Testing your solution

Introduction

In this reading, you will learn about how you can build testing into your take-home assignments and a coding interview, you'll also explore the practice of writing unit tests.

Testing is a key part of software development. Ideally, the scope of your tests is comprehensive; however, there are always external considerations that can affect this ideal outcome. Historically, testing has often been done at the end, so if a production deadline is looming, tests can be the first thing to be reduced. Test Driven Development (TDD) is an approach developed to avoid this outcome. This discussion of tests will focus on two instances, a coding interview and a take-home assignment.

Take-home assignment

If you have been allowed to take some code home, you may have time to implement more comprehensive tests.

There are many types of tests that you can choose to run. For example:

- Integration tests: These test how various components of an application interact with one another. An integration test will not be in-depth because the external dependencies will be simulated rather than using actual instances. Further, unit tests assess individual lines of code, while integration tests take more of a global approach.
- Functional tests: This is an automated test that proves that a system operates as expected. This test is concerned with the capabilities of the system.
- Regression tests: This is to test that a change does not cause an error in the existing code. It ensures that when a system change is made, it does not affect its operations.

The level of testing you employ will always be time-dependent. A crucial part of the process is creating a working application. Testing shows your thoroughness and ensures that the application will work. The degree to which you can include these strategies may be limited depending on how much time you have.

Coding interview

Unit testing is an approach that confirms your code is working as expected. While many tests are written before code goes into production, you will not have an opportunity to write all these when engaging in a coding interview. Unit testing is a manageable approach that can be incorporated into your solution, demonstrating your attention to detail. It will demonstrate your good practice of testing your code.

A unit test is less concerned with an application's overall operation. Instead, it tests that each of the individual components works as expected.

Considerations when writing tests

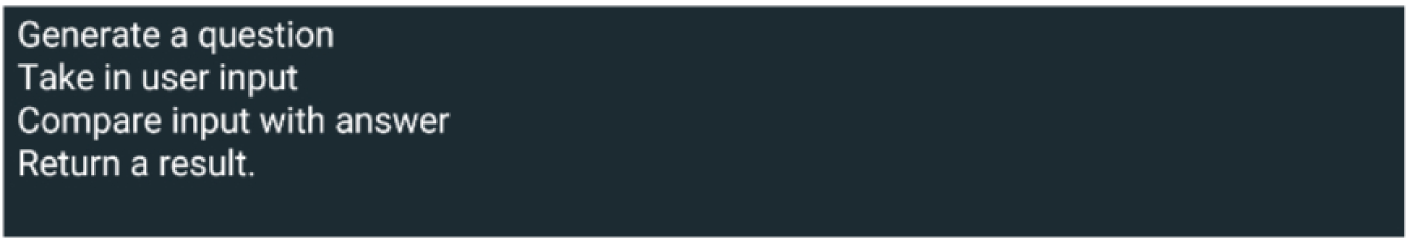
- Readability: Make the tests readable for other developers. This habit should be internalized regardless of whether you work in a team. Tests with a clear purpose identify problems should they arise. They also signpost what a section of code is supposed to achieve. This has the added effect of increasing maintainability.
- Clear outcomes: Your tests should, whenever possible, be deterministic. This is to say that the result should always be the same regardless of the conditions. A deterministic test will always fail when buggy code is written.

Tests dependent on a combination of different conditions can be challenging to debug.

- Automation: Tests should always have the potential to be automated so that they can be run quickly whenever a change has been made to code. This is a cornerstone of CI/CD (Continuous Integration/ Continuous Development).

