datatest Documentation

Release 0.8.3

Shawn Brown

Nov 26, 2017

Documentation

Version 0.8.3

Datatest provides validation tools for test driven data-wrangling. It extends Python's unittest package to provide testing tools for asserting data correctness.

Datatest can help prepare messy data that needs to be cleaned, integrated, formatted, and verified. It can provide structure for the tidying process, automate checklists, log discrepancies, and measure progress.

See the project's README file for supported versions, backward compatibility, and more.

CHAPTER 1

Tutorials

1.1 Introduction

For unittest-style validation, the *DataTestCase* extends the standard `unittest.TestCase` with methods for asserting validity and managing discrepancies.

The basic structure of a datatest suite mirrors that of a unittest suite:

```
import datatest


class TestExample(datatest.DataTestCase):
    def test_one(self):
        ...

    def test_two(self):
        ...


if __name__ == '__main__':
    datatest.main()
```

1.1.1 Validation

Data is validated by calling *assertValid(data, requirement)* to assert that *data* satisfies the given *requirement*. The requirement's **type** determines how the data is validated.

When *requirement* is a `set`, the elements of data are tested for membership in this set:

```
def test_membership_in_set(self):
    data = ['x', 'x', 'y', 'y', 'z', 'z']
    requirement = {'x', 'y', 'z'}  # <- set
    self.assertValid(data, requirement)
```

When *requirement* is a **function** (or other callable type), the elements are passed to the function one at a time. When the function returns true, an element is considered valid:

```
def test_function_returns_true(self):
    data = ['X', 'X', 'Y', 'Y']
    def requirement(x):  # <- callable (helper function)
        return x.isupper()
    self.assertValid(data, requirement)
```

When *requirement* is a `compiled` **regular expression**, elements are valid if they match the given pattern:

```
def test_regex_matches(self):
    data = ['foo', 'foo', 'foo', 'bar', 'bar', 'bar']
    requirement = re.compile('^\w\w\w$')  # <- regex object
    self.assertValid(data, requirement)
```

When *requirement* is a string, non-container, non-callable, or non-regex object, then elements are checked for equality:

```
def test_equality(self):
    data = ['x', 'x', 'x']
    requirement = 'x'  # <- other (not container, callable, or regex)
    self.assertValid(data, requirement)
```

When *requirement* is a **sequence** (list, tuple, etc.), elements are checked for equality and order:

```
def test_order(self):
    data = ['x', 'x', 'y', 'y', 'z', 'z']
    requirement = ['x', 'x', 'y', 'y', 'z', 'z']  # <- sequence
    self.assertValid(data, requirement)
```

When *requirement* is a `dict` (or other mapping), elements of matching keys are validated according to the requirement value's type:

```
def test_mapping(self):
    data = {'x': 'foo', 'y': 'bar'}
    requirement = {'x': 'foo', 'y': 'bar'}  # <- mapping
    self.assertValid(data, requirement)
```

You can run the above examples (`test_validation.py`) to see this behavior yourself.

Note: In the above examples, we used the variable names *data* and *requirement* to help explain the validation behavior. But in practice, it helps to use more descriptive names because these labels are used when reporting validation errors.

1.1.2 Error Messages

When validation fails, a *`ValidationError`* is raised. A ValidationError contains the differences detected in the *data* under test. To demonstrate this we will reuse tests from before, but this time the *data* will contain invalid elements which will trigger test case failures.

In the following test, we assert that *data* contains all of the elements in the required set:

```
def test_membership_in_set(self):
    data = ['x', 'x2', 'y', 'y', 'z', 'z']  # <- "x2" not in required set
    required_elements = {'x', 'y', 'z'}
    self.assertValid(data, required_elements)
```

But this assertion fails because `'x2'` does not appear in the requirement but it does appear in the data. The test fails with an *Extra* difference:

```
Traceback (most recent call last):
  File "docs/_static/test_errors.py", line 10, in test_membership_in_set
    self.assertValid(data, required_elements)
datatest.validation.ValidationError: does not satisfy set membership (1 difference): [
    Extra('x2'),
]
```

Here, we use a helper-function to assert that all of the elements are uppercase:

```
def test_function_returns_true(self):
    data = ['X', 'X', 'Y', 'y']
    def uppercase(x):
        return x.isupper()
    self.assertValid(data, uppercase)
```

Because `'y'` is lower-case, the test fails with an *Invalid* difference:

```
Traceback (most recent call last):
  File "docs/_static/test_errors.py", line 16, in test_function_returns_true
    self.assertValid(data, uppercase)
datatest.validation.ValidationError: does not satisfy 'uppercase' condition (1 diffe
rence): [
    Invalid('y'),
]
```

When comparing dictionaries, a dictionary of differences is raised if validation fails:

```
def test_mapping1(self):
    data = {
        'x': 'foo',
        'y': 'BAZ',  # <- required value is 'bar'
    }
    required_values = {
        'x': 'foo',
        'y': 'bar',
    }
    self.assertValid(data, required_values)
```

For the key `'y'`, the value under test is `'BAZ'` but the expected value is `'bar'`. The test fails with a dictionary of this *Invalid* difference:

```
Traceback (most recent call last):
  File "docs/_static/test_errors.py", line 42, in test_mapping1
    self.assertValid(data, required_values)
datatest.validation.ValidationError: does not satisfy mapping requirement (1 differe
nce): {
    'y': Invalid('BAZ', 'bar'),
}
```

