PyTests: GrADS Test Suite

From OpenGrads Wiki

The automatic testing suite for GrADS is called *PyTests* and is implemented using PyUnit

Contents

(http://pyunit.sourceforge.net/), Python's standard unit testing framework. PyUnit is Python's version of JUnit, the de facto unit testing framework for Java written by Kent Beck and Erich Gamma. Although *PyTests* are written in Python, they are meant to test all aspects of GrADS, not only elements of the GrADS-Python interface. PyUnit is an easy to use and elegant object oriented framework for writing automatic tests; individual tests can be written using the Python interface to GrADS or using the GrADS native scripting language (or a mix of both). This document outlines the general procedure for running and interpreting the existing tests and for developing new GrADS unit tests within the PyUnit framework.

Running the existing tests

If all you want do is to run the existing tests, it couldn't be simpler. The only requeriment is that you have python and PyUnit, something that comes standard in most installations of Python v2.3 and newer. Once you download the GrADS sources you do the usual `configure` and `make`:

```
% ./configure
% make
```

If the build is sucessful just type

```
% make check
```

for running the test suite. If all goes well you should see output like this:

```
 Welcome to the PyUnit-based GrADS Unit Tests

 Testing with GrADS Data Files from ./data/
 Testing with GrADS binaries from ../src/

+ Will test GrADS binary <../src/gradsc>
+ Will test GrADS binary <../src/gradsnc>
+ Will test GrADS binary <../src/gradsnc4>
+ Will test GrADS binary <../src/gradshdf>
+ Will test GrADS binary <../src/gradsdods>

test_01_Open (TestModelFile.Classic_grb) ... ok
test_02_Prints (TestModelFile.Classic_grb) ... ok
...
test_01_Open (TestModelFile.DAP_pdef) ... ok
test_03_Display (TestModelFile.DAP_pdef) ... ok

----------------------------------------------------------------------
Ran 140 tests in 150.737s

OK
```

If something goes wrong, instead of `OK` you will see a report about the number of tests that have failed. The preceding lines would also indicate which particular tests have failed, e.g.,

```
test_01_Open (TestModelFile.Classic_grb) ... ERROR
```

The string `test_01_Open` is the name of the test which have failed, while the string in parenthesis (`TestModelFile.Classic_grb`) indicates that the testhave failed for the classic Grads binary `gradsc`, reading the `model.grb` test file. See file `TestModelFile.py` under directory `pytests/` for examing the particular test code. In this example, you should look for the function `test_01_Open` under the `TestModelFile` class. In some cases, the actual test code is implemented through a GrADS native script, and the Python function in question is a simple wrapper which invokes the script.

Tests are run only for the binaries that have been built. For example, if `gradshdf` and `gradsnc4` are missing you will see the output

```
+ Will test GrADS binary <../src/gradsc>
+ Will test GrADS binary <../src/gradsnc>
- Not testing GrADS binary <../src/gradsnc4>, file missing
- Not testing GrADS binary <../src/gradshdf>, file missing
+ Will test GrADS binary <../src/gradsdods>
```

In the testing report there is a subtle distinction between an *error* and a test *failure*. An *error* occurs when the test code in question raises an exception (say, GrADS returns an error), while a *failure* is the result of an assertion that could not

be verified. For example, the command `query file` returns an unexpected number of variables on file. In any case, an *error* or a *failure* should be considered as an abnormal condition and its cause investigated.

Writing your own tests

Although you can write most of your tests in the GrADS native scripting language, a working knowledge of Python, PyUnit and object oriented concepts is necessary to hook up your tests in the framework. (Alternatively, you can ask for help from a developer familiar with the testing code suite.) For getting started with Python consult the Python Documentation (http://www.python.org/doc/) site, in particular the Tutorial (http://docs.python.org/tut/tut.html) and Beginner's Guide (http://wiki.python.org/moin/BeginnersGuide) to Python. *PyUnit* is documented in the Python Library Reference (http://docs.python.org/lib/lib.html) (see the section on unit testing framework (http://docs.python.org/lib/module-unittest.html); reading the first section on *Basic Example* plus perhaps the next section is sufficient for having a working knowledge of PyUnit. So, do your homework before reading the next sections of this document.

Implementation overview

By now you should have read the *PyUnit* documentation and have a working knowledge of how to write and organize basic tests. In particular you should know what a *test fixture* is.

Standard test cases are run for each of the available GrADS binaries (`gradsc`, `gradshdf`, etc.), with test fixtures defined for the standard test files in GRIB, NetCDF and HDF formats (`model.grb, model.nc` and `model.hdf`), as well as for a binary file using a file based PDEF metadata (`pdef.dat`.) As of this writing, fixtures for station data and OPeNDAP datasets are yet to be implemented.

The core of the python code is contained in file `pytests/TestModelFile.py`. This file contains the two main testing classes (`TestModelFile` and `TestPdefFile`) which are subclasses of the `unittest.TestCase` superclass. Specific classes for testing each binary/data file are constructed by subclassing `TestModelFile` and `TestPdefFile`. The function `run_all_tests()` assembles the test suite and run all tests. The stand alone script `grads_tests.py` allows for specification of the data and binary directories,

```
Usage: grads_tests.py [options]

Options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -b BINDIR, --bindir=BINDIR
                        Directory for GrADS binaries
```

```
  -d DATADIR, --datadir=DATADIR
                        Directory for GrADS binaries
  -v LEVEL, --verbose=LEVEL
                        verbose level (default=2)
```

(The script `grads_tests.sh` are used by the GNU autotools and is simple wrapper around `grads_tests.py`.)

The initial implementation of *PyTests* includes only very basic tests, and some of them only asserts if a given command can be completed without raising an exception. However, there are tests that ascertain whether each of the data files are read correctly and whether each variable can be displayed with the correct default contour levels. Extensive use is made of script `lats4d.gs` which writes out the input file in several formats. These files are read them back and carefully compared the retrieved fields with the corresponding input fields. As already mentioned, these tests need to be extended to include station data and input OPeNDAP datasets. In addition, one needs to detected the presence of each GrADS feature (say, availability of `printim`) and skip those tests when a given feature is missing from the binary being tested.

Contribution of testing code by the GrADS community at large is a very desirable and hopefully attainable goal. A well crafted testing suite is a very important ingredient for a stable and reliable multi-platform application such as GrADS.

Writing unit tests in python

This requires some familiarity with the `grads.py`, the GrADS client class for python, see Python Interface to GrADS. If you want to write a tested case called *new_test* be run under the `TestModelFile` fixture all you need to is to add the following function to the `TestModelFile` class:

```
def test_new_test(self):
   ... enter your code here ...
```

and this test will be executed for all GrADS binaries/input file formats. Study file `TestModelFile.py` and reuse any of the existing fixtures to create new ones for testing novel capabilities.

Writing unit tests as native GrADS scripts

Individual test cases can be written entirely in the GrADS native scripting language, and only a single line of code is necessary the wrapper python test code. For example, let's say you have written a `new_test.gs` script to be run under the `TestModelFile` fixture. All you need to is to add the following function to the `TestModelFile` class:

```
def test_new_test(self):
   self.ga.cmd('run new_test')
```

Your test case script`new_test.gs` should be self contained and designed to run unattended. It should run quietly if all is well. If the script determines that the test was not successful, it should write the following output:

```
 say '<RC> 1 </RC>'
 say '</IPC>'
```

This output will cause the Python interface to raise a `GrADSError` exception and the testing framework will then consider the test as returning an error. More sophisticated hybrid scripts can also be written where the output of the script is captured in the pyhton test code (via method `rword`) and compared to some expected result.