datatest Documentation

Release 0.11.1

Shawn Brown

Apr 27, 2021

DOCUMENTATION

Datatest helps to speed up and formalize data-wrangling and data validation tasks. It was designed to work with poorly formatted data by detecting and describing validation failures.

- *Validate* the format, type, set membership, and more from a variety of data sources including pandas `DataFrames` and `Series`, NumPy `ndarrays`, built-in data structures, etc.
- Smart *comparison behavior* applies the appropriate validation method for a given data requirement.
- Automatic *data handling* manages the validation of single elements, sequences, sets, dictionaries, and other containers of elements.
- *Difference objects* characterize the discrepancies and deviations between a dataset and its requirements.
- *Acceptance managers* distinguish between ideal criteria and acceptable differences.

Test driven data-wrangling is a process for taking data from a source of unverified quality or format and producing a verified, well-formatted dataset. It repurposes software testing practices for data preparation and quality assurance projects. **Pipeline validation** monitors the status and quality of data as it passes through a pipeline and identifies *where* in a pipeline an error occurs.

See the project README file for full details regarding supported versions, backward compatibility, and more.

CHAPTER

ONE

TABLE OF CONTENTS

1.1 Introduction

"... tidy datasets are all alike but every messy dataset is messy in its own way" —Hadley Wickham[1]

1.1.1 A Tour of Datatest

This document introduces *datatest*'s support for validation, error reporting, and acceptance declarations.

Validation

The validation process works by comparing some *data* to a given *requirement*. If the requirement is satisfied, the data is considered valid. But if the requirement is not satisfied, a `ValidationError` is raised.

The `validate()` function checks that the *data* under test satisfies a given *requirement*:

```
from datatest import validate

data = ...
requirement = ...
validate(data, requirement)
```

Smart Comparisons

The `validate()` function implements smart comparisons and will use different validation methods for different *requirement* types.

For example, when *requirement* is a `set`, validation checks that elements in *data* are members of that set:

```
from datatest import validate

data = ['A', 'B', 'A']
requirement = {'A', 'B'}
validate(data, requirement)
```

When *requirement* is a **function**, validation checks that the function returns True when applied to each element in *data*:

[1] Wickham, Hadley. "Tidy Data." Journal of Statistical Software 59, no. 10, August 2014.

```
from datatest import validate

data = [2, 4, 6, 8]

def is_even(x):
    return x % 2 == 0

validate(data, requirement=is_even)
```

When *requirement* is a **type**, validation checks that the elements in *data* are a instances of that type:

```
from datatest import validate

data = [2, 4, 6, 8]
requirement = int
validate(data, requirement)
```

And when *requirement* is a `tuple`, validation checks for tuple elements in *data* using multiple methods at the same time—one method for each item in the required tuple:

```
from datatest import validate

data = [('a', 2), ('b', 4), ('c', 6)]

def is_even(x):
    return x % 2 == 0

requirement = (str, is_even)
validate(data, requirement)
```

In addition to the examples above, several other validation behaviors are available. For a complete listing with detailed examples, see *Validation*.

Automatic Data Handling

Along with the smart comparison behavior, validation can apply a given *requirement* to *data* objects of different formats.

The following examples perform type-checking to see if elements are `int` values. Switch between the different tabs below and notice that the same *requirement* (`requirement = int`) works for all of the different *data* formats:

Element

Group

Mapping

Mapping of Groups

An individual element:

```
from datatest import validate

data = 42
requirement = int  # <- Same for all formats.
validate(data, requirement)
```

A *data* value is treated as single element if it's a string, tuple, or non-iterable object.

A group of elements:

```
from datatest import validate

data = [1, 2, 3]
requirement = int  # <- Same for all formats.
validate(data, requirement)
```

A *data* value is treated as a group of elements if it's any iterable other than a string, tuple, or mapping (e.g., in this case a `list`).

A mapping of elements:

```
from datatest import validate

data = {'A': 1, 'B': 2, 'C': 3}
requirement = int  # <- Same for all formats.
validate(data, requirement)
```

When *data* is a mapping, its values are checked as individual elements if they are strings, tuples, non-iterable objects, or nested mappings.

A mapping of groups of elements:

```
from datatest import validate

data = {'X': [1, 2, 3], 'Y': [4, 5, 6], 'Z': [7, 8, 9]}
requirement = int  # <- Same for all formats.
validate(data, requirement)
```

A mapping's values are treated as a group of individual elements when they are any iterable other than a string, tuple, or another nested mapping.

Of course, not all formats are comparable. When *requirement* is itself a mapping, there's no clear way to handle validation if *data* is a single element or a non-mapping container. In cases like this, the validation process will error-out before the data elements can be checked.

In addition to built-in generic types, Datatest also provides automatic handling for several third-party data types.

