Writing Unit Tests

How to ensure code works and is maintainable, whether it is C++, Java, or Python

Unit tests are typically automated tests written and run by software developers to ensure that a section of an application (known as the "**unit**") meets its design and behaves as intended.[2] In procedural programming, a unit could be an **entire module**, but it is more commonly an **individual function** or **procedure**

- Wikipedia entry on Unit Tests

CTA Software Programming Standards (Revision 1a)

4.4 Functionality and Quality Tests

Unit Tests All Code developed for CTA shall include Unit Tests covering at least 60% of lines of code (80% for Python and non-compiled code) at the level of functions, methods, and classes.

[7] http://docs.astropy.org/en/stable/development/docguide.html

We'll go in detail into the Programming Standards document in a future Developer Seminar.

Part of a more general test suite

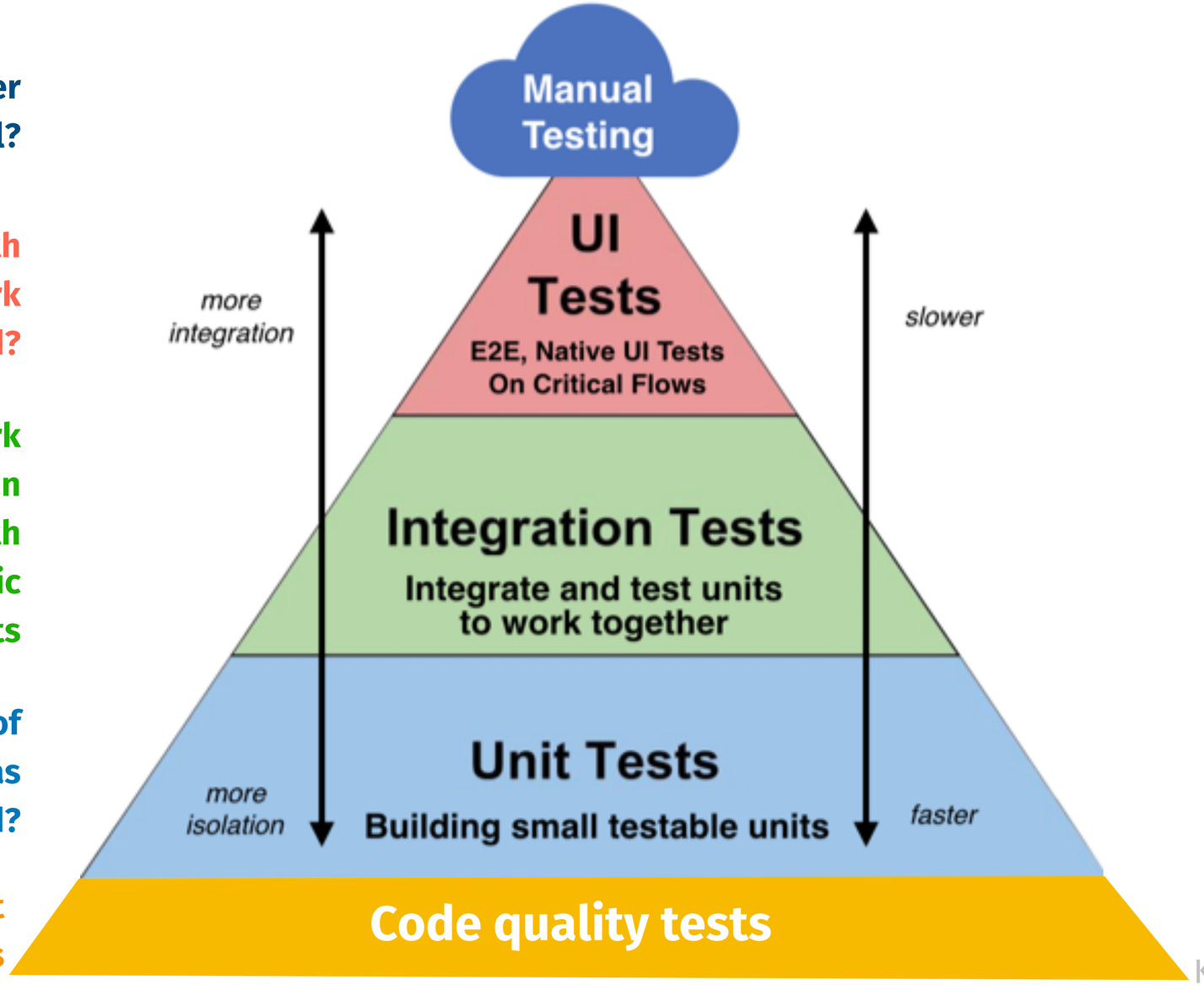
How is the user experience in general?

