Software Engineering Lab #12
Unit Testing

In software testing, testing is commonly divided into several categories, based on how complex the component being tested is. Most of our time will be focused on the lowest level – **unit testing**.

Unit testing is the testing of the smallest possible pieces of a program. Often, this means individual functions or methods which cannot be divided further. Unit tests are used to test a single unit in isolation, verifying that it works as expected, without considering what the rest of the program does. This makes it easy to narrow down and identify the actual problem.

For software development in Python, there is a framework for unit testing called **unittest,** which is a Python module included with Python 2.1 and later. The "**unittest**" module is sometimes referred to as "**PyUnit**".

1. Unit Testing Example

Before we start writing a test suite with Python unittest module, let's consider an example of a set of utility functions to convert to and from Roman numerals which will be used to demonstrate how to write a test suite.

In Roman numerals, there are seven characters which are repeated and combined in various ways to represent numbers.

1. `I = 1`
2. `V = 5`
3. `X = 10`
4. `L = 50`
5. `C = 100`
6. `D = 500`
7. `M = 1000`

There are some general rules for constructing Roman numerals:

1) Characters are additive. `I` is `1`, `II` is `2`, and `III` is `3`. `VI` is `6` (literally, "`5` and `1`"), `VII` is `7`, and `VIII` is `8`.
2) The tens characters (`I`, `X`, `C`, and `M`) can be repeated up to three times. At `4`, you have to subtract from the next highest fives character. You can't represent `4` as `IIII`; instead, it is represented as `IV` ("`1` less than `5`"). `40` is written as `XL` ("`10` less than `50`"), `41` as `XLI`, `42` as `XLII`, `43` as `XLIII`, and then `44` as `XLIV` ("`10` less than `50`, then `1` less than `5`").
3) Similarly, at `9`, you have to subtract from the next highest tens character: `8` is `VIII`, but `9` is `IX` ("`1` less than `10`"), not `VIIII` (since the `I` character cannot be repeated four times). `90` is `XC`, `900` is `CM`.
4) The fives characters cannot be repeated. `10` is always represented as `X`, never as `VV`. `100` is always `C`, never `LL`.
5) Roman numerals are always written highest to lowest, and read left to right, so order of characters matters very much. `DC` is `600`; `CD` is a completely different number (`400`, "`100` less than `500`"). `CI` is `101`; `IC` is not even a valid Roman numeral (because you can't subtract `1` directly from `100`; you would have to write it as `XCIX`, "`10` less than `100`, then `1` less than `10`").

These rules lead to a number of interesting observations:

1) There is only one correct way to represent a number as Roman numerals.
2) The converse is also true: if a string of characters is a valid Roman numeral, it represents only one number (*i.e.* it can only be read in one way).
3) There is a limited range of numbers that can be expressed as Roman numerals, specifically `1` through `3999`. (The Romans did have several ways of expressing larger numbers, for instance by having a bar over a numeral to represent that its normal value should be multiplied by `1000`, but we're not going to deal with that. For the purposes of this demonstration, Roman numerals go from `1` to `3999`).

4) There is no way to represent **0** in Roman numerals. (Amazingly, the ancient Romans had no concept of **0** as a number. Numbers were for counting things you had; how can you count what you don't have?)
5) There is no way to represent negative numbers in Roman numerals.
6) There is no way to represent decimals or fractions in Roman numerals.

Given all of this, we can define requirements for a function to convert from an integer to a Roman numeral (**toRoman** function) and a function to convert from a Roman numeral back to an integer (**fromRoman** function). Suppose both **toRoman** and **fromRoman** are defined in **roman.py** the requirements for them is as follows:

1) **toRoman** should return the Roman numeral representation for all integers **1** to **3999**.
2) **toRoman** should fail when given an integer outside the range **1** to **3999**.
3) **toRoman** should fail when given a non-integer decimal.
4) **fromRoman** should take a valid Roman numeral and return the number that it represents.
5) **fromRoman** should fail when given an invalid Roman numeral.
6) If you take a number, convert it to Roman numerals, then convert that back to a number, you should end up with a number you started with. So **fromRoman(toRoman(n)) == n** for all **n** in **1..3999**.
7) **toRoman** should always return a Roman numeral using uppercase letters.
8) **fromRoman** should only accept uppercase Roman numerals (*i.e.* it should fail when given lowercase input).

