Writing Unittest Rules

- Name your unit tests clearly and consistently: avoid non-descriptive unit tests names
- Prefix the names of your test functions/methods with `test_` and the names of your test classes with Test
- Save your test code in files that start with `test_`
- Put unit tests for your classes in a separate class file as well. Hence, by separating your class implementation and your unit tests, you will automatically be prevented from testing private methods and private properties.
- You shouldn't test private methods and private properties because doing so will cause your unit tests to become a barrier to refactoring. You will always have the option to *change the internal implementation of your classes* should the need arise.
- **What to test?** Broadly speaking, you should test your custom business logic. How thoroughly you test that business logic will probably vary between situations.
    - On one end of the spectrum, you might choose to implement just a few tests that only cover the code paths that you believe are most likely to contain a bug.
    - On the other end of the spectrum, you might choose to implement a large suite of unit tests that are incredibly thorough and test a wide variety of scenarios.
- You shouldn't consider the percentage of code coverage to be an end-goal though. Instead, you should strive to increase the state coverage of your unit tests.

  e.g. `double getFraction(Integer a){ return 1/a;}`

  You should probably test a few different inputs for this method, even if it means that you will have achieved 100% code coverage several times over. Three different states that you might consider testing are a "positive" input, a "negative" input and a "0" input.
- Unit tests should always create their own test data to execute against.
- Set Up All Conditions for Testing: typically, methods perform some sort of operation upon data; so in order to test your methods, you'll need to set up the data required by the method. This might be **as simple as** declaring a few variables, or **as complex as** creating a number of records in a database.
- You should be sure to write unit tests that verify your code behaves as expected in **normal** scenarios as well as in more **unexpected** scenarios, like boundary conditions or error conditions.
- Test unexpected conditions:
    - Bad Input Values: One potentially unexpected condition that the code might encounter is an unexpected value, like null, being passed to the push() method. You have a few implementation options for handling this scenario.
    - Boundary Conditions: e.g. a list can only contain 1,000 records; if a 1,001th object were to be added to list-based stack implementation, an exception would be thrown.
- Don't make unnecessary assertion: have only one logical assertion per test
- Assertion syntax: no special assertion syntax in py.test; you can use the standard Python assert statements

  e.g.

```
def test_assert_introspection():
    assert True         # assertTrue()
    assert 1 == 1       # assertEqual()
    assert not 1 == 2   # assertNotEqual()
    assert not False    # assertFalse()

...
```

- Exception handling: use the raises() function that takes the expected exception type as the first parameter. The other parameters are either:
  - *a string specifying the function or method call that is supposed to raise the exception* or
  - *the actual callable, followed by its arguments*

  e.g.

```
def test_sort_exception(self):
    py.test.raises(NameError, "self.alist.sort(int_compare)")
    py.test.raises(ValueError, self.alist.remove, 6)
...
```

- Test only one code unit at a time: your architecture must support testing units (i.e., classes or very small groups of classes) independently, not all chained together.
- Avoid dependencies between tests: a test should be able to stand on its own. It should not rely on any other test, nor should it depend on tests being run in a specific order.
- Organize your tests in hierarchies and test suites by creating a directory tree and placing/grouping your test files in the appropriate directories.
- Then you can just run py.test with no arguments in the directory that contains your tests, the tool will search the current directory and its subdirectories for files that start with `test_` , then it will automagically invoke all the test functions/methods it finds in those files.
- Verify the results are correct