Black and White Box Testing

Black Box Testing

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.

Testing Methods

Test-Driven Development

1.5.3 Test-Driven Development

White Box Testing

Another testing methodology is *white box* testing. Unlike black box testing, white box testing involves examining the code to devise test cases. One consideration in white box tests is *test coverage*—a description of how well your test cases cover the different behaviors of your code.

Note that while white box and black box testing are different, they are not mutually exclusive, but rather complementary. One might start by forming a set of black box test cases, implement the algorithm, then create more test cases to reach a desired level of test coverage.

```
def aFunction(a: int, b: int, c: int) -> int:
    answer = 0
    if a < b:
        answer = b - a
    if b < c:
        answer = answer * (b -c)
    else:
        answer = answer + 42
    return ans
```

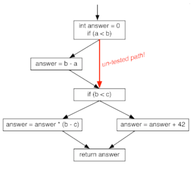
We will discuss three levels of test coverage: statement coverage, decision coverage, and path coverage. To illustrate each of these types of coverage, we will use the (contrived) code example shown above.

Statement coverage means that every statement in the function is executed. In our example, if we were to use only one test with *a=1, b=2,* and *c=3*, we would not have statement coverage. Specifically, this test case (by itself) does not execute line 10 of the code (the body of the **else**). Our lack of coverage indicates that we have not adequately tested our function—it has behaviors which our test case has not exercised.

Statement coverage is a minimal starting point, but if we want to test our code well, we likely want a stronger coverage metric. To see how statement coverage falls short, notice that we have not tested any cases where the a < b test evaluates to false—that is, a case where we do not enter the body of the **if** statement (line 4).

A stronger level of coverage is *decision coverage*—in which all possible outcomes of decisions are exercised. For boolean tests, this means we construct a test case where the expression evaluates to true, and another where it evaluates to false. If we have a **switch/case** statement, we must construct at least one test case per choice in the case statement, plus one for the default case, even if it is implicit—that is, if we do not write it down, and it behaves as if we wrote **default: break;**.

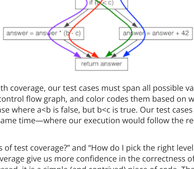
To understand decision coverage more fully, it helps to visualize it. In order to visualize decision coverage, we need to understand the concept of a *control flow graph* (often abbreviated "CFG"). A control flow graph is a directed graph (in the mathematical sense) whose nodes are basic blocks (*boxes*) and whose edges represent the possible ways the control flow arrow can move (*arrows*). A basic block is a contiguous sequence of statements which must be executed in an "all-or-nothing" manner; the execution arrow cannot jump into nor out of the middle of a basic block. The CFG shows how the control flow arrow can move from one basic block to the next.



The figure above shows the control flow graph for our example function from the previous figure. The very first basic block has a lone arrow coming into it, indicating that it is the start—the execution arrow can only reach it by calling the function. This first basic block has two statements: the declaration/initialization of answer and the **if** statement which tests if a < b. The basic block ends here because the execution arrow could go two places—indicated by the two edges leaving this block. The final block has no arrows going out of it; when the execution arrow moves to the end of this block, there is nowhere else to go within this function. At this point, the function is done and returns its answer. Note that we draw the CFG for one function at a time. We could also draw how the execution arrow moves *between* functions; that is called a *call graph*.

Decision coverage corresponds to having a suite of test cases which covers every edge in the graph. In our graph, if a < b is true, we will take the left edge coming out of the first block (corresponding to going into the "then" clause of the **if/else**). If it is false, we will take the right edge, which skips over this clause, and as there is no **else** clause, goes to the next if statement. This second edge—representing the path of our execution arrow when a < b is false—is highlighted in red in the figure above, since it represents the shortcomings of our test cases so far with respect to decision coverage.

We can obtain decision coverage on this function by adding a test case where the execution arrow traverses this edge of the control flow graph—or put a different way, where the a < b decision evaluates to false. An example of this test case would be a=5, b=2, and c=1.



An even stronger type of test coverage is *path coverage*. To achieve path coverage, our test cases must span all possible valid paths through the control flow graph (following the direction of the arrows). The figure above shows the four paths through our example control flow graph, and color codes them based on which of our test cases cover them. Note that there is one path (shown in red) which we have not yet covered. This path corresponds to the case where a<b is false, but b<c is true. Our test cases which gave us decision coverage tested each of these conditions separately, but none of the tests tested both of these together at the same time—where our execution would follow the red path. We can add another test case (e.g., a=5, b=22, c=99) to gain path coverage.

At this point you may be wondering "Why do we have all of these levels of test coverage?" and "How do I pick the right level of coverage for what I am doing?" Software engineers define and discuss multiple levels of test coverage because the higher levels of coverage give us more confidence in the correctness of our code, but at the cost of more test cases. While our example only requires 2, 3, and 4 test cases for the three levels of coverage we discussed, it is a simple (and contrived) piece of code. The number of paths through the control flow graph is exponential in the number of conditional choices—if we have 6 **if/else** statements one after the other, then there are 64 possible paths through the control flow graph. If we increase the number of **if/else** statements from 6 to 16, then the number of paths grows to 65536—quite a lot!

So how do you pick the right level of test coverage? As with many things, the answer is "it depends". The first aspect of this decision is "how confident do you need to be in your code?" If you are doing preliminary testing of your algorithm by hand in Step 4 (and will test the implemented algorithm once you have translated it to code), then statement coverage is a reasonable choice. Here, testing more cases is quite time consuming (you are doing them by hand), and you will do more test cases later (once it is translated to code and compiled, where you can have the computer run the tests).

By contrast, if you are testing a piece of critical software which will be deployed when you finish testing, only achieving statement coverage would be woefully insufficient. Here, you would likely want to aim for more than just the minimum to achieve path coverage. Unfortunately, no amount of testing can ensure the code is correct—it can only increase our confidence that it is. If you require absolute certainty that the code is correct, you must formally prove it (which is beyond the scope of this course).

White Box Testing