Developer Guide

Release 10.3

The Sage Development Team

Jul 03, 2024

CONTENTS

1 Table of Contents **3**

1.1 First Steps . . . 3
1.2 Working on GitHub . . . 8
1.3 Working with Git . . . 20
1.4 Writing Code for Sage . . . 36
1.5 Testing Sage . . . 57
1.6 Updating Sage Documentation . . . 111
1.7 More on Coding for Sage . . . 114
1.8 Packaging . . . 138

2 Indices and tables **167**

Bibliography **169**

Index **171**

Note: Sage development moved to GitHub in February 2023, from the Sage Trac server, which had been the center of Sage development for a long time. After the transition, this guide was updated accordingly. However, the legacy is still with us in many aspects of the current Sage development.

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 or create a new class
- Contribute a useful tutorial for a mathematical topic
- Translate an existing document to a new language
- Upgrade a package, create a fast new C library, etc.

This document tells you what you need to know to do all the above. We also discuss how to share your new and modified code with other Sage users around the globe.

To begin with, you need of course your own copy of Sage source code to change it. Use our Installation guide to get the source code and build Sage from source. If you have never worked on software before, pay close attention to the prerequisites to build on your platform.

Now here is a brief overview of this guide.

- *First Steps*: To share changes with the Sage community, you need to learn about revision control. We use the software Git for this purpose. Here we walk you through from setting up Git on your platform and to preparing a local branch to share with all Sage users.

  Note: As an easy way to get started, you can run and edit Sage's code and contribute your changes using Gitpod, a free online development environment based on VS Code. It will launch a pre-made workspace with all dependencies and tools installed so that you can start contributing straight away. Start by going to Gitpod, and read *our Gitpod guidelines* to learn more.

- *Working on GitHub*: All changes go through the Sage repository on GitHub at some point. It contains bug reports, enhancement proposals, changes in progress, and indeed all the history of Sage today. You have to be familiar with it to be involved in Sage development.
- *Working with Git*: Here we give an in-depth guide for working with Git for Sage development. Read this when you need help on Git in a tricky situation such as merge conflict.
- *Writing Code for Sage*: This is a guide on conventions in writing code and documentation. A beginning developer should read this to be a good developer. As conventions evolve over time, also experienced Sage contributors may want to review this chapter once in a while.
- *Testing Sage*: We value testing Sage highest. Every change of Sage source code has a risk to break Sage, and must be tested before being merged. This part explains our various tools to help test Sage.
- *Updating Sage Documentation*: All features of Sage are documented in our manuals. This part explains the technical aspect of updating Sage documentation.
- *More on Coding for Sage*: When you need to know the technical details of Sage for deep coding, read this.
- *Packaging*: Sage is composed of many third-party packages and its own distribution packages. This part is for advanced developers.

For more details, see the table of contents below. No matter where you start, good luck and welcome to Sage development!

CHAPTER

ONE

TABLE OF CONTENTS

1.1 First Steps

1.1.1 Development Walk-through

This section is a concise overview of the Sage development process. We will see how to make changes to the Sage source code and record them in the Git revision control system.

In the sections of the following chapter *Working on GitHub*, we will look at communicating these changes back to the Sage project. All changes to Sage source code have to go through the Sage repository on GitHub.

For examples, we assume your name Alice. Always replace it with your own name.

Checking Git

First, open a shell (for instance, Terminal on Mac) and check that Git works:

```
[alice@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 of them 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 *Installing Git* for instructions.

Because we also track who does what changes with Git, you must tell Git how you want to be known. Check if Git knows you:

```
[alice@localhost ~]$ git config --global user.name
Alice Adventure
[alice@localhost ~]$ git config --global user.email
alice@wonderland.com
```

If you have multiple computers, then use the same name on each of them. This name/email combination ends up in commits. So if it's not set yet, do it now before you forget! This only needs to be done once. See *Configuring Git* for instructions.

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 from scratch) download it from our Sage repository on GitHub:

```
[alice@localhost ~]$ git clone --origin upstream https://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 next need to switch to the develop branch (latest development release):

```
[alice@localhost ~]$ cd sage
[alice@localhost sage]$ git checkout develop
```

Next, build Sage, following the instruction in the file README.md in `SAGE_ROOT`. If all prerequisites to build are in place, the commands `./configure && make -j4` will do it. Additional details can be found in the section on installation from source in the Sage installation guide. If you wish to use conda-forge, see the section on conda.

Note: macOS allows changing directories without using exact capitalization. Beware of this convenience when compiling for macOS. Ignoring exact capitalization when changing into `SAGE_ROOT` can lead to build errors for dependencies requiring exact capitalization in path names.

Branching out

In order to start modifying Sage, we want to make a new *branch* in the local Sage repo. 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 push to your fork of the Sage repository on GitHub).

To begin with, type the command `git branch`. You will see the following:

```
[alice@localhost sage]$ git branch
* develop
  master
```

The asterisk shows you which branch you are on. Without an argument, the `git branch` command displays a list of all local branches with the current one marked by an asterisk.

It is easy to create a new branch. First make sure you are on the branch from which you want to branch out. That is, if you are not currently on the `develop` branch, type the command `git checkout develop`:

```
[alice@localhost sage]$ git checkout develop
Switched to branch 'develop'
Your branch is up-to-date with 'origin/develop'.
```

Then use the `git branch` command to create a new branch, as follows:

```
[alice@localhost sage]$ git branch last_twin_prime
```

Also note that `git branch` creates a new branch, but does not switch to it. For this, you have to use `git checkout`:

```
[alice@localhost sage]$ git checkout last_twin_prime
Switched to branch 'last_twin_prime'
```

Now if you use the command `git branch`, you will see the following:

```
[alice@localhost sage]$ git branch
  develop
* last_twin_prime
  master
```

Note that unless you explicitly push a branch to a remote Git repository, the branch is a local branch that is only 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 last_twin_prime develop` to create and switch to the new branch based on `develop` in one command.

The history

It is always a good idea to check that you are making your edits on the branch that you think you are on. The following command shows you the topmost commit in detail, including its changes to files:

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

To dig deeper, you can inspect the log:

```
[alice@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 tools are available to help you visualize the history tree better. For instance, `tig` is a very nice text-mode tool.

Editing the source code

Once you have your own branch, feel free to make any changes to source files as you like. The chapter *Writing Code for Sage* explains how your code should look like to fit into Sage, and how we ensure high code quality throughout.

The Git command `git status` is probably the most important of all Git commands. It tells you which files changed, and how to continue with recording the changes:

```
[alice@localhost sage]$ git status
On branch last_twin_prime
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)

```

(continues on next page)

(continued from previous page)

```
    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:

```
[alice@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:

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

to rebuild the Sage library and then start Sage.

Note: All changes to Python files take effect immediately after restarting Sage (unless you have used `./configure --disable-editable` when you built Sage). Hence you can just start Sage instead of `./sage -br` if only Python files were modified.

If you made changes to *third-party packages* installed as part of Sage, then you have to run

```
[alice@localhost sage]$ make build
```

as if you were installing Sage from scratch. However, this time only, the packages which were changed (or which depend on a changed package) will be rebuilt, so it should be much faster than building Sage the first time.

Note: If you have pulled a branch from the GitHub Sage repository, it may depend on changes to third-party packages, so `./sage -br` may fail. If this happens (and you believe the code in this branch should compile), try running `make build`.

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:

```
[alice@localhost sage]$ make distclean && make build
```

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

Note: If you switch between branches based on different releases, the timestamps of modified files will change. This triggers recythonization and recompilation of modified files on subsequent builds, whether or not you have made any additional changes to files. To minimize the impact of switching between branches, install ccache using the command

```
[alice@localhost sage]$ ./sage -i ccache
```

Recythonization will still occur when rebuilding, but the recompilation stage first checks whether previously compiled files are cached for reuse before compiling them again. This saves considerable time rebuilding.

Making commits

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.

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:

```
[alice@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)

[alice@localhost sage]$ git add src/sage/primes/last_pair.py
[alice@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:

```
[alice@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.

1.1.2 Setting Up Git

Installing Git

Depending on your platform, use the following to install Git:

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

Fedora
: Run `sudo yum install git-core`

Windows (WSL)
: We strongly recommend to install the package using the Linux distribution's package manager. Native Windows installations of Git may also work, but there are possible pitfalls.

macOS
: Install the Xcode Command Line Tools.

Some further resources for installation help are:

- Section 1.5 of the Git book
- The Git homepage for the most recent information
- Git install help page on GitHub

Configuring Git

Assuming your name `alice` and email address `alice@wonderland.com`,

```
[alice@localhost ~]$ git config --global user.name "Alice Adventure"
[alice@localhost ~]$ git config --global user.email alice@wonderland.com
```

This will write the settings into your Git configuration file `~/.gitconfig` with your name and email:

```
[user]
    name = Alice Adventure
    email = alice@wonderland.com
```

Of course, replace `Alice Adventure` and `alice@wonderland.com` with your actual name and email address.

This is the basic Git configuration for now. For further tips on configuring Git, see *Configuration tips*.

1.2 Working on GitHub

1.2.1 The Sage Repository on GitHub

The center of Sage development is the SageMath organization on GitHub, which consists of many repositories related with Sage. The most important one among them is of course the Sage repository, which we call "the Sage repo" for short.

Obtaining a GitHub account

To share your work on Sage, you need a GitHub account. If you do not have one yet, choose a username and create an account. In the following, we assume your username "alice". So you always read your own username if you see "alice".

Using the GitHub CLI

GitHub provides a command-line interface, the GitHub CLI, that can be used instead of the web interface. The central component of the GitHub CLI is the `gh` command that you can use in your terminal.

Installation

The page github_cli: Command-line interface for GitHub documents how to install the `gh` command for your platform. Or see GitHub CLI from GitHub.

Configuration

You have to authenticate to your GitHub account to allow `gh` command to interact with GitHub. Typically the authorization proceeds as follows:

```
[alice@localhost sage]$ gh auth login
? What is your preferred protocol for Git operations? HTTPS
? Authenticate Git with your GitHub credentials? Yes
? How would you like to authenticate GitHub CLI? Login with a web browser

! First copy your one-time code: 3DA8-5ADA
Press Enter to open github.com in your browser...
✓ Authentication complete.
- gh config set -h github.com git_protocol https
✓ Configured git protocol
✓ Logged in as sage
```

where a web browser is used to enter credentials. You can also use an authentication token instead, in which case you must first generate a Personal Access Token here.

Next set the default repo for the `gh` command:

```
[alice@localhost sage]$ gh repo set-default sagemath/sage
```

and check:

```
[alice@localhost sage]$ gh repo view
sagemath/sage
...
```

which will show the default repo along with its readme, which is quite long.

`gh` extensions

`gh` is extendable; e.g. a useful extension to `gh` allows testing of Sage's GitHub Actions locally, using Docker. It is called `act` and can be installed by running:

```
[alice@localhost sage]$ gh extension install https://github.com/nektos/gh-act
```

Append `--force` flag to the command above to force an upgrade of the extension. More details on configuring and using `gh act` are in *Testing on Multiple Platforms*.

Linking Git to your GitHub account

In order for your Git to work with GitHub, your GitHub account needs to be linked with your Git. No action is needed if you have already contributed to any other project on GitHub and set up Git credentials or SSH keys for this.

The above dialogue from `gh auth login` linked your Git to GitHub by HTTPS protocol using your GitHub credentials. Alternatively you may like to use popular Git Credential Manager which stores your credentials natively to your platform. For more information, see Caching your GitHub credentials in Git.

If you prefer SSH to HTTPS for authenticating Git to GitHub, then you follow *Git authentication through SSH* to generate an SSH keypair and add the SSH public key to your GitHub account. A simple way to upload the public key is to choose SSH protocol for Git operations in the dialogue from `gh auth login` command above. For more details, see Connecting to GitHub with SSH.

We assume HTTPS protocol in the rest of this guide.

Forking the Sage repository

The first step is to create your personal fork of the Sage repo on GitHub. After logging in to your GitHub account, visit the Sage repo https://github.com/sagemath/sage, and simply click "Fork" on the Sage repo. Then your fork of the Sage repo is created at https://github.com/alice/sage.

Next if you don't have a local Git repo of Sage, then start afresh cloning your fork:

```
[alice@localhost ~]$ git clone https://github.com/alice/sage.git
Cloning into 'sage'...
remote: Enumerating objects: 914565, done.
remote: Counting objects: 100% (2738/2738), done.
remote: Compressing objects: 100% (855/855), done.
remote: Total 914565 (delta 1950), reused 2493 (delta 1875), pack-reused 911827
Receiving objects: 100% (914565/914565), 331.09 MiB | 11.22 MiB/s, done.
Resolving deltas: 100% (725438/725438), done.
Updating files: 100% (9936/9936), done.
[alice@localhost ~]$ cd sage
[alice@localhost sage]$ git remote -v
origin  https://github.com/alice/sage.git (fetch)
origin  https://github.com/alice/sage.git (push)
```

```
[alice@localhost ~]$ git clone git@github.com:alice/sage.git
Cloning into 'sage'...
remote: Enumerating objects: 914565, done.
remote: Counting objects: 100% (2738/2738), done.
remote: Compressing objects: 100% (855/855), done.
remote: Total 914565 (delta 1950), reused 2493 (delta 1875), pack-reused 911827
Receiving objects: 100% (914565/914565), 331.09 MiB | 11.22 MiB/s, done.
Resolving deltas: 100% (725438/725438), done.
Updating files: 100% (9936/9936), done.
[alice@localhost ~]$ cd sage
[alice@localhost sage]$ git remote -v
origin  git@github.com:alice/sage.git (fetch)
origin  git@github.com:alice/sage.git (push)
```

If you already have a local Git repo and only want to link your fork as `origin` remote, then do:

```
[alice@localhost sage]$ git remote add origin https://github.com/alice/sage.git
[alice@localhost sage]$ git remote -v
origin  https://github.com/alice/sage.git (fetch)
```

(continues on next page)

(continued from previous page)

```
origin  https://github.com/alice/sage.git (push)
[alice@localhost sage]$ git fetch origin
remote: Enumerating objects: 1136, done.
remote: Counting objects: 100% (1084/1084), done.
remote: Compressing objects: 100% (308/308), done.
remote: Total 1136 (delta 825), reused 982 (delta 776), pack-reused 52
Receiving objects: 100% (1136/1136), 2.62 MiB | 5.30 MiB/s, done.
Resolving deltas: 100% (838/838), completed with 145 local objects.
From https://github.com/alice/sage
 * [new branch]      develop     -> origin/develop
```

```
[alice@localhost sage]$ git remote add origin git@github.com:alice/sage.git
[alice@localhost sage]$ git remote -v
origin  git@github.com:alice/sage.git (fetch)
origin  git@github.com:alice/sage.git (push)
[alice@localhost sage]$ git fetch origin
remote: Enumerating objects: 1136, done.
remote: Counting objects: 100% (1084/1084), done.
remote: Compressing objects: 100% (308/308), done.
remote: Total 1136 (delta 825), reused 982 (delta 776), pack-reused 52
Receiving objects: 100% (1136/1136), 2.62 MiB | 5.30 MiB/s, done.
Resolving deltas: 100% (838/838), completed with 145 local objects.
From git@github.com:alice/sage
 * [new branch]      develop     -> origin/develop
```

You also add the Sage repo `sagemath/sage` as your remote `upstream`:

```
[alice@localhost sage]$ git remote add upstream https://github.com/sagemath/sage.git
[alice@localhost sage]$ git remote -v
origin  https://github.com/alice/sage.git (fetch)
origin  https://github.com/alice/sage.git (push)
upstream    https://github.com/sagemath/sage.git (fetch)
upstream    https://github.com/sagemath/sage.git (push)
```

```
[alice@localhost sage]$ git remote add upstream git@github.com:sagemath/sage.git
[alice@localhost sage]$ git remote -v
origin  git@github.com:alice/sage.git (fetch)
origin  git@github.com:alice/sage.git (push)
upstream    git@github.com:sagemath/sage.git (fetch)
upstream    git@github.com:sagemath/sage.git (push)
```

To prevent accidental pushes to `upstream` (instead of `origin`), you may want to disable it by running:

```
[alice@localhost sage]$ git remote set-url --push upstream DISABLE
```

Of course, you can give arbitrary names to your Git remotes, but `origin` and `upstream` are the established defaults, which will make it easier to use tools such as the GitHub CLI.

Reporting bugs

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

- Search through our Google groups sage-devel, sage-support for postings related to your possible bug (it may have been fixed/reported already). You also search the GitHub issues to see if anyone else has already opened an issue about your bug.
- If you do not find anything but you are not sure that you have found a bug, ask about it on sage-devel.
- If you are sure that you have found a bug, then create on GitHub a new issue about the bug.

  A bug report should contain:

  - An explicit and **reproducible example** illustrating your bug (and/or the steps required to reproduce the buggy behavior). It also helps to describe what behaviour is expected.
  - The **version of Sage** you run, as well as the version of the optional packages that may be involved in the bug.
  - If relevant, describe your **operating system** as accurately as you can and the architecture of your CPU (32 bit, 64 bit, …).

Follow *Opening an issue* for further guide. Thank you in advance for reporting bugs to improve Sage!

Planning an enhancement

In addition to bug reports, you should also open an issue if you have some new code or an idea that makes Sage better. If you have a feature request, start a discussion on sage-devel first, and then if there seems to be a general agreement that you have a good idea, open an issue describing the idea.

Before opening a new issue, consider the following points:

- Make sure that nobody else has opened an issue (or a PR) about the same or closely related issue. Search through the existing issues and PRs with some key words.
- It is much better to open several specific issues than one that is very broad. Indeed, a single issue which deals with lots of different issues can be quite problematic, and should be avoided.
- Be precise: If foo does not work on macOS 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 issue must be solvable. For example, it would be silly to open an issue whose purpose was “Make Sage the best mathematical software in the world”. There is no metric to measure this properly and it is highly subjective.
- If appropriate, provide URLs to background information or sage-devel conversation relevant to the issue you are reporting.

Opening an issue

Whether it’s reporting a bug or planning an enhancement, issue should be opened on our Sage repo sagemath/sage on GitHub.

- Think of an apt title. People scan through the titles of issues to decide which ones to look into further. So write a title that concisely describes what the issue is about.
- Describe the issue in detail in the issue body. What is the issue? How can we solve the issue? Add links to relevant issues/PRs, and other resources.

  You may use GitHub mention `@USERNAME` to get attention from the people who would be interested in the issue or has expertise in this issue.

- Add appropriate labels to the created issue:
  - **Type** labels with prefix `t:` such as `t: bug`, `t: enhancement`, `t: feature`, `t: performance`, `t: refactoring`, `t: tests`
  - **Component** labels with prefix `c:` such as `c: basic arithmetic`, `c: linear algebra`, `c: geometry`, etc.
  - **Priority** labels with prefix `p:` such as `p: trivial / 5`, `p: minor / 4`, `p: major / 3`, `p: critical / 2`, and `p: blocker / 1`

  If the issue is not expected to be solved in the near future, you may add `wishlist item` label.

Creating a Pull Request

If you worked on an issue, and prepared a fix for a bug or wrote code for enhancing Sage, then you create a PR on the Sage repo sagemath/sage.

In addition to what were said about opening an issue, the following applies:

- The title should concisely describe what the PR does. If the PR solves an issue, describe briefly what the PR solves (do not simply put the issue number in the title).
- Explain what the PR solves in detail in the body. If the PR solves an issue, you may mention the issue here.
- Add type, component, and priority labels. If this PR solves an existing issue, please duplicate the labels of the issue to this PR.
- **Dependencies**: Use the phrase `- Depends on`, followed by the issue or PR reference. Repeat this in separate lines if there is more than one dependency. This format is understood by various dependency managers.

If you are working on a PR and the PR is not yet quite ready for review, then open the PR as draft.

The status of a PR

If a PR is in the state of draft, the review process does not start. Otherwise, review process will start for the PR as soon as a reviewer gets interested with the PR, and the status of the PR will be indicated by **status** labels with prefix `s:`.

- `s: needs review`: The code is ready to be peer-reviewed. If the code is not yours, then you can review it. See *Reviewing Code*.
- `s: needs work`: Something needs to be changed in the code. The reason should appear in the comments.
- `s: needs info`: The author of the PR or someone else should answer to a question or provide information to proceed the review process.
- `s: positive review`: The PR has been reviewed positively, and the release manager will merge it to the `develop` branch of the Sage repo in due time.

If the PR does not get positive review and it is decided to close the PR, then the PR will get one of **resolution** labels: `r: duplicate`, `r: invalid`, `r: wontfix`, `r: worksforme`.

The stopgap

When Sage returns wrong results, an issue and a PR should be created:

- A stopgap issue with all available details.
- A stopgap PR (e.g. github issue #12699)

The stopgap PR does not fix the problem but adds a warning that will be printed whenever anyone uses the relevant code, 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.",
    ISSUE_NUM)
```

Replace `ISSUE_NUM` by the reference number for the stopgap issue. On the stopgap issue, enter the reference number for the stopgap PR. Stopgap issues and PRs should be marked as critical.

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.

Commenting issues and PRs

Anyone can comment on an issue or a PR. If a PR is linked to an issue, you may not be sure where the comment should go. Then

- Comments on the reported issue should go on the issue.
- Comments on the submitted code should go on the PR.

Checks on PRs

If you manage to fix a bug or enhance Sage, you are our hero. See *Development Walk-through* for making changes to the Sage source code and *Creating a Pull Request* to create a PR for the changes.

For each push to a PR, automated tests for the branch of the PR run on GitHub Actions.

- A linting workflow checks that the code of the current branch adheres to the style guidelines using *Pycodestyle* (in the `pycodestyle-minimal` configuration) and *Relint*.

  In order to see details when it fails, you can click on the check and then select the most recent workflow run.

- The build and test workflow on GitHub Actions builds Sage for the current branch (incrementally on top of an installation of the `develop` branch) and runs the test. Details are again available by clicking on the check.

  The automatic workflow runs on a container based on `ubuntu-focal-standard`. To request a run of the workflow on a different platform, you can issue a workflow dispatch. You can select any of the platforms for which a prebuilt container image exists.

- The build documentation workflow on GitHub Actions builds the HTML documentation for the current branch.

  A link to the built doc is added in a comment, and so you can easily inspect changes to the documentation without the need to locally rebuild the docs yourself.

  If the doc build fails, you can go to Actions tab and examine documentation build workflow and choose the particular branch to see what went wrong.

Final notes

- Every bug fixed should result in a doctest.
- There are many enhancements possible for Sage and too few developers to implement all the good ideas.
- If you are a developer, be nice and try to solve a stale/old issue 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 PRs.

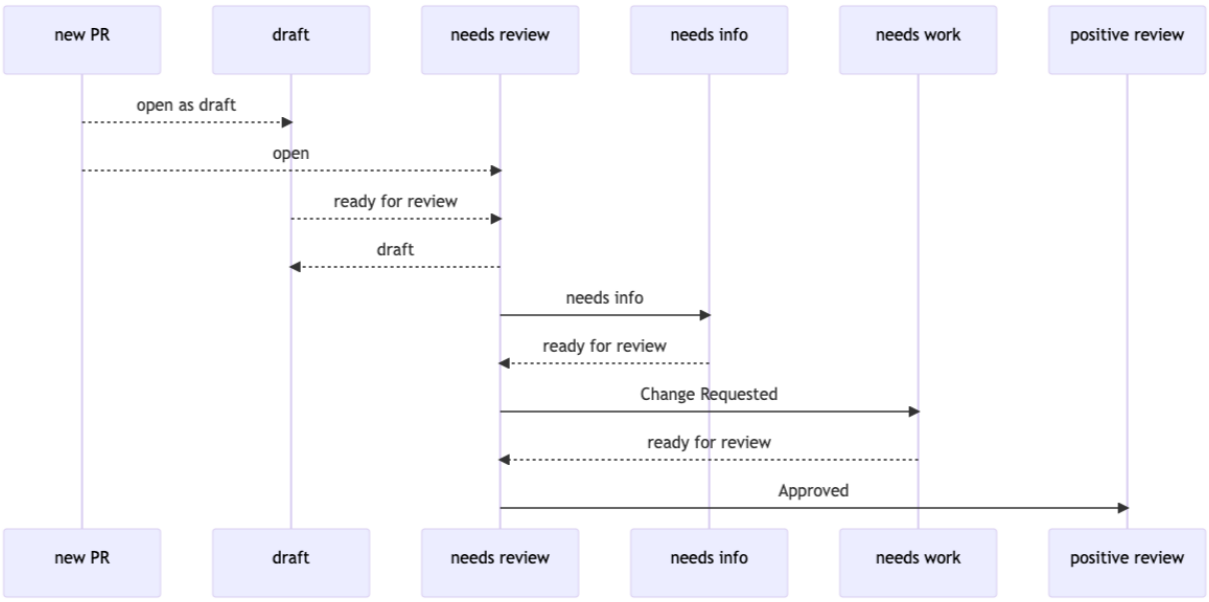
1.2.2 Using Git with GitHub

We continue our introduction to Sage development from *Development Walk-through*. We discuss how to push your local changes to your fork of the GitHub Sage repository so that your changes can be reviewed for inclusion in Sage.

Before proceeding, check that you have `origin` and `upstream` remotes right:

```
[alice@localhost sage]$ git remote -v
origin  https://github.com/alice/sage.git (fetch)
origin  https://github.com/alice/sage.git (push)
upstream    https://github.com/sagemath/sage.git (fetch)
upstream    https://github.com/sagemath/sage.git (push)
```

Development workflow at a glance



1. Alice creates a *new local branch* and *commits* changes to the Sage source files.
2. Alice pushes the local branch to the remote `origin`, her fork of the Sage repo on GitHub, and with it *creates a PR* to the Sage repo. When ready, Alice sets the PR to `needs review` status.
3. Bob, a developer acting as reviewer, *examines the PR*, looks through the changes, leaves comments on the PR, and requests fixes (`needs work`).

4. Alice makes more commits on top of her local branch, and pushes the new commits to the remote `origin`. These new commits are reflected in the PR.
5. Bob looks through the changes in the new commits and reviews the changes.
6. After a few of iterations of commenting and fixing, finally the reviewer Bob is satisfied, and then he approves the PR and sets it to `positive review` status.

Creating a new PR

Suppose you have written an algorithm for calculating the last twin prime, committed the code to a local branch based upon `develop` branch. Now you want to add it to Sage. You would first open a PR for that:

```
[alice@localhost sage]$ gh pr create
? Where should we push the 'last-twin-prime' branch? user/sage

Creating pull request for user:last-twin-prime into develop in sagemath/sage

? Title Last twin prime
? Choose a template PULL_REQUEST_TEMPLATE.md
? Body <Received>
? What's next? Submit as draft
https://github.com/sagemath/sage/pull/12345
```

This will create a new PR titled "Last twin prime" in the Sage repo for the branch pushed to your fork `alice/sage` from the local branch on your desktop. The title is automatically derived from the last commit title. If you don't like this, then you can use the `-t` switch to specify it explicitly. See the manual page of the command gh pr create for details.

If you did not provide enough details about the PR at the prompts, you may want to edit the PR further via the web interface.

Checking out an existing PR

If you want to base your work on an existing PR or want to review the code of a PR, then you would run:

```
[alice@localhost sage]$ gh pr checkout 12345
remote: Enumerating objects: 7, done.
remote: Counting objects: 100% (7/7), done.
remote: Compressing objects: 100% (7/7), done.
remote: Total 7 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (7/7), 25.50 KiB | 2.83 MiB/s, done.
From https://github.com/sagemath/sage
 * [new ref]               refs/pull/12345/head -> last-twin-prime
Switched to branch 'last-twin-prime'
```

The command `gh pr checkout` downloads the branch of the PR. Just like the `create` command, you can specify the local branch name explicitly using the `-b` switch if you want.

Uploading more changes to GitHub

Once you have created a PR, edit the appropriate files and commit your changes to your local branch as described in *Editing the source code* and *Making commits*.

If you are ready to share the changes up to now, upload your new commits to your fork by:

```
[alice@localhost sage]$ git push origin
Enumerating objects: 13, done.
Counting objects: 100% (13/13), done.
Delta compression using up to 12 threads
Compressing objects: 100% (7/7), done.
Writing objects: 100% (7/7), 1.98 KiB | 1.98 MiB/s, done.
Total 7 (delta 6), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (6/6), completed with 6 local objects.
To https://github.com/alice/sage.git
 + 352d842907...56ffdab967 last-twin-prime -> last-twin-prime
```

Note that you do not push the branch to the remote `upstream` the Sage repo. Instead the new commits pushed to the remote `origin` are shown in the PR at the Sage repo.