We're going to write a test suite even before we write the actual code that performs functions we need. This follows the principle of test-driven development – you write a failing automated **test case** that define a desired function then produce code to pass that test case. Unit testing is important in all phases of development:

- Before writing code, it forces you to detail your requirements in a useful fashion.
- While writing code, it keeps you from over-coding. When all the test cases pass, the function is complete.
- When refactoring code, it assures you that the new version behaves the same way as the old version.
- When maintaining code, it helps you assert that your latest change in the unit is not broken.

2. Testing for Success

The most fundamental part of unit testing is constructing individual test cases. A test case should be able to:

- Run completely by itself, without any human input. Unit testing is about automation.
- Determine by itself whether the function it is testing has passed or failed, without a human interpreting the results.
- Run in isolation, separate from any other test cases (even if they test the same function).

From the first requirement – **toRoman** should return the Roman numeral representation for all integers **1** to **3999** – we can write a test case for known input/output pairs.

Our test module will be defined as follows:

```
"""romantest.py - unit test for roman.py"""

import roman0 as roman
import unittest

class KnownValues(unittest.TestCase):    # (1)
    knownValues = ( (1, 'I'),
                    (2, 'II'),
                    (3, 'III'),
                    (4, 'IV'),
                    (5, 'V'),
                    (6, 'VI'),
                    (7, 'VII'),
                    (8, 'VIII'),
                    (9, 'IX'),
                    (10, 'X'),
                    (50, 'L'),
                    (100, 'C'),
                    (500, 'D'),
                    (1000, 'M'),
                    (31, 'XXXI'),
                    (148, 'CXLVIII'),
                    (294, 'CCXCIV'),
                    (312, 'CCCXII'),
                    (421, 'CDXXI'),
                    (528, 'DXXVIII'),
                    (621, 'DCXXI'),
                    (782, 'DCCLXXXII'),
                    (870, 'DCCCLXX'),
                    (941, 'CMXLI'),
                    (1043, 'MXLIII'),
                    (1110, 'MCX'),
                    (1226, 'MCCXXVI'),
                    (1301, 'MCCCI'),
                    (1485, 'MCDLXXXV'),
                    (1509, 'MDIX'),
                    (1607, 'MDCVII'),
                    (1754, 'MDCCLIV'),
                    (1832, 'MDCCCXXXII'),
                    (1993, 'MCMXCIII'),
                    (2074, 'MMLXXIV'),
                    (2152, 'MMCLII'),
                    (2212, 'MMCCXII'),
                    (2343, 'MMCCCXLIII'),
                    (2499, 'MMCDXCIX'),
                    (2574, 'MMDLXXIV'),
                    (2646, 'MMDCXLVI'),
                    (2723, 'MMDCCXXIII'),
                    (2892, 'MMDCCCXCII'),
                    (2975, 'MMCMLXXV'),
                    (3051, 'MMMLI'),
                    (3185, 'MMMCLXXXV'),
                    (3250, 'MMMCCL'),
                    (3313, 'MMMCCCXIII'),
                    (3408, 'MMMCDVIII'),
                    (3501, 'MMMDI'),
                    (3610, 'MMMDCX'),
                    (3743, 'MMMDCCXLIII'),
                    (3844, 'MMMDCCCXLIV'),
                    (3888, 'MMMDCCCLXXXVIII'),
                    (3940, 'MMMCMXL'),
                    (3999, 'MMMCMXCIX'))                     # (2)

    def testToRomanKnownValues(self):                       # (3)
