Repeated Test Framework Documentation

Release 0.0.1

Tony Flury

Aug 12, 2017

Contents

CHAPTER 1

Basics

Repeated Test Framework

Introduction

The repeated Test Framework is designed to be used with the `unittest` standard library module (unittest for Python 2.7, unittest for Python 3.5), to make to generate multiple test cases against the same functionality where the difference between the test cases is different test input and differing expected results.

Features

The Framework provides the following features :

- Supports Python 2 and Python 3
- Easy to use
  - Uses a list of dictionaries (or any Iterable of mappings) to define the data for the test cases.
  - Requires only a single generic test function which takes the test case data and executes the test of the functionality.
  - Can decorate a entirely empty unittest.TestCase class - no boiler plate coded needed within the class.
  - Using the default settings, ensures a unique and predictable set of test method names, and useful documentation strings for each test case.
  - The automatically generated test methods work correctly with unittest module module default test detection, loaders, execution and reporting functionality.
  - Supports the use of the normal commandline usage of the unittest module, including execution of specific test cases.
- Behind the scenes
  - Automatically generates a test method on a unittest.TestCase, one for each entry the test data list/Iterable.

  - By generating unique documentation strings and test names, ensures useful test result reporting from the unittest module.
  - By generating multiple test methods, ensures test separation so that testing continues after a test failure.
- Also
  - Allows for customisation of the name and the documentation strings of the generated test method, using any of the data from the relevant test_case.
  - Provides additional decorators allowing the application of unittest test method decorators (`skip`, `skipIf` etc) to one or more of the automatically generated test cases. Can also apply your own arbitrary test method decorators to the generated test case methods.
  - Can combine Automatically generated test methods and explicitly provided test method on the same unittest.TestCase class.

See Using the Framework for full details of how to use the Framework, including how to customise the Framework, and how to apply decorators to the generated test methods.

See Why Use the Framework for a more detailed comparison of the Framework against other traditional ways of using the unittest module to achieve the same multiple test cases for the same functionality item with different data.

Installation

Installation is very simple :

```
$ pip install repeated-test-framework
```

To upgrade an existing installation use

```
$ pip install --upgrade repeated-test-framework
```

Getting Started

The following code snippet will illustrate the simplest use of the Framework to execute a small number of test case against the multiplication operation - a trivial example which is still illustrative of the key points.

```
from repeatedtestframework import GenerateTestMethods

def test_method_wrapper(index, a, b, result):
    def test_method(self):
        """The actual test method which gets replicated"""
        self.assertEqual( a * b, result)
    return test_method

@GenerateTestMethods(
    test_name = 'test_multiplication',
    test_method = test_method_wrapper,
    test_input = [  {'a':1, 'b';2, 'result':2 },
                    {'a':2, 'b':2, 'result':4 },
                    {'a':3, 'b':2, 'result':6 },
                    {'a':3, 'b':4, 'result':11 } ] )
class TestCases(unittest.TestCase):
    pass
```

Although the example above is trivial, it does illustrate the key features of the framework as noted.

- The data to be used is provided as a list of dictionaries; the `input_data` attribute on the GenerateTestMethods decorator.
- A `test_name` attribute is provided - which is a human readable string which is included verbatim into the test method name - as such it can only include alphabetic, numeric and underscore (_) characters.
- Regardless of the number of test data items the decorator only needs a a single test execution method (`test_method` in the example) is required. The Framework replicates this method into the multiple test methods on the decorated class.
- The framework does require the test function to be wrapped in method which accepts the attributes from the `input_data` iterator - in the example below this wrapping function is `test_method_wrapper`. As shown in the example, the wrapper function it does not need to do anything at all other than wrap the test function, and accept the test data as a set of arguments which can then be used by the wrapped test function.
- The unittest.TestCase class being decorated by the Framework can be entirely empty (as in the example), or it can include set Up and clear down methods as required by the test cases, or it could even include one or more *hand-written* test case methods (so long as the method names do not clash).

