CHAPTER 16

Testing, 1-2-3

How do you know that your program works? Can you rely on yourself to write flawless code all the time? Meaning no disrespect, I would guess that's unlikely. It's quite easy to write correct code in Python most of the time, certainly, but chances are your code will have bugs.[1] Debugging is a fact of life for programmers—an integral part of the craft of programming. However, the only way to get started debugging is to *run your program*. Right? And simply running your program might not be enough. If you have written a program that processes files in some way, for example, you will need some files to run it on. Or if you have written a utility library with mathematical functions, you will need to supply those functions with parameters in order to get your code to run.

Programmers do this kind of thing all the time. In compiled languages, the cycle goes something like "edit, compile, run," around and around. In some cases, even getting the program to compile may be a problem, so the programmer simply switches between editing and compiling. In Python, the compilation step isn't there—you simply edit and run. Running your program is what testing is all about.

In this chapter, I discuss the basics of testing. I give you some notes on how to let testing become one of your programming habits and show you some useful tools for writing your tests. In addition to the testing and profiling tools of the standard library, I show you how to use the code analyzers PyChecker and PyLint.

For more on programming practice and philosophy, see Chapter 19. There, I also mention logging, which is somewhat related to testing.

Test First, Code Later

To plan for change and flexibility, which is crucial if your code is going to survive even to the end of your own development process, it's important to set up tests for the various parts of your program (so-called *unit tests*). It's also a very practical and pragmatic part of designing your application. Rather than the intuitive "code a little, test a little" practice, the Extreme Programming crowd (a relatively new movement in software design and development) has introduced the highly useful, but somewhat counterintuitive, dictum "test a little, code a little."

1. Did you know that the original computer bug was, in fact, a moth? It was found stuck in a relay in the Mark II computer at Harvard in 1945. The term *bug* for a computer glitch and the related word *debugging* are credited to Grace Hopper, who taped the original bug into her logbook. The logbook—with the bug—is on display at the US Naval Surface Weapons Center in Dahlgren, Virginia. (See `http://en.wikipedia.org/wiki/Software_bug` for more information.)

In other words, test first and code later. This is also known as *test-driven programming*. While this may be unfamiliar at first, it can have many advantages, and it does grow on you over time. Eventually, once you've used test-driven programming for a while, writing code without having tests in place will seem really backwards.

Precise Requirement Specification

When developing a piece of software, you must first know what problem the software will solve—what objectives it will meet. You can clarify your goals for the program by writing a *requirement specification*, a document (or just some quick notes) describing requirements the program must satisfy. It is then easy to check at some later time whether the requirements are indeed satisfied. But many programmers dislike writing reports and in general prefer to have their computer do as much of their work as possible. Here's good news: you can specify the requirements in Python and have the interpreter check whether they are satisfied!

■ **Note** There are many types of requirements, including such vague concepts as client satisfaction. In this section, I focus on *functional* requirements—that is, what is required of the program's functionality.

The idea is to start by writing a test program, and *then* write a program that passes the tests. The test program is your requirement specification and helps you stick to those requirements while developing the program.

Let's take a simple example. Suppose you want to write a module with a single function that will compute the area of a rectangle with a given height and a given width. Before you start coding, you write a unit test with some examples for which you know the answers. Your test program might look like the one in Listing 16-1.

Listing 16-1. *A Simple Test Program*

```
from area import rect_area
height = 3
width = 4
correct_answer = 12
answer = rect_area(height, width)
if answer == correct_answer:
    print 'Test passed '
else:
    print 'Test failed '
```

In this example, I call the function `rect_area` (which I haven't written yet) on the height 3 and width 4, and compare the answer with the correct one, which is 12.[2]

2. Of course, testing only one case like this won't give you much confidence in the correctness of the code. A real test program would probably be a lot more thorough.

If you then carelessly implement `rect_area` (in the file `area.py`) as follows, and try to run the test program, you would get an error message:

```
def rect_area(height, width):
    return height * height # This is wrong...
```

You could then examine the code to see what was wrong, and replace the returned expression with `height * width`.

Writing a test before you write your code isn't just a preparation for finding bugs—it's much more profound than that. It's a preparation for seeing whether your code works at all. It's a bit like the old Zen koan: "Does a tree falling in the forest make a sound if no one is there to hear it?" Well, of course it does (sorry, Zen monks), but the sound doesn't have any impact on you or anyone else. With your code, the question is, "Until you test it, does it actually *do* anything?" Philosophy aside, it can be useful to adopt the attitude that a feature doesn't really exist (or isn't really a feature) until you have a test for it. Then you can clearly demonstrate that it's there and is doing what it's supposed to do. This isn't only useful while developing the program initially, but also when you later extend and maintain the code.