Does interaction with the user interface work as expected?

Do the units work together as an application? or with larger/more realistic inputs

Does each piece of functionality work as expected?

Does the code compile and meet style, security, and design guidelines



Code checking and Compilation

A C/C++/Java compiler will alert you to *syntax* and *typing* errors **at *compile time***

- All code gets compiled and checked at once.
- Still benefits from **static code analysis** (e.g. **cppcheck**) to catch common non-syntax errors

The Python compiler will alert you to *syntax* errors only **at *run time*.**

- Not all code gets compiled, only that which is imported and run
- "*Duck-typed*", so in general no way for the compiler to check correctness of types
- To provide equivalent of checks provided by a C/C++ compiler:
  - static code analysis (e.g. **pylint**, **pyflakes**) find common problems
  - type checkers (**mypy**) alert user to possible typing errors (if the developer includes some *type hints* in the code)

My code compiles! Why do I need tests?

Compilation only ensures code is syntactically correct

It does not ensure that it *works as expected or meets requirements*

Even fully compiled code from C, C++, Java, etc NEED UNIT TESTS.

You probably already write tests without thinking...

If you develop like me, you:

- write some throw-away dummy code that you use to see if the algorithm is doing what you expect (sometimes only when debugging!)
- Sometimes it's bits of Jupyter Notebooks, sometimes its small example files.
- Often it is not very formal: doesn't test everything, perhaps lots of interaction needed, will probably never work in the future...

Unit Tests just formalize this: stress automation, and provide a place to keep around bits of testing/example code so they can be repeated systematically.

For the next section, we will focus on unit testing in Python (but don't worry, we will get to C++ and Java as well)

Choosing a Unit Test Framework

Unit Test Frameworks help you write tests:

- Provide a common way to write individual tests, including ways to check for problems or expected behavior
- Organizes and **run the tests** for you, giving **failure reports**, **coverage reports**, and even jump into a **debugger** on failure
- Provide features to write advanced tests:
  - **setup input** common to multiple tests
  - run **parameterized** tests (same tests with a range of inputs)
  - **mock-up complex inputs** so tests mixing multiple modules can be decoupled
  - **optional tests** , for example for those that need to fetch large data files for input, or for unimplemented features

Unit Test Frameworks

pytest.org

For Python we highly recommend *PyTest*

- for greater ease of use over the built-in UnitTest library

```
% conda install pytest
```

```
% pip install pytest
```

- Writing tests is nearly trivial, and just based on *name* of the function and file
  - test files have filename that starts with "**test_**"
  - tests themselves are simple functions starting with "**test_**"
- Common test setup is provided by *fixtures* (see later)
- Provides a lot of advanced functionality on top

Making tests as *easy as possible* helps ensure developers *write them*.

Writing Tests: The simplest case

The function you want to test

```python
def weighted_mean_1d(values, weights):
    scale = np.sum(weights)
    return np.multiply(values, weights).sum() / scale
```

The most basic test: does it run without error?

```python
def test_weighted_mean_1d():
    mean = weighted_mean_1d( [1,2,3,4], [1,1,1,1] )
```

But that's somewhat pointless: there should be at least an *assertion*:

```python
    assert (mean - (1+2+3+4)/4.0) < 0.001
```

What else could you test here?

- more thorough **correctness**: e.g. do weights have the right effect?
- **varying inputs**: what happens if the user gives another input type than expected? What about edge cases? Null/None values?
- are **errors handled** as expected?

Running tests (pytest)

Find and run all tests:

```
pytest
```

Run tests under a specific directory

```
pytest mymodule/some_subdir
```

Run all tests that match a keyword:

```
pytest -k hillas
```

Mix and match options

```
pytest ctapipe/image -k hillas
```

Run tests, but stop when one fails and enter the debugger:

```
pytest --pdb
```

Run only the tests that failed the last time

```
pytest --last-failed
```

Run all tests, but skip ones that already passed, and continue with last failed:

```
pytest --stepwise
```

- any many more (see --help)

Skipped and XFail tests

For tests that cannot be run now *(use sparingly)*

- feature not yet implemented, input data not available yet

```
import pytest

@pytest.mark.skip(
    reason="no way of testing this"
)
def test_some_thing():
    ...

@pytest.mark.xfail   # expected to fail
def test_future_feature():
   ...
```

- See also **skipif** (e.g. skip test if some dependency version is too low, or other condition you specfy)

Xfail inside a test

```
import slow_module

if slow_module.slow_function():
    pytest.xfail("slow_module taking too long")
```

Run the xfails tests anyway

```
pytest --runxfail
```

What to do when a test fails?