Finishing it up

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

Once you are happy with the changes you pushed, they must be reviewed by someone else before they can be included in the next release of Sage. To mark your PR as ready for review, you should set it to `needs review` status.

Merging the upstream develop branch

It commonly happens that `develop` branch at the remote `upstream` was updated and you need to merge the upstream changes to your local branch. Then you do:

```
[alice@localhost sage]$ git fetch upstream develop:develop
```

This fast-forwards your local `develop` branch to the upstream `develop` branch.

Now you go back to your working branch and merge the updated `develop` branch:

```
[alice@localhost sage]$ git merge develop
....
```

If there was no upstream change conflicting with the changes you made locally, this merge operation will finish cleanly. Otherwise, you are in *merge conflict*. This rarely happens since Git is smart in merging changes. However, once merge conflict occurs, you have to manually resolve the conflicts. The conflict resolving procedure is explained in *Conflict resolution*.

1.2.3 Reviewing Code

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 PR. This document lists things that the reviewer must check before deciding that a PR is ready for inclusion into Sage.

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

Check the GitHub checks: We require all checks have passed.

Read the diff: Click "Files changed" tab of the PR. Read through the changes of all modified files. We use pull request reviews. You can add comments directly to changed lines.

Build the code: (This is optional if the **build and test** check has passed.) While you read the code, you can *rebuild Sage with the new code*. If you do not know how to **download the code**, *see here*.

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

- **The purpose**: Does the code address the PR'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 conventions*? *Cython's conventions*?
- **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 PR contains a bugfix, does it add a doctest illustrating that the bug has been fixed? This new doctest should contain the issue or PR number, for example `See :issue:`12345``.
- **Speedup**: Can the PR make any existing code slower? if the PR claims to speed up some computation, does the PR contain code examples to illustrate the claim? The PR should explain how the speedup is achieved.
- **Build the manuals**: Does the reference manual build without errors (check both html and pdf)? See *The Sage Manuals* to learn how to build the manuals.
- **Look at the manuals**: Does the reference manual look okay? The changes may have typos that allow the documentation to build without apparent errors but that may cause badly formatted output or broken hyperlinks.
- **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 PR's status (see *The status of a PR*):

- **positive review**: If the answers to the questions above and other reasonable questions are *"yes"*, you can set the PR to `positive review` status.
- **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 PR's status to `needs work` status.
- **needs info**: If something is not clear to you and prevents you from going further with the review, ask your question and set the PR's status to `needs info` status.

- 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.

For more advice on reviewing, see How to Referee Sage Trac Tickets (caveat: mercurial was replaced with Git and Trac with GitHub).

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 implementation is mathematically correct. Please refrain from additional feature requests or open-ended discussion about alternative implementations. If you want the code written differently, your suggestion should be a clear and actionable request.

Reviewing and closing PRs

PRs can be closed when they have positive review or for other reasons.

If a PR is closed for a reason other than positive review, use one of the **resolution** labels `r: duplicate`, `r: invalid`, `r: wontfix`, and `r: worksforme`. Add a comment explaining why the issue has been closed if that's not already clear from the discussion.

If you think an issue has been prematurely be closed, feel free to reopen it.

Reasons to invalidate PRs

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

No Patch Bombs: Code that goes into Sage is peer-reviewed. If you show up with 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 PRs that are Sage specific, you hopefully will understand why.

No Support Discussions: GitHub is not meant to be a system to track down problems when using Sage. An issue 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 PR.

Solution Must Be Achievable: Issues must be achievable. Many times, issues 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.

The release process

It is good for developers and reviewers to be aware of the procedure that the Sage Release Manager uses to make releases. Here it is as of 2023:

Beta Release Stage: For preparing a new beta release or the first release candidate, all positively reviewed PRs with the forthcoming release milestone are considered. PRs that have dependencies not merged yet are ignored. The Release Manager merges PRs in batches of 10 to 20 PRs, taking the PR priority into account. If a merge conflict of a PR to the Release Manager's branch occurs, the PR is set back to "needs work" status by the Release Manager, and the list of the PRs already merged to the Release Manager's branch is posted. The author of the PR needs to identify conflicting PRs in the list, make merge commits and declare them as dependencies, before setting back to "positive review" status. Each batch of merged PRs then undergoes integration testing. If problems are detected, a PR will be set back to "needs work" status and unmerged. When a batch of PRs is ready, the Release Manager closes these PRs and proceeds to the next batch. After a few batches, a new beta release is tagged, pushed to the `develop` branch on the Sage repository on GitHub, and announced on `sage-release`.

Release Candidate Stage: After the first release candidate has been made, the project is in the release candidate stage, and a modified procedure is used. Now **only PRs with a priority set to "blocker" are considered**. PRs with all other priorities, including "critical", are ignored. Hence if a ticket is important enough to merit inclusion in this stage, it should be set to "blocker" by adding `p: blocker / 1` label.

Blocker PRs: The goal of the release process is to make a stable release of high quality. Be aware that there is a risk/benefit trade-off in merging a PR. The benefit of merging a PR is the improvement that the PR brings, such as fixing a bug. However, any code change has a risk of introducing unforeseen new problems and thus delaying the release: If a new issue triggers another release candidate, it delays the release by 1-2 weeks. Hence developers should use "blocker" priority sparingly and should indicate the rationale on the PR. Though there is no one fixed rule or authority that determines what is appropriate for "blocker" status,

- PRs introducing new features are usually not blockers – unless perhaps they round out a set of features that were the focus of development of this release cycle.
- PRs that make big changes to the code, for example refactoring PRs, are usually not blockers.

Final Release: If there is no blocker PR for the last release candidate, the Release Manager turns it to the final release. It is tagged with the release milestone, and announced on `sage-release`.

1.3 Working with Git

1.3.1 Git Basics

Git is a tool to exchange commits (file changes) and branches (organized of commits) with other developers.

As a distributed revision control system, Git does not have the notion of a central server. However, for Sage development, Git communicates with other developers via the Sage repository on GitHub. Hence we assume that throughout this guide.

Git authentication through SSH

In order to push changes securely to a remote repository, Git uses public-key cryptography. This section will show you how to set up the necessary cryptographic keys for the case that you want to use SSH(Secure Shell) protocol to authenticate your Git to GitHub, instead of HTTPS protocol.

Generating your SSH keys

Check whether you already have suitable SSH keys by inspecting `.ssh` directory in your home directory. If you don't have suitable SSH keys yet, you can create a key pair with the `ssh-keygen` tool.

Follow either the detailed instructions or the following brief instructions:

```
[alice@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 alice@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.

Adding your public key for authentication to GitHub

Follow the procedure Adding a new SSH key to your GitHub account. Then check that it works by:

```
[alice@localhost sage]$ git remote add origin git@github.com:alice/sage.git
[alice@localhost sage]$ git remote -v
origin  git@github.com:alice/sage.git (fetch)
origin  git@github.com:alice/sage.git (push)
```

Pushing your changes to a remote

Push your branch to the remote `origin` with either

```
[alice@localhost sage]$ git push --set-upstream origin HEAD:my_branch
```

or

```
[alice@localhost sage]$ git push origin HEAD:my_branch
```

if your branch already has an upstream branch. Here "upstream" means the the remote `origin`, which is *upstream* to your local Git repo.

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.

Checking out a PR

If you want to work with the changes of a PR branch, you must make a local copy of the branch. 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 branch into your local repository. This is achieved by:

```
[alice@localhost sage]$ git fetch upstream pull/12345/head
remote: Enumerating objects: 12, done.
remote: Counting objects: 100% (12/12), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 12 (delta 9), reused 11 (delta 9), pack-reused 0
Unpacking objects: 100% (12/12), 2.22 KiB | 206.00 KiB/s, done.
From https://github.com/sagemath/sage
 * branch                  refs/pull/12345/head -> FETCH_HEAD
```

The `pull/12345/head` branch refers to the branch of the PR #12345 of the remote `upstream`. The 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:

```
[alice@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 the branch. You can now edit files and commit changes to your local branch.

Getting changes from a remote

A common task during development is to synchronize your local copy of the branch with the branch on the GitHub Sage repo. In particular, assume you downloaded the branch of a PR made by someone else, say Bob, and made some suggestions for improvements on the PR. Now Bob incorporated your suggestions into his branch, and you want to get the added changes to complete your review. Assuming that you originally got your local branch as in *Checking out a PR*, you can just issue:

```
[bob@localhost sage]$ git pull upstream pull/12345/head
From https://github.com/sagemath/sage
 * branch                  refs/pull/35608/head -> FETCH_HEAD
Merge made by the 'ort' strategy.
 src/doc/common/python3.inv          | Bin 98082 -> 131309 bytes
 src/doc/common/update-python-inv.sh |   7 +++++--
 2 files changed, 4 insertions(+), 3 deletions(-)
```

This command downloads the changes from the branch of the PR and merges them into your local branch.

Updating develop

The `develop` branch can be updated just like any other branch. However, your local copy of the develop branch should stay **identical** to the GitHub Sage repo develop branch.

If you accidentally added commits to your local copy of `develop`, 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:

```
[alice@localhost sage]$ git checkout develop
[alice@localhost sage]$ git pull --ff-only upstream develop
```

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:

```
[alice@localhost sage]$ git checkout develop
[alice@localhost sage]$ git reset --hard upstream/develop
```

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 `develop` (see *Updating develop*) and now your Git history looks like this (see *The history*):

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

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 `develop`. This is called **rebase**, and it rewrites your current branch:

  ```
  git checkout my_branch
  git rebase -i develop
  ```

In terms of the commit graph, this results in:

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

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 `develop` at the same time (see *Getting changes from a remote*):

```
git checkout my_branch
git pull -r develop
```

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

```
git checkout my_branch
git merge develop
```

The result is the following commit graph:

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

  - **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 `develop` at the same time (see *Getting changes from a remote*):

```
git checkout my_branch
git pull develop
```

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.

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:

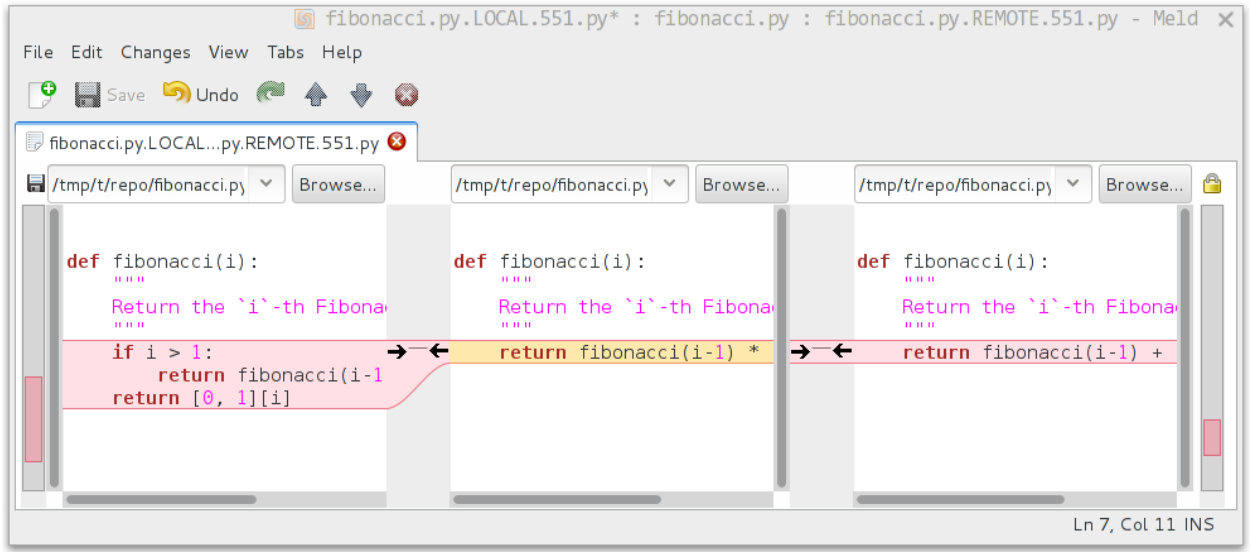
```
[alice@laptop]$ git mergetool

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':
```

(continues on next page)

(continued from previous page)

```
  {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.

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. Bob corrected 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 turned those changes into a new commit:

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

He made his changes a PR to the GitHub Sage repo and got it merged to the `develop` branch. His `fibonacci` function is not yet perfect but is certainly better than the original.

Meanwhile, Alice changed 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 merged her branch with the latest `develop` branch fetched from the GitHub Sage repo:

```
[alice@home sage]$ git add fibonacci.py
[alice@home sage]$ git commit -m 'corrected recursion formula, must be + instead of *'
[alice@home sage]$ git fetch upstream develop:develop
[alice@home sage]$ git merge develop
...
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
    return fibonacci(i-1) + fibonacci(i-2)
=======
    if i > 1:
        return fibonacci(i-1) * fibonacci(i-2)
    return [0, 1][i]
>>>>>>> 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 the GitHub Sage repo:

```
[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
```

(continues on next page)

(continued from previous page)

```
|\
| * 41675df corrected recursion formula, must be + instead of *
* | 14ae1d3 return correct seed values
|/
* 14afe53 initial commit
[alice@laptop sage]$ git push origin
```

This time, there is no merge conflict since Alice's branch already merged the `develop` branch.

1.3.2 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 *Development Walk-through* and *Git Basics* instead.

Detached heads and reviewing PRs

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:

```
[alice@localhost sage]$ git checkout f9a0d54099d758ccec731a38929902b2b9d0b988
Note: switching to 'f9a0d54099d758ccec731a38929902b2b9d0b988'.

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 switching back to a branch.

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

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at f9a0d54099 Fix a slow doctest in matrix_integer_dense_hnf.py
```

If it is not stored in your local Git repository, you need to download it from the `upstream` repo first:

```
[alice@localhost sage]$ git fetch upstream f9a0d54099d758ccec731a38929902b2b9d0b988
From https://github.com/sagemath/sage
 * branch                  f9a0d54099d758ccec731a38929902b2b9d0b988 -> FETCH_HEAD
[alice@localhost sage]$ git checkout FETCH_HEAD
HEAD is now at f9a0d54099 Fix a slow doctest in matrix_integer_dense_hnf.py
```

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

```
[alice@localhost sage]$ git status
HEAD detached at f9a0d54099
nothing to commit, working tree 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 PRs. Just check out the commit (look at the "Commits" tab of the PR) 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.

Update branch to latest Sage version

- You have a compiled and working new SageMath version `n`, and
- you want to work on a branch `some_code` which is based on some old SageMath version `o`
- by updating this branch from version `o` to `n`
- with only recompiling changed files (and not all touched files from `o` to `n`),
- then continue reading this section.

Introduction

When developing, quite frequently one ends up with a branch which is not based on the latest (beta) version of SageMath.

Note: Continue working on a feature based on an old branch is perfectly fine and usually there is no need to merge in this latest SageMath version.

However sometimes there is a need for a merge, for example

- if there are conflicts with the latest version or
- one needs a recent feature or
- simply because the old SageMath version is not available on your machine any longer.

Then merging in the latest SageMath version has to be done.

Merge in the latest Sage version

(This is the easy way without minimizing the recompilation time.)

Suppose we are on our current working branch `some_code` (branch is checked out). Then

```
git merge develop
```

does the merging, i.e. we merge the latest development version into our working branch.

However, after this merge, we need to (partially) recompile SageMath. Sometimes this can take ages (as many files are touched and their timestamps are renewed) and there is a way to avoid it.

Minimize the recompilation time

Suppose we are on some new SageMath (e.g. on branch `develop`) which was already compiled and runs successfully, and we have an "old" branch `some_code`, that we want to bring onto this SageMath version (without triggering unnecessary recompilations).

We first create a new working tree in a directory `new_worktree` and switch to this directory:

```
[alice@localhost sage]$ git worktree add new_worktree
[alice@localhost sage]$ cd new_worktree
```

Here we have a new copy of our source files. Thus no timestamps etc. of the original repository will be changed. Now we do the merge:

```
[alice@localhost sage/new_worktree]$ git checkout some_code
[alice@localhost sage/new_worktree]$ git merge develop
```

And go back to our original repository:

```
[alice@localhost sage/new_worktree]$ git checkout develop
[alice@localhost sage/new_worktree]$ cd ..
```

We can now safely checkout `some_code`:

```
[alice@localhost sage]$ git checkout some_code
```

We still need to call

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

but only changed files will be recompiled.

To remove the new working tree simply use

```
[alice@localhost sage]$ rm -r new_worktree
```

Why not merging the other way round?

Being on some new SageMath (e.g. on branch `develop`) which runs successfully, it would be possible to merge in our branch `some_code` into develop. This would produce the same source files and avoid unnecessary recompilations. However, it makes reading Git's history very unpleasant: For example, it is hard to keep track of changes etc., as one cannot simply pursue the first parent of each Git commit (`git log --first-parent`).

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 `develop` branch:

```
[alice@localhost sage]$ git checkout develop
```

As long as you did not make any changes to the `develop` 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:

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

    Commit message
...
[alice@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:

```
[alice@localhost sage]$ git reset --hard HEAD~5
[alice@localhost sage]$ git rm sage
[alice@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:

```
[alice@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:

```
[alice@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 a public repo.

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`

```
[alice@localhost sage]$ git log --oneline
9621dae added file C
7873447 added file B
```

(continues on next page)

(continued from previous page)

```
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:

```
[alice@localhost sage]$ 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
[alice@localhost sage]$ git checkout -b second_branch
Switched to a new branch 'second_branch'
[alice@localhost sage]$ git branch
  first_branch
* second_branch
[alice@localhost sage]$ git log --oneline
5b5588e base commit
```

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

```
[alice@localhost sage]$ git cherry-pick 7873447
[second_branch 758522b] added file B
 1 file changed, 1 insertion(+)
 create mode 100644 file_B.py
[alice@localhost sage]$ 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:

```
[alice@localhost sage]$ git checkout first_branch
Switched to branch 'first_branch'
[alice@localhost sage]$ 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:

```
[alice@localhost sage]$ git cherry-pick 9621dae
[first_branch 5844535] added file C
 1 file changed, 1 insertion(+)
 create mode 100644 file_C.py
[alice@localhost sage]$ git log --oneline
5844535 added file C
```

(continues on next page)

(continued from previous page)

```
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:

```
[alice@localhost sage]$ git log --oneline
9621dae added file C
7873447 added file B
bf817a5 added file A
5b5588e base commit
[alice@localhost sage]$ git checkout -b second_branch
Switched to a new branch 'second_branch'
[alice@localhost sage]$ 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.

1.3.3 Git Tips and References

This chapter contains additional material about the Git revision control system. 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 = Alice Adventure
    email = alice@wonderland.com

[core]
    editor = emacs
```

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

```
[alice@localhost ~]$ git config --global user.name "Alice Adventure"
[alice@localhost ~]$ git config --global user.email alice@wonderland.com
[alice@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:

```
[alice@localhost ~]$ git config --global alias.ci "commit -a"
[alice@localhost ~]$ git config --global alias.co checkout
[alice@localhost ~]$ git config --global alias.st "status -a"
[alice@localhost ~]$ git config --global alias.stat "status -a"
[alice@localhost ~]$ git config --global alias.br branch
[alice@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:

```
[alice@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:

```
[alice@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)[%an]%Creset' --abbrev-commit --date=relative
```

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

```
[alice@localhost ~]$ git lg
* 6d8e1ee - (HEAD, origin/my-fancy-feature, my-fancy-feature) NF - a fancy file (45␣
↪minutes ago) [Matthew Brett]
*   d304a73 - (origin/placeholder, placeholder) Merge pull request #48 from hhuuggoo/
↪master (2 weeks ago) [Jonathan Terhorst]
|\
| * 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 Webster]
* 68f6752 - Initial implementation of AxisIndexer - uses 'index_by' which needs to be␣
↪changed to a call on an Axes object - this is all very sketchy right now. (2 weeks␣
↪ago) [Corr
*   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 weeks ago) [Jonathan Terhorst]
| * 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

- gittutorial is an introductory tutorial from Git project.
- Git magic is an extended introduction with intermediate detail.
- The Git parable is an easy read explaining the concepts behind Git.
- 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 Kit has an excellent series of tutorials as well as videos and screencasts.
- Git ready is a nice series of tutorials.
- A good but technical page on Git concepts

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 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

1.4 Writing Code for Sage

1.4.1 Setting up Your Workspace

Text editors and IDEs for use with Sage

In meta github issue #30500 we are collecting instructions how to configure various text editors and IDEs for use with Sage.

Gitpod

Gitpod is a free service that will let you build and run Sage from an online development environment based on VS Code. Without needing to install anything on your computer, Gitpod creates a virtual fully-functional workspace with all the dependencies and tools pre-installed.

To get started, go to Gitpod and log in using your GitHub or GitLab account. Wait while Gitpod creates a workspace. The first time, it may take some time to build Sage.

You can now run and edit Sage's code. Contributing your changes follows the normal *Git workflow*. For this to work, you first have to authorize Gitpod with GitHub:

1. In the running Gitpod workspace, generate a new SSH key pair by `ssh-keygen -f tempkey`.
2. Save the private key as a secure environment variable in Gitpod using `gp env PRIVATE_SSH_KEY="$(<tempkey)"`, or by using the Gitpod UI.
3. Register the public key with GitHub following the instructions in *Generating your SSH keys*.
4. Close this Gitpod workspace.

After following this procedure, every new Gitpod workspace will have a working `origin` remote to which you can push your changes.

You can also use your VS Code Desktop to keep your local IDE configuration while still benefiting from Gitpod's high-spec servers and automated prebuilds.

1.4.2 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.

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

Python code style

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

- **PEP 0008**
- **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)]
```

- 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():
    def __init__(self, x):
    self._x  = 1
```

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

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

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
        pkgs/     # install, patch, and metadata from spkgs
    src/
        setup.py
        ...
        sage/            # Sage library
            ext_data/    # extra Sage resources (formerly src/ext)
        bin/             # the scripts in local/bin that are tracked
    upstream/            # tarballs of upstream sources
    local/               # installed binaries
```

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

If you want to create a new directory (package) in the Sage library `SAGE_ROOT/src/sage` (say, `measure_theory`), that directory will usually contain an empty file `__init__.py`, which marks the directory as an ordinary package (see *Ordinary packages vs. implicit namespace packages*), and also a file `all.py`, listing imports from this package that are user-facing and important enough to be in the global namespace of Sage 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_measure', '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 *
```

Adding new top-level packages below `sage` should be done sparingly. It is often better to create subpackages of existing packages.

Non-Python Sage source code and supporting files can be included in one of the following places:

- In the directory of the Python code that uses that file. When the Sage library is installed, the file will be installed in the same location as the Python code. For example, `SAGE_ROOT/src/sage/interfaces/maxima.py` needs to use the file `SAGE_ROOT/src/sage/interfaces/maxima.lisp` at runtime, so it refers to it as

  ```
  os.path.join(os.path.dirname(__file__), 'sage-maxima.lisp')
  ```

- In an appropriate subdirectory of `SAGE_ROOT/src/sage/ext_data/`. (At runtime, it is then available in the directory indicated by `SAGE_EXTCODE`). For example, if `file` is placed in `SAGE_ROOT/src/sage/ext_data/directory/` it can be accessed with

```
from sage.env import SAGE_EXTCODE
file = os.path.join(SAGE_EXTCODE, 'directory', 'file')
```

In both cases, the files must be listed (explicitly or via wildcards) in the section `options.package_data` of the file `SAGE_ROOT/pkgs/sagemath-standard/setup.cfg.m4` (or the corresponding file of another distribution).

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.

Headings of Sage library code files

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

```
r"""
<Short one-line summary that ends with no period>

<Paragraph description>

EXAMPLES::

<Lots and lots of examples>

AUTHORS:

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

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

"""

# ****************************************************************************
#       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.
#                  https://www.gnu.org/licenses/
# ****************************************************************************
```

As an example, see `SAGE_ROOT/src/sage/rings/integer.pyx`, which contains the implementation for **Z**. The names of the people who made major contributions to the file appear in the `AUTHORS` section. You can add your name to the list if you belong to the people, but refrain from being verbose in the description. The `AUTHORS` section shows very rough overview of the history, especially if a lot of people have been working on that source file. The authoritative log for who wrote what is always the git repository (see the output of `git blame`).

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

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 …".

  For methods of a class, it is recommended to refer to the `self` argument in a descriptive way, unless this leads to a confusion. For example, if `self` is an integer, then `this integer` or `the integer` is more descriptive, and it is preferable to write

```
Return whether this integer is prime.
```

- A **longer description**.

  This is optional if the one-sentence description does not need more explanations.

  Start with assumptions of the object, if there are any. For example,

```
The poset is expected to be ranked.
```

  if the function raises an exception when called on a non-ranked poset.

  Define your terms

```
The lexicographic product of `G` and `H` is the graph with vertex set ...
```

  and mention possible aliases

```
The tensor product is also known as the categorical product and ...
```

- An **INPUT** and an **OUTPUT** block describing the input/output of the function.

  The INPUT block describes all arguments that the function accepts.

  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.

```
INPUT:

- ``n`` -- integer

- ``p`` -- prime integer (default: `2`); coprime with ``n``
```

  The OUTPUT block describes the expected output. This is required if the one-sentence description of the function needs more explanation.

```
OUTPUT: the plaintext decrypted from the ciphertext ``C``
```

It is often the case that the output consists of several items.

```
OUTPUT: a tuple of

- the reduced echelon form `H` of the matrix `A`

- the transformation matrix `U` such that `UA = H`
```

You are recommended to be verbose enough for complicated outputs.

```
OUTPUT:

The decomposition of the free module on which this matrix `A` acts from
the right (i.e., the action is `x` goes to `xA`), along with whether
this matrix acts irreducibly on each factor. The factors are guaranteed
to be sorted in the same way as the corresponding factors of the
characteristic polynomial.
```

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

  These examples are used for documentation, but they are also tested before each release just like TESTS block.

  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 interest 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 links 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`.
```

  The bibliographical reference should go in Sage's master bibliography file, `SAGE_ROOT/src/doc/en/reference/references/index.rst`:

```
.. [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).

  Almost all bibliographic information should be put in the master bibliography file, see below. Citations will then link to the master bibliography where the reader can find the bibliographic details (see below for citation syntax). REFERENCE blocks in individual docstrings are therefore usually not necessary.

  Nevertheless, a REFERENCE block can be useful if there are relevant sources which are not explicitly mentioned in the docstring or if the docstring is particularly long. In that case, add the bibliographic information to the master bibliography file, if not already present, and add a reference block to your docstring as follows:

```
REFERENCES:

For more information, see [Str1969]_, or one of the following references:
```

(continues on next page)

(continued from previous page)

```
- [Sto2000]_

- [Voe2003]_
```

Note the trailing underscores which makes the citations into hyperlinks. See below for more about the master bibliography file. For more about citations, see the Sphinx/reST markup for citations. For links to GitHub issues and PRs or wikipedia, see *Hyperlinks*.

- A **TESTS** block (highly recommended).

  Formatted just like EXAMPLES, containing tests that are not relevant to users. In particular, these blocks are not shown when users ask for help via `foo?`: they are stripped by the function `sage.misc.sagedoc.skip_TESTS_block()`.

  Special and corner cases, like number zero, one-element group etc. should usually go to this block. This is also right place for most tests of input validation; for example if the function accepts `direction='up'` and `direction='down'`, you can use this block to check that `direction='junk'` raises an exception.

  For the purposes of removal, A "TESTS" block is a block starting with "TESTS:" (or the same with two colons), on a line on its own, and ending either with a line indented less than "TESTS", or with a line with the same level of indentation – not more – matching one of the following:

  – a Sphinx directive of the form ".. foo:", optionally followed by other text.

  – text of the form "UPPERCASE:", optionally followed by other text.

  – lines which look like a reST header: one line containing anything, followed by a line consisting only of whitespace, followed by a string of hyphens, equal signs, or other characters which are valid markers for reST headers: `- = ` : ' " ~ _ ^ * + # < >`. However, lines only containing double colons :: do not end "TESTS" blocks.

  Sometimes (but rarely) one has private or protected methods that don't need a proper `EXAMPLES` doctest. In these cases, one can either write traditional doctest using the `TESTS` block or use pytest to test the method. In the latter case, one has to add `TESTS: pytest` to the docstring, so that the method is explicitly marked as tested.

Note about Sphinx directives vs. other blocks

The main Sphinx directives that are used in Sage are:

`.. MATH::`, `.. NOTE::`, `.. PLOT::`, `.. RUBRIC::`, `.. SEEALSO::`, `.. TODO::`, `.. TOPIC::` and `.. WARNING::`.

They must be written exactly as above, so for example `WARNING::` or `.. WARNING ::` will not work.

