

LINUX VOICE

TUTORIAL

BEN EVERARD

CODE NINJA: UNIT TESTING

All* good programmers make sure their software works properly before releasing it to the world. *most

WHY DO THIS?

- So you don't embarrass yourself in an international computing magazine.
- To make sure that your code works as expected.
- To catch any regression errors before they cause users problems.

In issue 7's Code Ninja, we had a piece of code that was supposed to output the Roman numerals for a particular number. As some of you noticed, it did not quite work as it should have. This month we'll take a look at what we should have done to save ourselves the embarrassment of publishing code that doesn't work properly: testing.

Testing is the process of making sure code works as it's supposed to. This can mean anything from informally entering a few values and making sure it's working properly, to a full suite of tests that run automatically and rigorously test everything to make sure it's working as expected.

The simplest form of testing (and the one that would have saved us two issues ago) is unit testing. This is where you check a particular block of code (typically a function or method) and ensure it's working correctly. Just to recap, our code from the previous tutorial was:

```
symbols = [('M', 1000), ('C M', 900), ('D', 500),
            ('C D', 400), ('C', 100), ('X C', 90), ('L', 50),
            ('X L', 40), ('X', 10), ('I X', 9), ('V', 5),
            ('I V', 4), ('I', 1)]
```

```
def romannumeral(number):
    while number > 0:
        for symbol, value in symbols:
            if number - value >= 0:
                print symbol,
                number = number - value
                continue
```

```
number_in = raw_input("Enter a number: ")
romannumeral(int(number_in))
```

This isn't particularly conducive to testing, because the same function that calculates the value also outputs it. In other words, there's nowhere to catch

```
ben@ben-All-Series: ~/romantest
ben@ben-All-Series:~/romantest$ python -m unittest roman-test
FFFFE
ERROR: test_9 (roman-test.Test)
Traceback (most recent call last):
  File "roman-test.py", line 20, in test_9
    self.assertEqual(romannumeral(9), "IX")
NameError: global name 'romannumeral' is not defined
FAIL: test_1800 (roman-test.Test)
Traceback (most recent call last):
  File "roman-test.py", line 26, in test_1800
    self.assertEqual(romannumeral(1800), "MDCCC")
AssertionError: 'MDCKCLXLXIXI' != 'MDCCC'
FAIL: test_29 (roman-test.Test)
Traceback (most recent call last):
  File "roman-test.py", line 22, in test_29
    self.assertEqual(romannumeral(29), "XXIX")
AssertionError: 'XIXVIVI' != 'XXIX'
```

If only we'd run this two months ago, we could have spared ourselves some embarrassment.

and test the value of the Roman numeral before it's sent to the terminal.

The first thing we need to do then, is re-factor the code so that the function returns the text for the Roman numeral rather than printing it. The function then becomes:

```
def romannumeral(number):
    outstring = ""
    while number > 0:
        for symbol, value in symbols:
            if number - value >= 0:
                outstring += symbol
                number = number - value
                continue
    return outstring
```

This also removes the spaces from between the symbols, so we'll remove them in the symbols list of tuples as well.

Now you can capture what Roman numerals the code is producing, and so you can now automate testing of them.

Test-driven development

There is a school of thought on software development that says that the first thing you should do when embarking on a new project is write a test. In this paradigm (known as test-driven development or TDD), the tests aren't just a way to find bugs, but form the specification for the program itself.

The process follows these steps:

- 1 Write a new test.
- 2 Run all tests and see if any fail.
- 3 If one or more tests fail, write new code to

fix the problem.

- 4 Run the tests again.
- 5 If the tests pass, clean up the code, then return to step one.

The software then evolves as new tests are added to specify new behaviour. The software is always fully tested because new features are only added after there is a test to define the behaviour, and since the tests are all run in each iteration, there shouldn't be any regressions.

PyUnitest

Testing libraries help you manage individual test cases and run them appropriately. The most popular such module for Python is **PyUnitest**. This is usually included with Python, so you shouldn't have to go hunting around for anything.

With a few test cases added, the code becomes:

```
import unittest
// symbols list
// roman numbers function
class Test(unittest.TestCase):
```

```
def test_9(self):
    self.assertEqual(romannumeral(9), "IX")
def test_29(self):
    self.assertEqual(romannumeral(29), "XXIX")
def test_707(self):
    self.assertEqual(romannumeral(707), "DCCVII")
def test_1800(self):
    self.assertEqual(romannumeral(1800), "MDCCC")

if __name__ == '__main__':
    number_in = raw_input("Enter a number: ")
    print romannumeral(int(number_in))
```

You'll need to add the **symbols** list (with the spaces removed), and the **romannumerals()** function from the previous code (they're omitted here to save space). We've called this file **roman-test.py**.