Further Information

- Full Documentation
- On PyPi (Python Package Index)
- Source code on GitHub

Troubleshooting & Bugs

Note: Every care is taken to try to ensure that this code comes to you bug free. If you do find an error - please report the problem on :

- GitHub Issues
- By email to : Tony Flury

License

This software is covered by the provisions of Apache Software License 2.0 License.

Why use the repeated Test Framework

The repeated Test Framework is designed to reduce the amount of repetitive code required to test some function/method with a lot of different data sets, while keep the benefits of using the unittest module :

Summary comparison

The following table shows a side-by-side comparison of using *Multiple Expilicit Test cases*, using a *Using a loop witin a single test method*, and using the *Repeated Test Framework*. It can be seen that the Framework retains the advantages of the explicit single test methods while also keeping the small code footprint that can normally only be acheieved by having a loop within a single test method.

Advantages

	Multiple Expilicit Test cases	*Using a loop witin a single test method*	*Repeated Test Framework*
Unique test methods	Yes	No	Yes
Unique doc strings with input data	Yes	No	Yes
Testing continues after failure	Yes	No	Yes
Decorate test cases (skip etc)	Yes	No	Yes
Test data in one place in source	No	Yes	Yes

Disadvantages

	Multiple Expilicit Test cases	*Using a loop witin a single test method*	*Repeated Test Framework*
Repititive code	Yes	No	No

The remainder of this page explores in detail the methodologies compared in the above table; giving an example of the same tests (and the same deliberate error) in each methodology, and describing the advantages and disadvantages which the table outlines.

Multiple Expilicit Test cases

The 'standard' way to use the unittest module is to write multiple test methods, with each method testing single input data point. As an illustration, *Example 1 - Multiple explicit Test cases* shows an trivial example of this methodology where a single item of functionality (in this case integer multiplication) is being tested against with multiple input values.

Example 1 - Multiple explicit Test cases

```
class TestCases(unittest.TestCase):

    def test_test1(self):
        """Confirm that 1*2 == 2"""
        self.assertEqual(1*2, 2)

    def test_test2(self):
        """Confirm that 2*2 == 2"""
        self.assertEqual(2*2, 4)

    def test_test3(self):
        """Confirm that 3*2 == 6"""
        self.assertEqual(3*2, 6)

    def test_test3(self):
        """Confirm that 3*4 == 11"""
         self.assertEqual(3*2, 11)
```

This testing methodology has a number of distinct and important advantages:

Unique Test cases Each test case can be executed from the command line (or from another script) as required - maybe to help diagnose a bug, or confirm a bug fix.

Unique documentation strings. Unique documentation strings means that testing output can include a descripton of the functionality being tested, and the input data being used (as well as the expected result). This can be very useful in documenting what has been tested and what input data is being used.

Test Separation Any test failure will not stop the execution of the remaining test cases.

However this methodogy has a distinct disadvantage in the case being discussed where the same functionality is being tested with mutiple different input data points: there is considerable repetition of very similar code, with the essential the difference between each test method being the data point being tested.

Using a loop witin a single test method

The most obvious way to remove the repititive code in *Example 1 - Multiple explicit Test cases* would be to refactor the tests into a single test method with a loop (after all a competent developer would never write 10 lines of code when that code could be written as a 4 line loop). *Example 2 - Single Test method with a loop* shows the same tests being executed using a single test method with a loop, and a list defining the test data.

Example 2 - Single Test method with a loop

```
class TestCases(unittest.TestCase):

    def test_testAll(self):
        """Confirm that all test cases work"""
        test_input = [(1,2,2),(2,2,4),(3,2,6), (3,4,11)]
        for in1, in2, result in test_input:
            self.assertEqual(in1*in2, result)
```

Example 2 clearly has far less code for any reasonable number of test cases, but despite the reduction of repition compared to *Example 1 - Multiple explicit Test cases*, but this also brings some distinct advantages when testing.