Some other directives are also available, but less frequently used, namely:

`.. MODULEAUTHOR::`, `.. automethod::`, `.. autofunction::`, `.. image::`, `.. figure::`.

Other blocks shall not be used as directives; for example `.. ALGORITHM::` will not be shown at all.

Sage documentation style

All Sage documentation is written in reStructuredText (reST) and is processed by Sphinx. See https://www.sphinx-doc.org/rest.html for an introduction. Sage imposes these styles:

- Lines should be shorter than 80 characters. If in doubt, read PEP8: Maximum Line Length.
- All reST and Sphinx directives (like `.. WARNING::`, `.. NOTE::`, `.. MATH::`, etc.) are written in uppercase.
- Code fragments are quoted with double backticks. This includes function arguments and the Python literals like ``` ``True`` ```, ``` ``False`` ``` and ``` ``None`` ```. For example:

```
If ``check`` is ``True``, then ...
```

Sage's master BIBLIOGRAPHY file

All bibliographical references should be stored in the master bibliography file, `SAGE_ROOT/src/doc/en/reference/references/index.rst`, in the format

```
.. [Gau1801] \C. F. Gauss, *Disquisitiones Arithmeticae*, 1801.

.. [RSA1978] \R. Rivest, A. Shamir, L. Adleman,
             "A Method for Obtaining Digital Signatures and
             Public-Key Cryptosystems".
             Communications of the ACM **21** (February 1978),
             120-126. :doi:`10.1145/359340.359342`.
```

The part in brackets is the citation key: given these examples, you could then use `[Gau1801]_` in a docstring to provide a link to the first reference. Note the trailing underscore which makes the citation a hyperlink.

When possible, the key should have this form: for a single author, use the first three letters of the family name followed by the year; for multiple authors, use the first letter of each of the family names followed by the year. Note that the year should be four digits, not just the last two – Sage already has references from both 1910 and 2010, for example.

When abbreviating the first name of an author in a bibliography listing, be sure to put a backslash in front of it. This ensures that the letter (`C.` in the example above) will not be interpreted as a list enumerator.

For more about citations, see the Sphinx/reST markup for citations.

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.
```

(continues on next page)

(continued from previous page)

```
    - ``y`` -- integer (default: `2`); the description of the
      argument ``y``

    OUTPUT: the point as a tuple

    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 [BCDT2001]_ to determine
        whether an elliptic curve `E` over `Q` is modular.

    ...

    .. SEEALSO::

        :func:`line`

    TESTS::

        sage: A.point(42, 0)  # Check for corner case y=0
        xxx
    """
    <body of the function>
```

The master bibliography file would contain

```
.. [BCDT2001] Breuil, Conrad, Diamond, Taylor,
              "Modularity ...."
```

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.

Fine points on styles

A Sage developer, in writing code and docstrings, should follow the styles suggested in this manual, except special cases with good reasons. However, there are some details where we as a community did not reach to an agreement on the official style. These are

- one space:

  ```
  This is the first sentence. This is the second sentence.
  ```

  vs two spaces:

  ```
  This is the first sentence.  This is the second sentence.
  ```

  between sentences.

- tight list:

  ```
  - first item
  - second item
  - third item
  ```

  vs spaced list:

  ```
  - first item

  - second item

  - third item
  ```

There are different opinions on each of these, and in reality, we find instances in each style in our codebase. Then what should we do? Do we decide on one style by voting? There are different opinions even on what to do!

We can at least do this to prevent any dispute about these style conflicts:

- Acknowledge different authors may have different preferences on these.
- Respect the style choice of the author who first wrote the code or the docstrings.

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):
        """
```

(continues on next page)

(continued from previous page)

```
        <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**:

`` `x^2 + y^2 = 1` `` yields $x^2 + y^2 = 1$.

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}
```

$$f(x) = x^2 - 1$$
$$g(x) = x^x - f(x - 2)$$

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{align}
```

$$1 + ... + n = n(n+1)/2 \tag{1.1}$$
$$= O(n^2) \tag{1.2}$$
$$\tag{1.3}$$

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

Shortcuts are available which preserve readability, e.g. `\\ZZ` (**Z**), `\\RR` (**R**), `\\CC` (**C**), and `\\QQ` (**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.

  All blocks within the same docstring are linked: Variables set in a doctest keep their values for the remaining doctests within the same docstring. It is good practice to use different variable names for different values, as it makes the data flow in the examples easier to understand for human readers. (It also makes the data flow analysis in the Sage doctester more precise.) In particular, when unrelated examples appear in the same docstring, do not use the same variable name for both examples.

- **Preparsing:** As in Sage's console, $4/3$ returns $4/3$ and not 1.3333333333333333 as in Python. 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
```

- **Wrap long doctest lines:** Note that all doctests in EXAMPLES blocks get formatted as part of our HTML and PDF reference manuals. Our HTML manuals are formatted using the responsive design provided by the Furo theme. Even when the browser window is expanded to make use of the full width of a wide desktop screen, the style will not allow code boxes to grow arbitrarily wide.

  It is best to wrap long lines when possible so that readers do not have to scroll horizontally (back and forth) to follow an example.

  - Try to wrap long lines somewhere around columns 80 to 88 and try to never exceed column 95 in the source file. (Columns numbers are from the left margin in the source file; these rules work no matter how deep the docstring may be nested because also the formatted output will be nested.)
  - If you have to break an expression at a place that is not already nested in parentheses, wrap it in parentheses:

```
sage: (len(list(Permutations(['a', 'b', 'c', 'd', 'e', 'f', 'g'])))
....:    == len(list(Permutations(7))))
True
```

  - If the output in your only example is very wide and cannot be reasonably reformatted to fit (for example, large symbolic matrices or numbers with many digits), consider showing a smaller example first.
  - No need to wrap long `import` statements. Typically, the `import` statements are not the interesting parts of the doctests. Users only need to be able to copy-paste them into a Sage session or source file:

```
sage: from sage.rings.polynomial.multi_polynomial_ring import MPolynomialRing_
↪polydict, MPolynomialRing_polydict_domain  # this is fine
```

  - Wrap and indent long output to maximize readability in the source code and in the HTML output. But do not wrap strings:

```
sage: from sage.schemes.generic.algebraic_scheme import AlgebraicScheme_quasi
sage: P.<x, y, z> = ProjectiveSpace(2, ZZ)
sage: S = P.subscheme([])
sage: T = P.subscheme([x - y])
sage: U = AlgebraicScheme_quasi(S, T); U
Quasi-projective subscheme X - Y of Projective Space of dimension 2
 over Integer Ring,
  where X is defined by: (no polynomials)
    and Y is defined by: x - y
sage: U._repr_()
↪                                                                              ␣
↪                                                                              ␣
↪          # this is fine
'Quasi-projective subscheme X - Y of Projective Space of dimension 2 over␣
↪Integer Ring, where X is defined by:\n  (no polynomials)\nand Y is defined␣
↪by:\n  x - y'
```

    Also, if there is no whitespace in the doctest output where you could wrap the line, do not add such whitespace. Just don't wrap the line:

```
sage: B47 = RibbonGraph(4,7, bipartite=True); B47
Ribbon graph of genus 9 and 1 boundary components
sage: B47.sigma()
↪                                                                              ␣
↪                                                                              ␣
↪                  # this is fine
(1,2,3,4,5,6,7)(8,9,10,11,12,13,14)(15,16,17,18,19,20,21)(22,23,24,25,26,27,
↪28)(29,30,31,32)(33,34,35,36)(37,38,39,40)(41,42,43,44)(45,46,47,48)(49,50,
↪51,52)(53,54,55,56)
```

  - Doctest tags for modularization purposes such as `# needs sage.modules` (see *Special markup to influence doctests*) should be aligned at column 88. Clean lines from consistent alignment help reduce visual clutter. Moreover, at the maximum window width, only the word `# needs` will be visible in the HTML output without horizontal scrolling, striking a thoughtfully chosen balance between presenting the information and reducing visual clutter. (How much can be seen may be browser-dependent, of course.) In visually dense doctests, you can try to sculpt out visual space to separate the test commands from the annotation.
  - Doctest tags such as `# optional - pynormaliz` that make the doctest conditional on the presence of optional packages, on the other hand, should be aligned so that they are visible without having to scroll horizontally. The *doctest fixer* uses tab stops at columns 48, 56, 64, … for these tags.

- **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 doctests*.

Special markup to influence doctests

Overly complicated output in the example code can be shortened by an ellipsis marker `...`:

```
sage: [ZZ(n).ordinal_str() for n in range(25)]
['0th',
 '1st',
 '2nd',
 '3rd',
 '4th',
 '5th',
 ...
 '21st',
 '22nd',
 '23rd',
 '24th']
sage: ZZ('sage')
Traceback (most recent call last):
...
TypeError: unable to convert 'sage' to an integer
```

On the proper usage of the ellipsis marker, see Python's documentation.

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
```

  Doctests are expected to pass with any state of the pseudorandom number generators (PRNGs). 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.

One can also avoid the `random`-tag by checking basic properties:

```
sage: QQ.random_element().parent() is QQ
True
sage: QQ.random_element() in QQ
True
sage: a = QQ.random_element()
sage: b = QQ._random_nonzero_element()
sage: c = QQ._random_nonzero_element()
sage: (a/c) / (b/c) == a/b
True
```

Distribution can be checked with loops:

```
sage: found = {i: False for i in range(-2, 3)}
sage: while not all(found.values()):
....:     found[ZZ.random_element(-2, 3)] = True
```

This is mathematically correct, as it is guaranteed to terminate. However, there is a nonzero probability of a timout.

- **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
```

(continues on next page)

(continued from previous page)

```
[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)     # 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/needs:** A line tagged with `optional - FEATURE` or `needs FEATURE` is not tested unless the `--optional=KEYWORD` flag is passed to `sage -t` (see *Run optional doctests*). 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: oeis(60843)                  # optional - internet
A060843: Busy Beaver problem: a(n) = maximal number of steps that an
n-state Turing machine can make on an initially blank tape before
eventually halting.
```

  - **known bugs:** For lines that describe known bugs, you can use `# optional - bug`, although `# known bug` is preferred.

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

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

  - **modularization:** To enable *separate testing of the distribution packages* of the modularized Sage library, doctests that depend on features provided by other distribution packages can be tagged `# needs FEATURE`. For example:

```
Consider the following calculation::

    sage: a = AA(2).sqrt()  # needs sage.rings.number_field
    sage: b = sqrt(3)       # needs sage.symbolic
    sage: a + AA(b)         # needs sage.rings.number_field sage.symbolic
    3.146264369941973?
```

Note:

- Any words after `# optional` and `# needs` are interpreted as a list of package (spkg) names or other feature tags, separated by spaces.
- Any punctuation other than underscores (_) and periods (.), that is, 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 - depends on package CHomP` but simply `# optional - CHomP`.
- Optional tags are case-insensitive, so you could also write `# optional - chOMP`.

If `# optional` or `# needs` is placed right after the `sage:` prompt, it is a block-scoped tag, which applies to all doctest lines until a blank line is encountered.

These tags can also be applied to an entire file. If one of the first 10 lines of a file starts with any of `r""" sage.doctest: optional - FEATURE`, `# sage.doctest: needs FEATURE`, or `.. sage.doctest: optional - FEATURE` (in `.rst` files), etc., then this applies to all doctests in this file.

When a file is skipped that was explicitly given as a command line argument, a warning is displayed.

Note: If you add such a line to a file, you are strongly encouraged to add a note to the module-level documentation, saying that the doctests in this file will be skipped unless the appropriate conditions are met.

- **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(2^31 + 2^13)
8193                    # 32-bit
2147491840              # 64-bit
```

Per coding style (*Python code style*), the magic comment should be separated by at least 2 spaces.

For multiline doctests, the comment should appear on the first physical line of the doctest (the line with the prompt `sage:`), not on the continuation lines (the lines with the prompt `....:`):

```
sage: print(ZZ.random_element())          # random
42
sage: for _ in range(3):                  # random
....:     print(QQ.random_element())
1
1/77
-1/2
```

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.

Running automated doctests

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.

General coding style regarding whitespace

Use spaces instead of tabs for indentation. The only exception is for makefiles, in which tabs have a syntactic meaning different from spaces.

Do not add trailing whitespace.

Sage provides editor configuration for Emacs, using the file `.dir-locals.el`, to use spaces instead of tabs. Regarding trailing whitespace, see https://www.emacswiki.org/emacs/DeletingWhitespace for various solutions.

If you use another editor, we recommend to configure it so you do not add tabs to files. See *Text editors and IDEs for use with Sage*.

Global options

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

Miscellaneous minor things

Some decisions are arbitrary, but common conventions make life easier.

- Non-ASCII names in identifiers:
  - Translate *ä* and *ö* to *ae* and *oe*, like `moebius_function` for Möbius function.
  - Translate *á* to *a*, like `lovasz_number` for Lovász number.
- Common function keyword arguments:

  This is a list of some keyword arguments that many functions and methods take. For consistency, you should use the keywords from the list below with the meaning as explained here. Do not use a different keyword with the same meaning (for example, do not use `method`; use `algorithm` instead).
  - `algorithm`, a string or `None`: choose between various implementation or algorithm. Use `None` as a default that selects a sensible default, which could depend on installed optional packages.
  - `certificate`, a Boolean with `False` as default: whether the function should return some kind of certificate together with the result. With `certificate=True` the return value should be a pair (r, c) where r is the result that would be given with `certificate=False` and c is the certificate or `None` if there is no meaningful certificate.
  - `proof`, a Boolean with `True` as default: if `True`, require a mathematically proven computation. If `False`, a probabilistic algorithm or an algorithm relying to non-proved hypothesis like RH can be used.
  - `check`, a Boolean: do some additional checks to verify the input parameters. This should not otherwise influence the functioning of the code: if code works with `check=True`, it should also work with `check=False`.

- `coerce`, a Boolean: convert the input parameters to a suitable parent. This is typically used in constructors. You can call a method with `coerce=False` to skip some checks if the parent is known to be correct.
- `inplace`, a Boolean: whether to modify the object in-place or to return a copy.

1.5 Testing Sage

1.5.1 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 from source and the top level directory is:

```
[jdemeyer@localhost sage]$ pwd
/home/jdemeyer/sage
```

See the section *Running automated doctests* 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@localhost sage]$ ./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@localhost sage]$ ./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
```

(continues on next page)

(continued from previous page)

```
    [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@localhost sage]$ ./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@localhost sage]$ cd src/sage/games/
[jdemeyer@localhost games]$ ls
__init__.py  hexad.py       sudoku.py           sudoku_backtrack.pyx
all.py       quantumino.py  sudoku_backtrack.c
[jdemeyer@localhost 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.

You can also run the Sage doctester as follows:

```
[jdemeyer@localhost sage]$ ./sage -tox -e doctest -- src/sage/games/sudoku.py
```

See *Development and Testing Tools* for more information about tox.

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@localhost sage]$ ./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]
```

(continues on next page)

(continued from previous page)

```
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
Total time for all tests: 4.9 seconds
    cpu time: 3.6 seconds
    cumulative wall time: 3.6 seconds
[jdemeyer@localhost sage]$ ./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@localhost sage]$ sage -t src/sage/games/sudoku.py
sage -t  "src/sage/games/sudoku.py"
**********************************************************************
File "/home/jdemeyer/sage/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@localhost sage]$ ./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/sbox.pyx
    [124 tests, 0.8 s]
```

(continues on next page)

(continued from previous page)

```
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@localhost sage]$ ./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
```

(continues on next page)

(continued from previous page)

```
    [42 tests, 0.1 s]
sage -t --long src/sage/crypto/sbox.pyx
    [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@localhost sage]$ ./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]
```

(continues on next page)

(continued from previous page)

```
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/sbox.pyx
    [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@localhost sage]$ ./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
```

(continues on next page)

(continued from previous page)

```
    [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/sbox.pyx
    [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@localhost sage]$ ./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. (When this runs under the control of the GNU make jobserver, then Sage will request as most this number of job slots.)

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

```
[jdemeyer@localhost sage]$ 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.

The underlying command for running these tests is `sage -t --all`. For example, `make ptestlong` executes the command `sage -t -p --all --long --logfile=logs/ptestlong.log`. So if you want to add extra flags when you run these tests, for example `--verbose`, you can execute `sage -t -p --all --long`

`--verbose --logfile=path/to/logfile`. Some of the extra testing options are discussed here; run `sage -t -h` for a complete list.

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

(continues on next page)

(continued from previous page)

```
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@localhost sage]$ ./sage --preparse my_sage_script.sage
[mvngu@localhost sage]$ 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@localhost sage]$ ./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.combinat.affine_permutation)
Running doctests with ID 2018-02-07-13-23-13-89fe17b1.
Git branch: develop
Using --optional=sagemath_doc_html,sage
Doctesting 1 file.
sage -t /opt/sage/sage_stable/src/sage/combinat/affine_permutation.py
    [338 tests, 4.32 s]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
Total time for all tests: 4.4 seconds
```

(continues on next page)

(continued from previous page)

```
    cpu time: 3.6 seconds
    cumulative wall time: 4.3 seconds
```

Optional arguments

Run long doctests

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@localhost sage]$ ./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@localhost sage]$ ./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@localhost sage]$ ./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
```

(continues on next page)

(continued from previous page)

```
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@localhost sage]$ ./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`.

Doctests start from a random seed:

```
[kliem@localhost sage]$ ./sage -t src/sage/doctest/tests/random_seed.rst
Running doctests with ID 2020-06-23-23-22-59-49f37a55.
...
Doctesting 1 file.
sage -t --warn-long 89.5 --random-seed=112986622569797306072457879734474628454 src/
↪sage/doctest/tests/random_seed.rst
**********************************************************************
File "src/sage/doctest/tests/random_seed.rst", line 3, in sage.doctest.tests.random_
↪seed
Failed example:
    randint(5, 10)
Expected:
    9
Got:
    8
**********************************************************************
1 item had failures:
   1 of   2 in sage.doctest.tests.random_seed
    [1 test, 1 failure, 0.00 s]
----------------------------------------------------------------------
sage -t --warn-long 89.5 --random-seed=112986622569797306072457879734474628454 src/
↪sage/doctest/tests/random_seed.rst  # 1 doctest failed
----------------------------------------------------------------------
Total time for all tests: 0.0 seconds
    cpu time: 0.0 seconds
    cumulative wall time: 0.0 seconds
```

This seed can be set explicitly to reproduce possible failures:

```
[kliem@localhost sage]$ ./sage -t --warn-long 89.5                                \
                         --random-seed=112986622569797306072457879734474628454 \
                         src/sage/doctest/tests/random_seed.rst
Running doctests with ID 2020-06-23-23-24-28-14a52269.
...
Doctesting 1 file.
sage -t --warn-long 89.5 --random-seed=112986622569797306072457879734474628454 src/
↪sage/doctest/tests/random_seed.rst
**********************************************************************
File "src/sage/doctest/tests/random_seed.rst", line 3, in sage.doctest.tests.random_
↪seed
Failed example:
    randint(5, 10)
Expected:
    9
Got:
    8
**********************************************************************
1 item had failures:
   1 of   2 in sage.doctest.tests.random_seed
    [1 test, 1 failure, 0.00 s]
----------------------------------------------------------------------
sage -t --warn-long 89.5 --random-seed=112986622569797306072457879734474628454 src/
↪sage/doctest/tests/random_seed.rst  # 1 doctest failed
----------------------------------------------------------------------
Total time for all tests: 0.0 seconds
    cpu time: 0.0 seconds
    cumulative wall time: 0.0 seconds
```

It can also be set explicitly using the environment variable `SAGE_DOCTEST_RANDOM_SEED`.

Run optional doctests

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.

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

```
[roed@localhost sage]$ ./sage -t --optional=sagemath_doc_html,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@localhost sage]$ ./sage -t --optional=sagemath_doc_html,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` and `sagemath_doc_html`:

```
[roed@localhost sage]$ ./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
```

If you want Sage to detect external software or other capabilities (such as magma, latex, internet) automatically and run all of the relevant tests, then add `external`:

```
[roed@localhost sage]$ ./sage -t --optional=external src/sage/rings/real_mpfr.pyx
Running doctests with ID 2016-03-16-14-10-21-af2ebb67.
Using --optional=external
```

(continues on next page)

(continued from previous page)

```
External software to be detected: cplex,gurobi,internet,latex,macaulay2,magma,maple,
↪mathematica,matlab,octave,scilab
Doctesting 1 file.
sage -t --warn-long 28.0 src/sage/rings/real_mpfr.pyx
    [5 tests, 0.04 s]
----------------------------------------------------------------------
All tests passed!
----------------------------------------------------------------------
Total time for all tests: 0.5 seconds
    cpu time: 0.0 seconds
    cumulative wall time: 0.0 seconds
External software detected for doctesting: magma
```

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

```
[roed@localhost sage]$ ./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 doctests 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@localhost sage]$ ./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!
```

(continues on next page)

(continued from previous page)

```
----------------------------------------------------------------------
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@localhost sage]$ ./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/src/sage/plot/plot.py
    [304 tests, 69.0 s]
...
```

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@localhost sage]$ ./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 ".../site-packages/sage/doctest/forker.py", line 573, in _run
        self.execute(example, compiled, test.globs)
      File ".../site-packages/sage/doctest/forker.py", line 835, in execute
        exec compiled in globs
      File "<doctest sage.schemes.elliptic_curves.constructor[0]>", line 1, in
↪<module>
        EllipticCurve([Integer(0),Integer(0)])
      File ".../site-packages/sage/schemes/elliptic_curves/constructor.py", line 346,␣
↪in EllipticCurve
        return ell_rational_field.EllipticCurve_rational_field(x, y)
      File ".../site-packages/sage/schemes/elliptic_curves/ell_rational_field.py",␣
↪line 216, in __init__
        EllipticCurve_number_field.__init__(self, Q, ainvs)
      File ".../site-packages/sage/schemes/elliptic_curves/ell_number_field.py", line␣
↪159, in __init__
```

(continues on next page)

(continued from previous page)

```
        EllipticCurve_field.__init__(self, [field(x) for x in ainvs])
      File ".../site-packages/sage/schemes/elliptic_curves/ell_generic.py", line 156,␣
↪in __init__
        "Invariants %s define a singular curve."%ainvs
    ArithmeticError: Invariants [0, 0, 0, 0, 0] define a singular curve.
> .../site-packages/sage/schemes/elliptic_curves/ell_generic.py(156)__init__()
-> "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@localhost sage]$ ./sage -t --gdb \
                            src/sage/schemes/elliptic_curves/constructor.py
exec gdb --eval-commands="run" --args /home/roed/sage/local/var/lib/sage/venv-python3.
↪9/bin/python3 sage-runtests --serial --timeout=0 --stats-path=/home/roed/.sage/
↪timings2.json --optional=pip,sage,sage_spkg src/sage/schemes/elliptic_curves/
↪constructor.py
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.
```

(continues on next page)

(continued from previous page)

```
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@localhost sage]$ ./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@localhost sage]$ ./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]
----------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
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@localhost sage]$ ./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@localhost sage]$ ./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!
```

(continues on next page)

(continued from previous page)

```
----------------------------------------------------------------------
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@localhost sage]$ ./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@localhost sage]$ ./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/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@localhost sage]$ ./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!
```

(continues on next page)

(continued from previous page)

```
----------------------------------------------------------------------
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@localhost sage]$ ./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 which is not part of a package (which is not in a directory containing an `__init__.py` file), the testing code loads the globals from that file into the namespace before running tests. To disable this behaviour (and require imports to be explicitly specified), use the `--force-lib` option.

Auxiliary files

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

```
[roed@localhost sage]$ ./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@localhost sage]$ 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]
```

(continues on next page)

(continued from previous page)

```
----------------------------------------------------------------------
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 and pass/fail status for each file, use the `--stats-path` flag; the default location of this file is `~/.sage/timings2.json`. The doctester reads it if it exists, for the purpose of sorting the files so that slower tests are run first (and thus multiple processes are utilized most efficiently):

```
[roed@localhost sage]$ ./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.
...
```

At the end of the doctest run, Sage updates the json file if it exists or creates a new one.

The recorded pass/fail status of the files can be used for running only those files that failed their most recent test by using the `--failed` flag (`-f` for short).

Using the option `--baseline-stats-path known-test-failures.json`, it is possible to distinguish files with known doctest failures from new failures. The file `known-test-failures.json` should be prepared in the same format as `timings2.json`.

Source files marked as failed there will be marked as "[failed in baseline]" failures in the doctest report; and if there are only baseline failures, no new failures, then `sage -t` will exit with status code 0 (success).

Options for testing in virtual environments

The distribution packages of the modularized Sage library can be tested in virtual environments. Sage has infrastructure to create such virtual environments using `tox`, which is explained in detail in *Testing distribution packages*. Our examples in this section refer to this setting, but it applies the same to any user-created virtual environments.

The virtual environments, set up in directories such as `pkgs/sagemath-standard/.tox/sagepython-sagewheels-nopypi-norequirements` contain installations of built (non-editable) wheels.

To test all modules of Sage that are installed in a virtual environment, use the option `--installed` (instead of `--all`):

```
[mkoeppe@localhost sage]$ pkgs/sagemath-standard/.tox/sagepython-.../sage -t   \
                            -p4 --installed
```

This tests against the doctests as they appear in the installed copies of the files (in `site-packages/sage/...`). Note that these installed copies should never be edited, as they can be overwritten without warning.

When testing a modularized distribution package other than sagemath-standard, the top-level module `sage.all` is not available. Use the option `--environment` to select an appropriate top-level module:

```
[mkoeppe@localhost sage]$ pkgs/sagemath-categories/.tox/sagepython-.../sage -t \
                            -p4 --environment sage.all__sagemath_categories    \
                            --installed
```

To test the installed modules against the doctests as they appear in the source tree (`src/sage/...`):

```
[mkoeppe@localhost sage]$ pkgs/sagemath-categories/.tox/sagepython-.../sage -t \
                            -p4 --environment sage.all__sagemath_categories    \
                            src/sage/structure
```

Note that testing all doctests as they appear in the source tree does not make sense because many of the source files may not be installed in the virtual environment. Use the option `--if-installed` to skip the source files of all Python/Cython modules that are not installed in the virtual environment:

```
[mkoeppe@localhost sage]$ pkgs/sagemath-categories/.tox/sagepython-.../sage -t \
                            -p4 --environment sage.all__sagemath_categories    \
                            --if-installed src/sage/schemes
```

This option can also be combined with `--all`:

```
[mkoeppe@localhost sage]$ pkgs/sagemath-categories/.tox/sagepython-.../sage -t \
                            -p4 --environment sage.all__sagemath_categories    \
                            --if-installed --all
```

The doctest fixer

Sage provides a development tool that assists with updating doctests.

Updating doctest outputs

By default, `./sage --fixdoctests` runs the doctester and replaces the expected outputs of all examples by the actual outputs from the current version of Sage:

```
[mkoeppe@localhost sage]$ ./sage --fixdoctests \
                            --overwrite src/sage/arith/weird.py
```

For example, when applied to this Python file:

```
| r"""
| ...
|
| EXAMPLES::
|
|     sage: 2 + 2
|     5
|     sage: factor("91")
|     "7" * "13"
| ...
```

the doctest fixer edits the file as follows:

```
| r"""
| ...
|
| EXAMPLES::
|
|     sage: 2 + 2
|     4
|     sage: factor("91")
|     Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
|     ...
|     TypeError: unable to factor '91'
| ...
```

As this command edits the source file, it may be a good practice to first use `git commit` to save any changes made in the file.

After running the doctest fixer, it is a good idea to use `git diff` to check all edits that the automated tool made.

An alternative to this workflow is to use the option `--keep-both`. When expected and actual output of an example differ, it duplicates the example, marking the two copies `# optional - EXPECTED` and `# optional - GOT`. (Thus, when re-running the doctester, neither of the two copies is run; this makes `./sage --fixdoctests` idempotent.)

