Test Driven Development
Unit Testing

Walter Cazzola

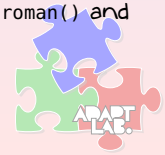
Dipartimento di Informatica e Comunicazione
Università degli Studi di Milano
e-mail: cazzola@dico.unimi.it

Test Driven Development — Unit Testing
Case Study: Roman Numerals

The rules for Roman numerals lead to some observations:

1. The function φ that translates a number from its Arabic form to the Roman one is bijective, that is
   - every number in its Arabic form has only a correct Roman representation
   - φ is invertible
2. There is a limited range of numbers that can be expressed as Roman numerals, specifically 1 through 3999.
3. There is no way to represent 0 in Roman numerals.
4. There is no way to represent negative numbers in Roman numerals.
5. There is no way to represent fractions or non-integer numbers in Roman numerals.

So the `roman.py` module will provide the functions `to_roman()` and `from_roman()`.



Test Driven Development — Unit Testing
Case Study: Roman Numerals

I can imagine that all of you can developed such a module, but what about writing a test case proving it correct?

To write code that checks the correctness of other code is called test-driven development
- `to_roman()` and `from_roman()` can be written and tested as a unit
- separate from any larger program that imports them

Python has a framework for unit testing, the appropriately-named `unittest` module.

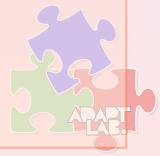
Unit testing is an important part of an overall testing-centric development strategy.
- Before writing code, writing unit tests forces you to detail your requirements in a useful fashion.
- While writing code, unit tests keep you from over-coding. When all the test cases pass, the function is complete.
- When refactoring code, they can help prove that the new version behaves the same way as the old version.

Test Driven Development — Unit Testing
to_roman(): First Check

```
import unittest, roman1 as roman
class KnownValues(unittest.TestCase):
 known_values = ((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'))
  def test_to_roman_known_values(self):
    for integer, numeral in self.known_values:
      result = roman.to_roman(integer)
      self.assertEqual(numeral, result)
if __name__ == '__main__': unittest.main()
```

```
def to_roman(n): pass                    # roman1.py
```

```
[10:50]cazzola@ulik:~/esercizi-pa/tdd>python3 good-tests.py
F
======================================================================
FAIL: test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "good-tests.py", line 18, in test_to_roman_known_values
    self.assertEqual(numeral, result)
AssertionError: 'I' != None
----------------------------------------------------------------------
Ran 1 test in 0.001s
FAILED (failures=1)
```

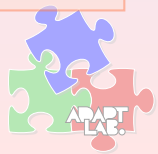


Test Driven Development — Unit Testing

to_roman(): Second Check


```
roman_numeral_map = ((’M’,  1000), (’CM’, 900), (’D’,  500), (’CD’, 400), (’C’,  100), (’XC’, 90),
         (’L’,  50), (’XL’, 40), (’X’,  10), (’IX’, 9), (’V’,  5), (’IV’, 4), (’I’,  1))

def to_roman(n):
  result = ’’
  for numeral, integer in roman_numeral_map:
    while n >= integer:
      result += numeral
      n -= integer
  return result
```

```
[11:29]cazzola@ulik:~/esercizi-pa/tdd>python3 good-tests.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok

----------------------------------------------------------------------
Ran 1 test in 0.002s

OK
```



Test Driven Development — Unit Testing

to_roman(): Testing Bad Inputs


```
class ToRomanBadInput(unittest.TestCase):
  def test_too_large(self): self.assertRaises(roman.OutOfRangeError, roman.to_roman, 4000)
  def test_zero(self): self.assertRaises(roman.OutOfRangeError, roman.to_roman, 0)
  def test_negative(self): self.assertRaises(roman.OutOfRangeError, roman.to_roman, -1)
if __name__ == ’__main__’: unittest.main()
```

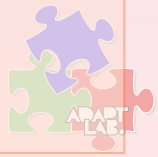
```
[11:59]cazzola@ulik:~/esercizi-pa/tdd>python3 tests.py
.EEE
======================================================================
ERROR: test_negative (__main__.ToRomanBadInput)
to_roman should fail with negative input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "tests.py", line 13, in test_negative
    self.assertRaises(roman.OutOfRangeError, roman.to_roman, -1)
AttributeError: ’module’ object has no attribute ’OutOfRangeError’
======================================================================
ERROR: test_too_large (__main__.ToRomanBadInput)
to_roman should fail with large input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "tests.py", line 7, in test_too_large
    self.assertRaises(roman.OutOfRangeError, roman.to_roman, 4000)
AttributeError: ’module’ object has no attribute ’OutOfRangeError’
======================================================================
ERROR: test_zero (__main__.ToRomanBadInput)
to_roman should fail with 0 input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "tests.py", line 10, in test_zero
    self.assertRaises(roman.OutOfRangeError, roman.to_roman, 0)
AttributeError: ’module’ object has no attribute ’OutOfRangeError’
----------------------------------------------------------------------
Ran 4 tests in 0.004s
FAILED (errors=3)
```

