BRIGHAM YOUNG UNIVERSITY, DEPARTMENT OF PHYSICS AND ASTRONOMY

# FORTPY Unit Testing Framework Documentation

Conrad W Rosenbrock

September 11, 2013

## 1 QUICKSTART

Instead of reading the entire document from start to finish, here is a quick checklist of how to enable unit tests for a subroutine.

- Create an output template that describes how to compare the output files that will be created by the tests.
- Alter the fortran code to add a fortpy XML element (as a comment) on the first line of the output files that need to be compared.
- Determine what the method needs to run successfully: list any methods that need to be run first, variables that need values initialized etc.
- Create an XML file with the same name as the fortran code file that contains the method to be tested. Add a testing group to it with tags for the pre-reqs and variables that the method needs to run.

Once those steps have been completed, you can run the unit testing script located in fortpy/scripts/runtests.py. PS: you may need to read some of the documentation after all if you don't know how to do any of the things in the list. To get you started, here is the tag that needs to be at the top of output files to compare. If it isn't there, they are assumed to be version 1.

```xml
<fortpy version="2"></fortpy>
```

# 2 Overview

The python package fortpy.testing allows unit tests to be dynamically generated using XML tags that decorate a subroutine or function. This document outlines the various XML tags and attributes available and how they change the behavior of the test generation and execution.

## 2.1 Steps Performed to Generate a Test

Whenever the unit testing framework runs, it performs the following in order:

- Parse all modules in the specified directory as well as any dependencies of those modules.
- Look through all the XML doctags to find testing groups.
- If a testing group was found, complete the steps in the next list; otherwise do nothing.

For each module method that has a testing group:

- Create a folder for the module.method.
- Examine the code files that the subroutine or function needs to compile and see which have been modified.
- Copy any new/modified files across to the folder.
- Generate a *tester.f90* fortran program.
- Generate a Makefile for the program with the module in the correct order for dependencies.
- Compile the program; if it compiles run all tests specified in the testing group.

When the fortran program is generated, the framework determines which variables are needed and initializes them based on the directives in the testing group. Any executable dependencies (i.e. methods that need to be run before the one being tested) are executed in the correct order. Results of comparing the output files to the model output files are saved to a *filename.results* file in the test directory.

## 2.2 Testing Groups as Decorators

The heart of all unit testing is the **<group>** tag with attribute *purpose*="testing". Any subroutines/functions without a testing group (i.e. the group tag with purpose set to testing) are ignored by the unit test generator.

The following elements have meaning within the group tag. Each is described in its own section below.

- **<prereq>** specifies an executable dependency that needs to be run before the unit test will execute.
- **<instance>** specifies that an instance of a variable needs to have its value changed or one of its methods run.
- **<outcome>** specifies a set of execution tests to run and how to evaluate the results.
- **<mapping>** changes the default matching of parameter names to local program variables.
- **<global>** specifies a custom variable that needs to be initialized in the fortran program for use by all the methods in the unit test.

Because testing groups can become large and obstruct the readability of the code, the framework supports writing docstrings in a separate file that has the same name as the fortran code file, but an extension of xml instead of f90. An example testing group is shown below.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<fortpy mode="docstring">
  <decorates name="derivative_structure_generator.gen_multilattice_derivatives">
    <group name="stuct_enum_tests" purpose="testing" staging="./testruns/">
      <outcome target="./struct_enum.out" compare="./struct_enum.out.{}"
               cases="001,002,003,004,005,006" sourcepath="./tests/enum"
               sources="struct_enum.in.{}" rename="struct_enum.in"
               template="struct_enum.out.xml" comparepath="./tests/enum" />
      <global name="dFull" type="real(dp)" modifiers="pointer"
              default="> null()" dimensions=":,:" />
      <global name="labelFull" type="integer" modifiers="pointer"
              dimensions=":,:" />
      <global name="digitFull" type="integer" modifiers="pointer"
              dimensions=":" />
      <global name="equivalencies" type="integer" modifiers="pointer"
              dimensions=":" />
      <global name="fname" type="character" kind="80"
              default=" 'struct_enum.in'" />
      <global name="cRange" type="integer" modifiers="pointer"
              dimensions=":,:" />
      <global name="LatDim" type="integer" />
      <prereq method="io_utils.read_input"
              paramlist="title,LatDim,parLV,nDFull,dFull,k,equivalencies,
                    nMin,nMax,eps,full,labelFull,digitFull,fname,
                    cRange,conc_check" />
      <instance name="platTyp" conditionals="LatDim==3:'b'|LatDim==2:'s'" />
    </group>
  </decorates>
```