When comparing numbers, numeric deviations are raised when differences are encountered:

```
def test_mapping2(self):
    data = {
        'x': 11,
        'y': 13,
```

```
    }
    required_values = {
        'x': 10,
        'y': 15,
    }
    self.assertValid(data, required_values)
```

A *Deviation* shows the numeric difference between the value under test and the expected value:

```
Traceback (most recent call last):
  File "docs/_static/test_errors.py", line 53, in test_mapping2
    self.assertValid(data, required_values)
datatest.validation.ValidationError: does not satisfy mapping requirement (2 differe
nces): {
    'x': Deviation(+1, 10),
    'y': Deviation(-2, 15),
}
```

You can run the above examples (`test_errors.py`) and change the values to see how differences are handled. When running these tests, you can use the `-f` command line flag to stop at the first error.

1.1.3 Allowances

It's not always possible to correct certain data errors. Sometimes, two equally authoritative sources report different results. Sometimes, a lack of information makes correction impossible. In any case, there are situations where it's appropriate to allow certain discrepancies for the purposes of data processing.

Similar Context Managers

Allowances are similar, in concept, to the Standard Library's `contextlib.suppress()` and `TestCase.assertRaises()` context managers. But rather than acting on an error itself, allowances operate on a collection of difference objects.

Datatest provides allowances in the form of context managers. Allowances operate on a ValidationError's collection of differences. Allowing **all** of an error's differences will suppress the error entirely. While allowing **some** of them will re-raise the error with the remaining differences.

Revisiting the set-membership failure above, we might decide that it's appropriate to allow *Extra* differences without triggering a test failure. To do this, we use the *allowedExtra()* context manager:

```
def test_membership_in_set(self):
    data = ['x', 'x2', 'y', 'y', 'z', 'z']  # <- "x2" is extra
    required_elements = {'x', 'y', 'z'}
    with self.allowedExtra():
        self.assertValid(data, required_elements)
```

Numeric deviations can be allowed with the *allowedDeviation()* or *allowedPercentDeviation()* context managers:

```
def test_mapping2(self):
    data = {
        'x': 11,  # <- +1
        'y': 13,  # <- -2
    }
```

```
    required_values = {
        'x': 10,
        'y': 15,
    }
    with self.allowedDeviation(2):  # allows +/- 2
        self.assertValid(data, required_values)
```

Sometimes, statistical outliers or mismatched data create a situation where a more-general allowance would be too broad, misleading, or otherwise unsuitable. In cases like this, we can allow individually specified differences with the *allowedSpecific()* context manager.

Below for the key of `'z'`, the value under test is `1000` but the required value is `20`. While we could allow this with `allowedDeviation(980)`, doing so is overly-vague given we are testing values that should range from 10 to 20. A more appropriate solution is to allow a single specified difference:

```
def test_mapping3(self):
    data = {
        'x': 10,
        'y': 15,
        'z': 1000,
    }
    required_values = {
        'x': 10,
        'y': 15,
        'z': 20,
    }

    known_outliers = {
        'z': Deviation(+980, 20),
    }

    with self.allowedSpecific(known_outliers):
        self.assertValid(data, required_values)
```

1.2 Querying Data

Datatest provides built-in classes for loading, querying, and iterating over the data under test. Although users familiar with other tools (Pandas, SQLAlchemy, etc.) should feel encouraged to use whatever they find to be most productive.

The following examples demonstrate datatest's *DataSource*, *DataQuery*, and *DataResult* classes. Users can follow along and type the commands themselves at Python's interactive prompt (>>>). For these examples, we will use the following data:

A	B	C
x	foo	20
x	foo	30
y	foo	10
y	bar	20
z	bar	10
z	bar	10

1.2.1 Loading Data

You can load the data from a CSV file (`example.csv`) with *DataSource.from_csv()*:

```
>>> import datatest
>>> source = datatest.DataSource.from_csv('example.csv')
```

1.2.2 Getting Field Names

You can get a list of field names with the *fieldnames* attribute:

```
>>> source.fieldnames
('A', 'B', 'C')
```

The fetch() Method

In the following examples, we call *fetch()* to eagerly evaluate the queries and display their results. In daily use, it's more efficient to leave off the "`.fetch()`" part and validate the *un-fetched* queries instead (which takes advantage of lazy evaluation).

1.2.3 Selecting Data

Calling our source like a function returns a *DataQuery* for the specified field or fields.

Select elements from column **A**:

```
>>> source('A').fetch()
['x', 'x', 'y', 'y', 'z', 'z']
```

Select elements from column **A** as a `set`:

```
>>> source({'A'}).fetch()
{'x', 'y', 'z'}
```

Select elements from column **A** as a `tuple`:

```
>>> source(('A',)).fetch()
('x', 'x', 'y', 'y', 'z', 'z')
```

The container type used in the selection determines the container type returned in the result. You can think of the selection as a template that describes the values and data types returned by the query. When the outer container type is not specified, it defaults to a `list`. In the first example we selected `'A'` which is used as shorthand for `['A']`:

```
>>> source(['A']).fetch()
['x', 'x', 'y', 'y', 'z', 'z']
```

Multiple Columns

Select elements from columns **A** and **B** as a list of tuples:

```
>>> source(('A', 'B')).fetch()  # Returns a list of tuples.
[('x', 'foo'),
 ('x', 'foo'),
 ('y', 'foo'),
 ('y', 'bar'),
```

```
 ('z', 'bar'),
 ('z', 'bar')]
```

Select elements from columns **A** and **B** as a set of tuples:

```
>>> source({('A', 'B')}).fetch()  # Returns a set of tuples.
{('x', 'foo'),
 ('y', 'foo'),
 ('y', 'bar'),
 ('z', 'bar')}
```

Compatible sequence and set types can be selected as inner and outer containers as needed.

In addition to lists, tuples, and sets, users can also select `frozensets`, `namedtuples`, etc. However, normal object limitations still apply—for example, sets can not contain mutable objects like lists or other sets.

Groups of Columns

Selecting groups of elements is accomplished using a `dict` or other mapping type. The key specifies how the elements are grouped and the value specifies the fields from which elements are selected.

For each unique value of column **A**, we select a list of elements from column **B**:

```
>>> source({'A': 'B'}).fetch()
{'x': ['foo', 'foo'],
 'y': ['foo', 'bar'],
 'z': ['bar', 'bar']}
```

As before, the types used in the selection determine the types returned in the result. For unique values of column **A**, we can select a `set` of elements from column **B** with the following:

```
>>> source({'A': {'B'}}).fetch()
{'x': {'foo'},
 'y': {'foo', 'bar'},
 'z': {'bar'}}
```

To group by multiple columns, we use a `tuple` of key fields. For each unique tuple of **A** and **B**, we select a list of elements from column **C**:

```
>>> source({('A', 'B'): 'C'}).fetch()
{('x', 'foo'): ['20', '30'],
 ('y', 'foo'): ['10'],
 ('y', 'bar'): ['20'],
 ('z', 'bar'): ['10', '10']}
```

Although selection types can be specified as needed, remember that dictionary keys must be immutable (`str`, `tuple`, `frozenset`, etc.).

1.2.4 Narrowing a Selection

Selections can be narrowed to rows that satisfy given keyword arguments.

Narrow a selection to rows where column **B** equals "foo":

```
>>> source(('A', 'B'), B='foo').fetch()
[('x', 'foo'), ('x', 'foo'), ('y', 'foo')]
```

The keyword column does not have to be in the selected result:

```
>>> source('A', B='foo').fetch()
['x', 'x', 'y']
```

Narrow a selection to rows where column **A** equals "x" *or* "y":

```
>>> source(('A', 'B'), A=['x', 'y']).fetch()
[('x', 'foo'),
 ('x', 'foo'),
 ('y', 'foo'),
 ('y', 'bar')]
```

Narrow a selection to rows where column **A** equals "y" *and* column **B** equals "bar":

```
>>> source([('A', 'B', 'C')], A='y', B='bar').fetch()
[('y', 'bar', '20')]
```

Only one row matches the above keyword conditions.

1.2.5 Additional Operations

DataQuery objects also support methods for operating on selected values.

Sum the elements from column **C**:

```
>>> source('C').sum().fetch()
100
```

Group by column **A** the sums of elements from column **C**:

```
>>> source({'A': 'C'}).sum().fetch()
{'x': 50, 'y': 30, 'z': 20}
```

Group by columns **A** and **B** the sums of elements from column **C**:

```
>>> source({('A', 'B'): 'C'}).sum().fetch()
{('x', 'foo'): 50,
 ('y', 'foo'): 10,
 ('y', 'bar'): 20,
 ('z', 'bar'): 20}
```

Select *distinct* elements:

```
>>> source('A').distinct().fetch()
['x', 'y', 'z']
```

Map elements with a function:

```
>>> def uppercase(value):
...     return str(value).upper()
...
>>> source('A').map(uppercase).fetch()
['X', 'X', 'Y', 'Y', 'Z', 'Z']
```

Filter elements with a function:

```
>>> def not_z(value):
...     return value != 'z'
...
>>> source('A').filter(not_z).fetch()
['x', 'x', 'y', 'y']
```

Since each method returns a new DataQuery, it's possible to chain together multiple method calls to transform the data as needed:

```
>>> def not_z(value):
...     return value != 'z'
...
>>> def uppercase(value):
...     return str(value).upper()
...
>>> source('A').filter(not_z).map(uppercase).fetch()
['X', 'X', 'Y', 'Y']
```

CHAPTER 2

How-to Guide

2.1 How to Install Datatest

The easiest way to install datatest is to use pip:

```
pip install datatest
```

To upgrade an existing installation, use the "`--upgrade`" option:

```
pip install --upgrade datatest
```

2.1.1 Stuntman Mike

If you need bug-fixes or features that are not available in the current stable release, you can "pip install" the development version directly from GitHub:

```
pip install --upgrade https://github.com/shawnbrown/datatest/archive/master.zip
```

All of the usual caveats for a development install should apply—only use this version if you can risk some instability or if you know exactly what you're doing. While care is taken to never break the build, it can happen.

2.1.2 Safety-first Clyde

If you need to review and test packages before installing, you can install datatest manually.

Download the latest **source** distribution from the Python Package Index (PyPI):

> https://pypi.python.org/pypi/datatest#downloads

Unpack the file (replacing X.Y.Z with the appropriate version number) and review the source code:

```
tar xvfz datatest-X.Y.Z.tar.gz
```

Change to the unpacked directory and run the tests:

```
cd datatest-X.Y.Z
python setup.py test
```

Don't worry if some of the tests are skipped. Tests for optional data sources (like pandas DataFrames or MS Excel files) are skipped when the related third-party packages are not installed.

If the source code and test results are satisfactory, install the package:

```
python setup.py install
```

2.2 How to Compare Two Files

To compare two data sources that have the same field names, we can create a single query and call it twice (once for each source). The pair of results can then be passed to *assertValid()*.

Below, we implement this with a helper-class ("ReferenceTestCase") that has a single "assertReference()" method:

```
import datatest


def setUpModule():
    global source_data, source_reference
    with datatest.working_directory(__file__):
        source_data = datatest.DataSource.from_csv('mydata.csv')
        source_reference = datatest.DataSource.from_csv('myreference.csv')


class ReferenceTestCase(datatest.DataTestCase):
    def assertReference(self, select, **where):
        """
        assertReference(select, **where)
        assertReference(query)

        Asserts that the query results from the data under test
        match the query results from the reference data.
        """
        if isinstance(select, datatest.DataQuery):
            query = select
        else:
            query = datatest.DataQuery(select, **where)
        data = query(source_data)
        requirement = query(source_reference)
        self.assertValid(data, requirement)

...
```

Test-cases that inherit from this class can use "assertReference()":

```
...

class TestMyData(ReferenceTestCase):
    def test_select_syntax(self):
        self.assertReference({('A', 'B')}, B='foo')

    def test_query_syntax(self):
```

```
        query = datatest.DataQuery({'A': 'C'}).sum()
        self.assertReference(query)


if __name__ == '__main__':
    datatest.main()
```

2.3 How to Assert Types

```
import datatest


class TypeTestCase(datatest.DataTestCase):
    def assertValidType(self, data, type, msg=None):
        """Assert that *data* elements are instances of *type*."""
        def check_type(x):
            return isinstance(x, type)
        msg = msg or 'must be instance of {0!r}'.format(type.__name__)
        self.assertValid(data, check_type, msg)

...
```

```
...

class TestTypes(TypeTestCase):
    def test_types(self):
        data = [-2, -1, 0, 1, 2]
        self.assertValidType(data, int)


if __name__ == '__main__':
    datatest.main()
```

2.4 How to Assert Subsets and Supersets

To assert subset or superset relations, use a `set` *requirement* together with the `allowedMissing()` or `allowedExtra()` context managers:

```
import datatest


class SetTestCase(datatest.DataTestCase):
    def assertSubset(self, data, requirement, msg=None):
        """Assert that set of *data* is a subset of *requirement*."""
        with self.allowedMissing():
            self.assertValid(data, set(requirement), msg)

    def assertSuperset(self, data, requirement, msg=None):
        """Assert that set of *data* is a superset of *requirement*."""
        with self.allowedExtra():
            self.assertValid(data, set(requirement), msg)
```

```
...
```

```
...

class TestSubset(SetTestCase):
    def test_subset(self):
        data = {'a', 'b'}
        requirement = {'a', 'b', 'c', 'd'}
        self.assertSubset(data, requirement)


if __name__ == '__main__':
    datatest.main()
```

2.5 How to Assert Fuzzy Matches

When comparing strings of text, it can sometimes be useful to assert that values are similar instead of asserting that they are exactly the same. To do this, we define an assertion that implements approximate string matching (also called fuzzy matching):

```
import difflib
import datatest


class FuzzyTestCase(datatest.DataTestCase):
    def assertFuzzy(self, data, requirement, cutoff=0.6, msg=None):
        """Assert that measures of string similarity are greater than
        or equal to *cutoff*.

        Similarity measures are determined using the ratio() method
        of the difflib.SequenceMatcher class. The values range from
        1.0 (exactly the same) to 0.0 (completely different).
        """
        class FuzzyMatcher(str):
            def __eq__(inner_self, other):
                if not isinstance(other, str):
                    return NotImplemented
                matcher = difflib.SequenceMatcher(a=inner_self, b=other)
                return matcher.ratio() >= cutoff  # <- closes over cutoff

        func = lambda x: FuzzyMatcher(x) if isinstance(x, str) else x
        requirement = datatest.DataQuery.from_object(requirement).map(func)

        msg = msg or 'string similarity of {0} or higher'.format(cutoff)
        self.assertValid(data, requirement, msg)
...
```

Tests that inherit from `FuzzyTestCase`, can use the `assertFuzzy()` method to check for approximate string matches:

```
...

class TestFuzzy(FuzzyTestCase):
    def test_assertion(self):
```

```
        scraped_data = {
            'MKT-GA4530': '4 1/2 inch Angle Grinder',
            'FLK-87-5': 'Fluke 87 5 Multimeter',
            'LEW-K2698-1': 'Lincoln Electric Easy MIG 180',
        }
        catalog_reference = {
            'MKT-GA4530': '4-1/2in Angle Grinder',
            'FLK-87-5': 'Fluke 87-5 Multimeter',
            'LEW-K2698-1': 'Lincoln Easy MIG 180',
        }
        self.assertFuzzy(scraped_data, catalog_reference, 0.75)


if __name__ == '__main__':
    datatest.main()
```

2.5.1 Allowing Fuzzy Differences

While it's usually more efficient to assert approximate matches and allow specific differences, it may be appropriate to allow approximate differences in certain cases. To do this, we can extend `FuzzyTestCase` from the previous example by defining an allowance:

```
class FuzzyTestCase(datatest.DataTestCase):
    ...

    def allowedFuzzy(self, cutoff=0.6, msg=None):
        """Allows Invalid differences whose measures of similarity are
        greater than or equal to *cutoff*.

        Similarity measures are determined using the ratio() method
        of the difflib.SequenceMatcher class. The values range from
        1.0 (exactly the same) to 0.0 (completely different).
        """
        def approx_diff(invalid, expected):
            matcher = difflib.SequenceMatcher(a=expected, b=invalid)
            return matcher.ratio() >= cutoff  # <- closes over cutoff
        msg = msg or 'string similarity of {0} or higher'.format(cutoff)
        return self.allowedInvalid() & self.allowedArgs(approx_diff, msg)
```

2.6 How to Assert an Interval

```
import datatest


class IntervalTestCase(datatest.DataTestCase):
    def assertInside(self, data, lower, upper, msg=None):
        """Assert that *data* elements fall inside given interval."""
        def interval(x):
            return lower <= x <= upper
        msg = msg or 'interval from {0!r} to {1!r}'.format(lower, upper)
        self.assertValid(data, interval, msg)

    def assertOutside(self, data, lower, upper, msg=None):
```

```
        """Assert that *data* elements fall outside given interval."""
        def not_interval(x):
            return not lower <= x <= upper
        msg = msg or 'interval from {0!r} to {1!r}'.format(lower, upper)
        self.assertValid(data, not_interval, msg)

...
```

```
...

class TestInterval(IntervalTestCase):
    def test_interval(self):
        data = [5, 7, 4, 5, 9]
        self.assertInside(data, lower=5, upper=10)


if __name__ == '__main__':
    datatest.main()
```

CHAPTER 3

Reference

3.1 Unittest Support

3.1.1 Basic Example

This short example demonstrates unittest-style testing of data in a CSV file (`mydata.csv`):

```
import datatest


def setUpModule():
    global source
    with datatest.working_directory(__file__):
        source = datatest.DataSource.from_csv('mydata.csv')


class TestMyData(datatest.DataTestCase):
    def test_header(self):
        fieldnames = source.fieldnames
        required_names = ['user_id', 'active']
        self.assertValid(fieldnames, required_names)

    def test_active_column(self):
        active = source({'active'})
        accepted_values = {'Y', 'N'}
        self.assertValid(active, accepted_values)

    def test_user_id_column(self):
        user_id = source(['user_id'])
        def positive_int(x):  # <- Helper function.
            return int(x) > 0
        self.assertValid(user_id, positive_int)
```

```
if __name__ == '__main__':
    datatest.main()
```

A data test-case is created by subclassing *datatest.DataTestCase* and individual tests are defined with methods whose names start with "test".

Inside each method, a call to *self.assertValid()* checks that the data satisfies a given requirement.

3.1.2 Command-Line Interface

The datatest module can be used from the command line just like unittest. To run the program with test discovery use the following command:

```
python -m datatest
```

Run tests from specific modules, classes, or individual methods with:

```
python -m datatest test_module1 test_module2
python -m datatest test_module.TestClass
python -m datatest test_module.TestClass.test_method
```

The syntax and command-line options (-f, -v, etc.) are the same as unittest—see unittest's command-line documentation for full details.

Note: Tests are ordered by **file name** and then by **line number** (within each file) when running datatest from the command-line.

3.1.3 DataTestCase

class datatest.**DataTestCase**(*methodName='runTest'*)

This class extends unittest.TestCase with methods for asserting data validity. In addition to the new functionality, familiar methods (like setUp, addCleanup, etc.) are still available.

assertValid(*data*, *requirement*, *msg=None*)

Fail if the *data* under test does not satisfy the *requirement*.

The given *data* can be a set, sequence, iterable, mapping, or other object. The *requirement* type determines how the data is validated (see below).

Set membership: When *requirement* is a set, elements in *data* are checked for membership in this set. On failure, a *ValidationError* is raised which contains *Missing* or *Extra* differences:

```
def test_mydata(self):
    data = ...
    requirement = {'A', 'B', 'C', ...}  # <- set
    self.assertValid(data, requirement)
```

Regular expression match: When *requirement* is a regular expression object, elements in *data* are checked to see if they match the given pattern. On failure, a *ValidationError* is raised with *Invalid* differences:

```
def test_mydata(self):
    data = ...
```

```
    requirement = re.compile(r'^[0-9A-F]*$')  # <- regex
    self.assertValid(data, requirement)
```

Sequence order: When *requirement* is a list or other sequence, elements in *data* are checked for matching order and value. On failure, an `AssertionError` is raised:

```
def test_mydata(self):
    data = ...
    requirement = ['A', 'B', 'C', ...]  # <- sequence
    self.assertValid(data, requirement)
```

