

# another pair of eyes: reviewing code well

adam dangoor

twitter: [adamdangoor](#)  
github: [adamtheturtle](#)

Hello everyone, I'm Adam. And I'm going to talk about code review.

# why?

- 
- 
- 

but why code review?

# why?

- great reviews have helped



“Your code sucks, and I hate you”  
- biog post by jml

Well, the inspiration for this talk is a bunch of people who I used to work with who mostly are contributors to Twisted.

I got to experience the really rigorous approach to review which is also present in the Twisted project and I have become a better programmer thanks to that.

# why?

- great reviews have helped
- bad reviews have harmed
- 
- 

And I've also had some really bad code review experiences. The kind that make you want to not go into work the next day.

I never want to make people feel like that, but when your role is to judge someone else's work it is easy to upset them if you're not careful.

# why?

- great reviews have helped
- bad reviews have harmed
- hardly talked about
- 

I guess that some of you have experience with code review. It is becoming an almost fundamental part of the software engineering process.

But, perhaps because of that, code review is sometimes seen as boring grunt work, and not a skill to be honed.

# why?

- great reviews have helped
- bad reviews have harmed
- hardly talked about
- 

I spend almost a third as much time reviewing code as I do writing it.

Yet when I go to programming forums or Twitter, or read blog posts or studies on how we can improve our skills, I see very little about review.

# why?

- great reviews have helped
- bad reviews have harmed
- hardly talked about
- little to no training

When a developer starts out, they are often paired with a mentor or given easy-fix bugs to train on.

But we are expected to know how to review code without ever having been taught.

# the plan

- what code review is
- how I review code
- some common pitfalls
- tools which can help
- becoming a better programmer

So today I'm going to talk about

what code review is

how I review code

some common pitfalls

tools which can help

and how code review can make you a better programmer



# what is code review?

- 
- 
- 

So what is code review?

It's a practice where someone will read over someone else's code and look for things which can be improved.

# what is code review?

- catch bugs early - prevention is cheaper than cure
- 
- 

We do this before merging code because we want to catch bugs as early as possible.

# what is code review?

- catch bugs early - prevention is cheaper than cure
- 
- 

Yes, we have automated tests, but code review can sometimes be the final gate keeper before code hits the real world.

# what is code review?

- catch bugs early - prevention is cheaper than cure
- find other defects early

We also use it to catch non-bug defects which can harm the project, such as performance issues or maintainability problems.

# what is code review?

- catch bugs early - prevention is cheaper than cure
- find other defects early
- share knowledge

But code review isn't just about making the one change-set we are reviewing better.

It's about making all the software we ship better.

It's important to think about how we can use code review to improve ourselves and the contributors we work with.

# what is code review?

- catch bugs early - prevention is cheaper than cure
- find other defects early
- share knowledge

I saw a talk last year at the Write the Docs conference by Paul Roeland and he said that we should stop aiming to be code ninjas and rock stars and start aiming to be code nurses and code tree surgeons.

I kind of see a reviewer's role as just like that.

It's not as thrilling as the ninja feature author, but it's really important to the health of the project.

the project's other goals

All of this happens within the context of the project's goals, which usually include:

\* Shipping code fast

# setting the scene


So let's back up and go over how we get to the review stage.

In the last few companies I have worked for, it's been something like this.



# setting the scene: the issue

The name of the user's latest employer needs to be shown in our app's new prototype feature - user profiles #2866

 **Open** adamtheturtle opened this issue 9 hours ago · 0 comments

Eleanor, our programmer, is new to the project.

She's looking to make her first contribution, so she picks a task from the task tracker.

This one is written by me.

> The name of the user's latest employer needs to be shown in our app's new prototype feature - user profiles.

# setting the scene: the branch

```
eleanor~> git checkout -b show-employer-2866
```

She creates a branch so that she can make changes to the code without worrying about breaking it.

Using branches for development is a great way to create a low risk environment for people to experiment in.

And it allows a reviewer to easily look at only the proposed change.

# setting the scene: the code

```
def get_user_profile(user):  
    """  
    Get a dict representing a user's profile.  
    """  
    user_profile = {}  
    user_profile['Age'] = user.get_age(unit='years')  
    user_profile['Twitter Handle'] = user.twitter_handle  
    return user_profile
```

After searching around the code Eleanor finds that there is already a method to get date-sorted employment history.