The condition **__name__ == '__main__'** is true when the code is being run from the command line, so this allows us to still run it normally with **python roman-test.py**, but it means that the code works properly when imported into the test module.

The tests are all methods of a class that inherits from **unittest.TestCase**, and they all call one of the **assert** methods. Here we've used **assertEqual()** to check that the value returned from the **romannumeral()** function is the right value.

If you call the file containing the code **roman-test.py**, you can run the tests with:

```
python -m unittest roman-test
```

Ah, it seems that three of the four tests fail. It turns out that there's an error in our code that generates the Roman numerals. The **continue** statement should be a **break** statement. If you make this change, you should find that all the tests pass.

Getting assertive

In this example, we've used **assertEqual** to check if a particular test passes or not, but there are many different methods you can use. Some of the most

Other forms of testing

Here we've looked at unit testing. This is great for making sure that a particular part of your program is working properly, but in complex software, this can still miss bugs.

■ Integration testing

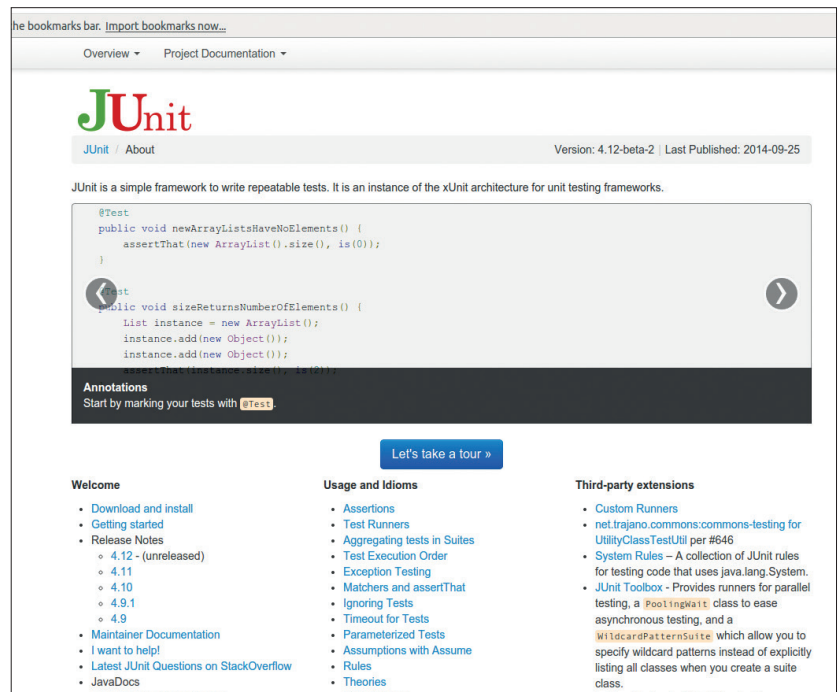
Unit testing checks that each bit of code works correctly by itself. The next step is to make sure that all the bits of code work properly when combined together. This is known as integration testing.

■ Systems testing

Most programs aren't isolated bits of code, but part of a larger ecosystem. For example, they might get some of their data from an external database, or send files to another server. Systems testing is where you test that the software works correctly with all the external services that it uses.

■ Usability testing

Bugs aren't just bits of code that don't work properly. They can also be things that don't work as the user expects them, or confusing GUIs. Usability testing is where you put real users in front of the software and make sure it works as they expect it to.




useful are **assertTrue(statement)**, **assertRaises(exception)**, **assertIsInstance(object, class)** and **assertAlmostEqual(value1, value2)**. You can get a full list from the documentation at <https://docs.python.org/2/library/unittest.html>.

The above test cases only check four numbers. It's trivially easy to add more test cases (we kept it short to save space). In fact, in this case, it would be possible to set an upper bound (say, 1000), and enter the correct data for every possible number. This way we could ensure that it was definitely producing the correct output. This is known as exhaustive testing.

However, if the software had a wider range of inputs, then it may not be practical to run an exhaustive test. In this case, we'd have to be selective in which values we test. We want to pick the values that are most likely to lead to an error.

There aren't any hard-and-fast rules about this, but there are a few guidelines that can help you. You want broad coverage. That

means that you don't want to cluster all your tests in one area. You also want to check areas where the output flips from one case to the next (eg 8 and 9 which go from VIII to IX). Edge and corner cases can also be fertile sources of errors. This is where you push one or more parameters to their maximum values.

If you create a good suite of tests when developing a particular part of a piece of software, then you can use these tests to ensure that you don't accidentally introduce a bug (or regression) into this area as you add features, or fix other bugs. This is known as regression testing, and as software becomes more complex, it becomes more important. 

Java's *JUnit* is probably the best known of the unit testing frameworks, but *SUnit* (written for the Smalltalk programming language) came first.

“Testing libraries help you manage individual test cases and run them appropriately.”