Creating Test Cases

Practical Tips for Designing Test Cases

Writing good tests cases requires a lot of thought: you need to think about a wide variety of things that can go wrong. Here are some suggestions to guide your thinking, as well as important lessons. First, some thoughts for testing error cases:

- Make sure your tests cover every error case. Think about all the inputs that the program cannot handle, i.e., ones where you would expect to receive an error message. If the program requires a number, give it "xyz", which is not a valid number. An even better test case might be "123xyz" which starts out as a number but isn't entirely a number. How the program should handle this depends on the requirements of the program, but you want to make sure it handles it correctly.
- Be sure to test "too many" as well as "too few". If a program requires exactly N things, test it with at least one case greater than N and at least one case with fewer than N. For example, if a program requires a line of input with exactly 10 characters, test it with 9 and with 11.
- Any given test case can only test one "error message and exit" condition. This means that if you want to test two different error conditions, you need two different test cases: one for each error condition. Suppose that our program requires a three letter string of lower-case letters. We might be tempted to test with "aBcD" to test two things at once, but this will only test one (and believing you tested both is problematic!) To see why this rule exists, think about the algorithm to check for these errors:

  - Check if the input string does not have exactly three letters
    - If it does not then:
      - *print "Error: the input needs to be 3 letters"*
      - exit the program
  - Check if the input string is made up entirely of lowercase letters
    - If it does not then:
      - *print "Error: the input needs to be all lowercase letters"*
      - exit the program

Violating one input condition will cause the program to exit! This is also true of other situations where the program rejects the input, even if does not exit.

Test exactly at the boundary of validity. If a program requires between 7 and 18 things, you should have test cases with 6, 7,8, 17, 18, and 19 things. You need to make sure that 6 and 19 are rejected while 7, 8, 17, and 18 are accepted. Testing exactly at the boundaries is important because of the common "off by one" mistake—maybe the programmer wrote < when he should have written <= or >= when he should have written > or something similar. If you test with values that are right at the boundary, you will find these mistakes.

However, testing is not just about checking error handling. You want to make sure that the algorithm correctly handles valid inputs too.

Handling Valid Inputs

Think carefully about whether or not there are any special cases where one particular input value (or set of values has to be treated unusually). For example, in poker an Ace is usually ranked the highest; however, it can have the lowest ranking in an "Ace Low Straight" (5 4 3 2 A). If you are testing code related to poker hands, you would want to explicitly test this case, since it requires treating an input value differently from normal.

Think carefully about the requirements, and consider whether something could be misinterpreted, easily mis-implemented, or have variations which could seem correct. Suppose your algorithm works with sequences of decreasing numbers. You should test with a sequence like 7 6 6 5 4, which has two equal numbers in it. Checking equal numbers is a good idea here, since people might have misunderstood whether the sequence is strictly decreasing (equal numbers don't count as continuing to decrease) or non-increasing (equal numbers do count).

Think about types. What would happen if the programmer used the wrong type in a particular place? This could mean that the programmer used a type which was too small to hold the required answer (such as a 32-bit integer when a 64-bit integer is required), used an integer type when a floating point type is required, or used a type of the wrong signedness (signed when unsigned is required or vice versa).

Consider any kind of off-by-one error that the programmer might have been able to make. Does the algorithm seem like it could involve counting? What if the programmer was off by one at either end of the range she counted over? Does it involve numerical comparison? What if < and <= (or > and >=) were mixed up?

Whenever you have a particular type of problem in mind, think about how that mistake would affect the answer relative to the correct behavior, and make sure they are different. For example, suppose you are writing a program that takes two sequences of integers and determine which one has more even numbers in it. You are considered that the programmer might have an off-by-one error where he accidentally misses the last element of the sequence. Would this be a good test case?

- Sequence 1: 2 3 5 6 9 8
- Sequence 2: 1 4 2 8 7 6

This would not be a good test case for this particular problem. If the program is correct, it will answer "Sequence 2" (which has 4 compared to 3). However, if the algorithm mistakenly skips the last element, it will still answer "Sequence 2" (because it will count 3 elements in Sequence 2, and 2 elements in Sequence 1). A good test case to cover this off-by-one-error would be

- Sequence 1: 2 3 5 6 9 8
- Sequence 2: 1 4 2 8 7 3

Now a correct program will report a tie (3 + 3) and a program with this particular bug will report Sequence 2 as having more even numbers.