Mapping comparison: When *requirement* is a dict (or other mapping), elements of matching keys are checked for equality. This comparison also requires *data* to be a mapping. On failure, a *ValidationError* is raised with *Invalid* or *Deviation* differences:

```
def test_mydata(self):
    data = ...  # <- Should also be a mapping.
    requirement = {'A': 1, 'B': 2, 'C': ...}  # <- mapping
    self.assertValid(data, requirement)
```

Function comparison: When *requirement* is a function or other callable, elements in *data* are checked to see if they evaluate to True. When the function returns False, a *ValidationError* is raised with *Invalid* differences:

```
def test_mydata(self):
    data = ...
    def requirement(x):  # <- callable (helper function)
        return x.isupper()
    self.assertValid(data, requirement)
```

Other comparison: When *requirement* does not match any previously specified type (e.g., str, float, etc.), elements in *data* are checked to see if they are equal to the given object. On failure, a *ValidationError* is raised which contains *Invalid* or *Deviation* differences:

```
def test_mydata(self):
    data = ...
    requirement = 'FOO'
    self.assertValid(data, requirement)
```

allowedMissing(*msg=None*)

Allows *Missing* elements without triggering a test failure:

```
with self.allowedMissing():
    data = {'A', 'B'}  # <- 'C' is missing
    requirement = {'A', 'B', 'C'}
    self.assertValid(data, requirement)
```

allowedExtra(*msg=None*)

Allows *Extra* elements without triggering a test failure:

```
with self.allowedExtra():
    data = {'A', 'B', 'C', 'D'}  # <- 'D' is extra
    requirement = {'A', 'B', 'C'}
    self.assertValid(data, requirement)
```

allowedInvalid(*msg=None*)

Allows *Invalid* elements without triggering a test failure:

```
with self.allowedInvalid():
    data = {'xxx': 'A', 'yyy': 'E'}  # <- 'E' is invalid
    requirement = {'xxx': 'A', 'yyy': 'B'}
    self.assertValid(data, requirement)
```

allowedDeviation(*tolerance*, */*, *msg=None*)
allowedDeviation(*lower*, *upper*, *msg=None*)

Allows numeric *Deviations* within a given *tolerance* without triggering a test failure:

```
with self.allowedDeviation(5):  # tolerance of +/- 5
    data = ...
    requirement = ...
    self.assertValid(data, requirement)
```

Specifying different *lower* and *upper* bounds:

```
with self.allowedDeviation(-2, 3):  # tolerance from -2 to +3
    data = ...
    requirement = ...
    self.assertValid(data, requirement)
```

Deviations within the given range are suppressed while those outside the range will trigger a test failure.

Empty values (None, empty string, etc.) are treated as zeros when performing comparisons.

allowedPercentDeviation(*tolerance*, */*, *msg=None*)
allowedPercentDeviation(*lower*, *upper*, *msg=None*)

Allows *Deviations* with percentages of error within a given *tolerance* without triggering a test failure:

```
with self.allowedPercentDeviation(0.03):  # tolerance of +/- 3%
    data = ...
    requirement = ...
    self.assertValid(data, requirement)
```

Specifying different *lower* and *upper* bounds:

```
with self.allowedPercentDeviation(-0.02, 0.01):  # tolerance from -2% to +1%
    data = ...
    requirement = ...
    self.assertValid(data, requirement)
```

Deviations within the given range are suppressed while those outside the range will trigger a test failure.

Empty values (None, empty string, etc.) are treated as zeros when performing comparisons.

allowedSpecific(*differences*, *msg=None*)

Allows individually specified *differences* without triggering a test failure:

```
two_diffs = self.allowedSpecific([
    Missing('C'),
    Extra('D'),
])
with two_diffs:
    data = {'A', 'B', 'D'}  # <- 'D' extra, 'C' missing
    requirement = {'A', 'B', 'C'}
    self.assertValid(data, requirement)
```

The *differences* argument can be a `list` or `dict` of differences or a single difference.

allowedKey (*function*, *msg=None*)
Allows differences in a mapping where *function* returns True. For each difference, *function* will receive the associated mapping **key** unpacked into one or more arguments.

allowedArgs (*function*, *msg=None*)
Allows differences where *function* returns True. For the 'args' attribute of each difference (a tuple), *function* must accept the number of arguments unpacked from 'args'.

allowedLimit (*number*, *msg=None*)
Allows a limited *number* of differences without triggering a test failure:

```
with self.allowedLimit(2):  # Allows up to two differences.
    data = ['47306', '1370', 'TX']  # <- '1370' and 'TX' invalid
    requirement = re.compile('^\d{5}$')
    self.assertValid(data, requirement)
```

If the count of differences exceeds the given *number*, the test will fail with a *ValidationError* containing all observed differences.

3.1.4 Test Runner Program

@datatest.**mandatory**
A decorator to mark whole test cases or individual methods as mandatory. If a mandatory test fails, DataTestRunner will stop immediately (this is similar to the `--failfast` command line argument behavior):

```
@datatest.mandatory
class TestFileFormat(datatest.DataTestCase):
    def test_columns(self):
        ...
```

@datatest.**skip**(*reason*)
A decorator to unconditionally skip a test:

```
@datatest.skip('Not finished collecting raw data.')
class TestSumTotals(datatest.DataTestCase):
    def test_totals(self):
        ...
```

@datatest.**skipIf**(*condition*, *reason*)
A decorator to skip a test if the condition is true.

