datatest Documentation

Release 0.8.2

Shawn Brown

Jun 11, 2017

Contents

1 `datatest` — Test driven data wrangling **3**

2 Querying Data **5**
- 2.1 Loading Data . . . 5
- 2.2 Getting Field Names . . . 5
- 2.3 Selecting Data . . . 6
- 2.4 Narrowing a Selection . . . 7
- 2.5 Additional Operations . . . 7

3 Unittest-Style **9**
- 3.1 Basic Example . . . 9
- 3.2 Command-Line Interface . . . 10
- 3.3 DataTestCase . . . 10
- 3.4 Test Runner Program . . . 13

4 Data Handling **15**
- 4.1 working_directory . . . 15
- 4.2 DataSource . . . 15
- 4.3 DataQuery . . . 16
- 4.4 DataResult . . . 18

5 Error and Difference **19**
- 5.1 ValidationError . . . 19
- 5.2 Differences . . . 19

Python Module Index **21**

Version 0.8.2 (Installation Instructions)

CHAPTER 1

`datatest` — Test driven data wrangling

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.

CHAPTER 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:

one	two	three
a	x	100
a	x	100
b	x	100
b	y	100
c	y	100
c	y	100

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')
```

Getting Field Names

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

```
>>> source.fieldnames
['one', 'two', 'three']
```

The execute() Method

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

Selecting Data

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

Select elements from column **one** as a `list`:

```
>>> source(['one']).execute()
['a', 'a', 'b', 'b', 'c', 'c']
```

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

```
>>> source({'one'}).execute()
{'a', 'b', 'c'}
```

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. Because set objects can not contain duplicates, the second example above has only one element for each unique value in the column.

Multiple Columns

Select elements from columns **one** and **two** as a list of `tuple` values:

```
>>> source([('one', 'two')]).execute()  # Returns a list of tuples.
[('a', 'x'),
 ('a', 'x'),
 ('b', 'x'),
 ('b', 'y'),
 ('c', 'y'),
 ('c', 'y')]
```

Select elements from columns **one** and **two** as a set of tuple values:

```
>>> source({('one', 'two')}).execute()  # Returns a set of tuples.
{('a', 'x'),
 ('b', 'x'),
 ('b', 'y'),
 ('c', 'y')}
```

Compatible sequence and set types can be selected as inner and outer containers as needed. A selection's outer container must always hold a single element (a string or inner container).

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

Select groups of elements from column **one** that contain lists of elements from column **two** as a `dict`:

```
>>> source({'one': ['two']}).execute()  # Grouped by key.
{'a': ['x', 'x'],
 'b': ['x', 'y'],
 'c': ['y', 'y']}
```

Select groups of elements from columns **one** and **two** (using a `tuple`) that contain lists of elements from column **three**:

```
>>> source({('one', 'two'): ['three']}).execute()
{('a', 'x'): ['100', '100'],
 ('b', 'x'): ['100'],
 ('b', 'y'): ['100'],
 ('c', 'y'): ['100', '100']}
```

When selecting groups of elements, you must provide a dictionary with a single key-value pair. As before, the selection types determine the result types, but keep in mind that dictionary keys must be immutable (`str`, `tuple`, `frozenset`, etc.).

Narrowing a Selection

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

Narrow a selection to rows where column **two** equals "x":

```
>>> source([('one', 'two')], two='x').execute()
[('a', 'x'),
 ('a', 'x'),
 ('b', 'x')]
```

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

```
>>> source(['one'], two='x').execute()
['a',
 'a',
 'b']
```

Narrow a selection to rows where column **one** equals "a" *or* "b":

```
>>> source([('one', 'two')], one=['a', 'b']).execute()
[('a', 'x'),
 ('a', 'x'),
 ('b', 'x'),
 ('b', 'y')]
```

Narrow a selection to rows where column **one** equals "b" *and* column **two** equals "y":

