Black Box Testing

/run/media/lijunjie/资料/Learning Video/writing-running-fixing-code/03_testing-and-debugging/01_testing/02_black-box-testing_instructions.html

The testing methodology that most people think of first is *black box testing*. In black box testing, the tester considers only the expected behavior of the function—not any implementation details—to devise test cases. The lack of access to the implementation details is where this testing methodology gets its name—the function's implementation is treated as a "black box" which the tester cannot look inside.

Black box testing does not mean that the tester thinks of a few cases in an *ad hoc* manner and calls it a day. Instead, the tester—whose goal is to craft test cases that are likely to expose errors—should contemplate what cases are likely to be error prone from the way the function behaves. For example, suppose you needed to test a function to sum a list of integers. Without seeing the code, what test cases might you come up with?

We might start with a simple test case to test the basic functionality of the code—for example, seeing that the function gives an answer of 15 when given an input of *{1,2,3,4,5}* . However, we should also devise more difficult test cases, particularly those which test *corner cases* —inputs that require specific behavior unlike other cases. In this example, we might test with the empty list (that is, the list with no numbers in it), and see that we get 0 as our answer. We might make sure to test with a list that has negative numbers in it, or one with many very large numbers in it. You should contemplate what sorts of errors these test cases might expose.

Observe that we were able to contemplate good test cases for our hypothetical problem without going through Steps 1–5. You can actually come up with a set of black box tests for a problem before you start on it. Some programmers advocate a test-first development approach. One advantage is that if you have a comprehensive test-suite written before you start, you are unlikely to skimp on testing after you implement your code. Another advantage is that by thinking about your corner cases in advance, you are less likely to make mistakes in developing and implementing your algorithm.