Robot Nitpicks

Never send a human to do a machine's job

Moshe Zadka – https://cobordism.com

PyBay 2018

The Easiest Most Useless Code Review

- Mechanical
- Annoying

The easiest, most useless, code review is the one that goes line by line, nearly context free, and comments on style or best practices violations. Examples are "there is a missing space before the +" and "this expression is too deeply nested".

The more self-aware variant of this will precede the comment with a "nit" or "nitpick". Nitpicks are not useful, but are easy to give.

There is no need for humans to enforce that. Computers can check for spaces. Computers can check for nesting levels of expressions. Computers can identify the :code:‘for/append‘ pattern and suggest a list comprehension.

Why?

- Speed bumps
- Feeling useful

So why do people do that? Some of the more toxic variety is the "speed bump" motivation. This developer wants, for whatever reason, to delay the PR. Maybe they think the underlying feature is a bad idea. Maybe they had a fight with the submitting developer. Regardless, this is a pretty toxic motivation, and we are better off being charitable.

The more common motivation is feeling useful. Often new developers, or ones with little experience, or ones with imposter syndrome, want to make sure they count as "being useful". The mindless make-work of "point out all nits" is visible, and is guaranteed to finish successfully in a well-known amount of time. While this motivation is not toxic per-se, it is definitely not a good sign!

Why not?

- Team solidarity
- Mentor and help

It is important to remove the ability to use the nitpick as a crutch. This will force the developer to recognize they are in over their head, and need help. Good teams help each other. The team recognition that someone needs help is valuable. They can be paired up with a mentor, or use pair programming or figure out other ways to get them up to speed.

Nits

A taxonomy.

Nits come in all shapes and sizes. Since this is a talk about how to get robots to do them, it is important to be clear them. What we have not clarified properly to ourselves is very hard to teach a program to do. So the discussion, nitpicky as it seems, is critical to finding solutions.

Coding Style

Cite chapter and verse.

The classic nitpick is the coding style nitpick. It does not matter where the style comes from – official style, one of the many guides published by companies or even a house style. This nitpick enforces "Section 3, bullet 2" of the coding style guidelines.

Example – Line length

- PEP-8 says 80
- Black says 88
- Some say 100, 120...
- Literally just counting

The ur-example is the line length. Presumably the team has already decided on a specific max length. Regardless of which length, this comment is literally just counting characters. Probably not manually, either – just letting an editor tool count them. It adds nothing useful to the discussion.

Consistency

Compare two places, ask why.

Why did you choose to address X here this way, but then address X there this other way? This is inconsistent. Especially annoying if the places are far apart, and the need for consistency is somewhat irrelevant.

Example – Literals or constructors

- `{}` vs `dict()`
- Paint the bikeshed fluourescent green

Either there is a specific house style asking for one or the other, or there is not. If there is, see previous example. In the more common case where there is none, asking for consistency is not useful. Maybe in one place `dict()` is more useful (for example, it is supposed to look symmetric to something else) whereas in another place, the slight speed-up from using `{}` is worth it. If there is a specific overriding concern with one or the other, raise it. Otherwise, do not worry.

Best Practices

Plethora to pick from!

The nice thing about standards is that there are so many to choose from. One person's "best practice" is another's "obsolete methodology". However, pointing out best practices, regardless of the source of the list thereof, is still a fairly mechanical comment. After all, best practices are usually formulated in a way that is easy to apply: use composition rather than inheritance, avoid use of magic methods and so on.

Local Practices

Especially annoying to new people.

"Please remember that all our variable names must be more than three letters, and that all imports only import modules." This is a great way to make new people feel less welcome, by making sure that, at the moment they think they are "mostly done", they find they have a few more hours of make-work of complying with standards they never knew existed! Unfortunately, this is actually pretty realistic.

Example – No print

...or fluourescent pink.

"We do not use print. Please use our logging library for all developer or operations facing output. Remember to never send it formatted strings, but rather a message and details, so that it can be available to automated analysis."

Humans are inconsistent

Easy to miss exceptions!

Humans, try as they might, are not perfectly consistent. In a long pull request, or several pull requests, it is easy to miss something. A rule like "double quotes for user-facing strings, single quotes for internal ones" is unbearably easy to violate in one out of a hundred times. When both the one writing the pull request, and the one reviewing it, are humans, eventually exceptions to all "always do this" rules will find their way into the code base.

Humans get tired

"I've done most files in this PR."

Worse, humans get *worse* at this. At 5pm, after a long day of reviewing PRs, it is easy to get distracted. It is easy to skip the last couple of files in the diff, and to hit the big approve button.

Delay tactics

"I'm going to comment on every single line."

Humans are complicated. Some of them are not very nice, sometimes. Some nitpicks are used as a delay tactic – for whatever reason, the commenter does not want the change to land. Rather than coming out and admitting it, they will haggle over each space placement, every variable name, every choice of formatting.