And there's already a get user profile function which returns a dictionary later handled by a view layer and displayed in the app.

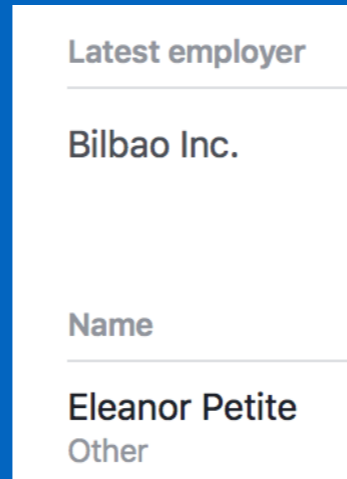
# setting the scene: the code

```
eleanor~> git diff
diff --git a/api/profiles.py b/api/profiles.py
index 776e5e0..7fff02c 100644
--- a/api/profiles.py
+++ b/api/profiles.py
@@ -2,7 +2,7 @@
     user_profile['Age'] = user.get_age(unit='years')
     user_profile['Twitter Handle'] = user.twitter_handle
+    user_profile['Latest employer'] = \
+    user.get_employers()[0].name
     return user_profile
```

So she adds some new code that she thinks might work.

It takes the date-sorted employment history and displays the first item from that list in a new field.

# setting the scene: the manual test



And then she runs the application locally and checks her profile.

Sure enough her latest employer is shown.

do we need a review?

So this change works.

Eleanor has seen that it works.

Do we need to spend time reviewing it?

**do we need a review?  
(yes)**

In my opinion, all code written as part of a team project should be reviewed.

I've heard objections to this - particularly when the changeset is trivial or absolutely urgent.

# do we need a review for trivial changes?

I'm going to propose that very little of what we do is trivial. Programming is hard, and authors have an inherent bias towards thinking that their code is good.

Even if we might waste a few minutes reviewing a really trivial change, we save hours debugging the bug which shipped when someone confidently merged their code which had a typo. And even if we are absolutely certain that the code can't break, a review means that now at least two people know where this code is.



# do we need a review for urgent work?

If the changeset is absolutely urgent and there really isn't anyone around to review it, it might be worth merging.

Maybe there's a data destroying bug that you've just shipped to production and need to revert.

But at the very least I suggest that an after-the-fact review takes place.

# stage fright

So Eleanor submits her code for review.

And she's nervous, because it is her first contribution to the project.

She doesn't mention that and she puts the code up for review.

# who should do the review?

- 
- 

Now we've got to decide who should do the review.

# who should do the review?

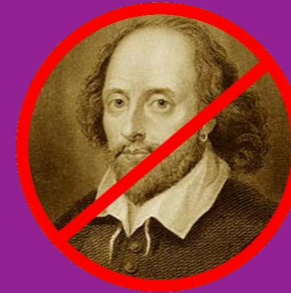
- someone with authorisation (technical or social)
- 
- 

Who gets permission to merge code is a whole problem that I won't go into,

but at least where I've worked it has been the case that anyone can review code, and anyone can merge it.

# who should do the review?

- someone with authorisation (technical or social)
- not an author
- 



As programmers we are biased towards thinking that our own code is working and readable.

And so the ideal reviewer is someone who has had no part in writing the code.

# who should do the review?

- someone with authorisation (technical or social)
- not an author
- someone who has written nearby code
- 

If we choose someone who has recently worked on the changed files then they are likely to have the best context to find the highest number of bugs.

# who should do the review?

- someone with authorisation (technical or social)
- not an author
- someone who has written nearby code
- or someone who has reviewed nearby code

According to a Microsoft study the most useful review comments come from people who have previously reviewed the modified files.

But choosing recent authors and reviewers conflicts with our goal to share knowledge with the team.

That may be OK but it is at least something to consider.

**but what about  
knowledge sharing?**

A good workaround to the knowledge sharing issue is to allow developers who are unfamiliar with the code to do a review but not to designate them as a gate-keeper - have someone else have to give the Looks Good To Me command.



# code review as a new starter

In fact, when I join a team, I like to jump straight into reviewing code.

When there's no pressure of knowing that I have to take responsibility for the code shipping it is a great way to learn how things are done.