Test Driven Development — Unit Testing

to_roman(): Testing Bad Inputs


```
class OutOfRangeError(ValueError): pass
```

```
[13:41]cazzola@ulik:~/esercizi-pa/tdd>python3 tests.py
.FFF
======================================================================
FAIL: test_negative (__main__.ToRomanBadInput)
to_roman should fail with negative input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "tests.py", line 13, in test_negative
    self.assertRaises(roman.OutOfRangeError, roman.to_roman, -1)
AssertionError: OutOfRangeError not raised by to_roman

======================================================================
FAIL: test_too_large (__main__.ToRomanBadInput)
to_roman should fail with large input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "tests.py", line 7, in test_too_large
    self.assertRaises(roman.OutOfRangeError, roman.to_roman, 4000)
AssertionError: OutOfRangeError not raised by to_roman

======================================================================
FAIL: test_zero (__main__.ToRomanBadInput)
to_roman should fail with 0 input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "tests.py", line 10, in test_zero
    self.assertRaises(roman.OutOfRangeError, roman.to_roman, 0)
AssertionError: OutOfRangeError not raised by to_roman

----------------------------------------------------------------------
Ran 4 tests in 0.004s
FAILED (failures=3)
```

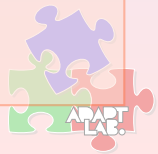


Test Driven Development — Unit Testing

to_roman(): Testing Bad Inputs


```
roman_numeral_map = ((’M’,  1000), (’CM’, 900), (’D’,  500), (’CD’, 400), (’C’,  100), (’XC’, 90),
        (’L’,  50), (’XL’, 40), (’X’,  10), (’IX’, 9), (’V’,  5), (’IV’, 4), (’I’,  1))

class OutOfRangeError(ValueError): pass

def to_roman(n):
  if not (0 < n < 4000):
    raise OutOfRangeError(’number out of range (must be 1..3999)’)
  result = ’’
  for numeral, integer in roman_numeral_map:
    while n >= integer:
      result += numeral
      n -= integer
  return result
```

```
[13:47]cazzola@ulik:~/esercizi-pa/tdd>python3 tests.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok
test_negative (__main__.ToRomanBadInput)
to_roman should fail with negative input ... ok
test_too_large (__main__.ToRomanBadInput)
to_roman should fail with large input ... ok
test_zero (__main__.ToRomanBadInput)
to_roman should fail with 0 input ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.010s

OK
```



Test Driven Development — Unit Testing
from_roman(): What about the Reverse?

What about the reverse? Which are the interesting spots?

- to check the conversion correctness on known Roman numbers

```
def test_from_roman_known_values(self):
  for integer, numeral in self.known_values:
    result = roman5.from_roman(numeral)
    self.assertEqual(integer, result)
```

- to check that `from_roman()` implements the inverse of `to_roman()` for all the admissible values.

```
class RoundtripCheck(unittest.TestCase):
  def test_roundtrip(self):
    for integer in range(1, 4000):
      numeral = roman5.to_roman(integer)
      result = roman5.from_roman(numeral)
      self.assertEqual(integer, result)
```


Test Driven Development — Unit Testing
from_roman(): What about the Reverse? (Cont'd)

```
roman_numeral_map = (('M',  1000), ('CM', 900), ('D',  500), ('CD', 400), ('C',  100), ('XC', 90),
       ('L',  50), ('XL', 40), ('X',  10), ('IX', 9), ('V',  5), ('IV', 4), ('I',  1))

def from_roman(s):
  """convert Roman numeral to integer"""
  result = 0
  index = 0
  for numeral, integer in roman_numeral_map:
    while s[index:index+len(numeral)] == numeral:
      result += integer
      index += len(numeral)
  return result
```

```
[15:20]cazzola@ulik:~/esercizi-pa/tdd>python3 both-tests.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok
test_roundtrip (__main__.RoundtripCheck)
from_roman(to_roman(n))==n for all n ... ok
test_negative (__main__.ToRomanBadInput)
to_roman should fail with negative input ... ok
test_too_large (__main__.ToRomanBadInput)
to_roman should fail with large input ... ok
test_zero (__main__.ToRomanBadInput)
to_roman should fail with 0 input ... ok

----------------------------------------------------------------------
Ran 5 tests in 0.330s

OK
```


