1 Example usage 3
2 API 9
3 Misc 11
   3.1 Docstring Handling 11
4 Indices and tables 13
Python Module Index 15
Index 17
DDT (Data-Driven Tests) allows you to multiply one test case by running it with different test data, and make it appear as multiple test cases.

You can find (and fork) the project on Github.

DDT should work on Python2 and Python3, but we only officially test it for versions 2.7 and 3.5-3.8.

Contents:
Example usage

DDT consists of a class decorator `ddt` (for your `TestCase` subclass) and two method decorators (for your tests that want to be multiplied):

- `data`: contains as many arguments as values you want to feed to the test.
- `file_data`: will load test data from a JSON or YAML file.

**Note:** Only files ending with `.yml` and `.yaml` are loaded as YAML files. All other files are loaded as JSON files.

Normally each value within `data` will be passed as a single argument to your test method. If these values are e.g. tuples, you will have to unpack them inside your test. Alternatively, you can use an additional decorator, `unpack`, that will automatically unpack tuples and lists into multiple arguments, and dictionaries into multiple keyword arguments. See examples below.

This allows you to write your tests as:

```python
import unittest

from ddt import ddt, data, file_data, unpack
from test.mycode import larger_than_two, has_three_elements, is_a_greeting

try:
    import yaml
except ImportError:  # pragma: no cover
    have_yaml_support = False
else:
    have_yaml_support = True

# A good-looking decorator
needs_yaml = unittest.skipUnless(have_yaml_support, "Need YAML to run this test")
```
class Mylist(list):
    pass

class MyClass:
    def __init__(self, **kwargs):
        for field, value in kwargs.items():
            setattr(self, field, value)
    
    def __eq__(self, other):
        return isinstance(other, dict) and vars(self) == other or 
        isinstance(other, MyClass) and vars(self) == vars(other)
    
    def __str__(self):
        return "TestObject %s" % vars(self)

def annotated(a, b):
    r = Mylist([a, b])
    setattr(r, "__name__", "test_$d_greater_than_$d" % (a, b))
    return r

def annotated2(listIn, name, docstring):
    r = Mylist(listIn)
    setattr(r, "__name__", name)
    setattr(r, "__doc__", docstring)
    return r

@ddt
class FooTestCase(unittest.TestCase):
    def test_undecorated(self):
        self.assertTrue(larger_than_two(24))

    @data
    def test_larger_than_two(self, value):
        self.assertTrue(larger_than_two(value))

    @data
    def test_not_larger_than_two(self, value):
        self.assertFalse(larger_than_two(value))

    @data
    def test_greater(self, value):
        a, b = value
        self.assertGreater(a, b)

    @data
    def test_greater_with_name_docstring(self, value):
        a, b = value
        self.assertIsNotNone(getattr(value, "__name__"))
        self.assertIsNotNone(getattr(value, "__doc__"))

@file_data('data/test_data_dict_dict.json')
def test_file_data_json_dict_dict(self, start, end, value):
    self.assertLess(start, end)
    self.assertLess(value, end)
    self.assertGreater(value, start)

@file_data('data/test_data_dict.json')
def test_file_data_json_dict(self, value):
    self.assertTrue(has_three_elements(value))

@file_data('data/test_data_list.json')
def test_file_data_json_list(self, value):
    self.assertTrue(is_a_greeting(value))

@needs_yaml
@file_data('data/test_data_dict_dict.yaml')
def test_file_data_yaml_dict_dict(self, start, end, value):
    self.assertLess(start, end)
    self.assertLess(value, end)
    self.assertGreater(value, start)

@needs_yaml
@file_data('data/test_data_dict.yaml')
def test_file_data_yaml_dict(self, value):
    self.assertTrue(has_three_elements(value))

@needs_yaml
@file_data('data/test_data_list.yaml')
def test_file_data_yaml_list(self, value):
    self.assertTrue(is_a_greeting(value))

@data(((3, 2), (4, 3), (5, 3))
@unpack
def test_tuples_extracted_into_arguments(self, first_value, second_value):
    self.assertTrue(first_value > second_value)

@data(([3, 2], [4, 3], [5, 3])
@unpack
def test_list_extracted_into_arguments(self, first_value, second_value):
    self.assertTrue(first_value > second_value)

@unpack
@data(("first": 1, "second": 3, "third": 2),
      "first": 4, "second": 6, "third": 5))
def test_dicts_extracted_into_kwargs(self, first, second, third):
    self.assertTrue(first < third < second)

@data(u'ascii', u'non-ascii-\N{SNOWMAN}')
def test_unicode(self, value):
    self.assertIn(value, (u'ascii', u'non-ascii-\N{SNOWMAN}'))

@data(3, 4, 12, 23)
def test_larger_than_two_with_doc(self, value):
    """Larger than two with value {0}""
    self.assertTrue(larger_than_two(value))

@data(3, 4, 12, 23)
def test_doc_missing_args(self, value):
"""Missing args with value {0} and {1}""
self.assertTrue(larger_than_two(value))

@data(3, 4, 12, 23)
def test_doc_missing_kargs(self, value):
    """Missing kargs with value {value} {value2}""
    self.assertTrue(larger_than_two(value))

@data(([3, 2], [4, 3], [5, 3])
@unpack
def test_list_extracted_with_doc(self, first_value, second_value):
    """Extract into args with first value {} and second value {}""
    self.assertTrue(first_value > second_value)

if have_yaml_support:
    # This test will only succeed if the execution context is from the ddt
directory. pyyaml cannot locate test.test_example.MyClass otherwise!

@ddt
class YamlOnlyTestCase(unittest.TestCase):
    @file_data('data/test_custom_yaml_loader.yaml', yaml.FullLoader)
def test_custom_yaml_loader(self, instance, expected):
        """Test with yaml tags to create specific classes to compare""
        self.assertEqual(expected, instance)

Where test_data_dict_dict.json:

{
    "positive_integer_range": {
        "start": 0,
        "end": 2,
        "value": 1
    },
    "negative_integer_range": {
        "start": -2,
        "end": 0,
        "value": -1
    },
    "positive_real_range": {
        "start": 0.0,
        "end": 1.0,
        "value": 0.5
    },
    "negative_real_range": {
        "start": -1.0,
        "end": 0.0,
        "value": -0.5
    }
}

and test_data_dict_dict.yaml:

positive_integer_range:
    start: 0
    end: 2
    value: 1

(continues on next page)
negative_integer_range:
    start: -2
    end: 0
    value: -1

positive_real_range:
    start: 0.0
    end: 1.0
    value: 0.5

negative_real_range:
    start: -1.0
    end: 0.0
    value: -0.5

and test_data_dict.json:

```json
{
    "unsorted_list": [10, 12, 15],
    "sorted_list": [15, 12, 50]
}
```

and test_data_dict.yaml:

```yaml
unsorted_list:
- 10
- 15
- 12

sorted_list: [15, 12, 50]
```

and test_data_list.json:

```json
[
    "Hello",
    "Goodbye"
]
```

and test_data_list.yaml:

```yaml
- "Hello"
- "Goodbye"
```

And then run them with your favourite test runner, e.g. if you use pytest:

```
$ pytest test/test_example.py
```

The number of test cases actually run and reported separately has been multiplied.
DDT will try to give the new test cases meaningful names by converting the data values to valid python identifiers.

**Note:** Python 2.7.3 introduced hash randomization which is by default enabled on Python 3.3 and later. DDT's default mechanism to generate meaningful test names will **not** use the test data value as part of the name for complex types if hash randomization is enabled.
You can disable hash randomization by setting the `PYTHONHASHSEED` environment variable to a fixed value before running tests (``export PYTHONHASHSEED=1`` for example).
class ddt.TestNameFormat
An enum to configure how mk_test_name() to compose a test name. Given the following example:

@data("a", "b")
def testSomething(self, value):
    ...