        """toRoman should give known result with known input"""
        for integer, numeral in self.knownValues:
            result = roman.toRoman(integer)
            self.assertEqual(numeral, result)

    def testFromRomanKnownValues(self):
        """fromRoman should give known result with known input"""
        for integer, numeral in self.knownValues:
            result = roman.fromRoman(numeral)               # (4) (5)
            self.assertEqual(integer, result)               # (6)

# TODO: add more test cases here

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

(1) To write a test case, first create a subclass of the **`TestCase`** class of the **`unittest`** module. This class provides many useful methods which you can use in your test case to test specific conditions.

(2) This test case uses a list of integer/numeral pairs which includes the lowest ten numbers, the highest number, every number that translates to a single-character Roman numeral, and a random sampling of

other valid number. The point of unit test is not to test every possible input, but to test a representative sample.

(3) Every individual test is its own method, which must take no parameters and return no value. If the method exits normally without raising an exception, the test is considered passed; if the method raises an exception, the test is considered failed.

(4) Here we call the actual **toRoman** function. (Well, the function hasn't be written yet, but once it is, this is the line that will call it). Notice that we have now defined the API for the **toRoman** function: it must take an integer (the number to convert) and return a string (the Roman numeral representation). If the API is different than that, this test is considered failed.

(5) Also notice that we are not trapping any exceptions when we call **toRoman**. This is intentional. **toRoman** shouldn't raise an exception when we call it with valid input, and these input values are all valid. If **toRoman** raises an exception, this test is considered failed.

(6) Assuming the **toRoman** function was defined correctly, called correctly, completed successfully, and returned a value, the last step is to check whether it returned the right value. This is a common question, and the **TestCase** class provides a method, **assertEqual**, to check whether two values are equal. If the result returned from **toRoman** (**result**) does not match the known value we were expecting (**numeral**), **assertEqual** will raise an exception and the test will fail. If the two values are equal, **assertEqual** will do nothing. If every value returned from toRoman matches the known value we expect, **assertEqual** never raises an exception, so **testToRomanKnownValues** eventually exits normally, which means **toRoman** has passed this test.

Also note that the test case is defined in a method with the name begin with “test” in a class derived from the **TestCase** class.

3. Testing for Failure

It is not enough to test that our functions succeed when given good input; we must also test that they fail when given bad input. And not just any sort of failure; they must fail in the way we expect.

Remember our other requirements for **toRoman**:

2) **toRoman** should fail when given an integer outside the range **1** to **3999**.
3) **toRoman** should fail when given a non-integer decimal.

In Python, functions indicate failure by raising exceptions, and the **unittest** module provides methods for testing whether a function raises a particular exception when given bad input.

The test case for testing bad input to **toRoman** will be defined as follows:

