      git@github.com:github_user_name/sage.git (push)
trac        git@trac.sagemath.org:sage.git (fetch)
trac        git@trac.sagemath.org:sage.git (push)
$ git fetch github
remote: Counting objects: 107, done.
remote: Compressing objects: 100% (63/63), done.
remote: Total 74 (delta 41), reused 40 (delta 10)
Unpacking objects: 100% (74/74), done.
From github.com:github_user_name/sage
* [new branch]      master     -> github/master
```

Develop

You now use the github repository to develop your ticket branch; First create a new branch:

```
$ git checkout -b my_branch --track github/master
Branch my_branch set up to track remote branch master from github.
Switched to a new branch 'my_branch'
$ git push github my_branch
Total 0 (delta 0), reused 0 (delta 0)
To git@github.com:github_user_name/sage.git
 * [new branch]      my_branch -> my_branch
```

Because of the `--track` option, the `git pull` command will default to downloading your coauthor's changes from your github branch. Alternatively, you can create a new branch on your fork's GitHub webpage.

At this point you can use the GitHub workflow that you prefer. In particular, your choices are

- Give your coauthors write permissions to your github fork. Every author edits/commits to their own local copy and they jointly push to your github branch.
- Have every coauthor create their own fork and send you (the lead author) pull requests to your GitHub fork.
- Use the GitHub web page editing & commiting feature, that way you can make changes without ever using your local machine.

Push to Trac

When you are satisfied with your branch, you push it to the Sage trac server:

```
$ git push trac HEAD:u/user/description
```

and then fill in the "Branch" field in the trac ticket description as explained in *Pushing Your Changes to a Ticket*.

CHAPTER

TWO

SAGE TRAC AND TICKETS

All changes to Sage source code require a ticket on the Sage trac server.

2.1 The Sage Trac Server

All changes to Sage source code have to go through the Sage Trac development server. The purpose of the Sage trac server is to

1. Provide a place for discussion on issues and store a permanent record.
2. Provide a repository of source code and all proposed changes.
3. Link these two together.

There is also a wiki for more general organizational web pages, like Sage development workshops.

Thus if you find a bug in Sage, if you have new code to submit, want to review new code already written but not yet included in Sage, 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 search or browse the tickets. For a list of recent changes, just visit the Sage trac timeline.

2.1.1 Obtaining an Account

You need a trac account if you want to *change* anything on the Sage trac server, even if you just want to comment on a ticket. To obtain one, send an email to `sage-trac-account@googlegroups.com` containing:

- your full name,
- preferred username,
- contact email,
- and reason for needing a trac account

Your trac account also grants you access to the sage wiki. Make sure you understand the review process, and the procedures for opening and closing tickets before making changes. The remainder of this chapter contains various guidelines on using the trac server.

2.1.2 Trac authentication through SSH

There are two avenues to prove to the trac server that you are who you claim to be. First, to change the ticket web pages you need to log in to trac using a username/password. Second, there is public key cryptography used by git when copying new source files to the repository. This section will show you how to setup both.

Generating and Uploading your SSH Keys

The git installation on the development server uses SSH keys to decide if and where you are allowed to upload code. No SSH key is required to report a bug or comment on a ticket, but as soon as you want to contribute code yourself you need to provide trac with the public half of your own personal key. In recent versions of Sage, you can use Sage to generate an upload an SSH key

```
sage: dev.upload_ssh_key()
The trac git server requires your SSH public key to be able to identify you.
Upload "/home/vbraun/.ssh/id_dsa.pub" to trac? [Yes/no] y
Trac username: user
Trac password:
Your key has been uploaded.
```

You can also manually generate an SSH key and upload it to trac. This is described in the following two sections.

Manually Generating your SSH Keys

If you don't have a private key yet, you can create it with the `ssh-keygen` tool:

```
[user@localhost ~]$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/user/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/user/.ssh/id_rsa.
Your public key has been saved in /home/user/.ssh/id_rsa.pub.
The key fingerprint is:
ce:32:b3:de:38:56:80:c9:11:f0:b3:88:f2:1c:89:0a user@localhost
The key's randomart image is:
+--[ RSA 2048]----+
|  ....           |
|   ..            |
|   .o+           |
| o o+o.          |
|E + .  .S        |
|+o .   o.        |
|. o   +.o        |
|      oB         |
|     o+..        |
+-----------------+
```

This will generate a new random private RSA key in the `.ssh` folder in your home directory. By default, they are

`~/.ssh/id_rsa` Your private key. Keep safe. **Never** hand it out to anybody.

`~/.ssh/id_rsa.pub` The corresponding public key. This and only this file can be safely disclosed to third parties.

The `ssh-keygen` tool will let you generate a key with a different file name, or protect it with a passphrase. Depending on how much you trust your own computer or system administrator, you can leave the passphrase empty to be able to login without any human intervention.

If you have accounts on multiple computers you can use the SSH keys to log in. Just copy the **public** key file (ending in `.pub`) to `~/.ssh/authorized_keys` on the remote computer and make sure that the file is only read/writeable by yourself. Voila, the next time you ssh into that machine you don't have to provide your password.

Manually Linking your Public Key to your Trac Account

The Sage trac server needs to know one of your public keys. You can upload it in the preferences, that is

1. Go to http://trac.sagemath.org
2. Log in with your trac username/password
3. Click on "Preferences"
4. Go to the "SSH Keys" tab
5. Paste the content of your public key file (e.g. `~/.ssh/id_rsa.pub`)
6. Click on "Save changes"

Note that this does **not** allow you to ssh into any account on trac, it is only used to authenticate you to the gitolite installation on trac. You can test that you are being authenticated correctly by issuing some basic gitolite commands, for example:

```
[user@localhost ~]$ ssh git@trac.sagemath.org info
hello user, this is git@trac running gitolite3 (unknown) on git 1.7.9.5

 R W      sage
[user@localhost ~]$ ssh git@trac.sagemath.org help
hello user, this is gitolite3 (unknown) on git 1.7.9.5

list of remote commands available:

    desc
    help
    info
    perms
    writable
```

2.1.3 Reporting Bugs

If you think you have found a bug in Sage, here is the procedure:

- Search through our Google groups for postings related to your possible bug (it may have been fixed/reported already):
  - `sage-devel`: http://groups.google.com/group/sage-devel
  - `sage-support`: http://groups.google.com/group/sage-support

  Similarly, you can search *The Sage 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. A bug report should contain:
  - An explicit and **reproducible example** illustrating your bug (and/or the steps required to reproduce the buggy behavior).
  - The **version** of Sage you run, as well as the version of the optional packages that may be involved in the bug.
  - Describe your **operating system** as accurately as you can and your architecture (32-bit, 64-bit, ...).
- You might be asked to open a new ticket. In this case, follow the *Guidelines for Opening Tickets*.

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

2.1.4 Guidelines for Opening Tickets

In addition to bug reports (see *Reporting Bugs*), you should also open a ticket if you have some new code that makes Sage a better tool. 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.

- Do you already have a **trac account**? If not, *click here*.

Before opening a new ticket, consider the following points:

- 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.
- For bug reports: the ticket's description should contain the information described at *Reporting Bugs*.
- If appropriate, provide URLs to background information or sage-devel conversation relevant to the problem you are reporting.

When creating the ticket, you may find useful to read *The Ticket Fields*.

Unless you know what you are doing, leave the milestone field to its default value.

2.1.5 The Ticket Fields

When you open a new ticket or change an existing ticket, you will find a variety of fields that can be changed. Here is a comprehensive overview (for the 'status' field, see *The status of a ticket*):

- **Reported by:** The trac account name of whoever created the ticket. Cannot be changed.
- **Owned by:** Trac account name of owner, by default the person in charge of the Component (see below). Generally not used in the Sage trac.
- **Type:** One of `enhancement` (e.g. a new feature), `defect` (e.g. a bug fix), or `task` (rarely used).
- **Priority:** The priority of the ticket. Keep in mind that the "blocker" label should be used very sparingly.
- **Milestone:** Milestones are usually goals to be met while working toward a release. In Sage's trac, we use milestones instead of releases. Each ticket must have a milestone assigned. If you are unsure, assign it to the current milestone.
- **Component:** A list of components of Sage, pick one that most closely matches the ticket.
- **Keywords:** List of keywords. Fill in any keywords that you think will make your ticket easier to find. Tickets that have been worked on at Sage days `NN` (some number) ofter have `sdNN` as keyword.
- **Cc:** List of trac user names to Cc (send emails for changes on the ticket). Note that users that enter a comment are automatically substcribed to email updates and don't need to be listed under Cc.
- **Merged in:** The Sage release where the ticket was merged in. Only changed by the release manager.
- **Authors:** Real name of the ticket author(s).
- **Reviewers:** Real name of the ticket reviewer(s).

- **Report Upstream:** If the ticket is a bug in an upstream component of Sage, this field is used to summarize the communication with the upstream developers.
- **Work issues:** Issues that need to be resolved before the ticket can leave the "needs work" status.
- **Branch:** The Git branch containing the ticket's code (see *Branching Out*). It is displayed in green color, unless there is a conflict between the branch and the latest beta release (red color). In this case, the branch should be merged or rebased.
- **Dependencies:** Does the ticket depend on another ticket? Sometimes, a ticket requires that another ticket be applied first. If this is the case, put the dependencies as a comma-separated list (`#1234, #5678`) into the "Dependencies:" field.
- **Stopgaps:** See *Stopgaps*.

2.1.6 The status of a ticket

The status of a ticket appears right next to its number, at the top-left corner of its page. It indicates who has to work on it.

- **new** – the ticket has only been created (or the author forgot to change the status to something else).

  If you want to work on it yourself it is better to leave a comment to say so. It could avoid having two persons doing the same job.
- **needs_review** – the code is ready to be peer-reviewed. If the code is not yours, then you can review it. See *The reviewer's check list*.
- **needs_work** – something needs to be changed in the code. The reason should appear in the comments.
- **needs_info** – somebody has to answer a question before anything else can happen. It should be clear from the comments.
- **positive_review** – the ticket has been reviewed, and the release manager will close it.

The status of a ticket can be changed using a form at the bottom of the ticket's page. Leave a comment explaining your reasons whenever you change it.

2.1.7 Stopgaps

When Sage returns wrong results, two tickets should be opened:

- A main ticket with all available details.
- A "stopgap" ticket (e.g. trac ticket #12699)

This second ticket does not fix the problem but adds a warning that will be printed whenever anyone uses the relevant code. This, until the problem is finally fixed.

To produce the warning message, use code like the following:

```
from sage.misc.stopgap import stopgap
stopgap("This code contains bugs and may be mathematically unreliable.",
    TICKET_NUM)
```

Replace `TICKET_NUM` by the ticket number for the main ticket. On the main trac ticket, enter the ticket number for the stopgap ticket in the "Stopgaps" field (see *The Ticket Fields*). Stopgap tickets should be marked as blockers.

Note: If mathematically valid code causes Sage to raise an error or crash, for example, there is no need for a stopgap. Rather, stopgaps are to warn users that they may be using buggy code; if Sage crashes, this is not an issue.

2.1.8 Working on Tickets

If you manage to fix a bug or enhance Sage you are our hero. See *Sage Development Process* for making changes to the Sage source code, uploading them to the Sage trac server, and finally putting your new branch on the trac ticket. The following are some other relevant issues:

- The Patch buildbot wil automatically test your ticket. See the patchbot wiki for more information about its features and limitations. Make sure that you look at the log, especially if the patch buildbot did not give you the green blob.
- Every bug fixed should result in a doctest.
- 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. In this context, this 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.

2.1.9 Reviewing and closing Tickets

Tickets can be closed when they have positive review or for other reasons. To learn how to review, please see *The reviewer's check list*.

Only the Sage release manager will close tickets. Most likely, this is not you nor will your trac account have the necessary permissions. 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-duplictate/invalid/wontfix*. You should also comment on the ticket, explaining why it should be closed. If another developer agrees, he sets the ticket to *positive review*.

A related issue is re-opening tickets. You should refrain from re-opening a ticket that is already closed. Instead, open a new ticket and provide a link in the description to the old ticket.

2.1.10 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.

CHAPTER

THREE

WRITING CODE FOR SAGE

3.1 General Conventions

There are many ways to contribute to Sage including sharing scripts and Sage worksheets that implement new functionality using Sage, improving to the Sage library, or to working on the many underlying libraries distributed with Sage [1]. This guide focuses on editing the Sage library itself.

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.

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.

3.1.1 Python Code Style

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.
- Whitespace before and after assignment and binary operator of the lowest priority in the expression:

```
i = i + 1
c = (a+b) * (a-b)
```

- No whitespace before or after the = sign if it is used for keyword arguments:

```
def complex(real, imag=0.0):
    return magic(r=real, i=imag)
```

- No whitespace immediately inside parenthesis, brackets, and braces:

```
spam(ham[1], {eggs: 2})
[i^2 for i in range(3)]
```

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

- 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():
    return 1
```

  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`.

- Use CamelCase for class names:

```
class SomeValue(object):
    def __init__(self, x):
    self._x  = 1
```

  and factory functions that mimic object constructors, for example `PolynomialRing` or:

```
def SomeIdentityValue(x):
    return SomeValue(1)
```

3.1.2 Files and Directory Structure

Roughly, the Sage directory tree is layout like this. Note that we use `SAGE_ROOT` in the following as a shortcut for the (arbitrary) name of the directory containing the Sage sources:

```
SAGE_ROOT/
    sage          # the Sage launcher
    Makefile      # top level Makefile
    build/        # sage's build system
        deps
        install
        ...
        pkgs/     # install, patch, and metadata from spkgs
    src/
        setup.py
        module_list.py
        ...
        sage/     # sage library (formerly devel/sage-main/sage)
        ext/      # extra sage resources (formerly devel/ext-main)
        mac-app/  # would no longer have to awkwardly be in extcode
        bin/      # the scripts in local/bin that are tracked
    upstream/     # tarballs of upstream sources
    local/        # installed binaries
```

Python Sage library code goes into `src/` and 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.

Note: You are encouraged to include miscellaneous notes, emails, design discussions, etc., in your package. Make these plain text files (with extension `.txt`) in a subdirectory called `notes`. For example, see `SAGE_ROOT/src/sage/ext/notes/`.

If you want to create a new directory in the Sage library `SAGE_ROOT/src/sage` (say, `measure_theory`), that directory should contain a file `__init__.py` that contains the single line `import all` 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 that are important enough to be in the Sage's global namespace at startup. The file `all.py` might look like this:

```
from borel_measure import BorelMeasure
from banach_tarski import BanachTarskiParadox
```

but it is generally better to use the lazy import framework:

```
from sage.misc.lazy_import import lazy_import
lazy_import('sage.measure_theory.borel_measue', 'BorelMeasure')
lazy_import('sage.measure_theory.banach_tarski', 'BanachTarskiParadox')
```

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

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

3.1.3 Learn by copy/paste

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.

3.1.4 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

EXAMPLES::

<Lots and lots of examples>
"""

#*****************************************************************************
#       Copyright (C) 2013 YOUR NAME <your email>
#
# This program is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License 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/
#*****************************************************************************
```

As an example, see `SAGE_ROOT/src/sage/rings/integer.pyx` which contains the implementation for **Z**. The `AUTHORS:` section is redundant, the authoritative log for who wrote what is always the git repository (see the

output of `git blame`). Nevertheless, it is sometimes useful to have a very rough overview over the history, especially if a lot of people have been working on that source file.

All code included with Sage must be licensed under the GPLv2+ or a compatible, that is, less restrictive license (e.g. the BSD license).

3.1.5 Documentation Strings

The docstring of a function: content

Every function must have a docstring that includes the following information. You can use the existing functions of Sage as templates.

- A **one-sentence description** of the function.

  It must be followed by a blank line and end in a period. It describes the function or method's effect as a command ("Do this", "Return that"), not as a description like "Returns the pathname ...".

- An **INPUT** and an **OUTPUT** block describing the input/output of the function. This is not optional.

  The INPUT block describes all arguments that the function accepts, and the OUTPUT section describes its expected output.

  1. 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, rather than `int`.
  2. Mention the default values of the input arguments when applicable.

  Example:

```
INPUT:

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

OUTPUT:

A 5-tuple consisting of integers in this order:

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
```

  You can start the OUTPUT block with a dash if you prefer:

```
OUTPUT:

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

- An **EXAMPLES** block for examples. This is not optional.

  These examples are used both for:

  1. Documentation
  2. Automatic testing before each release.

  They should have good coverage of the functionality in question.

- A **SEEALSO** block (highly recommended) with links to related parts of Sage. This helps users find the features that interests them and discover the new ones.

```
.. SEEALSO::

    :ref:`chapter-sage_manuals_links`,
    :meth:`sage.somewhere.other_useful_method`,
    :mod:`sage.some.related.module`.
```

  See *Hyperlinks* for details on how to setup link in Sage.

