LSST DESC Notes

LSST DESC Coding Guidelines

The Computing (CO, formally Computing Infrastructure or CI) Working Group, February, 2018; most recently updated in July, 2022
Contributions from: Phil Marshall, Mike Jarvis, Rachel Mandelbaum, Yao-Yuan Mao, Mike Wang, Seth Digel, Andy Connolly, Matthew R Becker, Johann Cohen-Tanugi, Heather Kelly, Francois Lanusse, Joanne Bogart

Contents

Introduction

We develop a lot of code collaboratively, in collaboration-wide (and often public) repositories, for a wide range of applications (including cosmology analysis, LSST Science Pipeline image processing and feature measurement, large scale data challenge simulation and so on).

The way in which we develop code should make us more nimble, not less, and so these guidelines are kept purposely minimal; however, we are also striving to produce high-quality code that enables a high level of scientific reproducibility, so we need at least *some* structure.

Ideally, you should find it easy to start contributing to an existing piece of LSST DESC code or to contribute new code and make it usable by anyone in the collaboration. Adopting some lightweight standards helps with that.

This short document gives some simple guidelines to follow when writing and reviewing code in the DESC. Specifically, we are talking about any piece of code that will be used in some capacity in a DESC paper. This is a dynamic document that the collaboration will develop over time. There is also a longer and more comprehensive document produced by the CI2 study group (*accessible only to DESC members*), some aspects of which have been superseded by this document and the DESC software policy.

In this document, we're only talking about DESC Tools and Analysis software (like DESCQA and CCL), but you might nonetheless find that getting in the habit of writing your personal experimental code to the same standards is helpful when contributing to collaboration code later.

Guidelines for Coders

Getting Started

- LSST DESC has a GitHub organization that all members should automatically have access to if they have added their GitHub ID to the membership database (*accessible only to DESC members*). Look for a repository (repo) corresponding to the project you are interested in. There are many useful tips for how to effectively use the DESC's GitHub organization, manage teams, etc. here (*accessible only to DESC members*).
- When starting a new GitHub repo in the LSSTDESC organization, the first decision is whether to make it "private" or "public". "Private" repos can be viewed only by the admins/creators until they add GitHub teams to the repo with read or write access; private repos are not accessible by anyone with the link. "Public" repos can be viewed by anyone on the web. In both cases, "Write" (or "Push") permissions can be

granted on a team-by-team basis by the repo admins. We encourage people to develop software in the open, in public repos. Open software is beneficial because it allows others to learn, improve, re-use, and contribute to the code. However, we understand that some members of the collaboration may be uncomfortable with this approach. At minimum, we strongly recommend that DESC code should be shared within the collaboration: this can be achieved by using a private repo that has the Members or Full Members team added with Read permissions. (Note that adding them all with Write permissions will auto-subscribe them as watchers of the repository, which requires them to take action to avoid getting notifications. For this reason, it is worth creating a GitHub team for the team of developers (dev team) and giving that team Write access.)

- For new projects consider using the LSSTDESC/gcr-catalogs repo as a starting point, which includes examples such as setting up GitHub Actions and a BSD license. You can also look at the LSSTDESC/Coord package which is a very lightweight package you might model yours after. (For instance, it includes instructions on how to get Sphinx documentation working with github.io and notes on how to make your package pip installable.)
- For those who are setting up a repo for a new code base, you may find it useful to first check out some tips on GitHub usage (*accessible only to DESC members*). There is also guidance about information to put in the README and licensing here (*accessible only to DESC members*).
- Most projects use issues or tickets to track code development. If you'd like to start contributing, you should look through these and pick one that you think you might like to tackle. Comment on the issue to propose the work you'd like to do. Feel free to ask questions or ask for advice about how to proceed.
- Or alternatively, maybe you have an idea for a feature or improvement that is not currently an issue. Go ahead and open a new issue.
- This is a social coding environment. People want to help you succeed. Keep talking.
- If you have technical trouble (e.g, working with git or setting up continuous integration), you can ask for help on the #desc-github-help or #desc-software-help Slack channels respectively.
- The DESC software policy describes DESC members' responsibilities to ensure robust collaborative software development.