```
</fortpy>
```

It generates the following fortran code:

```
!!<summary>Auto-generated unit test for derivative_structure_generator.
!!gen_multilattice_derivatives using FORTPY. Generated on 2013-09-10
!!15:44:37.636712.</summary>
PROGRAM UNITTEST_gen_multilattice_derivatives
  use derivative_structure_generator
  use io_utils, only: read_input
  use fortpy

  integer :: nmax
  logical :: full
  character(1) :: plattyp
  character(80) :: title
  integer :: LatDim
  integer :: k
  real(dp) :: eps
  real(dp), pointer :: dFull(:,:) => null()
  integer, pointer :: equivalencies(:)
  logical :: conc_check
  integer, pointer :: cRange(:,:)
  integer :: nmin
  character(80) :: fname = 'struct_enum.in'
  integer :: ndfull
  integer, pointer :: labelFull(:,:)
  integer, pointer :: digitFull(:)
  real(dp) :: parlv(3,3)

  call read_input(title, LatDim, parLV, nDFull, dFull, k, equivalencies, &
                  nMin, nMax, eps, full, labelFull, digitFull, fname, &
                  cRange, conc_check)
  if (LatDim==3) then
    platTyp = 'b'
  elseif (LatDim==2) then
    platTyp = 's'
  endif
  call gen_multilattice_derivatives(title, parLV, nDFull, dFull, k, nMin, &
                                    nMax, pLatTyp, eps, full, labelFull, &
                                    digitFull, equivalencies, conc_check, cRange)

END PROGRAM UNITTEST_gen_multilattice_derivatives
```

### 2.3 Additional Parameter Attributes

The unit testing framework gives meaning to these additional attributes when they appear on a **<parameter>** tag of a method.

- ***range*** specifies a list of min:max values of the same type as the parameter for each dimension. Right before the method is tested, auto-generated code loops through the variables values to make sure that each one falls within the range.
- ***random*** specifies that for unit-testing purposes, the value of the auto-generated variable for this parameter should be random. For a multi-dimensional variable, random values are assigned to each element in the array. If a *range* is specified, all the random values will fall within the range.
- ***regular*** when "true", the framework will automatically create a variable of the same name and type in the calling program's context. If *random* is not specified, the value will need to be initialized manually using a *prereq* or *instance* tag. If multiple methods define a regular variable with the same name, the framework will croak if they are not of the same type with the same modifiers. In that case, either *terminate* the pre-req chain or remove the *regular* from one of the parameters.

When the documentation refers to auto-matching, it means the auto-generation of variables based on the name and type of the **<parameter>** that had the *regular* attribute.

**NOTE:** the *range* and *random* attributes haven't been implemented yet.

## 3 Testing Group Tags and Attributes

### 3.1 <global> Tag

The global tag gives definition and initialization information for those variables that will be declared as variables in the fortran program. Since the context of the program is a parent to all methods contexts called from the program, we call them "global" variables.

```
<global name="dFull" type="real(dp)" modifiers="pointer"
        default="> null()" dimensions=":,:" />
```

- ***default*** specifies the default value to assign to the variable as part of initialization. Can be the name of an existing variable or a valid fortran function. The contents of the attribute will be pasted directly into the program after the "=", e.g. int i =[default]. Any spaces etc. that need to appear after the equals should be in the attribute.
- ***dimensions*** specifies that an array of the type should be defined. The number of elements in each dimension is specified separated by commas.
- ***kind*** specifies the precision (kind) of int or real types, or the name of a derived type being defined.

- ***modifiers*** lists definition modifiers that should appear after the type declaration. Possible values are: allocatable, pointer, etc. The dimension keyword should not appear here, rather the dimensions should be declared using *dimensions*.

- ***name*** specifies the name to use when defining and referencing the variable within the program context. It should match the names of method parameters if auto-matching is used; otherwise the *paramlist* attribute of pre-requisite calls, or **<mapping>** tags for the method being tested, should be used.

- ***type*** specifies a valid fortran type for the variable. If the variable is a derived type, the fortran type should be "type" and the name of the derived type should appear in *kind*.

## 3.2 <INSTANCE> TAG

The instance tag allows the values of variables to be changed between calls to pre-requisite/dependency methods and the actual executable being tested. The variable's value can be changed via direct assignment or a call to one of its methods if it is a fortran class.

