# Chapter 5

# Testing and Debugging

You've written it so it must work, right? By now you know that is not necessarily true. We all make mistakes. To be a successful programmer you need to be able to reliably find and fix your mistakes. This chapter deals with two related efforts: writing test cases to determine if your program is running correctly, and finding the source of the problem when testing reveals that something is wrong.

Python has two built-in systems to help you with debugging. You should not need either of these for short programs, but they come in handy as your programs become longer and more complex. Knowing how to use a debugger is a good skill for any programmer. Just don't fall into the trap of relying on a debugger instead of writing your code carefully.

## 5.1 Testing Your Code

Probably the most common mistake made by beginning programmers is insufficiently testing their code. Most of us need to believe what we are doing is correct or we'll never do anything. It is hard to surrender that certainty when it comes time for testing. Too many beginners are satisfied with one or two test cases; sometimes students even hand in code without ever running it and are surprised to hear that it doesn't work correctly. If you want to be a successful programmer, integrate a simple testing methodology into your coding practice. Far from slowing you down; you'll find that this helps you to develop good code faster than you were previously writing buggy code.

### When To Test

There are three issues with testing: when to test, which test cases to use, and what to do with the information if a test comes back with a different response than you expect. The *when* question is easy: test continually as you develop. Beginners tend to write an entire program and then test it. If you do that, when a test fails every line of the program is a potential source of the problem. If you test as you develop and are confident that the program is correct before you add a new function, if errors appear when you test after that addition you know the likely source of those errors.

For example, think back to Program 4.3.5, which asks the user to enter strings and says whether each string is a palindrome. To test this program we need a more precise statement of what it should do. Here is such a statement:

Write a program that repeatedly asks the user to input a string; the input loop is terminated by an empty string. For each non-empty string the program should strip off any punctuation characters and determine if what is left forms a palindrome. The program's response to a non-empty input $< s >$ should be "$< s >$ is a palindrome." or "$< s >$ is not a palindrome."

What follows is an outline of the program we wrote to solve this problem.

```
# This program reads strings from the user and
# says if they are palindromes: the same when
# read backwards as when read forwards

def StripPunctuation(s):
    # This returns a string just like s only all
    # of the non-letters are removed and the letters
    # are all changed to lower-case.

def Reverse(s):
    # This returns the reversal of string s:
    # if s is 'abc' this returns 'cba'.

def IsPalindrome(s):
    # This returns True if string s is a palindrome
    # and False if it is not.
    # It calls StripPunctuation to remove all of the
    # non-letters from s

def main():
    # This has the input loop.  It reads strings from
    # the user, tests if they are palindromes, and
    # prints the answer. The loop terminates when it
    # gets an empty string.

main()
```

The skeleton of Program 4.3.5

When we developed this program in section 4.3 we first wrote main() with a *stub*, or dummy function, for IsPalindrome(), then we wrote both IsPalindrome() and Reverse(), then finally StripPunctuation(). We would test the program in exactly these pieces: one set of tests to ensure that our main() loop is working correctly, one set of tests after completing IsPalindrome() and Reverse(), to ensure that the main functionality of our program is correct, and a final set of tests on the completed program.

### Test Cases

Picking out test cases for a new component of your program involves thinking about what it does. There are three broad categories of test cases:

- Typical cases

- Boundary cases
- Extreme cases

*Typical cases* are test cases that exercise the basic functionality of your code. You should use enough to cover all of the possibilities several times. For example, if you are testing a function that determines if a number is prime, you should test with several numbers that are prime and several numbers that are not. If you are testing a function that decomposes a string into a list of English words (such as turning "thisisatest" into ["this", "is", "a", "test"], you might use strings that are single English words, such as "bob", strings with two words, such as "testtwo", strings with a larger number of words, such as "onetwothreefourfive", and strings that can be decomposed in multiple ways, such as "onestone" (which could be ["one", "stone"] or ["ones", "tone"]). You should also include negative test cases: strings that don't contain any word, such as "pxq" and strings that contain some words but can't be completely decomposed into words, such as "testfailswxq".

*Boundary cases* apply in situations where there is a range of possible inputs. Bugs frequently hide at the edge of the range of possibilities. For example, with a function that tests for prime numbers, the bottom end of the range would be in the numbers 1 and 2; 1 is generally not considered to be prime while 2 is definitely prime. You should test both. For the program that decomposes strings, you should test both the empty string and strings of one letter. If the unit you are testing is based on a loop, make sure it is both starting and stopping at the right point. A common error results when a *for*-loop goes one step too far, or stops one step earlier than it should. Try to find test cases that check for this.

*Extreme cases* test input you might not ordinary think about. For example, with the prime number tester, what will it respond if you give it a negative number? There is no *a priori* right answer, but you should code your programs to handle smoothly unexpected input. Programs should not crash, regardless of the input they are given. For the prime number function you should include 0 and a negative number as test cases. For the string decomposition function you should test the empty string. You should also test very large inputs to ensure that you haven't unintentionally placed a limit to the size of the input your program can handle. There is no upper end to the range of prime numbers, but you might test a very large number to ensure that you haven't accidentally put a limit on the size of the numbers your function can handle: 15458863 is the millionth prime number. As with numbers there is no upper boundary on the size of strings but you should test a very large input, such as "IreadthenewstodayohboyAboutaluckymanwhomadethegradeAndthoughthenewswasrathersadWellIjusthadtolaughIsawthephotographHeblewhismindoutinacarHedidn'tnoticethatthelightshadchangedAcrowdofpeoplestoodandstaredThey'dseenhisfacebeforeNobodywasreallysureIfhewasfromtheHouseofLords" Depending on the algorithms you are using, large inputs sometimes are processed very, very slowly; when testing you need to take this into account and use large, but still practical, test cases.