When exceptions are expected by an example, it is standard practice to abbreviate the tracebacks using `...`. The doctest fixer uses this abbreviation automatically when formatting the actual output, as shown in the above example. To disable it so that the details of the exception can be inspected, use the option `--full-tracebacks`. This is particularly useful in combination with `--keep-both`:

```
[mkoeppe@localhost sage]$ ./sage --fixdoctests --keep-both --full-tracebacks \
                          --overwrite src/sage/arith/weird.py
```

This will give the following result on the above example:

```
| r"""
| ...
|
| EXAMPLES::
|
|     sage: 2 + 2                                 # optional - EXPECTED
|     5
|     sage: 2 + 2                                 # optional - GOT
|     4
|     sage: factor("91")                          # optional - EXPECTED
|     "7" * "13"
|     sage: factor("91")                          # optional - GOT
|     Traceback (most recent call last):
|     ...
|     File "<doctest...>", line 1, in <module>
|     factor("91")
|     File ".../src/sage/arith/misc.py", line 2680, in factor
|     raise TypeError("unable to factor {!r}".format(n))
|     TypeError: unable to factor '91'
| ...
| """
```

To make sure that all doctests are updated, you may have to use the option `--long`:

```
[mkoeppe@localhost sage]$ ./sage --fixdoctests --long \
                          --overwrite src/sage/arith/weird.py
```

If you are not comfortable with allowing this tool to edit your source files, you can use the option `--no-overwrite`, which will create a new file with the extension `.fixed` instead of overwriting the source file:

```
[mkoeppe@localhost sage]$ ./sage --fixdoctests \
                          --no-overwrite src/sage/arith/weird.py
```

Managing `# optional` and `# needs` tags

When a file uses a `# sage.doctest: optional/needs FEATURE` directive, the doctest fixer automatically removes the redundant `# optional/needs FEATURE` tags from all `sage:` lines. Likewise, when a block-scoped tag `sage: # optional/needs FEATURE` is used, then the doctest fixer removes redundant tags from all doctests in this scope. For example:

```
| # sage.doctest: optional - sirocco, needs sage.rings.number_field
| r"""
| ...
|
| EXAMPLES::
|
|     sage: # needs sage.modules sage.rings.number_field
|     sage: Q5 = QuadraticField(5)
|     sage: V = Q5^42                                  # needs sage.modules
|     sage: T = transmogrify(V)           # optional - bliss sirocco
```

is automatically transformed to:

```
| # sage.doctest: optional - sirocco, needs sage.rings.number_field
| r"""
| ...
|
| EXAMPLES::
|
|     sage: # needs sage.modules
|     sage: Q5 = QuadraticField(5)
|     sage: V = Q5^42
|     sage: T = transmogrify(V)               # optional - bliss
```

The doctest fixer also aligns the `# optional/needs FEATURE` tags on individual doctests at a fixed set of tab stops.

The doctester may issue style warnings when `# optional/needs` tags are repeated on a whole block of doctests, suggesting to use a block-scoped tag instead. The doctest fixer makes these changes automatically.

There are situations in which the doctester and doctest fixer show too much restraint and a manual intervention would improve the formatting of the doctests. In the example below, the doctester does not issue a style warning because the first doctest line does not carry the `# needs` tag:

```
| EXAMPLES::
|
|     sage: set_verbose(-1)
|     sage: P.<x,y,z> = ProjectiveSpace(QQbar, 2)     # needs sage.rings.number_field
|     sage: C = Curve([x^3*y + 2*x^2*y^2 + x*y^3      # needs sage.rings.number_field
|     ....:             + x^3*z + 7*x^2*y*z
|     ....:             + 14*x*y^2*z + 9*y^3*z], P)
|     sage: Q = P([0,0,1])                            # needs sage.rings.number_field
|     sage: C.tangents(Q)                             # needs sage.rings.number_field
|     [x + 4.147899035704788?*y,
|      x + (1.426050482147607? + 0.3689894074818041?*I)*y,
|      x + (1.426050482147607? - 0.3689894074818041?*I)*y]
```

To change this example, there are two approaches:

1. Just add the line `sage: # needs sage.rings.number_field` at the beginning and run the doctest fixer, which will remove the tags on the individual doctests that have now become redundant.

2. Insert a blank line after the first doctest line, splitting the block into two. Now the `# needs` tag is repeated on the whole second block, so running the doctest fixer will add a block-scoped tag and remove the individual tags:

```
| EXAMPLES::
|
|     sage: set_verbose(-1)
|
|     sage: # needs sage.rings.number_field
|     sage: P.<x,y,z> = ProjectiveSpace(QQbar, 2)
|     sage: C = Curve([x^3*y + 2*x^2*y^2 + x*y^3
|     ....:             + x^3*z + 7*x^2*y*z
|     ....:             + 14*x*y^2*z + 9*y^3*z], P)
|     sage: Q = P([0,0,1])
|     sage: C.tangents(Q)
|     [x + 4.147899035704788?*y,
|      x + (1.426050482147607? + 0.36898940748180 41?*I)*y,
|      x + (1.426050482147607? - 0.36898940748180 41?*I)*y]
```

In places where the doctester issues a doctest dataflow warning (`Variable ... referenced here was set only in doctest marked '# optional - FEATURE'`), the doctest fixer automatically adds the missing `# optional/needs` tags.

Sometimes code changes can make existing `# optional/needs FEATURE` tags unnecessary. In an installation or virtual environment where `FEATURE` is not available, you can invoke the doctest fixer with the option `--probe FEATURE`. Then it will run examples marked `# optional/needs - FEATURE` silently, and if the example turns out to work anyway, the tag is automatically removed.

Note: Probing works best when the doctests within a docstring do not reuse the same variable for different values.

To have the doctest fixer take care of the `# optional/needs` tags, but not change the expected results of examples, use the option `--only-tags`. This mode is suitable for mostly unattended runs on many files.

With the option `--verbose`, the doctest fixer shows the doctester's messages one by one and reports the changes made.

Warning: While the doctest fixer guarantees to preserve any comments that appear before `# optional/needs` and all parenthesized comments of the form `# optional - FEATURE (EXPLANATION)`, any free-form comments that may be mixed with the doctest tags will be lost.

If you don't want to update any doctests, you can use the option `--no-test`. In this mode, the doctest fixer does not run the doctester and only normalizes the style of the `# optional` tags.

Use in virtual environments

The doctest fixer can also run tests using the Sage doctester installed in a virtual environment:

```
[mkoeppe@localhost sage]$ ./sage --fixdoctests --overwrite                    \
                          --distribution sagemath-categories                  \
                          src/sage/geometry/schemes/generic/*.py
```

This command, using `--distribution`, is equivalent to a command that uses the more specific options `--venv` and `--environment`:

```
[mkoeppe@localhost sage]$ ./sage --fixdoctests --overwrite                    \
                          --venv pkgs/sagemath-categories/.tox/sagepython-... \
                          --environment sage.all__sagemath_categories
                          src/sage/geometry/schemes/generic/*.py
```

Either way, the options `--keep-both`, `--full-tracebacks`, and `--if-installed` are implied.

In this mode of operation, when the doctester encounters a global name that is unknown in its virtual environment (`NameError`), the doctest fixer will look up the name in its own environment (typically a full installation of the Sage library) and add a `# needs ...` tag to the doctest.

Likewise, when the doctester runs into a `ModuleNotFoundError`, the doctest fixer will automatically add a `# needs ...` tag.

The switch `--distribution` can be repeated; the given distributions will be tested in sequence. Using `--distribution all` is equivalent to a preset list of `--distribution` switches. With the switch `--fixed-point`, the doctest fixer runs the given distributions until no more changes are made.

Updating baseline files

The modularized distribution packages `pkgs/sagemath-categories` and `pkgs/sagemath-repl` contain files `known-test-failures*.json` for use with the option `--baseline-stats-path`, see section *Auxiliary files*.

After running the doctesters of the distributions, for example, via `sage --fixdoctests`, you can use the test results stored in `timings2.json` files to update the `known-test-failures*.json` files. This update can be done using the command:

```
[mkoeppe@localhost sage]$ ./sage --fixdoctests --no-test                      \
                          --update-known-test-failures --distribution all
```

1.5.2 Testing on Multiple Platforms

Sage is intended to build and run on a variety of platforms, including all major Linux distributions, as well as macOS, and Windows with WSL (Windows Subsystem for Linux).

There is considerable variation among these platforms. To ensure that Sage continues to build correctly on users' machines, it is crucial to test changes to Sage, in particular when external packages are added or upgraded, on a wide spectrum of platforms.

Testing PRs with GitHub Actions

GitHub Actions are automatically and constantly testing GitHub PRs to identify errors early and ensure code quality. In particular, Build & Test workflows perform an incremental build of Sage and run doctests on a selection of major platforms including Ubuntu, macOS, and Conda.

Sage buildbots

Before a new release, the release manager runs a fleet of buildbots to make it sure that Sage builds correctly on all of our supported platforms.

Test reports on sage-release

Sage developers and users are encouraged to test releases that are announced on Sage Release on their machines and to report the results (successes and failures) by responding to the announcements.

Testing on multiple platforms using Docker

Docker is a popular virtualization software, running Linux operating system images ("Docker images") in containers on a shared Linux kernel. These containers can be run using a Docker client on your Linux, Mac, or Windows box, as well as on various cloud services.

To get started, you need to install a Docker client. The clients are available for Linux, Mac, and Windows. The clients for the latter are known as "Docker Desktop".

Make sure that your Docker client is configured to provide enough RAM to the containers (8 GB are a good choice). In Docker Desktop this setting is in Preferences -> Resources -> Advanced.

Note: All examples in this section were obtained using Docker Desktop for Mac; but the command-line user interface for the other platforms is identical.

As an alternative, you can also run Docker in GitHub Codespaces (or another cloud service) using a container with the Docker-in-Docker feature. Sage provides a suitable dev container configuration .devcontainer/tox-docker-in-docker:

All major Linux distributions provide ready-to-use Docker images, which are published via Docker Hub or other container registries. For example, to run the current stable (LTS) version of Ubuntu interactively, you can use the shell command:

```
[mkoeppe@sage sage]$ docker run -it ubuntu:latest
root@9f3398da43c2:/#
```

Here `ubuntu` is referred to as the "image (name)" and `latest` as the "tag". Other releases of Ubuntu are available under different tags, such as `xenial` or `devel`.

The above command drops you in a root shell on the container:

```
root@9f3398da43c2:/# uname -a
Linux 9f3398da43c2 4.19.76-linuxkit #1 SMP Thu Oct 17 19:31:58 UTC 2019 x86_64 x86_64␣
↪x86_64 GNU/Linux
root@9f3398da43c2:/# df -h
Filesystem      Size  Used Avail Use% Mounted on
overlay         181G  116G   56G  68% /
tmpfs            64M     0   64M   0% /dev
tmpfs           2.7G     0  2.7G   0% /sys/fs/cgroup
shm              64M     0   64M   0% /dev/shm
/dev/sda1       181G  116G   56G  68% /etc/hosts
tmpfs           2.7G     0  2.7G   0% /proc/acpi
tmpfs           2.7G     0  2.7G   0% /sys/firmware
```

Exiting the shell terminates the container:

```
root@9f3398da43c2:/# ^D
[mkoeppe@sage sage]$
```

Let us work with a distclean Sage source tree. If you are using git, a good way to get one (without losing a precious installation in `SAGE_LOCAL`) is by creating a new worktree:

```
[mkoeppe@sage sage] git worktree add worktree-ubuntu-latest
[mkoeppe@sage sage] cd worktree-ubuntu-latest
[mkoeppe@sage worktree-ubuntu-latest] ls
COPYING.txt ... Makefile ... configure.ac ... src tox.ini
```

This is not bootstrapped (`configure` is missing), so let's bootstrap it:

```
[mkoeppe@sage worktree-ubuntu-latest] make configure
...
```

We can start a container again with same image, `ubuntu:latest`, but this time let's mount the current directory into it:

```
[mkoeppe@sage worktree-ubuntu-latest]$ docker run -it --mount type=bind,source=$(pwd),
↪target=/sage ubuntu:latest
root@39d693b2a75d:/# mount | grep sage
osxfs on /sage type fuse.osxfs (rw,nosuid,nodev,relatime,user_id=0,group_id=0,allow_
↪other,max_read=1048576)
root@39d693b2a75d:/# cd sage
root@39d693b2a75d:/sage# ls
COPYING.txt ... Makefile ... config configure configure.ac ... src tox.ini
```

Typical Docker images provide minimal installations of packages only:

```
root@39d693b2a75d:/sage# command -v python
root@39d693b2a75d:/sage# command -v gcc
root@39d693b2a75d:/sage#
```

As you can see above, the image `ubuntu:latest` has neither a Python nor a GCC installed, which are among the build prerequisites of Sage. We need to install them using the Linux distribution's package manager first.

Sage facilitates testing various Linux distributions on Docker as follows.

Discovering the system's package system

```
root@39d693b2a75d:/sage# build/bin/sage-guess-package-system
debian
```

Let's install gcc, hoping that the Ubuntu package providing it is simply named `gcc`. If we forgot what the package manager on Debian-derived Linux distributions is called, we can ask Sage for a reminder:

```
root@39d693b2a75d:/sage# build/bin/sage-print-system-package-command debian install␣
↪gcc
apt-get install gcc
```

We remember that we need to fetch the current package lists from the server first:

```
root@39d693b2a75d:/sage# apt-get update
root@39d693b2a75d:/sage# apt-get install gcc
```

Using Sage's database of distribution prerequisites

The source code of the Sage distribution contains a database of package names in various distributions' package managers. For example, the file `build/pkgs/_prereq/distros/debian.txt` contains the following

```
# This file, build/pkgs/_prereq/distros/debian.txt, contains names
# of Debian/Ubuntu packages needed for installation of Sage from source.
#
# In addition, the files build/pkgs/SPKG/distros/debian.txt contain the names
# of packages that provide the equivalent of SPKG.
#
# Everything on a line after a # character is ignored.
binutils
make
m4
perl
# python3-minimal is not enough on debian buster, ubuntu bionic - it does not have␣
↪urllib
python3    # system python for bootstrapping the build
tar
bc
gcc
# On debian buster, need C++ even to survive 'configure'. Otherwise:
# checking how to run the C++ preprocessor... /lib/cpp
# configure: error: in `/sage':
# configure: error: C++ preprocessor "/lib/cpp" fails sanity check
g++
# Needed if we download some packages from a https upstream URL
ca-certificates
```

From this information, we know that we can use the following command on our container to install the necessary build prerequisites:

```
root@39d693b2a75d:/sage# apt-get install binutils make m4 perl python3 \
                                         tar bc gcc g++ ca-certificates
Reading package lists... Done
Building dependency tree
Reading state information... Done
tar is already the newest version (1.29b-2ubuntu0.1).
The following additional packages will be installed:
...
Done.
```

(The Sage Installation Guide also provides such command lines for some distributions; these are automatically generated from the database of package names.)

Now we can start the build:

```
root@39d693b2a75d:/sage# ./configure
checking for a BSD-compatible install... /usr/bin/install -c
checking for root user... yes
configure: error: You cannot build Sage as root, switch to an unprivileged user.
(If building in a container, use --enable-build-as-root.)
```

Let's just follow this helpful hint:

```
root@39d693b2a75d:/sage# ./configure --enable-build-as-root
checking for a BSD-compatible install... /usr/bin/install -c
...
```

Using Sage's database of equivalent distribution packages

At the end of the `./configure` run, Sage issued a message like the following:

```
configure: notice: the following SPKGs did not find equivalent system packages:
                   arb boost_cropped bzip2 ... zeromq zlib
checking for the package system in use... debian
configure: hint: installing the following system packages is recommended and
                 may avoid building some of the above SPKGs from source:
configure:   $ sudo apt-get install libflint-arb-dev ... libzmq3-dev libz-dev
configure: After installation, re-run configure using:
configure:   $ make reconfigure
```

This information comes from Sage's database of equivalent system packages. For example:

```
root@39d693b2a75d:/sage# ls build/pkgs/arb/distros/
arch.txt      conda.txt       debian.txt      gentoo.txt
root@39d693b2a75d:/sage# cat build/pkgs/arb/distros/debian.txt
libflint-arb-dev
```

Note that these package equivalencies are based on a current stable or testing version of the distribution; the packages are not guaranteed to exist in every release or derivative distribution.

The Sage distribution is intended to build correctly no matter what superset of the set of packages forming the minimal build prerequisites is installed on the system. If it does not, this is a bug of the Sage distribution and should be reported and fixed on a ticket. Crucial part of a bug report is the configuration of the system, in particular a list of installed packages and their versions.

Let us install a subset of these packages:

```
root@39d693b2a75d:/sage# apt-get install libbz2-dev bzip2 libz-dev
Reading package lists... Done
...
Setting up zlib1g-dev:amd64 (1:1.2.11.dfsg-0ubuntu2) ...
root@39d693b2a75d:/sage#
```

Committing a container to disk

After terminating the container, the following command shows the status of the container you just exited:

```
root@39d693b2a75d:/sage# ^D
[mkoeppe@sage worktree-ubuntu-latest]$ docker ps -a | head -n3
CONTAINER ID   IMAGE           COMMAND       CREATED         STATUS
39d693b2a75d   ubuntu:latest   "/bin/bash"   8 minutes ago   Exited (0) 6 seconds ago
9f3398da43c2   ubuntu:latest   "/bin/bash"   8 minutes ago   Exited (0) 8 minutes ago
```

We can go back to the container with the command:

```
[mkoeppe@sage worktree-ubuntu-latest]$ docker start -a -i 39d693b2a75d
root@9f3398da43c2:/#
```

Here, `39d693b2a75d` is the container id, which appeared in the shell prompts and in the output of `docker ps`.

We can create a new image corresponding to its current state:

```
root@39d693b2a75d:/# ^D
[mkoeppe@sage worktree-ubuntu-latest]$ docker commit 39d693b2a75d ubuntu-latest-
↪minimal-17
sha256:4151c5ca4476660f6181cdb13923da8fe44082222b984c377fb4fd6cc05415c1
```

where `ubuntu-latest-minimal-17` is an arbitrary symbolic name for the new image. The output of the command is the id of the new image. We can use either the symbolic name or the id to refer to the new image.

We can run the image and get a new container with the same state as the one that we terminated. Again we want to mount our worktree into it; otherwise, because we did not make a copy, the new container will have no access to the worktree:

```
[mkoeppe@sage worktree-ubuntu-latest]$ docker run -it \
  --mount type=bind,source=$(pwd),target=/sage ubuntu-latest-minimal-17
root@73987568712c:/# cd sage
root@73987568712c:/sage# command -v gcc
/usr/bin/gcc
root@73987568712c:/sage# command -v bunzip2
/usr/bin/bunzip2
root@73987568712c:/sage# ^D
[mkoeppe@sage worktree-ubuntu-latest]$
```

The image `ubuntu-latest-minimal-17` can be run in as many containers as we want and can also be shared with other users or developers so that they can run it in a container on their machine. (See the Docker documentation on how to share images on Docker Hub or to save images to a tar archive.)

This facilitates collaboration on fixing portability bugs of the Sage distribution. After reproducing a portability bug on a container, several developers can work on fixing the bug using containers running on their respective machines.

Generating dockerfiles

Sage also provides a script for generating a `Dockerfile`, which is a recipe for automatically building a new image:

```
[mkoeppe@sage sage]$ build/bin/write-dockerfile.sh debian ":standard: :optional:" >␣
↪Dockerfile
```

(The second argument is passed to `sage -package list` to find packages for the listed package types.)

The `Dockerfile` instructs the command `docker build` to build a new Docker image. Let us take a quick look at the generated file; this is slightly simplified:

```
[mkoeppe@sage sage]$ cat Dockerfile
# Automatically generated by SAGE_ROOT/build/bin/write-dockerfile.sh
# the :comments: separate the generated file into sections
# to simplify writing scripts that customize this file
...
```

First, it instructs `docker build` to start from an existing base image…:

```
...
ARG BASE_IMAGE=ubuntu:latest
FROM ${BASE_IMAGE}
...
```

Then, to install system packages…:

```
...
RUN apt-get update && DEBIAN_FRONTEND=noninteractive apt-get install -qqq --no-
↪install-recommends --yes binutils make m4 perl python3 ... libzmq3-dev libz-dev &&␣
↪apt-get clean
```

Then, to bootstrap and configure…:

```
RUN mkdir -p /sage
WORKDIR /sage
ADD Makefile VERSION.txt README.md bootstrap configure.ac sage ./
ADD src/doc/bootstrap src/doc/bootstrap
ADD m4 ./m4
ADD build ./build
RUN ./bootstrap
ADD src/bin src/bin
ARG EXTRA_CONFIGURE_ARGS=""
RUN ./configure --enable-build-as-root ${EXTRA_CONFIGURE_ARGS} || (cat config.log;␣
↪exit 1)
```

Finally, to build and test…:

```
ARG NUMPROC=8
ENV MAKE="make -j${NUMPROC}"
ARG USE_MAKEFLAGS="-k"
RUN make ${USE_MAKEFLAGS} base-toolchain
ARG TARGETS_PRE="all-sage-local"
RUN make ${USE_MAKEFLAGS} ${TARGETS_PRE}
ADD src src
ARG TARGETS="build ptest"
RUN make ${USE_MAKEFLAGS} ${TARGETS}
```

You can customize the image build process by passing build arguments to the command `docker build`. For example:

```
[mkoeppe@sage sage]$ docker build . -f Dockerfile \
  --build-arg BASE_IMAGE=ubuntu:latest \
  --build-arg NUMPROC=4 \
  --build-arg EXTRA_CONFIGURE_ARGS="--with-python=/usr/bin/python3.42"
```

These arguments (and their default values) are defined using `ARG` commands in the `Dockerfile`.

The above command will build Sage from scratch and will therefore take quite long. Let us instead just do a partial build, consisting of one small package, by setting the arguments `TARGETS_PRE` and `TARGETS`. We use a silent build (`make V=0`):

```
[mkoeppe@sage sage]$ docker build . -f Dockerfile \
  --build-arg TARGETS_PRE=ratpoints \
  --build-arg TARGETS=ratpoints \
  --build-arg USE_MAKEFLAGS="V=0"
Sending build context to Docker daemon    285MB
Step 1/28 : ARG BASE_IMAGE=ubuntu:latest
...
Step 2/28 : FROM ${BASE_IMAGE}
 ---> 549b9b86cb8d
...
Step 25/28 : RUN make SAGE_SPKG="sage-spkg -y -o" ${USE_MAKEFLAGS} ${TARGETS_PRE}
...
make[1]: Entering directory '/sage/build/make'
```

(continues on next page)

(continued from previous page)

```
sage-logger -p 'sage-spkg -y -o  ratpoints-2.1.3.p5' '/sage/logs/pkgs/ratpoints-2.1.3.
↪p5.log'
[ratpoints-2.1.3.p5] installing. Log file: /sage/logs/pkgs/ratpoints-2.1.3.p5.log
  [ratpoints-2.1.3.p5] successfully installed.
make[1]: Leaving directory '/sage/build/make'

real  0m18.886s
user  0m1.779s
sys   0m0.314s
Sage build/upgrade complete!
...
---> 2d06689d39fa
Successfully built 2d06689d39fa
```

We can now start a container using the image id shown in the last step:

```
[mkoeppe@sage sage]$ docker run -it 2d06689d39fa bash
root@fab59e09a641:/sage# ls -l logs/pkgs/
total 236
-rw-r--r-- 1 root root 231169 Mar 26 22:07 config.log
-rw-r--r-- 1 root root   6025 Mar 26 22:27 ratpoints-2.1.3.p5.log
root@fab59e09a641:/sage# ls -l local/lib/*rat*
-rw-r--r-- 1 root root 177256 Mar 26 22:27 local/lib/libratpoints.a
```

You can customize the image build process further by editing the `Dockerfile`. For example, by default, the generated `Dockerfile` configures, builds, and tests Sage. By deleting or commenting out the commands for the latter, you can adjust the Dockerfile to stop after the `configure` phase, for example.

`Dockerfile` is the default filename for Dockerfiles. You can change it to any other name, but it is recommended to use `Dockerfile` as a prefix, such as `Dockerfile-debian-standard`. It should be placed within the tree rooted at the current directory (`.`); if you want to put it elsewhere, you need to learn about details of "Docker build contexts".

Note that in contrast to the workflow described in the above sections, the `Dockerfile` **copies** a snapshot of your Sage worktree into the build container, using `ADD` commands, instead of mounting the directory into it. This copying is subject to the exclusions in the `.gitignore` file (via a symbolic link from `.dockerignore`). Therefore, only the sources are copied, but not your configuration (such as the file `config.status`), nor the `$SAGE_LOCAL` tree, nor any other build artefacts.

Because of this, you can build a Docker image using the generated `Dockerfile` from your main Sage development tree. It does not have to be distclean to start, and the build will not write into it at all. Hence, you can continue editing and compiling your Sage development tree even while Docker builds are running.

Debugging a portability bug using Docker

Let us do another partial build. We choose a package that we suspect might not work on all platforms, `surf`, which was marked as "experimental" in 2017:

```
[mkoeppe@sage sage]$ docker build . -f Dockerfile \
  --build-arg BASE_IMAGE=ubuntu:latest \
  --build-arg NUMPROC=4 \
  --build-arg TARGETS_PRE=surf \
  --build-arg TARGETS=surf
Sending build context to Docker daemon    285MB
Step 1/28 : ARG BASE_IMAGE=ubuntu:latest
Step 2/28 : FROM ${BASE_IMAGE}
```

(continues on next page)

(continued from previous page)

```
 ---> 549b9b86cb8d
...
Step 24/28 : ARG TARGETS_PRE="all-sage-local"
 ---> Running in 17d0ddb5ad7b
Removing intermediate container 17d0ddb5ad7b
 ---> 7b51411520c3
Step 25/28 : RUN make SAGE_SPKG="sage-spkg -y -o" ${USE_MAKEFLAGS} ${TARGETS_PRE}
 ---> Running in 61833bea6a6d
make -j4 build/make/Makefile --stop
...
[surf-1.0.6-gcc6] Attempting to download package surf-1.0.6-gcc6.tar.gz from mirrors
...
[surf-1.0.6-gcc6] http://mirrors.mit.edu/sage/spkg/upstream/surf/surf-1.0.6-gcc6.tar.
↪gz
...
[surf-1.0.6-gcc6] Setting up build directory for surf-1.0.6-gcc6
...
[surf-1.0.6-gcc6] /usr/bin/ld: cannot find -lfl
[surf-1.0.6-gcc6] collect2: error: ld returned 1 exit status
[surf-1.0.6-gcc6] Makefile:504: recipe for target 'surf' failed
[surf-1.0.6-gcc6] make[3]: *** [surf] Error 1
...
[surf-1.0.6-gcc6] Error installing package surf-1.0.6-gcc6
...
Makefile:2088: recipe for target '/sage/local/var/lib/sage/installed/surf-1.0.6-gcc6'␣
↪failed
make[1]: *** [/sage/local/var/lib/sage/installed/surf-1.0.6-gcc6] Error 1
make[1]: Target 'surf' not remade because of errors.
make[1]: Leaving directory '/sage/build/make'
...
Error building Sage.

The following package(s) may have failed to build (not necessarily
during this run of 'make surf'):

* package:         surf-1.0.6-gcc6
  last build time: Mar 26 22:07
  log file:        /sage/logs/pkgs/surf-1.0.6-gcc6.log
  build directory: /sage/local/var/tmp/sage/build/surf-1.0.6-gcc6

...
Makefile:31: recipe for target 'surf' failed
make: *** [surf] Error 1
The command '/bin/sh -c make SAGE_SPKG="sage-spkg -y -o" ${USE_MAKEFLAGS} ${TARGETS_
↪PRE}'
returned a non-zero code: 2
```

Note that no image id is shown at the end; the build failed, and no image is created. However, the container in which the last step of the build was attempted exists:

```
[mkoeppe@sage sage]$ docker ps -a |head -n3
CONTAINER ID        IMAGE                      COMMAND                  CREATED      ␣
↪       STATUS
61833bea6a6d        7b51411520c3               "/bin/sh -c 'make SA…"   9 minutes␣
↪ago       Exited (2) 1 minute ago
73987568712c        ubuntu-latest-minimal-17   "/bin/bash"              24 hours ago␣
↪       Exited (0) 23 hours ago
```

We can copy the build directory from the container for inspection:

```
[mkoeppe@sage sage]$ docker cp 61833bea6a6d:/sage/local/var/tmp/sage/build ubuntu-
↪build
[mkoeppe@sage sage]$ ls ubuntu-build/surf*/src
AUTHORS          TODO            curve           misc
COPYING          acinclude.m4    debug           missing
ChangeLog        aclocal.m4      dither          mkinstalldirs
INSTALL          background.pic  docs            mt
Makefile         config.guess    draw            src
Makefile.am      config.log      drawfunc        surf.1
Makefile.global  config.status   examples        surf.xpm
Makefile.in      config.sub      gtkgui          yaccsrc
NEWS             configure       image-formats
README           configure.in    install-sh
```

Alternatively, we can use `docker commit` as explained earlier to create an image from the container:

```
[mkoeppe@sage sage]$ docker commit 61833bea6a6d
sha256:003fbd511016fe305bd8494bb1747f0fbf4cb2c788b4e755e9099d9f2014a60d
[mkoeppe@sage sage]$ docker run -it 003fbd511 bash
root@2d9ac65f4572:/sage# (cd /sage/local/var/tmp/sage/build/surf* && /sage/sage --
↪buildsh)

Starting subshell with Sage environment variables set.  Don't forget
to exit when you are done.
...
Note: SAGE_ROOT=/sage
(sage-buildsh) root@2d9ac65f4572:surf-1.0.6-gcc6$ ls /usr/lib/libfl*
/usr/lib/libflint-2.5.2.so  /usr/lib/libflint-2.5.2.so.13.5.2  /usr/lib/libflint.a  /
↪usr/lib/libflint.so
(sage-buildsh) root@2d9ac65f4572:surf-1.0.6-gcc6$ apt-get update && apt-get install␣
↪apt-file
(sage-buildsh) root@2d9ac65f4572:surf-1.0.6-gcc6$ apt-file update
(sage-buildsh) root@2d9ac65f4572:surf-1.0.6-gcc6$ apt-file search "/usr/lib/libfl.a"
flex-old: /usr/lib/libfl.a
freebsd-buildutils: /usr/lib/libfl.a
(sage-buildsh) root@2d9ac65f4572:surf-1.0.6-gcc6$ apt-get install flex-old
(sage-buildsh) root@2d9ac65f4572:surf-1.0.6-gcc6$ ./spkg-install
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
...
  /usr/bin/install -c  surf /sage/local/bin/surf
 /usr/bin/install -c -m 644 ./surf.1 /sage/local/share/man/man1/surf.1
...
make[1]: Leaving directory '/sage/local/var/tmp/sage/build/surf-1.0.6-gcc6/src'
(sage-buildsh) root@2d9ac65f4572:surf-1.0.6-gcc6$ exit
root@2d9ac65f4572:/sage# exit
[mkoeppe@sage sage]$
```

A standard case of bitrot.

