SciPy Core Developer Guide

Decision making process

SciPy has a formal governance model, documented in SciPy Project Governance. The section below documents in an informal way what happens in practice for decision making about code and commit rights. The formal governance model is leading, the below is only provided for context.

Code ¶

Any significant decisions on adding (or not adding) new features, breaking backwards compatibility or making other significant changes to the codebase should be made on the scipy-dev mailing list after a discussion (preferably with full consensus).

Any non-trivial change (where trivial means a typo, or a one-liner maintenance commit) has to go in through a pull request (PR). It has to be reviewed by another developer. In case review doesn't happen quickly enough and it is important that the PR is merged quickly, the submitter of the PR should send a message to mailing list saying he/she intends to merge that PR without review at time X for reason Y unless someone reviews it before then.

Changes and new additions should be tested. Untested code is broken code.

Commit rights

Who gets commit rights is decided by the SciPy Steering Council; changes in commit rights will then be announced on the scipy-dev mailing list.

Deciding on new features

The general decision rule to accept a proposed new feature has so far been conditional on:

1. The method is applicable in many fields and "generally agreed" to be useful,
2. It fits the topic of the submodule, and does not require extensive support frameworks to operate,
3. The implementation looks sound and unlikely to need much tweaking in the future (e.g., limited expected maintenance burden),
4. Someone wants to contribute it, and
5. Someone wants to review it.

The last criterion is often a sticking point for proposed features. Code cannot be merged until it has been thoroughly reviewed, and there is always a backlog of maintenance tasks that compete for reviewers' time. Ideally, contributors should line up a reviewer with suitable domain expertise before beginning work.

Although it's difficult to give hard rules on what "generally useful and generally agreed to work" means, it may help to weigh the following against each other:

- Is the method used/useful in different domains in practice? How much domain-specific background knowledge is needed to use it properly?
- Consider the code already in the module. Is what you are adding an omission? Does it solve a problem that you'd expect the module be able to solve? Does it supplement an existing feature in a significant way?
- Consider the equivalence class of similar methods / features usually expected. Among them, what would in principle be the minimal set so that there's not a glaring omission in the offered features remaining? How much stuff would that be? Does including a representative one of them cover most use cases? Would it in principle sound reasonable to include everything from the minimal set in the module?
- Is what you are adding something that is well understood in the literature? If not, how sure are you that it will turn out well? Does the method perform well compared to other similar ones?
- Note that the twice-a-year release cycle and backward-compatibility policy makes correcting things later on more difficult.

The scopes of the submodules also vary, so it's probably best to consider each as if it's a separate project - "numerical evaluation of special functions" is relatively well-defined, but "commonly needed optimization algorithms" less so.

Development on GitHub

SciPy development largely takes place on GitHub; this section describes the expected way of working for issues, pull requests and managing the main `scipy` repository.

Labels and Milestones

Each issue and pull request normally gets at least two labels: one for the topic or component (`scipy.stats`, `Documentation`, etc.), and one for the nature of the issue or pull request (`enhancement`, `maintenance`, `defect`, etc.). Other labels that may be added depending on the situation:

- `easy-fix`: for issues suitable to be tackled by new contributors.
- `needs-work`: for pull requests that have review comments that haven't been addressed.
- `needs-decision`: for issues or pull requests that need a decision.
- `needs-champion`: for pull requests that were not finished by the original author, but are worth resurrecting.
- `backport-candidate`: bugfixes that should be considered for backporting by the release manager.

A milestone is created for each version number for which a release is planned. Issues that need to be addressed and pull requests that need to be merged for a particular release should be set to the corresponding milestone. After a pull request is merged, its milestone (and that of the issue it closes) should be set to the next upcoming release - this makes it easy to get an overview of changes and to add a complete list of those to the release notes.

Pull request review workflow

When reviewing pull requests, please make use of pull request workflow features, see Using workflow features.