- An **ALGORITHM** block (optional).

  It indicates what algorithm and/or what software is used, e.g. `ALGORITHM: Uses Pari`. Here's a longer example with a bibliographical reference:

```
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.
```

- A **NOTE** block for tips/tricks (optional).

```
.. NOTE::

    You should note that this sentence is indented at least 4
    spaces. Never use the tab character.
```

- A **WARNING** block for critical information about your code (optional).

  For example known situations for which the code breaks, or anything that the user should be aware of.

```
.. 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 future improvements (optional).

  It can contain disabled doctests to demonstrate the desired feature. Here's an example of a TODO block:

```
    .. TODO::

        Add to ``have_fresh_beers`` an interface with the faster
        algorithm "Buy a Better Fridge" (BaBF)::

            sage: have_fresh_beers('Bière de l\'Yvette', algorithm="BaBF") # not implemented
            Enjoy !
```

- A **PLOT** block to illustrate with pictures the output of a function.

  Generate with Sage code an object `g` with a `.plot` method, then call `sphinx_plot(g)`:

```
  .. PLOT::

      g = graphs.PetersenGraph()
      sphinx_plot(g)
```

- A **REFERENCES** block to list related books or papers (optional)

  It should cite the books/research papers relevant to the code, e.g. the source of the algorithm that it implements.

```
  This docstring is referencing [SC]_. Just remember that references
  are global, so we can also reference to [Nat2000]_ in the ALGORITHM
  block, even if it is in a separate file. However we would not
  include the reference here since it would cause a conflict.

  REFERENCES:

  .. [SC] Conventions for coding in sage.
     http://www.sagemath.org/doc/developer/conventions.html.
```

  See the Sphinx/ReST markup for citations. For links toward trac tickets or wikipedia, see *Hyperlinks*.

- A **TESTS** block (optional)

  Formatted just like EXAMPLES, containing tests that are not relevant to users.

Template

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

```
def point(self, x=1, y=2):
    r"""
    Return 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 begin at the same indentation
      as the backtick.

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

    OUTPUT:

    The point as a tuple.

    .. SEEALSO::
```

```
        :func:`line`

    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 'r' to an integer

    .. NOTE::

        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 ...."
    """
    <body of the function>
```

You are strongly encouraged to:

- Use LaTeX typesetting (see *LaTeX Typesetting*).
- Liberally describe what the examples do.

  Note: There must be a blank line after the example code and before the explanatory text for the next example (indentation is not enough).

- Illustrate the exceptions raised by the function with examples (as given above: "It is an error to [..]", ...)
- Include many examples.

  They are helpful for the users, 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.

Private functions

Functions whose names start with an underscore are considered private. They do not appear in the reference manual, and their docstring should not contain any information that is crucial for Sage users. You can make their docstrings be part of the documentation of another method. For example:

```
class Foo(SageObject):

    def f(self):
        """
        <usual docstring>

        .. automethod:: _f
        """
        return self._f()

    def _f(self):
         """
         This would be hidden without the ``.. automethod::``
         """
```

Private functions should contain an EXAMPLES (or TESTS) block.

A special case is the constructor `__init__`: due to its special status the `__init__` docstring is used as the class docstring if there is not one already. That is, you can do the following:

```
sage: class Foo(SageObject):
....:     # no class docstring
....:     def __init__(self):
....:         """Construct a Foo."""
sage: foo = Foo()
sage: from sage.misc.sageinspect import sage_getdoc
sage: sage_getdoc(foo)              # class docstring
'Construct a Foo.\n'
sage: sage_getdoc(foo.__init__)     # constructor docstring
'Construct a Foo.\n'
```

LaTeX Typesetting

In Sage's documentation LaTeX code is allowed and is marked with **backticks or dollar signs**:

`` `x^2 + y^2 = 1` `` and `$x^2 + y^2 = 1$` both yield $x^2 + y^2 = 1$.

Backslashes: For LaTeX commands containing backslashes, either use double backslashes or begin the docstring with a `r"""` instead of `"""`. Both of the following are valid:

```
def cos(x):
    """
    Return `\\cos(x)`.
    """

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

MATH block: This is similar to the LaTeX syntax `\[<math expression>\]` (or `$$<math expression>$$`). For instance:

```
.. MATH::

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

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

The **aligned** environment works as it does in LaTeX:

```
.. MATH::

    \begin{aligned}
     f(x) & = x^2 - 1 \\
     g(x) & = x^x - f(x - 2)
    \end{aligned}
```

$$\begin{aligned} f(x) &= x^2 - 1 \\ g(x) &= x^x - f(x - 2) \end{aligned}$$

When building the PDF documentation, everything is translated to LaTeX and each MATH block is automatically wrapped in a math environment – in particular, it is turned into `\begin{gather} block \end{gather}`. So if you want to use a LaTeX environment (like `align`) which in ordinary LaTeX would not be wrapped like this, you must add a **:nowrap:** flag to the MATH mode. See also Sphinx's documentation for math blocks.

```
.. MATH::
   :nowrap:

   \begin{align}
      1+...+n &= n(n+1)/2\\
      &= O(n^2)\\
   \end{tabular}
```

$$\begin{aligned} 1 + ... + n &= n(n+1)/2 \\ &= O(n^2) \end{aligned} \tag{3.1}$$

(3.2)

(3.3)

Readability balance: in the interactive console, LaTeX formulas contained in the documentation are represented by their LaTeX code (with backslashes stripped). In this situation `\\frac{a}{b}` is less readable than `a/b` or `a b^{-1}` (some users may not even know LaTeX code). Make it pleasant for everybody as much as you can manage.

Commons rings ($\mathbf{Z}, \mathbf{N}, ...$): The Sage LaTeX style is to typeset standard rings and fields using the locally-defined macro `\\Bold` (e.g. `\\Bold{Z}` gives $\mathbf{Z}$).

Shortcuts are available which preserve readability, e.g. `\\ZZ` ($\mathbf{Z}$), `\\RR` ($\mathbf{R}$), `\\CC` ($\mathbf{C}$), and `\\QQ` ($\mathbf{Q}$). They appear as LaTeX-formatted `\\Bold{Z}` in the html manual, and as `Z` in the interactive help. Other examples: `\\GF{q}` ($\mathbf{F}_q$) and `\\Zmod{p}` ($\mathbf{Z}/p\mathbf{Z}$).

See the file `SAGE_ROOT/src/sage/misc/latex_macros.py` for a full list and for details about how to add more macros.

Writing Testable Examples

The examples from Sage's documentation have a double purpose:

- They provide **illustrations** of the code's usage to the users
- They are **tests** that are checked before each release, helping us avoid new bugs.

All new doctests added to Sage should **pass all tests** (see *Running Sage's doctests*), i.e. running `sage -t your_file.py` should not give any error messages. Below are instructions about how doctests should be written.

What doctests should test:

- **Interesting examples** of what the function can do. This will be the most helpful to a lost user. It is also the occasion to check famous theorems (just in case):

```
sage: is_prime(6) # 6 is not prime
False
sage: 2 * 3 # and here is a proof
6
```

- All **meaningful combinations** of input arguments. For example a function may accept an `algorithm="B"` argument, and doctests should involve both `algorithm="A"` and `algorithm="B"`.
- **Corner cases:** the code should be able to handle a 0 input, or an empty set, or a null matrix, or a null function, ... All corner cases should be checked, as they are the most likely to be broken, now or in the future. This probably belongs to the TESTS block (see *The docstring of a function: content*).
- **Systematic tests** of all small-sized inputs, or tests of **random** instances if possible.

Note: Note that **TestSuites** are an automatic way to generate some of these tests in specific situations. See `SAGE_ROOT/src/sage/misc/sage_unittest.py`.

The syntax:

- **Environment:** doctests should work if you copy/paste them in Sage's interactive console. For example, the function `AA()` in the file `SAGE_ROOT/src/sage/algebras/steenrod/steenrod_algebra.py` includes an EXAMPLES block containing the following:

```
sage: from sage.algebras.steenrod.steenrod_algebra import AA as A
sage: A()
mod 2 Steenrod algebra, milnor basis
```

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

- **Preparsing:** As in Sage's console, $4/3$ returns $4/3$ and not 1 as in Python 2.7. Testing occurs with full Sage preparsing of input within the standard Sage shell environment, as described in *Sage Preparsing*.
- **Writing files:** If a test outputs to a file, the file should be a temporary file. Use `tmp_filename()` to get a temporary filename, or `tmp_dir()` to get a temporary directory. An example from `SAGE_ROOT/src/sage/plot/graphics.py`):

```
sage: plot(x^2 - 5, (x, 0, 5), ymin=0).save(tmp_filename(ext='.png'))
```

- **Multiline doctests:** You may write tests that span multiple lines, using the line continuation marker `....:`

```
sage: for n in srange(1,10):
....:     if n.is_prime():
....:         print n,
2 3 5 7
```

- **Split long lines:** You may want to split long lines of code with a backslash. Note: this syntax is non-standard and may be removed in the future:

```
sage: n = 123456789123456789123456789\
....:     123456789123456789123456789
sage: n.is_prime()
False
```

- **Doctests flags:** flags are available to change the behaviour of doctests: see *Special Markup to Influence Tests*.

Special Markup to Influence Tests

There are a number of magic comments that you can put into the example code that change how the output is verified by the Sage doctest framework. Here is a comprehensive list:

- **random:** The line will be executed, but its output will not be checked with the output in the documentation string:

```
sage: c = CombinatorialObject([1,2,3])
sage: hash(c)  # random
1335416675971793195
sage: hash(c)  # random
This doctest passes too, as the output is not checked
```

  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.

  When possible, avoid the problem, e.g.: rather than checking the value of the hash in a doctest, one could illustrate successfully using it as a key in a dict.

- **long time:** The line is only tested if the `--long` option is given, e.g. `sage -t --long f.py`.

  Use it for doctests that take more than a second to run. No example should take more than about 30 seconds:

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

- **tol** or **tolerance:** The numerical values returned by the line are only verified to the given tolerance. It is useful when the output is subject to numerical noise due to system-dependent (floating point arithmetic, math libraries, ...) or non-deterministic algorithms.
  - This may be prefixed by `abs[olute]` or `rel[ative]` to specify whether to measure **absolute** or **relative** error (see the Wikipedia article Approximation_error).
  - If none of `abs/rel` is specified, the error is considered to be `absolute` when the expected value is **zero**, and is `relative` for **nonzero** values.

```
sage: n(pi)  # abs tol 1e-9
3.14159265358979
sage: n(pi)  # rel tol 2
6
sage: n(pi)  # abs tol 1.41593
2
sage: K.<zeta8> = CyclotomicField(8)
sage: N(zeta8)  # absolute tolerance 1e-10
0.7071067812 + 0.7071067812*I
```

  Multiple numerical values: the representation of complex numbers, matrices, or polynomials usually involves several numerical values. If a doctest with tolerance contains several numbers, each of them is checked individually:

```
sage: print "The sum of 1 and 1 equals 5"  # abs tol 1
The sum of 2 and 2 equals 4
sage: e^(i*pi/4).n() # rel tol 1e-1
0.7 + 0.7*I
sage: ((x+1.001)^4).expand() # rel tol 2
x^4 + 4*x^3 + 6*x^2 + 4*x + 1
sage: M = matrix.identity(3) + random_matrix(RR,3,3)/10^3
sage: M^2 # abs tol 1e-2
[1 0 0]
[0 1 0]
[0 0 1]
```

The values that the doctesting framework involves in the error computations are defined by the regular expression `float_regex` in `sage.doctest.parsing`.

- **not implemented** or **not tested:** The line is never tested.

  Use it for very long doctests that are only meant as documentation. It can also be used for todo notes of what will eventually be implemented:

  ```
  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.

Note: Skip all doctests of a file/directory

- **file:** If one of the first 10 lines of a file starts with any of `r""" nodoctest` (or `""" nodoctest` or `# nodoctest` or `% nodoctest` or `.. nodoctest`, or any of these with different spacing), then that file will be skipped.
- **directory:** If a directory contains a file `nodoctest.py`, then that whole directory will be skipped.

Neither of this applies to files or directories which are explicitly given as command line arguments: those are always tested.

- **optional:** A line flagged with `optional - keyword` is not tested unless the `--optional=keyword` flag is passed to `sage -t` (see *Run Optional Tests*). The main applications are:
  - **optional packages:** When a line requires an optional package to be installed (e.g. the `sloane_database` package):

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

  - **internet:** For lines that require an internet connection:

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

  - **bug:** For lines that describe bugs. Alternatively, use `# known bug` instead: it is an alias for `optional bug`.

    ```
    The following should yield 4.  See :trac:`2`. ::

        sage: 2+2  # optional: bug
        5
        sage: 2+2  # known bug
        5
    ```

Note:

- Any words after `# optional` are interpreted as a list of package names, separated by spaces.

  - Any punctuation (periods, commas, hyphens, semicolons, ...) after the first word ends the list of packages. Hyphens or colons between the word `optional` and the first package name are allowed. Therefore, you should not write `optional: needs package CHomP` but simply `optional: CHomP`.
  - Optional tags are case-insensitive, so you could also write `optional: chOMP`.

- **indirect doctest:** in the docstring of a function `A(...)`, a line calling `A` and in which the name `A` does not appear should have this flag. This prevents `sage --coverage <file>` from reporting the docstring as "not testing what it should test".

  Use it when testing special functions like `__repr__`, `__add__`, etc. Use it also when you test the function by calling `B` which internally calls `A`:

```
This is the docstring of an ``__add__`` method. The following
example tests it, but ``__add__`` is not written anywhere::

    sage: 1+1 # indirect doctest
    2
```

- **32-bit** or **64-bit:** for tests that behave differently on 32-bit or 64-bit machines. Note that this particular flag is to be applied on the **output** lines, not the input lines:

```
sage: hash(-92039082390482309489023849023848 4)
-873977844            # 32-bit
6874330978542788722   # 64-bit
```

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.

3.1.6 Running Automated Tests

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 *Running Sage's doctests* 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 ... ]
```

The Sage doctesting framework is based on the standard Python doctest module, but with many additional features (such as parallel testing, timeouts, optional tests). The Sage doctester recognizes `sage:` prompts as well as `>>>` prompts. It also preparses the doctests, just like in interactive Sage sessions.

Your file passes the tests if the code in it will run when entered at the `sage:` prompt with no extra 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.

For more information, see *Running Sage's doctests*.

Testing ReST Documentation

Run `sage -t <filename.rst>` to test the examples in verbatim environments in ReST documentation.

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:

```
EXAMPLES::

        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/src/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/src/doc/en/tutorial/` for many examples of how to include automated testing in ReST documentation for Sage.

3.1.7 The Pickle Jar

Sage maintains a pickle jar at `SAGE_ROOT/src/ext/pickle_jar/pickle_jar.tar.bz2` which is a tar file of "standard" pickles created by `sage`. This pickle jar is used to ensure that sage maintains backward compatibility by have having `sage.structure.sage_object.unpickle_all()` check that `sage` can always unpickle all of the pickles in the pickle jar as part of the standard doc testing framework.

Most people first become aware of the pickle_jar when their patch breaks the unpickling of one of the "standard" pickles in the pickle jar due to the failure of the doctest:

```
sage -t src/sage/structure/sage_object.pyx
```

When this happens an error message is printed which contains the following hints for fixing the uneatable pickle:

```
----------------------------------------------------------------------
** This error is probably due to an old pickle failing to unpickle.
** See sage.structure.sage_object.register_unpickle_override for
** how to override the default unpickling methods for (old) pickles.
** NOTE: pickles should never be removed from the pickle_jar!
----------------------------------------------------------------------
```

For more details about how to fix unpickling errors in the pickle jar see `sage.structure.sage_object.register_unpickle_override()`

Warning: Sage's pickle jar helps to ensure backward compatibility in sage. Pickles should **only** be removed from the pickle jar after the corresponding objects have been properly deprecated. Any proposal to remove pickles from the pickle jar should first be discussed on sage-devel.

3.1.8 Global Options

Global options for classes can be defined in Sage using `GlobalOptions`.

3.2 The reviewer's check list

All code that goes into Sage is peer-reviewed. Two reasons for this are:

- Because a developer cannot think of everything at once;
- Because a fresh pair of eyes may spot a mathematical error, a corner-case in the code, insufficient documentation, a missing consistency check, etc.

Anybody (e.g. you) can do this job for somebody else's ticket. This document lists things that the reviewer must check before deciding that a ticket is ready for inclusion into Sage.

- Do you know what the **trac server** is? If not, *click here*.
- Do you have a **trac account**? If not, *click here*.

You can now begin the review by reading the diff code.

Read the diff: the diff (i.e. the ticket's content) can be obtained by clicking on the (green) branch's name that appears on the trac ticket. If that name appears in red (see *The Ticket Fields*) you can say so in a comment and set the ticket to `needs_work` (see *The status of a ticket*).