Pandas

Pandas (integrated API)

NumPy

Squint

Databases

Datatest can work with `pandas` DataFrame, Series, Index, and MultiIndex objects:

```
import pandas as pd
import datatest as dt

df = pd.DataFrame([('x', 1, 12.25),
                   ('y', 2, 33.75),
                   ('z', 3, 101.5)],
                  columns=['A', 'B', 'C'])

dt.validate(df[['A', 'B']], (str, int))
```

For users who prefer a more tightly integrated API, Datatest provides the `validate` accessor for testing pandas objects:

```
import pandas as pd
import datatest as dt

dt.register_accessors()

df = pd.DataFrame([('x', 1, 12.25),
                   ('y', 2, 33.75),
                   ('z', 3, 101.5)],
                  columns=['A', 'B', 'C'])

df[['A', 'B']].validate((str, int))
```

After calling the *`register_accessors()`* function, you can use *`validate()`* as a *method* of your existing DataFrame, Series, Index, and MultiIndex objects.

Handling is also supported for `numpy` objects including one- or two-dimentional array, recarray, and structured array objects.

```
import numpy as np
import datatest as dt

a = np.array([('x', 1, 12.25),
              ('y', 2, 33.75),
              ('z', 3, 101.5)],
             dtype='U10, int32, float32')

dt.validate(a[['f0', 'f1']], (str, int))
```

Datatest also works well with `squint` Select, Query, and Result objects:

```
from squint import Select
from datatest import validate

select = Select([('A', 'B', 'C'),
                 ('x', 1, 12.25),
                 ('y', 2, 33.75),
                 ('z', 3, 101.5)])

validate(select(('A', 'B')), (str, int))
```

Origins

Squint was originally part of Datatest itself—it grew out of Datatest's old validation API. But as Datatest matured, the need for a built-in query interface stoped making sense. This *simple query interface* was named "Squint" and the code was moved into its own project.

Database queries can also be validated directly:

```
import sqlite3
from datatest import validate

conn = sqlite3.connect(':memory:')
conn.executescript('''
```

(continues on next page)

(continued from previous page)

```
    CREATE TABLE mydata(A, B, C);
    INSERT INTO mydata VALUES('x', 1, 12.25);
    INSERT INTO mydata VALUES('y', 2, 33.75);
    INSERT INTO mydata VALUES('z', 3, 101.5);
''')
cursor = conn.cursor()

cursor.execute('SELECT A, B FROM mydata;')
validate(cursor, (str, int))
```

This requires a `cursor` object to conform to Python's DBAPI2 specification (see **PEP 249**). Most of Python's database packages support this interface.

Errors

When validation fails, a *`ValidationError`* is raised. A ValidationError contains a collection of difference objects—one difference for each element in *data* that fails to satisfy the *requirement*.

Difference objects can be one of four types: *`Missing`*, *`Extra`*, *`Deviation`* or *`Invalid`*.

"Missing" Differences

In this example, we check that the list `['A', 'B']` contains members of the set `{'A', 'B', 'C', 'D'}`:

```
from datatest import validate

data = ['A', 'B']
requirement = {'A', 'B', 'C', 'D'}
validate(data, requirement)
```

This fails because the elements `'C'` and `'D'` are not present in *data*. They appear below as *`Missing`* differences:

```
Traceback (most recent call last):
  File "example.py", line 5, in <module>
    validate(data, requirement)
datatest.ValidationError: does not satisfy set membership (2 differences): [
    Missing('C'),
    Missing('D'),
]
```

"Extra" Differences

In this next example, we will reverse the previous situation by checking that elements in the list `['A', 'B', 'C', 'D']` are members of the set `{'A', 'B'}`:

```
from datatest import validate

data = ['A', 'B', 'C', 'D']
requirement = {'A', 'B'}
validate(data, requirement)
```

Of course, this validation fails because the elements `'C'` and `'D'` are not members of the *requirement* set. They appear below as *`Extra`* differences:

```
Traceback (most recent call last):
  File "example.py", line 5, in <module>
    validate(data, requirement)
datatest.ValidationError: does not satisfy set membership (2 differences): [
    Extra('C'),
    Extra('D'),
]
```

"Invalid" Differences

In this next example, the *requirement* is a tuple, `(str, is_even)`. It checks for tuple elements where the first value is a string and the second value is an even number:

```
from datatest import validate

data = [('a', 2), ('b', 4), ('c', 6), (1.25, 8), ('e', 9)]

def is_even(x):
    return x % 2 == 0

requirement = (str, is_even)
validate(data, requirement)
```

Two of the elements in *data* fail to satisfy the *requirement*: `(1.25, 8)` fails because 1.25 is not a string and `('e', 9)` fails because 9 is not an even number. These are represented in the error as *`Invalid`* differences:

```
Traceback (most recent call last):
  File "example.py", line 9, in <module>
    validate(data, requirement)
datatest.ValidationError: does not satisfy `(str, is_even())` (2 differences): [
    Invalid((1.25, 8)),
    Invalid(('e', 9)),
]
```

"Deviation" Differences

In the following example, the *requirement* is a dictionary of numbers. The *data* elements are checked against *requirement* elements of the same key:

```
from datatest import validate

data = {
    'A': 100,
    'B': 200,
    'C': 299,
    'D': 405,
}

requirement = {
    'A': 100,
    'B': 200,
    'C': 300,
    'D': 400,
```

(continues on next page)

(continued from previous page)

```
}

validate(data, requirement)
```

This validation fails because some of the values don't match (C: `299 300` and D: `405 400`). Failed quantitative comparisons raise *`Deviation`* differences:

```
Traceback (most recent call last):
  File "example.py", line 17, in <module>
    validate(data, requirement)
datatest.ValidationError: does not satisfy mapping requirements (2 differences): {
    'C': Deviation(-1, 300),
    'D': Deviation(+5, 400),
}
```

Acceptances

Sometimes a failing test cannot be addressed by changing the data itself. Perhaps two equally-authoritative sources disagree, perhaps it's important to keep the original data unchanged, or perhaps a lack of information makes correction impossible. For cases like these, datatest can accept certain discrepancies when users judge that doing so is appropriate.

The *`accepted()`* function returns a context manager that operates on a ValidationError's collection of differences.

Accepted Type

Without an acceptance, the following validation would fail because the values `'C'` and `'D'` are not members of the set (see below). But if we decide that *`Extra`* differences are acceptible, we can use `accepted(Extra)`:

Using Acceptance

No Acceptance

```
from datatest import (
    validate,
    accepted,
    Extra,
)

data = ['A', 'B', 'C', 'D']
requirement = {'A', 'B'}
with accepted(Extra):
    validate(data, requirement)
```

```
from datatest import (
    validate,
    accepted,
    Extra,
)

data = ['A', 'B', 'C', 'D']
requirement = {'A', 'B'}
validate(data, requirement)
```

```
Traceback (most recent call last):
  File "example.py", line 9, in <module>
    validate(data, requirement)
datatest.ValidationError: does not satisfy set membership (2 differences): [
    Extra('C'),
    Extra('D'),
]
```

Using the acceptance, we suppress the error caused by all of the *Extra* differences. But without the acceptance, the ValidationError is raised.

Accepted Instance

If we want more precision, we can accept a specific difference—rather than all differences of a given type. For example, if the difference *Extra('C')* is acceptible, we can use `accepted(Extra('C'))`:

Using Acceptance

No Acceptance

```
from datatest import (
    validate,
    accepted,
    Extra,
)

data = ['A', 'B', 'C', 'D']
requirement = {'A', 'B'}
with accepted(Extra('C')):
    validate(data, requirement)
```

```
Traceback (most recent call last):
  File "example.py", line 10, in <module>
    validate(data, requirement)
datatest.ValidationError: does not satisfy set membership (1 difference): [
    Extra('D'),
]
```

This acceptance suppresses the extra `'C'` but does not address the extra `'D'` so the ValidationError is still raised. This remaining error can be addressed by correcting the data, modifying the requirement, or altering the acceptance.

```
from datatest import (
    validate,
    accepted,
    Extra,
)

data = ['A', 'B', 'C', 'D']
requirement = {'A', 'B'}
validate(data, requirement)
```

```
Traceback (most recent call last):
  File "example.py", line 9, in <module>
    validate(data, requirement)
datatest.ValidationError: does not satisfy set membership (2 differences): [
```

(continues on next page)

(continued from previous page)

```
    Extra('C'),
    Extra('D'),
]
```

Accepted Container of Instances

We can also accept multiple specific differences by defining a container of difference objects. To build on the previous example, we can use `accepted([Extra('C'), Extra('D')])` to accept the two differences explicitly:

Using Acceptance

No Acceptance

```
from datatest import (
    validate,
    accepted,
    Extra,
)

data = ['A', 'B', 'C', 'D']
requirement = {'A', 'B'}
with accepted([Extra('C'), Extra('D')]):
    validate(data, requirement)
```

```
from datatest import (
    validate,
    accepted,
    Extra,
)

data = ['A', 'B', 'C', 'D']
requirement = {'A', 'B'}
validate(data, requirement)
```

```
Traceback (most recent call last):
  File "example.py", line 9, in <module>
    validate(data, requirement)
datatest.ValidationError: does not satisfy set membership (2 differences): [
    Extra('C'),
    Extra('D'),
]
```

Accepted Tolerance

When comparing quantative values, you may decide that deviations of a certain magnitude are acceptible. Calling *`accepted.tolerance(5)`* returns a context manager that accepts differences within a tolerance of plus-or-minus five without triggering a test failure:

Using Acceptance

No Acceptance

```
from datatest import validate
from datatest import accepted

data = {
    'A': 100,
    'B': 200,
    'C': 299,
    'D': 405,
}
requirement = {
    'A': 100,
    'B': 200,
    'C': 300,
    'D': 400,
}
with accepted.tolerance(5):  # accepts ±5
    validate(data, requirement)
```

```
from datatest import validate
from datatest import accepted

data = {
    'A': 100,
    'B': 200,
    'C': 299,
    'D': 405,
}
requirement = {
    'A': 100,
    'B': 200,
    'C': 300,
    'D': 400,
}
validate(data, requirement)
```

```
Traceback (most recent call last):
  File "example.py", line 16, in <module>
    validate(data, requirement)
datatest.ValidationError: does not satisfy mapping requirements (2 differences): {
    'C': Deviation(-1, 300),
    'D': Deviation(+5, 400),
}
```

Other Acceptances

In addtion to the previous examples, there are other acceptances available for specific cases—*accepted.keys()*, *accepted.args()*, *accepted.percent()*, etc. For a list of all possible acceptances, see *Acceptances*.

