Unit Testing Overview

```
#=================================================
def read_devices_info(devices_file):

    devices_list = []

    file = open(devices_file,'r')    # Open the CSV file
    csv_devices = csv.reader(file)   # Create CSV reader

    # List comprehension put CSV into list of lists
    return [dev_info for dev_info in csv_devices]
```

Code

Unit test code

```
import util

#=================================================
class TestUtil(unittest.TestCase):

    def test_csv_read(self):

        print '\n**** Testing reading CSV file ***'

        # Test that we can correctly read in CSV values
        csv_devices_list = util.read_devices_info(csv_test_input_filename)
        self.assertEqual(csv_devices_list, csv_test_devices_list, "...")
```


Unit Testing

Reasons for Unit Testing

- **Early defect detection**: Finding bugs early reduces costs later
- **Code quality**: including more thorough error checking
- **Regression testing** and **automation**: run scripts to make sure code still works

Unit Test Components

- `import unittest`
- Test `setUp` and `tearDown`
- Test case (specific test of one piece of functionality) and test suite (combination of related tests grouped together)
- Inherit `unittest.TestCase`

Unit Test Basics

- Import `unittest`
- Test class inherits `'TestCase'`
- Override `setUp()` and `tearDown()`
- Each method is a test case
- Test methods begin with "`test`"
- Main: run `unittest.main()`

```
import unittest
...
class TestUtil(unittest.TestCase):

   def setUp(self):
      ...
   def test_csv_read(self):
      ...
   def test_csv_write(self):
      ...
   def tearDown(self):
      ...

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


Testing for Success and Failure

- **Assert**: Use `assert` to test for success or failure
- `assertEqual`, `assertNotEqual`, `assertTrue`, `assertFalse`, `etc.`
- **Process**: Import modules, use `setup()` to create necessary environment, and make function or method calls to actual code.

```
import util

class TestIO(unittest.TestCase):

   def test_csv_read(self):
      util.read_devices_info(...)
      self.assertEqual(...)

   def test_csv_write(self):
      util.write_devices_info(...)
      self.assertTrue(...)
```

Unit Testing Overview

"Unit testing" is the testing of individual units of code, often in an automated fashion. Unit testing frameworks exist for most programming languages, and Python is no different. The earliest and simplest framework for Python is found in the library called `unittest`.

The general idea of unit testing is that unit test code is written, to exercise individual units of functionality in an application. Thus, your application will have some number of lines of code for the actual application itself, and it will also have a similar collection of code whose purpose is to test the actual application code.

The application code is the function `read_devices_info`, in the `util.py` module. The purpose of this code is to take as input a device's file name, open the file, and read it using the CSV library, into a data structure that will hold device information.

```
#===============================================================================
def read_devices_info(devices_file):

    devices_list = []

    file = open(devices_file,'r')   # Open the CSV file
    csv_devices = csv.reader(file)  # Create the CSV reader for file

    # Use list comprehension to put CSV data into list of lists
    return [dev_info for dev_info in csv_devices]
```

Below is unit test code, whose purpose is to test the functionality of that specific `read_devices_info` function. At a very high level, the unit test code will invoke the `read_devices_info` function with a known file, and will examine the output of the function, comparing it to the known correct output. If the values match, the test passes; if it does not match, there may be an issue with the function.

```
import util

#===============================================================
class TestUtil(unittest.TestCase):

    def test_csv_read(self):
        print '\n**** Testing reading CSV file ***'

        # Test that the code correctly read in CSV values
        csv_devices_list = util.read_devices_info(csv_test_input_filename)
        self.assertEqual(csv_devices_list, csv_test_devices_list, "...")
```

Unit Testing Basics

The following piece of code shows the structure of a class used for unit testing:

```
import unittest
...
class TestUtil(unittest.TestCase):
   def setUp(self):
      ...
   def test_csv_read(self):
      ...
   def test_csv_write(self):
      ...
   def tearDown(self):
      ...
if __name__ == '__main__':
   unittest.main()
```

You must do the following in order to create unit tests for your application:

- **Create module:** Create your unit test module file, for example, `test_util.py`.
- **Import** `unittest`: Import `unittest`, the unit testing library, which you will use for creating your testing class and test cases.
- **Define class:** Define a class that inherits from the `unittest.TestCase` class. Your class can have whatever name you like, but it must be derived from `unittest` `TestCase` class.
- **Methods:** Each of your methods will be an individual test case, that will be run by Python when you execute your unit test Python module.
- **Method names:** Be aware that every method name that is a test case *must* begin with the string 'test'. Methods that do not begin with this string are ignored when the unit test is run.
- **Run:** In order to run your unit test, your main application code will simply call `unittest.main()`. Python, through functionality in the `unittest` class, will automatically begin executing your unit tests, which are defined as methods within the class.

setUp() and tearDown()

Quite often your tests will need to set up something at the beginning of each test, and take it down at the end. Python's `unittest` provides this capability through the `setUp()` and `tearDown()` methods of the `TestCase` class.

- **setUp()**: This method is called before every test is run. Use this method to initialize variables and objects for testing in your test cases.
- **tearDown()**: This method is called after every test is run. Use this method to remove any resources you have created, to clean up variables and objects if necessary.

The following code sample shows more detail regarding the `setUp()` and `tearDown()` methods that are part of your unit testing class:

```
class TestUtil(unittest.TestCase):
   def setUp(self):
      # for setting up before
      # each test is run

   # Test cases

   def tearDown(self):
      # for tearing down when
      # tests are complete
```

Your unit tests may also make use of constant values that never change – these values can be stored in global variables within the unit test module, accessible to all test cases.

Testing for Success or Failure

The purpose of each test case is to determine whether the unit of functionality being tested is performing correctly. In order to do test functionality, the `TestCase` class provides several 'assertion' methods. Some of the more frequently used assertions are:

- **Assert Equal**: test whether two items are equal
- **Assert Not Equal**: test whether two items are not equal
- **Assert True**: test whether the item has the value True
- **Assert False**: test whether the item has the value False
- Many other tests are available including Assert Greater Than, Assert Less Than, Assert Is, and Assert Is Instance.

The sequence of operations for all test cases is:

- Call setUp() to initialize the unit test objects and variables that will be used in this specific test
- Execute the code in the test case (for example, read information from a CSV file)
- Use assertions to test if the code executed successfully or not
- Call tearDown() to restore the initial state of the unit test.