Build the code: while you read the code, you can *rebuild Sage with the new code*. If you do not know how to **download the code**, *click here* (with git trac) or *here* (git only).

The following should generally be checked while reading and testing the code:

- **The purpose**: Does the code address the ticket's stated aim? Can it introduce any new problems? Does testing the new or fixed functionality with a variety of input, not just the examples in the documentation, give expected and robust output (and no unexpected errors or crashes)?
- **User documentation**: Is the use of the new code clear to a user? Are all mathematical notions involved standard, or is there explanation (or a link to one) provided? Can he/she find the new code easily if he/she needs it?
- **Code documentation**: Is the code sufficiently commented so that a developer does not have to wonder what exactly it does?
- **Conventions**: Does the code respect *Sage's conventions*? *Python's convention*? *Cython's convention*?
- **Doctest coverage**: Do all functions contain doctests? Use `sage -coverage <files>` to check it. Are all aspects of the new/modified methods and classes tested (see *Writing Testable Examples*)?
- **Bugfixes**: If the ticket contains a bugfix, does it add a doctest illustrating that the bug has been fixed? This new doctest should contain the ticket number, for example `See :trac:`12345``.
- **Speedup**: Can the ticket make any existing code slower? if the ticket claims to speed up some computation, does the ticket contain code examples to illustrate the claim? The ticket should explain how the speedup is achieved.
- **Manuals**: Does the reference manual build without errors (check both html and pdf)? See *The Sage Manuals* to learn how to build the manuals.
- **Run the tests**: Do all doctests pass without errors? Unrelated components of Sage may be affected by the change. Check all tests in the whole library, including "long" doctests (this can be done with `make ptestlong`) and any optional doctests related to the functionality. See *Running Sage's doctests* for more information.

You are now ready to change the ticket's status (see *The status of a ticket*):

- **positive review**: If the answers to the questions above and other reasonable questions are *"yes"*, you can set the ticket to `positive_review`. Add your full name to the "reviewer" field (see *The Ticket Fields*).
- **needs_work**: If something is not as it should, write a list of all points that need to be addressed in a comment and change the ticket's status to `needs_work`.
- **needs_info**: If something is not clear to you and prevents you from going further with the review, ask your question and set the ticket's status to `needs_info`.
- If you **do not know what to do**, for instance if you don't feel experienced enough to take a final decision, explain what you already did in a comment and ask if someone else could take a look.

Reviewer's commit: if you can fix the issues yourself, you may make a commit in your own name and mark the commit as a reviewer's patch. To learn how *click here* (git trac) or *here* (git only). This contribution must also be reviewed, for example by the author of the original patch.

For more advice on reviewing, see [WSblog].

Note: "The perfect is the enemy of the good"

The point of the review is to ensure that the Sage code guidelines are followed and that the the implementation is mathematically correct. Please refrain from additional feature requests or open-ended discussion about alternative implementations. If you want the patch written differently, your suggestion should be a clear and actionable request.

REFERENCES:

3.3 Running Sage's tests

3.3.1 Running Sage's doctests

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 6.0 from source and the top level Sage directory is:

```
[jdemeyer@sage sage-6.0]$ pwd
/scratch/jdemeyer/build/sage-6.0
```

See the section *Running Automated Tests* 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 (see *Optional Arguments* for more options). 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".

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-6.0]$ ./sage -t src/sage/games/sudoku.py
Running doctests with ID 2012-07-03-03-36-49-d82849c6.
Doctesting 1 file.
sage -t src/sage/games/sudoku.py
    [103 tests, 3.6 s]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
Total time for all tests: 4.8 seconds
    cpu time: 3.6 seconds
    cumulative wall time: 3.6 seconds
```

The numbers output by the test show that testing the sudoku module takes about four seconds, while testing all specified modules took the same amount of time; the total time required includes some startup time for the code that runs the tests. 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-6.0]$ ./sage -t src/sage/games/sudoku.py
Running doctests with ID 2012-07-03-03-39-02-da6accbb.
Doctesting 1 file.
sage -t src/sage/games/sudoku.py
    [103 tests, 3.6 s]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
Total time for all tests: 4.9 seconds
    cpu time: 3.6 seconds
    cumulative wall time: 3.6 seconds
```

but not:

```
[jdemeyer@sage sage-6.0]$ ./sage -t sage/games/sudoku.py
Running doctests with ID 2012-07-03-03-40-53-6cc4f29f.
No files matching sage/games/sudoku.py
No files to doctest
```

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

```
[jdemeyer@sage sage-6.0]$ cd src/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
Running doctests with ID 2012-07-03-03-41-39-95ebd2ff.
Doctesting 1 file.
sage -t sudoku.py
    [103 tests, 3.6 s]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
Total time for all tests: 5.2 seconds
    cpu time: 3.6 seconds
    cumulative wall time: 3.6 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.

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-6.0]$ ./sage -t src/sage/games/sudoku.py
Running doctests with ID 2012-07-03-03-43-24-a3449f54.
Doctesting 1 file.
sage -t src/sage/games/sudoku.py
    [103 tests, 3.6 s]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
Total time for all tests: 4.9 seconds
    cpu time: 3.6 seconds
    cumulative wall time: 3.6 seconds
[jdemeyer@sage sage-6.0]$ ./sage -t "src/sage/games/sudoku.py"
Running doctests with ID 2012-07-03-03-43-54-ac8ca007.
Doctesting 1 file.
sage -t src/sage/games/sudoku.py
    [103 tests, 3.6 s]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
Total time for all tests: 4.9 seconds
    cpu time: 3.6 seconds
    cumulative wall time: 3.6 seconds
```

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

```
[jdemeyer@sage sage-6.0]$ sage -t src/sage/games/sudoku.py
sage -t  "src/sage/games/sudoku.py"
**********************************************************************
File "/home/jdemeyer/sage/sage-6.0/src/sage/games/sudoku.py", line 515:
    sage: next(h.solve(algorithm='backtrack'))
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>
        next(h.solve(algorithm='backtrack'))###line 515:
    sage: next(h.solve(algorithm='backtrack'))
      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  "src/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.

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-6.0]$ ./sage -t src/sage/crypto
Running doctests with ID 2012-07-03-03-45-40-7f837dcf.
Doctesting 24 files.
sage -t src/sage/crypto/__init__.py
    [0 tests, 0.0 s]
sage -t src/sage/crypto/all.py
    [0 tests, 0.0 s]
sage -t src/sage/crypto/boolean_function.pyx
    [252 tests, 4.4 s]
sage -t src/sage/crypto/cipher.py
    [10 tests, 0.0 s]
sage -t src/sage/crypto/classical.py
    [718 tests, 11.3 s]
sage -t src/sage/crypto/classical_cipher.py
    [130 tests, 0.5 s]
sage -t src/sage/crypto/cryptosystem.py
    [82 tests, 0.1 s]
sage -t src/sage/crypto/lattice.py
    [1 tests, 0.0 s]
sage -t src/sage/crypto/lfsr.py
    [31 tests, 0.1 s]
sage -t src/sage/crypto/stream.py
    [17 tests, 0.1 s]
sage -t src/sage/crypto/stream_cipher.py
    [114 tests, 0.2 s]
sage -t src/sage/crypto/util.py
    [122 tests, 0.2 s]
sage -t src/sage/crypto/block_cipher/__init__.py
    [0 tests, 0.0 s]
sage -t src/sage/crypto/block_cipher/all.py
```

```
    [0 tests, 0.0 s]
sage -t src/sage/crypto/block_cipher/miniaes.py
    [430 tests, 1.3 s]
sage -t src/sage/crypto/block_cipher/sdes.py
    [290 tests, 0.9 s]
sage -t src/sage/crypto/mq/__init__.py
    [0 tests, 0.0 s]
sage -t src/sage/crypto/mq/mpolynomialsystem.py
    [320 tests, 9.1 s]
sage -t src/sage/crypto/mq/mpolynomialsystemgenerator.py
    [42 tests, 0.1 s]
sage -t src/sage/crypto/mq/sbox.py
    [124 tests, 0.8 s]
sage -t src/sage/crypto/mq/sr.py
    [435 tests, 5.5 s]
sage -t src/sage/crypto/public_key/__init__.py
    [0 tests, 0.0 s]
sage -t src/sage/crypto/public_key/all.py
    [0 tests, 0.0 s]
sage -t src/sage/crypto/public_key/blum_goldwasser.py
    [135 tests, 0.2 s]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
Total time for all tests: 38.1 seconds
    cpu time: 29.8 seconds
    cumulative wall time: 35.1 seconds
```

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

```
[jdemeyer@sage sage-6.0]$ ./sage -t --long src/sage/crypto/
Running doctests with ID 2012-07-03-03-48-11-c16721e6.
Doctesting 24 files.
sage -t --long src/sage/crypto/__init__.py
    [0 tests, 0.0 s]
sage -t --long src/sage/crypto/all.py
    [0 tests, 0.0 s]
sage -t --long src/sage/crypto/boolean_function.pyx
    [252 tests, 4.2 s]
sage -t --long src/sage/crypto/cipher.py
    [10 tests, 0.0 s]
sage -t --long src/sage/crypto/classical.py
    [718 tests, 10.3 s]
sage -t --long src/sage/crypto/classical_cipher.py
    [130 tests, 0.5 s]
sage -t --long src/sage/crypto/cryptosystem.py
    [82 tests, 0.1 s]
sage -t --long src/sage/crypto/lattice.py
    [1 tests, 0.0 s]
sage -t --long src/sage/crypto/lfsr.py
    [31 tests, 0.1 s]
sage -t --long src/sage/crypto/stream.py
    [17 tests, 0.1 s]
sage -t --long src/sage/crypto/stream_cipher.py
    [114 tests, 0.2 s]
sage -t --long src/sage/crypto/util.py
    [122 tests, 0.2 s]
sage -t --long src/sage/crypto/block_cipher/__init__.py
```

```
    [0 tests, 0.0 s]
sage -t --long src/sage/crypto/block_cipher/all.py
    [0 tests, 0.0 s]
sage -t --long src/sage/crypto/block_cipher/miniaes.py
    [430 tests, 1.1 s]
sage -t --long src/sage/crypto/block_cipher/sdes.py
    [290 tests, 0.7 s]
sage -t --long src/sage/crypto/mq/__init__.py
    [0 tests, 0.0 s]
sage -t --long src/sage/crypto/mq/mpolynomialsystem.py
    [320 tests, 7.5 s]
sage -t --long src/sage/crypto/mq/mpolynomialsystemgenerator.py
    [42 tests, 0.1 s]
sage -t --long src/sage/crypto/mq/sbox.py
    [124 tests, 0.7 s]
sage -t --long src/sage/crypto/mq/sr.py
    [437 tests, 82.4 s]
sage -t --long src/sage/crypto/public_key/__init__.py
    [0 tests, 0.0 s]
sage -t --long src/sage/crypto/public_key/all.py
    [0 tests, 0.0 s]
sage -t --long src/sage/crypto/public_key/blum_goldwasser.py
    [135 tests, 0.2 s]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
Total time for all tests: 111.8 seconds
    cpu time: 106.1 seconds
    cumulative wall time: 108.5 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 five seconds. With this optional argument, it required 82 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-6.0]$ ./sage -tp 4 src/sage/crypto/
Running doctests with ID 2012-07-07-00-11-55-9b17765e.
Sorting sources by runtime so that slower doctests are run first....
Doctesting 24 files using 4 threads.
sage -t src/sage/crypto/boolean_function.pyx
    [252 tests, 3.8 s]
sage -t src/sage/crypto/block_cipher/miniaes.py
    [429 tests, 1.1 s]
sage -t src/sage/crypto/mq/sr.py
    [432 tests, 5.7 s]
sage -t src/sage/crypto/mq/sbox.py
    [123 tests, 0.8 s]
sage -t src/sage/crypto/block_cipher/sdes.py
    [289 tests, 0.6 s]
sage -t src/sage/crypto/classical_cipher.py
    [123 tests, 0.4 s]
sage -t src/sage/crypto/stream_cipher.py
    [113 tests, 0.1 s]
sage -t src/sage/crypto/public_key/blum_goldwasser.py
    [134 tests, 0.1 s]
sage -t src/sage/crypto/lfsr.py
    [30 tests, 0.1 s]
sage -t src/sage/crypto/util.py
    [121 tests, 0.1 s]
sage -t src/sage/crypto/cryptosystem.py
    [79 tests, 0.0 s]
sage -t src/sage/crypto/stream.py
    [12 tests, 0.0 s]
sage -t src/sage/crypto/mq/mpolynomialsystemgenerator.py
    [40 tests, 0.0 s]
sage -t src/sage/crypto/cipher.py
    [3 tests, 0.0 s]
sage -t src/sage/crypto/lattice.py
    [0 tests, 0.0 s]
sage -t src/sage/crypto/block_cipher/__init__.py
    [0 tests, 0.0 s]
sage -t src/sage/crypto/all.py
    [0 tests, 0.0 s]
sage -t src/sage/crypto/public_key/__init__.py
    [0 tests, 0.0 s]
sage -t src/sage/crypto/__init__.py
    [0 tests, 0.0 s]
sage -t src/sage/crypto/public_key/all.py
    [0 tests, 0.0 s]
sage -t src/sage/crypto/mq/__init__.py
    [0 tests, 0.0 s]
sage -t src/sage/crypto/block_cipher/all.py
    [0 tests, 0.0 s]
sage -t src/sage/crypto/mq/mpolynomialsystem.py
    [318 tests, 8.4 s]
sage -t src/sage/crypto/classical.py
    [717 tests, 10.4 s]
----------------------------------------------------------------------
All tests passed!
```

```
----------------------------------------------------------------------
Total time for all tests: 12.9 seconds
    cpu time: 30.5 seconds
    cumulative wall time: 31.7 seconds
[jdemeyer@sage sage-6.0]$ ./sage -tp 4 --long src/sage/crypto/
Running doctests with ID 2012-07-07-00-13-04-d71f3cd4.
Sorting sources by runtime so that slower doctests are run first....
Doctesting 24 files using 4 threads.
sage -t --long src/sage/crypto/boolean_function.pyx
    [252 tests, 3.7 s]
sage -t --long src/sage/crypto/block_cipher/miniaes.py
    [429 tests, 1.0 s]
sage -t --long src/sage/crypto/mq/sbox.py
    [123 tests, 0.8 s]
sage -t --long src/sage/crypto/block_cipher/sdes.py
    [289 tests, 0.6 s]
sage -t --long src/sage/crypto/classical_cipher.py
    [123 tests, 0.4 s]
sage -t --long src/sage/crypto/util.py
    [121 tests, 0.1 s]
sage -t --long src/sage/crypto/stream_cipher.py
    [113 tests, 0.1 s]
sage -t --long src/sage/crypto/public_key/blum_goldwasser.py
    [134 tests, 0.1 s]
sage -t --long src/sage/crypto/lfsr.py
    [30 tests, 0.0 s]
sage -t --long src/sage/crypto/cryptosystem.py
    [79 tests, 0.0 s]
sage -t --long src/sage/crypto/stream.py
    [12 tests, 0.0 s]
sage -t --long src/sage/crypto/mq/mpolynomialsystemgenerator.py
    [40 tests, 0.0 s]
sage -t --long src/sage/crypto/cipher.py
    [3 tests, 0.0 s]
sage -t --long src/sage/crypto/lattice.py
    [0 tests, 0.0 s]
sage -t --long src/sage/crypto/block_cipher/all.py
    [0 tests, 0.0 s]
sage -t --long src/sage/crypto/public_key/__init__.py
    [0 tests, 0.0 s]
sage -t --long src/sage/crypto/mq/__init__.py
    [0 tests, 0.0 s]
sage -t --long src/sage/crypto/all.py
    [0 tests, 0.0 s]
sage -t --long src/sage/crypto/block_cipher/__init__.py
    [0 tests, 0.0 s]
sage -t --long src/sage/crypto/__init__.py
    [0 tests, 0.0 s]
sage -t --long src/sage/crypto/public_key/all.py
    [0 tests, 0.0 s]
sage -t --long src/sage/crypto/mq/mpolynomialsystem.py
    [318 tests, 9.0 s]
sage -t --long src/sage/crypto/classical.py
    [717 tests, 10.5 s]
sage -t --long src/sage/crypto/mq/sr.py
    [434 tests, 88.0 s]
----------------------------------------------------------------------
All tests passed!
```

```
----------------------------------------------------------------------
Total time for all tests: 90.4 seconds
    cpu time: 113.4 seconds
    cumulative wall time: 114.5 seconds
```

As the number of threads increases, the total testing time decreases.