Consider all major categories of inputs, and be sure you cover them.

- For numerical inputs, these would generally be negative, zero, and positive. One is also usually a good number to be sure you cover.
- For sequences of data, your tests should cover an empty sequence, a single element sequence, and a sequence with many elements.
- For characters: lowercase letters, uppercase letters, digits, punctuation, spaces, non-printable characters.
- For many algorithms, there are problem-specific categories that you should consider. For example, if you are testing a function related to prime numbers (e.g., isPrime), then you should consider prime and composite (not prime) numbers as input categories to cover.
- When you combine two ways to categorize data, cover all the combinations. For example, if you have a sequence of numbers, you should test with an empty list, a one element sequence with 0, a one element sequence with a negative number, a one element sequence with a positive number, and have each of negative, zero, and positive numbers appearing in your many-element sequences.

An important corollary of the previous rules is that if your algorithm gives a set of answers where you can list all possible ones (true/false, values from an enum, a string from a particular set, etc.), then your test cases should ensure that you get every answer at least once. Furthermore, if there are other conditions that you think are important, you should be sure that you get all possible answers for each of these conditions. For example, if you are getting a yes/no answer, for a numeric input, you should test with a negative number that gives yes, a negative number that gives no, a positive number that gives yes, a positive number that gives no, and zero (zero being only one input, will have one answer).

All of this advice is a great starting point, but the most important thing for testing is to think carefully—imagine all the things that could go wrong, think carefully about how to test them, and make sure your test cases are actually testing what you think they are testing.

Generating Test Cases

One difficulty with testing arises from the fact that you want to test the cases you did not think of—but if you do not think of these cases, then how do you know to write tests for them? One approach to such a problem is to generate the test cases according to some algorithm. If the function we are testing takes a single integer as input, we might iterate over some large range (say -1,000,000 to 1,000,000) and use each integer in that range as a test case.

Another possibility is to generate the test cases (pseudo-)randomly (called, unsurprisingly, *random testing*). Note that *pseudo-random* means that the numbers look random (no "obvious" pattern) to a human, but are generated by an algorithm which will produce the same answer each time if they start from the same initial state (called a "seed"). Random testing can be appealing as it can hit a wide range of cases quickly that you might not think of at all. For example, if your algorithm has 6 parameters, and you decide you want to test 100,000 possibilities for each parameter in all combinations, you will need 100,000^6 = 10^30 test cases—even if you can do 10 trillion test cases per second (which would be beyond "fast" by modern standards), they will take about 3 million years to run! With random testing, you *could* run a few thousands or millions of cases, and rely on the Law of Large Numbers to make it likely that you encounter a lot of varieties of relationships between the parameters.

One tricky part about generating test cases algorithmically is that we need some way to verify the answer—and the function we are trying to test is what computes that answer. We obviously cannot trust our function to check itself, leaving us a seeming "chicken and egg" problem. In a very few cases, it may be appealing to write two versions of the function which can be used to check each other. This approach is appropriate when you are writing a complex implementation in order to achieve high performance, but you could also write a simpler (but slower) implementation whose correctness you would more readily be sure of. Here, it makes sense to implement both, and test the complex/optimized algorithm against the simpler/slower algorithm on many test cases.

We often test properties of the answer to see that it correct. For example, if we are testing an algorithm to compute the square root of a number, we can check that the answer times itself produces the original input (that is that the square root of *n*-squared equals *n*—or is within the expected error given the precision of the numbers involved.) Testing this property of the answer assures us that it was correct without requiring us to know (or be able to compute by other means) what it is. The previous technique works well for invertible functions (that is, where we can apply some inverse operation that should result in the original input), but many programming problems do not fit into this case.

We may, however, be able to test other properties of the system to increase our confidence that it is correct. For example, imagine that we are writing software to simulate a network, which routes messages from sources to destinations (the details of how to implement this are beyond the skills we have learned so far, but that is not important for the example). Even without knowing the right answer, we can still test that certain properties of the system are obeyed: every message we send should be delivered to the proper destination, that delivery should happen exactly one time, no improper destinations should receive the messages, the delivery should occur in some reasonable time frame, and so on.