Automatic Docker-based build testing using tox

tox is a Python package that is widely used for automating tests of Python projects.

If you are using Docker locally, install `tox` for use with your system Python, for example using:

```
[mkoeppe@sage sage]$ pip install --user tox
```

If you run Docker-in-Docker on GitHub Codespaces using our dev container configuration .devcontainer/tox-docker-in-docker, `tox` is already installed.

Sage provides a sophisticated tox configuration in the file `$SAGE_ROOT/tox.ini` for the purpose of portability testing.

A tox "environment" is identified by a symbolic name composed of several Tox "factors".

The **technology** factor describes how the environment is run:

- `docker` builds a Docker image as described above.
- `local` runs testing on the host OS instead. We explain this technology in a later section.

The next two factors determine the host system configuration: The **system factor** describes a base operating system image.

- Examples are `ubuntu-focal`, `debian-buster`, `archlinux-latest`, `fedora-30`, `slackware-14.2`, `centos-7-i386`, and `ubuntu-bionic-arm64`.
- See `$SAGE_ROOT/tox.ini` for a complete list, and to which images on Docker hub they correspond.

The **packages factor** describes a list of system packages to be installed on the system before building Sage:

- `minimal` installs the system packages known to Sage to provide minimal prerequisites for bootstrapping and building the Sage distribution. This corresponds to the packages `_bootstrap` and `_prereq`.
- `standard` additionally installs all known system packages that are equivalent to standard packages of the Sage distribution, for which the mechanism `spkg-configure.m4` is implemented. This corresponds to the packages listed by:

  ```
  [mkoeppe@sage sage]$ sage --package list --has-file=spkg-configure.m4 :standard:
  ```

- `maximal` does the same for all standard and optional packages. This corresponds to the packages listed by:

  ```
  [mkoeppe@sage sage]$ sage --package list :standard: :optional:
  ```

The factors are connected by a hyphen to name a system configuration, such as `debian-buster-standard` and `centos-7-i386-minimal`.

Finally, the **configuration** factor (which is allowed to be empty) controls how the `configure` script is run.

The factors are connected by a hyphen to name a tox environment. (The order of the factors does not matter; however, for consistency and because the ordered name is used for caching purposes, we recommend to use the factors in the listed order.)

To run an environment:

```
[mkoeppe@sage sage]$ tox -e docker-slackware-14.2-minimal
[mkoeppe@sage sage]$ tox -e docker-ubuntu-bionic-standard
```

Arbitrary extra arguments to `docker build` can be supplied through the environment variable `EXTRA_DOCKER_BUILD_ARGS`. For example, for a non-silent build (`make V=1`), use:

```
[mkoeppe@sage sage]$ EXTRA_DOCKER_BUILD_ARGS="--build-arg USE_MAKEFLAGS=\"V=1\"" \
  tox -e docker-ubuntu-bionic-standard
```

By default, tox uses `TARGETS_PRE=all-sage-local` and `TARGETS=build`, leading to a complete build of Sage without the documentation. If you pass positional arguments to tox (separated from tox options by `--`), then both `TARGETS_PRE` and `TARGETS` are set to these arguments. In this way, you can build some specific packages instead of all of Sage, for example:

```
[mkoeppe@sage sage]$ tox -e docker-centos-8-standard -- ratpoints
```

If the build succeeds, this will create a new image named `sage-centos-8-standard-with-targets:9.1.beta9-431-gca4b5b2f33-dirty`, where

- the image name is derived from the tox environment name and the suffix `with-targets` expresses that the `make` targets given in `TARGETS` have been built;
- the tag name describes the git revision of the source tree as per `git describe --dirty`.

You can ask for tox to create named intermediate images as well. For example, to create the images corresponding to the state of the OS after installing all system packages (`with-system-packages`) and the one just after running the `configure` script (`configured`):

```
[mkoeppe@sage sage]$ DOCKER_TARGETS="with-system-packages configured with-targets" \
  tox -e docker-centos-8-standard -- ratpoints
...
Sending build context to Docker daemon ...
Step 1/109 : ARG BASE_IMAGE=fedora:latest
Step 2/109 : FROM ${BASE_IMAGE} as with-system-packages
...
Step 109/109 : RUN yum install -y zlib-devel || echo "(ignoring error)"
...
Successfully built 4bb14c3d5646
Successfully tagged sage-centos-8-standard-with-system-packages:9.1.beta9-435-
↪g861ba33bbc-dirty
Sending build context to Docker daemon ...
...
Successfully tagged sage-centos-8-standard-configured:9.1.beta9-435-g861ba33bbc-dirty
...
Sending build context to Docker daemon ...
...
Successfully tagged sage-centos-8-standard-with-targets:9.1.beta9-435-g861ba33bbc-
↪dirty
```

Let's verify that the images are available:

```
[mkoeppe@sage sage]$ docker images | head
REPOSITORY                                  TAG                                 IMAGE␣
↪ID
sage-centos-8-standard-with-targets         9.1.beta9-435-g861ba33bbc-dirty     ␣
↪7ecfa86fceab
sage-centos-8-standard-configured           9.1.beta9-435-g861ba33bbc-dirty     ␣
↪4314929e2b4c
sage-centos-8-standard-with-system-packages  9.1.beta9-435-g861ba33bbc-dirty     ␣
↪4bb14c3d5646
...
```

Automatic build testing on the host OS using tox -e local-direct

The `local` technology runs testing on the host OS instead.

In contrast to the `docker` technology, it does not make a copy of the source tree. It is most straightforward to run it from a separate, distclean git worktree.

Let us try a first variant of the `local` technology, the tox environment called `local-direct`. Because all builds with tox begin by bootstrapping the source tree, you will need autotools and other prerequisites installed in your system. See `build/pkgs/_bootstrap/distros/*.txt` for a list of system packages that provide these prerequisites.

We start by creating a fresh (distclean) git worktree:

```
[mkoeppe@sage sage] git worktree add worktree-local
[mkoeppe@sage sage] cd worktree-local
[mkoeppe@sage worktree-local] ls
COPYING.txt ... Makefile ... configure.ac ... src tox.ini
```

Again we build only a small package. Build targets can be passed as positional arguments (separated from tox options by `--`):

```
[mkoeppe@sage worktree-local] tox -e local-direct -- ratpoints
local-direct create: /Users/mkoeppe/.../worktree-local/.tox/local-direct
local-direct run-test-pre: PYTHONHASHSEED='2211987514'
...
src/doc/bootstrap:48: installing src/doc/en/installation/debian.txt...
bootstrap:69: installing 'config/config.rpath'
configure.ac:328: installing 'config/compile'
configure.ac:113: installing 'config/config.guess'
...
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
...
sage-logger -p 'sage-spkg -y -o  ratpoints-2.1.3.p5' '.../worktree-local/logs/pkgs/
↪ratpoints-2.1.3.p5.log'
[ratpoints-2.1.3.p5] installing. Log file: .../worktree-local/logs/pkgs/ratpoints-2.1.
↪3.p5.log
  [ratpoints-2.1.3.p5] successfully installed.
...
  local-direct: commands succeeded
  congratulations :)
```

Let's investigate what happened here:

```
[mkoeppe@sage worktree-local]$ ls -la
total 2576
drwxr-xr-x  35 mkoeppe  staff    1120 Mar 26 22:20 .
drwxr-xr-x  63 mkoeppe  staff    2016 Mar 27 09:35 ..
...
lrwxr-xr-x   1 mkoeppe  staff      10 Mar 26 20:34 .dockerignore -> .gitignore
-rw-r--r--   1 mkoeppe  staff      74 Mar 26 20:34 .git
...
-rw-r--r--   1 mkoeppe  staff    1212 Mar 26 20:41 .gitignore
...
drwxr-xr-x   7 mkoeppe  staff     224 Mar 26 22:11 .tox
...
-rw-r--r--   1 mkoeppe  staff    7542 Mar 26 20:41 Makefile
...
```

(continues on next page)

(continued from previous page)

```
lrwxr-xr-x   1 mkoeppe  staff     114 Mar 26 20:45 config.log -> .tox/local-direct/
↪log/config.log
-rwxr-xr-x   1 mkoeppe  staff   90411 Mar 26 20:46 config.status
-rwxr-xr-x   1 mkoeppe  staff  887180 Mar 26 20:45 configure
-rw-r--r--   1 mkoeppe  staff   17070 Mar 26 20:41 configure.ac
...
lrwxr-xr-x   1 mkoeppe  staff     103 Mar 26 20:45 logs -> .tox/local-direct/log
drwxr-xr-x  24 mkoeppe  staff     768 Mar 26 20:45 m4
lrwxr-xr-x   1 mkoeppe  staff     105 Mar 26 20:45 prefix -> .tox/local-direct/local
-rwxr-xr-x   1 mkoeppe  staff    4868 Mar 26 20:34 sage
drwxr-xr-x  16 mkoeppe  staff     512 Mar 26 20:46 src
-rw-r--r--   1 mkoeppe  staff   13478 Mar 26 20:41 tox.ini
drwxr-xr-x   4 mkoeppe  staff     128 Mar 26 20:46 upstream
```

There is no `local` subdirectory. This is part of a strategy to keep the source tree clean to the extent possible. In particular:

- `tox` configured the build to use a separate `$SAGE_LOCAL` hierarchy in a directory under the tox environment directory `.tox/local-direct`. It created a symbolic link `prefix` that points there, for convenience:

  ```
  [mkoeppe@sage worktree-local]$ ls -l prefix/lib/*rat*
  -rw-r--r--  1 mkoeppe  staff  165968 Mar 26 20:46 prefix/lib/libratpoints.a
  ```

- Likewise, it created a separate `logs` directory, again under the tox environment directory, and a symbolic link.

This makes it possible for advanced users to test several `local` tox environments (such as `local-direct`) out of one worktree. However, because a build still writes configuration scripts and build artefacts (such as `config.status`) into the worktree, only one `local` build can run at a time in a given worktree.

The tox environment directory will be reused for the next `tox` run, which will therefore do an incremental build. To start a fresh build, you can use the `-r` option.

Automatic build testing on the host OS with best-effort isolation using tox -e local

`tox -e local` (without `-direct`) attempts a best-effort isolation from the user's environment as follows:

- All environment variables are set to standard values; with the exception of `MAKE` and `EXTRA_CONFIGURE_ARGS`. In particular, `PATH` is set to just `/usr/bin:/bin:/usr/sbin:/sbin`; it does not include `/usr/local/bin`.

Note, however, that various packages have build scripts that use `/usr/local` or other popular file system locations such as `/opt/sfw/`. Therefore, the isolation is not complete. Using `/usr/local` is considered standard behavior. On the other hand, we consider a package build script that inspects other file system locations to be a bug of the Sage distribution, which should be reported and fixed on a ticket.

Automatic build testing on macOS with a best-effort isolated installation of Homebrew

XCode on macOS does not provide the prerequisites for bootstrapping the Sage distribution. A good way to install them is using the Homebrew package manager.

In fact, Sage provides a tox environment that automatically installs an isolated copy of Homebrew with all prerequisites for bootstrapping:

```
[mkoeppe@sage worktree-local]$ tox -e local-homebrew-macos-minimal -- lrslib
local-homebrew-macos-minimal create: .../worktree-local/.tox/local-homebrew-macos-
↪minimal
local-homebrew-macos-minimal run-test-pre: PYTHONHASHSEED='4246149402'
...
Initialized empty Git repository in .../worktree-local/.tox/local-homebrew-macos-
↪minimal/homebrew/.git/
...
Tapped 2 commands and 4942 formulae (5,205 files, 310.7MB).
==> Downloading https://ftp.gnu.org/gnu/gettext/gettext-0.20.1.tar.xz
...
==> Pouring autoconf-2.69.catalina.bottle.4.tar.gz
...
==> Pouring pkg-config-0.29.2.catalina.bottle.1.tar.gz
  .../worktree-local/.tox/local-homebrew-macos-minimal/homebrew/Cellar/pkg-config/0.
↪29.2: 11 files, 623.4KB
==> Caveats
==> gettext
gettext is keg-only, which means it was not symlinked into .../worktree-local/.tox/
↪local-homebrew-macos-minimal/homebrew,
because macOS provides the BSD gettext library & some software gets confused if both␣
↪are in the library path.

If you need to have gettext first in your PATH run:
  echo 'export PATH=".../worktree-local/.tox/local-homebrew-macos-minimal/homebrew/
↪opt/gettext/bin:$PATH"' >> ~/.bash_profile

For compilers to find gettext you may need to set:
  export LDFLAGS="-L.../worktree-local/.tox/local-homebrew-macos-minimal/homebrew/opt/
↪gettext/lib"
  export CPPFLAGS="-I.../worktree-local/.tox/local-homebrew-macos-minimal/homebrew/
↪opt/gettext/include"
...
local-homebrew-macos-minimal run-test: commands[0] | bash -c 'export PATH=.../
↪worktree-local/.tox/local-homebrew-macos-minimal/homebrew/bin:/usr/bin:/bin:/usr/
↪sbin:/sbin && . .homebrew-build-env && ./bootstrap && ./configure --prefix=.../
↪worktree-local/.tox/local-homebrew-macos-minimal/local    && make -k V=0 ... lrslib'
...
bootstrap:69: installing 'config/config.rpath'
...
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
...
configure: notice: the following SPKGs did not find equivalent system packages: arb␣
↪cbc cliquer ... tachyon xz zeromq
checking for the package system in use... homebrew
configure: hint: installing the following system packages is recommended and may␣
↪avoid building some of the above SPKGs from source:
configure:   $ brew install cmake gcc gsl mpfi ninja openblas gpatch r readline xz␣
↪zeromq
...
sage-logger -p 'sage-spkg -y -o  lrslib-062+autotools-2017-03-03.p1' '.../worktree-
↪local/logs/pkgs/lrslib-062+autotools-2017-03-03.p1.log'
[lrslib-062+autotools-2017-03-03.p1] installing. Log file: .../worktree-local/logs/
↪pkgs/lrslib-062+autotools-2017-03-03.p1.log
  [lrslib-062+autotools-2017-03-03.p1] successfully installed.
...
```

(continues on next page)

(continued from previous page)

```
  local-homebrew-macos-minimal: commands succeeded
  congratulations :)
```

The tox environment uses the subdirectory `homebrew` of the environment directory `.tox/local-homebrew-macos-minimal` as the Homebrew prefix. This installation does not interact in any way with a Homebrew installation in `/usr/local` that you may have.

The test script sets the `PATH` to the `bin` directory of the Homebrew prefix, followed by `/usr/bin:/bin:/usr/sbin:/sbin`. It then uses the script `$SAGE_ROOT/.homebrew-build-env` to set environment variables so that Sage's build scripts will find "keg-only" packages such as `gettext`.

The `local-homebrew-macos-minimal` environment does not install Homebrew's `python3` package. It uses XCode's `/usr/bin/python3` as system python. However, because various packages are missing that Sage considers as dependencies, Sage builds its own copy of these packages and of `python3`.

The `local-homebrew-macos-standard` environment additionally installs (in its separate isolated copy of Homebrew) all Homebrew packages known to Sage for which the `spkg-configure.m4` mechanism is implemented; this is similar to the `docker-standard` tox environments described earlier. In particular it installs and uses Homebrew's `python3` package.

By using configuration factors, more variants can be tested. The `local-homebrew-macos-standard-python3_xcode` environment installs the same packages, but uses XCode's `/usr/bin/python3`.

The `local-homebrew-macos-standard-python3_pythonorg` expects an installation of Python 3.10 in `/Library/Frameworks/Python.framework`; this is where the binary packages provided by python.org install themselves.

Automatic build testing with a best-effort isolated installation of Conda

Sage provides environments `local-conda-forge-standard` and `local-conda-forge-minimal` that create isolated installations of Miniconda in the subdirectory `conda` of the environment directory. They do not interact in any way with other installations of Anaconda or Miniconda that you may have on your system.

The environments use the conda-forge channel and use the `python` package and the compilers from this channel.

Options for build testing with the local technology

The environments using the `local` technology can be customized by setting environment variables.

- If `SKIP_SYSTEM_PKG_INSTALL` is set to `1` (or `yes`), then all steps of installing system packages are skipped in this run. When reusing a previously created tox environment, this option can save time and also give developers more control for experiments with system packages.
- If `SKIP_BOOTSTRAP` is set to `1` (or `yes`), then the bootstrapping phase is skipped. When reusing a previously created tox environment, this option can save time.
- If `SKIP_CONFIGURE` is set to `1` (or `yes`), then the `configure` script is not run explicitly. When reusing a previously created tox environment, this option can save time. (The `Makefile` may still rerun configuration using `config.status --recheck`.)

The `local` technology also defines a special target `bash`: Instead of building anything with `make`, it just starts an interactive shell. For example, in combination with the above options:

```
[mkoeppe@sage worktree-local]$ SKIP_SYSTEM_PKG_INSTALL=yes SKIP_BOOTSTRAP=1 SKIP_
↪CONFIGURE=1 tox -e local-homebrew-macos-minimal -- bash
```

Automatic testing on multiple platforms on GitHub Actions

The Sage source tree includes a default configuration for GitHub Actions that runs our portability tests on a multitude of platforms on every push of a tag (but not of a branch) to a repository for which GitHub Actions are enabled.

In particular, it automatically runs on our main repository sagemath/sage on every release tag.

This is defined in the files

- $SAGE_ROOT/.github/workflows/ci-linux.yml (which calls $SAGE_ROOT/.github/workflows/docker.yml) and
- $SAGE_ROOT/.github/workflows/ci-macos.yml.

GitHub Actions runs these build jobs on 2-core machines with 7 GB of RAM memory and 14 GB of SSD disk space, cf. here, and has a time limit of 6h per job. This could be just barely enough for a typical `minimal` build followed by `make ptest` to succeed; for added robustness, we split it into two jobs. Our workflow stores Docker images corresponding to various build phases within these two jobs on GitHub Packages (ghcr.io).

Build logs can be inspected during the run and become available as "artifacts" when all jobs of the workflow have finished. Each job generates one tarball. "Annotations" highlight certain top-level errors or warnings issued during the build.

In addition to these automatic runs in our main repository, all Sage developers can run the same tests on GitHub Actions in their personal forks of the Sage repository. To prepare this, enable GitHub Actions in your fork of the Sage repository.

As usual we assume that `origin` is the name of the remote corresponding to your GitHub fork of the Sage repository:

```
$ git remote -v | grep origin
origin         https://github.com/mkoeppe/sage.git (fetch)
origin         https://github.com/mkoeppe/sage.git (push)
```

Then the following procedure triggers a run of tests with the default set of system configurations.

- Push your branch to `origin` (your fork).
- Go to the Actions tab of your fork and select the workflow you would like to run, for example "CI Linux".
- Click on "Run workflow" above the list of workflow runs and select your branch as the branch on which the workflow will run.

For more information, see the GitHub documentation.

Alternatively, you can trigger a run of tests by creating and pushing a custom tag as follows.

- Create a ("lightweight", not "annotated") tag with an arbitrary name, say `ci` (for "Continuous Integration"):

  ```
  git tag -f ci
  ```

- Then push the tag to your GitHub repository:

  ```
  git push -f origin ci
  ```

(In both commands, the "force" option (`-f`) allows overwriting a previous tag of that name.)

Either way, when the workflow has been triggered, you can inspect it by using the workflow status page in the "Actions" tab of your repository.

Here is how to read it. Each of the items in the left pane represents a full build of Sage on a particular system configuration. A test item in the left pane is marked with a green checkmark in the left pane if `make build doc-html` finished without error. (It also runs package testsuites and the Sage doctests but failures in these are not reflected in the left pane; see below.)

The right pane ("Artifacts") offers archives of the logs for download.

Scrolling down in the right pane shows "Annotations":

- Red "check failure" annotations appear for each log file that contains a build error. For example, you might see:

```
docker (fedora-28, standard)
artifacts/logs-commit-8ca1c2df8f1fb4c6d54b44b34b4d8320ebecb164-tox-docker-fedora-
↪28-standard/logs/pkgs/sagetex-3.4.log#L1
==== ERROR IN LOG FILE artifacts/logs-commit-
↪8ca1c2df8f1fb4c6d54b44b34b4d8320ebecb164-tox-docker-fedora-28-standard/logs/
↪pkgs/sagetex-3.4.log ====
```

- Yellow "check warning" annotations. There are 2 types of these:

  a) Package testsuite or Sage doctest failures, like the following:

```
docker (fedora-30, standard)
artifacts/logs-commit-8ca1c2df8f1fb4c6d54b44b34b4d8320ebecb164-tox-docker-
↪fedora-30-standard/logs/ptest.log#L1
==== TESTSUITE FAILURE IN LOG FILE artifacts/logs-commit-
↪8ca1c2df8f1fb4c6d54b44b34b4d8320ebecb164-tox-docker-fedora-30-standard/logs/
↪ptest.log ====
```

  b) Notices from ./configure about not finding equivalent system packages, like the following:

```
docker (fedora-31, standard)
artifacts/logs-commit-8ca1c2df8f1fb4c6d54b44b34b4d8320ebecb164-tox-docker-
↪fedora-31-standard/config.log#L1
configure: notice: the following SPKGs did not find equivalent system␣
↪packages: arb cbc cddlib cmake eclib ecm fflas_ffpack flint fplll givaro gp
```

Clicking on the annotations does not take you to a very useful place. To view details, click on one of the items in the pane. This changes the right pane to a log viewer.

The `docker` workflows automatically push images to `ghcr.io`. You find them in the Packages tab of your GitHub repository.

In order to pull them for use on your computer, you need to first generate a Personal Access Token providing the `read:packages` scope as follows. Visit https://github.com/settings/tokens/new (this may prompt you for your GitHub password). As "Note", type "Access ghcr.io"; then in "Select scopes", select the checkbox for `read:packages`. Finally, push the "Generate token" button at the bottom. This will lead to a page showing your token, such as `de1ec7ab1ec0ffee5ca1dedbaff1ed0ddba11`. Copy this token and paste it to the command line:

```
$ echo de1ec7ab1ec0ffee5ca1dedbaff1ed0ddba11 | docker login ghcr.io --username YOUR-
↪GITHUB-USERNAME
```

where you replace the token by your token, of course, and `YOUR-GITHUB-USERNAME` by your GitHub username.

Now you can pull the image and run it:

```
$ docker pull ghcr.io/YOUR-GITHUB-USERNAME/sage/sage-fedora-31-standard-
↪configured:f4bd671
$ docker run -it ghcr.io/YOUR-GITHUB-USERNAME/sage/sage-fedora-31-standard-
↪configured:f4bd671 bash
```

Using our pre-built Docker images published on ghcr.io

Our portability CI on GitHub Actions builds Docker images for all tested Linux platforms (and system package configurations) and makes them available on GitHub Packages (ghcr.io).

This makes it easy for developers to debug problems that showed up in the build logs for a given platform.

The image version corresponding to the latest development release receives the additional Docker tag `dev`, see for example the Docker image for the platform ubuntu-focal-standard. Thus, for example, the following command will work:

```
$ docker run -it ghcr.io/sagemath/sage/sage-ubuntu-focal-standard-with-targets-
↪optional:dev bash
Unable to find image 'ghcr.io/sagemath/sage/sage-ubuntu-focal-standard-with-targets-
↪optional:dev' locally
dev: Pulling from sagemath/sage/sage-ubuntu-focal-standard-with-targets-optional
d5fd17ec1767: Already exists
67586203f0c7: Pull complete
b63c529f4777: Pull complete
...
159775d1a3d2: Pull complete
Digest: sha256:e6ba5e12f59c6c4668692ef4cfe4ae5f242556482664fb347bf260f32bf8e698
Status: Downloaded newer image for ghcr.io/sagemath/sage/sage-ubuntu-focal-standard-
↪with-targets-optional:dev
root@8055a7ba0607:/sage# ./sage
┌────────────────────────────────────────────────────────────────────┐
│ SageMath version 9.6, Release Date: 2022-05-15                     │
│ Using Python 3.8.10. Type "help()" for help.                       │
└────────────────────────────────────────────────────────────────────┘
sage:
```

Images whose names end with the suffix `-with-targets-optional` are the results of full builds and a run of `make ptest`. They also contain a copy of the source tree and the full logs of the build and test.

Also smaller images corresponding to earlier build stages are available:

- `-with-system-packages` provides a system installation with system packages installed, no source tree,
- `-configured` contains a partial source tree (`SAGE_ROOT`) and has completed the bootstrapping phase and the run of the `configure` script,
- `-with-targets-pre` contains a partial source tree (`SAGE_ROOT`) and a full installation of all non-Python packages (`SAGE_LOCAL`),
- `-with-targets` contains the full source tree and a full installation of Sage, including the HTML documentation, but `make ptest` has not been run yet.

Using our pre-built Docker images for development in VS Code

VS Code is very convenient for developing with Docker containers thanks to the Visual Studio Code Dev Containers extension.

If the extension is not already installed, then in VS Code, click the "Extension" icon on the left (or press `Ctrl` + `Shift` + `X`; on macOS, `Command` + `Shift` + `X`) to open a list of extensions. Search for "Dev Containers" and install it.

The extension needs a `devcontainer.json` configuration file to work. Sage provides sample `devcontainer.json` configuration files $SAGE_ROOT/.devcontainer/*/devcontainer.json for this purpose.

If you open the sage folder in VS Code, it may prompt you whether you would like to open the current directory in the dev container (yes). If it does not, use the command palette (`Ctrl` + `Shift` + `P`), enter the command "Dev Containers: Reopen Folder in Container" , and hit `Enter`.

If the above `code .` command does not work, start VS Code as a regular application, then in the command palette of VS Code, enter "Dev Containers: Open Folder in Container", and hit `Enter`, and choose the directory `$SAGE_ROOT` of your local Sage repository.

VS Code then prompts you to choose a dev container configuration. For example, choose "ubuntu-jammy-standard" .devcontainer/portability-ubuntu-jammy-standard/devcontainer.json, which uses the Docker image based on `ubuntu-jammy-standard`, the most recent development version of Sage (`dev` tag), and a full installation of the Sage distribution (`with-targets`). Other dev container configurations are described below.

Once VS Code starts configuring the dev container, by clicking on "show log", you can see what it does:

- It pulls the prebuilt image from ghcr.io (via $SAGE_ROOT/.devcontainer/portability-Dockerfile); note that these are multi-gigabyte images, so it may take a while.
- As part of the "onCreateCommand", it installs additional system packages to support VS Code and for development.
- Then, as part of the "updateContentCommand", it bootstraps and configures the source tree and starts to build Sage from source, reusing the installation (`SAGE_LOCAL`, `SAGE_VENV`) from the prebuilt image.

After VS Code finished configuring the dev container (when the message "Done. Press any key to close the terminal." appears in the terminal named "Configuring"), your local Sage repository at `$SAGE_ROOT` is available in the container at the directory `/workspaces/<repository name>`. To use Sage in a terminal, open a new terminal in VS Code, type `./sage` and hit `Enter`.

