REVIEW FEEDBACK

Harry Turnbull 12/06

12 June 2020 / 08:00 AM / Reviewer: Katherine James

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

GENERAL FEEDBACK

Feedback: Well done on an excellent review. You are extremely competent and this came across really well in the review today and in your fantastic steadiness across the course goals. Keep it up!

For fun, here are some dictionaries I might have supplied:
http://www.gwicks.net/dictionaries.htm

I CAN TDD ANYTHING – Strong

Feedback: Testing syntax: You started by using to equal instead of to eq. These are subtly different. It was good to see you realise this from the error message and adjust to eq.

Testing approach: I was really pleased to see that you were basing your tests on overall input-output behaviours (acceptance criteria). This means that your tests are nicely decoupled from your code and that you can change your implementation without breaking your tests.

The only test that didn’t conform to this was when you tested that your code could read in an external text file. This was more of a structure first test. Remember, we want to be testing the behaviour of the program, rather than internal implementation details. I would have preferred the dictionary read to have occurred as a refactor rather than as a test, as if the implementation changed at a later point and we switched to a gem instead of a txt, then this test would break. Testing behaviours means we turn implementation into a black box and as long as it works, we are happy.

Test progression:
Nice test progression. You started by identifying examples based on each case (correct/incorrectly spelt) and basing your early tests on these, adding more examples to break the assumptions of your code and using these examples to drive some simple if-statements in your code which upgraded through refactors. You then moved onto more complex versions of the simple examples by identifying that an assumption of your code was that it always assumed lowercase words. This identification of assumptions and elimination through testing was excellent to see. You then moved onto testing multiple words that were correctly spelt.

Red-green-refactor (RGR) cycle:
You stuck to your RGR cycle meticulously.

I CAN PROGRAM FLUENTLY – Steady

Feedback: You are nicely fluent with programming in Ruby. You used an array with to denote your dictionary which was a good choice, you showed good familiarity with in-built methods through your immediate selection of .include? and join for rejoining your string. You used map to manipulate the elements in the array which was a good choice. You were slightly unfamiliar with reading in files with their paths, which you wanted to do to load the dictionary, but used the Ruby docs as a reference, which was great to see as this is exactly the right thing to do when unsure of the syntax.

I CAN DEBUG ANYTHING – Steady

Feedback: You knew which error messages to expect from failing tests and were familiar with the common types of errors that occurred. You used prints to get some visibility into your code and form a feedback loop (and sanity check that words were being matched by the if-statement). You could identify the most important part of the error message in the stack trace and solved bugs well.

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. Note that method names should be actionable, so should contain a verb, eg: check_spelling as opposed to spellcheck.

I CAN REFACTOR ANYTHING –Steady

Feedback: You looked for your first refactor at just the right time and added a dictionary (array) to your code, refactoring to check if a correctly spelt word was included in the dictionary and to return a string interpolated result. You could have broken this refactor up into two refactors, the first to switch to string interpolated returns, then to check the dictionary.

I HAVE A METHODICAL APPROACH TO SOLVING PROBLEMS – Strong

Feedback: You stuck to you RGR cycle nicely, leaving making the code refactored for refactor steps and doing the simplest thing on a green step. You kept a good log of the tests you wrote in your IO table, which is a really nice reference. You used three panes for easy reference between tests, readme and code.

I USE AN AGILE DEVELOPMENT PROCESS – Strong

Feedback:
You asked excellent, to the point questions to determine the main technical requirements of the program. You started by establishing the main program operation, then clarified the input type, how the output should be formatted and established which words would need to be checked.

You established finer details, clarifying points like how capitalisation should be treated, hyphenated words and punctuation. It might have been nice to type the output for these to prevent ambiguity for the hyphenated output, as I realised later that I wasn’t sure if you meant the output would be ~old-wrrld~ or old-~wrrld~.

You typed up a list of words in an input-output (IO) table and considered both correctly spelt versions and incorrectly spelt versions nicely.

I WRITE CODE THAT IS EASY TO CHANGE – Steady

Feedback: Git: You committed whenever your tests went red and green, or when you had to make another significant change, such as changing your testing matchers. Note there isn't much point committing twice in a row if you forget the red commit, this will just serve to have the relevant labels in your commit history. Commit messages should ideally start with a capital letter. In the work place, you will have to establish what your company has decided is the best 'format' for commit messages, but starting capitalised is generally a good guideline. This seems to be a reasonable guide with some more detail: https://www.freecodecamp.org/news/writing-good-commit-messages-a-practical-guide/

Decoupled tests: 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.

Variable names: You chose variable names such that it was clear what they represented.

I CAN JUSTIFY THE WAY I WORK – Steady

Feedback: You let me know when you got the errors you were expecting and what these errors were. You let me know what you were adding to the code to make the error change/go away. You let me know aspects such as that your matcher was wrong, and that you had to update the tests as a result. You vocalised your thoughts well when changing to a new level of test complexity.