Checking these properties does not check that the program gave the right answer (e.g., it may have routed the message by an incorrect but workable path), but it checks for certain classes of errors on those test cases. As with all test cases, this increases our confidence that the program is right, but does not prove that it is right. Of course, we would need to test the other aspects of the answer in other ways—which may involve looking at the details of fewer answers by hand to ensure all details are right.

This approach requires development of code which is not part of the main application—a program which not only sends the requests into the network model, but also tracks the status of those requests and checks for the proper outcomes. This additional program is called a *test harness*—it is a program you write to run and test the main parts of your code. Developing such infrastructure can be time consuming, but is often a good investment of time, especially for large projects.

Asserts

In any form of testing, it can be useful to not only check the end result, but also that the *invariants* of the system are maintained in the middle of its execution. An invariant is a property that is (or should be) always true at a given point in the program. When you know an invariant that should be true at a given point in the code, you can write an *assert statement*, which checks that it is true. *assert expr* checks that *expr* is true. If it is true, then nothing happens, and execution continues as normal. However, if *expr* is false, then it prints a message stating that an assertion failed, and raises an error.

As an example, suppose we want to write a function that prints the prime factors of a given number. For example, printFactors(12) would print "2 * 2 * 3" because those are the prime numbers that can be multiplied together to get 12 (such a factorization is unique for any positive integer). We have written the Python code for this algorithm below, and included two assert statements which check the invariants of our algorithm:

```
def printFactors(n):
    if n <= 1:
        return
    p = 2
    while not isPrime(n):
        assert isPrime(p)   # p should always be prime
        assert p < n        # p should always be less than n
        if n % p == 0:
            print(p, end=' * ')
            n = int(n / p)
        else:
            p = nextPrimeAfter(p)   # helper function to get next prime
            pass
        pass
    print(n)
    pass
```

Notice the two *assert* statements in this code. The first, on line 6, checks that p is prime (using our *isPrime* function). This fact should always be true here—if it is not, our algorithm is broken (we may end up printing non-prime factors of the numbers, which is incorrect). How could such an error occur? One possibility would be a bug in *nextPrimeAfter*—the helper function we have written to find the next prime after a particular prime. Another possibility would be if we accidentally modify p in a way that we do not intend to. Of course, if our code is correct, neither of these will happen, and the *assert* will do nothing, but the point is to help us if we do make a mistake.

The second *assert* (line 7) checks another invariant of our algorithm: *p* should always be less than *n* inside this loop (think about why...). As with the first *assert*, we hope that it has no effect—just checking that the expression is always true—but if something is wrong, it will help us detect the problem.

Note that *assert* statements are an example of the principle that if our program has an error we want it to *fail fast*—that is, we would rather the program crash as soon after the error occurs as possible. The longer the program runs after an error occurs, the more likely it is to give incorrect answers and the more difficult it is to debug. Ideally, when our program has an error, we will have an assert fail immediately after it, pointing out exactly what the problem is and where it lies.

In almost all cases, giving the wrong answer (due to an undetected error) is significantly worse than the program detecting the situation and crashing. To see this tradeoff, consider the case of software to compute a missile launch trajectory. If there is an error in such software, would you rather it give incorrect launching instructions, or print a message that an error occurred and abort?

Many novice and intermediate programmers worry that *asserts* will slow their program down. In general, the slowdown is negligible, especially on fast modern computers. In many situations, 1–2% performance just does not matter—do you really care if you get your answer in 0.1 seconds versus 0.11 seconds? However, there are performance critical situations where ever bit of speed matters. For these situations, you can pass the -O option to python when you run it, which turns on optimizations and turns off debugging features such as assertions. Note that performance critical code is the domain of expert programmers who also have a deep understanding of the underlying hardware—such considerations are well beyond the scope of this specialization. We also note that if you need to write performance critical code, Python is generally a poor choice, as all known implementations of Python impose a 10x-20x slowdown compared to code written and compiled in languages which target high performance, such as C or C++.