Software Packaging and Licensing

Please see the DESC repository guidelines (*accessible only to DESC members)* in addition to notes below.

- All DESC software should carry a BSD-3-Clause license (as noted in the above-linked repository guidelines) when at all possible. If your package includes code from some other code base, please make sure it is attributed and licensed properly. For example, if you are using code licensed under GPL, then your package must be GPL as well.
- Try to take advantage of the existing tooling in your language for software packaging. For example, it is worth making a serious effort to ensure that all Python packages

are pip installable and use setuptools. Mature packages released publicly should be submitted to pypi or conda-forge. If a Python package has difficult binary dependencies, e.g. CCL with CLASS, then please consider making a conda-forge recipe to enable distribution via conda.
- For packages in C/C++ please use a Makefile, autotools, Scons, or cmake. Further, these kinds of packages typically have compiler-dependent settings that need to be determined at build time. Autotools and cmake will do this for you when used properly. If you do want a simple Makefile, make sure to respect the standard environment variables for compilers and flags such as CC, CFLAGS etc.
- Please think about the distinction between libraries and applications when deciding on dependencies and their version for your code. For example, if your Python package lists "numpy==1.12.4" in its installation requirements, then it won't be able to be easily co-installed with other software. Fully specified dependencies like this are only appropriate for applications meant to be installed and used in production by end users. In the vast majority of cases, we write libraries and our dependencies should reflect that. Use liberal constraints, like fixing only the major version "numpy >=1.11.0,<2", or ideally none at all! Pip has the operator "~=" which can be useful here.
- Reproducibility is not an excuse to fully fix your dependencies. If your package gives a different answer on a numpy minor version change, then you are probably doing something you shouldn't be. Further, end users can use tools like Docker and conda to make reproducible environments if they need them.

Coding Style

- Suggestions are PEP 8 for Python and one of Google, Sutter & Stroustrup, or LSSTDM for C++/C, but don't sweat it too much.
- Linters like flake8 can be very good for catching bugs as you write code.
- Consider using a formatter like black to automatically format how your code is split across lines and so on, saving you the trouble and ensuring consistency.
- Readability is paramount. **Code is read far more often than it is written.**
- Use clear, explanatory variable names. E.g. num_galaxies, not n2.
- Classes should typically be a singular noun. E.g. GalaxyList, not Galaxies. It usually helps make the code that uses that class more readable.
- Functions (and methods) should typically be a verb or verb phrase. Usually in the imperative mood. E.g. calculate_coefficients(), not coefficients().
- Methods that return booleans should typically use appropriate helping verbs. E.g. galaxy.has_spiral_arms() or image.is_contiguous(). In Python, these might be properties rather than methods if appropriate.
- Attributes (or properties) are typically nouns (e.g. galaxy.shape, or image.pixel_scale), but they can also be verb phrases if they return a boolean (e.g. image.is_contiguous) as mentioned above.
- [Python-specific] Don't hide significant calculations in a property. If it looks like galaxy.shape, then that value should either already be stored somehow or be fairly trivially calculable from existing stored values. If you need to run some complicated

algorithm to calculate the quantity, give the user a clue about that by making it a regular method and calling it e.g. calculate_shape() or measure_shape() so they know to store the answer rather than maybe use galaxy.shape multiple times in a bit of code.

- Don't use obtuse shorthand. E.g., you probably should write out scipy.special, not import it "as ssp" and then use ssp in the code. Other developers will probably not understand the short version, especially if it is only used a handful of times in a given file. (If it is all over the place in a given code base, then perhaps it is worth using the shorthand.)
- The style should be reasonably consistent within the project you're working on. It is not necessary that your style conforms to other styles being used in DESC (although if you don't know what style to use, looking at other people's code is a good way to get a feel for good style). More important is for a given code base to be internally fairly consistent.
- More experienced developers may often be able to provide style guidance as well.