Combining Acceptances

Acceptances can also be combined using the operators `&` and `|` to define more complex criteria:

```
from datatest import (
    validate,
    accepted,
)

# Accept up to five missing differences.
with accepted(Missing) & accepted.count(5):
    validate(..., ...)

# Accept differences of ±10 or ±5%.
with accepted.tolerance(10) | accepted.percent(0.05):
    validate(..., ...)
```

To learn more about these features, see *Composability* and *Order of Operations*.

Data Handling Tools

Working Directory

You can use *working_directory* (a context manager and decorator) to assure that relative file paths behave consistently:

```
import pandas as pd
from datatest import working_directory

with working_directory(__file__):
    my_df = pd.read_csv('myfile.csv')
```

Repeating Container

You can use a *RepeatingContainer* to operate on multiple objects at the same time rather than duplicating the same operation for each object:

Using RepeatingContainer

No RepeatingContainer

```
import pandas as pd
from datatest import RepeatingContainer

repeating = RepeatingContainer([
    pd.read_csv('file1.csv'),
    pd.read_csv('file2.csv'),
])
```

(continues on next page)

(continued from previous page)

```
counted1, counted2 = repeating['C'].count()

filled1, filled2 = repeating.fillna(method='backfill')

summed1, summed2 = repeating[['A', 'C']].groupby('A').sum()
```

In the three statements above, operations are performed on multiple pandas DataFrames using single lines of code. Results are then unpacked into individual variable names. Compare this example with code in the "No RepeatingContainer" tab.

```
import pandas as pd

df1 = pd.read_csv('file1.csv')
df2 = pd.read_csv('file2.csv')

counted1 = df1['C'].count()
counted2 = df2['C'].count()

filled1 = df1.fillna(method='backfill')
filled2 = df2.fillna(method='backfill')

summed1 = df1[['A', 'C']].groupby('A').sum()
summed2 = df2[['A', 'C']].groupby('A').sum()
```

Without a RepeatingContainer, operations are duplicated for each individual DataFrame.

1.1.2 Automated Data Testing

In addition to being used directly in your own projects, you can also use Datatest with a testing framework like pytest or unittest. Automated testing of data is a good solution when you need to validate and manage:

- batch data before loading
- datasets for an important project
- datasets intended for publication
- status of a long-lived, critical data system
- comparisons between your data and some reference data
- data migration projects
- complex data-wrangling processes

Data testing is a form of *acceptance testing*—akin to operational acceptance testing. Using an incremental approach, we check that data properties satisfy certain requirements. A test suite should include as many tests as necessary to determine if a dataset is *fit for purpose*.

Pytest

With `pytest`, you can use datatest functions and classes just as you would in any other context. And you can run pytest using its normal, console interface (see Usage and Invocations).

To facilitate incremental testing, datatest implements a "mandatory" marker to stop the session early when a mandatory test fails:

```
@pytest.mark.mandatory
def test_columns():
    ...
```

You can also use the `-x` option to stop testing after the first failure of any test:

```
pytest -x
```

If needed, you can use `--ignore-mandatory` to ignore "mandatory" markers and continue testing even when a mandatory test fails:

```
pytest --ignore-mandatory
```

Pytest Samples

Pandas

Pandas (integrated)

Squint

SQL

```
#!/usr/bin/env python3
import pytest
import pandas as pd
import datatest as dt
from datatest import (
    Missing,
    Extra,
    Invalid,
    Deviation,
)


@pytest.fixture(scope='session')
@dt.working_directory(__file__)
def df():
    return pd.read_csv('example.csv')  # Returns DataFrame.


@pytest.mark.mandatory
def test_column_names(df):
    required_names = {'A', 'B', 'C'}
    dt.validate(df.columns, required_names)


def test_a(df):
    requirement = {'x', 'y', 'z'}
```

(continues on next page)

(continued from previous page)

```
    dt.validate(df['A'], requirement)


# ...add more tests here...


if __name__ == '__main__':
    import sys
    sys.exit(pytest.main(sys.argv))
```

```
#!/usr/bin/env python3
import pytest
import pandas as pd
import datatest as dt
from datatest import (
    Missing,
    Extra,
    Invalid,
    Deviation,
)


@pytest.fixture(scope='session')
@dt.working_directory(__file__)
def df():
    return pd.read_csv('example.csv')  # Returns DataFrame.


@pytest.fixture(scope='session', autouse=True)
def pandas_integration():
    dt.register_accessors()


@pytest.mark.mandatory
def test_column_names(df):
    required_names = {'A', 'B', 'C'}
    df.columns.validate(required_names)


def test_a(df):
    requirement = {'x', 'y', 'z'}
    df['A'].validate(requirement)


# ...add more tests here...


if __name__ == '__main__':
    import sys
    sys.exit(pytest.main(sys.argv))
```

```
#!/usr/bin/env python3
import pytest
import squint
from datatest import (
    validate,
```

(continues on next page)

