review-tipps

January 1, 2021

1 Code Review - a couple of notes

Much like scientific peer review, code review has established itself in industry, science and the open source community as a tool to facilitate "clean" and consistent code, distribute knowledge within teams and help avoid errors in production and publications [1].

Since the result of scientific work is as much code as it is manuscript, it deserves as much care.

1.1 Review code

As a reviewer, assume good intentions and good reasons. Code exists to solve a specific problem, and an author usually has good reasons to writing a piece of software the way he does (didn't know how to write better is a good reason, too!). In these reviews, try to look at the text, equations and graphs first, and then - if there is time - at the code. Evaluate the code from these perspectives:

- does the code meet the requirements (eg. produce the correct output)
- is the code architecturally sound (eg. is not a sprawling mess of spaghetti code)
- is the code futureproof (eg. can be parameterized to solve a similar code)
- does the code use correct ("pythonic") coding idioms (eg. list comprehension, list destructuring etc)
- is the code resusable (eg. usable operations abstracted into functions or classes)
- does the code rely on external libraries where appropriate (eg. numpy, pandas, obspy)

Then (slightly adapted from thoughtbots code review guide)

- Communicate which ideas you feel strongly about and those you don't.
- Identify ways to simplify the code while still solving the problem.
- If discussions turn too philosophical or academic, move the discussion offline to a regular Friday afternoon technique discussion. In the meantime, let the author make the final decision on alternative implementations.
- Offer alternative implementations, but assume the author already considered them. ("What do you think about using a custom validator here?")
- Seek to understand the author's perspective.
- Remember that you are here to provide feedback, not to be a gatekeeper.
- Accept that many programming decisions are opinions. Discuss tradeoffs, which you prefer, and reach a resolution quickly.
- Ask good questions; don't make demands. ("What do you think about naming this :user_id?")
- Good questions avoid judgment and avoid assumptions about the author's perspective.
- Ask for clarification. ("I didn't understand. Can you clarify?")

- Avoid selective ownership of code. (“mine”, “not mine”, “yours”)
- Avoid using terms that could be seen as referring to personal traits. (“dumb”, “stupid”). Assume everyone is intelligent and well-meaning.
- Be explicit. Remember people don’t always understand your intentions online.
- Be humble. (“I’m not sure - let’s look it up.”)
- Don’t use hyperbole. (“always”, “never”, “endlessly”, “nothing”)
- Don’t use sarcasm.
- Keep it real. If emoji, animated gifs, or humor aren’t you, don’t force them. If they are, use them with aplomb.
- Talk synchronously (e.g. chat, screensharing, in person) if there are too many “I didn’t understand” or “Alternative solution:” comments. Post a follow-up comment summarizing the discussion.

1.2 Having Your Code Reviewed

Remember that reviews are meant to help, not to judge, and bad good is better than no code. Then

- Be grateful for the reviewer’s suggestions. (“Good call. I’ll make that change.”)
- Don’t take it personally. The review is of the code, not you.
- Keeping the previous point in mind: assume the best intention from the reviewer’s comments.
- Explain why the code exists. (“It’s like that because of these reasons. Would it be more clear if I rename this class/file/method/variable?”)
- Seek to understand the reviewer’s perspective.
- Try to respond to every comment.
- Final editorial control rests with you

1.3 Some notes on Readability

Code is written for humans, not computers (the compiler will translate for the computer, don’t worry), so treat your code a you would your manuscripts. Make it painfully clear to read, and trivial to understand. Good code is boring and read many time more than written, and the reader doesn’t know what you know now. And that reader will probably be you, a few weeks, months or years from now. Be kind to future-you .

It clearly helps to make your code look like any other code, by using pythonic idioms and adhering to an established style guide.

Comments are like the manual to your smartphone. You won’t read them. And if you have to read the comments, the code is propably not well written.

> Comments. A delicate matter, requiring taste and judgement. I tend to err on the side of eliminating comments, for several reasons. First, if the code is clear, and uses good type names and variable names, it should explain itself. Second, comments aren’t checked by the compiler, so there is no guarantee that they’re right, especially after the code is modified. A misleading comment can be very confusing. Third, the issue of typography: comments clutter code.
>
> Rob Pike, “Notes on Programming in C”

When you do have to write comments (or jupyter text cells), explain the concepts, methods and reasoning behind the code, not the implementation.

1.4 Common mistakes w/ Jupyter Notebooks

Jupyter has it own sets of unique traps and pitfalls, so be aware of those while reviewing Notebooks.

- *Out of Order Execution and Hidden State*: it is trivial to get to an inconsistent state in jupyter notebooks, because cells are not neccessarily run from top to bottom. So ALWAYS ALWAYS run "Kernel -> Restart & Run All" before submitting a notebook.
- *Spaghetti code*: Resist the tendency to write non-modular spaghetti code. Write functions, classes even imporatable modules where appropriate. Jupyter is data exploration, application and illustration of code, not about implementing complex codes or algorithms

See also:

[1] Christian Bird Alberto Bacchelli: Expectations, Outcomes, and Challenges of Modern Code Review, https://www.microsoft.com/en-us/research/publication/expectations-outcomes-and-challenges-of-modern-code-review/

[ ]: