Unit Testing

Good code

- Good code is:
  - Correct
  - Easy to navigate and to understand
  - Easy to modify
- This means that good code is:
  - Composed of largely independent, single-purpose methods
  - Simple and straightforward, not overly clever
  - Well-documented
  - Supported by an extensive test set

Unit testing

- A *unit test* is a test of a single method or function
- Unit tests may be performed individually, or in any order
- Unit testing results in:
  - Code with fewer errors
  - Methods that are single purpose
  - Methods that are largely independent of one another
  - Programs that are easier to maintain and modify
- Unit tests also provide examples of what each method is supposed to do

Philosophy

- Thorough testing is desirable, but testing is work
- The more work it is, the less it will get done
- Therefore, testing must be made as simple and easy as possible
- Conclusions:
  - Use a testing framework that does most of the work for you
  - Running all the tests should be as simple as a single button click
  - When all the tests pass, the user should see only a success indicator ("OK" in IDLE, green bar in other IDEs)
    - Therefore, methods being tested should not require any input and should not provide any output
  - Failed tests should indicate exactly what and how a method failed

`import` statements

- Larger programs are often written in more than one file (or ***module***)
- Unit tests are usually written in a different module than the module being tested
- To use functions that are in a different module, you need to ***import*** that module
  - **`import`** statements should be the first lines in your module
- For example, suppose you want to call a function named **`myfun`** in a file named **`myprog.py`** -- you can do this in either of two ways:
  1. At the top of the program, say **`import myprog`**
     In the code, call the function by saying **`myprog.myfun(`*args*`)`**
  2. At the top of the program, say **`from myprog import *`**
     In the code, call the function by saying **`myfun(`*args*`)`**

Structure of the test file

- The test file has a moderately complex structure

- ```
  import unittest
  from name_of_module import *

  class NameOfClass(unittest.TestCase):
      # You can define variables
      # and functions here

      # Test methods go here--
      #   the name of each test method
      #   begins with "test_"

  unittest.main()
  ```
  ```

Structure of test methods

- Each test has a name beginning with `test_` and has one parameter named `self`
- Inside the test function is just normal Python code--you can use all the usual Python statements (if statements, assignment statements, loops, function calls, etc.) but you ***should not do input or output***
  - I/O in tests will just slow down testing and make it more difficult
  - For the same reason, the code being tested should also be free of I/O
- Here are the three most common tests you can use:
  - `self.assertTrue(`*boolean_expression_that_should_be_true*`)`
  - `self.assertFalse(`*boolean_expression_that_should_be_false*`)`
  - `self.assertEqual(`*first_expression*`,` *second_expression*`)`
- Of these, `self.assertEqual` gives you more information when it fails, because it tells you the value of the two expressions

Example code to be tested

- This is on file `parity.py`:

```
def is_even(n):
    """Test if the argument is even"""
    return n % 2 == 0

def is_odd(n):
    """Test if the argument is odd"""
    return n % 2 == 1
```

Example test and result

- ```
  import unittest
  from parity import *

  class TestEvenOrOdd(unittest.TestCase)

      def test_even(self):
          self.assertTrue(is_even(6))
          self.assertFalse(is_even(9))

  unittest.main()
  ```
- ```
  >>> ====================== RESTART ======================
  >>>
  .
  ----------------------------------------------------------
  Ran 1 test in 0.032s

  OK
  >>>
  ```

Example of test failure

- Suppose we do: `self.assertTrue(is_even(9))`

- ```
  >>> =========================== RESTART ===========================
  >>> 
  F
  ===================================================================
  ===
  FAIL: test_even (__main__.TestEvenOrOdd)
  -------------------------------------------------------------------
  ---
  Traceback (most recent call last):
    File "/Users/dave/Box Sync/Programming/Python3_programs/
  parity_test.py", line 8, in test_even
      self.assertTrue(is_even(9))
  AssertionError: False is not true

  -------------------------------------------------------------------
  ---
  Ran 1 test in 0.041s