(continued from previous page)

```
    accepted,
    working_directory,
    Missing,
    Extra,
    Invalid,
    Deviation,
)


@pytest.fixture(scope='session')
@working_directory(__file__)
def select():
    return squint.Select('example.csv')


@pytest.mark.mandatory
def test_column_names(select):
    required_names = {'A', 'B', 'C'}
    validate(select.fieldnames, required_names)


def test_a(select):
    requirement = {'x', 'y', 'z'}
    validate(select('A'), requirement)


# ...add more tests here...


if __name__ == '__main__':
    import sys
    sys.exit(pytest.main(sys.argv))
```

```
#!/usr/bin/env python3
import pytest
import sqlite3
from datatest import (
    validate,
    accepted,
    working_directory,
    Missing,
    Extra,
    Invalid,
    Deviation,
)


@pytest.fixture(scope='session')
def connection():
    with working_directory(__file__):
        conn = sqlite3.connect('example.sqlite3')
    yield conn
    conn.close()


@pytest.fixture(scope='function')
```

(continues on next page)

(continued from previous page)

```
def cursor(connection):
    cur = connection.cursor()
    yield cur
    cur.close()


@pytest.mark.mandatory
def test_column_names(cursor):
    cursor.execute('SELECT * FROM mytable LIMIT 0;')
    column_names = [item[0] for item in cursor.description]
    required_names = {'A', 'B', 'C'}
    validate(column_names, required_names)


def test_a(cursor):
    cursor.execute('SELECT A FROM mytable;')
    requirement = {'x', 'y', 'z'}
    validate(cursor, requirement)


# ...add more tests here...


if __name__ == '__main__':
    import sys
    sys.exit(pytest.main(sys.argv))
```

Unittest

Datatest provides a handful of tools for integrating data validation with a `unittest` test suite. While normal datatest functions work fine, this integration provides an interface that is more consistent with established unittest conventions (e.g., "mixedCase" methods, decorators, and helper classes).

Datatest's *`DataTestCase`* extends `unittest.TestCase` to provide unittest-style wrappers for validation and acceptance (see *reference docs* for full details):

```
from datatest import DataTestCase, Extra

class TestMyData(DataTestCase):
    def test_one(self):
        data = ['A', 'B', 'C', 'D']
        requirement = {'A', 'B'}
        with self.accepted(Extra):
            self.assertValid(data, requirement)
```

Datatest includes a *`@mandatory`* decorator to help with incremental testing:

```
from datatest import DataTestCase, mandatory

class TestMyData(DataTestCase):
    @mandatory
    def test_one(self):
        data = ['A', 'A', 'B', 'B']
        requirement = {'A', 'B'}
        self.assertValid(data, requirement)
```

Datatest also provides a *main()* function and test runner that runs tests in decleration order (by the line number on which each test is defined). You can invoke datatest's runner using:

```
python -m datatest
```

In addition to using the *@mandatory* decorator, you can use the `-f` option to stop after any failing test:

```
python -m datatest -f
```

You can also use features directly from `unittest`. This includes decorators like `@skip()` and `@skipIf()`, functions like `addModuleCleanup()`, and features like Class and Module Fixtures:

```
import unittest
from datatest import DataTestCase

class TestMyData(DataTestCase):
    @unittest.skip('Data not yet collected.')
    def test_one(self):
        data = ...
        requirement = ...
        self.assertValid(data, requirement)
```

Unittest Samples

Pandas

Pandas (integrated)

Squint

SQL

```
#!/usr/bin/env python3
import pandas as pd
import datatest as dt
from datatest import (
    Missing,
    Extra,
    Invalid,
    Deviation,
)


@dt.working_directory(__file__)
def setUpModule():
    global df
    df = pd.read_csv('example.csv')


class TestMyData(dt.DataTestCase):
    @dt.mandatory
    def test_column_names(self):
        required_names = {'A', 'B', 'C'}
        self.assertValid(df.columns, required_names)

    def test_a(self):
        requirement = {'x', 'y', 'z'}
```

(continues on next page)

(continued from previous page)

```
        self.assertValid(df['A'], requirement)

    # ...add more tests here...


if __name__ == '__main__':
    from datatest import main
    main()
```

```
#!/usr/bin/env python3
import pandas as pd
import datatest as dt
from datatest import (
    Missing,
    Extra,
    Invalid,
    Deviation,
)


@dt.working_directory(__file__)
def setUpModule():
    global df
    df = pd.read_csv('example.csv')
    dt.register_accessors()  # Register pandas accessors.


class TestMyData(dt.DataTestCase):
    @dt.mandatory
    def test_column_names(self):
        required_names = {'A', 'B', 'C'}
        df.columns.validate(required_names)

    def test_a(self):
        requirement = {'x', 'y', 'z'}
        df['A'].validate(requirement)

    # ...add more tests here...


if __name__ == '__main__':
    from datatest import main
    main()
```

```
#!/usr/bin/env python3
import squint
from datatest import (
    DataTestCase,
    mandatory,
    working_directory,
    Missing,
    Extra,
    Invalid,
    Deviation,
)

```

(continues on next page)

(continued from previous page)

```

@working_directory(__file__)
def setUpModule():
    global select
    select = squint.Select('example.csv')


class TestMyData(DataTestCase):
    @mandatory
    def test_column_names(self):
        required_names = {'A', 'B', 'C'}
        self.assertValid(select.fieldnames, required_names)

    def test_a(self):
        requirement = {'x', 'y', 'z'}
        self.assertValid(select('A'), requirement)

    # ...add more tests here...


if __name__ == '__main__':
    from datatest import main
    main()
```

```
#!/usr/bin/env python3
import sqlite3
from datatest import (
    DataTestCase,
    mandatory,
    working_directory,
    Missing,
    Extra,
    Invalid,
    Deviation,
)


@working_directory(__file__)
def setUpModule():
    global connection
    connection = sqlite3.connect('example.sqlite3')


def tearDownModule():
    connection.close()


class MyTest(DataTestCase):
    def setUp(self):
        cursor = connection.cursor()
        self.addCleanup(cursor.close)

        self.cursor = cursor

    @mandatory
    def test_column_names(self):
```

