Testing for the Terrified

How to write tests, conquer guilt, and level up

Frances Hocutt
Rackspace
LISA19
October 30, 2019

What this talk is not

An extended argument for automated testing

Integration tests, functional tests, etc

Intermediate+ unit testing strategies

An introduction to test-driven development

What this talk is

If automated testing is so great, why don’t we always do it?

A harm reduction approach to learning testing

Getting from “I know the idea behind unit testing” to “I can look at some code and write a few tests for it”

Let’s write some tests!

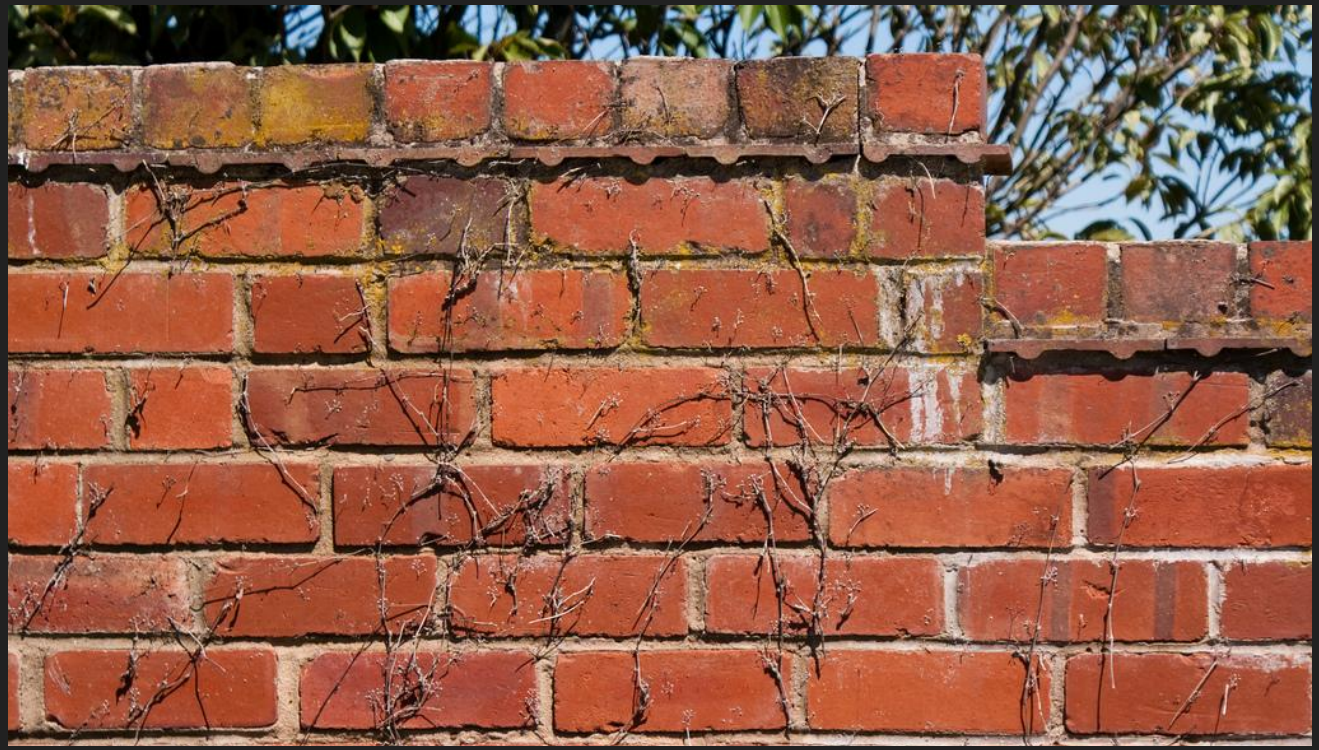
Where to go next?

Testing is pretty cool

Automated unit testing:

- Promotes helpful code habits (modularity, reusability, etc)
- Reduces debugging time
- Reduces duplicated work
- Is free documentation

So why don’t we test?



“Brick Wall,” by Pleasence, https://flic.kr/p/6wd9uJ

Why don’t we test?

“I can write code, testing it can’t be that hard…”

[Time passes]

“Okay, there are three or four kind of different frameworks and I have no idea which is actually best...”

Why don’t we test?

“I guess this thing talks to an API, so do I need a mock or a stub or maybe it’s a patch and I’m not even sure I’m testing something real at this point…”

Why don’t we test?

“Why would you even bother submitting a patch without tests?”

“Untested code is legacy code.”

“What do you mean you’ve never done test-driven development?”

Why don’t we test?

"I would, but I looked at a tutorial and it told me I should be writing tests first, and I don't even usually write code around here."

Why don’t we test?

“I know I’m supposed to test all of my code and I can’t… oh look, a production incident, I’ll totally come back to this.”

Why don’t we test?

“Well, this script is a one-time use thing…”

Harm reduction

People have reasons for what they do, and it’s more respectful and effective to meet them where they’re at and offer resources for making the changes they want to make.

Unit test basics

Controlled input == expected output

Unit testing skills

Write code that isolates side effects/network interactions

Find the parts of your code that are easier to work with

Figure out useful inputs to test with

Write code that tests what you want to test

Where to start

Write tests that document what your code does now

Start somewhere! 10% coverage >>> 0%

Start with pure functions (one input → one output, always)

- No I/O, no disk writes, no API calls, no changes to global variables, no other side effects

Give the functions some inputs, find the outputs, write it down in a test

Let’s test!

`commandbuilder.py` is an interactive wrapper script for the (made up) `whyohwhy` tool (whose authors should perhaps take a look at command-line UX)

Where do we start?

You just saw me:

- Look at existing code
- Find a pure function
- Find a friendly testing framework
- Write some failing tests
- Pick some reasonable inputs
- Feed them to the function
- Write some tests to pin down the current output of the function
- Test that the function raises exceptions when appropriate

What next?

Practice the easy stuff!

Practice refactoring your code so there’s more easy stuff!

Gear up for the harder stuff

- Read code
- Pair -- but make sure your goals line up
- Look at more advanced testing resources once you have a sense of how this works (*Working Effectively with Legacy Code* is a classic)

Use a code coverage tool to get some numbers on there (e.g. coverage.py)

Thank you!

frances.hocutt@rackspace.com