  FAILED (failures=1)
  >>> 
  ```
  ```

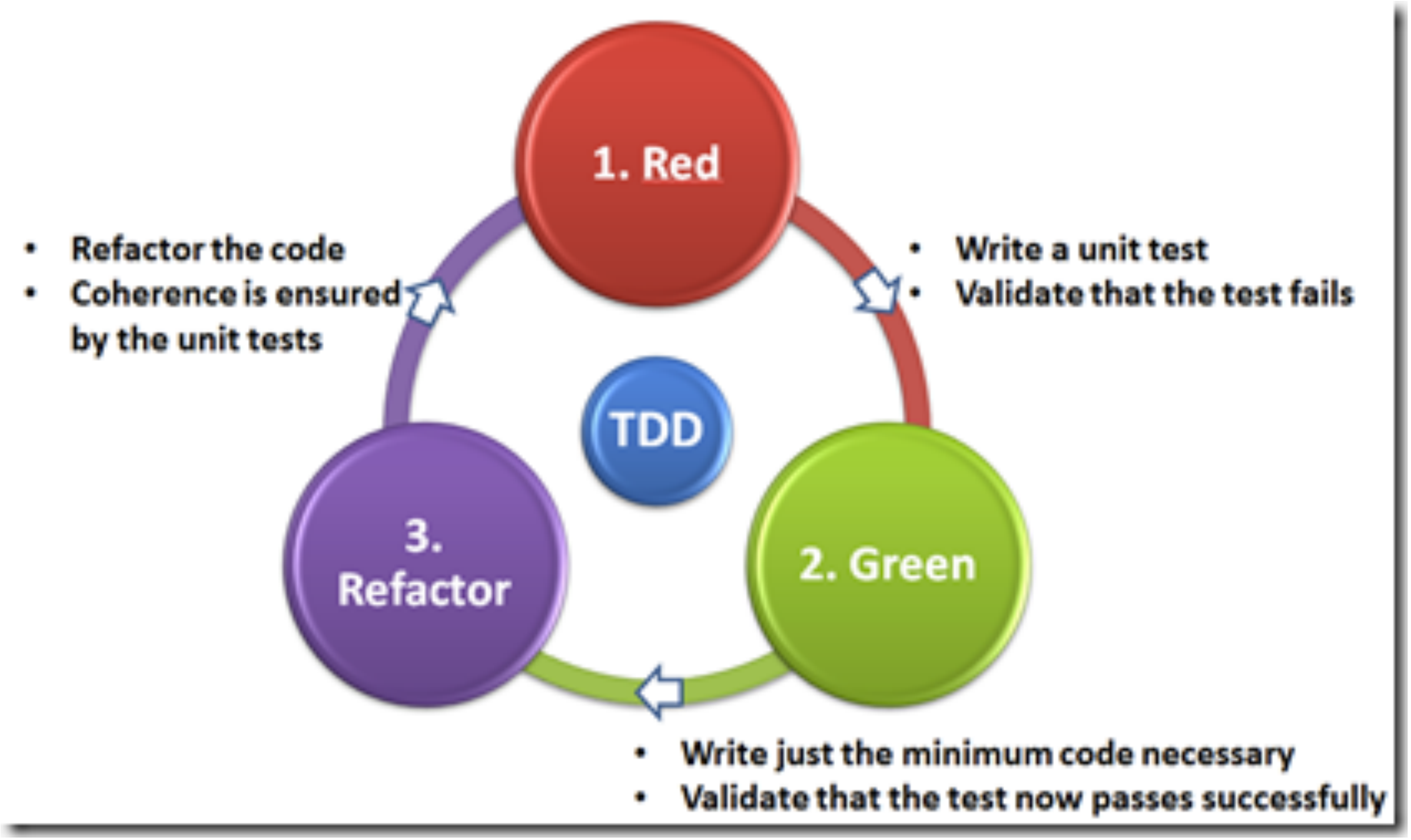
Another example failure

- ```
  def test_arithmetic(self):
      n = 0
      for i in range(0, 10): # do ten times
          n = n + 0.1
      self.assertEqual(1.0, n)
  ```