(continues on next page)

(continued from previous page)

```
        self.cursor.execute('SELECT * FROM mytable LIMIT 0;')
        column_names = [item[0] for item in self.cursor.description]
        required_names = {'A', 'B', 'C'}
        self.assertValid(column_names, required_names)

    def test_a(self):
        self.cursor.execute('SELECT A FROM mytable;')
        requirement = {'x', 'y', 'z'}
        self.assertValid(self.cursor, requirement)


if __name__ == '__main__':
    from datatest import main
    main()
```

Data for Script Samples

Download File

example.csv
example.sqlite3

The test samples given on this page were written to check the following dataset:

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

1.1.3 Data Pipeline Validation

Datatest can be used to validate data as it flows through a data pipeline. This can be useful in a production environment because the data coming into a pipeline can change in unexpected ways. An up-stream provider could alter its format, the quality of a data source could degrade over time, previously unheard-of errors or missing data could be introduced, etc.

Well-structured pipelines are made of discrete, independent steps where the output of one step becomes the input of the next step. In the simplest case, the steps themselves can be functions. And in a pipeline framework, the steps are often a type of "task" or "job" object.

A simple pipeline could look something like the following:

```
...

def pipeline(file_path):
    data = load_from_source(file_path)   # STEP 1
```

(continues on next page)

(continued from previous page)

```
    data = operation_one(data)          # STEP 2

    data = operation_two(data)          # STEP 3

    save_to_destination(data)           # STEP 4
```

You can simply add calls to *validate()* *between* the existing steps:

```
...

def pipeline(file_path):
    data = load_from_source(file_path)  # STEP 1

    validate(data.columns, ['user_id', 'first_name', 'last_name'])

    data = operation_one(data)          # STEP 2

    validate(data.columns, ['user_id', 'full_name'])
    validate(data, (int, str))

    data = operation_two(data)          # STEP 3

    validate.unique(data['user_id'])

    save_to_destination(data)           # STEP 4
```

You could go further in a more sophisticated pipeline framework and define tasks dedicated specifically to validation.

Tip: When possible, it's best to call *validate()* once for a batch of data rather than for individual elements. Doing this is more efficient and failures provide more context when fixing data issues or defining appropriate acceptances.

```
# Efficient:

validate(data, int)

for x in data:
    myfunc(x)
```

```
# Inefficient:

for x in data:
    validate(x, int)
    myfunc(x)
```

Toggle Validation On/Off Using __debug__

Sometimes it's useful to perform comprehensive validation for debugging purposes but disable validation for production runs. You can use Python's `__debug__` constant to toggle validation on or off as needed:

```
...

def pipeline(file_path):
    data = load_from_source(file_path)  # STEP 1

    if __debug__:
        validate(data.columns, ['user_id', 'first_name', 'last_name'])

    data = operation_one(data)          # STEP 2

    if __debug__:
        validate(data.columns, ['user_id', 'full_name'])
        validate(data, (int, str))

    data = operation_two(data)          # STEP 3

    if __debug__:
        validate.unique(data['user_id'])

    save_to_destination(data)           # STEP 4
```

Validation On

In the example above, you can run the pipeline with validation by running Python in *unoptimized* mode. In unoptimized mode, `__debug__` is True and `assert` statements are executed normally. Unoptimized mode is the default mode when invoking Python:

```
python simple_pipeline.py
```

Validation Off

To run the example above without validation, run Python in *optimized* mode. In optimized mode, `__debug__` is False and `assert` statements are skipped. You can invoke optimized mode using the `-O` command line option:

```
python -O simple_pipeline.py
```

Validate a Sample from a Larger Data Set

Another option for dealing with large data sets is to validate a small sample of the data. Doing this can provide some basic sanity checking in a production pipeline but it could also allow some invalid data to pass unnoticed. Users must decide if this approach is appropriate for their specific use case.

DataFrame Example

With Pandas, you can use the `DataFrame.sample()` method to get a random sample of items for validation:

```
...

def pipeline(file_path):
    data = load_from_source(file_path)  # STEP 1

    validate(data.columns, ['user_id', 'first_name', 'last_name'])

    data = operation_one(data)          # STEP 2

    sample = data.sample(n=100)
    validate(sample.columns, ['user_id', 'full_name'])
    validate(sample, (int, str))

    data = operation_two(data)          # STEP 3

    sample = data.sample(n=100)
    validate.unique(sample['user_id'])

    save_to_destination(data)           # STEP 4
```