Dealing with pull requests

- When merging contributions, a committer is responsible for ensuring that those meet the requirements outlined in Contributing to SciPy. Also check that new features and backwards compatibility breaks were discussed on the scipy-dev mailing list.
- New code goes in via a pull request (PR).
- Merge new code with the green button. In case of merge conflicts, ask the PR submitter to rebase (this may require providing some git instructions).
- Backports and trivial additions to finish a PR (really trivial, like a typo or PEP8 fix) can be pushed directly.
- For PRs that add new features or are in some way complex, wait at least a day or two before merging it. That way, others get a chance to comment before the code goes in.
- Squashing commits or cleaning up commit messages of a PR that you consider too messy is OK. Make sure though to retain the original author name when doing this.
- Make sure that the labels and milestone on a merged PR are set correctly.
- When you want to reject a PR: if it's very obvious you can just close it and explain why, if not obvious then it's a good idea to first explain why you think the PR is not suitable for inclusion in SciPy and then let a second committer comment or close.

Backporting

All pull requests (whether they contain enhancements, bug fixes or something else), should be made against master. Only bug fixes are candidates for backporting to a maintenance branch. The backport strategy for SciPy is to (a) only backport fixes that are important, and (b) to only backport when it's reasonably sure that a new bugfix release on the relevant maintenance branch will be made. Typically, the developer who merges an important bugfix adds the `backport-candidate` label and pings the release manager, who decides on whether and when the backport is done. After the backport is completed, the `backport-candidate` label has to be removed again.

A good strategy for a backport pull request is to combine several master branch pull requests, to reduce the burden on continuous integration tests and to reduce the merge commit cluttering of maintenance branch history. It is generally best to have a single commit for each of the master branch pull requests represented in the backport pull request. This way, history is clear and can be reverted in a straightforward manner if needed.

Release notes

When a PR gets merged, consider if the changes need to be mentioned in the release notes. What needs mentioning: new features, backwards incompatible changes, deprecations, and "other changes" (anything else noteworthy enough, see older release notes for the kinds of things worth mentioning).

Release note entries are maintained on the wiki, (e.g. https://github.com/scipy/scipy/wiki/Release-note-entries-for-SciPy-1.2.0). The release manager will gather content from there and integrate it into the html docs. We use this mechanism to avoid merge conflicts that would happen if every PR touched the same file under `doc/release/` directly.

Changes can be monitored (Atom feed) and pulled (the wiki is a git repo: `https://github.com/scipy/scipy.wiki.git`).

Other

PR status page: When new commits get added to a pull request, GitHub doesn't send out any notifications. The `needs-work` label may not be justified anymore though. This page gives an overview of PRs that were updated, need review, need a decision, etc.

Cross-referencing: Cross-referencing issues and pull requests on GitHub is often useful. GitHub allows doing that by using `gh-xxxx` or `#xxxx` with `xxxx` the issue/PR number. The `gh-xxxx` format is strongly preferred, because it's clear that that is a GitHub link. Older issues contain `#xxxx` which is about Trac (what we used pre-GitHub) tickets.

PR naming convention: Pull requests, issues and commit messages usually start with a three-letter abbreviation like `ENH:` or `BUG:`. This is useful to quickly see what the nature of the commit/PR/issue is. For the full list of abbreviations, see writing the commit message.

Licensing

SciPy is distributed under the modified (3-clause) BSD license. All code, documentation and other files added to SciPy by contributors is licensed under this license, unless another license is explicitly specified in the source code. Contributors keep the copyright for code they wrote and submit for inclusion to SciPy.

Other licenses that are compatible with the modified BSD license that SciPy uses are 2-clause BSD, MIT and PSF. Incompatible licenses are GPL, Apache and custom licenses that require attribution/citation or prohibit use for commercial purposes.