Documentation

- Docstrings (or other similar user-oriented documentation) should fully describe everything the user will need to understand to use your code.
- In-code comments should describe to a later developer (e.g., yourself in 2 years) how your code works, where the algorithm comes from, etc.
- Good variable and function names can be self-documenting. E.g. num_galaxies rather than n2, as mentioned above.
- Document the code as you go. Don't plan to go back later to add documentation.
- In particular, writing a good docstring for every function as you code it will make web-based API documentation easy to generate automatically, later - and a README can help others in the collaboration start learning what you are doing so they can help out.
- It is often helpful to try a first pass at the documentation before you've even written any code. Often this will help crystallize aspects of the API design and help you write better code that makes sense for how a user will want to work with it.
- Similarly, implementing complicated math should often start with a long in-code comment explaining (to yourself!) how the math works that you will be trying to implement. It makes the subsequent coding much easier and serves as a record of the intended operation for future reference.
- If some code is implementing an equation in a paper, cite the paper (e.g., "This is equation (7) from Dodelson et al, 2002.") If it would be useful for the user to be aware of the reference, e.g. in cases where an algorithm is re-implemented and attribution needs to be given to its originators, make sure to also include this information in the doc string, not just the inline code comments.
- If you are using a snippet of code from some other source, make sure to leave a comment with the URL or some indication of where you got it. Further, make sure your use is consistent with the license of the source code you copied. If you have questions, ask!

- For Python, consider using Sphinx as the doc parser. For C++, doxygen is pretty standard. But we don't dictate either of these as any kind of requirement.
- Know your audience. There is always a trade-off between your individual coding speed (important for being able to react to changing circumstances and constraints) and the collaborative development of the code (important for overall development speed, and collective understanding of the code). The documentation that you will need for yourself when you look back in 6 weeks time may be different from the documentation that a new DESC member needs now to be able to get up to speed and start contributing. Being nimble as a group means reaching the right level of documentation at the right time.

Commits

- Ideally, each commit should be a single atomic change in functionality. The new code should:
    - include tests that the functionality change is correct;
    - include appropriate documentation of any new functionality;
    - pass the new tests as well as the rest of the test suite.
- This isn't always possible of course -- you may discover a bug in the code after you commit, or you might think of additional tests to add -- but it's helpful to have this as a goal. It greatly helps troubleshoot problems down the line.
- The commit message should briefly summarize the change you made. This facilitates finding a relevant commit later (e.g., to revert later or to cherry-pick onto a different branch) "Fix indexing bug in such-and-such module" is more helpful than just "Fix bug", etc.
- The git command "git add -p" is a great tool for selecting just some subset of the local changes that you want to commit. And "git rebase -i" will let you combine new changes with old commits if you discover that you missed something related to a particular change. Mike Jarvis discussed these and other useful git commands in a (non-recorded) hack week presentation (*accessible only to DESC members*).
- Another exception to the "atomic commit" ideal is if you don't know how to fix a bug. It is perfectly reasonable to commit something with the message "Tried to implement XXX. Fails YYY test." And then ping some other developer in the project and ask them to help you figure out how to fix it.

Tests

- Some form of testing should always be done. However, not all aspects of the list below are always relevant. Use common sense to determine which items are important to ensure correctness of your code. (But be aware that your code reviewer may ask for more - see sections below on code review.)
- Unit tests are good for testing small bits of code to check that each function does what it is supposed to do.
- Regression tests may be helpful to make sure existing functionality is preserved in the future through possible code refactoring.
- User interface tests check that the code behaves sensibly if the user does something they shouldn't do (e.g., gives bad inputs, forgets a parameter, etc.)