Parallel Testing the Whole Sage Library

The main Sage library resides in the directory `SAGE_ROOT/src/`. 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-6.0]$ ./sage -tp 10 --long src/
```

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-6.0]$ 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
```

The differences are:

- `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/src/doc/common`
  - `SAGE_ROOT/src/doc/en`
  - `SAGE_ROOT/src/doc/fr`

  Finally, the commands doctest the Sage library. For more details on these command, see the file `SAGE_ROOT/Makefile`.
- `make testlong` — This command doctests the standard documentation:
  - `SAGE_ROOT/src/doc/common`
  - `SAGE_ROOT/src/doc/en`
  - `SAGE_ROOT/src/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.

Beyond the Sage Library

Doctesting also works fine for files not in the Sage library. For example, suppose we have a Python script called `my_python_script.py`:

```
[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
```

Then we can doctest it just as with Sage library files:

```
[mvngu@sage sage-6.0]$ ./sage -t my_python_script.py
Running doctests with ID 2012-07-07-00-17-56-d056f7c0.
Doctesting 1 file.
sage -t my_python_script.py
    [1 test, 0.0 s]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
Total time for all tests: 2.2 seconds
    cpu time: 0.0 seconds
    cumulative wall time: 0.0 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 sage-6.0]$ cat my_sage_script.sage
def cube(n):
    r"""
    Return the cube of n.

    EXAMPLES::

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

Then we can doctest it just as for Python files:

```
[mvngu@sage build]$ sage-6.0/sage -t my_sage_script.sage
Running doctests with ID 2012-07-07-00-20-06-82ee728c.
Doctesting 1 file.
sage -t my_sage_script.sage
```

```
    [1 test, 0.0 s]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
Total time for all tests: 2.5 seconds
    cpu time: 0.0 seconds
    cumulative wall time: 0.0 seconds
```

Alternatively, we can preparse it to convert it to a Python script, and then doctest that:

```
[mvngu@sage build]$ sage-6.0/sage --preparse my_sage_script.sage
[mvngu@sage build]$ cat my_sage_script.sage.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
[mvngu@sage build]$ sage-6.0/sage -t my_sage_script.sage.py
Running doctests with ID 2012-07-07-00-26-46-2bb00911.
Doctesting 1 file.
sage -t my_sage_script.sage.py
    [2 tests, 0.0 s]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
Total time for all tests: 2.3 seconds
    cpu time: 0.0 seconds
    cumulative wall time: 0.0 seconds
```

Doctesting from Within Sage

You can run doctests from within Sage, which can be useful since you don't have to wait for Sage to start. Use the `run_doctests` function in the global namespace, passing it either a string or a module:

```
sage: run_doctests(sage.coding.sd_codes)
Doctesting /Users/roed/sage/sage-5.3/src/sage/coding/sd_codes.py
Running doctests with ID 2012-07-07-04-32-36-81f3853b.
Doctesting 1 file.
sage -t /Users/roed/sage/sage-5.3/src/sage/coding/sd_codes.py
    [18 tests, 0.3 s]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
Total time for all tests: 0.4 seconds
    cpu time: 0.2 seconds
    cumulative wall time: 0.3 seconds
```

Optional Arguments

Run Long Tests

Ideally, doctests should not take any noticeable amount of time. If you really need longer-running doctests (anything beyond about one second) then you should mark them as:

```
sage: my_long_test()  # long time
```

Even then, long doctests should ideally complete in 5 seconds or less. We know that you (the author) want to show off the capabilities of your code, but this is not the place to do so. Long-running tests will sooner or later hurt our ability to run the testsuite. Really, doctests should be as fast as possible while providing coverage for the code.

Use the `--long` flag to run doctests that have been marked with the comment `# long time`. These tests are normally skipped in order to reduce the time spent running tests:

```
[roed@sage sage-6.0]$ sage -t src/sage/rings/tests.py
Running doctests with ID 2012-06-21-16-00-13-40835825.
Doctesting 1 file.
sage -t tests.py
    [18 tests, 1.1 s]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
Total time for all tests: 2.9 seconds
    cpu time: 0.9 seconds
    cumulative wall time: 1.1 seconds
```

In order to run the long tests as well, do the following:

```
[roed@sage sage-6.0]$ sage -t --long src/sage/rings/tests.py
Running doctests with ID 2012-06-21-16-02-05-d13a9a24.
Doctesting 1 file.
sage -t tests.py
    [20 tests, 34.7 s]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
Total time for all tests: 46.5 seconds
    cpu time: 25.2 seconds
    cumulative wall time: 34.7 seconds
```

To find tests that take longer than the allowed time use the `--warn-long` flag. Without any options it will cause tests to print a warning if they take longer than 1.0 second. Note that this is a warning, not an error:

```
[roed@sage sage-6.0]$ sage -t --warn-long src/sage/rings/factorint.pyx
Running doctests with ID 2012-07-14-03-27-03-2c952ac1.
Doctesting 1 file.
sage -t --warn-long src/sage/rings/factorint.pyx
**********************************************************************
File "src/sage/rings/factorint.pyx", line 125, in sage.rings.factorint.base_exponent
Failed example:
    base_exponent(-4)
Test ran for 4.09 s
**********************************************************************
File "src/sage/rings/factorint.pyx", line 153, in sage.rings.factorint.factor_aurifeuillian
Failed example:
    fa(2^6+1)
```

```
Test ran for 2.22 s
**********************************************************************
File "src/sage/rings/factorint.pyx", line 155, in sage.rings.factorint.factor_aurifeuillian
Failed example:
    fa(2^58+1)
Test ran for 2.22 s
**********************************************************************
File "src/sage/rings/factorint.pyx", line 163, in sage.rings.factorint.factor_aurifeuillian
Failed example:
    fa(2^4+1)
Test ran for 2.25 s
**********************************************************************
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
Total time for all tests: 16.1 seconds
    cpu time: 9.7 seconds
    cumulative wall time: 10.9 seconds
```

You can also pass in an explicit amount of time:

```
[roed@sage sage-6.0]$ sage -t --long --warn-long 2.0 src/sage/rings/tests.py
Running doctests with ID 2012-07-14-03-30-13-c9164c9d.
Doctesting 1 file.
sage -t --long --warn-long 2.0 tests.py
**********************************************************************
File "tests.py", line 240, in sage.rings.tests.test_random_elements
Failed example:
    sage.rings.tests.test_random_elements(trials=1000)  # long time (5 seconds)
Test ran for 13.36 s
**********************************************************************
File "tests.py", line 283, in sage.rings.tests.test_random_arith
Failed example:
    sage.rings.tests.test_random_arith(trials=1000)   # long time (5 seconds?)
Test ran for 12.42 s
**********************************************************************
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
Total time for all tests: 27.6 seconds
    cpu time: 24.8 seconds
    cumulative wall time: 26.3 seconds
```

Finally, you can disable any warnings about long tests with `--warn-long 0`.

Run Optional Tests

You can run tests that require optional packages by using the `--optional` flag. Obviously, you need to have installed the necessary optional packages in order for these tests to succeed. See http://www.sagemath.org/packages/optional/ in order to download optional packages.

By default, Sage only runs doctests that are not marked with the `optional` tag. This is equivalent to running

```
[roed@sage sage-6.0]$ sage -t --optional=sage src/sage/rings/real_mpfr.pyx
Running doctests with ID 2012-06-21-16-18-30-a368a200.
Doctesting 1 file.
sage -t src/sage/rings/real_mpfr.pyx
```

```
    [819 tests, 7.0 s]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
Total time for all tests: 8.4 seconds
    cpu time: 4.1 seconds
    cumulative wall time: 7.0 seconds
```

If you want to also run tests that require magma, you can do the following:

```
[roed@sage sage-6.0]$ sage -t --optional=sage,magma src/sage/rings/real_mpfr.pyx
Running doctests with ID 2012-06-21-16-18-30-a00a7319
Doctesting 1 file.
sage -t src/sage/rings/real_mpfr.pyx
    [823 tests, 8.4 s]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
Total time for all tests: 9.6 seconds
    cpu time: 4.0 seconds
    cumulative wall time: 8.4 seconds
```

In order to just run the tests that are marked as requiring magma, omit `sage`:

```
[roed@sage sage-6.0]$ sage -t --optional=magma src/sage/rings/real_mpfr.pyx
Running doctests with ID 2012-06-21-16-18-33-a2bc1fdf
Doctesting 1 file.
sage -t src/sage/rings/real_mpfr.pyx
    [4 tests, 2.0 s]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
Total time for all tests: 3.2 seconds
    cpu time: 0.1 seconds
    cumulative wall time: 2.0 seconds
```

To run all tests, regardless of whether they are marked optional, pass `all` as the `optional` tag:

```
[roed@sage sage-6.0]$ sage -t --optional=all src/sage/rings/real_mpfr.pyx
Running doctests with ID 2012-06-21-16-31-18-8c097f55
Doctesting 1 file.
sage -t src/sage/rings/real_mpfr.pyx
    [865 tests, 11.2 s]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
Total time for all tests: 12.8 seconds
    cpu time: 4.7 seconds
    cumulative wall time: 11.2 seconds
```

Running Tests in Parallel

If you're testing many files, you can get big speedups by using more than one thread. To run doctests in parallel use the `--nthreads` flag (`-p` is a shortened version). Pass in the number of threads you would like to use (by default Sage just uses 1):

```
[roed@sage sage-6.0]$ sage -tp 2 src/sage/doctest/
Running doctests with ID 2012-06-22-19-09-25-a3afdb8c.
Sorting sources by runtime so that slower doctests are run first....
Doctesting 8 files using 2 threads.
sage -t src/sage/doctest/control.py
    [114 tests, 4.6 s]
sage -t src/sage/doctest/util.py
    [114 tests, 0.6 s]
sage -t src/sage/doctest/parsing.py
    [187 tests, 0.5 s]
sage -t src/sage/doctest/sources.py
    [128 tests, 0.1 s]
sage -t src/sage/doctest/reporting.py
    [53 tests, 0.1 s]
sage -t src/sage/doctest/all.py
    [0 tests, 0.0 s]
sage -t src/sage/doctest/__init__.py
    [0 tests, 0.0 s]
sage -t src/sage/doctest/forker.py
    [322 tests, 15.5 s]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
Total time for all tests: 17.0 seconds
    cpu time: 4.2 seconds
    cumulative wall time: 21.5 seconds
```

Doctesting All of Sage

To doctest the whole Sage library use the `--all` flag (`-a` for short). In addition to testing the code in Sage's Python and Cython files, this command will run the tests defined in Sage's documentation as well as testing the Sage notebook:

```
[roed@sage sage-6.0]$ sage -t -a
Running doctests with ID 2012-06-22-19-10-27-e26fce6d.
Doctesting entire Sage library.
Sorting sources by runtime so that slower doctests are run first....
Doctesting 2020 files.
sage -t /Users/roed/sage/sage-5.3/src/sage/plot/plot.py
    [304 tests, 69.0 s]
...
```

If you want to just run the notebook tests, use the `--sagenb` flag instead.

Debugging Tools

Sometimes doctests fail (that's why we run them after all). There are various flags to help when something goes wrong. If a doctest produces a Python error, then normally tests continue after reporting that an error occurred. If you use the flag `--debug` (`-d` for short) then you will drop into an interactive Python debugger whenever a Python exception occurs. As an example, I modified `sage.schemes.elliptic_curves.constructor` to produce an error:

```
[roed@sage sage-6.0]$ sage -t --debug src/sage/schemes/elliptic_curves/constructor.py
Running doctests with ID 2012-06-23-12-09-04-b6352629.
Doctesting 1 file.
**********************************************************************
File "sage.schemes.elliptic_curves.constructor", line 4, in sage.schemes.elliptic_curves.constructor
```

```
Failed example:
    EllipticCurve([0,0])
Exception raised:
    Traceback (most recent call last):
      File "/Users/roed/sage/sage-5.3/local/lib/python2.7/site-packages/sage/doctest/forker.py", line
        self.execute(example, compiled, test.globs)
      File "/Users/roed/sage/sage-5.3/local/lib/python2.7/site-packages/sage/doctest/forker.py", line
        exec compiled in globs
      File "<doctest sage.schemes.elliptic_curves.constructor[0]>", line 1, in <module>
        EllipticCurve([Integer(0),Integer(0)])
      File "/Users/roed/sage/sage-5.3/local/lib/python2.7/site-packages/sage/schemes/elliptic_curves/
        return ell_rational_field.EllipticCurve_rational_field(x, y)
      File "/Users/roed/sage/sage-5.3/local/lib/python2.7/site-packages/sage/schemes/elliptic_curves/
        EllipticCurve_number_field.__init__(self, Q, ainvs)
      File "/Users/roed/sage/sage-5.3/local/lib/python2.7/site-packages/sage/schemes/elliptic_curves/
        EllipticCurve_field.__init__(self, [field(x) for x in ainvs])
      File "/Users/roed/sage/sage-5.3/local/lib/python2.7/site-packages/sage/schemes/elliptic_curves/
        "Invariants %s define a singular curve."%ainvs
    ArithmeticError: Invariants [0, 0, 0, 0, 0] define a singular curve.
> /Users/roed/sage/sage-5.3/local/lib/python2.7/site-packages/sage/schemes/elliptic_curves/ell_generi
-> "Invariants %s define a singular curve."%ainvs
(Pdb) l
151                 if len(ainvs) == 2:
152                     ainvs = [K(0),K(0),K(0)] + ainvs
153                 self.__ainvs = tuple(ainvs)
154                 if self.discriminant() == 0:
155                     raise ArithmeticError, \
156  ->                       "Invariants %s define a singular curve."%ainvs
157                 PP = projective_space.ProjectiveSpace(2, K, names='xyz');
158                 x, y, z = PP.coordinate_ring().gens()
159                 a1, a2, a3, a4, a6 = ainvs
160                 f = y**2*z + (a1*x + a3*z)*y*z \
161                     - (x**3 + a2*x**2*z + a4*x*z**2 + a6*z**3)
(Pdb) p ainvs
[0, 0, 0, 0, 0]
(Pdb) quit
**********************************************************************
1 items had failures:
   1 of   1 in sage.schemes.elliptic_curves.constructor
***Test Failed*** 1 failures.
sage -t src/sage/schemes/elliptic_curves/constructor.py
    [64 tests, 89.2 s]
----------------------------------------------------------------------
sage -t src/sage/schemes/elliptic_curves/constructor.py # 1 doctest failed
----------------------------------------------------------------------
Total time for all tests: 90.4 seconds
    cpu time: 4.5 seconds
    cumulative wall time: 89.2 seconds
```

Sometimes an error might be so severe that it causes Sage to segfault or hang. In such a situation you have a number of options. The doctest framework will print out the output so far, so that at least you know what test caused the problem (if you want this output to appear in real time use the `--verbose` flag). To have doctests run under the control of gdb, use the `--gdb` flag:

```
[roed@sage sage-6.0]$ sage -t --gdb src/sage/schemes/elliptic_curves/constructor.py
gdb -x /home/roed/sage-6.0.b5/local/bin/sage-gdb-commands --args python /home/roed/sage-6.0.b5/local/
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
```

```
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu"...
[Thread debugging using libthread_db enabled]
[New Thread 0x7f10f85566e0 (LWP 6534)]
Running doctests with ID 2012-07-07-00-43-36-b1b735e7.
Doctesting 1 file.
sage -t src/sage/schemes/elliptic_curves/constructor.py
    [67 tests, 5.8 s]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
Total time for all tests: 15.7 seconds
    cpu time: 4.4 seconds
    cumulative wall time: 5.8 seconds

Program exited normally.
(gdb) quit
```

Sage also includes valgrind, and you can run doctests under various valgrind tools to track down memory issues: the relevant flags are `--valgrind` (or `--memcheck`), `--massif`, `--cachegrind` and `--omega`. See http://wiki.sagemath.org/ValgrindingSage for more details.

Once you're done fixing whatever problems where revealed by the doctests, you can rerun just those files that failed their most recent test by using the `--failed` flag (`-f` for short):

```
[roed@sage sage-6.0]$ sage -t -fa
Running doctests with ID 2012-07-07-00-45-35-d8b5a408.
Doctesting entire Sage library.
Only doctesting files that failed last test.
No files to doctest
```

Miscellaneous Options

There are various other options that change the behavior of Sage's doctesting code.

Show only first failure The first failure in a file often causes a cascade of others, as NameErrors arise from variables that weren't defined and tests fail because old values of variables are used. To only see the first failure in each doctest block use the `--initial` flag (`-i` for short).

Show skipped optional tests To print a summary at the end of each file with the number of optional tests skipped, use the `--show-skipped` flag:

```
[roed@sage sage-6.0]$ sage -t --show-skipped src/sage/rings/finite_rings/integer_mod.pyx
Running doctests with ID 2013-03-14-15-32-05-8136f5e3.
Doctesting 1 file.
sage -t sage/rings/finite_rings/integer_mod.pyx
    2 axiom tests not run
    1 cunningham test not run
    2 fricas tests not run
    1 long test not run
    3 magma tests not run
```

