D. Testing and Debugging

We hate to bring this up, but Dr. Pangloss was wrong. We do not live in "the best of all possible worlds." There are some places where it rains too little, and others where it rains too much. Some places are too cold, some too hot, and some too hot in the summer and too cold in the winter. The "M" in MIT stands for Massachusetts rather than Maui. Sometimes the stock market goes down—a lot. And, worst of all, our programs don't always function properly the first time we run them.

Books have been written about how to deal with this last problem, and there is a lot to be learned from reading these books. However, in the interests of providing you with some hints that might help you get that next problem set in on time, this chapter provides a Spark Notes version of the topic. While all of the programming examples are in Python, the general principles are applicable to getting any complex system to work.

Testing is the process of running a program to try and ascertain whether or not it works as intended. Debugging is the process of trying to fix a program that one knows does not work as intended.

Testing and debugging are not processes that one should begin to think about after a program has been built. Good programmers design their programs in ways that make them easier to test debug. The key to doing this is breaking the program up into components that can be implemented, tested, and debugged independently of each other. At this point in the book, we have discussed only one mechanism for modularizing programs, the function. So, for now, all of our examples will be based around functions. When we get to other mechanisms, in particular classes, we will return to some of the topics covered in this chapter.

The first step in getting a program to work is getting the language system to agree to run it. I.e., eliminating syntax and static semantic errors that can be detected without running the program. If you haven't gotten past that point in your programming, you're not ready for this chapter. Spend a bit more time working on small programs, and then come back.

D.1 Testing

The most important thing to say about testing is that its purpose is to show that bugs exist, not to show that a program is bug-free. To quote Edsger Dijkstra, "Program testing can be used to show the presence of bugs, but never to show their absence!"[1] Or, as Albert Einstein reputedly once said, "No amount of experimentation can ever prove me right; a single experiment can prove me wrong."

Why is this so? Even the simplest of programs has billions of possible inputs. Consider, for example, a program that purports to meet the specification:

```
def isBigger(x, y):
    """requires: x and y are ints
       returns True if x is less than y and False otherwise."""
```

Running it on all pairs of `ints` would be, to say the least, tedious. The best we can do is to run it on pairs of `ints` that have a reasonable probability of producing the wrong answer if there is a bug in the program.

[1] "Notes On Structured Programming," Technical University Eindhoven, T.H. Report 70-WSK-03, April 1970.

The key to testing is finding a collection of inputs, called a **test suite**, that have a high likelihood of revealing bugs, yet don't take too long to run. The key to doing this is partitioning the space of all possible inputs into subsets that provide equivalent information about the correctness of the program, and then constructing a test suite that contains one input from each partition. Of course this is far easier said than done. Typically, people rely on heuristics based on exploring different paths through some combination of the code and the specifications. Heuristics based on exploring paths through the code fall into a class called **glass box testing**. Heuristics based on exploring paths through the specification fall into a class called **black box testing**.

D.1.2. Black Box Testing

In principle, black box tests are constructed without looking at the code to be tested. Black box testing allows testers in implementers to be drawn from separate populations. When those of us who teach programming courses generate test cases for the problem sets we assign students, we are developing black box test suites. Developers of commercial software often have quality assurance groups that are largely independent of development groups.

This independence reduces the likelihood of generating test suites that exhibit mistakes correlated with mistakes in the code. Suppose, for example, that the author of a program made the implicit, but invalid, assumption that the program would never be invoked with a certain class of inputs. If the same person constructed the test suite for the program, he or she would likely repeat the mistake.

Another positive feature of black box testing is that it is robust with respect to implementation changes. Since the test data is generated without knowledge of the implementation, it need not be changed when the implementation is changed.

As we said earlier, a good way to generate black box test data is to explore paths through a specification. Consider, the specification:

```
def sqrt(x, epsilson):
    """requires: x, epsilon floats
                 x >= 0
                 epsilon > 0
        returns res s.t. x-epsilon <= res*res <= x+epsilon"""
```

There seems to be only two distinct paths through this specification. One corresponding to x = 0 and one corresponding to x > 0. However, common sense tells us that while this it is necessary to test these two cases, it is hardly sufficient.