Note: Your Sage at `$SAGE_ROOT` was configured and rebuilt inside the dev container. In particular, `$SAGE_ROOT/venv`, `$SAGE_ROOT/prefix`, and (possibly) `$SAGE_ROOT/logs` will be symbolic links that work inside the dev container, but not in your local file system; and also the script `$SAGE_ROOT/sage` will not work. Hence after working with the dev container, you will want to remove `logs` if it is a symbolic link, and rerun the `configure` script.

The Sage source tree contains premade configuration files for all platforms for which our portability CI builds Docker images, both in the `minimal` and `standard` system package configurations. The configuration files can be generated using the command `tox -e update_docker_platforms` (see $SAGE_ROOT/tox.ini for environment variables that take effect).

You can edit a copy of the configuration file to change to a different platform, another version, or build stage. After editing the configuration file, run "Dev Containers: Rebuild Container" from the command palette. See the VS Code devcontainer.json reference and the GitHub introduction to dev containers for more information.

In addition to the `$SAGE_ROOT/.devcontainer/portability-.../devcontainer.json` files, Sage also provides several other sample `devcontainer.json` configuration files in the directory `$SAGE_ROOT/.devcontainer`.

Files named `$SAGE_ROOT/.devcontainer/develop-.../devcontainer.json` configure containers from a public Docker image that provides SageMath and then updates the installation of SageMath in this container by building from the current source tree.

- develop-docker-computop/devcontainer.json configures a container with the Docker image from the 3-manifolds project, providing SnapPy, Regina, PHCPack, etc.

After VS Code finished configuring the dev container, to use Sage in a terminal, open a new terminal in VS Code, type `./sage` and hit `Enter`.

Files named `$SAGE_ROOT/.devcontainer/downstream-.../devcontainer.json` configure containers with an installation of downstream packages providing SageMath from a package manager (`downstream-archlinux-...`, `downstream-conda-forge`; see also the _sagemath dummy package), or from a public Docker image that provides SageMath (`docker-cocalc`, `docker-computop`). These `devcontainer.json` configuration files are useful for testing user scripts on these deployments of SageMath. You may also find it useful to copy these configurations into your own projects (they should work without change) or to adapt them to your needs.

- downstream-archlinux-latest/devcontainer.json configures a container with an installation of Arch Linux and its SageMath package. (The suffix `latest` indicates the most recent version of Arch Linux as available on Docker Hub.)
- downstream-conda-forge-latest/devcontainer.json configures a container with an installation of conda-forge and its SageMath package.
- downstream-docker-cocalc/devcontainer.json configures a container with the CoCalc Docker image.
- downstream-docker-computop/devcontainer.json configures a container with the Docker image from the 3-manifolds project, providing SnapPy, Regina, PHCPack, etc.

After VS Code finished configuring the dev container, to use Sage in a terminal, open a new terminal in VS Code, type `sage` and hit `Enter`. (Do not use `./sage`; this will not work because the source tree is not configured.)

1.5.3 Development and Testing Tools

Tox

tox is a popular package that is used by a large number of Python projects as the standard entry point for testing and linting.

Sage includes tox as a standard package and uses it for three purposes:

- For portability testing of the Sage distribution, as we explain in *Testing on Multiple Platforms*. This is configured in the file `SAGE_ROOT/tox.ini`.
- For testing modularized distributions of the Sage library. This is configured in `tox.ini` files in subdirectories of `SAGE_ROOT/pkgs/`, such as `SAGE_ROOT/pkgs/sagemath-standard/tox.ini`. Each distribution's configuration defines tox environments for testing the distribution with different Python versions and different ways how the dependencies are provided. We explain this in *Packaging the Sage Library for Distribution*.
- As an entry point for testing and linting of the Sage library, as we describe below. This is configured in the file `SAGE_ROOT/src/tox.ini`.

The tox configuration `SAGE_ROOT/src/tox.ini` can be invoked by using the command `./sage --tox`. (If `tox` is available in your system installation, you can just type `tox` instead.)

This configuration provides an entry point for various testing/linting methods, known as "tox environments". We can type `./sage --advanced` to see what is available:

```
$ ./sage --advanced
SageMath version 9.2
...
Testing files:
...
--tox [options] <files|dirs> -- general entry point for testing
                                and linting of the Sage library
   -e <envlist>     -- run specific test environments; default:
                       doctest,coverage,startuptime,pycodestyle-minimal,relint,
↪codespell,rst
      doctest               -- run the Sage doctester
                               (same as "sage -t")
      coverage              -- give information about doctest coverage of files
                               (same as "sage --coverage[all]")
      startuptime           -- display how long each component of Sage takes to␣
↪start up
                               (same as "sage --startuptime")
      pycodestyle-minimal   -- check against Sage's minimal style conventions
```

(continues on next page)

(continued from previous page)

```
     relint                -- check whether some forbidden patterns appear
                              (includes all patchbot pattern-exclusion plugins)
     codespell             -- check for misspelled words in source code
     rst                   -- validate Python docstrings markup as reStructuredText
     coverage.py           -- run the Sage doctester with Coverage.py
     coverage.py-html      -- run the Sage doctester with Coverage.py, generate␣
↪HTML report
     pyright               -- run the static typing checker pyright
     pycodestyle           -- check against the Python style conventions of PEP8
     cython-lint           -- check Cython files for code style
   -p auto          -- run test environments in parallel
   --help           -- show tox help
```

Doctest

The command `./sage -tox -e doctest` runs the Sage doctester. This is equivalent to using the command `./sage -t`; see *Running Sage's Doctests*.

Note: `doctest` is a special tox environment that requires that Sage has been built already. A virtual environment is created by tox, but Sage is invoked in the normal Sage environment.

Doctest with Coverage.py

The command `./sage -tox -e coverage.py` runs the Sage doctester (*Running Sage's Doctests*) in the normal Sage environment, but under the control of Coverage.py for code coverage analysis.

If invoked as `./sage -tox -e coverage.py-html`, additionally a detailed HTML report is generated.

Configuration: `[coverage:run]` block in `SAGE_ROOT/src/tox.ini`

Documentation: https://coverage.readthedocs.io

Note: `coverage.py` is a special tox environment that requires that Sage has been built already. A virtual environment is created by tox, but the **coverage** package is installed into the normal Sage environment, and Sage is invoked from there.

Coverage

The command `./sage -tox -e coverage` checks that each function has at least one doctest (typically in an **EXAMPLES** or **TESTS** block, see *The docstring of a function: content*).

Without additional arguments, this command is equivalent to using the command `./sage --coverageall` and gives a short report with a one-line summary for each module of the Sage library.

If invoked with arguments, for example `./sage -tox -e coverage -- src/sage/geometry src/sage/combinat/tableau.py`, it is equivalent to using the command `./sage --coverage`, which includes details on the modules in the given files or directories.

Note: `coverage` is a special tox environment that requires that Sage has been built already. A virtual environment is created by tox, but Sage is invoked in the normal Sage environment.

Startuptime

The command `./sage -tox -e startuptime` measures the time for loading each module that is imported during the start up phase of Sage. It is equivalent to using the command `./sage --startuptime`.

Without additional arguments, the command gives a short report that lists the modules with the longest contributions to the overall startup time, sorted by time.

If invoked with arguments, for example `sage -tox -e startuptime -- sage.rings src/sage/geometry/polyhedron`, it provides details on the given modules, packages, source files, or directories.

Note: `startuptime` is a special tox environment that requires that Sage has been built already. A virtual environment is created by tox, but Sage is invoked in the normal Sage environment.

Pycodestyle

Pycodestyle (formerly known as pep8) checks Python code against the style conventions of PEP 8. Tox automatically installs pycodestyle in a separate virtual environment on the first use.

Sage defines two configurations for pycodestyle. The command `./sage -tox -e pycodestyle-minimal` uses pycodestyle in a minimal configuration. As of Sage 9.5, the entire Sage library conforms to this configuration:

```
$ ./sage -tox -e pycodestyle-minimal -- src/sage/
pycodestyle-minimal installed: pycodestyle==2.8.0
pycodestyle-minimal run-test-pre: PYTHONHASHSEED='28778046'
pycodestyle-minimal run-test: commands[0] | pycodestyle --select E401,E70,W605,E711,
↪E712,E721 sage
___________ summary ____________
  pycodestyle-minimal: commands succeeded
  congratulations :)
```

When preparing a branch for a Sage ticket, developers should verify that `./sage -tox -e pycodestyle-minimal` passes. When the Sage patchbot runs on the ticket, it will perform similar coding style checks; but running the check locally reduces the turnaround time from hours to seconds.

The second configuration is used with the command `./sage -tox -e pycodestyle` and runs a more thorough check:

```
$ ./sage -tox -e pycodestyle -- src/sage/quadratic_forms/quadratic_form.py
pycodestyle installed: pycodestyle==2.8.0
pycodestyle run-test-pre: PYTHONHASHSEED='2520226550'
pycodestyle run-test: commands[0] | pycodestyle sage/quadratic_forms/quadratic_form.py
sage/quadratic_forms/quadratic_form.py:135:9: E225 missing whitespace around operator
sage/quadratic_forms/quadratic_form.py:163:64: E225 missing whitespace around operator
sage/quadratic_forms/quadratic_form.py:165:52: E225 missing whitespace around operator
sage/quadratic_forms/quadratic_form.py:173:42: E228 missing whitespace around modulo␣
↪operator
...
sage/quadratic_forms/quadratic_form.py:1620:9: E266 too many leading '#' for block␣
↪comment
sage/quadratic_forms/quadratic_form.py:1621:9: E266 too many leading '#' for block␣
↪comment
25      E111 indentation is not a multiple of 4
2       E117 over-indented
129     E127 continuation line over-indented for visual indent
```

(continues on next page)

(continued from previous page)

```
1       E128 continuation line under-indented for visual indent
4       E201 whitespace after '['
4       E202 whitespace before ']'
2       E222 multiple spaces after operator
7       E225 missing whitespace around operator
1       E228 missing whitespace around modulo operator
25      E231 missing whitespace after ','
1       E262 inline comment should start with '# '
3       E265 block comment should start with '# '
62      E266 too many leading '#' for block comment
2       E272 multiple spaces before keyword
2       E301 expected 1 blank line, found 0
17      E303 too many blank lines (2)
ERROR: InvocationError for command .../pycodestyle sage/quadratic_forms/quadratic_
↪form.py (exited with code 1)
_____________ summary _____________
ERROR:   pycodestyle: commands failed
```

When preparing a branch for a PR that adds new code, developers should verify that `./sage -tox -e pycodestyle` does not issue warnings for the added code. This will avoid later cleanup PRs as the Sage codebase is moving toward full PEP 8 compliance.

On the other hand, it is usually not advisable to mix coding-style fixes with productive changes on the same PR because this would makes it harder for reviewers to evaluate the changes.

By passing the options `--count -qq` we can reduce the output to only show the number of style violation warnings. This can be helpful for planning work on coding-style clean-up PRs that focus on one or a few related issues:

```
$ ./sage -tox -e pycodestyle -- --count -qq src/sage
pycodestyle installed: pycodestyle==2.8.0
pycodestyle run-test-pre: PYTHONHASHSEED='3166223974'
pycodestyle run-test: commands[0] | pycodestyle --count -qq sage
557     E111 indentation is not a multiple of 4
1       E112 expected an indented block
194     E114 indentation is not a multiple of 4 (comment)
...
7       E743 ambiguous function definition 'l'
335     W291 trailing whitespace
4       W292 no newline at end of file
229     W293 blank line contains whitespace
459     W391 blank line at end of file
97797
ERROR: InvocationError for command .../pycodestyle --count -qq sage (exited with code␣
↪1)
_____________ summary _____________
ERROR:   pycodestyle: commands failed
```

Installation: (for manual use:) `pip install -U pycodestyle --user`

Usage:

- With tox: See above.
- Manual: Run `pycodestyle path/to/the/file.py`.
- VS Code: The minimal version of pycodestyle is activated by default in `SAGE_ROOT/.vscode/settings.json` (the corresponding setting is `"python.linting.pycodestyleEnabled": true`). Note that the `settings.json` file is not ignored by Git so be aware to keep it in sync with the Sage repo on GitHub. For further details, see the official VS Code documentation.

Configuration: `[pycodestyle]` block in `SAGE_ROOT/src/tox.ini`

Documentation: https://pycodestyle.pycqa.org/en/latest/index.html

Cython-lint

Cython-lint checks Cython source files for coding style.

Ruff

Ruff is a powerful and fast linter for Python code, written in Rust.

It comes with a large choice of possible checks, and has the capacity to fix some of the warnings it emits.

Relint

Relint checks all source files for forbidden text patterns specified by regular expressions.

Our configuration of relint flags some outdated Python constructions, plain TeX commands when equivalent LaTeX commands are available, common mistakes in documentation markup, and modularization anti-patterns.

Configuration: `SAGE_ROOT/src/.relint.yml`

Documentation: https://pypi.org/project/relint/

Codespell

Codespell uses a dictionary to check for misspelled words in source code.

Sage defines a configuration for codespell:

```
$ ./sage -tox -e codespell -- src/sage/homology/
codespell installed: codespell==2.1.0
codespell run-test-pre: PYTHONHASHSEED='1285169064'
codespell run-test: commands[0] | codespell '--skip=*.png,*.jpg,*.JPG,*.inv,*.dia,*.
↪pdf,*.ico,*#*,*~*,*.bak,*.orig,*.log,*.sobj,*.tar,*.gz,*.pyc,*.o,*.sws,*.so,*.a,.DS_
↪Store' --skip=doc/ca,doc/de,doc/es,doc/hu,doc/ja,doc/ru,doc/fr,doc/it,doc/pt,doc/tr␣
↪--skip=src/doc/ca,src/doc/de,src/doc/es,src/doc/hu,src/doc/ja,src/doc/ru,src/doc/fr,
↪src/doc/it,src/doc/pt,src/doc/tr '--skip=.git,.tox,worktree*,dist,upstream,logs,
↪local,cythonized,scripts-3,autom4te.cache,tmp,lib.*,*.egg-info' --dictionary=- --
↪dictionary=/Users/mkoeppe/s/sage/sage-rebasing/src/.codespell-dictionary.txt --
↪ignore-words=/Users/mkoeppe/s/sage/sage-rebasing/src/.codespell-ignore.txt sage/
↪homology
sage/homology/hochschild_complex.py:271: mone ==> mono, money, none
sage/homology/hochschild_complex.py:277: mone ==> mono, money, none
sage/homology/hochschild_complex.py:280: mone ==> mono, money, none
sage/homology/chain_complex.py:2185: mor ==> more
sage/homology/chain_complex.py:2204: mor ==> more
sage/homology/chain_complex.py:2210: mor ==> more
sage/homology/chain_complex.py:2211: mor ==> more
sage/homology/chain_complex.py:2214: mor ==> more
sage/homology/chain_complex.py:2215: mor ==> more
ERROR: InvocationError for command .../codespell '--skip=*.png,...' --dictionary=- --
↪dictionary=/Users/mkoeppe/s/sage/sage-rebasing/src/.codespell-dictionary.txt --
↪ignore-words=/Users/mkoeppe/s/sage/sage-rebasing/src/.codespell-ignore.txt sage/
```

(continues on next page)

(continued from previous page)

```
↪homology (exited with code 65)
____________ summary _____________
ERROR:   codespell: commands failed
```

Configuration:

- `[testenv:codespell]` block in `SAGE_ROOT/src/tox.ini`
- `SAGE_ROOT/src/.codespell-dictionary.txt` and `SAGE_ROOT/src/.codespell-ignore.txt`

Pytest

Pytest is a testing framework. It is included in the Sage distribution as an optional package.

Currently, Sage only makes very limited use of pytest, for testing the package `sage.numerical.backends` and some modules in `sage.manifolds`.

Installation:

- `./sage -i pytest pytest_xdist`.

Usage:

- Tox, Sage doctester: At the end of `./sage -t` (or `./sage --tox -e doctest`), Pytest is automatically invoked.
- Manual: Run `./sage -pytest path/to/the/test_file.py` or `./sage -pytest` to run all tests. The additional argument `-n` can be used to distribute tests across multiple CPUs to speed up test execution. For example, `./sage -pytest -n 4` will run 4 tests in parallel, while `./sage -pytest -n auto` will spawn a number of workers processes equal to the number of available CPUs.
- VS Code: Install the Python extension and follow the offical VS Code documentation.

Configuration: `SAGE_ROOT/src/conftest.py`

Documentation: https://docs.pytest.org/en/stable/index.html

Pyright

Pyright is static type checker.

Installation:

- (for manual use:) `npm install -g pyright`, see documentation for details.

Usage:

- Tox: Run `./sage -tox -e pyright path/to/the/file.py`
- Manual: Run `pyright path/to/the/file.py`. If you want to check the whole Sage library, you most likely run out of memory with the default settings. You can use the following command to check the whole library:

  ```
  NODE_OPTIONS="--max-old-space-size=8192" pyright
  ```

- VS Code: Install the Pylance extension.

Configuration: `SAGE_ROOT/pyrightconfig.json`

Documentation: https://github.com/microsoft/pyright#documentation

Pyflakes

Pyflakes checks for common coding errors.

Act

act is a tool, written in Go, and using Docker, to run GitHub Actions locally; in particular, it speeds up developing Actions. We recommend using `gh extension` facility to install `act`.

```
[alice@localhost sage]$ gh extension install https://github.com/nektos/gh-act
```

Extra steps needed for configuration of Docker to run Actions locally can be found on act's GitHub

Here we give a very short sampling of `act`'s capabilities. If you installed standalone `act`, it should be invoked as `act`, not as `gh act`. After the set up, one can e.g. list all the available linting actions:

```
[alice@localhost sage]$ gh act -l | grep lint
0       lint-pycodestyle        Code style check with pycodestyle                    ␣
↪     Lint                                              lint.yml                 push,
↪pull_request
0       lint-relint             Code style check with relint                         ␣
↪     Lint                                              lint.yml                 push,
↪pull_request
0       lint-rst                Validate docstring markup as RST                     ␣
↪     Lint                                              lint.yml                 push,
↪pull_request
[alice@localhost sage]$
```

run a particular action `lint-rst`

```
[alice@localhost sage]$ gh act -j lint-rst
...
```

and so on.

By default, `act` pulls all the data needed from the next, but it can also cache it, speeding up repeated runs quite a lot. The following repeats running of `lint-rst` using cached data:

```
[alice@localhost sage]$ gh act -p false -r -j lint-rst
[Lint/Validate docstring markup as RST]   Start image=catthehacker/ubuntu:act-latest
...
| rst: commands[0] /home/alice/work/software/sage/src> flake8 --select=RST
|   rst: OK (472.60=setup[0.09]+cmd[472.51] seconds)
|   congratulations :) (474.10 seconds)
...
[Lint/Validate docstring markup as RST]     Success - Main Lint using tox -e rst
[Lint/Validate docstring markup as RST]  Run Post Set up Python
[Lint/Validate docstring markup as RST]     docker exec cmd=[node /var/run/act/
↪actions/actions-setup-python@v4/dist/cache-save/index.js] user= workdir=
[Lint/Validate docstring markup as RST]     Success - Post Set up Python
[Lint/Validate docstring markup as RST]   Job succeeded
```

Here `-p false` means using already pulled Docker images, and `-r` means do not remove Docker images after a successful run which used them. This, and many more details, can be found by running `gh act -h`, as well as reading `act`'s documentation.

1.6 Updating Sage Documentation

1.6.1 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

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/local/share/doc/sage/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 doctests*):

```
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*:

GitHub issue	`issue:`17596``	github issue #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>`_ ``

Anonymous hyperlinks: Using a single underscore creates an *explicit target name* `"link name"` which needs to be unique in the current page. Using the same target name twice in the same page creates an error while building the documentation saying `WARNING: Duplicate explicit target name: ...`. To avoid this issue, one can change the target names to be all different or another option is to use anonymous hyperlinks with two underscores, as in `` see `this page <http://www.example.com>`__ or `this page <http://www.example2.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`.

Making portions of the reference manual conditional on optional features

For every dynamically detectable feature such as `graphviz` or `sage.symbolic` (see `sage.features`), Sage defines a Sphinx tag that can be used with the Sphinx directive ".. ONLY::". Because Sphinx tags have to use Python identifier syntax, Sage uses the format `feature_`, followed by the feature name where dots are replaced by underscores. Hence, conditionalizing on the features of the previous examples would look as follows:

```
.. ONLY:: feature_graphviz
```

and:

```
.. ONLY:: feature_sage_symbolic
```

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/sage_docbuild/__init__.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://www.sphinx-doc.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)
```

1.7 More on Coding for Sage

1.7.1 Coding in Python for Sage

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

Python language standard

Sage library code needs to be compatible with all versions of Python that Sage supports. The information regarding the supported versions can be found in the files `build/pkgs/python3/spkg-configure.m4` and `src/setup.cfg.m4`.

Python 3.9 is the oldest supported version. Hence, all language and library features that are available in Python 3.9 can be used; but features introduced in Python 3.10 cannot be used. If a feature is deprecated in a newer supported version, it must be ensured that deprecation warnings issued by Python do not lead to failures in doctests.

Some key language and library features have been backported to older Python versions using one of two mechanisms:

- `from __future__ import annotations` (see Python reference for __future__) modernizes type annotations according to PEP 563 (Postponed evaluation of annotations). All Sage library code that uses type annotations should include this `__future__` import and follow PEP 563.
- Backport packages
  - importlib_metadata (to be used in place of `importlib.metadata`),
  - importlib_resources (to be used in place of `importlib.resources`),
  - typing_extensions (to be used in place of `typing`).

  The Sage library declares these packages as dependencies and ensures that versions that provide features of Python 3.11 are available.

Meta github issue #29756 keeps track of newer Python features and serves as a starting point for discussions on how to make use of them in the Sage library.

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.structure.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. Note that the `.. skip` line should not be included in your code; it is here to prevent doctests from running on this fake example.

```
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/symbolic/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/generic_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/src/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)
```

(continues on next page)

(continued from previous page)

```
<... 'int'>
sage: b = 393939
sage: type(b)
<class '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 mix together the hash values of the components of the object that also play a part in comparison of objects by packing them into a tuple and hashing the tuple.
>
> If a class does not define an `__eq__()` method it should not define a `__hash__()` operation either; if it defines `__eq__()` but not `__hash__()`, its instances will not be usable as items in hashable collections. If a class defines mutable objects and implements an `__eq__()` method, it should not implement `__hash__()`, since the implementation of hashable collections requires that a key's hash value is immutable (if the object's hash value changes, it will be in the wrong hash bucket).

See https://docs.python.org/3/reference/datamodel.html#object.__hash__ for more information on the subject.

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.

A method or a function accepts input described in the `INPUT` block of *the docstring*. If the input cannot be handled by the code, then it may raise an exception. The following aims to guide you in choosing from the most relevant exceptions to Sage. Raise

- `TypeError`: if the input belongs to a class of objects that is not supported by the method. For example, a method works only with monic polynomials over a finite field, but a polynomial over rationals was given.
- `ValueError`: if the input has a value not supported by the method. For example, the above method was given a non-monic polynomial.
- `ArithmeticError`: if the method performs an arithmetic operation (sum, product, quotient, and the like) but the input is not appropriate.
- `ZeroDivisionError`: if the method performs division but the input is zero. Note that for non-invertible input values, `ArithmeticError` is more appropriate. As derived from `ArithmeticError`, `ZeroDivisionError` can be caught as `ArithmeticError`.
- `NotImplementedError`: if the input is for a feature not yet implemented by the method. Note that this exception is derived from `RuntimeError`.

If no specific error seems to apply for your situation, `RuntimeError` can be used. In all cases, the string associated with the exception should describe the details of what went wrong.

Integer return values

Many functions and methods in Sage return integer values. Those should usually be returned as Sage integers of class `Integer` rather than as Python integers of class `int`, as users may want to explore the resulting integers' number-theoretic properties such as prime factorization. Exceptions should be made when there are good reasons such as performance or compatibility with Python code, for instance in methods such as `__hash__`, `__len__`, and `__int__`.

To return a Python integer `i` as a Sage integer, use:

```
from sage.rings.integer import Integer
return Integer(i)
```

To return a Sage integer `i` as a Python ineger, use:

```
return int(i)
```

Importing

We mention two issues with importing: circular imports and importing large third-party modules. See also *Dependencies and distribution packages* for a discussion of imports from the viewpoint of modularization.

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.

If your module needs to make some precomputed data available at the top level, you can reduce its load time (and thus startup time, unless your module is imported using `sage.misc.lazy_import`) by using the decorator `sage.misc.cachefunc.cached_function()` instead. For example, replace

```
big_data = initialize_big_data()  # bad: runs at module load time
```

by

```
from sage.misc.cachefunc import cached_function

@cached_function                  # good: runs on first use
def big_data():
    return initialize_big_data()
```

Static typing

Python libraries are increasingly annotated with static typing information; see the Python reference on typing.

For typechecking the Sage library, the project uses *pyright*; it automatically runs in the GitHub Actions CI and can also be run locally.

As of Sage 10.2, the Sage library only contains a minimal set of such type annotations. Pull requests that add more annotations are generally welcome.

The Sage library makes very extensive use of Cython (see chapter *Coding in Cython*). Although Cython source code often declares static types for the purpose of compilation to efficient machine code, this typing information is unfortunately not visible to static checkers such as Pyright. It is necessary to create type stub files (".pyi") that provide this information. Although various tools for writing and maintaining type stub files are available, creating stub files for Cython files involves manual work. There is hope that better tools become available soon, see for example cython/cython #5744. Contributing to the development and testing of such tools likely will have a greater impact than writing the typestub files manually.

For Cython modules of the Sage library, these type stub files would be placed next to the `.pyx` and `.pxd` files.

When importing from other Python libraries that do not provide sufficient typing information, it is possible to augment the library's typing information for the purposes of typechecking the Sage library:

- Create typestub files and place them in the directory `SAGE_ROOT/src/typings`. For example, the distribution **pplpy** provides the top-level package `ppl`, which publishes no typing information. We can create a typestub file `SAGE_ROOT/src/typings/ppl.pyi` or `SAGE_ROOT/src/typings/ppl/__init__.pyi`.
- When these typestub files are working well, it is preferable from the viewpoint of the Sage project that they are "upstreamed", i.e., contributed to the project that maintains the library. If a new version of the upstream library becomes available that provides the necessary typing information, we can update the package in the Sage distribution and remove the typestub files again from `SAGE_ROOT/src/typings`.
- As a fallback, when neither adding typing annotations to source files nor adding typestub files is welcomed by the upstream project, it is possible to contribute typestubs files instead to the typeshed community project.

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 GitHub PR 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 `*` 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 a global namespace:** this is when you want to remove a name from a global namespace (say, `sage.all` or some other `all.py` file) but you want to keep the functionality available with an explicit import. This case is similar as the previous one: use a lazy import with deprecation. One detail: in this case, you don't want the name `lazy_import` to be visible in the global namespace, so we add a leading underscore:

```
from sage.misc.lazy_import import lazy_import as _lazy_import
_lazy_import('sage.some.package', 'some_function', deprecation=666)
```

- **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 `experimental`. Here is an example:

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

- **Experimental class:** use the decorator `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_warning()` 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.

1.7.2 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,
    1. It is saved to a file.
    2. Cython is run on it with all the standard Sage libraries automatically linked if necessary.
    3. The resulting shared library file (`.so` / `.dll` / `.dylib`) is then loaded into your running instance of Sage.
    4. 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.
    5. 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. Then run `sage -b` to rebuild Sage.

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 that the code can be interrupted with `CTRL-C`.

Sage uses the cysignals package for this, see the cysignals documentation for more information.

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
        (<cyfunction make_integer at ...>, ('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),)
```

1.7.3 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/cypari2/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_form(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_form(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 :pari:`matfrobenius`

    EXAMPLES::

        sage: A = MatrixSpace(ZZ, 3)(range(9))
        sage: A.frobenius_form(0)
        [ 0  0  0]
        [ 1  0 18]
        [ 0  1 12]
        sage: A.frobenius_form(1)
        [x^3 - 12*x^2 - 18*x]
        sage: A.frobenius_form(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()))
```

(continues on next page)

(continued from previous page)

```
        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]
```

(continues on next page)

(continued from previous page)

```
        [ 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>
```

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

```
[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')
    polynomial ring, over a field, global ordering
    //   coefficients: ZZ/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);"))
```

(continues on next page)

(continued from previous page)

```
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) # random
[1]:
   [1]:
      0
   [2]:
      1
   [3]:
      0
...
```

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)
    # 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]
        pt.append(F(eval(L2)))
        # end block1 to compute pt
        L1 = L1[idx+8:] # repeat block 2 more times
        # start block2 for compute pt
        idx = L1.index("     ")
        idx2 = L1[idx:].index("\n")
        L2 = L1[idx:idx+idx2]
        pt.append(F(eval(L2)))
```

(continues on next page)

(continued from previous page)

```
        # 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)))
        # 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, script_subdirectory="", logfile=None,
             server=None, server_tmpdir=None):
    Expect.__init__(self,
                    name = 'octave',
                    prompt = '>',
                    command = "octave --no-line-editing --silent",
                    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:

    A 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
        [-0.333333, 0.666667, 0]

    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.

1.8 Packaging

1.8.1 Packaging Third-Party Code for Sage

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:

```
[alice@localhost sage]$ 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. For a few packages, `configure` checks whether they are available from the system, in which case the build of those packages is skipped. Standard packages 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.

Package source types

Orthogonal to the division by package types, a package has exactly one of the following source types:

1. A `normal` package:
   - comes from the tarball named in the required file `checksums.ini` and hosted on the Sage mirrors;
   - its version number is defined by the required file `package-version.txt`;
   - Sage installs the package using build and install scripts (see *Build and install scripts of normal packages*);
   - Sage records the version number of the package installed using a file in `$SAGE_LOCAL/var/lib/sage/installed/` and will rerun the installation if `package-version.txt` changes.

2. A `wheel` package:
   - comes from the wheel file named in the required file `checksums.ini` and hosted on the Sage mirrors;
   - per policy, only platform-independent wheels are allowed, i.e., `*-none-any.whl` files;
   - its version number is defined by the required file `package-version.txt`;
   - no build and install scripts are needed (with one exception: the package pip: Tool for installing and managing Python packages installs itself from its wheel using a custom install script);
   - Sage records the version number of the package installed using a file in `$SAGE_LOCAL/var/lib/sage/installed/` and will rerun the installation if `package-version.txt` changes.
3. A `pip` package:
   - is obtained directly from https://pypi.org/;
   - the version to be installed is determined using the required file `requirements.txt` – in its simplest form, this file just contains the name of the package (more details at https://pip.pypa.io/en/stable/user_guide/#requirements-files);
   - Sage installs the package using the `pip` package manager;
   - Sage delegates the recording of installed package version numbers to it;
   - by policy, no `standard` package is allowed to be a `pip` package.
4. A `script` package:
   - is not associated with a tarball;
   - the file `package-version.txt` is optional;
   - installing the package runs the installation script `spkg-install` or `spkg-install.in` (see *Build and install scripts of normal packages*);
   - Sage records the version number of the package installed using a file in `$SAGE_LOCAL/var/lib/sage/installed/` and will rerun the installation if `package-version.txt` changes.
5. A `dummy` package:
   - is only used for recording the names of equivalent system packages;
   - there is no `spkg-install` script, and attempts to install the package using Sage will give an error message.

To summarize: the package source type is determined as follows: if there is a file `requirements.txt`, it is a `pip` package. If not, then if there is a `checksums.ini` file, it is `normal` or `wheel`. Otherwise, if it has an `spkg-install` or `spkg-install.in` script, it is a `script` package, and if it does not, then it is a `dummy` package.

Directory structure

Third-party packages in Sage consist 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.

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.in
|-- SPKG.rst
`-- type
```

The following are some additional files which can be added:

```
SAGE_ROOT/build/pkgs/foo
|-- distros
|   |-- platform1.txt
|   `-- platform2.txt
|-- has_nonfree_dependencies
|-- huge
|-- patches
|   |-- bar.patch
|   `-- baz.patch
|-- spkg-check.in
|-- spkg-configure.m4
|-- spkg-src
`-- trees.txt
```

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.

Build and install scripts of normal packages

The `spkg-build.in` and `spkg-install.in` files are templates for `bash` scripts `spkg-build` and `spkg-install`, which build and/or install the package.

The `*.in` script templates should *not* be prefixed with a shebang line (`#!...`) and should not have the executable bit set in their permissions. These are added automatically when generating the scripts, along with some additional boilerplate, when the package is installed.

The `spkg-build.in` and `spkg-install.in` files in the Sage source tree need only focus on the specific steps for building and installing that package. If no `spkg-build.in` exists, then the `spkg-install.in` is responsible for both steps, though separating them is encouraged where possible.

It is also possible to include similar script templatess named `spkg-preinst.in` or `spkg-postinst.in` to run additional steps before or after the package has been installed into `$SAGE_LOCAL`. It is encouraged to put steps which modify already installed files in a separate `spkg-postinst.in` script template rather than combining them with `spkg-install.in`. This is because since github issue #24106, `spkg-install` does not necessarily install packages directly to `$SAGE_LOCAL`. However, by the time `spkg-postinst` is run, the installation to `$SAGE_LOCAL` is complete.

In the best case, the upstream project can simply be installed by the usual configure / make / make install steps. In that case, the `spkg-build.in` script template would simply consist of:

```
cd src
sdh_configure
sdh_make
```

See *Helper functions* for more on the helper functions `sdh_configure`, `sdh_make`, etc.

The `spkg-install.in` script template would consist of:

```
cd src
sdh_make_install
```

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

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

```
if [ "$SAGE_SPKG_INSTALL_DOCS" = yes ] ; then
    sdh_make doc
    sdh_install doc/ "$SAGE_SHARE"/doc/PACKAGE_NAME
fi
```

At build time `CFLAGS`, `CXXFLAGS`, `FCFLAGS`, and `F77FLAGS` are usually set to `-g -O2 -march=native` (according to debugging options and whether building fat binaries).

Slightly modified versions are available:

```
# No ``-march=native``.
export CFLAGS=$CFLAGS_NON_NATIVE

# ``-O3`` instead of ``-O2``.
export CFLAGS=$CFLAGS_O3

# Use flags as set by the user, possibly empty.
export CFLAGS=$ORIGINAL_CFLAGS
```

Likewise for `CXXFLAGS`, `FCFLAGS`, and `F77FLAGS`.

Note: Prior to Sage 9.1, the script templates were called `spkg-build`, `spkg-install`, etc., without the extension `.in`.

Prior to Sage 8.1 the shebang line was included, and the scripts were marked executable. However, this is no longer the case as of github issue #23179. Now the scripts in the source tree are deliberately written not to be directly executed, and are only made into executable scripts when they are copied to the package's build directory.

Build/install scripts may still be written in Python, but the Python code should go in a separate file (e.g. `spkg-install.py`), and can then be executed from the real `spkg-install.in` like:

```
exec sage-bootstrap-python spkg-install.py
```

or

```
exec python3 spkg-install.py
```

In more detail: `sage-bootstrap-python` runs a version of Python pre-installed on the machine, which is a build prerequisite of Sage. Note that `sage-bootstrap-python` accepts a wide range of Python versions, Python >= 2.6

and >= 3.4, see `SAGE_ROOT/build/tox.ini` for details. You should only use `sage-bootstrap-python` for installation tasks that must be able to run before Sage has made `python3` available. It must not be used for running `pip` or `setup.py` for any package.

`python3` runs the version of Python managed by Sage (either its own installation of Python 3 from an SPKG or a venv over a system python3. You should use this if you are installing a Python package to make sure that the libraries are installed in the right place.

By the way, there is also a script `sage-python`. This should be used at runtime, for example in scripts in `SAGE_LOCAL/bin` which expect Sage's Python to already be built.

Many packages currently do not separate the build and install steps and only provide a `spkg-install.in` file that does both. The separation is useful in particular for root-owned install hierarchies, where something like `sudo` must be used to install files. For this purpose Sage uses an environment variable `$SAGE_SUDO`, the value of which may be provided by the developer at build time, which should to the appropriate system-specific `sudo`-like command (if any). The following rules are then observed:

- If `spkg-build.in` exists, the generated script `spkg-build` is first called, followed by `$SAGE_SUDO spkg-install`.
- Otherwise, only `spkg-install` is called (without `$SAGE_SUDO`). Such packages should prefix all commands in `spkg-install.in` that write into the installation hierarchy with `$SAGE_SUDO`.

Install scripts of script packages

For script packages, it is also possible to use an install script named `spkg-install`. It needs to be an executable shell script; it is not subject to the templating described in the previous section and will be executed directly from the build directory.

Helper functions

In the `spkg-build`, `spkg-install`, and `spkg-check` scripts, the following functions are available. They are defined in the file `$SAGE_ROOT/build/bin/sage-dist-helpers`, if you want to look at the source code. They should be used to make sure that appropriate variables are set and to avoid code duplication. These function names begin with `sdh_`, which stands for "Sage-distribution helper".

- `sdh_die MESSAGE`: Exit the build script with the error code of the last command if it was non-zero, or with 1 otherwise, and print an error message. This is typically used like:

```
command || sdh_die "Command failed"
```

  This function can also (if not given any arguments) read the error message from stdin. In particular this is useful in conjunction with a heredoc to write multi-line error messages:

```
command || sdh_die << _EOF_
Command failed.
Reason given.
_EOF_
```

  Note: The other helper functions call `sdh_die`, so do not use (for example) `sdh_make || sdh_die`: the part of this after `||` will never be reached.

- `sdh_check_vars [VARIABLE ...]`: Check that one or more variables are defined and non-empty, and exit with an error if any are undefined or empty. Variable names should be given without the '$' to prevent unwanted expansion.
- `sdh_configure [...]`: Runs `./configure` with arguments `--prefix="$SAGE_LOCAL"`, `--libdir="$SAGE_LOCAL/lib"`, `--disable-static`, `--disable-maintainer-mode`, and `--disable-dependency-tracking`. Additional arguments to `./configure` may be given as arguments.
- `sdh_make [...]`: Runs `$MAKE` with the default target. Additional arguments to `$MAKE` may be given as arguments.
- `sdh_make_install [...]`: Runs `$MAKE install` with DESTDIR correctly set to a temporary install directory, for staged installations. Additional arguments to `$MAKE` may be given as arguments. If `$SAGE_DESTDIR` is not set then the command is run with `$SAGE_SUDO`, if set.
- `sdh_setup_bdist_wheel [...]`: Runs `setup.py bdist_wheel` with the given arguments, as well as additional default arguments used for installing packages into Sage.
- `sdh_pip_install [...]`: The equivalent of running `pip install` with the given arguments, as well as additional default arguments used for installing packages into Sage with pip. The last argument must be `.` to indicate installation from the current directory.

  `sdh_pip_install` actually does the installation via `pip wheel`, creating a wheel file in `dist/`, followed by `sdh_store_and_pip_install_wheel` (see below).
- `sdh_pip_editable_install [...]`: The equivalent of running `pip install -e` with the given arguments, as well as additional default arguments used for installing packages into Sage with pip. The last argument must be `.` to indicate installation from the current directory. See pip documentation for more details concerning editable installs.
- `sdh_pip_uninstall [...]`: Runs `pip uninstall` with the given arguments. If unsuccessful, it displays a warning.
- `sdh_store_and_pip_install_wheel .`: The current directory, indicated by the required argument `.`, must have a subdirectory `dist` containing a unique wheel file (`*.whl`).

  This command (1) moves this wheel file to the directory `$SAGE_SPKG_WHEELS` (`$SAGE_LOCAL/var/lib/sage/wheels`) and then (2) installs the wheel in `$SAGE_LOCAL`.

  Both of these steps, instead of writing directly into `$SAGE_LOCAL`, use the staging directory `$SAGE_DESTDIR` if set; otherwise, they use `$SAGE_SUDO` (if set).
- `sdh_install [-T] SRC [SRC...] DEST`: Copies one or more files or directories given as `SRC` (recursively in the case of directories) into the destination directory `DEST`, while ensuring that `DEST` and all its parent directories exist. `DEST` should be a path under `$SAGE_LOCAL`, generally. For `DESTDIR` installs, the `$SAGE_DESTDIR` path is automatically prepended to the destination.

  The `-T` option treats `DEST` as a normal file instead (e.g. for copying a file to a different filename). All directory components are still created in this case.

The following is automatically added to each install script, so you should not need to add it yourself.

- `sdh_guard`: Wrapper for `sdh_check_vars` that checks some common variables without which many/most packages won't build correctly (`SAGE_ROOT`, `SAGE_LOCAL`, `SAGE_SHARE`). This is important to prevent installation to unintended locations.

The following are also available, but rarely used.

- `sdh_cmake [...]`: Runs `cmake` in the current directory with the given arguments, as well as additional arguments passed to cmake (assuming packages are using the GNUInstallDirs module) so that `CMAKE_INSTALL_PREFIX` and `CMAKE_INSTALL_LIBDIR` are set correctly.

- `sdh_preload_lib EXECUTABLE SONAME`: (Linux only – no-op on other platforms.) Check shared libraries loaded by `EXECUTABLE` (may be a program or another library) for a library starting with `SONAME`, and if found appends `SONAME` to the `LD_PRELOAD` environment variable. See github issue #24885.

Allowing for the use of system packages

For a number of Sage packages, an already installed system version can be used instead, and Sage's top-level `./configure` script determines when this is possible. To enable this, a package needs to have a script called `spkg-configure.m4`, which can, for example, determines whether the installed software is recent enough (and sometimes not too recent) to be usable by Sage. This script is processed by the GNU M4 macro processor.

Also, if the software for a Sage package is provided by a system package, the `./configure` script can provide that information. To do this, there must be a directory `build/pkgs/PACKAGE/distros` containing files with names like

```
arch.txt
conda.txt
debian.txt
fedora.txt
homebrew.txt
...
```

corresponding to different packaging systems. Each system package should appear on a separate line. If the shell-style variable reference `${PYTHON_MINOR}` appears, it is replaced by the minor version of Python, e.g., 12 for Python 3.12.x. Everything on a line after a `#` character is ignored, so comments can be included in the files.

For example, if `./configure` detects that the Homebrew packaging system is in use, and if the current package can be provided by a Homebrew package called "foo", then the file `build/pkgs/PACKAGE/distros/homebrew.txt` should contain the single line "foo". If `foo` is currently uninstalled, then `./configure` will print a message suggesting that the user should run `brew install foo`. See *Using Sage's database of equivalent distribution packages* for more on this.

Important: All new standard packages should, when possible, include a `spkg-configure.m4` script and a populated `distros` directory. There are many examples in `build/pkgs`, including `build/pkgs/python3` and `build/pkgs/suitesparse`, to name a few.

Note that this may not be possible (as of this writing) for some packages, for example packages installed via pip for use while running Sage, like `matplotlib` or `scipy`. If a package is installed via pip for use in a separate process, like `tox`, then this should be possible.

Self-tests

The `spkg-check.in` file is an optional, but highly recommended, script template to run self-tests of the package. The format for the `spkg-check` is the same as `spkg-build` and `spkg-install`. 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 test suite that can be run with the standard `make check` target. In that case, the `spkg-check.in` script template would simply contain:

```
cd src
$MAKE check
```

Python-based packages

Python-based packages should declare `$(PYTHON)` as a dependency, and most Python-based packages will also have `$(PYTHON_TOOLCHAIN)` as an order-only dependency, which will ensure that fundamental packages such as `pip` and `setuptools` are available at the time of building the package.

The best way to install a Python-based package is to use `pip`, in which case the `spkg-install.in` script template might just consist of

```
cd src && sdh_pip_install .
```

Where `sdh_pip_install` is a function provided by `sage-dist-helpers` that points to the correct `pip` for the Python used by Sage, and includes some default flags needed for correct installation into Sage.

If `pip` will not work for a package but a command like `python3 setup.py install` will, you may use `sdh_setup_bdist_wheel`, followed by `sdh_store_and_pip_install_wheel ..`

For `spkg-check.in` script templates, use `python3` rather than just `python`. The paths are set by the Sage build system so that this runs the correct version of Python. For example, the `scipy` `spkg-check.in` file contains the line

```
exec python3 spkg-check.py
```

All normal Python packages and all wheel packages must have a file `install-requires.txt`. If a Python package is available on PyPI, this file must contain the name of the package as it is known to PyPI. Optionally, `install-requires.txt` can encode version constraints (such as lower and upper bounds). The constraints are in the format of the `install_requires` key of setup.cfg or setup.py.

It is strongly recommended to include comments (starting with `#`) in the file that explain why a particular lower or upper bound is warranted or why we wish to include or reject certain versions.

For example:

```
$ cat build/pkgs/sphinx/package-version.txt
3.1.2.p0
$ cat build/pkgs/sphinx/install-requires.txt
# gentoo uses 3.2.1
sphinx >=3, <3.3
```

The comments may include links to GitHub Issues/PRs, as in the following example:

```
$ cat build/pkgs/packaging/install-requires.txt
packaging >=18.0
# Issue #30975: packaging 20.5 is known to work
# but we have to silence "DeprecationWarning: Creating a LegacyVersion"
```

The currently encoded version constraints are merely a starting point. Developers and downstream packagers are invited to refine the version constraints based on their experience and tests. When a package update is made in order to pick up a critical bug fix from a newer version, then the lower bound should be adjusted. Setting upper bounds to guard against incompatible future changes is a complex topic; see github issue #33520.

The SPKG.rst file

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

```
PACKAGE_NAME: One line description
==================================

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.  Usually just an URL.

Dependencies
------------

Only put special dependencies here that are not captured by the
``dependencies`` file. Otherwise omit this section.

Special Update/Build Instructions
---------------------------------

If the tarball was modified by hand and not via an ``spkg-src``
script, describe what was changed. Otherwise omit this section.
```

with `PACKAGE_NAME` replaced by the SPKG name (= the directory name in `build/pkgs`).

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`:

```
pari ntl flint

----------
All lines of this file are ignored except the first.
```

For Python packages, common dependencies include `pip`, `setuptools`, and `future`. If your package depends on any of these, use `$(PYTHON_TOOLCHAIN)` instead. For example, here is the `dependencies` file for `configparser`:

```
$(PYTHON) | $(PYTHON_TOOLCHAIN)
```

(See below for the meaning of the `|`.)

If there are no dependencies, you can use

```
# no dependencies

----------
All lines of this file are ignored except the first.
```

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 pip simply because the `spkg-install` file uses pip).

  Alternatively, you can put the order-only dependencies in a separate file `dependencies_order_only`.

- 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.

Some packages are only needed for self-tests of a package (`spkg-check`). These dependencies should be declared in a separate file `dependencies_check`.

Some dependencies are optional in the sense that they are only a dependency if they are configured to be installed. These dependencies should be declared in a separate file `dependencies_optional`.

In order to check that the dependencies of your package are likely correct, the following command should work without errors:

```
[alice@localhost sage]$ 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.

Package tags

You can mark a package as "huge" by placing an empty file named `huge` in the package directory. For example, the package `polytopes_db_4d` is a large database whose compressed tarball has a size of 9 GB.

For some other packages, we have placed an empty file named `has_nonfree_dependencies` in the package directory. This is to indicate that Sage with this package installed cannot be redistributed, and also that the package can only be installed after installing some other, non-free package.

We use these tags in our continuous integration scripts to filter out packages that we cannot or should not test automatically.

Where packages are installed

The Sage distribution has the notion of several installation trees.

- `$SAGE_VENV` is the default installation tree for all Python packages, i.e., normal packages with an `install-requires.txt`, wheel packages, and pip packages with a `requirements.txt`.
- `$SAGE_LOCAL` is the default installation tree for all non-Python packages.
- `$SAGE_DOCS` (only set at build time) is an installation tree for the HTML and PDF documentation.

By placing a file `trees.txt` in the package directory, the installation tree can be overridden. For example, `build/pkgs/python3/trees.txt` contains the word `SAGE_VENV`, and `build/pkgs/sagemath_doc_html/trees.txt` contains the word `SAGE_DOCS`.

Patching sources

Actual changes to the source code must be via patches, which should be placed in the `patches/` directory, and must have the `.patch` extension. 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), and must have only one "prefix" level in the paths (that is, only one path level above the root of the upstream sources being patched). 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 directly under the `patches/` directly are applied automatically before running the `spkg-install` script (so long as they have the `.patch` extension). If you need to apply patches conditionally (such as only on a specifically platform), you can place those patches in a subdirectory of `patches/` and apply them manually using the `sage-apply-patches` script. For example, considering the layout:

```
SAGE_ROOT/build/pkgs/foo
|-- patches
|   |-- solaris
|   |   |-- solaris.patch
|   |-- bar.patch
|   `-- baz.patch
```

The patches `bar.patch` and `baz.patch` are applied to the unpacked upstream sources in `src/` before running `spkg-install`. To conditionally apply the patch for Solaris the `spkg-install` should contain a section like this:

```
if [ $UNAME == "SunOS" ]; then
    sage-apply-patches -d solaris
fi
```

where the `-d` flag applies all patches in the `solaris/` subdirectory of the main `patches/` directory.

When to patch, when to repackage, when to autoconfiscate

- Use unpatched original upstream tarball when possible.

  Sometimes it may seem as if you need to patch a (hand-written) `Makefile` because it "hard-codes" some paths or compiler flags:

  ```
  --- a/Makefile
  +++ b/Makefile
  @@ -77,7 +77,7 @@
   # This is a Makefile.
   # Handwritten.

  -DESTDIR = /usr/local
  +DESTDIR = $(SAGE_ROOT)/local
   BINDIR   = $(DESTDIR)/bin
   INCDIR   = $(DESTDIR)/include
   LIBDIR   = $(DESTDIR)/lib
  ```

  Don't use patching for that. Makefile variables can be overridden from the command-line. Just use the following in `spkg-install`:

  ```
  $(MAKE) DESTDIR="$SAGE_ROOT/local"
  ```

- Check if Debian or another distribution already provides patches for upstream. Use them, don't reinvent the wheel.
- If the upstream Makefile does not build shared libraries, don't bother trying to patch it.

  Autoconfiscate the package instead and use the standard facilities of Automake and Libtool. This ensures that the shared library build is portable between Linux and macOS.
- If you have to make changes to `configure.ac` or other source files of the autotools build system (or if you are autoconfiscating the package), then you can't use patching; make a *modified tarball* instead.
- If the patch would be huge, don't use patching. Make a *modified tarball* instead.
- Otherwise, *maintain a set of patches*.

How to maintain a set of patches

We recommend the following workflow for maintaining a set of patches.

- Fork the package and put it on a public git repository.

  If upstream has a public version control repository, import it from there. If upstream does not have a public version control repository, import the current sources from the upstream tarball. Let's call the branch `upstream`.
- Create a branch for the changes necessary for Sage, let's call it `sage_package_VERSION`, where `version` is the upstream version number.
- Make the changes and commit them to the branch.

- Generate the patches against the `upstream` branch:

```
rm -Rf SAGE_ROOT/build/pkgs/PACKAGE/patches
mkdir SAGE_ROOT/build/pkgs/PACKAGE/patches
git format-patch -o SAGE_ROOT/build/pkgs/PACKAGE/patches/ upstream
```

- Optionally, create an `spkg-src` file in the Sage package's directory that regenerates the patch directory using the above commands.
- When a new upstream version becomes available, merge (or import) it into `upstream`, then create a new branch and rebase it on top of the updated upstream:

```
git checkout sage_package_OLDVERSION
git checkout -b sage_package_NEWVERSION
git rebase upstream
```

  Then regenerate the patches.

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 contains 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 and tarball names

The `checksums.ini` file contains the filename pattern of the upstream tarball (without the actual version) and its checksums. So if upstream is `$SAGE_ROOT/upstream/FoO-1.3.tar.gz`, create a new file `$SAGE_ROOT/build/pkgs/foo/checksums.ini` containing only:

```
tarball=FoO-VERSION.tar.gz
```

Sage internally replaces the `VERSION` substring with the content of `package-version.txt`.

Upstream URLs

In addition to these fields in `checksums.ini`, the optional field `upstream_url` holds an URL to the upstream package archive.

The Release Manager uses the information in `upstream_url` to download the upstream package archive and to make it available on the Sage mirrors when a new release is prepared. On GitHub PRs upgrading a package, the PR description should no longer contain the upstream URL to avoid duplication of information.

Note that, like the `tarball` field, the `upstream_url` is a template; the substring `VERSION` is substituted with the actual version. It can also be written as `${VERSION}`, and it is possible to refer to the dot-separated components of a version by `VERSION_MAJOR`, `VERSION_MINOR`, and `VERSION_MICRO`.

For Python packages available from PyPI, you should use an `upstream_url` from `pypi.io`, which follows the format

```
upstream_url=https://pypi.io/packages/source/m/matplotlib/matplotlib-VERSION.tar.gz
```

Developers who wish to test a package update from a PR branch before the archive is available on a Sage mirror. Sage falls back to downloading package tarballs from the `upstream_url` after trying all Sage mirrors. (This can be disabled by using `./configure --disable-download-from-upstream-url`.) To speed up this process, you can trim `upstream/mirror_list` to fewer mirrors.

Utility script to create and maintain packages

The command `sage --package` offers a range of functionality for creating and maintaining packages of the Sage distribution.

Creating packages

Assuming that you have downloaded `$SAGE_ROOT/upstream/FoO-1.3.tar.gz`, you can use:

```
[alice@localhost sage]$ sage --package create foo                    \
                                        --version 1.3                \
                                        --tarball FoO-VERSION.tar.gz \
                                        --type experimental
```

to create `$SAGE_ROOT/build/pkgs/foo/package-version.txt`, `checksums.ini`, and `type` in one step.

You can skip the manual downloading of the upstream tarball by using the additional argument `--upstream-url`. This command will also set the `upstream_url` field in `checksums.ini` described above.

For Python packages available from PyPI, you can use:

```
[alice@localhost sage]$ sage --package create scikit_spatial --pypi   \
                                        --type optional
```

This automatically downloads the most recent version from PyPI and also obtains most of the necessary information by querying PyPI.

The `dependencies` file may need editing (watch out for warnings regarding `--no-deps` that Sage issues during installation of the package!). Also you may want to set lower and upper bounds for acceptable package versions in the file `install-requires.txt`.

By default, when the package is available as a platform-independent wheel, the `sage --package` creates a wheel package. To create a normal package instead (for example, when the package requires patching), you can use:

```
[alice@localhost sage]$ sage --package create scikit_spatial --pypi   \
                                        --source normal               \
                                        --type optional
```

To create a pip package rather than a normal or wheel package, you can use:

```
[alice@localhost sage]$ sage --package create scikit_spatial --pypi   \
                                        --source pip                  \
                                        --type optional
```

When the package already exists, `sage --package create` overwrites it.

Updating packages to a new version

A package that has the `upstream_url` information can be updated by simply typing:

```
[alice@localhost sage]$ sage --package update numpy 3.14.59
```

which will automatically download the archive and update the information in `build/pkgs/numpy/`.

For Python packages available from PyPI, there is another shortcut:

```
[alice@localhost sage]$ sage --package update-latest matplotlib
Updating matplotlib: 3.3.0 -> 3.3.1
Downloading tarball to ...matplotlib-3.3.1.tar.bz2
[............................................................]
```

If you pass the switch `--commit`, the script will run `git commit` for you.

If you prefer to make update a package `foo` by making manual changes to the files in `build/pkgs/foo`, you will need to run:

```
[alice@localhost sage]$ sage --package fix-checksum foo
```

which will modify the `checksums.ini` file with the correct checksums.

Obtaining package metrics

The command `sage --package metrics` computes machine-readable aggregated metrics for all packages in the Sage distribution or a given list of packages:

```
[alice@localhost sage]$ sage --package metrics
has_file_distros_arch_txt=181
has_file_distros_conda_txt=289
has_file_distros_debian_txt=172
has_file_distros_fedora_txt=183
has_file_distros_gentoo_txt=211
has_file_distros_homebrew_txt=95
has_file_distros_macports_txt=173
has_file_distros_nix_txt=72
has_file_distros_opensuse_txt=206
has_file_distros_slackware_txt=32
has_file_distros_void_txt=221
has_file_patches=63
has_file_spkg_check=106
has_file_spkg_configure_m4=262
has_file_spkg_install=322
has_tarball_upstream_url=291
line_count_file_patches=31904
line_count_file_spkg_check=585
line_count_file_spkg_configure_m4=3337
line_count_file_spkg_install=4342
packages=442
type_base=1
type_experimental=18
type_optional=151
type_standard=272
```

Developers can use these metrics to monitor the complexity and quality of the Sage distribution. Here are some examples:

- `has_file_patches` indicates how many packages have non-empty `patches/` directories, and `line_count_file_patches` gives the total number of lines in the patch files.

  Ideally, we would not have to carry patches for a package. For example, updating patches when a new upstream version is released can be a maintenance burden.

  Developers can help by working with the upstream maintainers of the package to prepare a new version that requires fewer or smaller patches, or none at all.

- `line_count_spkg_install` gives the total number of lines in `spkg-install` or `spkg-install.in` files; see *Build and install scripts of normal packages*.

  When we carry complex `spkg-install.in` scripts for normal packages, it may indicate that the upstream package's build and installation scripts should be improved.

  Developers can help by working with the upstream maintainers of the package to prepare an improved version.