```
    [440 tests, 4.0 s]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
Total time for all tests: 4.3 seconds
    cpu time: 2.4 seconds
    cumulative wall time: 4.0 seconds
```

Running tests with iterations Sometimes tests fail intermittently. There are two options that allow you to run tests repeatedly in an attempt to search for Heisenbugs. The flag `--global-iterations` takes an integer and runs the whole set of tests that many times serially:

```
[roed@sage sage-6.0]$ sage -t --global-iterations 2 src/sage/sandpiles
Running doctests with ID 2012-07-07-00-59-28-e7048ad9.
Doctesting 3 files (2 global iterations).
sage -t src/sage/sandpiles/__init__.py
    [0 tests, 0.0 s]
sage -t src/sage/sandpiles/all.py
    [0 tests, 0.0 s]
sage -t src/sage/sandpiles/sandpile.py
    [711 tests, 14.7 s]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
Total time for all tests: 17.6 seconds
    cpu time: 13.2 seconds
    cumulative wall time: 14.7 seconds
sage -t src/sage/sandpiles/__init__.py
    [0 tests, 0.0 s]
sage -t src/sage/sandpiles/all.py
    [0 tests, 0.0 s]
sage -t src/sage/sandpiles/sandpile.py
    [711 tests, 13.8 s]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
Total time for all tests: 14.3 seconds
    cpu time: 26.4 seconds
    cumulative wall time: 28.5 seconds
```

You can also iterate in a different order: the `--file-iterations` flag runs the tests in each file `N` times before proceeding:

```
[roed@sage sage-6.0]$ sage -t --file-iterations 2 src/sage/sandpiles
Running doctests with ID 2012-07-07-01-01-43-8f954206.
Doctesting 3 files (2 file iterations).
sage -t src/sage/sandpiles/__init__.py
    [0 tests, 0.0 s]
sage -t src/sage/sandpiles/all.py
    [0 tests, 0.0 s]
sage -t src/sage/sandpiles/sandpile.py
    [1422 tests, 13.3 s]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
Total time for all tests: 29.6 seconds
    cpu time: 12.7 seconds
```

```
    cumulative wall time: 13.3 seconds
```

Note that the reported results are the average time for all tests in that file to finish. If a failure in a file occurs, then the failure is reported and testing proceeds with the next file.

Using a different timeout On a slow machine the default timeout of 5 minutes may not be enough for the slowest files. Use the `--timeout` flag (`-T` for short) to set it to something else:

```
[roed@sage sage-6.0]$ sage -tp 2 --all --timeout 1
Running doctests with ID 2012-07-07-01-09-37-deb1ab83.
Doctesting entire Sage library.
Sorting sources by runtime so that slower doctests are run first....
Doctesting 2067 files using 2 threads.
sage -t src/sage/schemes/elliptic_curves/ell_rational_field.py
    Timed out!
...
```

Using absolute paths By default filenames are printed using relative paths. To use absolute paths instead pass in the `--abspath` flag:

```
[roed@sage sage-6.0]$ sage -t --abspath src/sage/doctest/control.py
Running doctests with ID 2012-07-07-01-13-03-a023e212.
Doctesting 1 file.
sage -t /home/roed/sage-6.0/src/sage/doctest/control.py
    [133 tests, 4.7 s]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
Total time for all tests: 7.1 seconds
    cpu time: 0.2 seconds
    cumulative wall time: 4.7 seconds
```

Testing changed files If you are working on some files in the Sage library it can be convenient to test only the files that have changed. To do so use the `--new` flag, which tests files that have been modified or added since the last commit:

```
[roed@sage sage-6.0]$ sage -t --new
Running doctests with ID 2012-07-07-01-15-52-645620ee.
Doctesting files changed since last git commit.
Doctesting 1 file.
sage -t src/sage/doctest/control.py
    [133 tests, 3.7 s]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
Total time for all tests: 3.8 seconds
    cpu time: 0.1 seconds
    cumulative wall time: 3.7 seconds
```

Running tests in a random order By default, tests are run in the order in which they appear in the file. To run tests in a random order (which can reveal subtle bugs), use the `--randorder` flag and pass in a random seed:

```
[roed@sage sage-6.0]$ sage -t --new --randorder 127
Running doctests with ID 2012-07-07-01-19-06-97c8484e.
Doctesting files changed since last git commit.
Doctesting 1 file.
sage -t src/sage/doctest/control.py
    [133 tests, 3.6 s]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
Total time for all tests: 3.7 seconds
    cpu time: 0.2 seconds
    cumulative wall time: 3.6 seconds
```

Note that even with this option, the tests within a given doctest block are still run in order.

Testing external files When testing a file that's not part of the Sage library, the testing code loads the globals from that file into the namespace before running tests. To model the behavior used on the Sage library instead (where imports must be explicitly specified), use the `--force-lib` flag.

Auxilliary files To specify a logfile (rather than use the default which is created for `sage -t --all`), use the `--logfile` flag:

```
[roed@sage sage-6.0]$ sage -t --logfile test1.log src/sage/doctest/control.py
Running doctests with ID 2012-07-07-01-25-49-e7c0e52d.
Doctesting 1 file.
sage -t src/sage/doctest/control.py
    [133 tests, 4.3 s]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
Total time for all tests: 6.7 seconds
    cpu time: 0.1 seconds
    cumulative wall time: 4.3 seconds
[roed@sage sage-6.0]$ cat test1.log
Running doctests with ID 2012-07-07-01-25-49-e7c0e52d.
Doctesting 1 file.
sage -t src/sage/doctest/control.py
    [133 tests, 4.3 s]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
Total time for all tests: 6.7 seconds
    cpu time: 0.1 seconds
    cumulative wall time: 4.3 seconds
```

To give a json file storing the timings for each file, use the `--stats_path` flag. These statistics are used in sorting files so that slower tests are run first (and thus multiple processes are utilized most efficiently):

```
[roed@sage sage-6.0]$ sage -tp 2 --stats-path ~/.sage/timings2.json --all
Running doctests with ID 2012-07-07-01-28-34-2df4251d.
Doctesting entire Sage library.
Sorting sources by runtime so that slower doctests are run first....
Doctesting 2067 files using 2 threads.
...
```

3.4 Contributing to Manuals and Tutorials

3.4.1 The Sage Manuals

Sage's manuals are written in ReST (reStructuredText), and generated with the software Sphinx:

Name	Files
Tutorial	`SAGE_ROOT/src/doc/en/tutorial`
Developer's guide	`SAGE_ROOT/src/doc/en/developer`
Constructions	`SAGE_ROOT/src/doc/en/constructions`
Installation guide	`SAGE_ROOT/src/doc/en/installation`
Reference manual	`SAGE_ROOT/src/doc/en/reference` (most of it is generated from the source code)

- Additionally, more specialized manuals can be found under `SAGE_ROOT/src/doc/en`.
- Some documents have been **translated** into other languages. In order to access them, change en/ into fr/,es/, de/... See *Document Names*.

Editing the documentation

(*Do you want to convert a Sage worksheet into documentation?* Click here)

After modifying some files in the Sage tutorial (`SAGE_ROOT/src/doc/en/tutorial/`), you will want to visualize the result. In order to build a **html** version of this document, type:

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

You can now open `SAGE_ROOT/src/doc/output/html/en/tutorial/index.html` in your web browser.

- Do you want to **add a new file** to the documentation? *Click here*.
- For more detailed information on the `--docbuild` command, see *Building the Manuals*.

Run doctests: All files must pass tests. After modifying a document (e.g. `tutorial`), you can run tests with the following command (see *Running Automated Tests*):

```
sage -tp SAGE_ROOT/src/doc/en/tutorial/
```

Reference manual: as this manual is mostly generated from Sage's source code, you will need to build Sage in order to see the changes you made to some function's documentation. Type:

```
sage -b && sage --docbuild reference html
```

Hyperlinks

The documentation can contain links toward modules, classes, or methods, e.g.:

```
:mod:`link to a module <sage.module_name>`
:mod:`sage.module_name` (here the link's text is the module's name)
```

For links toward classes, methods, or function, replace **:mod:** by **:class:**, **:meth:** or **func:** respectively. See Sphinx' documentation.

Short links: the link `:func:`~sage.mod1.mod2.mod3.func1`` is equivalent to `:func:`func1 <sage.mod1.mod2.mod3.func1>``: the function's name will be used as the link name, instead of its full path.

Local names: links between methods of the same class do not need to be absolute. If you are documenting `method_one`, you can write `:meth:`method_two``.

Global namespace: if an object (e.g. `integral`) is automatically imported by Sage, you can link toward it without specifying its full path:

```
:func:`A link toward the integral function <integral>`
```

Sage-specific roles: Sage defines several specific *roles*:

Trac server	`:trac:`17596``	trac ticket #17596
Wikipedia	`:wikipedia:`Sage_(mathematics_software)``	Wikipedia article Sage_(mathematics_software)
Arxiv	`:arxiv:`1202.1506``	Arxiv 1202.1506
On-Line Encyclopedia of Integer Sequences	`:oeis:`A000081``	OEIS sequence A000081
Digital Object Identifier	`:doi:`10.2752/175303708X390473``	doi:10.2752/175303708X390473
MathSciNet	`:mathscinet:`MR0100971``	MathSciNet MR0100971

http links: copy/pasting a http link in the documentation works. If you want a specific link name, use `` `link name <http://www.example.com>`_ ``

Broken links: Sphinx can report broken links. See *Building the Manuals*.

Adding a New File

If you added a new file to Sage (e.g. `sage/matroids/my_algorithm.py`) and you want its content to appear in the reference manual, you have to add its name to the file `SAGE_ROOT/src/doc/en/reference/matroids/index.rst`. Replace 'matroids' with whatever fits your case.

The combinat/ folder: if your new file belongs to a subdirectory of combinat/ the procedure is different:

- Add your file to the index stored in the `__init__.py` file located in the directory that contains your file.
- Add your file to the index contained in `SAGE_ROOT/src/doc/en/reference/combinat/module_list.rst`.

Building the Manuals

(Do you want to edit the documentation? Click here)

All of the Sage manuals are built using the `sage --docbuild` script. The content of the `sage --docbuild` script is defined in `SAGE_ROOT/src/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>
```

For example:

```
sage --docbuild reference html
```

Two **help** commands which give plenty of documentation for the `sage --docbuild` script:

```
sage --docbuild -h # short help message
sage --docbuild -H # a more comprehensive one
```

Output formats: All output formats supported by Sphinx (e.g. pdf) can be used in Sage. See http://sphinx.pocoo.org/builders.html.

Broken links: in order to build the documentation while reporting the broken links that it contains, use the `--warn-links` flag. Note that Sphinx will not rebuild a document that has not been updated, and thus not report its broken links:

```
sage --docbuild --warn-links reference html
```

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
```

Syntax Highlighting Cython Code

If you want to write *Cython* code in a ReST file, precede the code block by `.. code-block:: cython` instead of the usual `::`. Enable syntax-highlighting in a whole file with `.. highlight:: cython`. Example:

```
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)
```

3.5 Sage Coding Details

3.5.1 Coding in Python for Sage

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

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.

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 eventually 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"""
       Return 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/src/sage/functions/constants.py`:

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

    EXAMPLES::

        sage: pi
        pi
        sage: float(pi) # rel tol 1e-10
        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/src/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)
```

Sage Preparsing

To make Python even more usable interactively, there are a number of tweaks to the syntax made when you use Sage from the commandline or via the notebook (but not for Python code in the Sage library). Technically, this is implemented by a `preparse()` function that rewrites the input string. Most notably, the following replacements are made:

- Sage supports a special syntax for generating rings or, more generally, parents with named generators:

  ```
  sage: R.<x,y> = QQ[]
  sage: preparse('R.<x,y> = QQ[]')
  "R = QQ['x, y']; (x, y,) = R._first_ngens(2)"
  ```

- Integer and real literals are Sage integers and Sage floating point numbers. For example, in pure Python these would be an attribute error:

  ```
  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 preparsing, more examples involving raw literals, etc.

When a file `foo.sage` is loaded or attached in a Sage session, a preparsed version of `foo.sage` is created with the name `foo.sage.py`. The beginning of the preparsed file states:

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

You can explicitly preparse a file with the `--preparse` command-line option: running

```
sage --preparse foo.sage
```

creates the file `foo.sage.py`.

The following files are relevant to preparsing in Sage:

1. `SAGE_ROOT/src/bin/sage`
2. `SAGE_ROOT/src/bin/sage-preparse`
3. `SAGE_ROOT/src/sage/repl/preparse.py`

In particular, the file `preparse.py` contains the Sage preparser code.

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 $3 + 1/2$, 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.

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/src/sage/structure/mutability.py`.

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 == Mod(z, 2)` and `z == 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 == Mod(2,7)`. The goal is to make a hash function that is fast, but within reason respects any obvious natural inclusions and coercions.

Exceptions

Please avoid catch-all code like this:

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

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.

To summarize, only catch specific exceptions as in the following example:

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

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

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/src/sage/algebras/steenrod_algebra.py` started with a line:

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

and that the file `SAGE_ROOT/src/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. One important mechanism in Sage are lazy imports, which don't actually perform the import but delay it until the object is actually used. See `sage.misc.lazy_import` for more details of lazy imports, and *Files and Directory Structure* for an example using lazy imports for a new module.

Deprecation

When making a **backward-incompatible** modification in Sage, the old code should keep working and display a message indicating how it should be updated/written in the future. We call this a *deprecation*.

Note: Deprecated code can only be removed one year after the first stable release in which it appeared.

Each deprecation warning contains the number of the trac ticket that defines it. We use 666 in the examples below. For each entry, consult the function's documentation for more information on its behaviour and optional arguments.

- **Rename a keyword:** by decorating a function/method with `rename_keyword`, any user calling `my_function(my_old_keyword=5)` will see a warning:

```
from sage.misc.decorators import rename_keyword
@rename_keyword(deprecation=666, my_old_keyword='my_new_keyword')
def my_function(my_new_keyword=True):
    return my_new_keyword
```

- **Rename a function/method:** call `deprecated_function_alias()` to obtain a copy of a function that raises a deprecation warning:

```
from sage.misc.superseded import deprecated_function_alias
def my_new_function():
    ...

my_old_function = deprecated_function_alias(666, my_new_function)
```

- **Moving an object to a different module:** if you rename a source file or move some function (or class) to a different file, it should still be possible to import that function from the old module. This can be done using a `lazy_import()` with deprecation. In the old module, you would write:

```
from sage.misc.lazy_import import lazy_import
lazy_import('sage.new.module.name', 'name_of_the_function', deprecation=666)
```

  You can also lazily import everything using $\star$ or a few functions using a tuple:

```
from sage.misc.lazy_import import lazy_import
lazy_import('sage.new.module.name', '*', deprecation=666)
lazy_import('sage.other.module', ('func1', 'func2'), deprecation=666)
```

- **Remove a name from the global namespace:** the function `deprecated_callable_import()` imports an object into the global namespace. Any user who calls it sees a message inviting him to import the object manually:

```
from sage.misc.superseded import deprecated_callable_import
deprecated_callable_import(666,
                     'sage.combinat.name_of_the_module',
                     globals(),
                     locals(),
                     ["name_of_the_function"])
```

  Alternatively, a lazy_import with deprecation would also work in this case.

- **Any other case:** if none of the cases above apply, call `deprecation()` in the function that you want to deprecate. It will display the message of your choice (and interact properly with the doctest framework):

```
from sage.misc.superseded import deprecation
deprecation(666, "Do not use your computer to compute 1+1. Use your brain.")
```

Experimental/Unstable Code

You can mark your newly created code (classes/functions/methods) as experimental/unstable. In this case, no deprecation warning is needed when changing this code, its functionality or its interface.

This should allow you to put your stuff in Sage early, without worrying about making (design) changes later.

When satisfied with the code (when stable for some time, say, one year), you can delete this warning.

As usual, all code has to be fully doctested and go through our reviewing process.

- **Experimental function/method:** use the decorator `mark_as_experimental`. Here is an example:

```
from sage.misc.superseded import experimental
@experimental(66666)
def experimental_function():
    # do something
```

- **Experimental class:** use the decorator `mark_as_experimental` for its `__init__`. Here is an example:

```
from sage.misc.superseded import experimental
class experimental_class(SageObject):
    @experimental(66666)
    def __init__(self, some, arguments):
        # do something
```

- **Any other case:** if none of the cases above apply, call `experimental()` in the code where you want to warn. It will display the message of your choice:

```
from sage.misc.superseded import experimental_warning
experimental_warning(66666, 'This code is not foolproof.')
```

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.

3.5.2 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) and can directly call C and C++ code. As Cython is essentially a superset of the Python language, one often doesn't make a distinction between Cython and Python code in Sage (e.g. one talks of the "Sage Python Library" and "Python Coding Conventions").

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. Cython also allows Sage developers to interface with C and C++ much easier than using the Python C API directly.

Cython is a compiled version of Python. It was originally based on Pyrex but 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 or check out the Language Basics to get started immediately.

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 shared library file (`.so` / `.dll` / `.dylib`) 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.
   (e) A `cpdef` or `def` function, say `testfunction`, defined in a `%cython` cell in a worksheet can be imported and made available in a different `%cython` cell within the same worksheet by importing it as shown below:

   ```
   %cython
   from __main__ import testfunction
   ```

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/src/module_list.py`. See the `distutils.extension.Extension` class for more information on creating a new Cython extension.
   (b) Run `sage -b` to rebuild Sage.

   For example, in order to compile `SAGE_ROOT/src/sage/graphs/chrompoly.pyx`, we see the following lines in `module_list.py`:

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

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 -ggdb
#cfile foo.c
```

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):`.