if using just @ddt or together with DEFAULT:
    • testSomething_1_a
    • testSomething_2_b
if using INDEX_ONLY:
    • testSomething_1
    • testSomething_2

ddt.add_test(cls, test_name, test_docstring, func, *args, **kwargs)
Add a test case to this class.
The test will be based on an existing function but will give it a new name.

ddt.data(*values)
Method decorator to add to your test methods.
Should be added to methods of instances of unittest.TestCase.

ddt.ddt(arg=None, **kwargs)
Class decorator for subclasses of unittest.TestCase.
Apply this decorator to the test case class, and then decorate test methods with @data.
For each method decorated with @data, this will effectively create as many methods as data items are passed as parameters to @data.
The names of the test methods follow the pattern original_test_name_{ordinal}_{data}.
ordinal is the position of the data argument, starting with 1.
For data we use a string representation of the data value converted into a valid python identifier. If data.
__name__ exists, we use that instead.

For each method decorated with @file_data('test_data.json'), the decorator will try to load the
test_data.json file located relative to the python file containing the method that is decorated. It will, for each
test_name key create as many methods in the list of values from the data key.

Decorating with the keyword argument testNameFormat can control the format of the generated test names.
For example:

- @ddt(testNameFormat=TestNameFormat.DEFAULT) will be index and values.
- @ddt(testNameFormat=TestNameFormat.INDEX_ONLY) will be index only.
- @ddt is the same as DEFAULT.

ddt. feed_data (func, new_name, test_data_docstring, *args, **kwargs)
This internal method decorator feeds the test data item to the test.

ddt. file_data (value, yaml_loader=None)
Method decorator to add to your test methods.

Should be added to methods of instances of unittest.TestCase.

value should be a path relative to the directory of the file containing the decorated unittest.TestCase.
The file should contain JSON encoded data, that can either be a list or a dict.

In case of a list, each value in the list will correspond to one test case, and the value will be concatenated to the
test method name.

In case of a dict, keys will be used as suffixes to the name of the test case, and values will be fed as test data.

yaml_loader can be used to customize yaml deserialization. The default is None, which results in using the
yaml.safe_load method.

ddt. i data (iterable)
Method decorator to add to your test methods.

Should be added to methods of instances of unittest.TestCase.

ddt. mk_test_name (name, value, index=0, name_fmt=<TestNameFormat.DEFAULT: 0>)
Generate a new name for a test case.

It will take the original test name and append an ordinal index and a string representation of the value, and
convert the result into a valid python identifier by replacing extraneous characters with _.

We avoid doing str(value) if dealing with non-trivial values. The problem is possible different names with
different runs, e.g. different order of dictionary keys (see PYTHONHASHSEED) or dealing with mock objects.
Trivial scalar values are passed as is.

A “trivial” value is a plain scalar, or a tuple or list consisting only of trivial values.

The test name format is controlled by enum TestNameFormat as well. See the enum documentation for
further details.

ddt. process_file_data (cls, name, func, file_attr)
Process the parameter in the file_data decorator.

ddt. unpack (func)
Method decorator to add unpack feature.
This page contains some of the miscellaneous functionality.

### 3.1 Docstring Handling

If one of the passed data objects has a docstring, the resulting testcase borrows it.

```python
dl = Dataobj()
dl.__doc__ = """This is a new docstring"""
d2 = Dataobj()

@data(dl, d2)
def test_something(self, value):
    """This is an old docstring"""
    return value
```

The first of the resulting test cases will have """"This is a new docstring"""" as its docstring and the second will keep its old one (""""This is an old docstring"""").
CHAPTER 4

Indices and tables

- genindex
- modindex
- search
Python Module Index

d
ddt, 9
A
add_test () (in module ddt), 9

D
data () (in module ddt), 9
ddt (module), 9
ddt () (in module ddt), 9

F
feed_data() (in module ddt), 10
file_data() (in module ddt), 10

I
idata() (in module ddt), 10

M
mk_test_name() (in module ddt), 10

P
process_file_data() (in module ddt), 10

T
TestNameFormat (class in ddt), 9

U
unpack() (in module ddt), 10