# code review as a new starter


It teaches you about the project's quality bar and standards and how the process works from writing code to shipping it, much more than any JIRA workflow diagram will.

# how i do a review

... so, say I want to review Eleanor's changes, this is how I go about it.

# read the spec

The name of the user's latest employer needs to be shown in our app's new prototype feature - user profiles #2866

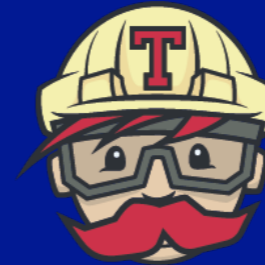
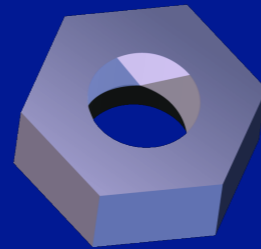
 Open adamtheturtle opened this issue 9 hours ago · 0 comments

I start by reading the issue spec.

Without reading the spec there's no way that I can know whether the code meets all the requirements.


Hopefully the spec is in an issue tracker and it has descriptions and examples of how the new functionality might be used or how to reproduce the bug if we're fixing a bug.

check ci



Then I check that the CI is passing.  
or at least that there are no new issues.

CI should be a signal of whether something existing has been broken, so it's good to check that first.



is the spec  
implemented?

Next, I look for evidence that the spec has been implemented.  
Ideally this means looking at the test changes.

If there's a bug to be fixed, I want to see a passing test which would fail if the bug were present.

If there's a new feature then I want to see tests which cover the spec.

**have we interpreted  
the spec the same?**

If it seems like the spec is ambiguous and I've interpreted it differently to the author, this is a good time to start a discussion about what we want.

**does the implementation  
look functionally correct?**

Then I look at the implementation.

I think about whether it has functionality that isn't tested.

That's important because even if the code works now, we want it to be safe from regressions.

I think about which missing test cases could convince us of that.



**has existing functionality  
been broken?**

Then I think about whether there's any risk that the changes could break existing functionality.

Hopefully our test suite will tell us but it is still worth considering.

**is it going to be easy to  
maintain?**

I think about whether the new code is going to be easy to maintain.

That might mean something as simple as understandable variable names and comments, but we also have to consider how easy the structure is to reason about.

is it going to be easy to  
maintain?

How branch-y is it?

How many side effects does it have?

Is it idempotent?

Is new code in the place I'd expect to find it?

All the things you'd think about when writing code.

**is everything  
documented?**

I check that everything user facing is documented.

# learning

I'm also looking out for techniques, patterns and tools that I don't know.

And it's always nice to say thanks to the author when you learn something.

# initiating a discussion

Then, I'll write a comment about anything that might be improved.

And I'll ask questions and start a discussion about anything I'm unsure about.

# concluding a review

- Great, this looks good to me, I'll merge it
- Please make the requested changes and then merge
- Please make the requested changes and then resubmit
- I'm not qualified to know whether this is a suitable change, let's find someone else to have a look

And at the end I'll give a conclusion. Usually I have a few options for conclusions.

I might say:

- \* Great, this looks good to me, I'll merge it
- \* Please make the requested changes and then merge
- \* Please make the requested changes and then resubmit
- \* I'm not qualified to know whether this is a suitable change, let's find someone else to have a look.

# getting better

I'd say the first step to becoming a better reviewer and doing those previous steps better is to take it seriously, like we do the rest of our craft.

If you learn about software from reading blog posts, look for blog posts about code review.

If you learn from more formal studies, search for studies about code review.



**ask for feedback**

And importantly, ask for feedback on your reviews, just like you ask for feedback on your code.

# reviewing eleanor's change

```
user_profile['Age'] = user.get_age(unit='years')
user_profile['Twitter Handle'] = user.twitter_handle
+ user_profile['Latest employer'] = \
+ user.get_employers()[0].name
return user_profile
```

So I look at Eleanor's code and I immediately spot a problem.

The "Twitter Handle" field is title case, but the new field is sentence case.

# reviewing eleanor's change



adamtheturtle commented 5 days ago • edited



The case of "Latest employer" isn't consistent.

I know that the Pull Request isn't up to scratch and I comment and reassign the issue to Eleanor.