```
>>> source([('one', 'two')], one='b', two='y').execute()
[('b', 'y')]  # Only 1 row matches these keyword conditions.
```

Additional Operations

Sum the values from column **three**:

```
>>> source(['three']).sum().execute()
600
```

Group by column **one** and sum the values from column **three** (for each group):

```
>>> source({'one': ['three']}).sum().execute()
{'a': 200,
 'b': 200,
 'c': 200}
```

Group by columns **one** and **two** and sum the values from column **three**:

```
>>> source({('one', 'two'): ['three']}).sum().execute()
{('a', 'x'): 200,
 ('b', 'x'): 100,
 ('b', 'y'): 100,
 ('c', 'y'): 200}
```

Select *distinct* values:

```
>>> source(['one']).distinct().execute()
['a', 'b', 'c']
```

Map values with a function:

```
>>> def uppercase(x):
...     return str(x).upper()
...
>>> source(['one']).map(uppercase).execute()
['A', 'A', 'B', 'B', 'C', 'C']
```

Filter values with a function:

```
>>> def not_c(x):
...     return x != 'c'
...
>>> source(['one']).filter(not_c).execute()
['a', 'a', 'b', 'b']
```

Multiple methods can be chained together:

```
>>> def not_c(x):
...     return x != 'c'
...
>>> def uppercase(x):
...     return str(x).upper()
...
>>> source(['one']).filter(not_c).map(uppercase).execute()
'AABB'
```

CHAPTER 3

Unittest-Style

Basic Example

```
import datatest

class TestMyData(datatest.DataTestCase):
    @classmethod
    def setUpClass(cls):
        data = [
            {'is_active': 'Y', 'member_id': 105},
            {'is_active': 'Y', 'member_id': 104},
            {'is_active': 'Y', 'member_id': 103},
            {'is_active': 'N', 'member_id': 102},
            {'is_active': 'N', 'member_id': 101},
            {'is_active': 'N', 'member_id': 100},
        ]
        cls.source = datatest.DataSource(data)

    def test_is_active(self):
        allowed_values = {'Y', 'N'}
        self.assertValid(self.source(['is_active']), allowed_values)

    def test_member_id(self):
        def positive_integer(x):  # <- Helper function.
            return isinstance(x, int) and x > 0
        self.assertValid(self.source(['member_id']), positive_integer)

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

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 the unittest documentation for full details.

Note: By default, tests are ordered by **module name** and **line number** (within each module).

Unlike strict unit testing, data preparation tests are often dependant on one another—this strict order-by-line-number behavior lets users design test suites appropriately. For example, asserting the population of a city will always fail when the 'city' column is missing. So it's appropriate to validate column names *before* validating the contents of each column.

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 *:class:'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 specified *differences* without triggering a test failure:

```
diffs = [
    Missing('C'),
    Extra('D'),
]
with self.allowedSpecific(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.

Note: In the deviation methods above, *tolerance* is a positional-only parameter—it cannot be specified using keyword syntax.

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*

CHAPTER 4

Data Handling

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.

DataSource

class datatest.**DataSource**(*data*, *fieldnames=None*)

A basic data source to quickly load and query 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 list 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(source, ['A'])
```

DataQuery

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

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

A class to query data from a *DataSource* object. Queries can be created, modified and passed around without actually computing the result—computation doesn't occur until the *execute()* 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. A *defaultsource* can be provided to associate the query with a specific DataSource object.

Queries are usually created from an existing source (the originating source is automatically associated with the new query):

```
source = DataSource(...)
query = source(['A'])  # <- DataQuery created from source.
```

Queries can be created directly as well:

```
source = DataSource(...)
query = DataQuery(source, ['A'])  # <- Direct initialization.
```

Queries can also be created independent of any single data source:

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

defaultsource
A property for setting a predetermined *DataSource* to use when *execute()* is called without a *source* argument.

When a query is created from a DataSource call, this property is assigned automatically. When a query is created directly, the value can be passed explicitly or it can be omitted.