Planning for Change

In addition to helping a great deal as you write the program, automated tests help you avoid accumulating errors when you introduce changes. As discussed in Chapter 19, you should be prepared to change your code, rather than clinging frantically to what you have, but change has its dangers. When you change some piece of your code, you very often introduce an unforeseen bug or two. If you have designed your program well (with a lot of abstraction and encapsulation), the effects of a change should be local, and affect only a small piece of the code. That means that debugging is easier *if you spot the bug*.

CODE COVERAGE

The concept of *coverage* is an important part of testing lore. When you run your tests, chances are you won't run all parts of your code, even though that would be the ideal situation. (Actually, the *ideal* situation would be to run through every possible state of your program, using every possible input, but that's really not going to happen.) One of the goals of a good test suite is to get good coverage, and one way of ensuring that is to use a coverage tool, which measures the percentage of your code that was actually run during the testing. At the time of writing, there is no really standardized coverage tool for Python, but a web search for something like "test coverage python" should turn up a few options. One option is the (currently undocumented) program `trace.py`, which comes with the Python distribution. You can run it as a program on the command line (possibly using the `-m` switch, saving you the trouble of finding the file), or you can import it as a module. For help on how to use it, you can either run the program with the `--help` switch or import the module and execute `help(trace)` in the interpreter.

At times, you may feel overwhelmed by the requirement to test everything extensively. Don't worry—you don't have to test hundreds of combinations of inputs and state variables, at least not to begin with. The most important part of test-driven programming is that you actually run your method (or function or script) repeatedly while coding, to get continual feedback on how you're doing. If you want to increase your confidence in the correctness of the code (as well as the coverage), you can always add more tests later.

The point is that if you don't have a thorough set of tests handy, you may not even discover that you have introduced a bug until later, when you no longer know how the error was introduced. And without a good suite of tests, it is much harder to pinpoint exactly what is wrong. You can't roll with the punches unless you see them coming. One way of making sure that you get good *test coverage* (that is, that your tests exercise much, if not most, of your code) is, in fact, to follow the tenets of test-driven programming. If you make sure that you have written the tests *before* you write the function, you can be certain that every function is tested.

The 1-2-3 (and 4) of Testing

Before we get into the nitty-gritty of writing tests, here's a breakdown of the test-driven development process (or one variation of it):

1. Figure out the new feature you want. Possibly document it, and then write a test for it.
2. Write some skeleton code for the feature, so that your program runs without any syntax errors or the like, but your test fails. It is important to see your test fail, so you are sure that it actually *can* fail. If there is something wrong with the test, and it always succeeds no matter what (this has happened to me many times), you aren't really testing anything. This bears repeating: see your test *fail* before you try to make it *succeed*.
3. Write dummy code for your skeleton, just to appease the test. This doesn't have to accurately implement the functionality; it just needs to make the test pass. This way, you can have all your tests pass all the time when developing (except the first time you run the test, remember?), even while initially implementing the functionality.
4. Rewrite (or *refactor*) the code so that it actually does what it's supposed to, all the while making sure that your test keeps succeeding.

You should keep your code in a healthy state when you leave it—don't leave it with any tests failing. Well, that's what they say. I find that I sometimes leave it with *one* test failing, which is the point at which I'm currently working, as a sort of "to-do" or "continue here" for myself. This is really bad form if you're developing together with others, though. You should never check failing code into the common code repository.

Tools for Testing

You may think that writing a lot of tests to make sure that every detail of your program works correctly sounds like a chore. Well, I have good news for you: there is help in the standard libraries (isn't there always?). Two brilliant modules are available to automate the testing process for you:

- `unittest`: A generic testing framework.
- `doctest`: A simpler module, designed for checking documentation, but excellent for writing unit tests as well.

Let's begin with a look at `doctest`, which is a great starting point.

doctest

Throughout this book, I use examples taken directly from the interactive interpreter. I find that this is an effective way to show how things work, and when you have such an example, it's easy to test it for yourself. In fact, interactive interpreter sessions can be a useful form of documentation to put in docstrings. For instance, let's say I write a function for squaring a number, and add an example to its docstring:

```
def square(x):
    '''
    Squares a number and returns the result.

    >>> square(2)
    4
    >>> square(3)
    9
    '''
    return x*x
```

As you can see, I've included some text in the docstring, too. What does this have to do with testing? Let's say the `square` function is defined in the module `my_math` (that is, a file called `my_math.py`). Then you could add the following code at the bottom:

```
if __name__=='__main__':
    import doctest, my_math
    doctest.testmod(my_math)
```

That's not a lot, is it? You simply import `doctest` and the `my_math` module itself, and then run the `testmod` (for "test module") function from `doctest`. What does this do? Let's try it:

```
$ python my_math.py
$
```

Nothing seems to have happened, but that's a good thing. The `doctest.testmod` function reads all the docstrings of a module and seeks out any text that looks like an example from the interactive interpreter. Then it checks whether the example represents reality.

■ **Note** If I were writing a real function here, I would (or should, according to the rules I laid down earlier) first write the docstring, run the script with `doctest` to see the test fail, add a dummy version (for example using `if` statements to deal with the specific inputs in the docstring) so that the test succeeds, and *then* start working on getting the implementation right. On the other hand, if you're going to do full-out "test-first, code-later" programming, the `unittest` framework (discussed later) might suit your needs better.

To get some more input, you can just give the `-v` (for "verbose") switch to your script:

```
$ python my_math.py -v
```

This command would result in the following output:

```
Running my_math.__doc__
0 of 0 examples failed in my_math.__doc__
Running my_math.square.__doc__
Trying: square(2)
Expecting: 4
ok

Trying: square(3)
Expecting: 9
ok
0 of 2 examples failed in my_math.square.__doc__
1 items had no tests:
    test
1 items passed all tests:
   2 tests in my_math.square
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
```

As you can see, a lot happened behind the scenes. The `testmod` function checks both the module docstring (which, as you can see, contains no tests) and the function docstring (which contains two tests, both of which succeed).

With this in place, you can safely change your code. Let's say that you want to use the Python exponentiation operator instead of plain multiplication, and use `x**2` instead of `x*x`. You edit the code, but accidentally forget to enter the number 2, ending up with `x**x`. Try it, and then run the script to test the code. What happens? This is the output you get:

```
**********************************************************************
Failure in example: square(3)
from line #5 of my_math.square
Expected: 9
Got: 27
**********************************************************************
1 items had failures:
   1 of   2 in my_math.square
***Test Failed*** 1 failures.
```

So the bug was caught, and you get a very clear description of what is wrong. Fixing the problem shouldn't be difficult now.

■ **Caution** Don't trust your tests blindly, and be sure to test enough cases. As you can see, the test using `square(2)` does *not* catch the bug because for `x==2`, `x**2` and `x**x` are the same thing!

For more information about the `doctest` module, you should again check out the library reference (`http://python.org/doc/lib/module-doctest.html`).

unittest

While `doctest` is very easy to use, `unittest` (based on the popular test framework JUnit, for Java) is more flexible and powerful. `unittest` may have a steeper learning curve than `doctest`, but I suggest that you take a look at this module, because it allows you to write very large and thorough test sets in a more structured manner.

I will give you just a gentle introduction here.`unittest` includes some features that you probably won't need for most of your testing. For complete details, see the Python Library Reference (`http://python.org/doc/lib/module-unittest.html`).

■ **Tip** A couple of interesting alternatives to the unit test tools in the standard library are `py.test` (`http://codespeak.net/py/dist/test.html`) and nose (`http://code.google.com/p/python-nose`).

Again, let's take a look at a simple example. You're going to write a module called `my_math` containing a function for calculating products, called `product`. So where do you begin? With a test, of course (in a file called `test_my_math.py`), using the `TestCase` class from the `unittest` module (see Listing 16-2).

Listing 16-2. *A Simple Test Using the unittest Framework*

```
import unittest, my_math

class ProductTestCase(unittest.TestCase):

    def testIntegers(self):
        for x in xrange(-10, 10):
            for y in xrange(-10, 10):
                p = my_math.product(x, y)
                self.failUnless(p == x*y, 'Integer multiplication failed')

    def testFloats(self):
        for x in xrange(-10, 10):
```

```
            for y in xrange(-10, 10):
                x = x/10.0
                y = y/10.0
                p = my_math.product(x, y)
                self.failUnless(p == x*y, 'Float multiplication failed')

if __name__ == '__main__': unittest.main()
```

The function `unittest.main` takes care of running the tests for you. It will instantiate all subclasses of `TestCase` and run all methods whose names start with `test`.

■ **Tip** If you define methods called `startUp` and `tearDown`, they will be executed before and after each of the test methods. You can use these methods to provide common initialization and cleanup code for all the tests, a so-called *test fixture*.

Running this test script will, of course, simply give you an exception about the module `my_math` not existing. Methods such as `failUnless` check a condition to determine whether the given test succeeds or fails. The module has many other methods, such as `failIf`, `failUnlessEqual`, and `failIfEqual`. See Table 16-1 for a brief overview. (Again, refer to the Python Library Reference, `http://python.org/doc/lib/testcase-objects.html`, for complete information.)

Table 16-1. *Some Useful TestCase Methods*

Method	Description
`assert_(expr[, msg])`	Fail if the expression is false, optionally giving a message. (Note the underscore.)
`failUnless(expr[, msg])`	Same as `assert_`.
`assertEqual(x, y[, msg])`	Fail if two values are different, printing both values in traceback.
`failUnlessEqual(x, y[, msg])`	Same as `assertEqual`.
`assertNotEqual(x, y[, msg])`	The opposite of `assertEqual`.
`failIfEqual(x, y[, msg])`	The same as `assertNotEqual`.
`assertAlmostEqual(x, y[, places[, msg]])`	Similar to `assertEqual`, but with some leeway for `float` values.
`failUnlessAlmostEqual(x, y[, places[, msg]])`	The same as `assertAlmostEqual`.
`assertNotAlmostEqual(x, y[, places[, msg]])`	The opposite of `assertAlmostEqual`.
`failIfAlmostEqual(x, y[, msg])`	The same as `assertNotAlmostEqual`.
`assertRaises(exc, callable, ...)`	Fail unless the callable raises `exc` when called (with optional arguments).

Method	Description
`failUnlessRaises(exc, callable, ...)`	Same as `assertRaises`.
`failIf(expr[, msg])`	Opposite of `assert_`.
`fail([msg])`	Unconditional failure, with an optional message, as for the other methods.

The `unittest` module distinguishes between *errors*, where an exception is raised, and *failures*, which result from calls to `failUnless` and the like. The next step is to write skeleton code, so we don't get errors—only failures. This simply means to create a module called `my_math` (that is, a file called `my_math.py`) containing the following:

```
def product(x, y):
    pass
```

All filler, no fun. If you run the test now, you should get two `FAIL` messages, like this:

```
FF
======================================================================
FAIL: testFloats (__main__.ProductTestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_my_math.py", line 17, in testFloats
    self.failUnless(p == x*y, 'Float multiplication failed')
AssertionError: Float multiplication failed

======================================================================
FAIL: testIntegers (__main__.ProductTestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_my_math.py", line 9, in testIntegers
    self.failUnless(p == x*y, 'Integer multiplication failed')
AssertionError: Integer multiplication failed

----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=2)
```

This was all expected, so don't worry too much. Now, at least, you know that the tests are really linked to the code—the code was wrong, and the tests failed. Wonderful.

The next step is to make it work. In this case, there isn't much to it, of course:

```
def product(x, y):
    return x * y
```

Now the output is simply as follows:

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.015s

OK
```

The two dots at the top are the tests. If you look closely at the jumbled output from the failed version, you'll see two characters on the top there as well: two Fs, indicating two failures.

Just for fun, change the `product` function so that it fails for the specific parameters 7 and 9:

```
def product(x, y):
    if x == 7 and y == 9:
        return 'An insidious bug has surfaced!'
    else:
        return x * y
```

If you run the test script again, you should get a single failure:

```
.F
======================================================================
FAIL: testIntegers (__main__.ProductTestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_my_math.py", line 9, in testIntegers
    self.failUnless(p == x*y, 'Integer multiplication failed')
AssertionError: Integer multiplication failed

----------------------------------------------------------------------
Ran 2 tests in 0.005s

FAILED (failures=1)
```

■ **Tip** There is also a GUI for `unittest`. See the PyUnit (another name for `unittest`) web page, `http://pyunit.sf.net`, for more information.

Beyond Unit Tests

Tests are clearly important, and for any somewhat complex project, they are absolutely vital. Even if you don't want to bother with structured suites of unit tests, you really must have some way of running your program to see whether it works. Having this capability in place *before* you do any significant amount of coding can save you a bundle of work (and pain) later on.

There are other ways of probulating (what, you don't watch *Futurama*?) your program, and here I'll show you several tools for doing just that: source code checking and profiling. Source code checking is a way of looking for common mistakes or problems in your code (a bit like what compilers can do for statically typed languages, but going far beyond that). Profiling is a way of finding out how fast your program really is. I discuss the topics in this order to honor the good old rule, "Make it work, make it better, make it faster." The unit testing helped make it work; source code checking can help make it better; and, finally, profiling can help make it faster.

Source Code Checking with PyChecker and PyLint

For quite some time, PyChecker (`http://pychecker.sf.net`) was the only tool for checking Python source code, looking for mistakes such as supplying arguments that won't work with a given function and so forth. (All right, there was `tabnanny`, in the standard library, but that isn't all that powerful, since it just checks your indentation.) Then along came PyLint (`http://www.logilab.org/projects/pylint`), which supports most of the features of PyChecker and quite a few more (such as whether your variable names fit a given naming convention, whether you're adhering to your own coding standards, and the like).

Installing the tools is simple. They are both available from several package manager systems (such as Debian APT and Gentoo Portage), and may also be downloaded directly from their respective web sites. You install using Distutils, with the standard command:

```
python setup.py install
```

PyLint also requires the Logilab Common libraries to work. Download that package, called `logilab-common`, available from the PyLint web site, and install it the same way as PyLint.

Once this is done, the tools should be available as command-line scripts (`pychecker` and `pylint` for PyChecker and PyLint, respectively) and as Python modules (with the same names).

■ **Note** In Windows, the two tools use the batch files `pychecker.bat` and `pylint.bat` as command-line tools. You may need to add these to your `PATH` environment variable to have the `pychecker` and `pylint` commands available on the command line.

To check files with PyChecker, you run the script with the file names as arguments, like this:

```
pychecker file1.py file2.py ...
```

With PyLint, you use the *module* (or package) names:

```
pylint module
```

You can get more information about both tools by running them with the `-h` command-line switch. When you run either of these commands, you will probably get quite a bit of output (most likely more output from `pylint` than from `pychecker`). Both tools are quite configurable with respect to which warnings you want to get (or suppress); see their respective documentation for more information.

Before leaving the checkers, let's see how you can combine them with unit tests. After all, it would be very pleasant to have them (or just one of them) run automatically as a test in your test suite, and to silently succeed if nothing is wrong. Then you could actually have a test suite that doesn't just test functionality, but code quality as well.

Both PyChecker and PyLint can be imported as modules (`pychecker.checker` and `pylint.lint`, respectively), but they aren't really designed to be used programmatically. When you import `pychecker.checker`, it will check the code that comes later (including imported modules), printing warnings to standard output. The `pylint.lint` module has an undocumented function called `Run`, which is used in the `pylint` script itself. This also prints out warnings rather than returning them in some way. Instead of grappling with these issues, I suggest using PyChecker and PyLint in the way they're meant to be used: as command-line tools. And the way of using command-line tools in Python is the `subprocess` module (or one of its older relatives; see the Python Library Reference for more information). Listing 16-3 is an example of the earlier test script, now with two code-checking tests.

Listing 16-3. *Calling External Checkers Using the subprocess Module*

```
import unittest, my_math
from subprocess import Popen, PIPE

class ProductTestCase(unittest.TestCase):

    # Insert previous tests here

    def testWithPyChecker(self):
        cmd = 'pychecker', '-Q', my_math.__file__.rstrip('c')
        pychecker = Popen(cmd, stdout=PIPE, stderr=PIPE)
        self.assertEqual(pychecker.stdout.read(), '')

    def testWithPyLint(self):
        cmd = 'pylint', '-rn', 'my_math'
        pylint = Popen(cmd, stdout=PIPE, stderr=PIPE)
        self.assertEqual(pylint.stdout.read(), '')

if __name__ == '__main__': unittest.main()
```

I've given some command-line switches to the checker programs, to avoid extraneous output that would interfere with the tests. For `pychecker`, I have supplied the `-Q` (quiet) switch. For `pylint`, I have supplied `-rn` (with `n` standing for "no") to turn off reports, meaning that it will display only warnings and errors. I have used `assertEqual` (instead of, for example, `failIf`) in order to have the actual output read from the `stdout` attribute displayed in the failure messages of `unittest` (this is, in fact, the main reason for using `assertEqual` instead of `failUnless` together with `==` in general).

The `pylint` command runs directly with a module name supplied, so that's pretty straightforward. To get `pychecker` to work properly, we need to get a file name. To get that, I've used the `__file__` property of the `my_math` module, `rstrip`ing away any `c` that may be found at the end of the file name (because the module may actually come from a `.pyc` file).

In order to appease PyLint (rather than configuring it to shut up about things such as short variable names, missing revisions, and docstrings), I have rewritten the `my_math` module slightly:

```
"""
A simple math module.
"""
__revision__ = '0.1'

def product(factor1, factor2):
    'The product of two numbers'
    return factor1 * factor2
```

If you run the tests now, you should not get any errors. Try to play around with the code and see if you can get any of the checkers to report errors while the functionality tests still work. (Feel free to drop either PyChecker or PyLint—one is probably enough.) For example, try to rename the parameters back to `x` and `y`, and PyLint should complain about short variable names. Or add `print 'Hello, world!'` after the `return` statement, and both checkers, quite reasonably, will complain (possibly giving different reasons for the complaint).

THE LIMITS OF AUTOMATIC CHECKING: WILL IT EVER END?

It should be obvious that there are limits to the capabilities of an automatic checker such as PyChecker or PyLint. While they are quite impressive in the breadth of errors and problems they can uncover, they can't know what your program is ultimately intended to do; hence, the need for custom-tailored unit tests. But beyond this obvious barrier, automatic checkers have other limits. If you like slightly theoretical oddities, you might be interested in a result from the exotic world of computation theory known as *the halting theorem*. Let's consider a hypothetical checker program that we could run like this:

```
halts.py myprog.py data.txt
```

As you can probably guess, the checker should check the behavior of `myprog.py` when run on the input `data.txt`. We want to check for only *one* thing: infinite loops (or infinite recursion, so *two* things, actually). In other words, the program `halts.py` should determine whether `myprog.py` would ever stop (halt) when run on `data.txt`. Given that existing checker programs can analyze the code and figure out which types the various variables must be for things to work, detecting such a simple thing as an infinite loop would seem like a breeze, right? Sorry, but no, not in the general case, anyway. According to the halting problem, it simply can't be done.

Don't take my word for it—the reasoning is actually quite simple. Assume that we *have* a working halting-checker, and assume (for simplicity) that it's written as a Python module. Now, let's assume that we write the following little insidious program, named `trouble.py`:

```
import halts, sys
name = sys.argv[1]
if halts.check(name, name):
    while True: pass
```

It uses the functionality of the `halts` module to check whether a program given as the first command-line argument will ever halt *if supplied with itself as input*. It could be run like this, for example:

```
trouble.py myprog.py
```

This would determine whether `myprog.py` would ever halt if supplied with `myprog.py` (that is, itself) as input. If the determination is that it *would* halt, `trouble.py` will enter an infinite loop. Otherwise, it will simply finish (that is, halt).

With me so far? Good. (If not, try rereading the previous stuff a couple of times; that usually helps.) Now consider the following slightly mind-bending scenario:

```
halts.py trouble.py trouble.py
```

Ta-da! What, it doesn't seem mind-bending to you? It just checks whether `trouble.py` would halt with `trouble.py` (that is, itself) as input. Sure, that's not so mind-bending in itself. But what would the result be? Consider the two alternatives: if `halts.py` says "yes"—that is, `trouble.py trouble.py` will halt—then `trouble.py trouble.py` is defined *not* to halt. We run into the same (converse) problem if we get a "no." Either way, `halts.py` is destined to get it wrong, and there is no way to fix it. We began the story by assuming that the checker actually worked, and now we have reached a contradiction, which means our assumption was wrong.

This doesn't mean that we can't detect *any* kinds of infinite looping, of course. Seeing a `while True` without a `break`, `raise`, or `return` would be a strong clue, for example. It's just not possible to detect this *in general*. Sadly, many other similar properties can't be automatically analyzed in general.[3] So even with such nifty tools as PyChecker and PyLint, we'll need to rely on manual debugging rooted in our knowledge of the special circumstances of our program. And, perhaps, we should try to avoid intentionally writing tricky programs such as `trouble.py`.

Profiling

Now that you've made your code work, and possibly made it better than the initial version, it may be time to make it faster. Then, again, it may not. One very important rule (along with such principles as KISS = Keep It Small and Simple, or YAGNI = You Ain't Gonna Need It) that you should heed when tempted to fiddle with your code to speed it up:

> *Premature optimization is the root of all evil.*
>
> —Donald Knuth, paraphrasing C. A. R. Hoare

Another way of stating this, in the words of Ken Thompson, co-inventor of UNIX, is "When in doubt, use brute force." In other words, don't worry about fancy algorithms or clever optimization tricks if you don't really, really need them. If the program is fast enough, chances are that the value of clean, simple, understandable code is much higher than that of a slightly faster program. After all, in a few months, faster hardware will probably be available anyway.

3. Check out *Computers Ltd: What They* Really *Can't Do* by David Harel (Oxford University Press, 2000) for a lot of interesting material on the subject.

But if you *do* need to optimize your program, because it simply isn't fast enough for your requirements, you absolutely should profile it before doing anything else. That is because it's really hard to guess where the bottlenecks are, unless your program is really simple. And if you don't know what's slowing down your program, chances are you'll be optimizing the wrong thing.

The standard library includes a nice profiler module called `profile` (and a faster drop-in C version, called `hotshot`). Using the profiler is straightforward. Just call its `run` method with a string argument.

```
>>> import profile
>>> from my_math import product
>>> profile.run('product(1, 2)')
```

■ **Note** In some Linux distributions, you may need to install a separate package in order to get the `profile` module to work. If it works, fine. If not, you might want to check out the relevant documentation to see if this is the problem.

This will give you a printout with information about how many times various functions and methods were called and how much time was spent in the various functions. If you supply a file name, for example, `'my_math.profile'`, as the second argument to run, the `results` will be saved to a file. You can then later use the `pstats` module to examine the profile:

```
>>> import pstats
>>> p = pstats.Stats('my_math.profile')
```

Using this `Stats` object, you can examine the results programmatically. (For details on the API, consult the standard library documentation.)

■ **Tip** The standard library also contains a module called `timeit`, which is a simple way of timing small snippets of Python code. The `timeit` module isn't really useful for detailed profiling, but it can be a nice tool when all you want to do is figure out how much time a piece of code takes to execute. Trying to do this yourself can often lead to inaccurate measurements (unless you know what you're doing). Using `timeit` is usually a better choice (unless you opt for a full profiling, of course). You can find more information about `timeit` in the Python Library Reference (`http://python.org/doc/lib/module-timeit.html`).

Now, if you're really worried about the speed of your program, you *could* add a unit test that profiles your program and enforces certain constraints (such as failing if the program takes more than a second to finish). It might be a fun thing to do, but it's not something I recommend. Obsessive profiling can easily take your attention away from things that really matter, such as clean, understandable code. If the program is *really* slow, you'll notice that anyway, because your tests will take forever to finish.

A Quick Summary

Here are the main topics covered in the chapter:

Test-driven programming: Basically, test-driven programming means to test first, code later. Tests let you rewrite your code with confidence, making your development and maintenance more flexible.

The `doctest` and `unittest` modules: These are indispensable tools if you want to do unit testing in Python. The `doctest` module is designed to check examples in docstrings, but can easily be used to design test suites. For more flexibility and structure in your suites, the `unittest` framework is very useful.

PyChecker and PyLint: These two tools read source code and point out potential (and actual) problems. They check everything from short variable names to unreachable pieces of code. With a little coding you can make them (or one of them) part of your test suite, to make sure all of your rewrites and refactorings conform to your coding standards.

Profiling: If you really care about speed and want to optimize your program (only do this if it's absolutely necessary), you should profile it first. Use the `profile` (or `hotshot`) module to find bottlenecks in your code.

New Functions in This Chapter

Function	Description
`doctest.testmod(module)`	Checks docstring examples. (Takes many more arguments.)
`unittest.main()`	Runs the unit tests in the current module.
`profile.run(stmt[, filename])`	Executes and profiles `statement`. Optionally, saves results to `filename`.

What Now?

Now you've seen all kinds of things you can do with the Python language and the standard libraries. You've seen how to probe and tweak your code until it screams (if you got serious about profiling, despite my warnings). If you *still* aren't getting the oomph you require, it's time to reach for heavier weapons. In the words of Neo in *The Matrix*. "We need guns. Lots of guns." In less metaphorical terms, it's time to pop the cover and tweak the engine with some low-level tools. (Wait, that was still metaphorical, wasn't it?)