PRs are often submitted with content copied or derived from unlicensed code or code from a default license that is not compatible with SciPy's license. For instance, code published on StackOverflow is covered by a CC-BY-SA license, which is not compatible due to the share-alike clause. These contributions cannot be accepted for inclusion in SciPy unless the original code author is willing to (re)license his/her code under the modified BSD (or compatible) license. If the original author agrees, add a comment saying so to the source files and forward the relevant communication to the scipy-dev mailing list.

Another common occurrence is for code to be translated or derived from code in R, Octave (both GPL-licensed) or a commercial application. Such code also cannot be included in SciPy. Simply implementing functionality with the same API as found in R/Octave/… is fine though, as long as the author doesn't look at the original incompatibly-licensed source code.

Version numbering

SciPy version numbering complies to PEP 440. Released final versions, which are the only versions appearing on PyPI, are numbered `MAJOR.MINOR.MICRO` where:

- `MAJOR` is an integer indicating the major version. It changes very rarely; a change in `MAJOR` indicates large (possibly backwards-incompatible) changes.
- `MINOR` is an integer indicating the minor version. Minor versions are typically released twice a year and can contain new features, deprecations and bug-fixes.
- `MICRO` is an integer indicating a bug-fix version. Bug-fix versions are released when needed, typically one or two per minor version. They cannot contain new features or deprecations.

Released alpha, beta and rc (release candidate) versions are numbered like final versions but with postfixes `a#`, `b#` and `rc#` respectively, with `#` an integer. Development versions are postfixed with `.dev0+<git-commit-hash>`.

Examples of valid SciPy version strings are:

```
0.16.0
0.15.1
0.14.0a1
0.14.0b2
0.14.0rc1
0.17.0.dev0+ac53f09
```

An installed SciPy version contains these version identifiers:

```
scipy.__version__            # complete version string, including git commit hash for dev
versions
scipy.version.short_version  # string, only major.minor.micro
scipy.version.version        # string, same as scipy.__version__
scipy.version.full_version   # string, same as scipy.__version__
scipy.version.release        # bool, development or (alpha/beta/rc/final) released version
scipy.version.git_revision   # string, git commit hash from which scipy was built
```

Deprecations

There are various reasons for wanting to remove existing functionality: it's buggy, the API isn't understandable, it's superseded by functionality with better performance, it needs to be moved to another SciPy submodule, etc.

In general it's not a good idea to remove something without warning users about that removal first. Therefore this is what should be done before removing something from the public API:

1. Propose to deprecate the functionality on the scipy-dev mailing list and get agreement that that's OK.
2. Add a `DeprecationWarning` for it, which states that the functionality was deprecated, and in which release. For Cython APIs, see Deprecating public Cython APIs for the practical steps.
3. Mention the deprecation in the release notes for that release.
4. Wait till at least 6 months after the release date of the release that introduced the `DeprecationWarning` before removing the functionality.
5. Mention the removal of the functionality in the release notes.

The 6 months waiting period in practice usually means waiting two releases. When introducing the warning, also ensure that those warnings are filtered out when running the test suite so they don't pollute the output.

It's possible that there is reason to want to ignore this deprecation policy for a particular deprecation; this can always be discussed on the scipy-dev mailing list.

Distributing

Distributing Python packages is nontrivial - especially for a package with complex build requirements like SciPy - and subject to change. For an up-to-date overview of recommended tools and techniques, see the Python Packaging User Guide. This document discusses some of the main issues and considerations for SciPy.