# eleanor's next steps

Eleanor sees my comment and kicks herself for making such a trivial mistake.

Why can't I do anything right? She thinks.  
Maybe I'm not good enough to be here.

# eleanor's next steps

```
eleanor~> ag --ignore-case "latest employer"  
  
-/Documents/bilbao_inc/api/profile.py  
user_profile['Latest employer'] = \  
  
-/Documents/bilbao_inc/api/details_email.py  
    "Latest employer: {latest_employer}".format(
```

She then searches the code for “Latest employer”, case insensitive.

She's confused because it appears elsewhere in the code just like how she wrote it.

# being nice to people

Beginners don't have the confidence to protect them from feeling hurt.

And some experienced developers don't either.

Let's be wary of imposter syndrome and use code review as a forum to make this industry a nicer place.

But you don't want to let bad code be merged just to avoid hurting someone's feelings.

being nice to people  
always say one nice  
thing<sup>[1]</sup>

[1] Divmod via jml

One tip is to always say one nice thing about the code, and when someone has made an effort there is always something nice you can say.

being nice to people  
address the patch, not  
the person

Always address the patch and not the person.

At a previous company I was lucky enough to be thrown into the deep end in a project that had quite a steep learning curve. It took some time to get into the mindset that it was *\*my code\** being reviewed and not me.

Don't assume that everyone gets that.



# what else could be better?



adamtheturtle commented 5 days ago • edited



The case of "Latest employer" isn't consistent.

But what else could be better with this review?

Well, Eleanor doesn't know what to do next.

**what else could be  
better?**

**clear next steps**

Remember that one of our goals is to ship code quickly.

And by not making it clear exactly what's needed I've slowed down the process and added unnecessary work for her.

what else could be  
better?  
the review stopped too  
soon

```
user_profile['Twitter Handle'] = user.twitter_handle  
+ user_profile['Latest employer'] = \  
+ user.get_employers()[0].name  
return user_profile
```

And there's a bug a few characters later.

If a user has no employers we will get an IndexError.

But I stopped my review when I spotted the first error.

And that means that there needs to be an extra round of review which delays us shipping the code.

**what else could be  
better?**

**when to stop**

Sometimes there's a trade off - if a patch is very large and based on a completely wrong assumption, scrutinising every line isn't worthwhile.

# the next round

```
eleanor~> git diff
diff --git a/api/profiles.py b/api/profiles.py
index 776e5e0..7fff02c 100644
--- a/api/profiles.py
+++ b/api/profiles.py
@@ -2,7 +2,7 @@
     user_profile['Age'] = user.get_age(unit='years')
     user_profile['Twitter Handle'] = user.twitter_handle
+
+     try:
+         user_profile['Latest employer'] = \
+             user.get_employers()[0].name
+     except:
+         user_profile['Latest employer'] = \
+             "This user does not have any employment
history in the system yet."
     return user_profile
```

So after a back and forth and a comment which pointed out the IndexError, Eleanor resubmits.

So that's a code review success! We stopped a bug with the review. That's great.

# the next round

- performance issue

- 

- 

I start the next round of review and comment that a try...except clause has performance overhead that we don't want.

# the next round

- performance issue
- pep 8 violation
- 

and that the message line is greater than 80 characters long and that violates pep 8.

# the next round

- performance issue
- pep 8 violation
- properties vs getters

And finally I mention that using a getter for the list of employers isn't very nice and a property should be used instead.





adamtheturtle commented

Jenca member



Thanks for addressing those points.  
Here are some new points:

- It is possible to check whether the length of the list zero, that's faster than using `try...except`.
- The message line is greater than 80 characters in length, that violates PEP 8.
- Using a getter for the list of employers isn't very nice and a property should be used instead.

On the surface these might seem like reasonable comments.

But really this isn't a great review.

# priorities and context

As a reviewer I need to consider what is important in the current context.

Sometimes that's security, sometimes that's performance and sometimes it is maintainability, and sometimes it's something else.

As this is prototype code there's no need for me to be overly concerned by tiny performance issues, so the suggestion is a waste of time.

# style guide vs style rules

And as for the next comment, PEP8 is a guide, not a ruleset.

If you want to follow a guide then that's fine, but if you want rules then I recommend using a tool to enforce them.