Non Unique testcases We only have one test method, so we can't use the command line to isolate and execute a single test case - (e.g. just test with an input of 3 & 4 - which fails in the above example). We also can't easily isolate and skip some input data (unless we edit the list).

Non Unique Documentation Strings With only one test case, and one documentation string to describe all of your test case, you will have limited logging as to what has been tested (depending on the verbosity level being used, the documentation strings will appear in your test output).

No Test Separation The loop system also has the disadvantage that any single failure will stop all further test execution in the list. The use of the subtest context manager can be used to ensure that testing continues after a failure in this example - it does not solve the other issues listed above.

Repeated Test Framework

The Repeated Test Framework provides a solution to all of these identified above by:

1. You write one generic method to execute the function/method which is under test.

2. You specify the actual test data as a list (in a similar to *Example 2 - Single Test method with a loop*).
3. Creating (behind the scenes) a unique test method for input data point
4. Allowing for customisation of both the names and documentation strings of those test methods.

Example 3 - Using the Repeated Test Framework

```
from repeatedtestframework import GenerateTestMethods

def test_method_wrapper(index, a, b, result):
    def test_method(self):
        """The actual test method which gets replicated"""
        self.assertEqual( a * b, result)
    return test_method

@GenerateTestMethods(
    test_name = 'test_multiplication',
    test_method = test_method_wrapper,
    test_cases = [
            {'a':1,'b':2, 'result':2 },
            {'a':2,'b':2, 'result':4 },
            {'a':3,'b':2, 'result':6 },
            {'a':3,'b':4, 'result':11 },]
        )
class TestCases(unittest.TestCase):
    pass
```

By default the test method names and documentation strings both contain the input data - allowing you to easily differentiate between the test methods both on the command line and in test result output. The test method names and documentation strings are completely customisable and can be edited to contain any data item which is part of your test input data.

As well as providing a simple method of generating many test cases, the Framework also provides methods for addong the normal unitest decorators to the generated methods, meaning that all of the unittest functionality is still available.

In the above example the Framework will create the following test methods :

		Test method arguments			
Test method name	Documentation string	*index*	*a*	*b*	*result*
test_000_test_multiplication	test_multiplication 000 {'a':1,'b':2,'result':2}	0	1	2	2
test_001_test_multiplication	test_multiplication 001 {'a':2,'b':2,'result':4}	1	2	2	4
test_002_test_multiplication	test_multiplication 002 {'a':3,'b':2,'result':6}	2	3	2	6
test_003_test_multiplication	test_multiplication 003 {'a':3,'b':4,'result':11}	3	3	4	11

From the above table it can be seen that by default the test method name includes an automatically generated index number, and the `test_name` attribute that is passed to the `GenerateTestMethods` decorator. The documentation string by default includes the `test_name` attribute, the generated index, as well as the data from the relevant item in the `test_cases` Iterator. The generated index, and each item test case data is passed to the `test_method` function as a set of keywords attributes, which can be used by the `test_method` function in anyway required.

For a guide on how to use the framework including how to customise test names, how to decorate individual test cases, and some useful usage suggestions see *Using the Repeated Test Framework*; For a full specificiation of the decorators - see *Module Interface*.

Using the Repeated Test Framework

The Framework is very easy to use; see the following sections for a full guide:

- *Simple Usage*
- *Customisation*
- *Decoratoring test methods*

A *Module Interface* is available.

Simple Usage

This first example illustrates using the Framework with default settings.

Listing 1.1: Example simple usage

```
1  from repeatedtestFramework import GenerateTestMethods
2
3  def test_method_wrapper(index, a, b, result):
4      """Wrapper for the test_method"""
5      def test_method(self):
6          """The actual test method which gets replicated"""
7          self.assertEqual( a * b, result)
8      return test_method
9
10 @GenerateTestMethods(
11     test_name = 'test_multiplication',
12     test_method = test_method_wrapper,
13     test_cases = [
14             {'a':1,'b':2, 'result':2 },
15             {'a':2,'b':2, 'result':4 },
16             {'a':3,'b':2, 'result':6 },
17             {'a':3,'b':4, 'result':11 },]
18         )
19 class TestCases(unittest.TestCase):
20     pass
```