Dependencies

Dependencies are things that a user has to install in order to use (or build/test) a package. They usually cause trouble, especially if they're not optional. SciPy tries to keep its dependencies to a minimum; currently they are:

Unconditional run-time dependencies:

- Numpy

Conditional run-time dependencies:

- pytest (to run the test suite)
- asv (to run the benchmarks)
- matplotlib (for some functions that can produce plots)
- Pillow (for image loading/saving)
- scikits.umfpack (optionally used in `sparse.linalg`)
- mpmath (for more extended tests in `special`)
- pydata/sparse (compatibility support in `scipy.sparse`)

Unconditional build-time dependencies:

- Numpy
- A BLAS and LAPACK implementation (reference BLAS/LAPACK, ATLAS, OpenBLAS, MKL are all known to work)
- Cython
- setuptools
- pybind11

Conditional build-time dependencies:

- wheel (`python setup.py bdist_wheel`)

- Sphinx (docs)
- PyData Sphinx theme (docs)
- Sphinx-Panels (docs)
- matplotlib (docs)
- LaTeX (pdf docs)
- Pillow (docs)

Furthermore of course one needs C, C++ and Fortran compilers to build SciPy, but those we don't consider to be dependencies and are therefore not discussed here. For details, see https://scipy.github.io/devdocs/building/.

When a package provides useful functionality and it's proposed as a new dependency, consider also if it makes sense to vendor (i.e. ship a copy of it with scipy) the package instead. For example, decorator is vendored in `scipy._lib`.

The only dependency that is reported to pip is Numpy, see `install_requires` in SciPy's main `setup.py`. The other dependencies aren't needed for SciPy to function correctly

Issues with dependency handling

There are some issues with how Python packaging tools handle dependencies reported by projects. Because SciPy gets regular bug reports about this, we go in a bit of detail here.

SciPy only reports its dependency on NumPy via `install_requires` if NumPy isn't installed at all on a system, or when building wheels with `bdist_wheel`. SciPy no longer uses `setup_requires` (which in the past invoked `easy_install`); build dependencies are now handled only via `pyproject.toml`. `pyproject.toml` relies on PEP 517; `pip` has `--no-use-pep517` and `--no-build-isolation` flags that may ignore `pyproject.toml` or treat it differently - if users use those flags, they are responsible for installing the correct build dependencies themselves.

Version ranges for NumPy and other dependencies

For dependencies it's important to set lower and upper bounds on their versions. For *build-time* dependencies, they are specified in `pyproject.toml` and the versions will *only* apply to the SciPy build itself. It's fine to specify either a range or a specific version for a dependency like `wheel` or `setuptools`. For NumPy we have to worry about ABI compatibility too, hence we specify the version with `==` to the lowest supported version (because NumPy's ABI is backward but not forward compatible).

For *run-time dependencies* (currently only `numpy`), we specify the range of versions in `pyproject.toml` and in `install_requires` in `setup.py`. Getting the upper bound right is slightly tricky. If we don't set any bound, a too-new version will be pulled in a few years down the line, and NumPy may have deprecated and removed some API that SciPy depended on by then. On the other hand if we set the upper bound to the newest already-released version, then as soon as a new NumPy version is released there will be no matching SciPy version that works with it. Given that NumPy and SciPy both release in a 6-monthly cadence and that features that get deprecated in NumPy should stay around for another two releases, we specify the upper bound as `<1.xx+3.0` (where `xx` is the minor version of the latest already-released NumPy.

Supported Python and NumPy versions

The Python versions that SciPy supports are listed in the list of PyPI classifiers in `setup.py`, and mentioned in the release notes for each release. All newly released Python versions will be supported as soon as possible. For the general policy on dropping support for a Python or NumPy version, see NEP 29. The final decision on dropping support is always taken on the scipy-dev mailing list.

The lowest supported Numpy version for a SciPy version is mentioned in the release notes and is encoded in `pyproject.toml`, `scipy/__init__.py` and the `install_requires` field of `setup.py`. Typically the latest SciPy release supports ~5-7 minor versions of NumPy: up to 2.5 years' old NumPy versions, (given that the frequency of NumPy releases is about 2x/year at the time of writing) plus two versions into the future.

Supported versions of optional dependencies and compilers is documented in Toolchain Roadmap. Note that not all versions of optional dependencies that are supported are tested well or at all by SciPy's Continuous Integration setup. Issues regarding this are dealt with as they come up in the issue tracker or mailing list.