```
<instance name="platTyp" conditionals="LatDim==3:'b'|LatDim==2:'s'" />
```

- ***conditionals*** specifies a list of possible assignments to make based on certain logic tests. If the condition starts with else, it should appear last and will be treated as an else part of the if statement; otherwise, the statements are turned into "if" or "else if" statements based on the order in which they appear. Separate statements should be "|"-separated while the condition:assignment should be colon-separated; see the example.

- ***method*** for a derived type (i.e. fortran class) specifies the method within the type that should be executed. If the method requires parameters, they should be explicitly noted in call; e.g. method="run(param1, param2)". The framework will determine if the method is a subroutine or function and call it appropriately.

- ***name*** the name of the variable that needs to have its value changed.

- ***value*** an explicit value to assign to the variable. Can be any valid fortran code based on the context of the assignment.

## 3.3 <MAPPING> TAG

Depending on how the method's parameters are defined, global variable names that are auto-generated by a pre-requisite may be different than the calling signature of the method being tested. In this case, you can specify a mapping to tell the framework that a parameter with name "A" should actually be specified by a variable with name "B".

```
<mapping name="latTyp" target="platTyp" />
```

- ***name*** the name of the *parameter* of the method being unit tested.

- ***target*** the name of the *variable* in the calling program's context.

### 3.4 <OUTCOME> Tag

The outcome tag specifies which input to use when running the executable and which output files from the execution need to be tested for conformity. In cases where multiple tests have identical setup, cases can be specified to ease implementation.

```
<outcome target="./struct_enum.out" compare="./struct_enum.out.{}"
        cases="001,002,003,004,005,006" sourcepath="./tests/enum"
        sources="struct_enum.in.{}" rename="struct_enum.in"
        template="struct_enum.out.xml" comparepath="./tests/enum" />
```

- ***cases*** a list of identifiers to specify multiple executions of this outcome. The *sources* and *compare* attributes are formatted using each case value in turn for multiple executions. A separate execution folder gets created for each case in the main tests folder.
- ***compare*** a "?"-separated list of model output files/values that the execution output should be compared to. The order of the elements should match the order in *target*. "?" is used because it is one of two characters in python that throw unconditional errors, so they won't interfere with code evaluation of values.
  - If the model output is a file, the name should start with "./", where . represents the path specified in *comparepath*. The file name is treated as part of a format string and python's **.format()** will be called on the text using the case identifier as the parameter. If no cases are specified, any format braces will be treated as part of the file name.
  - If the model output is a value, it should be entered as it would in a python console. Python's **eval()** is used to interpret the code.
- ***comparepath*** specifies the path, *relative to the code folder* from which model output files will be retrieved (i.e. the . represents the folder where the .f90 files are stored).
- ***generator*** specifies the name of a subroutine that should be called in order to generate the output file for comparison. The subroutine is called as-is, any parameters it needs should be specified here in the attribute. If those parameters don't exist as part of automatic matching, they should be declare using **<global>** tags in the testing group.
- ***mode*** specifies the comparison mode to use from the compare template. If not specified, the mode "default" will be used. To understand how comparison modes work, see that section of the documentation.
- ***rename*** a comma-separated list of new names for the input files. For example, if the input files in the sources folder are "input1,input2,input3" for cases 1, 2 and 3 but the program expects a file with the name of "input", you can specify that it should be renamed to input using this attribute. If one file doesn't need to be renamed, but others do, specify the same name for that file. In other words, there must be a matching entry *for each* file in *sources* if the attribute is specified.

- ***runtime*** specifies a tolerance on how long the unit test should take to run. Specify a range like "3-10m" where m = minutes and h = hours. The runtime attribute doesn't support runtimes longer than 24 hours. A unit test shouldn't take that long!
- ***sourcepath*** specifies the path to the input files *relative to the code folder.*
- ***sources*** a comma-separated list of input files that the unit tests needs to run. These names follow the same conventions as the *compare* file names. They will be processed with **.format()** and the case identifier (if specified). The source files should be in the *sourcepath* directory.
- ***target*** a comma-separated list of names of output files or program variables whose values need to be checked for consistency after the program has executed.
  - If the target is a file, it must start with "./" where . is represents the path to the folder that the program was executed in.
  - If the target is a variable name (includes derived type members) its value will be written to a file by the fortpy module that is automatically included in compilation by the framework. It may be compared to a hard-coded value or another file specified in *compare*. If the variable is a derived type, it should have a public member-procedure called **test_output**(*filename*) that saves its representation for testing purposes to the filename specified. Filenames have type character(80).
- ***template*** a comma-separated list of XML template filenames to use for comparing the model outputs to those generated by the executable.

## 3.5 <PREREQ> TAG

Often, before a method can be tested, many other methods need to be called to read input files, pre-process data etc. These methods can be specified in a **<prereq>** tag. **IMPORTANT!** the framework assumes that you have put the **<prereq>** and **<instance>** tags in the order that they need to be executed.

When a pre-req is specified, the framework will examine its testing groups to determine if it also has any pre-reqs specified. If it does, all of them are chained in the correct order so that they execute correctly. This also means that any auto-parameters specified by a pre-reqs testing group will be initialized automatically. If a parameter with the same name exists in multiple methods being executed, the framework assumes that they are the **same** variable. If this is not correct, the chaining should be prevented using the *terminate* attribute on the pre-req. In that case, *all* pre-reqs that are required should be specified explicitly and *paramlist* should be used to specify which variables go in which places in the calling signature.

