Review

Start date: Tue 10 May 11:00am BST
End date: Tue 10 May 12:00pm BST
Week number: 15
Zoom link: See confirmation email
Leave Feedback

Review History

- Tue 22 March (week 8, School results)
- Thu 31 March (week 9, Band pass filter)
- Tue 3 May (week 14, Spellcheck)
- **Tue 10 May (week 15, String calculator)**

Scorecard

Exercise: String calculator
Reviewer: Tamira Gunzburg
No show?: No
Language: Ruby

I use an agile process

Rating: ***Strong***
Excellent requirements collection. You systematically moved from input formats to the required transformation to output formats to edge cases, and mapped several simple and complex core cases as well as edge cases in an input-output table. You asked your client lots of questions and also made assumptions that you took care to validate with the client. Fabulous job.

I can model anything

Rating: ***Steady***
You chose a class so that you can add multiple methods later on should it be required. That is valid, but since your did not store anything in state, you could possibly have achieved the same with a function for now. You apply naming conventions correctly, and do extra research to ensure your code conforms with the Ruby standard. Well done.

I can TDD anything

Rating: ***Strong***
Red: Excellent choice for a first test: single term, so no need to split the string as yet, and no need to evaluate the sum. Your second test passed straight away, which is usually a flag in TDD. Indeed, you had made your code responsive to all input of one term, all within the first RGR cycle. However, you did make sure it failed properly given the wrong output - good instinct. Green: Great job hardcoding the return initially to get the test passing asap, and then using your refactor phase to get rid of the hardcoded value and make it responsive to any input of that type. As your programme becomes more complex, you may in addition want to split your green phase into two or more bits, where you hardcode initially and then write another test to force you to make the return value responsive. In this way, you split each "feature" into more tests and less code per test. It can help you spend less time in the green phase per cycle, and break things up into even smaller chunks. Refactor: Good refactor to remove the hardcoded return value of the first test and replace it with a method that turns any valid string number into a float. Good refactors to your test names too, and props for waiting with pulling out the class initialisation to the outer context until the refactor phase. Well done. If you're ever worried you'll forget a refactor that pops to mind, you can always add a comment as a reminder.

I can program fluently

Rating: ***Strong***
You are very familiar with the command line and the Ruby language. You do research to make sure you get the most out of the Ruby language and its functionality.

I can refactor anything

Rating: ***Strong***
Good work refactoring your code to remove hardcoded values, and researching Ruby built-in methods to achieving the best way to get the test passing for all cases of that nature. You also refactored your test code to remove duplication and improve readability. Great stuff.

I can debug anything

Rating: ***Steady***
When you got an error message running your first test, you carefully inspected the stack trace and found out the error had to do with how you were importing your implementation code into your test code. You didn't get too stuck. Similarly when you were trying to call the method on its own without calling it on the object of the class. You also use print statements to get additional insights into what your code is doing. Your debugging process is good. Look into "hypothesis-driven debugging" to make your process even more deliberate.

I write code that is easy to change

Rating: ***Strong***
Good job keeping your implementation code and test code decoupled, and committing to Git frequently. Good instinct to commit again after your refactor. This keeps your commit messages short and focused, and keeps commits with new functionality separate from commits without new functionality. In turn this makes your commit history easier to read and navigate later on. No need to commit after writing a failing test, rather keep your commits to working code (i.e. that passes all tests). You use expressive variable names which also helps make your code easier to read and change.

I have a methodological approach to problem solving

Rating: ***Strong***
You start with core cases, follow a regular RGR cycle, your tests progress logically, you collect requirements systematically, and you debug methodically. This rigour in your approach will stand you in very good stead when working on more complex challenges.

I can justify the way I work

Rating: ***Strong***
Great work vocalising what you are doing, what you are thinking, the options you are considering, and the reasons for the decisions you are making.

General feedback

Excellent work! You are methodical which allows you to analyse a problem and steadily work your way towards a robust solution without racing ahead or getting stuck. I recommend that you finish this exercise on your own time (with multiple operators) and think through different architectures for your solution, with their pros and cons. For future challenges, try breaking down your green phase into smaller chunks by writing more tests and less code per test (e.g. by hardoding initially) - this becomes useful as your programme becomes more complex and the new functionality requires more logic.

Video