**ITWorks Unittest Testing Standards**

* A testing unit should focus on one tiny bit of functionality and prove it correct.
* Each test unit must be fully independent. Each test must be able to run alone, and also within the test suite, regardless of the order that they are called. The implication of this rule is that each test must be loaded with a fresh dataset and may have to do some cleanup afterwards. This is usually handled by setUp() and tearDown() methods.
* Try hard to make tests that run fast. If one single test needs more than a few milliseconds to run, development will be slowed down or the tests will not be run as often as is desirable. In some cases, tests can’t be fast because they need a complex data structure to work on, and this data structure must be loaded every time the test runs. Keep these heavier tests in a separate test suite that is run by some scheduled task, and run all other tests as often as needed
* Learn your tools and learn how to run a single test or a test case. Then, when developing a function inside a module, run this function’s tests frequently, ideally automatically when you save the code.
* Always run the full test suite before a coding session, and run it again after. This will give you more confidence that you did not break anything in the rest of the code.
* It is a good idea to implement a hook that runs all tests before pushing code to a shared repository.
* If you are in the middle of a development session and have to interrupt your work, it is a good idea to write a broken unit test about what you want to develop next. When coming back to work, you will have a pointer to where you were and get back on track faster.
* The first step when you are debugging your code is to write a new test pinpointing the bug. While it is not always possible to do, those bug catching tests are among the most valuable pieces of code in your project.
* Use long and descriptive names for testing functions. The style guide here is slightly different than that of running code, where short names are often preferred. The reason is testing functions are never called explicitly. square() or even sqr() is ok in running code, but in testing code you would have names such as test\_square\_of\_number\_2(), test\_square\_negative\_number(). These function names are displayed when a test fails, and should be as descriptive as possible.
* When something goes wrong or has to be changed, and if your code has a good set of tests, you or other maintainers will rely largely on the testing suite to fix the problem or modify a given behavior. Therefore the testing code will be read as much as or even more than the running code. A unit test whose purpose is unclear is not very helpful in this case.
* Another use of the testing code is as an introduction to new developers. When someone will have to work on the code base, running and reading the related testing code is often the best thing that they can do to start. They will or should discover the hot spots, where most difficulties arise, and the corner cases. If they have to add some functionality, the first step should be to add a test to ensure that the new functionality is not already a working path that has not been plugged into the interface.