@datatest.**skipUnless**(*condition*, *reason*)
A decorator to skip a test unless the condition is true.

class datatest.**DataTestRunner**(*stream=None*, *descriptions=True*, *verbosity=1*, *failfast=False*, *buffer=False*, *resultclass=None*, *ignore=False*)
A data test runner (wraps unittest.TextTestRunner) that displays results in textual form.

resultclass
alias of DataTestResult

run(*test*)
Run the given tests in order of line number from source file.

class datatest.**DataTestProgram**(*module='__main__'*, *defaultTest=None*, *argv=None*, *testRunner=datatest.DataTestRunner*, *testLoader=unittest.TestLoader*, *exit=True*, *verbosity=1*, *failfast=None*, *catchbreak=None*, *buffer=None*, *warnings=None*)

datatest.**main**
alias of *DataTestProgram*

3.2 Data Handling

3.2.1 working_directory

class datatest.**working_directory**(*path*)
A context manager to temporarily set the working directory to a given *path*. If *path* specifies a file, the file's directory is used. When exiting the with-block, the working directory is automatically changed back to its previous location.

Use the global __file__ variable to load data relative to test file's current directory:

```
with datatest.working_directory(__file__):
    source = datatest.DataSource.from_csv('myfile.csv')
```

This context manager can also be used as a decorator.

3.2.2 DataSource

class datatest.**DataSource**(*data*, *fieldnames=None*)
A class to quickly load and query tabular data.

The given *data* should be an iterable of rows. The rows themselves can be lists (as below), dictionaries, or other sequences or mappings. *fieldnames* must be a sequence of strings to use when referencing data by field:

```
data = [
    ['x', 100],
    ['y', 200],
    ['z', 300],
]
fieldnames = ['A', 'B']
source = datatest.DataSource(data, fieldnames)
```

If *data* is an iterable of `dict` or `namedtuple` rows, then *fieldnames* can be omitted:

```
data = [
    {'A': 'x', 'B': 100},
    {'A': 'y', 'B': 200},
    {'A': 'z', 'B': 300},
]
source = datatest.DataSource(data)
```

classmethod **from_csv**(*file*, *encoding=None*, ***fmtparams*)
Create a DataSource from a CSV *file* (a path or file-like object):

```
source = datatest.DataSource.from_csv('mydata.csv')
```

If *file* is an iterable of files, data will be loaded and aligned by column name:

```
files = ['mydata1.csv', 'mydata2.csv']
source = datatest.DataSource.from_csv(files)
```

classmethod **from_excel** (*path*, *worksheet=0*)

Create a DataSource from an Excel worksheet. The *path* must specify to an XLSX or XLS file and the *worksheet* must specify the index or name of the worksheet to load (defaults to the first worksheet). This constructor requires the optional, third-party library xlrd.

Load first worksheet:

```
source = datatest.DataSource.from_excel('mydata.xlsx')
```

Specific worksheets can be loaded by name (a string) or index (an integer):

```
source = datatest.DataSource.from_excel('mydata.xlsx', 'Sheet 2')
```

fieldnames

A tuple of field names used by the data source.

__call__ (*select*, ***where*)

Calling a DataSource like a function returns a DataQuery object that is automatically associated with the source (see *DataQuery* for *select* and *where* syntax):

```
query = source('A')
```

This is a shorthand for:

```
query = DataQuery.from_object(source, 'A')
```

3.2.3 DataQuery

class datatest.**DataQuery** (*select*, ***where*)

A class to query data from a source object. Queries can be created, modified, and passed around without actually computing the result—computation doesn't occur until the query object itself or its *fetch()* method is called.

The *select* argument must be a container of one field name (a string) or of an inner-container of multiple filed names. The optional *where* keywords can narrow a selection to rows where fields match specified values.

Although DataQuery objects are usually created by *calling* an existing DataSource object like a function, it's possible to create them independent of any single data source:

```
query = DataQuery('A')
```

classmethod **from_object** (*source*, *select*, ***where*)
classmethod **from_object** (*object*)

Creates a query and associates it with the given object.

If the object is a DataSource, you must provide a *select* argument and may also narrow the selection with keyword arguments:

```
source = DataSource(...)
query = DataQuery.from_object(source, 'A')
```

A non-DataSource container may also be used:

```
list_object = [1, 2, 3, 4]
query = DataQuery.from_object(list_object)
```

sum()
Get the sum of non-None elements.

count()
Get the count of non-None elements.

avg()
Get the average of non-None elements. Strings and other objects that do not look like numbers are interpreted as 0.

min()
Get the minimum value from elements.

max()
Get the maximum value from elements.

distinct()
Filter elements, removing duplicate values.

apply(*function*)
Apply *function* to entire group keeping the resulting data. If element is not iterable, it will be wrapped as a single-item list.

map(*function*)
Apply *function* to each element, keeping the results. If the group of data is a set type, it will be converted to a list (as the results may not be distinct or hashable).

filter(*function=None*)
Filter elements, keeping only those values for which *function* returns True. If *function* is None, this method keeps all elements for which `bool` returns True.

reduce(*function*)
Reduce elements to a single value by applying a *function* of two arguments cumulatively to all elements from left to right.

fetch()
Executes query and returns an eagerly evaluated result.