A few things to note about this simple example :

1. The actual functionality under test is defined by the `test_method` function on lines 5-7. It is wrapped by the `test_method_wrapper` (lines 3-8); this wrap is neccessary as since the `test method` can only accept the self paramter, so the outer layer is required to give definition to the `input` and `expected_results` names which `test_method` needs.
2. The `test_cases` (lines 14 to 17) which defines the data for each separate test case is here is a list of dictionaries, but it could be any python Iterator which conatins a mapping per entry (for instance a generator which creates ordered dicts).
3. The TestCases class must be a sub class of unittest.TestCase, as per normal unittest module usage, but there are no other restrictions. The class could potentially contain other test methods (take care to ensure that method names don't clash), or the class could contain the normal test `setUp`, `setUpClass`, `tearDown` & `tearDownClass` methods to establish or destory the test Fixtures.

The example above is complete and will automatically generate test methods based on the input data - this is equivalent to the following code (including the deliberate error) :

```
class TestCases(unittest.TestCase):
    def test_000_test_multiplication(self):
        """test_multiplication 000 {'a':1,'b':2,'result':2}"""
        self.assertEqual(1*2, 2)

    def test_001_test_multiplication(self):
        """test_multiplication 001 {'a':2,'b':2,'result':4}"""
        self.assertEqual(2*2, 2)

    def test_002_test_multiplication(self):
        """test_multiplication 002 {'a':3,'b':2,'result':6}"""
        self.assertEqual(2*2, 6)

    def test_003_test_multiplication(self):
        """test_multiplication 003 {'a':3,'b':4,'result':11}"""
        self.assertEqual(3*4, 1)
```

See *Module Interface* for full details on the paramters and their usage

Return *to the top*

Customisation

The Framework has a number of options for customisation :

- *Method names & Documentation strings*:
- *Test Case Attributes*:

Method names & Documentation strings

Each test method is provided with a generated method name, and documentation string. The method names and documentation string are central to documenting your test suites and test results. Both the method name and documentation strings are generated in a predicatble fashion. The predictable method names means that individual test methods can be selected from the command line to be executed.

The format of the method name is controlled by the `method_name_template` attribute, and the format of this documentation string is controlled by using the `method_doc_template` string; both of these attributes are python format string (i.e. using the `format` method - see Format specification for full details).

The defaults for these attributes are :

- `method_name_template` : "test_{index:03d}_{test_name}"
- `method_doc_template` : "{test_name} {index:03d}: {test_data}"

Both the `method_name_template` and `method_doc_template` can contain the following keys :

- `test_name` : the value is the string passed into *test_name'* attrribute.
- `index` : the value is the start from zero index of the appropriate entry in the *test_case* iterator for this test case
- `test_data` : the value is the appropriate entry within the test_cases iterator for this test case.

Within the format strings the individual keys from the `test_data` dictionary can be accessed using the normal subscript notation (eg. :

```
>>> "_{test_data[a]:03d}_".format(test_data={'a':1, 'b':2})
"_001_
```

Warning: Unless you are using a custom dictionary and with an alternative `__str__` method, the `test_data` key **must not** be used within the `method_name_template` format string. During the formatting process, `test_data` value is converted to the string repesentation of a dictionary, and therefore by default it will contain characters which are not legal characters within a method name. It is possible to extract the individual data items within the `test_data` value using the format string subscript feature (illustrated above), but care still needs to be taken to ensure that any data extracted is valid for inclusion in a method name (e.g. only alphanumeric characters, or the underscore character _.