If not obvious from the traceback and error message:

- turn on **--showlocals** (see all local variables and their value)
- turn on **--pdb** to enter the pdb debugger:
  - then use standard debugger commands

```
u    - go up the stack one function
d    - go down the stack one function
l    - list the code currently executing
ll   - longer list of code
print(variable) - see a variable
```

Checking Test Coverage

A test coverage tool will tell you if all *lines of code* are covered by a test

- Generally included in your Unit Test Framework
- For PyTest, install the **pytest-cov** plugin (*conda install pytest-cov*)

pytest-cov:

```
pytest --cov=<module> --cov-report=html --cov-report=term
```

enable coverage

generate an HTML report

also summarize to the terminal

Test for expected Exceptions

```python
import pytest

def test_fails_as_expected():
    with pytest.raises(TypeError):
        my_function_that_expects_int("a string value")
```

- This test will **fail** if my_function_that_expects_int ***does not raise an TypeError exception*** when given a string*!*
- It will also fail if no exception is raised.

Parameterized Tests

Run the same test on multiple inputs

- Useful for testing options or wide ranges of input
- Each iteration is realized as its own test:
  - test_my_function[10] - FAILED
  - test_my_function[20] - PASSED
  - test_my_function[50] - PASSED

```
@pytest.mark.parameterize("num_iterations", [10,20,50])
def test_my_function(num_iterations):
    x = np.arange(100)
    assert my_function(x, num_iterations=num_iterations) == True
```

Test Fixtures: Same input for multiple tests

Problem:

- I want to write many test that use the same input
- That input requires a complex setup

Solutions:

- In some unit test frameworks, you write **setup** and **teardown** functions that can prepare inputs for a *group of tests.*
- in **pytest**, you have the notion of "**fixtures**", which are more generally re-usable results that can be shared between tests and can be re-executed for various scopes:
  - per test *function*
  - per *module*
  - per testing *session*

```python
class Fruit:
    def __init__(self, name):
        self.name = name

    def __eq__(self, other):
        return self.name == other.name


@pytest.fixture(scope="module")
def my_fruit():
    return Fruit("apple")


@pytest.fixture
def fruit_basket(my_fruit):
    return [Fruit("banana"), my_fruit]


def test_my_fruit_in_basket(my_fruit, fruit_basket):
    assert my_fruit in fruit_basket
```

Built-in Fixtures:

List all available (built-in + user-defined) with:

```
pytest --fixtures
```

Some useful built-in fixtures:

- **tmp_path** - a temporary directory as a *pathlib.Path* instance for each test (prevents collisions between tests, and stops you from spraying files everywhere)

```
def test_my_stuff(tmp_path):
    with open(tmp_path / "output.txt") as out:
       out.write("xxxxx")
```

- **caplog -** capture the python logger output, so you can test for correctness

```
def test_baz(caplog):
    func_under_test()
    for record in caplog.records:
        assert record.levelname != "CRITICAL"
    assert "wally" not in caplog.text
```

- **capsys -** capture stdout and stderr, so you can use them in tests (test for outputs)
- **monkeypatch -** modify objects or system variables non-destructively

for example *ctapipe*, there is a fixture called **example_event** that can be used as input for any test that needs an event to work with

Running Tests Automatically

Much better than running yourself: *do it automatically!*

- Run whenever a PR is opened, or when a commit is pushed!
- Test on multiple architectures
- No code can be merged until tests pass

Requires:

- Code is in a code management repository (GitLab, GitHub)
- A Continuous Integration system is available (GitLabCI, GitHub Actions, Azure Pipelines)

```yaml
name: CI

on: [push, pull_request]

jobs:
  tests:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        python-version: [3.7, 3.8]

    steps:

      - name: Tests
        run: |
          pytest --cov=ctapipe --cov-report=xml

      - uses: codecov/codecov-action@v1
```

Example *Github Actions* Pipeline running tests in a step

Use Unit Tests to Develop Code!

Write the unit test *first*, before you write the actual code!

- Let's you focus on the design before the implementation
- Leads to better code and clearer APIs
- When implementing, you know when you are done when the tests pass

XFail tests can be used (sparingly) for long term plans!