Interrupt and Signal Handling

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`.

Sage provides two related mechanisms to deal with interrupts:

- *Use sig_check()* if you are writing mixed Cython/Python code. Typically this is code with (nested) loops where every individual statement takes little time.
- *Use sig_on() and sig_off()* if you are calling external C libraries or inside pure Cython code (without any Python functions) where even an individual statement, like a library call, can take a long time.

The functions `sig_check()`, `sig_on()` and `sig_off()` can be put in all kinds of Cython functions: `def`, `cdef` or `cpdef`. You cannot put them in pure Python code (files with extension `.py`). These functions are specific to Sage. To use them, you **must** include the following in your `.pyx` file (it is not sufficient to do this in a `.pxd` file):

```
include "sage/ext/interrupt.pxi"
```

Note: Cython `cdef` or `cpdef` functions with a return type (like `cdef int myfunc():`) need to have an except value to propagate exceptions. Remember this whenever you write `sig_check()` or `sig_on()` inside such a function, otherwise you will see a message like `Exception KeyboardInterrupt: KeyboardInterrupt() in <function name> ignored`.

Using `sig_check()`

`sig_check()` can be used to check for pending interrupts. If an interrupt happens during the execution of C or Cython code, it will be caught by the next `sig_check()`, the next `sig_on()` or possibly the next Python statement. With the latter we mean that certain Python statements also check for interrupts, an example of this is the `print` statement. The following loop *can* be interrupted:

```
sage: cython('while True: print "Hello"')
```

The typical use case for `sig_check()` is within tight loops doing complicated stuff (mixed Python and Cython code, potentially raising exceptions). It is reasonably safe to use and gives a lot of control, because in your Cython code, 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()
```

This `KeyboardInterrupt` is treated like any other Python exception and can be handled as usual:

```
def catch_interrupts():
    try:
        while some_condition():
            sig_check()
            do_something()
    except KeyboardInterrupt:
        # (handle interrupt)
```

Of course, you can also put the `try`/`except` inside the loop in the example above.

The function `sig_check()` is an extremely fast inline function which should have no measurable effect on performance.

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

Another mechanism for interrupt handling is the pair of functions `sig_on()` and `sig_off()`. It is more powerful than `sig_check()` but also a lot more dangerous. 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
```

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 since you cannot call `foo` interactively:

```
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 any Python code or manipulate any Python object (even something trivial like `x = []`), an interrupt can mess up Python's internal state. When in doubt, try to use *sig_check()* instead.
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`. As with `sig_check()`, this exception can be handled in the usual way:

```
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: libGAP and NTL can raise `RuntimeError` and PARI can raise `PariError`. These exceptions behave exactly like the `KeyboardInterrupt` in the example above and can be caught by putting the `sig_on()` inside a `try`/`except` block. See *Error Handling in C Libraries* to see how this is implmented.

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 f1():
    sig_on()
    x = f2()
    sig_off()

def f2():
    sig_on()
    # ...
    sig_off()
    return ans
```

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()  # This must be OUTSIDE the try
    try:
        # (some long computation, potentially raising exceptions)
        return something
    finally:
        sig_off()
```

If you want to also catch this exception, you need a nested `try`:

```
def try_finally_and_catch_example():
    try:
        sig_on()
        try:
            # (some long computation, potentially raising exceptions)
        finally:
            sig_off()
    except Exception:
        print "Trouble!Trouble!"
```

`sig_on()` is implemented using the C library call `setjmp()` which takes a very small but still measurable amount of time. In very time-critical code, one can conditionally call `sig_on()` and `sig_off()`:

```
def conditional_sig_on_example(long n):
    if n > 100:
        sig_on()
    # (do something depending on n)
    if n > 100:
        sig_off()
```

This should only be needed if both the check (`n > 100` in the example) and the code inside the `sig_on()` block take very little time. In Sage versions before 4.7, `sig_on()` was much slower, that's why there are more checks like this in old code.

Other Signals

Apart from handling interrupts, `sig_on()` provides more general signal handling. For example, it handles `alarm()` time-outs by raising an `AlarmInterrupt` (inherited from `KeyboardInterrupt`) exception.

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 an exception is raised (`RuntimeError` for SIGABRT, `FloatingPointError` for SIGFPE and the custom exception `SignalError`, based on `BaseException`, otherwise):

```
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 be handled by a `try`/`except` block as explained above. A segmentation fault or `abort()` unguarded by `sig_on()` would simply terminate Sage. This applies only to `sig_on()`, the function `sig_check()` only deals with interrupts and alarms.

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.

Error Handling in C Libraries

Some C libraries can produce errors and use some sort of callback mechanism to report errors: an external error handling function needs to be set up which will be called by the C library if an error occurs.

The function `sig_error()` can be used to deal with these errors. This function may only be called within a `sig_on()` block (otherwise Sage will crash hard) after raising a Python exception. You need to use the Python/C API for this and call `sig_error()` after calling some variant of `PyErr_SetObject()`. Even within Cython, you cannot use the `raise` statement, because then the `sig_error()` will never be executed. The call to `sig_error()` will use the `sig_on()` machinery such that the exception will be seen by `sig_on()`.

A typical error handler implemented in Cython would look as follows:

```
include "sage/ext/interrupt.pxi"
from cpython.exc cimport PyErr_SetString

cdef void error_handler(char *msg):
    PyErr_SetString(RuntimeError, msg)
    sig_error()
```

In Sage, this mechanism is used for libGAP, NTL and PARI.

Advanced Functions

There are several more specialized functions for dealing with interrupts. 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 (like `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/src/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. The best way to do this is to use `alarm()`.

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

```
sage: alarm(0.5); factor(10^1000 + 3)
Traceback (most recent call last):
...
AlarmInterrupt
```

Releasing the Global Interpreter Lock (GIL)

All the functions related to interrupt and signal handling do not require the Python GIL (if you don't know what this means, you can safely ignore this section), they are declared `nogil`. This means that they can be used in Cython code inside `with nogil` blocks. If `sig_on()` needs to raise an exception, the GIL is temporarily acquired internally.

If you use C libraries without the GIL and you want to raise an exception before calling *sig_error()*, remember to acquire the GIL while raising the exception. Within Cython, you can use a with gil context.

Warning: The GIL should never be released or acquired inside a `sig_on()` block. If you want to use a `with nogil` block, put both `sig_on()` and `sig_off()` inside that block. When in doubt, choose to use `sig_check()` instead, which is always safe to use.

Unpickling Cython Code

Pickling for Python classes and extension classes, such as Cython, is different. This is discussed in the Python pickling documentation. For the unpickling of extension classes you need to write a `__reduce__()` method which typically returns a tuple `(f, args, ...)` such that `f(*args)` returns (a copy of) the original object. As an example, the following code snippet is the `__reduce__()` method from `sage.rings.integer.Integer`:

```
def __reduce__(self):
    '''
    This is used when pickling integers.

    EXAMPLES::

        sage: n = 5
        sage: t = n.__reduce__(); t
        (<built-in function make_integer>, ('5',))
        sage: t[0](*t[1])
        5
        sage: loads(dumps(n)) == n
        True
    '''
    # This single line below took me HOURS to figure out.
    # It is the *trick* needed to pickle Cython extension types.
    # The trick is that you must put a pure Python function
    # as the first argument, and that function must return
```

```
    # the result of unpickling with the argument in the second
    # tuple as input. All kinds of problems happen
    # if we don't do this.
    return sage.rings.integer.make_integer, (self.str(32),)
```

3.5.3 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 Sage [SageComponents]. 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*.

The PARI C Library Interface

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 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/src/sage/libs/pari/gen.pyx`, and this is where we add the method `matfrobenius`:

```
def matfrobenius(self, flag=0):
    r"""
    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
```

GAP

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 );
```

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/src/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.

LibGAP

The disadvantage of using other programs through interfaces is that there is a certain unavoidable latency (of the order of 10ms) involved in sending input and receiving the result. If you have to call functions in a tight loop this can be unacceptably slow. Calling into a shared library has much lower latency and furthermore avoids having to convert everything into a string in-between. This is why Sage includes a shared library version of the GAP kernel, available as *libgap* in Sage. The libgap analogue of the first example in *GAP* is:

```
sage: SimpleLieAlgebra = libgap.function_factory('SimpleLieAlgebra')
sage: L = SimpleLieAlgebra('G', 2, QQ)
sage: R = L.RootSystem();  R
<root system of rank 2>
sage: R.CartanMatrix()     # output is a GAP matrix
[ [ 2, -1 ], [ -3, 2 ] ]
sage: matrix(R.CartanMatrix())    # convert to Sage matrix
[ 2 -1]
[-3  2]
```

Singular

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];")
''
sage: singular.eval("setring R;")
''
sage: L = singular.eval("POINTS;")

sage: print L
[1]:
   [1]:
      0
   [2]:
      1
   [3]:
      0
[2]:
   [1]:
      -2
   [2]:
      -1
   [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=PolynomialRing(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))
```

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')
```

Note that these elements of L are defined modulo 5 in Singular, and they compare differently than you would expect from their print representation:

```
sage: sorted([(L[i][1], L[i][2], L[i][3]) for i in range(1,7)])
[(0, 0, 1), (0, 1, 0), (2, 2, 1), (2, -2, 1), (-2, 1, 1), (-2, -1, 1)]
```

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.

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/src/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)    # optional - octave
        [-0.33333299999999999, 0.66666700000000001, -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/src/sage/interfaces/` for examples of interfaces to other software packages.

3.6 Packaging Third-Party Code

3.6.1 Packaging Third-Party Code

One of the mottoes of the Sage project is to not reinvent the wheel: If an algorithm is already implemented in a well-tested library then consider incorporating that library into Sage. The current list of available packages are the subdirectories of `SAGE_ROOT/build/pkgs/`. The installation of packages is done through a bash script located in `SAGE_ROOT/build/bin/sage-spkg`. This script is typically invoked by giving the command:

```
[user@localhost]$ sage -i <options> <package name>...
```

options can be:

- -f: install a package even if the same version is already installed
- -s: do not delete temporary build directory
- -c: after installing, run the test suite for the spkg. This should override the settings of `SAGE_CHECK` and `SAGE_CHECK_PACKAGES`.
- -d: only download the package

The section *Directory Structure* describes the structure of each individual package in `SAGE_ROOT/build/pkgs`. In section *Building the package* we see how you can install and test a new spkg that you or someone else wrote. Finally, *Inclusion Procedure for New and Updated Packages* explains how to submit a new package for inclusion in the Sage source code.

Package types

Not all packages are built by default, they are divided into standard, optional and experimental ones:

- **standard** packages are built by default. They have stringent quality requirements: they should work on all supported platforms. In order for a new standard package to be accepted, it should have been optional for a while, see *Inclusion Procedure for New and Updated Packages*.
- **optional** packages are subject to the same requirements, they should also work on all supported platforms. If there are *optional doctests* in the Sage library, those tests must pass. Note that optional packages are not tested as much as standard packages, so in practice they might break more often than standard packages.
- for **experimental** packages, the bar is much lower: even if there are some problems, the package can still be accepted.

Directory Structure

Third-party packages in Sage consists of two parts:

1. The tarball as it is distributed by the third party, or as close as possible. Valid reasons for modifying the tarball are deleting unnecessary files to keep the download size manageable, regenerating auto-generated files or changing the directory structure if necessary. In certain cases, you may need to (additionally) change the filename of the tarball. In any case, the actual code must be unmodified: if you need to change the sources, add a *patch* instead. See also *Modified Tarballs* for automating the modifications to the upstream tarball.
2. The build scripts and associated files are in a subdirectory `SAGE_ROOT/build/pkgs/<package>`, where you replace `<package>` with a lower-case version of the upstream project name. If the project name contains characters which are not alphanumeric and are not an underscore, those characters should be removed or replaced by an underscore. For example, the project `FFLAS-FFPACK` is called `fflas_ffpack` in Sage and `path.py` is renamed `pathpy` in Sage.

As an example, let us consider a hypothetical FoO project. They (upstream) distribute a tarball `FoO-1.3.tar.gz` (that will be automatically placed in `SAGE_ROOT/upstream` during the installation process). To package it in Sage, we create a subdirectory containing as a minimum the following files:

```
SAGE_ROOT/build/pkgs/foo
|-- checksums.ini
|-- dependencies
|-- package-version.txt
|-- spkg-install
|-- SPKG.txt
`-- type
```

The following are some additional files which can be added:

```
SAGE_ROOT/build/pkgs/foo
|-- patches
|   |-- bar.patch
|   `-- baz.patch
|-- spkg-check
`-- spkg-src
```

We discuss the individual files in the following sections.

Package type

The file `type` should contain a single word, which is either `standard`, `optional` or `experimental`. See *Package types* for the meaning of these types.

Install Script

The `spkg-install` file is a shell script or Python script which installs the package. In the best case, the upstream project can simply be installed by the usual configure / make / make install steps. In that case, the build script would simply consist of:

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

cd src

./configure --prefix="$SAGE_LOCAL" --libdir="$SAGE_LOCAL/lib"
if [ $? -ne 0 ]; then
    echo >&2 "Error configuring PACKAGE_NAME."
    exit 1
fi

$MAKE
if [ $? -ne 0 ]; then
    echo >&2 "Error building PACKAGE_NAME."
    exit 1
fi

$MAKE install
if [ $? -ne 0 ]; then
    echo >&2 "Error installing PACKAGE_NAME."
    exit 1
fi
```

Note that the top-level directory inside the tarball is renamed to `src` before calling the `spkg-install` script, so you can just use `cd src` instead of `cd foo-1.3`.

If there is any meaningful documentation included but not installed by `make install`, then you can add something like the following to install it:

```
if [ "$SAGE_SPKG_INSTALL_DOCS" = yes ] ; then
    $MAKE doc
    if [ $? -ne 0 ]; then
        echo >&2 "Error building PACKAGE_NAME docs."
        exit 1
    fi
    mkdir -p "$SAGE_LOCAL/share/doc/PACKAGE_NAME"
    cp -R doc/* "$SAGE_ROOT/local/share/doc/PACKAGE_NAME"
fi
```

Self-Tests

The `spkg-check` file is an optional, but highly recommended, script to run self-tests of the package. It is run after building and installing if the `SAGE_CHECK` environment variable is set, see the Sage installation guide. Ideally, upstream has some sort of tests suite that can be run with the standard `make check` target. In that case, the `spkg-check` script would simply contain:

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

cd src
$MAKE check
```

The SPKG.txt File

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

```
= PACKAGE_NAME =

== Description ==

What does the package do?

== License ==

What is the license? If non-standard, is it GPLv3+ compatible?

== Upstream Contact ==

Provide information for upstream contact.

== Dependencies ==

Put a bulleted list of dependencies here:

* python
* readline

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

```
If the tarball was modified by hand and not via a spkg-src
script, describe what was changed.
```

with `PACKAGE_NAME` replaced by the the package name. Legacy `SPKG.txt` files have an additional changelog section, but this information is now kept in the git repository.

Package dependencies

Many packages depend on other packages. Consider for example the `eclib` package for elliptic curves. This package uses the libraries PARI, NTL and FLINT. So the following is the `dependencies` file for `eclib`:

```
$(INST)/$(PARI) $(INST)/$(NTL) $(INST)/$(FLINT)

----------
All lines of this file are ignored except the first.
It is copied by SAGE_ROOT/build/make/install into SAGE_ROOT/build/make/Makefile.
```

If there are no dependencies, you can use

```
# no dependencies

----------
All lines of this file are ignored except the first.
It is copied by SAGE_ROOT/build/make/install into SAGE_ROOT/build/make/Makefile.
```

There are actually two kinds of dependencies: there are normal dependencies and order-only dependencies, which are weaker. The syntax for the `dependencies` file is

```
normal dependencies | order-only dependencies
```

If there is no `|`, then all dependencies are normal.

- If package A has an **order-only dependency** on B, it simply means that B must be built before A can be built. The version of B does not matter, only the fact that B is installed matters. This should be used if the dependency is purely a build-time dependency (for example, a dependency on Python simply because the `spkg-install` file is written in Python).
- If A has a **normal dependency** on B, it means additionally that A should be rebuilt every time that B gets updated. This is the most common kind of dependency. A normal dependency is what you need for libraries: if we upgrade NTL, we should rebuild everything which uses NTL.

In order to check that the dependencies of your package are likely correct, the following command should work without errors:

```
[user@localhost]$ make distclean && make base && make PACKAGE_NAME
```

Finally, note that standard packages should only depend on standard packages and optional packages should only depend on standard or optional packages.

Patching Sources

Actual changes to the source code must be via patches, which should be placed in the `patches` directory. GNU patch is distributed with Sage, so you can rely on it being available. Patches must include documentation in their header (before the first diff hunk), so a typical patch file should look like this:

```
Add autodoc_builtin_argspec config option

Following the title line you can add a multi-line description of
what the patch does, where you got it from if you did not write it
yourself, if they are platform specific, if they should be pushed
upstream, etc...

diff -dru Sphinx-1.2.2/sphinx/ext/autodoc.py.orig Sphinx-1.2.2/sphinx/ext/autodoc.py
--- Sphinx-1.2.2/sphinx/ext/autodoc.py.orig  2014-03-02 20:38:09.000000000 +1300
+++ Sphinx-1.2.2/sphinx/ext/autodoc.py  2014-10-19 23:02:09.000000000 +1300
@@ -1452,6 +1462,7 @@

     app.add_config_value('autoclass_content', 'class', True)
     app.add_config_value('autodoc_member_order', 'alphabetic', True)
+    app.add_config_value('autodoc_builtin_argspec', None, True)
     app.add_config_value('autodoc_default_flags', [], True)
     app.add_config_value('autodoc_docstring_signature', True, True)
     app.add_event('autodoc-process-docstring')
```

Patches to files in `src/` need to be applied in `spkg-install`, that is, if there are any patches then your `spkg-install` script should contain a section like this:

```
for patch in ../patches/*.patch; do
    [ -r "$patch" ] || continue  # Skip non-existing or non-readable patches
    patch -p1 <"$patch"
    if [ $? -ne 0 ]; then
        echo >&2 "Error applying '$patch'"
        exit 1
    fi
done
```

which applies the patches to the sources.

Modified Tarballs

The `spkg-src` file is optional and only to document how the upstream tarball was changed. Ideally it is not modified, then there would be no `spkg-src` file present either.

However, if you really must modify the upstream tarball then it is recommended that you write a script, called `spkg-src`, that makes the changes. This not only serves as documentation but also makes it easier to apply the same modifications to future versions.

Package Versioning

The `package-version.txt` file containts just the version. So if upstream is `FoO-1.3.tar.gz` then the package version file would only contain `1.3`.

If the upstream package is taken from some revision other than a stable version or if upstream doesn't have a version number, you should use the date at which the revision is made. For example, the `database_stein_watkins` package with version `20110713` contains the database as of 2011-07-13. Note that the date should refer to the *contents* of the tarball, not to the day it was packaged for Sage. This particular Sage package for `database_stein_watkins` was created in 2014, but the data it contains was last updated in 2011.

If you apply any patches, or if you made changes to the upstream tarball (see *Directory Structure* for allowable changes), then you should append a `.p0` to the version to indicate that it's not a vanilla package.

Additionally, whenever you make changes to a package *without* changing the upstream tarball (for example, you add an additional patch or you fix something in the `spkg-install` file), you should also add or increase the patch level. So the different versions would be `1.3`, `1.3.p0`, `1.3.p1`, ... The change in version number or patch level will trigger re-installation of the package, such that the changes are taken into account.

Checksums

The `checksums.ini` file contains checksums of the upstream tarball. It is autogenerated, so you just have to place the upstream tarball in the `SAGE_ROOT/upstream/` directory and run:

```
[user@localhost]$ sage --fix-pkg-checksums
```

Building the package

At this stage you have a new tarball that is not yet distributed with Sage (`FoO-1.3.tar.gz` in the example of section *Directory Structure*). Now you need to manually place it in the `SAGE_ROOT/upstream/` directory and run `sage --fix-pkg-checksums` if you have not done that yet.

In order to update `build/make/Makefile`, which contains the rules to build all packages, you need to run the following command from `SAGE_ROOT`:

```
[user@localhost]$ ./configure
```

You need to re-run `./configure` whenever you change any package metadata: if you add or remove a package or if you change the version, type or dependencies of a package.

Now you can install the package using:

```
[user@localhost]$ sage -i package_name
```

or:

```
[user@localhost]$ sage -f package_name
```

to force a reinstallation. If your package contains a `spkg-check` script (see *Self-Tests*) it can be run with:

```
[user@localhost]$ sage -i -c package_name
```

or:

```
[user@localhost]$ sage -f -c package_name
```

If all went fine, open a ticket, put a link to the original tarball in the ticket and upload a branch with the code under `SAGE_ROOT/build/pkgs`.

Inclusion Procedure for New and Updated Packages

Packages that are not part of Sage will first become optional or experimental (the latter if they will not build on all supported systems). After they have been in optional for some time without problems they can be proposed to be included as standard packages in Sage.

To propose a package for optional/experimental inclusion please open a trac ticket with the respective `Component:` field set to either `packages:experimental` or `packages:optional`. The associated code requirements are described in the following sections.

After the ticket was reviewed and included, optional packages stay in that status for at least a year, after which they can be proposed to be included as standard packages in Sage. For this a trac ticket is opened with the `Component:` field set to `packages:standard`. Then make a proposal in the Google Group `sage-devel`.

Upgrading packages to new upstream versions or with additional patches includes opening a ticket in the respective category too, as described above.

License Information

If you are patching a standard Sage spkg, then you should make sure that the license information for that package is up-to-date, both in its `SPKG.txt` file and in the file `SAGE_ROOT/COPYING.txt`. For example, if you are producing an spkg which upgrades the vanilla source to a new version, check whether the license changed between versions.

Prerequisites for New Standard Packages

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

- **License**. For standard packages, the license must be compatible with the GNU General Public License, version 3. The Free Software Foundation maintains a long list of licenses and comments about them.
- **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 and on which Sage almost works 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.
- **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
  - Absence of memory leaks
  - Maintainable
  - Portability
  - Reasonable build time, size, dependencies
- **Previously an optional package**. A new standard package must have spent some time as an optional package. Or have a good reason why this is not possible.
- **Refereeing**. The code must be refereed, as discussed in *The Sage Trac Server*.

3.6.2 Packaging Old-Style SPKGs

This chapter explains old-style spkgs; It applies only to legacy optional spkgs and experimental spkgs.

> **Warning:** Old-style packages are **deprecated**, it is strongly suggested that you make a new-style package instead. See *Packaging Third-Party Code* for the modern way of packaging third-party software.

Creating an Old-Style SPKG

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.

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. When you compile Sage from a source distribution (or when you run `sage -p <pkg>`), the file `SAGE_ROOT/build/bin/sage-spkg` 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. The URL http://www.sagemath.org/download-packages.html lists 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 *Overview of Patching SPKGs*), 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. 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`: Old-style spkgs use Mercurial for its revision control system. 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. To create this Mercurial repository from scratch, you should do:

  ```
  hg init
  ```

  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 *The File spkg-install* 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 *The File SPKG.txt* 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/`. See *Overview of Patching SPKGs*. Patches to files in `src/` should be applied in `spkg-install`, and all patches must be self-documenting, i.e. the header must contain 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 only exception to this rule is for *removals* of unused files or directories. Some packages contain parts which are not needed for Sage. To save space, these may be removed directly from `src/`. But be sure to document this in the "Special Update/Build Instructions" section in `SPKG.txt`!

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 >&2 "SAGE_LOCAL undefined ... exiting"
    echo >&2 "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
    [ -r "$patch" ] || continue  # Skip non-existing or non-readable patches
    patch -p1 <"$patch"
    if [ $? -ne 0 ]; then
        echo >&2 "Error applying '$patch'"
        exit 1
    fi
done

./configure --prefix="$SAGE_LOCAL"
if [ $? -ne 0 ]; then
    echo >&2 "Error configuring PACKAGE_NAME."
    exit 1
fi

$MAKE
if [ $? -ne 0 ]; then
    echo >&2 "Error building PACKAGE_NAME."
    exit 1
fi

$MAKE install
if [ $? -ne 0 ]; then
    echo >&2 "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 [ $? -ne 0 ]; then
        echo >&2 "Error building PACKAGE_NAME docs."
        exit 1
    fi
    mkdir -p "$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 old-style `SPKG.txt` file is the same as described in *The SPKG.txt File*, but with a hand-maintained changelog appended since the contents are not part of the Sage repository tree. It should follow the following pattern:

```
== 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*) 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 -p 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: External Magma code goes in `SAGE_ROOT/src/ext/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.

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.

Overview of Patching SPKGs

Make sure you are familiar with the structure and conventions relating to spkg's; see the chapter *Packaging Old-Style SPKGs* 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.

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 `.hgignore` 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 *Packaging Old-Style SPKGs* 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 bzipped tar files or just plain tar files; the command `sage --pkg ...` produces the bzipped 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.

SPKG Versioning

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 SPKGs*. 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 -p http://URL/to/package-x.y.z.spkg
```

or:

```
sage -p /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`.

CHAPTER

FOUR

SAGE NOTEBOOK DEVELOPER GUIDE

4.1 Sage Notebook Developer Guide

Development of the Sage notebook currently occurs on Github using the Git revision control system. The development model for the Sage Notebook project is a git and github workflow.

To update to the latest development source, run the commands below, where `SAGE_ROOT` is the root directory of the Sage installation, and where `hackdir` is a directory you create for working on code changes (it need not have the name or location given below).

Warning: This will create a new sagenb repository ignoring any changes you have made to the files.

```
mkdir ~/hackdir
cd ~/hackdir
git clone git://github.com/sagemath/sagenb.git sagenb-git
cd SAGE_ROOT/devel
rm sagenb
ln -s ~/hackdir/sagenb sagenb
cd sagenb
../../sage setup.py develop
```

What this has done is to create a new directory, move to that directory, and create a clone of the most up-to-date version of the upstream notebook sources there. Then we remove a symbolic link `sagenb` in the Sage folder and replace it with a link to your clone of upstream, finally making sure that the notebook has the correct dependencies.

An advantage of having the separate directory for sagenb is that you would later be able to keep it and do development work in it even when you upgrade Sage, or even if you accidentally destroy your Sage installation somehow.

The rest of these instructions is some very generic documentation, slightly adapted to help develop the notebook using Git and Github.

The most important section involves how to update your new sagenb source repository and create a "fork" of the master copy, so that you will be able to request your changes to be merged in the Sage notebook, called a "pull request"; see *Git for Development*.

4.1.1 Following the Latest Source

These are the instructions if you just want to follow the latest *Sage Notebook* source, but you don't need to do any development for now.

The steps are:

- *Installing Git*

- get local copy of the Sage Notebook github git repository
- update local copy from time to time

Get the Local Copy of the Code

From the command line:

```
git clone git://github.com/sagemath/sagenb.git
```

You now have a copy of the code tree in the new `sagenb` directory.

Updating the Code

From time to time you may want to pull down the latest code. Do this with:

```
cd sagenb
git pull
```

The tree in `sagenb` will now have the latest changes from the initial repository.

4.1.2 Making a Patch

You've discovered a bug or something else you want to change in Sage Notebook — excellent!

You've worked out a way to fix it — even better!

You want to tell us about it — best of all!

The easiest way is to make a *patch* or set of patches. Here we explain how.

Making a patch is simple and quick, but it is not part of our normal workflow. So if you are going to be doing anything more than a once-off patch one time, please consider following the *Git for Development* model instead. See especially the part about "pull requests" at *The Editing Workflow*.

Making Patches

Overview

```
# tell git who you are
git config --global user.email you@yourdomain.example.com
git config --global user.name "Your Name Comes Here"
# get the repository if you don't have it
git clone git://github.com/sagemath/sagenb.git
# make a branch for your patching
cd sagenb
git branch the-fix-im-thinking-of
git checkout the-fix-im-thinking-of
# hack, hack, hack
# Tell git about any new files you've made
git add somewhere/tests/test_my_bug.py
# commit work in progress as you go
git commit -am 'BF - added tests for Funny bug'
# hack hack, hack
git commit -am 'BF - added fix for Funny bug'
```

```
# make the patch files
git format-patch -M -C master
```

You may attach a short generated patch file to the Sage Notebook mailing list or better, open an issue at the Sage Notebook github site (see *Git for Development*) and cut and paste your patch in a comment there. In either case we will thank you warmly.

In Detail

1. Tell git who you are so it can label the commits you've made:

   ```
   git config --global user.email you@yourdomain.example.com
   git config --global user.name "Your Name Comes Here"
   ```

2. If you don't already have one, clone a copy of the Sage Notebook repository:

   ```
   git clone git://github.com/sagemath/sagenb.git
   cd sagenb
   ```

3. Make a 'feature branch'. This will be where you work on your bug fix. It's nice and safe and leaves you with access to an unmodified copy of the code in the main branch:

   ```
   git branch the-fix-im-thinking-of
   git checkout the-fix-im-thinking-of
   ```

4. Do some edits, and commit them as you go:

   ```
   # hack, hack, hack
   # Tell git about any new files you've made
   git add somewhere/tests/test_my_bug.py
   # commit work in progress as you go
   git commit -am 'BF - added tests for Funny bug'
   # hack hack, hack
   git commit -am 'BF - added fix for Funny bug'
   ```

   Note the `-am` options to `commit`. The `m` flag just signals that you're going to type a message on the command line. The `a` flag — you can just take on faith — or see why the -a flag?.

5. When you have finished, check you have committed all your changes:

   ```
   git status
   ```

6. Finally, make your commits into patches. You want all the commits since you branched from the `master` branch:

   ```
   git format-patch -M -C master
   ```

   You will now have several files named for the commits:

   ```
   0001-BF-added-tests-for-Funny-bug.patch
   0002-BF-added-fix-for-Funny-bug.patch
   ```

   Although some projects would have you send these files to the Sage Notebook mailing list, we prefer submitting an issue request at the web interface to the Sage Notebook github page. See *The Editing Workflow* for how to create a "pull request" once you have created a Github account.

When you are done, to switch back to the main copy of the code, just return to the `master` branch:

```
git checkout master
```

Moving from Patching to Development

If you find you have done some patches, and you have one or more feature branches, you will probably want to switch to development mode. You can do this with the repository you have.

Fork the Sage Notebook repository on github — *Making Your Own Copy (Fork) of Sage Notebook*. Then:

```
# checkout and refresh master branch from main repo
git checkout master
git pull origin master
# rename pointer to main repository to 'upstream'
git remote rename origin upstream
# point your repo to default read / write to your fork on github
git remote add origin git@github.com:your-user-name/sagenb.git
# push up any branches you've made and want to keep
git push origin the-fix-im-thinking-of
```

Then you can, if you want, follow the *Development Workflow*.

4.1.3 Git for Development

Contents:

Making Your Own Copy (Fork) of Sage Notebook

You need to do this only once. The instructions here are very similar to the instructions at http://help.github.com/forking/ — please see that page for more detail. We're repeating some of it here just to give the specifics for the Sage Notebook project, and to suggest some default names.

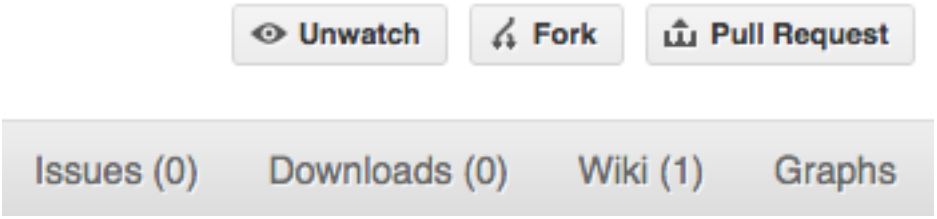
Set Up and Configure a Github Account

If you don't have a github account, go to the github page, and make one.

You then need to configure your account to allow write access — see the `Generating SSH keys` help on github help.

Create Your Own Forked Copy of Sage Notebook

1. Log into your github account.
2. Go to the Sage Notebook github home at Sage Notebook github.
3. Click on the *fork* button:



Now, after a short pause, you should find yourself at the home page for your own forked copy of Sage Notebook.

Set Up Your Fork

First you follow the instructions for *Making Your Own Copy (Fork) of Sage Notebook*.

Overview

```
git clone git@github.com:your-user-name/sagenb.git
cd sagenb
git remote add upstream git://github.com/sagemath/sagenb.git
```

In etail

Clone Your Fork

1. Clone your fork to the local computer with `git clone git@github.com:your-user-name/sagenb.git`
2. Investigate. Change directory to your new repo: `cd sagenb`. Then `git branch -a` to show you all branches. You'll get something like:

   ```
   * master
   remotes/origin/master
   ```

   This tells you that you are currently on the `master` branch, and that you also have a `remote` connection to `origin/master`. What remote repository is `remote/origin`? Try `git remote -v` to see the URLs for the remote. They will point to your github fork.