- Functional tests check that the code produces correct outputs for a variety of inputs.
- If you can test some fancy, efficient algorithm against a more obviously correct (but slower) algorithm, that's a great test to include.
- If there are specific special cases where the answer can be known analytically or via some other means, these are good functional tests as well.
- Think about edge cases. What might cause your code to fail? You should add tests that these edge cases work correctly.
- Integration tests check that your class/function/etc. works correctly with other parts of the overall code base.
- All of the above tests should be part of a test suite that is run regularly, ideally via a continuous integration [CI] system like GitHub Actions.
- Code validation, where you run the new code on a large set of data or perform a long calculation, is often important during development. The results should be made available to the code reviewer, but you do not necessarily want to include this in your continuous integration.
- If some kind of code validation was helpful, think whether a smaller/faster version of it could reasonably be included in your CI test suite.
- Don't forget demos and tutorial notebooks. These are good integration tests that show how various parts of your code fit together (and can also be included in the CI test suite!).
- See also Mike Jarvis's DE School lesson on unit tests.

Contributing Your Work

- Beyond the initial setup phase of a project, contributions to collaborative code bases should be made by pull request, so they can be reviewed by your collaborators.
- It is possible in GitHub to set up repositories with "branch protection", which enforces rules about not pushing directly to certain branches. Protecting the main branch ensures that all code integration to main occurs via pull request. This can be a useful guard against mistakes especially in repositories that include many people with write privileges. Please note that some older repositories might have a "master" rather than a main branch due to a change in nomenclature.
- A typical workflow for contributing work to a code base is as follows:
    - Make a branch for a new feature to be worked on. This typically corresponds to a single GitHub issue.
    - Develop the feature (typically as a single developer, but occasionally working as a small team).
    - Once you are ready for code review, submit a pull request to merge this branch into the main (or other) branch. Pull requests should ideally be very small, of order 500 lines of changed code or less. If your addition is too big, break it up into smaller chunks.
    - One or more people will normally review the code (see below). Each software development team will likely have a rule about how many reviewers are needed; a model that ensures adequate scrutiny while not being too costly in terms of time is to require one reviewer for bug fixes or simple feature implementation, and 2 for major feature implementation or API changes.

    - The code review will often involve requests for you to make some changes to the code. Make those changes and iterate until the code reviewer(s) are happy with them.
    - The code is then merged into the main branch -- either by you or by the lead developer for the code project.
    - The branch on which the code was developed should typically be deleted once the code has been merged to main
- Different groups may have different preferences for aspects of this procedure. If you're not sure what to do, just ask what the appropriate next step is.
- If you're the only developer of a particular bit of code, the above workflow might not make sense for you. However, still consider asking for code reviews occasionally. E.g. once the code is ready for use in a DESC analysis or paper. Even for software with just a single developer, the GitHub PR interface is handy for checking over changes before merging to main.

Guidelines for Code Reviewers

Who should review?

- Often a project team has one or more lead developers. At least one of these people should probably review every PR within a project.
- Less experienced developers should try to make a habit of reviewing code regularly. Reading and evaluating other code is a great way to learn better coding practices.
- Occasionally, it can be useful to ask DESC members outside the project team to review some code -- perhaps before a big software release. Ask on the Slack #desc-software-help channel if you are interested.
- In the LSST DESC, it is standard practice that any collaboration member can request code review from any of their fellow collaboration members. The understanding is that while the requested reviewer will try their best, they may not be able to do the review as requested due to time constraints.
- Most reviews should have a fast turnaround time. Only agree to review if you can do so in a timely manner: if you cannot, please reply promptly to the PR thread suggesting (and @ mentioning) some alternative reviewers.
- As a reviewer, if you realize that you don't have the appropriate expertise to review a particular section of the code, it is very helpful to @ mention someone who you think might. (e.g. "I don't quite follow the math here. @jdeveloper could you take a look and make sure this seems ok?") They can then just review that portion of the code, rather than the whole PR.

Questions to ask

- Does the code do what it is apparently intended to do?
- Is there a test (or example) of the typical way that users will want to use this code?