Building binary installers

> **Note**
>
> This section is only about building SciPy binary installers to *distribute*. For info on building SciPy on the same machine as where it will be used, see this scipy.org page.

There are a number of things to take into consideration when building binaries and distributing them on PyPI or elsewhere.

General

- A binary is specific for a single Python version (because different Python versions aren't ABI-compatible, at least up to Python 3.4).
- Build against the lowest NumPy version that you need to support, then it will work for all NumPy versions with the same major version number (NumPy does maintain backwards ABI compatibility).

Windows

- The currently most easily available toolchain for building Python.org compatible binaries for SciPy is installing MSVC (see https://wiki.python.org/moin/WindowsCompilers) and mingw64-gfortran. Support for this configuration requires numpy.distutils from NumPy >= 1.14.dev and a gcc/gfortran-compiled static `openblas.a`. This configuration is currently used in the Appveyor configuration for https://github.com/MacPython/scipy-wheels
- For 64-bit Windows installers built with a free toolchain, use the method documented at https://github.com/numpy/numpy/wiki/Mingw-static-toolchain. That method will likely be used for SciPy itself once it's clear that the maintenance of that toolchain is sustainable long-term. See the MingwPy project and this thread for details.
- The other way to produce 64-bit Windows installers is with `icc`, `ifort` plus `MKL` (or `MSVC` instead of `icc`). For Intel toolchain instructions see this article and for (partial) MSVC instructions see this wiki page.
- Older SciPy releases contained a .exe "superpack" installer. Those contain 3 complete builds (no SSE, SSE2, SSE3), and were built with https://github.com/numpy/numpy-vendor. That build setup is known to not work well anymore and is no longer supported. It used g77 instead of gfortran, due to complex DLL distribution issues (see gh-2829). Because the toolchain is no longer supported, g77 support isn't needed anymore and SciPy can now include Fortran 90/95 code.

OS X

- To produce OS X wheels that work with various Python versions (from python.org, Homebrew, MacPython), use the build method provided by https://github.com/MacPython/scipy-wheels.
- DMG installers for the Python from python.org on OS X can still be produced by `tools/scipy-macosx-installer/`. SciPy doesn't distribute those installers anymore though, now that there are binary wheels on PyPi.

Linux

- PyPi-compatible Linux wheels can be produced via the manylinux project. The corresponding build setup for TravisCI for SciPy is set up in https://github.com/MacPython/scipy-wheels.

Other Linux build-setups result to PyPi incompatible wheels, which would need to be distributed via custom channels, e.g. in a Wheelhouse, see at the wheel and Wheelhouse docs.

Making a SciPy release

At the highest level, this is what the release manager does to release a new SciPy version:

1. Propose a release schedule on the scipy-dev mailing list.
2. Create the maintenance branch for the release.
3. Tag the release.
4. Build all release artifacts (sources, installers, docs).
5. Upload the release artifacts.
6. Announce the release.
7. Port relevant changes to release notes and build scripts to master.

In this guide we attempt to describe in detail how to perform each of the above steps. In addition to those steps, which have to be performed by the release manager, here are descriptions of release-related activities and conventions of interest:

- Backporting
- Labels and Milestones
- Version numbering
- Supported Python and NumPy versions
- Deprecations

Proposing a release schedule

A typical release cycle looks like:

- Create the maintenance branch
- Release a beta version

- Release a “release candidate” (RC)
- If needed, release one or more new RCs
- Release the final version once there are no issues with the last release candidate

There’s usually at least one week between each of the above steps. Experience shows that a cycle takes between 4 and 8 weeks for a new minor version. Bug-fix versions don’t need a beta or RC, and can be done much quicker.

Ideally the final release is identical to the last RC, however there may be minor difference - it’s up to the release manager to judge the risk of that. Typically, if compiled code or complex pure Python code changes then a new RC is needed, while a simple bug-fix that’s backported from master doesn’t require a new RC.