Iterator Example

Sometimes you will need to work with exhaustible iterators of unknown size. It's possible for an iterator to yield more data than you can load into memory at any one time. Using Python's `itertools` module, you can fetch a sample of data for validation and then reconstruct the iterator to continue with data processing:

```
import itertools

...
```

```
...

def pipeline(file_path):
    iterator = load_from_source(file_path)  # STEP 1

    iterator = operation_one(iterator)      # STEP 2

    sample = list(itertools.islice(iterator, 100))
    validate(sample, (int, str))
    iterator = itertools.chain(sample, iterator)

    iterator = operation_two(iterator)      # STEP 3

    sample = list(itertools.islice(iterator, 100))
    validate.unique(item[0] for item in sample)
    iterator = itertools.chain(sample, iterator)

    save_to_destination(iterator)           # STEP 4
```

Tip: If you need perform multiple sampling operations on exhaustible iterators, you may want to define a function for doing so:

```
import itertools

def get_sample(iterable, n=100):
    iterator = iter(iterable)
    sample = list(itertools.islice(iterator, n))
    iterator = itertools.chain(sample, iterator)
    return sample, iterator

...
```

Calling this function returns a tuple that contains a *sample* and the reconstructed *iterator*:

```
...

sample, iterator = get_sample(iterator)
validate(sample, (int, str))

...
```

Important: As previously noted, validating samples of a larger data set should be done with care. If the sample is not representative of the data set as a whole, some validations could fail even when the data is good and some validations could pass even when the data is bad.

In some of the examples above, there are calls to *`validate.unique()`* but this validation only checks data included in the sample—duplicates in the remaining data set could pass unnoticed. This may be acceptable for some situations but not for others.

Testing Pipeline Code Itself

The pipeline validation discussed in this document is not a replacement for proper testing of the pipeline's code base itself. Pipeline code, should be treated with the same care and attention as any other software project.

1.1.4 Validating Pandas Objects

The `pandas` data analysis package is commonly used for data work. This page explains how datatest handles the validation of `DataFrame`, `Series`, `Index`, and `MultiIndex` objects.

Accessor Syntax

Examples on this page use the `validate` accessor:

```
# Accessor syntax:

df['A'].validate({'x', 'y', 'z'})
```

We could also use the equivalent non-accessor syntax:

```
# Basic syntax:

dt.validate(df['A'], {'x', 'y', 'z'})
```

DataFrame

For validation, `DataFrame` objects using the default index type are treated as sequences. DataFrames using an index of any other type are treated as mappings:

Default Index

Specified Index

```
import pandas as pd
import datatest as dt

dt.register_accessors()

df = pd.DataFrame(data={'A': ['foo', 'bar', 'baz', 'qux'],
                        'B': [10, 20, 'x', 'y']})


requirement = [
    ('foo', 10),
    ('bar', 20),
    ('baz', 'x'),
    ('qux', 'y'),
]

df.validate(requirement)
```

Since no index was specified, `df` uses the default `RangeIndex` type—which tells *`validate()`* to treat the DataFrame as a sequence.

```
import pandas as pd
import datatest as dt

dt.register_accessors()

df = pd.DataFrame(data={'A': ['foo', 'bar', 'baz', 'qux'],
                        'B': [10, 20, 'x', 'y']},
                  index=['I', 'II', 'III', 'IV'])

requirement = {
    'I': ('foo', 10),
    'II': ('bar', 20),
    'III': ('baz', 'x'),
    'IV': ('qux', 'y'),
}

df.validate(requirement)
```

In this example, we've specified an index and therefore `df` is handled as a mapping.

The distinction between implicit and explicit indexing is also apparent in error reporting. Compare the examples on each of the tabs below:

Default Index

Specified Index

```
import pandas as pd
import datatest as dt
```

(continues on next page)

(continued from previous page)

```
3
4  dt.register_accessors()
5
6  df = pd.DataFrame(data={'A': ['foo', 'bar', 'baz', 'qux'],
7                          'B': [10, 20, 'x', 'y']})
8
9
10 df.validate((str, int))
```

```
Traceback (most recent call last):
  File "example.py", line 10, in <module>
    df.validate((str, int))
datatest.ValidationError: does not satisfy `(str, int)` (2 differences): [
    Invalid(('baz', 'x')),
    Invalid(('qux', 'y')),
]
```

Since the DataFrame was treated as a sequence, the error includes a sequence of differences.

```
1  import pandas as pd
2  import datatest as dt
3
4  dt.register_accessors()
5
6  df = pd.DataFrame(data={'A': ['foo', 'bar', 'baz', 'qux'],
7                          'B': [10, 20, 'x', 'y']},
8                    index=['I', 'II', 'III', 'IV'])
9
10 df.validate((str, int))
```

```
Traceback (most recent call last):
  File "example.py", line 10, in <module>
    df.validate((str, int))
datatest.ValidationError: does not satisfy `(str, int)` (2 differences): {
    'III': Invalid(('baz', 'x')),
    'IV': Invalid(('qux', 'y')),
}
```

In this example, the DataFrame was treated as a mapping, so the error includes a mapping of differences.