Another good heuristic is testing boundary conditions. When looking at lists, it often means looking at the empty list, a list with exactly one element, and a list containing lists. When dealing with numbers, it typically means looking at very small and very large values as well as "typical values." For `sqrt`, it might make sense to try at least values of `x` and `epsilon` similar to those in the accompanying table.

The first four rows are intended to represent typical cases. Notice that the values for `x` include a perfect square, a number less than one, and a number with an irrational square root. If any of these tests fail, there is a bug in the program that needs to be fixed.

x	epsilon
0.0	0.0001
25.0	0.0001
0.5	0.0001
2.0	0.0001
2.0	1.0/2.0**64.0
1.0/2.0**64	1.0/2.0**64.0
2.0**64.0	1.0/2.0**64.0
1.0/2.0**64.0	2.0**64.0
2.0**64.0	2.0**64.0

The remaining rows test extremely large and small values of **`x`** and **`epsilon`**. If any of these tests fails, something needs to be fixed. Perhaps there is a bug in the code that needs to be fixed, or perhaps the specification needs to be changed to be easier to meet. It might for example, be unreasonable to expect to find an approximation of a square root when **`epsilon`** is ridiculously small.

Another important boundary condition to think about is aliasing. Consider, for example, the following code:

```
def copy(L1, L2):
    """requires: L1, L2 are lists
       mutates L2 to be a copy of L1"""
    while len(L2) > 0:
        L2.pop()
    for e in L1:
        L2.append(e)
```

It will work most of the time, but not when `L1` and `L2` refer to the same list. Any set of test data that did not include a call of the form `copy(L, L)`, would not reveal the bug.

D.1.3. Glass Box Testing

Black box testing should never be skipped, but it is rarely sufficient. Without looking at the internal structure of the code, it is impossible to know which test cases are likely to provide new information. Consider, the following trivial example:

```
def isPrime(x):
    """requires: x is a non-negative int
       returns True if x is prime; False otherwise"""
    assert type(x) == int and x >= 0
    if x <= 2:
        return False
    for i in range(2, x):
        if x%i == 0:
            return False
    return True
```

When looking at the code, it is clear that 0, 1, and 2 are treated as special cases, and therefore need to be tested. Without looking at the code, one might not try `isPrime(2)`, and would fail to detect the error[2].

[2] 0 and 1 are indeed special cases, since they are not primes. The bug was writing `<=2` instead of `<2` or `<=1`.

Glass box test suites are usually much easier to construct than black box suites. Specifications are usually incomplete and often pretty sloppy, making it a challenge to estimate how thoroughly a black box test suite explores the space. In contrast, the notion of a path through code is well defined, and it is relatively to understand how thoroughly one is exploring the space. There are, in fact, commercial tools that can be used to objectively measure the completeness of glass box tests.

A glass box test suite is **path complete** if it exercises every potential path through the program. This is typically impossible to achieve because it depends upon the number of times each loop is executed and the depth of each recursion. Furthermore, even a path complete test suite does not guarantee that all bugs will be exposed. Consider:

```
def abs(x):
    """requires: x is an int
       returns x if x>=0 and -x otherwise"""
    assert type(x) == int
    if x < -1:
        return -x
    else:
        return x
```

The specification suggests that there are two possible cases, `x` is either negative or it isn't. This suggests that the set of inputs `{2, -2}` is sufficient to explore all paths in the specification. This test suite has the additional nice property of forcing the program through all of it paths, so it looks like a complete glass box suite as well. The only problem is that this test suite will not expose the fact that `abs(-1)` will return `-1`.

More generally, glass box testing is usually better at uncovering errors of commission than at uncovering errors of omission. A testing strategy based entirely on exploring paths through a program is not likely to uncover a missing path, a fairly common programming error.