Test Driven Development — Unit Testing
from_roman(): Testing Bad Inputs

The problem is to define/understand what is a good and what is not a good Roman number

- too many repeated numerals, i.e., MMMM, DD, CCCC, LL, XXXX, VV, IIII, ...
- repeated (impossible) pairs, i.e., CMCM, CDCD, XCXC, XLXL, IXIX, IVIV, ...
- malformed antecedents, i.e., IIMXCC, VX, DCM, CMM, IXIV, MCMC, XCX, IVI, LM, LD, LC, ...

```
class FromRomanBadInput(unittest.TestCase):
  def test_too_many_repeated_numerals(self):
    '''from_roman should fail with too many repeated numerals'''
    for s in ('MMMM', 'DD', 'CCCC', 'LL', 'XXXX', 'VV', 'IIII'):
      self.assertRaises(roman.InvalidRomanNumeralError, roman.from_roman, s)
  def test_repeated_pairs(self):
    '''from_roman should fail with repeated pairs of numerals'''
    for s in ('CMCM', 'CDCD', 'XCXC', 'XLXL', 'IXIX', 'IVIV'):
      self.assertRaises(roman.InvalidRomanNumeralError, roman.from_roman, s)
  def test_malformed_antecedents(self):
    '''from_roman should fail with malformed antecedents'''
    for s in ('IIMXCC', 'VX', 'DCM', 'CMM', 'IXIV', 'MCMC', 'XCX', 'IVI', 'LM', 'LD', 'LC'):
      self.assertRaises(roman.InvalidRomanNumeralError, roman.from_roman, s)
```


Test Driven Development — Unit Testing
from_roman(): Testing Bad Inputs (Cont'd)

```
class InvalidRomanNumeralError(ValueError): pass
roman_numeral_pattern = re.compile('''
    ^                  # beginning of string
    M{0,3}             # thousands - 0 to 3 Ms
    (CM|CD|D?C{0,3}) # hundreds - 900 (CM), 400 (CD), 0-300 (0 to 3 Cs), or 500-800 (D and 0 to 3 Cs)
    (XC|XL|L?X{0,3}) # tens - 90 (XC), 40 (XL), 0-30 (0 to 3 Xs), or 50-80 (L, followed by 0 to 3 Xs)
    (IX|IV|V?I{0,3}) # ones - 9 (IX), 4 (IV), 0-3 (0 to 3 Is), or 5-8 (V, followed by 0 to 3 Is)
    $                  # end of string
    ''', re.VERBOSE)
def from_roman(s):
  if not roman_numeral_pattern.search(s):
    raise InvalidRomanNumeralError('Invalid Roman numeral: {0}'.format(s))
  result, index = 0, 0
  for numeral, integer in roman_numeral_map:
    while s[index:index+len(numeral)] == numeral:
      result += integer
      index += len(numeral)
  return result
```

```
[15:48]cazzola@ulik:~/esercizi-pa/tdd>python3 both-tests.py -v
test_malformed_antecedents (__main__.FromRomanBadInput)
from_roman should fail with malformed antecedents ... ok
test_repeated_pairs (__main__.FromRomanBadInput)
from_roman should fail with repeated pairs of numerals ... ok
test_too_many_repeated_numerals (__main__.FromRomanBadInput)
from_roman should fail with too many repeated numerals ... ok
test_roundtrip (__main__.RoundtripCheck)
from_roman(to_roman(n))==n for all n ... ok
                    [CUT]
test_zero (__main__.ToRomanBadInput)
to_roman should fail with 0 input ... ok
----------------------------------------------------------------------
Ran 8 tests in 0.321s
OK
```

References

- Jennifer Campbell, Paul Gries, Jason Montojo, and Greg Wilson.
  Practical Programming: An Introduction to Computer Science Using Python.
  The Pragmatic Bookshelf, second edition, 2009.
- Mark Pilgrim.
  Dive into Python 3.
  Apress*, 2009.
- Mark Summerfield.
  Programming in Python 3: A Complete Introduction to the Python Language.
  Addison-Wesley, October 2009.