- For complicated algorithms, are there tests that check correctness, say in a case where the answer is known analytically, or comparing them to a more direct (but slower) calculation?
- Are there tests of possibly problematic edge cases that might cause problems (e.g., RA values crossing from <360 to >0, nans in the input, singular matrices, non-convergence of an algorithm, etc.)?
- Is the user documentation clear?
- Is there suitable guidance for any non-obvious input parameters that the user needs to set?
- For complicated algorithms, are there suitable in-code comments to explain to a later developer how the code is designed to work? [Note: the reviewer doesn't necessarily need to understand all details of an algorithm. A comment referencing a paper or web page with more details is often sufficient.]
- Are classes, variables and functions well named (i.e., not confusing as to their intended function within the code.
- Are there any obvious inefficiencies where simple code redesign could help? (e.g., memory allocations in an inner loop that could be pulled outside, for loops in Python that could switch to list comprehension, the same file being opened multiple times, etc.) [Note: don't recommend code changes that make code less readable in order to be more efficient. Unless this is known to be a tall pole worthy of extensive optimization, readability matters more than efficiency!]
- Are there places in the code where the style differs significantly from other code in the same file or project?
- Are there places where the style causes the code to be difficult to read or understand?
- Do classes, variables, and function names follow the same style as others in the file/project? E.g., CamelCase, mixedCase, lowercase_with_underscores, etc.
- Are there repeated blocks of code that would benefit from being pulled out into a separate function? A good rule of thumb is that repeated blocks of code should only be combined if
    - They have been repeated at least 2-3 times.
    - The purpose of the repeated sections of code is the same.

What happens after the review?

- Once the PR is approved, it is the developer's job to merge the PR. This is to enable them to take care of any modifications left optional by the reviewer, or to first issue any remaining problems before merging (if they want to do things in this order).
- Occasionally a repo admin may merge an approved PR, if they are trying to get to a new release, for example.
- If changes are requested, you can now work on them with your new collaborator, your code reviewer. Generally speaking, repo admins and experienced contributors will work together to resolve disputes.
- A common pattern when you think you are ready to merge a PR is to state a timeline when you plan to merge. (e.g. "will merge this tomorrow afternoon unless I hear otherwise") This allows people who had wanted to review (or may currently be in the

process of reviewing) time to push back on this plan if they want more time to finish their review.

Comprehensive Code Reviews

A typical code review is on a single pull request, covering a relatively modest set of changes covering more or less a single topic, which are requested to be merged into the main branch. Occasionally though, one may want to get someone (usually external to the development team) to do a comprehensive code review of the whole API and overall design of the code already on the main branch. This should be rare -- at most once per project probably, since it is quite a lot of work. **Ideally, an experienced developer with a lot of code design experience will be involved in code design and reviews all along, so that this process will not be needed.**

- The focus of such a review should mostly be on high-level design, both in terms of the user-directed API choices, and also how aspects of the implementation will impact the efficiency and long-term maintainability. Comments on details are fine, but should not be the primary focus.
- A good choice for an external reviewer may be someone who plans to be a heavy user of the code, so will have a good perspective about the API decisions.
- For some projects, it may be helpful to get an expert developer who has some experience with the efficiency implications of the design to review the data structures and key algorithms, especially how the implementation will likely play out in typical computer architectures where the code will be run.
- To be most effective, an API review should be done well in advance of a "1.0" release, when the API is still being molded to some extent, but where there are enough use cases already fleshed out that the code is beginning to be fairly mature.
- GitHub doesn't enable line comments on the main repository, but one can trick it into giving you line comments with the procedure in the "Comprehensive Code Review" appendix (cf. e.g. the CCL review).

Further Reading

- Practical Lessons in Peer Code Review - Salsita Software blog
- Better Learning through Code Reviews - Capgemini Engineering blog
- Why code reviews matter (and actually save time!) - Atlassian Agile Coach
- 7 ways to uplevel your code review skills - Asana blog
- See also Mike Jarvis's DE School lesson on code reviews.

Guidelines for Coding Teams

Things to think about

- Continuous code review makes final review before release easy, and is really necessary for coding development carried out by teams.