To propose a schedule, send a list with estimated dates for branching and beta/rc/final releases to scipy-dev. In the same email, ask everyone to check if there are important issues/PRs that need to be included and aren’t tagged with the Milestone for the release or the “backport-candidate” label.

Creating the maintenance branch

Before branching, ensure that the release notes are updated as far as possible. Include the output of `tools/gh_lists.py` and `tools/authors.py` in the release notes.

Maintenance branches are named `maintenance/<major>.<minor>.x` (e.g. 0.19.x). To create one, simply push a branch with the correct name to the scipy repo. Immediately after, push a commit where you increment the version number on the master branch and add release notes for that new version. Send an email to scipy-dev to let people know that you’ve done this.

Updating upper bounds of dependencies

In master we do not set upper bounds, because we want to test new releases or development versions of dependencies there. In a maintenance branch however, the goal is to be able to create releases that stay working for years. Hence correct upper bounds must be set. The following places must be updated after creating a maintenance branch:

- **`pyproject.toml`: all build-time dependencies, as well as supported Python**

  and NumPy versions

- `setup.py`: supported Python and NumPy versions
- `scipy/__init__.py`: for NumPy version check

Each file has comments describing how to set the correct upper bounds.

Tagging a release

First ensure that you have set up GPG correctly. See https://github.com/scipy/scipy/issues/4919 for a discussion of signing release tags, and https://keyring.debian.org/creating-key.html for instructions on creating a GPG key if you do not have one. Note that on some platforms it may be more suitable to use `gpg2` instead of `gpg` so that passwords may be stored by `gpg-agent` as discussed in https://github.com/scipy/scipy/issues/10189. When preparing a release remotely, it may be necessary to set `pinentry-mode loopback` in `~/.gnupg/gpg-agent.conf` because use of `gpg2` will otherwise proceed via an inaccessible graphical password prompt.

To make your key more readily identifiable as you, consider sending your key to public keyservers, with a command such as:

```
gpg --send-keys <yourkeyid>
```

Check that all relevant commits are in the branch. In particular, check issues and PRs under the Milestone for the release (https://github.com/scipy/scipy/milestones), PRs labeled “backport-candidate”, and that the release notes are up-to-date and included in the html docs.

Then edit `setup.py` to get the correct version number (set `ISRELEASED = True`) and commit it with a message like `REL: set version to <version-number>`. Don’t push this commit to the SciPy repo yet.

Finally tag the release locally with `git tag -s <v1.x.y>` (the `-s` ensures the tag is signed). If `gpg2` is preferred, then `git config --global gpg.program gpg2` may be appropriate. Continue with building release artifacts (next section). Only push the release commit to the scipy repo once you have built the sdists and docs successfully. Then continue with building wheels. Only push the release tag to the repo once all wheels have been built successfully on TravisCI and Appveyor (if it fails, you have to move the tag otherwise - which is bad practice). Finally, after pushing the tag, also push a second commit which increments the version number and sets `ISRELEASED` to False again. This also applies with new release candidates, and for removing the `rc` affix when switching from release candidate to release proper.

Building release artifacts

Here is a complete list of artifacts created for a release:

- source archives (`.tar.gz`, `.zip` and `.tar.xz` for GitHub Releases, only `.tar.gz` is uploaded to PyPI)
- Binary wheels for Windows, Linx and OS X
- Documentation (`html`, `pdf`)
- A `README` file
- A `Changelog` file

Source archives, Changelog and README are built by running `paver release` in the repo root, and end up in `REPO_ROOT/release/`. Do this after you've created the signed tag locally. `paver release` will be sensitive to the version of Cython available in your build environment, so make sure your version matches the minimum requirements for the release. If this completes without issues, push the release commit (not the tag, see section above) to the scipy repo. If `pavement.py` is causing issues, it is also possible to simply use `python setup.py sdist` and perform the release notes task from `pavement.py` by hand.