Despite the limitations of glass box testing, there are a few rules of thumb that are usually worth following:

- Exercise both branches of all `if` statements.
- Make sure that each `except` clause is executed.
- For each `for` loop, have test cases in which
    - The loop is not entered (e.g., if there is statement such as `for e in L`, make sure that it is tested with `L = []`),
    - The body of the loop is executed exactly once, and
    - The body of the loop is executed more than once.
- For each `while` loop,
    - Look at the same kinds of cases as when dealing with `for` loops, and
    - Include test cases corresponding to all possible ways of exiting the loop (e.g., for `while len(L) > 0 and not L[i] == e`, find cases where each of the conjuncts hold).
- For recursive functions, include test cases that cause the function to return with no recursive calls, exactly one recursive call, and more than one recursive call.

D.1.4. Conducting tests

Testing is often thought of as occurring in two phases. One should always start with **unit testing**. During this phase one constructs and runs tests designed to ascertain whether individual modules (e.g., functions or classes) work properly. This is followed by **integration testing**, which is designed to ascertain whether the program as a whole behaves as intended. In practice one cycles through these two phases, since failures during integration testing lead to making changes to individual modules.

Integration testing is almost always more challenging than unit testing. One reason for this is that the intended behavior of an entire program is often considerably harder to characterize than the intended behavior of each of its parts. For example, characterizing the intended behavior of a word processor is considerably more challenging than characterizing the behavior of a function that counts the number of characters in the document. Problems of scale can also make integration testing difficult. It is not unusual for integration tests to take hours or even days to run.

In industry, the testing process is often highly automated. Testers[3] do not sit a terminals typing inputs and checking outputs. Instead, they use **test drivers** that autonomously

- Set up the environment needed to invoke the program (or unit) to be tested.
- Invoke the program to be tested with a pre-defined or automatically generated sequence of inputs,
- Save the results of these invocations,
- Check the acceptability of the results of the tests, and
- Prepare an appropriate report.

During unit testing, one often needs to build **stubs** as well as drivers. During unit testing drivers simulate parts of the program that use the unit being tested, whereas stubs simulate parts of the program used by the unit being tested. Ideally, a stub should

- Check the reasonableness of the environment and arguments supplied by the caller (calling a function with inappropriate arguments is a common error),
- Modify arguments and global variables in manner consistent with the specification, and
- Return values consistent with the specification.

Building adequate stubs is often a challenge. If the unit the stub is replacing is intended to perform some complex task, building a stub that performs actions consistent with the specification may be tantamount to writing the program the stub is designed to replace. One way to surmount this problem is to limit the set of arguments accepted by the stub, and create a table that contains the values to be returned for each combination of arguments to be used in the test suite.

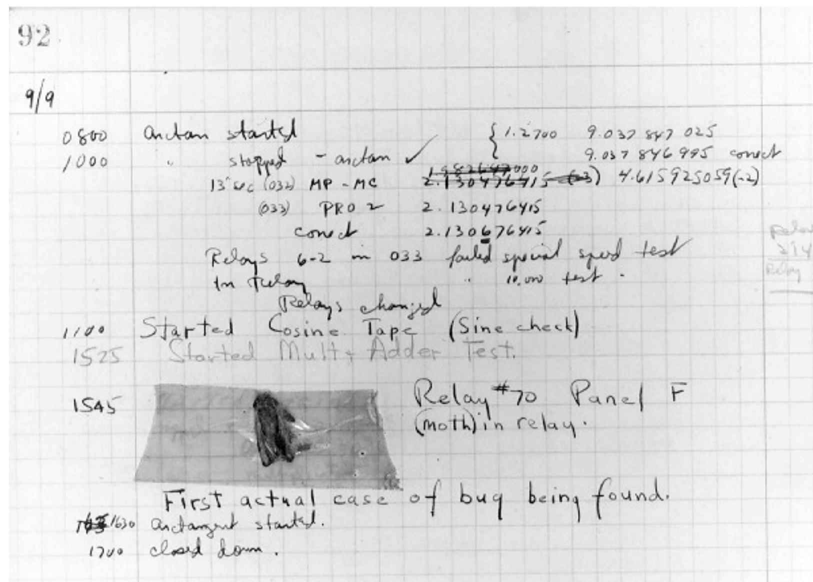
One attraction of automating the testing process is that it facilitates **regression testing**. As one goes about debugging a program, it is all too common to install a “fix” that breaks something that used to work. Whenever any change is made, not matter how small, one should check that the program sill passes all of the tests that it used to pass. However, doing this is practical only if it is not terribly labor intensive to rerun old tests.