__call__(*source=None*, *optimize=True*)
A DataQuery can be called like a function to execute it and return a value or *DataResult* appropriate for lazy evaluation:

```
query = source('A')
result = query()  # <- Returns DataResult (iterator)
```

Setting *optimize* to False turns-off query optimization.

3.2.4 DataResult

class datatest.**DataResult**(*iterable*, *evaluation_type*)
A simple iterator that wraps the results of *DataQuery* execution. This iterator is used to facilitate the lazy evaluation of data objects (where possible) when asserting data validity.

Although DataResult objects are usually constructed automatically, it's possible to create them directly:

```
iterable = iter([...])
result = DataResult(iterable, evaluation_type=list)
```

> **Warning:** When iterated over, the *iterable* **must** yield only those values necessary for constructing an object of the given *evaluation_type* and no more. For example, when the *evaluation_type* is a set, the *iterable* must not contain duplicate or unhashable values. When the *evaluation_type* is a `dict` or other mapping, the *iterable* must contain unique key-value pairs or a mapping.

evaluation_type
The type of instance returned by the *fetch* method.

fetch()
Evaluate the entire iterator and return its result:

```
result = DataResult(iter([...]), evaluation_type=set)
result_set = result.fetch()  # <- Returns a set of values.
```

When evaluating a `dict` or other mapping type, any values that are, themselves, *DataResult* objects will also be evaluated.

__wrapped__
The underlying iterator—useful when introspecting or rewrapping.

3.3 Error and Difference

3.3.1 ValidationError

exception datatest.**ValidationError**(*message*, *differences*)
Raised when a data validation fails.

message
The message given to the exception constructor.

differences
The differences given to the exception constructor.

args
The tuple of arguments given to the exception constructor.

3.3.2 Differences

class datatest.**Missing**(**args*)
A value **not found in data** that is in *requirement*.

class datatest.**Extra**(**args*)
A value found in *data* but is **not in requirement**.

class datatest.**Invalid**(*invalid*, *expected=None*)
A value in *data* that does not satisfy a function, equality, or regular expression *requirement*.

class datatest.**Deviation**(*deviation*, *expected*)
The difference between a numeric value in *data* and a matching numeric value in *requirement*.

deviation

percent_deviation

expected

For a full list of classes and objects, see the Package Index.

CHAPTER 4

Test Driven Data-Wrangling

In the practice of data science, data preparation is a huge part of the job. Practitioners often spend 50 to 80 percent of their time wrangling data[1][2][3][4]. This critically important phase is time-consuming, unglamorous, and often poorly structured.

The `datatest` package was created to support test driven data-wrangling and provide a disciplined approach to an otherwise messy process.

A datatest suite can facilitate quick edit-test cycles to help guide the selection, cleaning, integration, and formatting of data. Data tests can also help to automate check-lists, measure progress, and promote best practices.

4.1 Structuring a Test Suite

> *"Unix was not designed to stop you from doing stupid things, because that would also stop you from doing clever things."* —Doug Gwyn[5]

The structure of a datatest suite defines a data preparation workflow. The first tests should address essential prerequisites and the following tests should focus on specific requirements.

Typically, data tests should be defined in the following order:

1. load data sources (asserts that expected source data is present)
2. check for expected column names

[1] "Data scientists, according to interviews and expert estimates, spend from 50 percent to 80 percent of their time mired in this more mundane labor of collecting and preparing unruly digital data..." Steve Lohraug in *For Big-Data Scientists, 'Janitor Work' Is Key Hurdle to Insights*. Retrieved from http://www.nytimes.com/2014/08/18/technology/for-big-data-scientists-hurdle-to-insights-is-janitor-work.html

[2] "This [data preparation step] has historically taken the largest part of the overall time in the data mining solution process, which in some cases can approach 80% of the time." *Dynamic Warehousing: Data Mining Made Easy* (p. 19)

[3] Online poll of data mining practitioners: See image, *Data preparation (Oct 2003)*. Retrieved from http://www.kdnuggets.com/polls/2003/data_preparation.htm [While this poll is quite old, the situation has not changed drastically.]

[4] "As much as 80% of KDD is about preparing data, and the remaining 20% is about mining." *Data Mining for Design and Manufacturing* (p. 44)

[5] Doug Gwyn, Computer scientist for the U.S. Army Research Laboratory, as quoted in Michael Fitzgerald's *Introducing Regular Expressions* (p. 103)

3. validate format of values (data type or other regex)
4. assert set-membership requirements
5. assert sums, counts, or cross-column values

Note: Datatest's built-in test runner executes tests ordered by file name and then by line number within each file. You can control the order that tests are run by arranging the order they appear in the test file itself.

4.2 Data Preparation Workflow

Using a quick edit-test cycle, users can:

1. focus on a failing test
2. make small changes to the data
3. re-run the suite to check that the test now passes
4. then, move on to the next failing test

The work of cleaning and formatting data takes place outside of the datatest package itself. Users can work with with the tools they find the most productive (Excel, pandas, R, sed, etc.).

Python Module Index

d

Index

Symbols

A

C

D

E

F

I

M

P

R

S

V

W