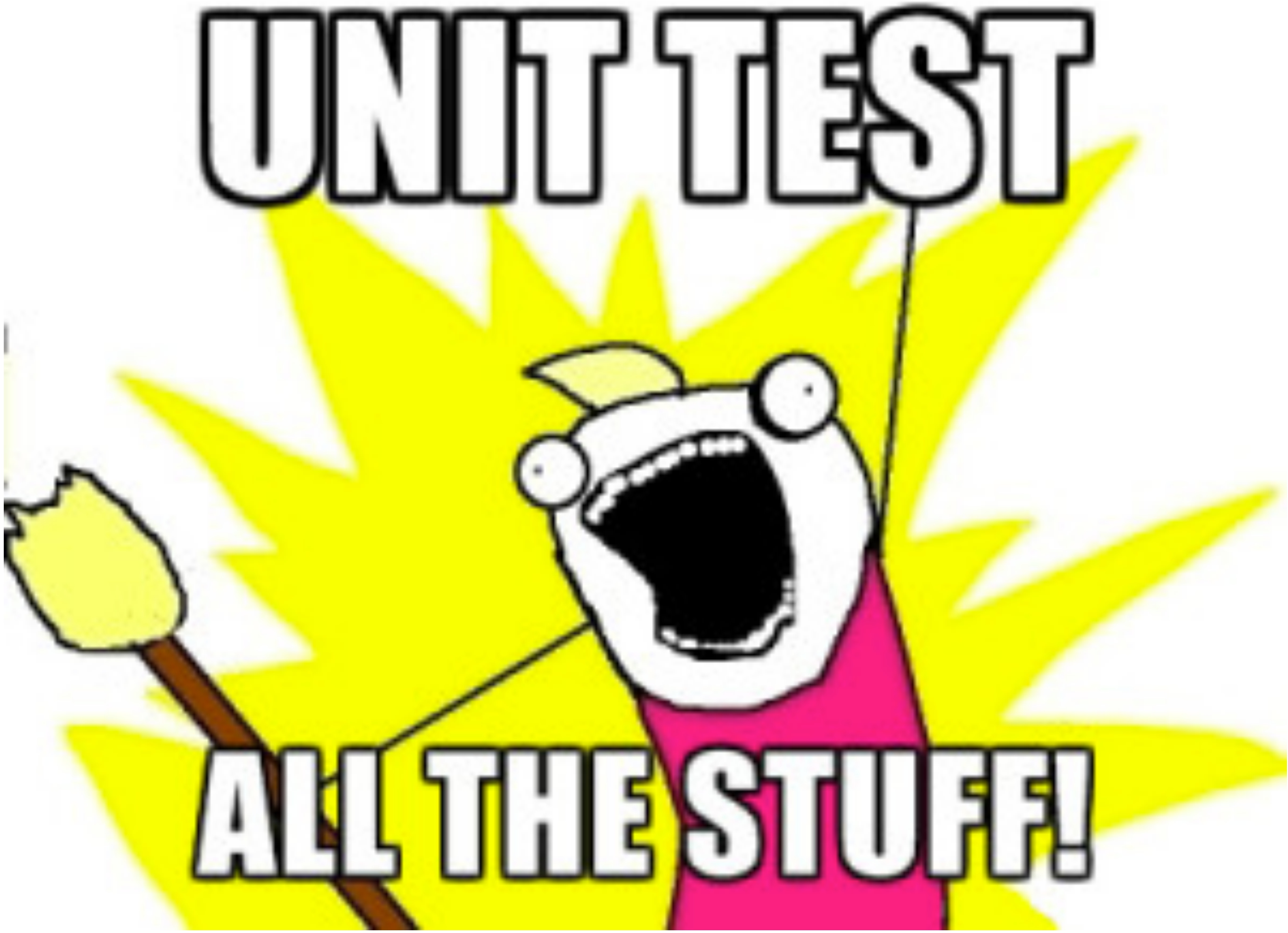
- `AssertionError: 1.0 != 0.9999999999999999`

- **Moral:** Never trust floating point numbers to be exactly equal

- To test floating point numbers, don't use **assertEquals(*x*, *y*)**

- Instead, use **assertAlmostEqual(*x*, *y*)** or **assertAlmostEqual(*x*,*y*,*d*)**, where *d* is the number of digits after the decimal point to round to (default 7)
  ```

Testing philosophy

- When testing, you are **not** trying to prove that your code is correct--you are trying to **find and expose flaws,** so that the code may be fixed
- If you were a lawyer, you would be a lawyer for the prosecution, not for the defense
- If you were a hacker, you would be trying to break into protected systems and networks
    - A ***white hat hacker*** tries to find security flaws in order to get the company to fix them--these are the "good guys"
    - A ***black hat hacker*** tries to find security flaws in order to exploit them--these are the "bad guys"

Testing “edge” cases

- Testing only the simple and most common cases is sometimes called ***garden path*** testing
    - All is sweetness and light, butterflies and flowers
    - Garden path testing is better than nothing
- Of course, you need to test these simple and common cases, but ***don’t stop there***
- To find the most flaws, **also** test the “edge” cases, those that are extreme or unexpected in one way or another

Example "edge" case

- Recall our code for **is_odd**:

```
def is_odd(n):
    """Test if the argument is odd"""
    return n % 2 == 1
```

- Here is another test for it:

```
def test_odd_when_negative(self):
     self.assertTrue(is_odd(-3))
     self.assertFalse(is_odd(-4))
```

- What is the result of `-3 % 2`? Is it `1`, or is it `-1`?

- In either event, here is some better code:

```
def is_odd(n):
     """Test if the argument is odd"""
    return not is_even(n)
```

More test methods

- The following are some of the methods available in test methods:
  - `assertEqual(`*a*`, `*b*`),` `assertEqual(`*a*`, `*b*`, `*message*`)`
  - `assertNotEqual(`*a*`, `*b*`),` `assertNotEqual(`*a*`, `*b*`, `*message*`)`
  - `assertTrue(`*x*`),` `assertTrue(`*x*`, `*message*`)`
  - `assertFalse(`*x*`),` `assertFalse(`*x*`, `*message*`)`
  - `assertIs(`*a*`, `*b*`),` `assertIs(`*a*`, `*b*`, `*message*`)`
  - `assertIsNot(`*a*`, `*b*`),` `assertIsNot(`*a*`, `*b*`, `*message*`)`
  - `assertIsNone(`*x*`),` `assertIsNone(`*x*`, `*message*`)`
  - `assertIsNotNone(`*x*`),` `assertIsNotNone(`*x*`, `*message*`)`
  - `assertIn(`*a*`, `*b*`),` `assertIn(`*a*`, `*b*`, `*message*`)`
  - `assertNotIn(`*a*`, `*b*`),` `assertNotIn(`*a*`, `*b*`, `*message*`)`
  - `assertIsInstance(a, b),` `assertIsInstance(`*a*`, `*b*`, `*message*`)`
  - `assertNotInstance(`*a*`, `*b*`),` `assertNotInstance(`*a*`, `*b*`, `*message*`)`
  - `assertRaises(`*exception*`, `*function*`, `*arg1*`, ..., `*argN*`)`
- Typically *b* and *x* are calls to the method being tested, while *a* is the expected result.

The `setUp` method

- If a method *changes* a globally accessible value, rather than just returning a value, then the order in which methods are called is important
  - Unit tests may be performed individually, or in any order
  - Good programming style minimizes the use of global variables
- If you define a **`setUp(self)`** method, it will be called before each and every test method
  - The job of **`setUp`** is to reset all global values to a known state
  - In the **`setUp`** method, don't forget to declare your variables to be **`global`**

Interrelated methods

- In general, a unit test should test *just one* method
  - You might have multiple tests for the same method, to test different aspects of it
- Some methods are interrelated and need to be tested together
  - For example, pushing something onto a stack and popping something from a stack
  - For more complicated interactions, you can create "mock objects"
    - This is an advanced topic, not covered here

How much is enough?

- **Rule:** Write ***at least one*** test method for each computational method
    - You can write more than one test method (with different names) for methods that do more than one thing, or handle more than one case
- There is no need for redundant testing; if `is_odd` works for both 5 and 6, it probably also works for 7 and 8
    - ...but it may not work for negative numbers, so test those as well
- There is no need to write unit tests to see if Python itself works
- **Rule:** Test every ***case*** you can think of that might possibly go wrong

Do it backwards and iteratively!

- The obvious thing to do is to write the code first, then write the tests for the code
- Here is how it's done by experts at writing testable code:
    1. Begin by writing a simple test for the code you plan to write
    2. Write the code
    3. Run the test, and debug until everything works (remember, errors might be in the test itself)
    4. Clean up (***refactor***) the code, making sure that it still works
    5. If the code doesn't yet do everything you want it to do, write a test for the next feature you want to add, and go to step 2.
- This is approach is called ***Test Driven Development*** (***TDD***)

Why is TDD better?

- When you start with the code, it is easy to write a function that is too complicated and difficult to test
- Writing the test first helps clarify what the code is supposed to do and how it is to be used
- Writing the test first helps keeps functions small and single-purpose
- TDD promotes steady, step-by-step progress, and helps avoid long, painful debugging sessions
- TDD simplifies and encourages code modification when updates are needed

Refactoring

- ***Refactoring*** is changing the code to make it better (cleaner, simpler, easier to use) ***without changing what it does***
- Refactoring should be a normal part of your programming
  - Each time you get a function to work correctly (or even sooner), you should see if there's a way you can make it better
- Common refactorings include:
  - Changing the name of variables or functions to better express their meaning
  - Eliminating useless or redundant code (such as `if success == True:`)
  - Breaking a function that does two things into two single-purpose functions
  - Simplifying a complex arithmetic expression by giving names to the parts, then using those names in the expression



http://elstarit.nl/?p=157

Special code in the program

- Programs typically have a method named **`main`** which is the starting point for everything that happens in the program

  - The program can be started automatically when it is loaded by putting this as the last line in the program:

    ```
    main()
    ```

  - If you are testing the individual methods of the program, you don't want to start the program automatically

- The following "magic" code, placed at the end of the program, will call the **`main`** method to start the program *if and only if* you run the code from this file:

  - ```
    if __name__ == '__main__':
        main()
    ```

- If you run tests from a separate file, the above code does not call the main method to start the program running
    ```

The End



http://www.adamslair.net/blog/?p=1463