- `has_file_spkg_check` indicates how many packages have an `spkg-check` or `spkg-check.in` file; see *Self-tests*.

- `has_file_spkg_configure_m4` indicates how many packages are prepared to check for an equivalent system package, and `has_file_distros_arch_txt`, `has_file_distros_conda_txt` etc. count how many packages provide the corresponding system package information.

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 can install the package using:

```
[alice@localhost sage]$ sage -i package_name
```

or:

```
[alice@localhost sage]$ sage -f package_name
```

to force a reinstallation. If your package contains a `spkg-check` script (see *Self-tests*) it can be run with:

```
[alice@localhost sage]$ sage -i -c package_name
```

or:

```
[alice@localhost sage]$ sage -f -c package_name
```

If all went fine, open a PR 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 GitHub PR added with labels `c: packages: experimental` or `c: packages: optional`. The associated code requirements are described in the following sections.

After the PR 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 GitHub PR is opened with the label `c: 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 PR 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.rst` or `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.

If an upstream tarball of a package cannot be redistributed for license reasons, rename it to include the string `do-not-distribute`. This will keep the release management scripts from uploading it to the Sage mirrors.

Sometimes an upstream tarball contains some distributable parts using a free software license and some non-free parts. In this case, it can be a good solution to make a custom tarball consisting of only the free parts; see *Modified tarballs* and the `giac` package as an example.

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 (Linux, macOS); see *Testing on Multiple Platforms*. It must be installed either from source as a normal package, or as a Python (platform-independent) wheel package, see *Package source types*.
- **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 Repository on GitHub*.

1.8.2 Packaging the Sage Library for Distribution

Modules, packages, distribution packages

The Sage library consists of a large number of Python modules, organized into a hierarchical set of packages that fill the namespace `sage`. All source files are located in a subdirectory of the directory `SAGE_ROOT/src/sage/`.

For example,

- the file `SAGE_ROOT/src/sage/coding/code_bounds.py` provides the module `sage.coding.code_bounds`;
- the directory containing this file, `SAGE_ROOT/src/sage/coding/`, thus provides the package `sage.coding`.

There is another notion of "package" in Python, the **distribution package** (also known as a "distribution" or a "pip-installable package"). Currently, the entire Sage library is provided by a single distribution, sagemath-standard, which is generated from the directory `SAGE_ROOT/pkgs/sagemath-standard`.

Note that the distribution name is not required to be a Python identifier. In fact, using dashes (`-`) is preferred to underscores in distribution names; **setuptools** and other parts of Python's packaging infrastructure normalize underscores to dashes. (Using dots in distribution names, to indicate ownership by organizations, still mentioned in PEP 423, appears to have largely fallen out of favor, and we will not use it in the SageMath project.)

A distribution that provides Python modules in the `sage.*` namespace, say mainly from `sage.PAC.KAGE`, should be named **sagemath-DISTRI-BUTION**. Example:

- The distribution sagemath-categories provides a small subset of the modules of the Sage library, mostly from the packages `sage.structure`, `sage.categories`, and `sage.misc`.

Other distributions should not use the prefix **sagemath-** in the distribution name. Example:

- The distribution sage-sws2rst provides the Python package `sage_sws2rst`, so it does not fill the `sage.*` namespace and therefore does not use the prefix **sagemath-**.

A distribution that provides functionality that does not need to import anything from the `sage` namespace should not use the `sage` namespace for its own packages/modules. It should be positioned as part of the general Python ecosystem instead of as a Sage-specific distribution. Examples:

- The distribution pplpy provides the Python package `ppl` and is a much extended version of what used to be `sage.libs.ppl`, a part of the Sage library. The package `sage.libs.ppl` had dependencies on `sage.rings` to convert to/from Sage number types. **pplpy** has no such dependencies and is therefore usable in a wider range of Python projects.
- The distribution memory-allocator provides the Python package `memory_allocator`. This used to be `sage.ext.memory_allocator`, a part of the Sage library.

Ordinary packages vs. implicit namespace packages

Each module of the Sage library must be packaged in exactly one distribution package. However, modules in a package may be included in different distribution packages. In this regard, there is an important constraint that an ordinary package (directory with `__init__.py` file) cannot be split into more than one distribution package.

By removing the `__init__.py` file, however, we can make the package an "implicit" (or "native") "namespace" package, following PEP 420. Implicit namespace packages can be included in more than one distribution package. Hence whenever there are two distribution packages that provide modules with a common prefix of Python packages, that prefix needs to be a implicit namespace package, i.e., there cannot be an `__init__.py` file.

For example,

- **sagemath-tdlib** will provide `sage.graphs.graph_decompositions.tdlib`,
- **sagemath-rw** will provide `sage.graphs.graph_decompositions.rankwidth`,
- **sagemath-graphs** will provide all of the rest of `sage.graphs.graph_decompositions` (and most of `sage.graphs`).

Then, none of

- `sage`,
- `sage.graphs`,
- `sage.graphs.graph_decomposition`

can be an ordinary package (with an `__init__.py` file), but rather each of them has to be an implicit namespace package (no `__init__.py` file).

For an implicit namespace package, `__init__.py` cannot be used any more for initializing the package.

In the Sage 9.6 development cycle, we still use ordinary packages by default, but several packages are converted to implicit namespace packages to support modularization.

Source directories of distribution packages

The development of the Sage library uses a monorepo strategy for all distribution packages that fill the `sage.*` namespace. This means that the source trees of these distributions are included in a single `git` repository, in a subdirectory of `SAGE_ROOT/pkgs`.

All these distribution packages have matching version numbers. From the viewpoint of a single distribution, this means that sometimes there will be a new release of some distribution where the only thing changing is the version number.

The source directory of a distribution package, such as `SAGE_ROOT/pkgs/sagemath-standard`, contains the following files:

- `sage` – a relative symbolic link to the monolithic Sage library source tree `SAGE_ROOT/src/sage/`
- MANIFEST.in – controls which files and directories of the monolithic Sage library source tree are included in the distribution

  The manifest should be kept in sync with the directives of the form `# sage_setup: distribution = sagemath-polyhedra` at the top of source files. Sage provides a tool `sage --fixdistributions` that assists with this task. For example:

  ```
  $ ./sage --fixdistributions --set sagemath-polyhedra \
      src/sage/geometry/polyhedron/base*.py
  ```

  adds or updates the directives in the specified files; and:

  ```
  $ ./sage --fixdistributions --add sagemath-polyhedra \
      src/sage/geometry/polyhedron
  ```

  adds the directive to all files in the given directory that do not include a directive yet.

  After a distribution has been built (for example, by the command `make pypi-wheels`) or at least an sdist has been built (for example, by the command `make sagemath_polyhedra-sdist`), the distribution directives in all files in the source distribution can be updated using the switch `--from--egg-info`:

  ```
  $ ./sage --fixdistributions --set sagemath-polyhedra --from-egg-info
  ```

  To take care of all distributions, use:

  ```
  $ ./sage --fixdistributions --set all --from-egg-info
  ```

- pyproject.toml, setup.cfg, and requirements.txt – standard Python packaging metadata, declaring the distribution name, dependencies, etc.
- `README.rst` – a description of the distribution
- `LICENSE.txt` – relative symbolic link to the same files in `SAGE_ROOT/src`
- `VERSION.txt` – package version. This file is updated by the release manager by running the `sage-update-version` script.

  Sometimes it may be necessary to upload a hotfix for a distribution package to PyPI. These should be marked by adding a suffix `.post1`, `.post2`; see PEP 440 on post-releases. For example, if the current development release is `9.7.beta8`, then such a version could be marked `9.7.beta8.post1`.

  Also sometimes when working on PRs it may be necessary to increment the version because a new feature is needed in another distribution package. Such versions should be marked by using the version number of the anticipated next development release and adding a suffix `.dev1`, `.dev2` … (see PEP 440 on developmental releases). For example, if the current development release is `9.7.beta8`, use `9.7.beta9.dev1`. If the current development release is the stable release `9.8`, use `9.9.beta0.dev1`.

After the PR is merged in the next development version, it will be synchronized again with the other package versions.

- `setup.py` – a setuptools-based installation script
- `tox.ini` – configuration for testing with tox

The technique of using symbolic links pointing into `SAGE_ROOT/src` has allowed the modularization effort to keep the `SAGE_ROOT/src` tree monolithic: Modularization has been happening behind the scenes and will not change where Sage developers find the source files.

Some of these files may actually be generated from source files with suffix `.m4` by the `SAGE_ROOT/bootstrap` script via the `m4` macro processor.

For every distribution package, there is also a subdirectory of `SAGE_ROOT/build/pkgs/`, which contains the build infrastructure that is specific to Sage-the-distribution. Note that these subdirectories follows a different naming convention, using underscores instead of dashes, see *Directory structure*. Because the distribution packages are included in the source tree, we set them up as "script packages" instead of "normal packages", see *Package source types*.

Dependencies and distribution packages

When preparing a portion of the Sage library as a distribution package, dependencies matter.

Build-time dependencies

If the portion of the library contains any Cython modules, these modules are compiled during the wheel-building phase of the distribution package. If the Cython module uses `cimport` to pull in anything from `.pxd` files, these files must be either part of the portion shipped as the distribution being built, or the distribution that provides these files must be installed in the build environment. Also, any C/C++ libraries that the Cython module uses must be accessible from the build environment.

Declaring build-time dependencies: Modern Python packaging provides a mechanism to declare build-time dependencies on other distribution packages via the file pyproject.toml (`[build-system] requires`); this has superseded the older `setup_requires` declaration. (There is no mechanism to declare anything regarding the C/C++ libraries.)

While the namespace `sage.*` is organized roughly according to mathematical fields or categories, how we partition the implementation modules into distribution packages has to respect the hard constraints that are imposed by the build-time dependencies.

We can define some meaningful small distributions that just consist of a single or a few Cython modules. For example, **sagemath-tdlib** (github issue #29864) would just package the single Cython module that must be linked with `tdlib`, `sage.graphs.graph_decompositions.tdlib`. Starting with the Sage 9.6 development cycle, as soon as namespace packages are activated, we can start to create these distributions. This is quite a mechanical task.

Reducing build-time dependencies: Sometimes it is possible to replace build-time dependencies of a Cython module on a library by a runtime dependency. In other cases, it may be possible to split a module that simultaneously depends on several libraries into smaller modules, each of which has narrower dependencies.

Module-level runtime dependencies

Any `import` statements at the top level of a Python or Cython module are executed when the module is imported. Hence, the imported modules must be part of the distribution, or provided by another distribution – which then must be declared as a run-time dependency.

Declaring run-time dependencies: These dependencies are declared in `setup.cfg` (generated from `setup.cfg.m4`) as install_requires.

Reducing module-level run-time dependencies:

- Avoid importing from `sage.PAC.KAGE.all` modules when `sage.PAC.KAGE` is a namespace package. The main purpose of the `*.all` modules is for populating the global interactive environment that is available to users at the `sage:` prompt. In particular, no Sage library code should import from `sage.rings.all`.

  To audit the Sage library for such imports, use `sage --tox -e relint`. In most cases, the imports can be fixed automatically using the tool `sage --fiximports`.

- Replace module-level imports by method-level imports. Note that this comes with a small runtime overhead, which can become noticeable if the method is called in tight inner loops.

- Sage provides the `lazy_import()` mechanism. Lazy imports can be declared at the module level, but the actual importing is only done on demand. It is a runtime error at that time if the imported module is not present. This can be convenient compared to local imports in methods when the same imports are needed in several methods.

- Avoid the "modularization anti-pattern" of importing a class from another module just to run an `isinstance(object, Class)` test, in particular when the module implementing `Class` has heavy dependencies. For example, importing the class `pAdicField` (or the function `is_pAdicField`) requires the libraries NTL and PARI.

  Instead, provide an abstract base class (ABC) in a module that only has light dependencies, make `Class` a subclass of `ABC`, and use `isinstance(object, ABC)`. For example, `sage.rings.abc` provides abstract base classes for many ring (parent) classes, including `sage.rings.abc.pAdicField`. So we can replace:

```
from sage.rings.padics.generic_nodes import pAdicFieldGeneric  # heavy␣
↪dependencies
isinstance(object, pAdicFieldGeneric)
```

  and:

```
from sage.rings.padics.generic_nodes import is_pAdicField      # heavy␣
↪dependencies
is_pAdicField(object)                                          # deprecated
```

  by:

```
import sage.rings.abc                                          # no dependencies
isinstance(object, sage.rings.abc.pAdicField)
```

  Note that going through the abstract base class only incurs a small performance penalty:

```
sage: object = Qp(5)

sage: from sage.rings.padics.generic_nodes import pAdicFieldGeneric
sage: %timeit isinstance(object, pAdicFieldGeneric)            # fast          ␣
↪              # not tested
68.7 ns ± 2.29 ns per loop (...)

sage: import sage.rings.abc
```

(continues on next page)

(continued from previous page)

```
sage: %timeit isinstance(object, sage.rings.abc.pAdicField)     # also fast
↪              # not tested
122 ns ± 1.9 ns per loop (...)
```

- If it is not possible or desired to create an abstract base class for `isinstance` testing (for example, when the class is defined in some external package), other solutions need to be used.

  Note that Python caches successful module imports, but repeating an unsuccessful module import incurs a cost every time:

```
sage: from sage.schemes.generic.scheme import Scheme
sage: sZZ = Scheme(ZZ)

sage: def is_Scheme_or_Pluffe(x):
....:     if isinstance(x, Scheme):
....:         return True
....:     try:
....:         from xxxx_does_not_exist import Pluffe            # slow on every
↪call
....:     except ImportError:
....:         return False
....:     return isinstance(x, Pluffe)

sage: %timeit is_Scheme_or_Pluffe(sZZ)                          # fast
↪              # not tested
111 ns ± 1.15 ns per loop (...)

sage: %timeit is_Scheme_or_Pluffe(ZZ)                           # slow
↪              # not tested
143 µs ± 2.58 µs per loop (...)
```

  The `lazy_import()` mechanism can be used to simplify this pattern via the `__instancecheck__()` method and has similar performance characteristics:

```
sage: lazy_import('xxxx_does_not_exist', 'Pluffe')

sage: %timeit isinstance(sZZ, (Scheme, Pluffe))                 # fast
↪              # not tested
95.2 ns ± 0.636 ns per loop (...)

sage: %timeit isinstance(ZZ, (Scheme, Pluffe))                  # slow
↪              # not tested
158 µs ± 654 ns per loop (...)
```

  It is faster to do the import only once, for example when loading the module, and to cache the failure. We can use the following idiom, which makes use of the fact that `isinstance` accepts arbitrarily nested lists and tuples of types:

```
sage: try:
....:     from xxxx_does_not_exist import Pluffe                # runs once
....: except ImportError:
....:     # Set to empty tuple of types for isinstance
....:     Pluffe = ()

sage: %timeit isinstance(sZZ, (Scheme, Pluffe))                 # fast
↪              # not tested
```

(continues on next page)

(continued from previous page)

```
95.9 ns ± 1.52 ns per loop (...)

sage: %timeit isinstance(ZZ, (Scheme, Pluffe))                # fast           ␣
↪              # not tested
126 ns ± 1.9 ns per loop (...)
```

Other runtime dependencies

If `import` statements are used within a method, the imported module is loaded the first time that the method is called. Hence the module defining the method can still be imported even if the module needed by the method is not present.

It is then a question whether a run-time dependency should be declared. If the method needing that import provides core functionality, then probably yes. But if it only provides what can be considered "optional functionality", then probably not, and in this case it will be up to the user to install the distribution enabling this optional functionality.

As an example, let us consider designing a distribution that centers around the package `sage.coding`. First, let's see if it uses symbolics:

```
(9.5.beta6) $ git grep -E 'sage[.](symbolic|functions|calculus)' src/sage/coding
src/sage/coding/code_bounds.py:        from sage.functions.other import ceil
...
src/sage/coding/grs_code.py:from sage.symbolic.ring import SR
...
src/sage/coding/guruswami_sudan/utils.py:from sage.functions.other import floor
```

Apparently it does not in a very substantial way:

- The imports of the symbolic functions `ceil()` and `floor()` can likely be replaced by the artithmetic functions `integer_floor()` and `integer_ceil()`.
- Looking at the import of `SR` by `sage.coding.grs_code`, it seems that `SR` is used for running some symbolic sum, but the doctests do not show symbolic results, so it is likely that this can be replaced.
- Note though that the above textual search for the module names is merely a heuristic. Looking at the source of "entropy", through `log` from `sage.misc.functional`, a runtime dependency on symbolics comes in. In fact, for this reason, two doctests there are already marked as `# needs sage.symbolic`.

So if packaged as **sagemath-coding**, now a domain expert would have to decide whether these dependencies on symbolics are strong enough to declare a runtime dependency (`install_requires`) on **sagemath-symbolics**. This declaration would mean that any user who installs **sagemath-coding** (`pip install sagemath-coding`) would pull in **sagemath-symbolics**, which has heavy compile-time dependencies (ECL/Maxima/FLINT/Singular/…).

The alternative is to consider the use of symbolics by **sagemath-coding** merely as something that provides some extra features, which will only be working if the user also has installed **sagemath-symbolics**.

Declaring optional run-time dependencies: It is possible to declare such optional dependencies as extras_require in `setup.cfg` (generated from `setup.cfg.m4`). This is a very limited mechanism – in particular it does not affect the build phase of the distribution in any way. It basically only provides a way to give a nickname to a distribution that can be installed as an add-on.

In our example, we could declare an `extras_require` so that users could use `pip install sagemath-coding[symbolics]`.

Doctest-only dependencies

Doctests often use examples constructed using functionality provided by other portions of the Sage library. This kind of integration testing is one of the strengths of Sage; but it also creates extra dependencies.

Fortunately, these dependencies are very mild, and we can deal with them using the same mechanism that we use for making doctests conditional on the presence of optional libraries: using `# optional - FEATURE` directives in the doctests. Adding these directives will allow developers to test the distribution separately, without requiring all of Sage to be present.

Declaring doctest-only dependencies: The extras_require mechanism mentioned above can also be used for this.

Dependencies of the Sage documentation

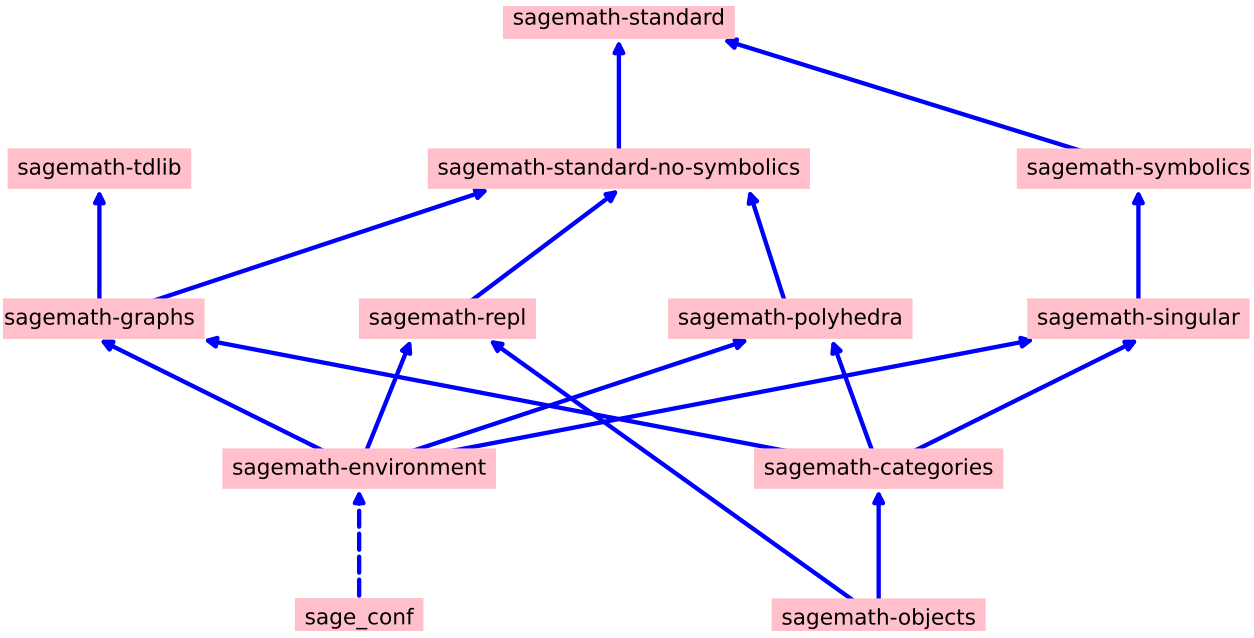
The documentation will not be modularized.

However, some parts of the Sage reference manual may depend on functionality provided by optional packages. These portions of the reference manual should be conditionalized using the Sphinx directive `.. ONLY::`, as explained in *Making portions of the reference manual conditional on optional features*.

Version constraints of dependencies

The version information for dependencies comes from the files `build/pkgs/*/install-requires.txt` and `build/pkgs/*/package-version.txt`. We use the m4 macro processor to insert the version information in the generated files `pyproject.toml`, `setup.cfg`, `requirements.txt`.

Hierarchy of distribution packages



Solid arrows indicate `install_requires`, i.e., a declared runtime dependency. Dashed arrows indicate `extras_require`, i.e., a declared optional runtime dependency. Not shown in the diagram are build dependencies and optional dependencies for testing.

- sage_conf is a configuration module. It provides the configuration variable settings determined by the `configure` script.
- sagemath-environment provides the connection to the system and software environment. It includes `sage.env`, `sage.features`, `sage.misc.package_dir`, etc.
- sagemath-objects provides a small fundamental subset of the modules of the Sage library, in particular all of `sage.structure`, a small portion of `sage.categories`, and a portion of `sage.misc`.
- sagemath-categories provides a small subset of the modules of the Sage library, building upon sagemath-objects. It provides all of `sage.categories` and a small portion of `sage.rings`.
- sagemath-repl provides the IPython kernel and Sage preparser (`sage.repl`), the Sage doctester (`sage.doctest`), and some related modules from `sage.misc`.

Testing distribution packages

Of course, we need tools for testing modularized distributions of portions of the Sage library.

- Distribution packages of the modularized Sage library must be testable separately!
- But we want to keep integration testing with other portions of Sage too!

Preparing doctests for modularized testing

Section *Writing testable examples* explains how to write doctests for Sage. Here we show how to prepare existing or new doctests so that they are suitable for modularized testing.

Per section *Special markup to influence doctests*, whenever an optional package is needed for a particular test, we use the doctest tag `# optional`. This mechanism can also be used for making a doctest conditional on the presence of a portion of the Sage library.

The available tags take the form of package or module names such as `sage.combinat`, `sage.graphs`, `sage.plot`, `sage.rings.number_field`, `sage.rings.real_double`, and `sage.symbolic`. They are defined via `Feature` subclasses in the module `sage.features.sagemath`, which also provides the mapping from features to the distributions providing them (actually, to SPKG names). Using this mapping, Sage can issue installation hints to the user.

For example, the package `sage.tensor` is purely algebraic and has no dependency on symbolics. However, there are a small number of doctests that depend on `sage.symbolic.ring.SymbolicRing` for integration testing. Hence, these doctests are marked as depending on the feature `sage.symbolic`.

By convention, because `sage.symbolic` is present in a standard installation of Sage, we use the keyword `# needs` instead of `# optional`. These two keywords have identical semantics; the tool *sage –fixdoctests* rewrites the doctest tags according to the convention.

When defining new features for the purpose of conditionalizing doctests, it may be a good idea to hide implementation details from feature names. For example, all doctests that use large finite fields have to depend on PARI. However, we have defined a feature `sage.rings.finite_rings` (which implies the presence of `sage.libs.pari`). Marking the doctests `# needs sage.rings.finite_rings` expresses the dependency in a clearer way than using `# needs sage.libs.pari`, and it will be a smaller maintenance burden when implementation details change.

Testing the distribution in virtual environments with tox

Chapter *Running Sage's Doctests* explains in detail how to run the Sage doctester with various options.

To test a distribution package of the modularized Sage library, we use a virtual environment in which we only install the distribution to be tested (and its Python dependencies).

Let's try it out first with the entire Sage library, represented by the distribution **sagemath-standard**. Note that after Sage has been built normally, a set of wheels for most installed Python distribution packages is available in `SAGE_VENV/var/lib/sage/wheels/`:

```
$ ls venv/var/lib/sage/wheels
Babel-2.9.1-py2.py3-none-any.whl
Cython-0.29.24-cp39-cp39-macosx_11_0_x86_64.whl
Jinja2-2.11.2-py2.py3-none-any.whl
...
scipy-1.7.2-cp39-cp39-macosx_11_0_x86_64.whl
setuptools-58.2.0-py3-none-any.whl
...
wheel-0.37.0-py2.py3-none-any.whl
widgetsnbextension-3.5.1-py2.py3-none-any.whl
zipp-3.5.0-py3-none-any.whl
```

However, in a build of Sage with the default configuration `configure --enable-editable`, there will be no wheels for the distributions `sage_*` and `sagemath-*`.

To create these wheels, use the command `make wheels`:

```
$ make wheels
...
$ ls venv/var/lib/sage/wheels/sage*
...
sage_conf-10.0b2-py3-none-any.whl
...
```

(You can also use `./configure --enable-wheels` to ensure that these wheels are always available and up to date.)

Note in particular the wheel for **sage-conf**, which provides configuration variable settings and the connection to the non-Python packages installed in `SAGE_LOCAL`.

We can now set up a separate virtual environment, in which we install these wheels and our distribution to be tested. This is where tox comes into play: It is the standard Python tool for creating disposable virtual environments for testing. Every distribution in `SAGE_ROOT/pkgs/` provides a configuration file `tox.ini`.

Following the comments in the file `SAGE_ROOT/pkgs/sagemath-standard/tox.ini`, we can try the following command:

```
$ ./bootstrap && ./sage -sh -c '(cd pkgs/sagemath-standard && SAGE_NUM_THREADS=16 tox␣
↪-v -v -e sagepython-sagewheels-nopypi)'
```

This command does not make any changes to the normal installation of Sage. The virtual environment is created in a subdirectory of `SAGE_ROOT/pkgs/sagemath-standard/.tox/`. After the command finishes, we can start the separate installation of the Sage library in its virtual environment:

```
$ pkgs/sagemath-standard/.tox/sagepython-sagewheels-nopypi/bin/sage
```

We can also run parts of the testsuite:

```
$ pkgs/sagemath-standard/.tox/sagepython-sagewheels-nopypi/bin/sage -tp 4 src/sage/
↪graphs/
```

The whole `.tox` directory can be safely deleted at any time.

We can do the same with other distributions, for example the large distribution **sagemath-standard-no-symbolics** (from github issue #35095), which is intended to provide everything that is currently in the standard Sage library, i.e., without depending on optional packages, but without the packages `sage.symbolic`, `sage.calculus`, etc.

Again we can run the test with `tox` in a separate virtual environment:

```
$ ./bootstrap && make wheels && ./sage -sh -c '(cd pkgs/sagemath-standard-no-
↪symbolics && SAGE_NUM_THREADS=16 tox -v -v -v -e sagepython-sagewheels-nopypi-
↪norequirements)'
```

Some small distributions, for example the ones providing the two lowest levels, sagemath-objects and sagemath-categories (from github issue #29865), can be installed and tested without relying on the wheels from the Sage build:

```
$ ./bootstrap && ./sage -sh -c '(cd pkgs/sagemath-objects && SAGE_NUM_THREADS=16 tox -
↪v -v -v -e sagepython)'
```

This command finds the declared build-time and run-time dependencies on PyPI, either as source tarballs or as prebuilt wheels, and builds and installs the distribution sagemath-objects in a virtual environment in a subdirectory of `pkgs/sagemath-objects/.tox`.

Building these small distributions serves as a valuable regression testsuite. However, a current issue with both of these distributions is that they are not separately testable: The doctests for these modules depend on a lot of other functionality from higher-level parts of the Sage library. This is being addressed in github issue #35095.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- search

BIBLIOGRAPHY

[SageComponents] See http://www.sagemath.org/links-components.html for a list

INDEX

C

CFLAGS, 141
CXXFLAGS, 141

E

environment variable
    CFLAGS, 141
    CXXFLAGS, 141
    F77FLAGS, 141
    FCFLAGS, 141
    MAKE, 65
    SAGE_LOCAL, 102, 103
    SAGE_ROOT, 4, 102
    SAGE_VENV, 103

F

F77FLAGS, 141
FCFLAGS, 141

M

MAKE, 65

P

Python Enhancement Proposals
    PEP 0008, 37
    PEP 0257, 37

S

SAGE_LOCAL, 102, 103
SAGE_ROOT, 4, 102
SAGE_VENV, 103