Test Case Attributes

As mentioned above the `test_cases` attribute is an iterator of mappings (in *Example simple usage* it is a list of dictionaries). The key/value pairs within those dictionaries are as a minimum the input and expected results, but they could be anything you would find useful, and the key's could be any string value which is a legal identifier (i.e. starts with a alphabetic character, and only contains alphabetic, numeric or underscores _ characters). Examples of extra uses for these key/value sets might be :

- To customise the error messages from the aseert calls within your test method; include in each dictionary an extra data item which is your customised message.
- To add a version of the test_data which can be used within your method name; include the usable form as an extra data item, and include a reference to that key within the `method_name_template`
- The ability to include arbitary key,value pairs within the data dictionary could be useful when using the *Decoratoring test methods*

See *Module Interface* for full details on the paramters and their usage

Return *to the top*

Decoratoring test methods

The unittest module includes a number of decorators that can be used to change the standard behaviour. These are :

- unittest.skip
- unittest.skipIf
- unittest.skipUnless
- unittest.expectedFailure

These decorators can still be used to decorate the entire TestCase class (either before or after the GenerateTestMethods is used). However since the test methods are automatically generated, it is not possible to use the unittest module decorators listed above.

The GenerateTestMethods has provided it's own equivalents which allow the selection of the test methods to be selected by using the test data itself :

- **skip** [skip the identified test method or methods] @*repeatedtestframework*.skip(reason, criteria = lambda test_data : True)

- **skipIf** [skip the identified test method or methods if the condition is True] @*repeatedtestframework*.skip(reason, condition, criteria = lambda test_data : True)
- **skipUnless** [skip the identified test method or methods if the condition is False] @*repeatedtestframework*.skip(reason, condition, criteria = lambda test_data : True)
- **expectedFailure** [mark the identified test method or methods as expecting to fail.] @*repeatedtestframework*.skip(criteria = lambda test_data : True)

The `skip` decorator is shown in the example below - all of the other decorators listed above work in the same way.

Listing 1.2: Decorator Example Usage

```
1  from repeatedtestframework import GenerateTestMethods
2  from repeatedtestframework import skip
3
4  def test_method_wrapper(index, a, b, result):
5      """Wrapper for the test_method"""
6      def test_method(self):
7          """The actual test method which gets replicated"""
8          self.assertEqual( a * b, result)
9      return test_method
10
11 @skip("This is a very boring test",
12     criteria = lambda test_data : test_data['a'] == 1)
13 @GenerateTestMethods(
14     test_name = 'test_multiplication',
15     test_method = test_method_wrapper,
16     test_cases = [
17             {'a':1,'b':2, 'result':2 },
18             {'a':2,'b':2, 'result':4 },
19             {'a':3,'b':2, 'result':6 },
20             {'a':3,'b':4, 'result':11 },]
21         )
22 class TestCases(unittest.TestCase):
23     pass
```

In the example above lines 11-13 demonstrate the use of the skip decorator from the framework - comparing it to the *unittest.skip* decorator the call above has the extra `criteria` paramter. The `criteria` parameter is a callable, which is invoked once for each generated test_method and is passed as a dictionary all of relevant test data for that test method, and also the test method index as an extra key. If the callable returns True for a particular test method (which it will do for the test method created for the first row of the `test_cases` parameter) then the unittest.skip decorator will be applied to that specific test method.

The following invocation of the skip decorator would be equivalent in the *Decorator Example Usage* above;

```
@skip("This is a very boring test",
    criteria = lambda test_data : test_data['index'] == 0)
```

The default for the `criteria` paramater for all 4 decorators is a simple callable that returns True in all cases. Therefore as a default the decorator applies to all the generated test methods.

