Assignment 1

Daniel Gordon
UW ID: xkcd
Hours to complete: Approximately 3 hours

Problem 1:

Dynamically typed languages often tend to be too flexible with return types. Most of my time in grad school, I have worked in Python, and I have dealt with this problem on at least a weekly basis. Usually what happens for me is I am using some library, and I am unsure about what type of object will be returned by a function. Often there is no way to see what will be returned other than by running the function, which may take inputs and be hard or time-consuming to call in an isolated manner. What I often do is try my best to determine the type of a returned object by looking at source code or documentation (if available), making a guess, and putting in a debugging statement or intentional error right before the value is used. Then, when I run the code, I can check the value by hand and make any necessary modifications. This can certainly cause issues though, as if the code is sufficiently hard to reach, it may take extra effort to try and run the specific lines. Furthermore, by only checking at runtime, I can never be totally sure that there isn't some other valid object type that the method could return. Writing valid code that works on all possible output types can cause increased code complexity and make maintenance much harder.

This problem is inherent to all dynamically typed languages as far as I understand. By allowing the programmers to be flexible while writing the initial code, it can cause static analysis of code to lose the ability to catch errors based on expected types. There are some frameworks in which you can annotate functions with a return type (like Google Closure for Javascript), and a compiler will check that annotated types match. However, since this is an addition to an existing language, if someone is using a completely valid but unannotated library written in Javascript, they will suffer from many of the same issues.

An obvious solution to this problem is to only use statically typed languages, but since dynamically-typed languages already exist, that won't always be an option. Annotation tools are another good solution, but require more work than strictly necessary to create valid code, so it may take some convincing that it will actually save time for the developer and for others who want to use the code later. The least invasive solution would be to comment every method with a return type, but in my opinion that is a worse solution since it requires equally as much work but provides fewer solutions since no type checker or compiler will ever use the commented type.

Problem 2:

There are clearly advantages to in-place operations – sometimes deep copies of objects are not needed and will waste both time and space being created. However if a programmer expects an operation

to be deep, but the result is actually shallow, there is often no good way to catch these errors. Even worse, this will often lead to bugs which are very hard to find since a seemingly unrelated variable modification will cause changes to another preexisting variable. I have found this to be specifically difficult when using Numpy, a python library for fast matrix-like operations. For instance

var1 = np.ones(10) # create a length 10 array filled with [1,1,1…]
var2 = var1[:2] # create a variable referring to elements 0,1,2 (shallow)
var 2 *= 2 # multiply all elements by 2 (shallow)
var3 = var2 + 1 # add 1 to all elements (deep)
	var1 = np.ones(10)
	var2 = 2 * var1[:2] # (deep)
	var3 = var2 + 1 # (deep)

In both cases, Var3 will be [3, 3], however in the first case, Var1 and Var2 share memory, so Var1 will also be modified by line 3.

This can be especially hard to catch because in both cases Var3 holds the expected value, and var1 may never be accessed again. But if var1 is accessed again, the results may be unexpected if method 1 is used. The problem can essentially only be solved in two ways. The first is to never use in-place or shallow operations, but as stated this has numerous performance implications. The second is to very clearly comment methods that are in-place, and potentially include conventions such as labeling all in-place methods with a suffix such as having two methods “add(x, y)” and “add_in_place(x, y).” This would only be a convention, which programmers could freely ignore, so the problem would still exist in many codebases.

Problem 3:

Writing unit tests are useful for two primary reasons. They help verify that the code behaves in an expected manner, and they automatically check that changes to the code don’t break previously expected behavior. One issue with unit tests is that they are limited to what the programmer thinks are sufficient tests. There could still easily be issues that these tests don’t catch, and similarly there could be functionality that isn’t explicitly tested but could be broken by some other change. I have run into this when working at Google. I created a progress slider that moved when the mouse dragged it. I wrote tests that took an x and y location for the mouse and checked that the slider was correctly positioned. Later, someone modified how the mouse’s x and y location were stored which broke my slider, but my tests still passed because I was testing on raw integer inputs rather than reading from a mouse location.

This type of bug seems to be common in large projects with many programmers adding and modifying code that affects others. It can also be caused by only writing unit tests using a few expected inputs. To try and prevent this in the future, we can try to write tests that cover as broad a range of inputs as we can think of. It is important to also think about what existing code may break because of a certain change, and to double check that things are still working even if all tests pass. Additionally, it may be useful to generate lots of random input tests to make sure coverage of possible inputs is large enough to catch most existing bugs.