---

[3] Or, for that matter, those who grade problem sets in large programming courses.

D.2. Debugging

There is a charming urban legend about how the process of fixing flaws in software came to be known as debugging. The photo below is of a September 9, 2007 page in a laboratory book from the group working on the Mark II Aiken Relay Calculator at Harvard University.



Some have claimed that the discovery of that unfortunate moth trapped in the Mark II, led to the use of the phrase debugging. However the wording, "First actual case of a bug being found," suggests that a less literal interpretation of the phrase was already common. Grace Murray Hopper, a leader of the Mark II project, made it clear that the term "bug" was already in wide use to describe problems with electronic systems during World War II. Well prior to that, *Hawkin's New Catechism of Electricity*, an 1896 electrical handbook, included the entry, "The term 'bug' is used to a limited extent to designate any fault or trouble in the connections or working of electric apparatus." In common English usage the word "bugbear," means "anything causing seemingly needless or excessive fear or anxiety."[4] Shakespeare seems to have shortened this to "bug," when he had Hamlet kvetch about "bugs and goblins in my life."[5]

The use of the word bug sometimes leads people to ignore the fundamental fact that if you wrote a program and it has a "bug," you messed up. Bugs do not crawl unbidden into flawless programs. If your program has a bug, it is because you put it there. Bugs do not breed in programs. If your program has multiple bugs, it is because you made multiple mistakes.

Runtime bugs can be categorized along two dimensions:

1. **Overt → covert**: An overt bug has an obvious manifestation, e.g., the program crashes with an unhandled exception or takes far longer (maybe forever) to run than it should. A covert bug has no obvious manifestation. The program may run to conclusion with no problem—other than providing an incorrect answer. Many bugs fall between the two extremes, and whether or not the bug is overt depends upon how carefully one examines the behavior of the program.

[4] *Webster's New World College Dictionary.*

[5] Act 5, scene 2.

2. **Persistent → intermittent**: A persistent bug occurs every time one runs the program. An intermittent bug occurs only some of the time, even when the program is run on the same inputs and seemingly under the same conditions.

The best kinds of bugs to have are overt and persistent. Developers can be under no illusion about the advisability of deploying the program. And if someone else is foolish enough to attempt to use it, they will quickly discover their folly. Perhaps the program will do something horrible before crashing, e.g., delete files, but at least the user will have reason to be worried (if not panicked). As we shall discuss later, good programmers try to write their programs in such a way that programming mistakes lead to bugs that are both overt and persistent. This is often called **defensive programming**.

The next step into the pit of undesirability is bugs that are overt but intermittent. One can live in a fool's paradise for a period of time, and maybe get so far are as deploying a system incorporating the flawed program, but "their foot shall slide in due time."[6] Sooner or later the bug will become manifest. If the conditions prompting the slide are easily reproducible, it is often relatively easy to track down and repair the problem. If the conditions provoking the bug are not clear, life is much harder. An air traffic control system that computes the correct location for planes almost all of the time, is far more dangerous than one that makes obvious mistakes all the time.

Programs that fail in covert ways are often highly dangerous. Since they are not apparently problematical, people use them and trust then to do the right thing. Increasingly, society relies on software to perform critical computations that are beyond the ability of humans to carry out or even check for correctness. Therefore, a program can provide undetected fallacious answer for long periods of time. Such programs can, and have, caused a lot of damage. A program that evaluates the risk of a bond portfolio and confidently spits out the wrong answer, can get a bank (and perhaps all of society) into a lot of trouble. A radiation therapy machine that delivers a little more or a little less radiation than intended can be the difference between life and death for a person with cancer. A program that makes a covert error only occasionally may or may not wreck less havoc than one that always commits such an error. Bugs that are both covert and intermittent are usually the hardest to fix.