As mentioned above *Customisation*, the test data can include arbitrary keys, which may not have any direct use in the test execution itself, but as shown above since the `criteria` callable is passed the full test data dictionary for each test method, a key could be included in the dictionary which is used to solely control the application of the decorator

Listing 1.3: Key use by Decorator

```
1  from repeatedtestframework import GenerateTestMethods
2  from repeatedtestframework import skip
3
4  def test_method_wrapper(index, a, b, result):
5      """Wrapper for the test_method"""
6      def test_method(self):
7          """The actual test method which gets replicated"""
8          self.assertEqual( a * b, result)
9      return test_method
10
11 @skip("This is a very boring test",
12     criteria = lambda test_data : test_data.get('skip', False))
13 @GenerateTestMethods(
14     test_name = 'test_multiplication',
15     test_method = test_method_wrapper,
16     test_cases = [
17             {'a':1,'b':2, 'result':2 },
18             {'skip':True, 'a':2,'b':2, 'result':4 },
19             {'a':3,'b':2, 'result':6 },
20             {'a':3,'b':4, 'result':11 },]
21         )
22 class TestCases(unittest.TestCase):
23     pass
```

In this example only the 2nd test case (a = 2, b = 2, result = 4) will have the unittest.skip decorator applied to as only that test case has a test case key of 'skip'. All of the other test cases will not have the decorator applied, as in those dictionaries, the 'skip' key is missing, and the criteria test uses a default of False in the case of a missing 'skip' key (line 12)

The framework also provides a method for applying any decorator method to the automatically generated test methods

```
repeatedtestframework.DecorateTestMethod(
                    criteria=lambda test_data: True, decorator_method=None,
                    decorator_args=None, decorator_kwargs=None)
```

The generic decorator has the following arguemnts

- `criteria` : as above a callable which is called for each test method, and is passed the test data dictionary appropriate to that method with the index added. The criteria should return True for all test_method to which the decorator should be applied, and False in all other cases.
- `decorator_method` : A callable which is the actual method with which the test method should be generated
- `decorator_args`: A tuple for the positional arguments for the decorator_method
- `decorator_kwargs`: A dictionary for the kwargs argument for the decorator_method

The example below shows using the DecorateTestMethod call as an alternative to the *skip* method as shown in *Decorator Example Usage*

Listing 1.4: Example of the DecorateTestMethod

```
1  import unittest
2
3  from repeatedtestframework import GenerateTestMethods
4  from repeatedtestframework import DecorateTestMethod
5
6  def test_method_wrapper(index, a, b, result):
```

```
    """Wrapper for the test_method"""
    def test_method(self):
        """The actual test method which gets replicated"""
        self.assertEqual( a * b, result)
    return test_method


@DecorateTestMethod( decorator_method = unittest.skip,
                     decorator_kwargs = {'reason': "This is a very boring test"},
                     criteria = lambda test_data : test_data.get('skip', False) )
@GenerateTestMethods(
    test_name = 'test_multiplication',
    test_method = test_method_wrapper,
    test_cases = [
             {'a':1,'b':2, 'result':2 },
             {'skip':True, 'a':2,'b':2, 'result':4 },
             {'a':3,'b':2, 'result':6 },
             {'a':3,'b':4, 'result':11 },]
        )
class TestCases(unittest.TestCase):
    pass
```

The two examples *Example of the DecorateTestMethod* and ref:*DecoratorExample* are functionally equivalent, but the former version (using the *skip* decorator is recommended for readability).

Note: Since the *DecorateTestMethod* can only access the test methods once they have been created, it must be invoked **after** the *GenerateTestMethods* decorator (i.e. it must appear before `GenerateTestMethods` in the decorator chain reading from the top).

Note: Using the decorators supplied by the framework will only apply the relevant unittest module decorator to the relevant test methods generated by the framework - any other test case which have been explicitly written in the unittest.TestCase class will be ignored by the decorators discussed above. Of course the usual unittest module decorators can be applied explicitly to those explicitly written test cases.

See *Module Interface* for full details on the paramters and their usage

Return *to the top*

Module Interface

Generate Test Methods Decorator

Skip Decorator

SkipIf Decorator

skipUnless Decorator

expectedFailure Decorator

DecorateTestMethod Decorator

Repeated Test Framework - License for use

Apache License
Version 2.0, January 2004
http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

   "Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf

of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

   You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.
8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.
9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

> To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "{}" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

Copyright Copyright (c) 2017 Tony Flury anthony.flury@btinternet.com

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

> http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

CHAPTER 2

Index

- genindex
- modindex