- Good documentation is not only important for enabling the existing set of users and developers to develop the code base and do their science, but will also make it easy for new people to contribute to your project.
- Try to be encouraging of new contributors. Helping them write good code will likely be beneficial to the team in the long run. (Plus it's just polite.)
- Using a good workflow engine (e.g. Pegasus or Parsl) can help streamline the development. For instance, wlpipe and daxpipe development with Pegasus mostly involve writing configuration files, rather than code, which significantly increases the efficiency of the project.
- More generally, try to leverage other existing code as much as possible rather than develop everything yourselves.

Versioning

- For the sake of simplifying communication and ensuring reproducibility of results, any code that is used by people outside of the development team, or even used for science results only by the dev team, should be versioned. This can be achieved with the GitHub "release" mechanism, which is a wrapper for git tag.
- Use M.m.r versioning, where M is the major version, m is the minor version, and r is the revision or patch. This is called Semantic Versioning.
- Normally, the initial development of a piece of code uses 0.x versions. During this phase, it is acceptable for APIs to change between versions in backwards-incompatible ways. Normally the user base during this period includes just the developers and some select test users who are willing to deal with these kinds of changes.
- Once you have code that you think is ready for a wider user base, you should normally bump up to version 1.0. At this point the API should be kept more stable and backwards compatible.
- Minor version updates (e.g. 1.2.3 -> 1.3.0) typically include new functionality, while as much as possible trying not to break code that uses previous versions. If you'd like to remove some functions, you should have them emit deprecation warnings, rather than outright remove them.
- Major version updates (e.g. 1.3.0 -> 2.0.0) are where APIs are allowed to be backwards incompatible. As much as possible, this should just be removing the deprecated code, rather than springing a new change on your users. But sometimes, that isn't possible, and you just need to document the API change you made.
- Patch version updates (e.g. 2.0.0 -> 2.0.1) should be reserved for bug fixes. All the APIs should be identical; you've just fixed some bug in the previous version.
- Consider using Versioneer to help keep track of versions automatically. It makes sure the version reported by your code matches the tagged release that appears on GitHub. It can also give a version for runs off of main (not recommended!) when necessary.

Appendix

Other References

Mike Jarvis discussed an early version of this document at an LSST DESC Hack/Sprint Week *(link accessible to DESC members)* in December, 2017. See the video of that presentation.

Mike Jarvis presented LSST DESC Dark Energy School lessons on unit tests in July, 2017 and code reviews in February, 2018. Videos of these and other software-related Dark Energy School lessons are available here.

The DESC CI2 study group report has some early recommendations (circa 2016) for DESC software development practices - largely superseded by these guidelines and the DESC software policy, but available for reference here *(link accessible to DESC members)*

Comprehensive code review process

Below are the steps to force GitHub to allow you to carry out a comprehensive code review of an entire code base. In the instructions below, all branch names are in italics.

- Go back to the very first commit in the repo. Hopefully just with something negligible like a Read.me file.
- Make a branch off of that called *empty*.
- Copy over from the *main* branch anything that you don't want to review. E.g. devel directory, installation instructions, license, etc. Whatever you don't want to clutter the code lines you will be reviewing. The git syntax for this is, e.g., `git checkout main devel` to copy over the whole devel directory. Commit these files.
- Go back to the current *main* and make a branch off of that called *review*.
- Merge from *empty* into *review*. (This should be a zero line merge, but is necessary to get the git history right for those non-code files.)
- Make a pull request from *review* into *empty* in GitHub.
- Add line comments in GitHub about the current version of the code.
- People can make changes directly into the *review* branch responding to the code review comments. Or do things in separate PRs if preferred. (CCL took the latter approach.)
- Once you are done, any changes on the *review* branch can be merged into *main*. Just switch the target branch of the PR on GitHub to *main* rather than *empty*, and you'll be able to see the new changes. This might also let you catch bugs in the new changed code. Then merge into *main.*