That's partly because humans are bad linters, we miss things that tools catch.

And it is partly because using tools frees us up to focus on other, more important things.

# automate the boring stuff flake8



**Hynek Schlawack**  
@hynek

 Follow

It's hard to overestimate how much good flake8 & CI did to Python. Unlintered code is always a mess & painful to read. :-|

7:56 AM - 13 Mar 2016 · City Centre, Berlin, Germany

  9  10

Flake8 is one of my favourites.

It combines a few tools and enforces some PEP8 guidelines as rules.

# automate the boring stuff




## coverage.py & coveralls

```
52 POSTGRES_DATABASE = os.environ.get('POSTGRES_DATABASE', 'bilbao-authorisation')
53
54 if POSTGRES_HOST is not None:
55     if POSTGRES_HOST.find('env:') == 0:
56         POSTGRES_HOST = os.environ.get(POSTGRES_HOST.split(':')[1])
57     SQLALCHEMY_DATABASE_URI = "postgres://%s:%s@%s/%s" % (
```

There's coverage.py and coveralls to check line coverage.

# requires.io

## requirements.txt

Package	Requirement	Latest	Status
 click 	==6.6 	6.6  	up-to-date
 psutil  	==4.1.0 	4.3.0 	outdated

There's requires.io which checks for vulnerabilities in dependencies.

automate the boring  
stuff

landscape.io



11

CODE ERRORS

✓ 161



4032

STYLE WARNINGS

^ 3280



1166

CODE SMELLS

There's landscape.io which notices common code errors.

automate the boring  
stuff

docs tools



and there are even tests for docs:

If you're using reStructuredText, there's `doc8` which lints your documentation.

There's Sphinx's link checker and spell checker.

And there are similar checkers for Markdown and probably whatever docs toolchain you use.



# trade-offs

Sometimes these tools will suggest changes which are actually worse.  
My personal preference is to just follow the tool's suggestions.  
Overall you code will be nicer and you save the cognitive burden of linting by eye.

# ignoring suggestions

```
# pylint thinks that this is too long for a function name.  
# Ignore that error.  
def update_current_search_index(): # pylint: disable-msg=C0103
```

If you really want, most tools allow you to ignore a particular issue.

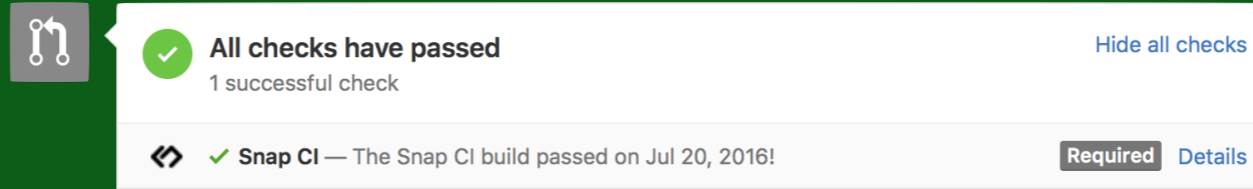
Here I've told pylint to ignore the function name length check just for this line.

# configuring suggestions

```
[flake8]  
ignore = D203  
exclude = .git,__pycache__,docs/source/conf.py,old,build,dist  
max-complexity = 10
```

If a suggestion keeps coming up in your project and as a team you don't like it, most of these tools are configurable in some way.

# github integration



A screenshot of a GitHub integration notification. The notification is white with a green border and is set against a dark green background. It features a pull request icon on the left. The main text reads "All checks have passed" with a green checkmark icon and "1 successful check" below it. A "Hide all checks" link is on the right. Below this, a specific check is listed: "Snap CI" with a green checkmark, followed by "The Snap CI build passed on Jul 20, 2016!". To the right of this check are "Required" and "Details" labels.



If you use GitHub these tools can tie in with Pull Requests and you can actually make it so that code can't be merged if they detect problems.

reviewable.io



Finally, reviewable is a really powerful review tool which has some great features that GitHub doesn't have.

Like you can review just the changes made since your last review.

# unrelated changes



adamtheturtle commented

Jenca member



Thanks for addressing those points.