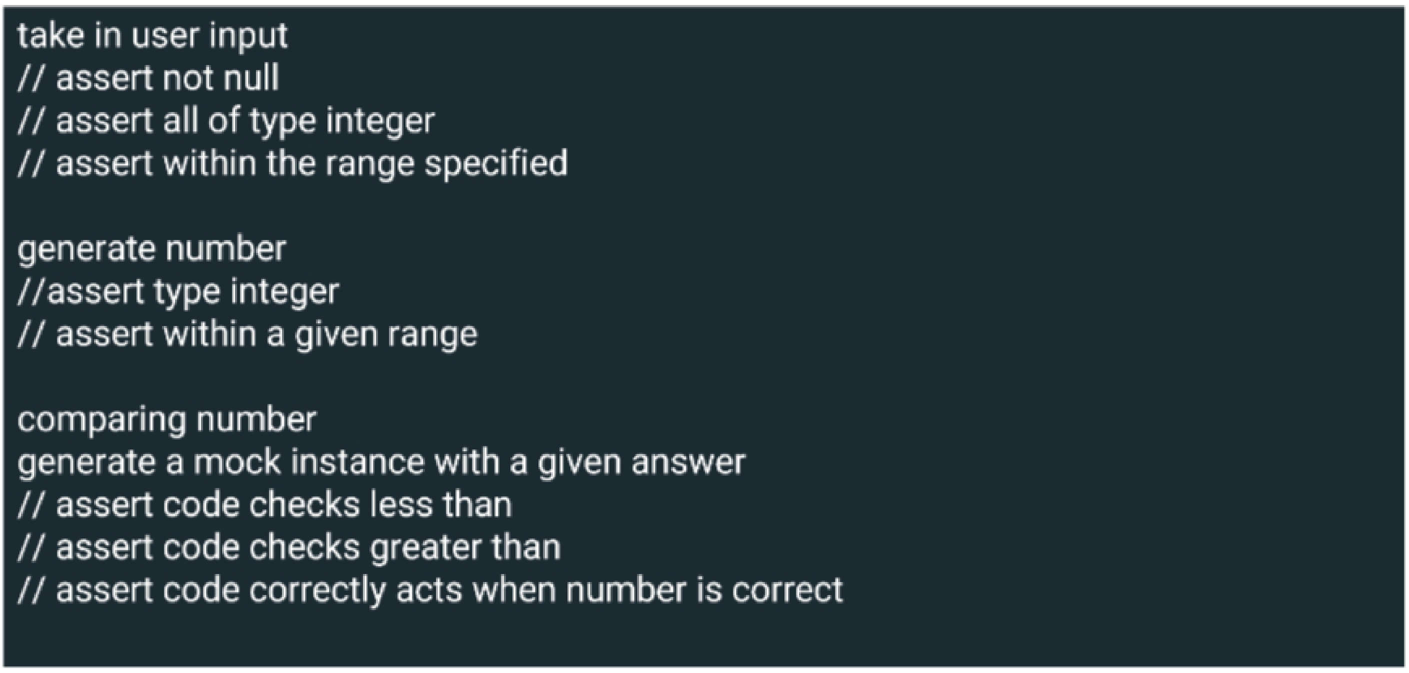
Putting unit tests into practice

The defining solutions reading outlines the steps needed to make a number-guessing game. Below is a screenshot of the pseudocode used to develop a viable solution.



```
Generate a question
Take in user input
Compare input with answer
Return a result.
```

Taking this example, imagine that there are only a few minutes left in the allotted time, what would you focus on? The most basic unit tests that can be written are assert cases, which determine that what is given is as expected. Libraries of these are available in most coding languages.



```
take in user input
// assert not null
// assert all of type integer
// assert within the range specified

generate number
//assert type integer
// assert within a given range

comparing number
generate a mock instance with a given answer
// assert code checks less than
// assert code checks greater than
// assert code correctly acts when number is correct
```

The first step may be to write a test case to determine whether the user input is valid. Lines 2 – 4 outline some tests that can be applied to validate user input, checking that something was entered before the user pressed return. You should determine that it is of the correct type and will not crash the program. Further, ensure that it is within the specified bounds. Next, you should consider checking the random number generated. Is it of the correct type and does it fall within acceptable parameters? Finally, it would help if you thought about including checks that ensure the right outcome when the conditional statements are activated.

The process for writing unit tests is standardized. It is a good idea to practice implementing them when you are writing code. This will help you quickly familiarize yourself with the process and can highlight your mindfulness in an environment where you are testing your code.

Conclusion

In this reading, you learned about how you can build testing into your take-home assignments and a coding interview and you also explored the practice of writing unit tests. After reading all of the above content you should now know that assessing your code is particularly important.

Often the practice of writing tests can be marginalized compared to the time spent implementing an application. To avoid this coding pitfall, ensure that you write tests as you code. While some testing can be in-depth and time-consuming, writing unit tests can be done quickly. This habit can lend weight to the argument that you are a fit candidate for a post. If you can incorporate unit tests into an assignment, particularly ones with time constraints, you will be well-prepared for a coding interview.