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.

map(*function*)
Apply *function* to each element keeping the resulting data.

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.

execute(*source=None*, *, *evaluate=True*, *optimize=True*)
Execute the query and return its result. The *source* should be a *DataSource* on which the query will operate. If *source* is omitted, the *defaultsource* is used.

By default, results are eagerly evaluated (and loaded into memory). For lazy evaluation, set *evaluate* to False to return a *DataResult* iterator instead.

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

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

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

This is a shorthand for calling the *execute()* method with *evaluate* set to False.

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)
```

When iterated over, the *iterable* must yield only those values necessary for constructing an object of the given *evaluation_type* and no more. When the *evaluation_type* is a set, the *iterable* must not contain duplicate values. When the *evaluation_type* is a dict or other mapping, the *iterable* must contain suitable key-value pairs or a mapping.

evaluation_type

The type of instance returned by the *evaluate* method.

evaluate()

Evaluate the entire iterator and return its result:

```
result = DataResult(iter([...]), evaluation_type=set)
result_set = result.evaluate()  # <- 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.

CHAPTER 5

Error and Difference

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.

Differences

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

class datatest.**Extra**(**args*)
A value found in *data* that 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

- Package Index

Python Module Index

d

datatest, 3

Index

Symbols

__call__() (datatest.DataQuery method), 17
__call__() (datatest.DataSource method), 16
__wrapped__ (datatest.DataResult attribute), 18

A

allowedArgs() (datatest.DataTestCase method), 12
allowedDeviation() (datatest.DataTestCase method), 11
allowedExtra() (datatest.DataTestCase method), 11
allowedInvalid() (datatest.DataTestCase method), 11
allowedKey() (datatest.DataTestCase method), 12
allowedLimit() (datatest.DataTestCase method), 13
allowedMissing() (datatest.DataTestCase method), 11
allowedPercentDeviation() (datatest.DataTestCase method), 12
allowedSpecific() (datatest.DataTestCase method), 12
args (datatest.ValidationError attribute), 19
assertValid() (datatest.DataTestCase method), 10
avg() (datatest.DataQuery method), 17

C

count() (datatest.DataQuery method), 17

D

DataQuery (class in datatest), 16
DataResult (class in datatest), 18
DataSource (class in datatest), 15
datatest (module), 1, 3, 8, 14, 18
DataTestCase (class in datatest), 10
DataTestProgram (class in datatest), 13
DataTestRunner (class in datatest), 13
defaultsource (datatest.DataQuery attribute), 17
Deviation (class in datatest), 19
deviation (datatest.Deviation attribute), 19
differences (datatest.ValidationError attribute), 19
distinct() (datatest.DataQuery method), 17

E

evaluate() (datatest.DataResult method), 18
evaluation_type (datatest.DataResult attribute), 18
execute() (datatest.DataQuery method), 17
expected (datatest.Deviation attribute), 19
Extra (class in datatest), 19

F

fieldnames (datatest.DataSource attribute), 16
filter() (datatest.DataQuery method), 17
from_csv() (datatest.DataSource class method), 16
from_excel() (datatest.DataSource class method), 16

I

Invalid (class in datatest), 19

M

main (in module datatest), 14
mandatory() (in module datatest), 13
map() (datatest.DataQuery method), 17
max() (datatest.DataQuery method), 17
message (datatest.ValidationError attribute), 19
min() (datatest.DataQuery method), 17
Missing (class in datatest), 19

P

percent_deviation (datatest.Deviation attribute), 19

R

reduce() (datatest.DataQuery method), 17
resultclass (datatest.DataTestRunner attribute), 13
run() (datatest.DataTestRunner method), 13

S

skip() (in module datatest), 13
skipIf() (in module datatest), 13
skipUnless() (in module datatest), 13
sum() (datatest.DataQuery method), 17

V

ValidationError, 19

W

working_directory (class in datatest), 15