Here are some new points:

- It is possible to check whether the length of the list zero, that's faster than using `try...except`.
- The message line is greater than 80 characters in length, that violates PEP 8.
- **Using a getter for the list of employers isn't very nice and a property should be used instead.**

If we go back to my review, we see a suggestion of an unrelated change - changing an existing getter to a property.

If you want an unrelated change, it is best to make an issue in your issue tracker for later.

A review might be a good place to spot required unrelated changes, but not to request them.

# unrelated changes



adamtheturtle commented

Jenca member



Thanks for addressing those points.

Here are some new points:

- It is possible to check whether the length of the list zero, that's faster than using `try...except`.
- The message line is greater than 80 characters in length, that violates PEP 8.
- **Using a getter for the list of employers isn't very nice and a property should be used instead.**

Asking for unrelated changes slows down shipping the code and it also makes the next review round less pleasant.

In fact as a reviewer I should request easily reviewable changes.

# reviewable changes

but what are reviewable changes?



# reviewable changes are short



Magnetic platform guide: ~200 line changes max

Twisted: ~200 line changes max

Statistically, more defects are found when change sets are shorter.

**reviewable changes  
include one logical unit of  
change**

As a reviewer you can ask that a patch be split up if it is difficult to review.

And you might be able to help decide where the splits should be.

Sometimes people like to refactor surrounding code before making a change and that refactoring can be separated into a new pull request.

# writing reviewable code

And when you spend a lot of time reviewing code, you start to get a good idea of what is going to be easy to review and you'll start to write code like that.

Writing easy to review code means reviewers can be more effective and find more bugs.

# silicon valley comma



**Jay**

@jaybuff

 Follow

The silicon valley comma: a trailing comma in an array.  
Annoyingly not valid in JSON.

8:58 PM - 18 Mar 2016

  4  2

When we focus on making a reviewer's life easier, we write code which can be extended without changing what's already there too much.

A trivial example of this is the silicon valley comma.

# silicon valley comma

```
landmarks = [  
    guggenheim,  
    fine_arts_museum,  
    basilica,  
+   euskalduna,  
]
```

VS

```
landmarks = [  
    guggenheim,  
    fine_arts_museum,  
-   basilica  
+   basilica,  
+   euskalduna,  
]
```

If you add a comma to the end of the last item in a vertical list then any additions can be shown as just a new line in a diff.

The diff doesn't include the modification of the last line to add a comma.

# semantic linefeeds

```
adam~> git diff
diff --git a/api/example.txt b/api/example.txt
index 776e5e0..7fff02c 100644
--- a/api/example.txt
+++ b/api/example.txt
@@ -2,7 +2,7 @@
-     The beauteous scheme is that now,
+     The beauty of this scheme is that now,
     if you change your mind
     about what a paragraph should look like,
```

by brandon rhodes

And similar applies to docs.

We mostly change sentences, not paragraphs so we should make it so that diffs of changed sentences are as small as possible. We can do this by writing our docs as one sentence or clause per line.

# tips vs requirements

The best reviews aren't laser focused on just providing the bare minimum information needed to ship, but they also don't add unnecessary blockers to merging.

Explicitly stating that a suggested improvement is optional can let the author choose whether, let's say, to learn the suggested tool or to not do that and just ship. In each case the priorities can be adjusted and when on boarding a new developer sometimes the pickier review can be best, to get them in sync with the rest of the team.

# learning by reviewing

Every review comes with a mini “how would I have done this?”.

When we read someone else’s approach and notice that it is *\*better\**, we learn techniques and patterns.





# TWISTED MATRIX LABS

<b>1ST</b>	<b>RODRIGC</b>	<b>63162</b> <b>!!!</b>
<b>2ND</b>	<b>ADIROIBAN</b>	<b>52565</b> <b>!!</b>
<b>3RD</b>	<b>LUKASA</b>	<b>27125</b> <b>!</b>

200 POINTS FOR SUBMITTING A TICKET FOR REVIEW!  
500 POINTS FOR CLOSING A TICKET!!  
1000 POINTS FOR A REVIEW!!!

Teams which value review get better reviews back, and get reviews done quickly.

My favourite example of this is Twisted's high score board which offers more points for reviews than for submitting code.

cheers

adam dangoor

twitter: [adamdangoor](#)  
github: [adamtheturtle](#)  
[stuffadammakes.com](http://stuffadammakes.com)

Thank you for having me