Program Quality and Code Reviews

IS597PR - John Weible

Factors to review for program quality

Program Behavior, Reliability, and Usability:

- ☐ Are there clear end-user instructions for how to operate the program? Should there be?
- ☐ Correctness with typical inputs?
- ☐ Correctness with rare, edge-case, or missing inputs?
- ☐ Does it provide a clear interface, outputs, and error feedback to users?
- ☐ Look for assumptions that may not hold true and ways a user or another programmer might do something unexpected that would cause a problem.
- ☐ Are any file directories hard coded into the program where they shouldn't be? (This is most common with explicit full paths that should be relative instead.)
- ☐ Good handling of exceptions?
- ☐ What happens if input data file(s) are missing or have different format than assumed?
- ☐ Does it have thorough test coverage of all the above?
- ☐ Are there any "tainted" (untrustable external input) data that get **executed**? This could occur in an eval() call, a system or shell command, an unsafely constructed SQL query, or similar. (These are all *code injections* that create *security vulnerabilities*, avoid them properly)
- ☐ Robustness under attack? (Too many possible things to list here, be careful.)

Maintainability & Readability:

- ☐ Is the code being managed properly in a stable VCS with remote backups? Are commits done early and often, with meaningful summary messages?
- ☐ Are symbol names (variables, classes, files, etc.) clear and helpful? Even if they are descriptive, check for things like incorrectly plural class names, or plural function names that return single items or vice-versa. Look also for spelling mistakes and that word tenses are consistent.
- ☐ Is the source code documented well? The purpose and behaviors of modules, classes, methods, and all major data structures should be explained. Use Docstrings and annotations for that. Also, within the code, all confusing or complex statements or algorithms should be clarified with in-line comments.
- ☐ As code gets modified, make sure the documentation gets updated to match! Wrong or obsolete type annotations and documentation is sometimes even worse than none.
- ☐ Citations/References: Does the code clearly show who wrote it, and does it contain references to the sources of any non-obvious inspirations or borrowed code fragments that have been used (and thus not entirely written by the primary author)?
- ☐ Is the code formatted/styled consistently? Especially in large team projects, there are usually established style guidelines, and correct code may be rejected solely for format/style reasons. In our case, pay attention to the style hints PyCharm provides (based on the PEP standards) and use the menu Code … cleanup or formatting features if needed.

Modularity & Reusability (refactor code if needed, to improve these):

- D.R.Y. = Don't Repeat Yourself. Look for nearly identical sections of code and/or variables that are duplicated. Refactor the code to remove this. The duplicates may even be in different files – unless it's there specifically to show slightly different ways to do the same thing, they should be refactored and merged somehow.
- Is the program modularized well? That means organized into well-designed functions and possibly multiple modules and/or classes, such that each part can be described, understood, and tested fairly independently of the others.
- Are best practices for scoping rules for variables (data objects) being followed? Correct all problems such as using "global" variables or more subtle situations where any non-parameters are being accessed or modified within functions. These mistakes cause unpredictable side-effects and bugs, making code unreliable and even make it impossible for unit tests go guarantee correctness.
- For a module or class, does the set of functions operate in an internally-consistent way? Is it consistent with typical APIs of other standard modules? If it's unnecessarily inconsistent and non-intuitive then all programmers (even the original author) will have to work harder.
- Is the code written in an overly-specific way that prevents it from being re-usable? Functions and classes should be re-usable within the application, other very similar applications, and potentially in much broader situations. A common mistake is hard-coding specific values or assumptions into code when a more general approach works as well or better.
- If you have some very similar specific functions, consider writing one more general or abstract function that can be wrapped by (and thus reused) to implement the more specific ones, or vice-versa. Also, this idea applies to class design, where an abstraction of the class may be more broadly useful. Use top-down and bottom-up design…

Python-Specific Common Traps to Avoid:

- Do not use the "global" keyword.
- Any time you mean to make a "copy" of a list, dictionary, array, DataFrame, or other complex data structure, make sure you know whether you're actually copying a pointer only, a shallow copy, or a complete "deep" copy of the potentially nested object.
- Never set a default value on a function parameter that is a literal mutable object. These get created when the function is parsed and defined, NOT when executed like you may assume. Instead, set the default to None and replace with the desired mutable object inside the function.
- Do not ignore, disable, or suppress warnings. People often make this mistake with Pandas. Instead, FIX the code so the warnings don't occur at all.
- Do not write production-oriented code in Jupyter. Generally, don't even use Jupyter for "development" work – you cannot properly debug or structure significant programs in a notebook. Jupyter is good for *PRESENTING* of samples of code+markup+data+plots. Use Python modules for everything else. Related to this, your custom functions and tests should ideally be in modules and *IMPORT* the modules into a notebook to call the functions. The D.R.Y. principle applies to Jupyter too!

Efficiency (refactor if needed):

- ☐ Is the code written efficiently? If you see much duplication or clumsy organization, these are indicators that refactoring is likely needed.
- ☐ Does the code perform efficiently? If it is fast enough to not matter in production use, don't worry. Don't optimize without reason! Profile to measure and find the bottlenecks first, before investing in optimization or complexity analysis, and focus on the areas of maximum benefit.

Samples of other Professional & Production-Quality Considerations [not required in 597PR]:

- ☐ Consider whether this is a commercial work, free open source, etc. Make sure the licenses of all the packages and other components that your code is dependent on have compatible licenses, and that attributions are included as required.
- ☐ Does the program have detailed or customizable output logging to help troubleshoot problems (bugs, bad data, concerns security concerns) after they occur?
- ☐ Is a good bug & issue tracking system being used? If so, are VCS commits properly linked to the tickets? This is just one aspect of many project and service management concerns.
- ☐ Is the code designed for robustness and resiliency with hardware failures? Typically, this means adding significant error-checking throughout and verifying each computation stage completed properly before the system proceeds to subsequent stages.
- ☐ Is the systems architecture designed for resiliency? For reasonable cost efficiency?
- ☐ If the code could be beneficially run on multiple platforms, is it designed well for that? Look for use of packages that aren't multi-OS compatible.

Code Reviews

Code Reviews are an excellent way for both the reviewer and the code author to learn and grow in professional coding skills.

Rules for WRITING code reviews:

1. Include some feedback on things done well.
2. Always be professional and courteous giving written or verbal feedback. It is NOT a competition, nor contest of egos.
3. Focus on the "Factors to review for Program Quality" above, but there can be other issues besides.
4. To do a good review, you will have to inspect the code in detail and run it yourself, possibly even with a stepping debugger to understand it well.
5. Prioritize and filter your review. We're not trying to achieve perfection. Focus on the most-significant things you identify for improvement rather than looking for every possible thing to list.
6. Remember everyone has different levels and types of knowledge and experience.
7. Provide clear, concrete, and constructive feedback.
8. When giving criticism on things for improvement, BE VERY SPECIFIC. It is far more instructive to give examples or suggestions on HOW to improve rather than giving vague statements like "X needs to be improved" or "X isn't designed well".
9. You may notice bugs and provide the code fragments to fix them. Or at least provide example inputs that will cause bug for replication. That's what pros do.
10. Remember when suggesting an improvement, there may be a good reason you've overlooked why it was done as is (but which should be documented).
11. Always include feedback on some things done well! [Yes, I listed this twice on purpose]

Rules for RECEIVING code reviews:

1. Don't take feedback personally.
2. Don't be discouraged either -- think of it as more opportunities to grow and learn.
3. Remember that EVERYONE writes code that could be improved. Perfection is a myth.
4. Even if you received a suggestion to implement, some alternative approach may be even better still. Or if you think the code is better as it is already, make sure you are clear about why – discuss it.
5. If something is unclear in the review, ask the reviewer about it.
6. Use the review to improve your work!