D.2.1. Learning to Debug

Debugging is a learned skill. Nobody does it well instinctively. The good news is that it's not hard to learn, and is a transferable skill. The same skills used to debug software can be used to find out what is wrong with other complex systems, e.g., laboratory experiments or sick humans.

For at least four decades people have been building tools called debuggers, and there are debugging tools built into Idle. These are supposed to help people find bugs in their programs. They can help, but only a little. What's much more important is how you approach the problem. Many experienced programmers don't even bother with them. Most programmers say that the two most important debugging tools is the print statement.

Debugging starts when testing has demonstrated that the program behaves in unexpected ways. Debugging is the process of searching for an explanation of that behavior. The key to being consistently good at debugging, is being systematic in conducting that search.

One starts by studying the available data. This includes the test results and the program text. Study all of the test results. Not only the tests that revealed the presence of a problem, but also

[6] Deuteronomy 32:35.

those tests that seemed to work perfectly. Trying to understand why one test worked and another did not is often illuminating. When looking at the program text, keep in mind that you don't completely understand it. If you did, there wouldn't be a bug.

Next, form a hypothesis that you believe to be consistent with all the data. The hypothesis could be as narrow as "if I change line 403 from x = y to y = x, the problem will go away" or as broad as "my program is not terminating because I have the wrong test in some while loop."

Next, design and run a repeatable experiment with the potential to refute the hypothesis. For example, one might put a print statement before and after each while loop. If these are always paired, than the hypothesis that a while loop is causing non-termination has been refuted. Decide before running the experiment how you would interpret various possible results. If you wait until after you run the experiment, you are more likely to fall prey to wishful thinking.

Finally, keep a record of what experiments you have run. This is particularly important. If one isn't careful, it is easy to waste countless hours trying the same experiment (or more likely an experiment that looks different but will give you the same information) over and over again.

Designing the experiment

Think of debugging as a search process, and each experiment as an attempt to reduce the size of the search space. It's often useful to think of the size search space as being the product of the amount of code to be examined and amount of testing needed to provoke the bug.

Let's look at a contrived example to see how one might go about debugging it. Imagine that you wrote the code in Figure Silly, and that you are so confident of your programming skills that you put it up on the web—without testing it. Suppose further that you receive an email saying, "I tested your !!**! program on the following 1000 string input, and it printed Yes. Yet any fool can see that it is not a palindrome. Fix it!"

```
def isPal(x):
    """requires x is a list
       returns True if the list is a palindrome; False otherwise"""
    assert type(x) == list
    temp = x
    temp.reverse
    if temp == x:
        return True
    else:
        return False

def silly(n):
    """requires: n is an int > 0
    Gets 3 inputs from user
    Prints 'Yes' if the inputs are a palindrome; 'No' otherwise"""
    assert type(n) == int and n > 0
    for i in range(n):
        result = []
        elem = input('Enter element: ')
        result.append(elem)
    print(result)
    if isPal(result):
        print('Yes')
    else:
        print('No')
```

Figure Silly

You could try and test it on the supplied input. But it might be more sensible to begin by trying it on something smaller. In fact, it would make sense to test it on a minimal non-Palindrome, e.g.,

```
>>> silly(2)
Enter element: a
Enter element: b
```

The good news is that it fails even this simple test, so you don't have to type in a thousand strings. The bad news is that you have no idea why it failed.

In this case, the code is small enough that you can probably stare at it, and find the bug. However, let's pretend that it is too large to do this, and start to systematically reduce the search space.

Often the best way to do this is to conduct a binary search. Find some point about halfway through the code, and devise an experiment that will allow you to decide if there is problem, which might be related to the symptom, before that point. (Of course, there may be problem after as well, but it is usually best to hunt down one problem at a time.) In choosing such a point, look for a place where there are some easily examined intermediate values that provide useful information. If an intermediate value is not what you expected, there is probably a problem that occurred prior to that point in the code. If the intermediate values all look fine, the bug probably lies somewhere later in the code. This process can be repeated until you have narrowed the region in which a problem is located to a few lines of code.