- Add tests for future features you absolutely want to implement
- If you don't have time to implement now, can mark them as "expected to fail" (xfail) for now, and turn that off later when time permits.

Make a unit test when fixing Bugs!

A reported bug in your code (or even feature request):

- Implies **something was missed** in your current set of tests.
- To fix the bug, first **add a unit test that re-produces it**
- Then fix the code until this test passes

Why?

- Ensure the bug doesn't re-occur later in future development!
- enforce behavior

Use tests to help guide other developers!

Unit tests are guards!

- What if somebody else modifies code that you wrote?
- What tests would you want to add to be sure that they don't break an something else that relies on it working in a particular way?

Unit tests are examples!

- Common problem in code: where to put examples?
- Unit tests are perfect for this: they not only test that code works, but show how to use code
- Often more enlightening than the documentation!

Unit testing in C++ (GTest)

https://github.com/google/googletest/


```
#include "gtest/gtest.h"

TEST (SquareRootTest, PositiveNos) {
    EXPECT_EQ (18.0, square-root (324.0));
    EXPECT_EQ (25.4, square-root (645.16));
    EXPECT_EQ (50.3321, square-root (2533.310224));
}

TEST (SquareRootTest, ZeroAndNegativeNos) {
    ASSERT_EQ (0.0, square-root (0.0));
    ASSERT_EQ (-1, square-root (-22.0));
}

int main(int argc, char **argv) {
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

Supports most of the same concepts as PyTest:

- parameterized tests
- *expectations*: raise error in test, but continue
- *fixtures* via SetUp/ TearDown methods

```
Running main() from user_main.cpp
[==========] Running 2 tests from
[----------] Global test environm
[----------] 2 tests from SquareR
[ RUN      ] SquareRootTest.Posit
..\user_sqrt.cpp(6862): error: Va
  Actual: 50.332
Expected: 50.3321
[  FAILED  ] SquareRootTest.Posit
[ RUN      ] SquareRootTest.ZeroA
[       OK ] SquareRootTest.ZeroA
[----------] 2 tests from SquareR

[----------] Global test environm
[==========] 2 tests from 1 test
[  PASSED  ] 1 test.
[  FAILED  ] 1 test, listed below
[  FAILED  ] SquareRootTest.Posit

 1 FAILED TEST
```

Unit testing in C++ (Catch)

https://github.com/catchorg/Catch2

```
unsigned int Factorial( unsigned int number ) {
    return number <= 1 ? number : Factorial(number-1)*number;
}
```

```
#define CATCH_CONFIG_MAIN  // This tells Catch to provide a main() - only do this in one cpp file
#include "catch.hpp"

unsigned int Factorial( unsigned int number ) {
    return number <= 1 ? number : Factorial(number-1)*number;
}

TEST_CASE( "Factorials are computed", "[factorial]" ) {
    REQUIRE( Factorial(1) == 1 );
    REQUIRE( Factorial(2) == 2 );
    REQUIRE( Factorial(3) == 6 );
    REQUIRE( Factorial(10) == 3628800 );
}
```

Unit tests in Java: JUnit5

https://junit.org/junit5/docs/current/user-guide/

```java
import static org.junit.jupiter.api.Assertions.assertEquals;
import example.util.Calculator;
import org.junit.jupiter.api.Test;

class MyFirstJUnitJupiterTests {

    private final Calculator calculator = new Calculator();

    @Test
    void addition() {
        assertEquals(2, calculator.add(1, 1));
    }

}
```

Supports most of the same concepts as PyTest:

- parameterized tests
- *assumptions* vs *assertions* (when to stop test)
- *fixtures* via setUp/ tearDown methods

General Recommendations Summary

- Test all possible inputs you might expect a user to try
- Test all known error/failure cases
- Test all logic paths
- Use tests to write code!
- Use tests to help guide other developers!
- Whenever a bug/issue is identified or fixed, add a test to make sure it doesn't come back!
- Automate testing with a CI system (for another seminar)

Backup

Advanced Topics: Mocks

Problem: I want to write a test for a function that depends on another very complex object/library/function that has too many dependencies and is already tested elsewhere.

Solution: Use a Mock object as a stand-in for that complex object/library/function, letting you avoid importing it.

Mock is part of the python standard library (unittest.mock), and can be used directly. There is also a friendly wrapper for *pytest* that provides a mocker fixture (https://github.com/pytest-dev/pytest-mock/):

```
% conda install pytest-mock
```