```
class ToRomanBadInput(unittest.TestCase):
    def testTooLarge(self):
        """toRoman should fail with large input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, 4000)

    def testZero(self):
        """toRoman should fail with 0 input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, 0)

    def testNegative(self):
        """toRoman should fail with negative input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, -1)

    def testNonInteger(self):
        """toRoman should fail with non-integer input"""
        self.assertRaises(roman.NotIntegerError, roman.toRoman, 0.5)
```

The next two requirements are similar to the first three, except they apply to **fromRoman** instead of **toRoman**:

4) **fromRoman** should take a valid Roman numeral and return the number that it represents.
5) **fromRoman** should fail when given an invalid Roman numeral.

Requirement #4 is handled in the same way as requirement #1, iterating through a sampling of known values and test each in turn. Requirement #5 is handled in the same way as requirements #2 and #3, by testing a series of bad inputs and making sure **fromRoman** raises the appropriate exception.

The test case for testing bad input to **fromRoman** will be defined as follows:

```
class FromRomanBadInput(unittest.TestCase):
    def testTooManyRepeatedNumerals(self):
        """fromRoman should fail with too many repeated numerals"""
        for s in ('MMMM', 'DD', 'CCCC', 'LL', 'XXXX', 'VV', 'IIII'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s)

    def testRepeatedPairs(self):
        """fromRoman should fail with repeated pairs of numerals"""
        for s in ('CMCM', 'CDCD', 'XCXC', 'XLXL', 'IXIX', 'IVIV'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s)

    def testMalformedAntecedent(self):
        """fromRoman should fail with malformed antecedents"""
        for s in ('IIMXCC', 'VX', 'DCM', 'CMM', 'IXIV',
                  'MCMC', 'XCX', 'IVI', 'LM', 'LD', 'LC'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s)
```

Note that we have to specify three custom exceptions in **roman.py** in order to pass the test for bad input (**roman.OutofRangeError**, **roman.NotIntegerError**, and **roman.InvalidRomanNumeralError**). This specification is naturally defined in our test cases.

4. Testing for Sanity

Often you will find that a unit of code contains a set of reciprocal functions, usually in the form of conversion functions where one converts A to B and the other converts B to A. In these cases, it is useful to create a "sanity check" to make sure that you can convert A to B and back to A without losing decimal precision, incurring rounding errors, or triggering any other sort of bug.

Consider this requirement:

6) If you take a number, convert it to Roman numerals, then convert that back to a number, you should end up with a number you started with. So **fromRoman(toRoman(n)) == n** for all **n** in **1..3999**.

The test case for testing **toRoman** against **fromRoman** will be defined as follows:

```
class SanityCheck(unittest.TestCase):
    def testSanity(self):
        """fromRoman(toRoman(n))==n for all n"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            result = roman.fromRoman(numeral)
            self.assertEqual(integer, result)
```

The last two requirements are different from the others because they seem both arbitrary and trivial:

7) **toRoman** should always return a Roman numeral using uppercase letters.
8) **fromRoman** should only accept uppercase Roman numerals (*i.e.* it should fail when given lowercase input).

In fact, they are somewhat arbitrary. We could, for instance, have stipulated that **fromRoman** accept lowercase and mixed case input. But they are not completely arbitrary; if **toRoman** is always returning uppercase output, then **fromRoman** must at least accept uppercase input, or our "sanity check" (requirement #6) would fail.

The test case for testing a character case in Roman numeral will be defined as follows:

```
class CaseCheck(unittest.TestCase):
    def testToRomanCase(self):
        """toRoman should always return uppercase"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            self.assertEqual(numeral, numeral.upper())          # (1)

    def testFromRomanCase(self):
        """fromRoman should only accept uppercase input"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            roman.fromRoman(numeral.upper())                    # (2) (3)
            self.assertRaises(roman.InvalidRomanNumeralError,
                              roman.fromRoman, numeral.lower())
```

(1) The most interesting thing about this test case is all the things it doesn't test. It doesn't test that the value returned from **toRoman** is right or even consistent; those questions are answered by separate test cases. We have a whole test case just to test for uppercase−ness. You might be tempted to combine this with the sanity check, since both run through the entire range of values and call **toRoman**. But that would violate one of fundamental rules in unit testing: each test case should answer only a single question. Imagine that you combined this case check with the sanity check, and then that test case failed. You would have to do further analysis to figure out which part of the test case failed to determine what the problem was. If you have to analyze the results of your unit testing just to figure out what they mean, it's a sure sign that you've mis−designed your test cases.

(2) There's a similar lesson to be learned here: even though "we know" that **toRoman** always returns uppercase, we are explicitly converting its return value to uppercase here to test that **fromRoman** accepts uppercase input. Why? Because the fact that **toRoman** always returns uppercase is an independent requirement. If we changed that requirement so that, for instance, it always returned lowercase, the **testToRomanCase** test case would have to change, but this test case would still work. This was another fundamental rule in unit testing: each test case must be able to work in isolation from any of the others.

(3) Note that we're not assigning the return value of **fromRoman** to anything. This is legal syntax in Python; if a function returns a value but nobody's listening, Python just throws away the return value. In this case, that's what we want. This test case doesn't test anything about the return value; it just tests that **fromRoman** accepts the uppercase input without raising an exception.