Lets' consider the palindrome program we wrote in section 4.3 and outlined above. We test this in three stages.

**Phase one** We test just the main() function, with a stub for the IsPalindrome() function. Here we just want to be sure the main loop is working.

Typical cases "test", "this is a test"

Boundary cases the empty string

Extreme cases "fox socks box knox knox in box fox in socks knox on fox in socks in box"

**Phase two** We test the Reverse() and IsPalindrome() functions. The program should now recognize palindromes. We haven't yet written the StripPunctuation() function, so we will use test strings that have only alphabetic letters.

Typical cases "madam", "maddam", "maday"

Boundary cases the empty string, "a"

Extreme cases "amanaplanacanalpanama", "abcdefghijklmnopqrstuvwxyzyxwvutsrqponmlkjihgfedcba"

**Phase three** Now we test the whole program, with special emphasis on the StripPunctuation() function:

Typical cases "A man, a plan, a canal: Panama!", "Drat such custard", "abc;a"

Boundary cases the empty string, "a"

Extreme cases "A man, a plan, a cat, a ham, a yak, a yam, a hat, a canal-Panama!"

## How to test

One simple way to manage testing, which works in any programming language, is to develop your program in such a what that it can always be run. That is how we developed program **??**. We even implemented a stub for the IsPalindrome() function so that main() would run when we hadn't written the rest of the code. This leads to a program design strategy known as *Top-Down Design* – at each step we implement one part of the program (typically one function), breaking that part down into steps that are each represented by functions. We give stubs for the new functions. This continues until the steps that we need to implement are so simple that we can code them without calling any new function. Some languages, though not Python, require function definitions to be present in the code before the calls to those functions. If you do this even though Python doesn't require it (in larger programs it does help you to find function definitions), Top-Down Design results in you writing the code file backwards – from the end to the start. As long as you use a good program editor, such as Idle,

that doesn't present any difficulties. If you use this strategy in each phase of testing you only need to run the program and type in the test cases.

An alternative for testing Python code makes use of Python's interactivity. When we run a program in Idle, the entire program is loaded into memory. When the program stops running, it stay's in Idle's memory. Individual functions can still be called as long as you give them appropriate arguments. For example, if you run program **??** and immediately enter the empty string, at the prompts you can directly call the Reverse(), IsPalindrome(), and StripPunctuation() functions. For example, here is a typical interaction:

When I run the program it types the program's prompt for input:

```
Enter a string:
```

I press the Return key to enter the empty string and the program halts. Idle then gives me a prompt:

```
>>>
```

I want to see the result of reversing a string, so at the prompt I type

```
>>> Reverse("abc")
```

Idle responds by printing the result of calling Reverse("abc") and then another prompt:

```
'cba'
>>>
```

I next check out IsPalindrome() by typing at the prompt

```
>>> IsPalindrome( "bob" )
```

to which Idle replies

```
True
>>>
```

Finally, I try calling StripPunctuation():

```
>>> StripPunctuation( "Naomi did I moan?" )
```

and Idle responds

```
'naomididimoan'
>>>
```

In this way we can quickly and easily test a variety of functions.

## Debugging – making use of testing results

No amount of testing can show that code is bug-free. Testing only tells us about the presence of errors, not their absence. Most students don't want to see it this way, but a successful test is one that reveals a problem. So celebrate when a test comes back with the wrong answer: you found a bug! But now, what do

you do? Fortunately, 99% of all bugs can be fixed by just carefully reading the code. If you have followed our methodology for developing programs the test that failed should involve only a small amount of new code. All of the prior code should have been thoroughly tested in earlier testing phases, so the potential source of the bug should be easy to localize. First, take a minute to find the simplest example you can of input that fails. If the problem is a function like StripPunctuation(), it is a lot easier to to work with a string of 2 or 3 characters than one of 15. Now it is just a matter of thinking carefully about your code. Check your logic; does the algorithm you are using really do what you think? Try stepping through the guilty function on paper, working through all of the steps of the example that failed. If done carefully, this almost always finds the problem. In tricky situations it helps if you can find two examples, only one of which runs correctly. If you can find why one succeeds and one fails you will probably find the bug. Don't give in to the temptation of monkey-coding – a cycle of random changes to the code followed by quick tests. This usually introduces more new bugs than it fixes. Stay in control of your code and don't make a change until you are confident it is correct. Any program you are asked to write in an introductory course will be short enough that you can keep it all in your head at one time. It is much faster to reason your way through the program than to complete it through arbitrary modifications.

The next two sections discuss *debuggers*, which are tools designed to help with the debugging process. These can be very helpful for longer programs. You probably won't need them for a first programming course, but some people find them helpful. Use them if you wish, but don't rely on them as a substitute for careful reading.