To build wheels, push a commit to a branch used for the current release at https://github.com/MacPython/scipy-wheels . This triggers builds for all needed Python versions on TravisCI. Update and check the `.travis.yml` and `appveyor.yml` config files what commit to build, and what Python and NumPy are used for the builds (it needs to be the lowest supported NumPy version for each Python version). See the README file in the scipy-wheels repo for more details. Note that because several months may pass between `SciPy` releases, it is sometimes necessary to update the versions of the `gfortran-install` and `multibuild` submodules used for wheel builds. If the wheels builds reveal issues that need to be fixed with backports on the maintenance branch, you may remove the local tags (for example `git tag -d v1.2.0rc1`) and restart with tagging above on the new candidate commit.

The TravisCI and Appveyor builds run the tests from the built wheels and if they pass, upload the wheels to a container pointed to at https://github.com/MacPython/scipy-wheels Once there are successful wheel builds, it is recommended to create a versioned branch in the `scipy-wheels` repo, which will for example be adjusted to point to different maintenance branch commits if there are multiple release candidates.

From there you can download them for uploading to PyPI. This can be done in an automated fashion using `tools/download-wheels.py`:

```
$ python tools/download-wheels.py 1.5.0rc1 -w REPO_ROOT/release/installers
```

The correct URL to use is shown in https://github.com/MacPython/scipy-wheels and should agree with the above one.

After this, we want to regenerate the README file, in order to have the MD5 and SHA256 checksums of the just downloaded wheels in it. Run:

```
$ paver write_release_and_log
```

Uploading release artifacts

For a release there are currently five places on the web to upload things to:

- PyPI (tarballs, wheels)
- Github releases (tarballs, release notes, Changelog)
- scipy.org (an announcement of the release)
- docs.scipy.org (html/pdf docs)

PyPI:

Upload first the wheels and then the sdist:

```
twine upload -s REPO_ROOT/release/installers/*.whl
twine upload -s REPO_ROOT/release/installers/scipy-1.x.y.tar.gz
```

If `gpg2` is preferred, then the above commands may also include `--sign-with gpg2`

Github Releases:

Use GUI on https://github.com/scipy/scipy/releases to create release and upload all release artifacts. At this stage, it is appropriate to push the tag and associate the new release (candidate) with this tag in the GUI. For example, `git push upstream v1.2.0rc1`, where `upstream` represents `scipy/scipy`. It is useful to check a previous release to determine exactly which artifacts should be included in the GUI upload process. Also, note that the release notes are not automatically

populated into the release description on GitHub, and some manual reformatting to markdown can be quite helpful to match the formatting of previous releases on the site. We generally do not include Issue and Pull Request lists in these GUI descriptions.

scipy.org:

Sources for the site are in https://github.com/scipy/scipy.org. Update the News section in `www/index.rst` and then do `make upload USERNAME=yourusername`. This is only for proper releases, not release candidates.

docs.scipy.org:

First build the scipy docs, by running `make dist` in `scipy/doc/`. Verify that they look OK, then upload them to the doc server with `make upload USERNAME=rgommers RELEASE=0.19.0`. Note that SSH access to the doc server is needed; ask @pv (server admin) or @rgommers (can upload) if you don't have that.

The sources for the website itself are maintained in https://github.com/scipy/docs.scipy.org/. Add the new SciPy version in the table of releases in `index.rst`. Push that commit, then do `make upload USERNAME=yourusername`. This is only for proper releases, not release candidates.

Wrapping up

Send an email announcing the release to the following mailing lists:

- scipy-dev
- numpy-discussion
- python-announce (not for beta/rc releases)

For beta and rc versions, ask people in the email to test (run the scipy tests and test against their own code) and report issues on Github or scipy-dev.

After the final release is done, port relevant changes to release notes, build scripts, author name mapping in `tools/authors.py` and any other changes that were only made on the maintenance branch to master.

<< Toolchain Roadmap

SciPy API Development Guide >>