Looking at silly, the halfway point is just before the line if `isPal(result)`. The obvious thing to check is whether `result` has the expected value, `['a', 'b']`. We this by inserting the statement `print(result)`. When the experiment is run, the program prints `['b']`, suggesting that something has already gone wrong. The next step is to print `result` roughly halfway through the loop. This quickly reveals that `result` is never more than one element long, suggesting that the initialization of `result` needs to be moved outside the `for` loop.

Let's try that, and see if `result` has the correct value after the `for` loop. It does, but unfortunately the program still prints `Yes`. Now, we have reason to believe that a second bug lies below the `print` statement. So, let's look at `isPal`. The line `if temp == x:` is about halfway through that function. So, we insert the line `print(temp, x)` before that line. When we run the code, we see that `temp` has the expected value, but `x` does not. Moving up the code, we insert a print statement after the line `temp = x`, and everything looks fine. We have now narrowed the bug to one line, `temp.reverse(x)`, which unexpectedly changed the value `x`. An aliasing bug has bitten us.

When the going gets tough

Joseph P. Kennedy, father of President Kennedy, said, "When the going gets tough, the tough get going."[7] But he never debugged a piece of software. This subsection contains a few pragmatic hints about what do when the debugging gets tough.

- Look for the usual suspects, e.g., have you
    - Passed arguments to a function in the wrong order,
    - Misspelled an identifier, e.g., typed a lower case letter when you should have typed an upper case one,
    - Failed to reinitialize a variable,

[7] He also said, "Don't buy a single vote more than necessary. I'll be damned if I'm going to pay for a landslide."

    - Tested that two floating point values are equal (==) instead of nearly equal (recall that the value of `math.sqrt(3)*math.sqrt(3) == 3` is `False`.),
    - Tested for object equality when you meant value equality (or vice versa),
    - Forgotten that some built in function has a side effect,
    - Created an unintentional alias, or
    - Made any other mistake that is typical for you.
- Keep in mind that the bug is probably not where you think it is. If it were, you would probably have found it long ago. One practical way to go about deciding where to look is asking where the bug cannot be. As Sherlock Holmes, said, "Eliminate all other factors, and the one which remains must be the truth."
- Try to explain the problem to somebody else. We all develop blind spots. It's often the case that merely attempting to explain the problem to another will lead you to see things you have missed. A good thing to try to explain is why the bug cannot be in certain places.
- Don't believe everything you read. In particular, don't believe the documentation. The code may not be doing what the comments suggest.
- Stop debugging and start writing documentation. This will help you approach the problem from a different perspective.
- Walk away, and try again tomorrow. This may mean that bug is fixed later than if you had stuck with it, but you will probably spend a lot less of your time looking for it, i.e., it is possible to trade latency for efficiency. (Students, this is an excellent reason to start work on programming problem sets earlier rather than later!)

And when you have found "the" bug

When you think you have fund a bug in your code, the temptation to start hacking up and testing a fix is almost irresistible. It is often better, however, to slow down a little. Remember that the goal is not to fix this bug, but to move rapidly and efficiently towards a bug-free program.

Ask yourself if this bug explains all the observed symptoms, or is it just the tip of the iceberg. If the latter, it may be better to think about taking care of this bug in concert with other changes. Suppose, for example, that you have discovered that the bug is the result of having accidentally mutated a `list`. You could circumvent the problem locally (perhaps by making a copy of the `list`), or you could consider using a `tuple` instead of a `list`, perhaps eliminating similar bugs elsewhere in the code.

Before making any change, try and understand the ramification of the proposed "fix." Will it break something else? Does it introduce excessive complexity? Does it offer the opportunity to tidy up other parts of the code? Also, always make sure that you can get back to where you are. There is nothing more frustrating than realizing that a long series of changes have left you further from the goal than when you started, and having no way to get back to where you started. Disk space is usually plentiful. Use it to store old versions of your program.

Finally, if there are many unexplained errors, you might consider whether finding and fixing bugs one at a time is even the right approach. Maybe you would be better off thinking about whether there is some better way to organize your program or some simpler algorithm that will be easier to implement correctly.