One upping

"I'm aware of the obscure rules in our local style guide, as well as historical versions of PEP-8"

Some of them just like to feel important. They will point out that a combination of section 2 subsection 3 and section 5 subsection 1 means that this code does not conform to the style guide, even though it appears to. They will point out the actual implication of obscure, unclear sentences in the style guide. While no doubt a fun intellectual exercise, this does nothing to advance actual business goals.

Time pressure

"QA needs this yesterday, I don't have time to fix all of that at the last minute!"

Reactions to nitpicks are just as bad. Most of us work under time pressure. There are external deadlines, internal deadlines, managers, project managers and end users to appease. After having allocated enough time to write the PR, it is annoying to have to wait for a review, and then fix a dozen niggling details that, let's be honest, there was no way you could predict beforehand.

Annoyance

"You're just doing this to me because..."

It is also easy to feel frustrated with our colleagues. The fundamental attribution fallacy means it is easy to jump to conclusion like "they are just toxic people who are out to get me" rather than "they were too tired to make comments of substance, and tried to feel like they were moving things along".

Configuring linters

Make the linter reflect current consensus.

Configuring a linter should not be a soapbox. Reflect current consensus on the team. Note that is *not* the same as reflecting *current status* of code. But if people think something is good, the linter should reflect it.

Linter plugin

If the linter doesn't do something, implement it.

Modern Python linters have the ability to execute plugins. Flake8 plugins are usually the easiest, with an entrypoint-based API to set them up. This means easy integration with :code:‘tox‘, where often the only thing needed for another plugin is to add it to the list of dependencies.

This has been so popular, that many "linters" do not even bother with a command line, and declare their public interfce to be flake8 plugins. As a good example for such a plugin, is :code:‘ebb-lint‘. It also shows the modern approach to linting, with fewer knobs. The only way to change settings or to disable checks is to fork and name it something else. This is exactly the right amount of difficulty needed to be a useful barrier to entry.

If your local team needs a different set of practices that badly, have a name-of-team-lint package published to either PyPI or a local repository that encodes those practices.

Ad-hoc programs

Just write a little script to check for copyright notices.

Sometimes the easiest thing is to just write a new utility. Especially for things that do not translate well to linting, such as overall structural issues (every file in :code:‘resources‘ should correspond to a file in :code:‘models‘), it is often easier to implement the logic in dedicated Python programs. Python programming is fun! Again, upload the package to PyPI or a private repository.

CI intergration

Automatic CI failure blocking merge on Lint errors.

The easiest way to let robots ”code review” is to use Continuous Integration. Hopefully, there is already an existing set-up, if only to run some tests on the code. If not, it is easy to get it for cheap or even free – Travis-CI, CircleCI, GitLab CI, Buildbot or even Jenkins, there are plenty of options. The trick is to set-up ”pre-merge” builds, and to enforce, in whatever development flow system is being used, a ”no merge on red” policy. Adding the linting run to this should be a straightforward exercise.

Supporting legacy with diff

Only run lint on files in diff (or only fail on lint errors in diff)

One way to avoid needing to revisit history when adding a new rule is to only run the linter on diffs. This does require some co-operation between the CI system and the linter, but usually can be achieved in a decent way. This is especially true if the compromise position of ”if you touch it, you buy it” is adopted: when touching a file, bring all of it up to date. This requires files to be small, which is probably advisable anyway.

Supporting legacy with white lists

Keep list of current failures, ignore ”known” failures.

Another way of supporting legacy is to have a ”white list” of failure messages (for reliablity, best to keep it kind of message, file and number of those, and avoid precise line numbers). Care must be taken to make it easy to *remove* failures from the white list when possible, but make it slightly harder to add failures to the white-list. This can be as simple as a commit-approval-bot that auto-approves any removal from the white list, but asks one of a specific list to approve addition to it.

Simple rule

If CI doesn't care, the humans don't care!

Regardless of how you implement lint, "good enough is good enough". Have a simple rule: never nitpick if the CI is green (if the CI is not green, the only nitpick needs to be "the CI is not green"). If a nit is important enough, and violated often enough, add it to the linter. Ideally, changes to linter configuration are co-versioned with the code, and discussed using the same flow that code changes are discussed: pull requests, technical designs, Jira tickets or any other methodology the team decided to use to evolve the code base.

Summary

- Humans gonna human
- Use that, don't fight that

Humans are good at *deciding*. Humans are good at writing tools. Humans are terrible at being consistent, avoiding mistakes, or not having emotions. If you work with the strengths humans have, and allow for our human frailty, you will have a happier team full of happier human people.

If you ever hire Vulcans or robots, please ignore everything I have said. I only know how to work well with humans.