   Now you want to connect to the upstream Sage Notebook github repository, so you can merge in changes from trunk.

Linking Your Repository to the Upstream Repo

```
cd sagenb
git remote add upstream git://github.com/sagemath/sagenb.git
```

`upstream` here is just the arbitrary name we're using to refer to the main Sage Notebook repository at Sage Notebook github.

Note that we've used `git://` for the URL rather than `git@`. The `git://` URL is read only. This means we that we can't accidentally (or deliberately) write to the upstream repo, and we are only going to use it to merge into our own code.

Just for your own satisfaction, show yourself that you now have a new 'remote', with `git remote -v show`, giving you something like:

```
upstream     git://github.com/sagemath/sagenb.git (fetch)
upstream     git://github.com/sagemath/sagenb.git (push)
origin       git@github.com:your-user-name/sagenb.git (fetch)
origin       git@github.com:your-user-name/sagenb.git (push)
```

Development Workflow

You already have your own forked copy of the Sage Notebook repository, by following *Making Your Own Copy (Fork) of Sage Notebook*. You have *Set Up Your Fork*. You have configured git by following *Configuration Tips*. Now you are ready for some real work.

Workflow Summary

In what follows we'll refer to the upstream Sage Notebook `master` branch, as "trunk".

- Don't use your `master` branch for anything. Consider deleting it.
- When you are starting a new set of changes, fetch any changes from trunk, and start a new *feature branch* from that.
- Make a new branch for each separable set of changes — "one task, one branch" (ipython git workflow).
- Name your branch for the purpose of the changes - e.g. `bugfix-for-issue-14` or `refactor-database-code`.
- If you can possibly avoid it, avoid merging trunk or any other branches into your feature branch while you are working.
- If you do find yourself merging from trunk, consider *Rebasing on trunk*
- Ask on the Sage Notebook mailing list if you get stuck.
- Ask for code review!

This way of working helps to keep work well organized, with readable history. This in turn makes it easier for project maintainers (that might be you) to see what you've done, and why you did it.

See linux git workflow and ipython git workflow for some explanation.

Consider Deleting Your Master Branch

It may sound strange, but deleting your own `master` branch can help reduce confusion about which branch you are on. See deleting master on github for details.

Update the Mirror of trunk

First make sure you have done *Linking Your Repository to the Upstream Repo*.

From time to time you should fetch the upstream (trunk) changes from github:

```
git fetch upstream
```

This will pull down any commits you don't have, and set the remote branches to point to the right commit. For example, 'trunk' is the branch referred to by (remote/branchname) `upstream/master` - and if there have been commits since you last checked, `upstream/master` will change after you do the fetch.

Make a New Feature Branch

When you are ready to make some changes to the code, you should start a new branch. Branches that are for a collection of related edits are often called 'feature branches'.

Making an new branch for each set of related changes will make it easier for someone reviewing your branch to see what you are doing.

Choose an informative name for the branch to remind yourself and the rest of us what the changes in the branch are for. For example `add-ability-to-fly`, or `buxfix-for-issue-42`.

```
# Update the mirror of trunk
git fetch upstream
# Make new feature branch starting at current trunk
git branch my-new-feature upstream/master
git checkout my-new-feature
```

Generally, you will want to keep your feature branches on your public github fork of Sage Notebook. To do this, you git push this new branch up to your github repo. Generally (if you followed the instructions in these pages, and by default), git will have a link to your github repo, called `origin`. You push up to your own repo on github with:

```
git push origin my-new-feature
```

In git >= 1.7 you can ensure that the link is correctly set by using the `--set-upstream` option:

```
git push --set-upstream origin my-new-feature
```

From now on git will know that `my-new-feature` is related to the `my-new-feature` branch in the github repo.

The Editing Workflow

Overview

```
# hack hack
git add my_new_file
git commit -am 'NF - some message'
git push
```

In More Detail

1. Make some changes
2. See which files have changed with `git status` (see git status). You'll see a listing like this one:

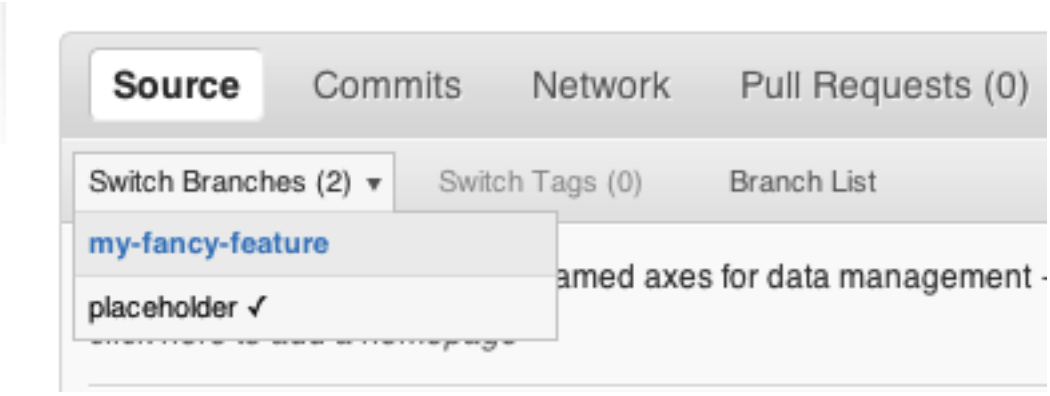
```
# On branch ny-new-feature
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#  modified:   README
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#  INSTALL
no changes added to commit (use "git add" and/or "git commit -a")
```

3. Check what the actual changes are with `git diff` (git diff).
4. Add any new files to version control `git add new_file_name` (see git add).
5. To commit all modified files into the local copy of your repo,, do `git commit -am 'A commit message'`. Note the `-am` options to `commit`. The `m` flag just signals that you're going to type a message on the command line. The `a` flag — you can just take on faith — or see why the -a flag? — and the helpful use-case description in the tangled working copy problem. The git commit manual page might also be useful.
6. To push the changes up to your forked repo on github, do a `git push` (see git push).

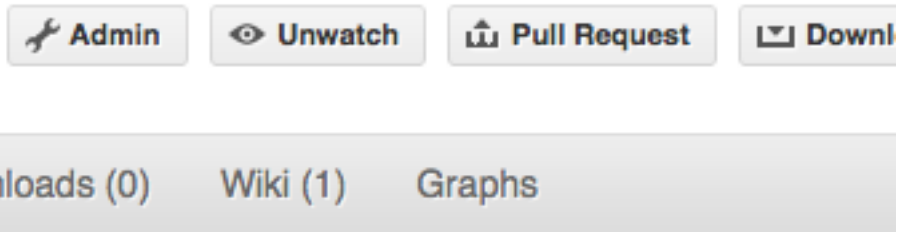
Ask for Your Changes to be Reviewed or Merged

When you are ready to ask for someone to review your code and consider a merge:

1. Go to the URL of your forked repo, say `http://github.com/your-user-name/sagenb`.
2. Use the 'Switch Branches' dropdown menu near the top left of the page to select the branch with your changes:



3. Click on the 'Pull request' button:



Enter a title for the set of changes, and some explanation of what you've done. Say if there is anything you'd like particular attention for - like a complicated change or some code you are not happy with.

If you don't think your request is ready to be merged, just say so in your pull request message. This is still a good way of getting some preliminary code review.

Some Other Things You Might Want to Do

Delete a Branch on Github

```
git checkout master
# delete branch locally
git branch -D my-unwanted-branch
# delete branch on github
git push origin :my-unwanted-branch
```

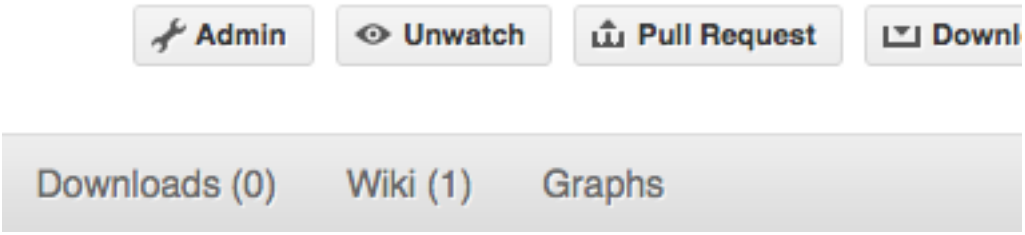
(Note the colon `:` before `test-branch`. See also: http://github.com/guides/remove-a-remote-branch

Several People Sharing a Single Repository If you want to work on some stuff with other people, where you are all committing into the same repository, or even the same branch, then just share it via github.

First fork Sage Notebook into your account, as from *Making Your Own Copy (Fork) of Sage Notebook*.

Then, go to your forked repository github page, say `http://github.com/your-user-name/sagenb`

Click on the 'Admin' button, and add anyone else to the repo as a collaborator:



Now all those people can do:

```
git clone git@githhub.com:your-user-name/sagenb.git
```

Remember that links starting with `git@` use the ssh protocol and are read-write; links starting with `git://` are read-only.

Your collaborators can then commit directly into that repo with the usual:

```
git commit -am 'ENH - much better code'
git push origin master # pushes directly into your repo
```

Explore Your Repository To see a graphical representation of the repository branches and commits:

```
gitk --all
```

To see a linear list of commits for this branch:

```
git log
```

You can also look at the network graph visualizer for your github repo.

Finally the *Fancy Log Output* `lg` alias will give you a reasonable text-based graph of the repository.

Rebasing on trunk Let's say you thought of some work you'd like to do. You *Update the Mirror of trunk* and *Make a New Feature Branch* called `cool-feature`. At this stage trunk is at some commit, let's call it E. Now you make some new commits on your `cool-feature` branch, let's call them A, B, C. Maybe your changes take a while, or you come back to them after a while. In the meantime, trunk has progressed from commit E to commit (say) G:

```
      A---B---C cool-feature
     /
D---E---F---G trunk
```

At this stage you consider merging trunk into your feature branch, and you remember that this here page sternly advises you not to do that, because the history will get messy. Most of the time you can just ask for a review, and not worry that trunk has got a little ahead. But sometimes, the changes in trunk might affect your changes, and you need to harmonize them. In this situation you may prefer to do a rebase.

rebase takes your changes (A, B, C) and replays them as if they had been made to the current state of `trunk`. In other words, in this case, it takes the changes represented by A, B, C and replays them on top of G. After the rebase, your history will look like this:

```
              A'--B'--C' cool-feature
             /
D---E---F---G trunk
```

See rebase without tears for more detail.

To do a rebase on trunk:

```
# Update the mirror of trunk
git fetch upstream
# go to the feature branch
git checkout cool-feature
# make a backup in case you mess up
git branch tmp cool-feature
# rebase cool-feature onto trunk
git rebase --onto upstream/master upstream/master cool-feature
```

In this situation, where you are already on branch `cool-feature`, the last command can be written more succinctly as:

```
git rebase upstream/master
```

When all looks good you can delete your backup branch:

```
git branch -D tmp
```

If it doesn't look good you may need to have a look at *Recovering From Mess-Ups*.

If you have made changes to files that have also changed in trunk, this may generate merge conflicts that you need to resolve - see the git rebase man page for some instructions at the end of the "Description" section. There is some related help on merging in the git user manual - see resolving a merge.

Recovering From Mess-Ups Sometimes, you mess up merges or rebases. Luckily, in git it is relatively straightforward to recover from such mistakes.

If you mess up during a rebase:

```
git rebase --abort
```

If you notice you messed up after the rebase:

```
# reset branch back to the saved point
git reset --hard tmp
```

If you forgot to make a backup branch:

```
# look at the reflog of the branch
git reflog show cool-feature

8630830 cool-feature@{0}: commit: BUG: io: close file handles immediately
278dd2a cool-feature@{1}: rebase finished: refs/heads/my-feature-branch onto 11ee694744f2552d
26aa21a cool-feature@{2}: commit: BUG: lib: make seek_gzip_factory not leak gzip obj
...
```

```
# reset the branch to where it was before the botched rebase
git reset --hard cool-feature@{2}
```

Rewriting Commit History

Note: Do this only for your own feature branches.

There's an embarassing typo in a commit you made? Or perhaps the you made several false starts you would like the posterity not to see.

This can be done via *interactive rebasing*.

Suppose that the commit history looks like this:

```
git log --oneline
eadc391 Fix some remaining bugs
a815645 Modify it so that it works
2dec1ac Fix a few bugs + disable
13d7934 First implementation
6ad92e5 * masked is now an instance of a new object, MaskedConstant
29001ed Add pre-nep for a copule of structured_array_extensions.
...
```

and `6ad92e5` is the last commit in the `cool-feature` branch. Suppose we want to make the following changes:

- Rewrite the commit message for `13d7934` to something more sensible.
- Combine the commits `2dec1ac`, `a815645`, `eadc391` into a single one.

We do as follows:

```
# make a backup of the current state
git branch tmp HEAD
# interactive rebase
git rebase -i 6ad92e5
```

This will open an editor with the following text in it:

```
pick 13d7934 First implementation
pick 2dec1ac Fix a few bugs + disable
pick a815645 Modify it so that it works
pick eadc391 Fix some remaining bugs

# Rebase 6ad92e5..eadc391 onto 6ad92e5
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
```

To achieve what we want, we will make the following changes to it:

```
r 13d7934 First implementation
pick 2dec1ac Fix a few bugs + disable
```

```
f a815645 Modify it so that it works
f eadc391 Fix some remaining bugs
```

This means that (i) we want to edit the commit message for `13d7934`, and (ii) collapse the last three commits into one. Now we save and quit the editor.

Git will then immediately bring up an editor for editing the commit message. After revising it, we get the output:

```
[detached HEAD 721fc64] FOO: First implementation
 2 files changed, 199 insertions(+), 66 deletions(-)
[detached HEAD 0f22701] Fix a few bugs + disable
 1 files changed, 79 insertions(+), 61 deletions(-)
Successfully rebased and updated refs/heads/my-feature-branch.
```

and the history looks now like this:

```
0f22701 Fix a few bugs + disable
721fc64 ENH: Sophisticated feature
6ad92e5 * masked is now an instance of a new object, MaskedConstant
```

If it went wrong, recovery is again possible as explained *above*.

Maintainer Workflow

This page is for maintainers — those of us who merge our own or other peoples' changes into the upstream repository.

Being as how you're a maintainer, you are completely on top of the basic stuff in *Development Workflow*.

The instructions in *Linking Your Repository to the Upstream Repo* add a remote that has read-only access to the upstream repo. Being a maintainer, you've got read-write access.

It's good to have your upstream remote have a scary name, to remind you that it's a read-write remote:

```
git remote add upstream-rw git@github.com:sagemath/sagenb.git
git fetch upstream-rw
```

Integrating Changes

Let's say you have some changes that need to go into trunk (`upstream-rw/master`).

The changes are in some branch that you are currently on. For example, you are looking at someone's changes like this:

```
git remote add someone git://github.com/someone/sagenb.git
git fetch someone
git branch cool-feature --track someone/cool-feature
git checkout cool-feature
```

So now you are on the branch with the changes to be incorporated upstream. The rest of this section assumes you are on this branch.

A Few Commits If there are only a few commits, consider rebasing to upstream:

```
# Fetch upstream changes
git fetch upstream-rw
# rebase
git rebase upstream-rw/master
```

Remember that, if you do a rebase, and push that, you'll have to close any github pull requests manually, because github will not be able to detect the changes have already been merged.

A Long Series of Commits If there are a longer series of related commits, consider a merge instead:

```
git fetch upstream-rw
git merge --no-ff upstream-rw/master
```

The merge will be detected by github, and should close any related pull requests automatically.

Note the `--no-ff` above. This forces git to make a merge commit, rather than doing a fast-forward, so that these set of commits branch off trunk then rejoin the main history with a merge, rather than appearing to have been made directly on top of trunk.

Check the History Now, in either case, you should check that the history is sensible and you have the right commits:

```
git log --oneline --graph
git log -p upstream-rw/master..
```

The first line above just shows the history in a compact way, with a text representation of the history graph. The second line shows the log of commits excluding those that can be reached from trunk (`upstream-rw/master`), and including those that can be reached from current HEAD (implied with the `..` at the end). So, it shows the commits unique to this branch compared to trunk. The `-p` option shows the diff for these commits in patch form.

Push to trunk

```
git push upstream-rw my-new-feature:master
```

This pushes the `my-new-feature` branch in this repository to the `master` branch in the `upstream-rw` repository.

CHAPTER

FIVE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

[WSblog] William Stein, How to Referee Sage Trac Tickets, http://sagemath.blogspot.com/2010/10/how-to-referee-sage-trac-tickets.html (Caveat: mercurial was replaced with git)

[SageComponents] See http://www.sagemath.org/links-components.html for a list

E

M

U