```
<prereq method="io_utils.read_input"
       paramlist="title,LatDim,parLV,nDFull,dFull,k,equivalencies,
               nMin,nMax,eps,full,labelFull,digitFull,fname,
               cRange,conc_check" />
```

- ***method*** a module.subroutine that should be executed before the method being tested.
- ***paramlist*** a comma-separated list of variables in the calling program's context in the order that they should be passed to the subroutine call.
- ***terminate*** if "true", the framework will not follow the dependency chain of the subroutine to determine if it has any pre-req or variable specifications for it to run correctly.

## 4 File Comparison Across Versions

Part of the unit testing framework's task is to compare the output of the executable that it generates with some "model" output files that are known to have the right answer. A problem arises in that code changes over time. As improvements or new features are added, the output files change and a simple **diff** comparison between them can't tell if they are actually related. When new functionality is added, we still expect the methods to be able to reproduce old results (if those results were actually true). With minor modifications to the fortran code, the framework will be enabled to compare output files from different file versions.

### 4.1 Output File Templates

An output file template is an XML file that specifies the structure of output files and which parts of output files are common between versions of the same file. The naming convention is to use the name of the output file and append a ".xml" to the end; e.g. *structs.out* becomes *structs.out.xml.* The framework assumes that the directory which holds all the templates is stored in a folder called "templates" within the code folder. If that is not the case, you should specify a custom folder as a argument to the unit testing script.

An output template has four sections: **<preamble>**, **<body>**, **outcomes** and **<comparisons>**. The preamble of an output file refers to those entries in the file that are not repeated for each element of an output set. For example, an output file may have a collection of structures with the first line in the file specifying how many there are. In that case, the line describing how many there are is part of the preamble. The individual structures would form the body. The body, then, is a template of a single element that is repeated many times within the output.

The outcomes section specifies which entries can be ignored because they are not relevant to the content (e.g. uncommented text entries that describe data).

The comparison section defines how individual elements in the output file should be compared. For example, if there is inherent randomness in a result, a tolerance can be specified and as long as the corresponding entries are "close enough", the comparison would be successful.

Each of these sections in the output file is covered in detail in the sections below. Here is an example of a complete output template for multiple versions of a standard output file.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<fortpy mode="template" versions="1,2">
  <comments versions="1,2">#</comments>
  <body stop="EOF" key="step.temperature" versions="1">
    <line id="step" type="int,float,int,float" values="1,3,2,*"
          names="stepnum, temperature, energy, cv, mcsteps,
                 rejected, concentrations" />
  </body>
  <comparisons versions="1,2" mode="strict">
    <compare id="step.energy" operator="finite" tolerance="0.02E-02" />
    <compare id="step.cv" operator="finite" tolerance="1E-14" />
    <compare id="step.rejected" operator="finite"
            tolerance="\$['mcsteps']/100" />
  </comparisons>
</fortpy>
```

## 4.2 <PREAMBLE> SECTION

***versions*** is the only attribute on the **<preamble>** tag; it specifies which versions are represented by its contents. Although this could easily be ascertained by enumerating all the children and then looking for unique versions, it is just as easy to put it as an attribute.
The preamble section can have two types of children, **<line>** and **<lines>**. Most of their attributes are common. We will treat those first:

- ***id*** a name to give the value extracted for this line in the output file.
- ***names*** specifies a list of names to assign to elements in the line. If a list of consective elements should be treated as a vector, specify a name followed by "*n" where *n* is the number of consecutive elements to take into the list. The total number of elements specified in names (with *n) should equal the total number of values. If there are fewer names than values, all remaining values are saved in the last name.
- ***type*** a comma-separated list of data types that appear in the line and the order in which they appear. For example, "1 3 5 0.0005 10 10" would be "int,float,int".
- ***values*** the number of values of each *type* to read. In the preceding example, values would be "3,1,2" meaning read 3 ints, 1 float and then 2 ints. If there are a variable number of items in the line, use "*". When "*" is used, *all* the remaining items on the line are extracted using the type associated with "*" in the list. There needs to be a one-to-one correspondence between *type* and *values*.
- ***versions*** specifies which output file versions have this line.
- ***store*** instructs the framework to make a value from the line available globally to any **<lines>** that appear after it. The format is name=$ where $ represents the list of values extracted from the line. The expression on the RHS of "=" can be any valid python

expression. It will be evaluated using **eval()** after replacing all $ with the name of the list containing the values; e.g. "nclusters=$[0]-1" will take the first value of the line, subtract 1 and store it in a variable called "nclusters" for use by other **<lines>**.

When *store* appears in the **<preamble>** the values it stores are available to any **<lines>** in the preamble and **<body>**. When it appears on a **<line>** inside the body, its values are only available to lines within the *same*, current, body element iteration.

The following attributes only appear in **<lines>**. The lines tag is almost identical to **<line>** except that it repeats *count* times. The alternative would be to have *count* identical **<line>** elements in a row in the template.

- ***count*** is the number of times to repeat the definition. The value can be an integer or the name of a variable that was saved using *store*.

## 4.3 <BODY> SECTION

The body is also composed of **<line>** and **<lines>** elements that have identical functionality. The difference between **<body>** and **<preamble>** is that the body definitions are repeated until a stop condition is met. In other words, the preamble is stepped through exactly once and each line in the preamble is matched with the corresponding line in the output file. When the body is processed, its definitions are stepped through repeatedly until the stop condition is met. Each time the body template is processed, it creates a body "block".

The body has extra attributes to define how it repeats and how to match the sets of values that it parses:

- ***count*** for a finite stop condition, specifies how many times to repeat the body template.
- ***key*** if elements of the body appear in a different order between versions of files, a key can be specified on which to compare body blocks. The key is an id.name combination from a **<line>** whose value will be used to match up body blocks.
- ***stop*** specifies when the body will stop iterating. Possible values are "finite" or "EOF". When finite is specified, the body will repeat exactly *count* times. When "EOF" is specified, the body will keep repeating until it runs out of lines to interpret.

## 4.4 <OUTCOMES> SECTION

The main purpose of an outcomes section is to describe how line values affect the outcome of a file comparison. Its children are **<ignore>** elements, each of which has a single attribute.

***id*** specifies the id of a line whose values should not be used in computing the percentage match between files.

## 4.5 <COMPARISONS> SECTION

As mentioned above, the comparisons section defines how specific line values are compared. The children of **<comparisons>** are **<compare>** elements. Comparisons are always used by the framework: if no comparisons section is specified, it generates one with the default *operator* of "equals" for all lines and line.name combinations.

- ***id*** specifies the line id or id.name pair that the comparison will be applied to.
- ***operator*** can either be "equals" or "finite". If the operator is finite, a *tolerance* formula can be applied to the values when determining whether they should be considered equal.
- ***tolerance*** a formula to use when comparing the elements specified by *id*. The $ represents the dictionary of named values compiled for the line if its *names* attribute was specified. Only named values can be compared with finite tolerance comparisons.

## 4.6 BUILT-IN COMPARISON TEMPLATES

FORTPY ships with some default templates for comparing straight-forward files.

**integer.xml** compares single integers, vectors and 2-D arrays.

```
<?xml version="1.0" encoding="UTF-8"?>
<fortpy mode="template" versions="1">
  <comments versions="1">#</comments>
  <body stop="EOF" versions="1">
    <line id="autovar" type="int" values="*" versions="1" />
  </body>
</fortpy>
```

**float.xml** compares single floats, vectors and 2-D arrays.

```
<?xml version="1.0" encoding="UTF-8"?>
<fortpy mode="template" versions="1">
  <comments versions="1">#</comments>
  <body stop="EOF" versions="1">
    <line id="autovar" type="float" values="*" versions="1" />
  </body>
</fortpy>
```