REVIEW FEEDBACK

Edward Phillips 20/05

20 May 2020 / 01:00 PM / Reviewer: Katherine James

Steady - You credibly demonstrated this in the session.
Improving - You did not credibly demonstrate this yet.

GENERAL FEEDBACK

Feedback: Thanks for a great review - you are nice and steady across all the course goals, well done and keep it up! Lots of other examples at this site: https://kata-log.rocks/tdd, if you want to practice the idea of continually using your existing code, which I suspect you might still have an inclination to discard from our conversation about your if-statements and the loop at the end.

I CAN TDD ANYTHING – Steady

Feedback: Testing approach:
Your testing approach was really nice. Your tests were based on the overall acceptance criteria, so were nicely decoupled from your code. Each test made you develop your code iteratively from the test before.

Test progression:
You laid out some tests in your IO table. You incremented the complexity really nice and iteratively. I might have been tempted to swap the case with the equal limits 10,10 with the 5, 10 limits, as the equal limits case is a special case of the general case. Otherwise, I would say pretty much perfect test progression in your IO table.

You provided good justification whenever you added an additional test to your progression, as you felt was necessary. These tests were good and were at the right level of complexity for when they were introduced.

Red-green-refactor (RGR) cycle:

Your RGR cycle was punctuated with regular commits to Git, which made your process really clear. You stuck to the cycle nicely, making your tests pass in as simple a way as possible.

Have a look at 31.06ish in the video. You'll see that you have local variables for the limits and that you need to make the code work for equal limits, that are supplied. Simply lifting those local variables and turning them into parameters would have taken care of this test (and the ones you introduced because you didn't take this approach). As you didn't see this, it meant you added some more tests and a fair bit more code, but you did do this nice and iteratively over several steps, sticking to your RGR cycle, so this wasn't a process problem, just a something you might see with experience.

You tried really hard not to throw away your old code and to continuously update it. On your last test, you said that you hadn't actually used the if-statements you had written before to fill in the loop. Remember that you are putting all the effort into developing that code for a reason with the simple tests - simple tests are there to develop the code that is going to go inside your loop, with minimal upgrades. Go look at 56.25 in the video and contemplate the code in the upper if-statement block and your loop respectively. Note - they're virtually the same! Just with pushes instead of hard-coded returns.

I CAN PROGRAM FLUENTLY – Steady

Feedback: You demonstrated familiarity with methods, if-else statements, default parameters and array manipulation. You seem to have good programming fluency.

While for loops are common and the go-to in most other programming languages, in Ruby, it can be argued that we should almost universally avoid them, as there are more efficient options. For loops are not as space-efficient as using the 'each' loop for example, because they store the variable that represents each element.

I was curious that you chose to use a nested if-statement to get your above the range test working in the simple test (26.29), rather than an if elseif else, which is what you refactored to after this step. I wasn't quite sure as to the reasoning behind this.

I CAN DEBUG ANYTHING – Steady

Feedback: You were able to find the most important part of the error message in the stack trace, were familiar with the common errors generated by failing tests and let me know which errors you were expecting.

I CAN MODEL ANYTHING – Steady

Feedback: You modelled your solution in a single method which I felt was a nice and simple implementation and provided a good place to start. It's best practice to make method names 'actionable', so ideally, method names should contain a verb.

I CAN REFACTOR ANYTHING –Steady

Feedback: You did a really nice refactor to remove the magic numbers, and to make your code more logical. Very nice. This was done at just the right time, as it meant that all you had to do to get the next test to pass was to change your local variables to argument names (unfortunately you didn't see this). Later on, you refactored to include default values in the parameters to do this. Watch out for symmetries in your code as refactoring opportunities.

I HAVE A METHODICAL APPROACH TO SOLVING PROBLEMS – Strong

Feedback: You thoroughly interrogated the technical requirements before starting coding, which was really done very well. You used a really nice and clear convention for your Git commits, to differentiate between red and green test commits. You commented out code after a refactor to make it easy to reinstate if something went wrong. You marked off tests you were done in your IO table and prioritised core over edge cases. This was all very good.

I USE AN AGILE DEVELOPMENT PROCESS – Strong

Feedback: You conducted a really thorough information gathering session. You started by verifying what a band pass filter actually did and then asked some follow up questions to verify the input type of the soundwave and how the limits would be passed in, as well as their type. You determine how the output should be formatted from the example I provided.

You really had a nice and thorough interrogation of possible edge cases and handled this really professionally, with concern for the client's preferences. You asked about non-integer inputs, asked for clarification on the corrupted inputs, considered that the limits might be the same, the lower one larger than the higher one and what should happen if no limits were supplied (defaults).

You verified how the program should be used - print or return (really nice) and created an excellent input-output (IO) table, allowing the client a chance to verify the examples you typed.

I WRITE CODE THAT IS EASY TO CHANGE – Steady

Feedback: You committed regularly on the red and green steps of your cycle, using a methodical convention. You missed out one or two commits. If you want to save a bit of time in reviews, you can just commit on greens.

I was pleased that you had your test suite properly decoupled from your implementation by making sure the tests were based solely on acceptance criteria, and not reliant on the current implementation. This makes changes to the code much easier.

I CAN JUSTIFY THE WAY I WORK – Strong

Feedback: You provided really nice justification for removing one of the tests from your IO table, as it was redundant. You verbalised your process really

nicely, so it was easy to follow what you were currently working on. You let me know which test you were testing, and what case this was testing, and how the program should behave in this case. You also provided a really nice justification for waiting to refactor the magic numbers out of your code.