Series

`Series` objects are handled the same way as DataFrames. Series with a default index are treated as sequences and Series with explicitly defined indexes are treated as mappings:

Default Index

Specified Index

```
1  import pandas as pd
2  import datatest as dt
3
4  dt.register_accessors()
5
6  s = pd.Series(data=[10, 20, 'x', 'y'])
```

(continues on next page)

(continued from previous page)

```


requirement = [10, 20, 'x', 'y']

s.validate(requirement)
```

```
import pandas as pd
import datatest as dt

dt.register_accessors()

s = pd.Series(data=[10, 20, 'x', 'y'],
              index=['I', 'II', 'III', 'IV'])

requirement = {'I': 10, 'II': 20, 'III': 'x', 'IV': 'y'}

s.validate(requirement)
```

Like before, the sequence and mapping handling is also apparent in the error reporting:

Default Index

Specified Index

```
import pandas as pd
import datatest as dt

dt.register_accessors()

s = pd.Series(data=[10, 20, 'x', 'y'])


s.validate(int)
```

```
Traceback (most recent call last):
  File "example.py", line 9, in <module>
    s.validate(int)
datatest.ValidationError: does not satisfy `int` (2 differences): [
    Invalid('x'),
    Invalid('y'),
]
```

```
import pandas as pd
import datatest as dt

dt.register_accessors()

s = pd.Series(data=[10, 20, 'x', 'y'],
              index=['I', 'II', 'III', 'IV'])

s.validate(int)
```

```
Traceback (most recent call last):
  File "example.py", line 9, in <module>
    s.validate(int)
datatest.ValidationError: does not satisfy `int` (2 differences): {
```

(continues on next page)

(continued from previous page)

```
    'III': Invalid('x'),
    'IV': Invalid('y'),
}
```

Index and MultiIndex

`Index` and `MultiIndex` objects are all treated as sequences:

```
import pandas as pd
import datatest as dt

dt.register_accessors()

index = pd.Index(['I', 'II', 'III', 'IV'])
requirement = ['I', 'II', 'III', 'IV']
index.validate(requirement)

multi = pd.MultiIndex.from_tuples([
    ('I', 'a'),
    ('II', 'b'),
    ('III', 'c'),
    ('IV', 'd'),
])
requirement = [('I', 'a'), ('II', 'b'), ('III', 'c'), ('IV', 'd')]
multi.validate(requirement)
```

1.2 How-to Guide

> *"Hell is other people's data."* —Jim Harris[1]

1.2.1 How to Install Datatest

The easiest way to install datatest is to use pip:

```
pip install datatest
```

If you are upgrading from version 0.11.0 or newer, use the `--upgrade` option:

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

[1] Harris, Jim. "Hell is other people's data", OCDQ (blog), August 06, 2010, Retrieved from http://www.ocdqblog.com/home/hell-is-other-peoples-data.html

Upgrading From Version 0.9.6

If you have an existing codebase of older datatest scripts, you should upgrade using the following steps:

- Install datatest 0.10.0 first:

```
pip install --force-reinstall datatest==0.10.0
```

- Run your existing code and check for DeprecationWarnings.
- Update the parts of your code that use deprecated features.
- Once your code is running without DeprecationWarnings, install the latest version of datatest:

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

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.

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.org/project/datatest/#files

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 NumPy arrays) 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
```

1.2.2 How to Get Started With Testing

Once you have reviewed the tutorials and have a basic understanding of datatest, you should be ready to start testing your own data.

1. Create a File and Add Some Sample Code

A simple way to get started is to create a **.py** file in the same folder as the data you want to test. It's a good idea to follow established testing conventions and make sure your filename starts with "**test_**".

Then, copy one of following the **pytest** or **unittest** code samples to use as a template for writing your own tests:

Pandas

Pandas (integrated)

Squint

SQL

```
#!/usr/bin/env python3
import pytest
import pandas as pd
import datatest as dt
from datatest import (
    Missing,
    Extra,
    Invalid,
    Deviation,
)


@pytest.fixture(scope='session')
@dt.working_directory(__file__)
def df():
    return pd.read_csv('example.csv')  # Returns DataFrame.


@pytest.mark.mandatory
def test_column_names(df):
    required_names = {'A', 'B', 'C'}
    dt.validate(df.columns, required_names)


def test_a(df):
    requirement = {'x', 'y', 'z'}
    dt.validate(df['A'], requirement)


# ...add more tests here...


if __name__ == '__main__':
    import sys
    sys.exit(pytest.main(sys.argv))
```

```
#!/usr/bin/env python3
import pytest
```

(continues on next page)

(continued from previous page)

```
import pandas as pd
import datatest as dt
from datatest import (
    Missing,
    Extra,
    Invalid,
    Deviation,
)


@pytest.fixture(scope='session')
@dt.working_directory(__file__)
def df():
    return pd.read_csv('example.csv')  # Returns DataFrame.


@pytest.fixture(scope='session', autouse=True)
def pandas_integration():
    dt.register_accessors()


@pytest.mark.mandatory
def test_column_names(df):
    required_names = {'A', 'B', 'C'}
    df.columns.validate(required_names)


def test_a(df):
    requirement = {'x', 'y', 'z'}
    df['A'].validate(requirement)


# ...add more tests here...


if __name__ == '__main__':
    import sys
    sys.exit(pytest.main(sys.argv))
```

```
#!/usr/bin/env python3
import pytest
import squint
from datatest import (
    validate,
    accepted,
    working_directory,
    Missing,
    Extra,
    Invalid,
    Deviation,
)


@pytest.fixture(scope='session')
@working_directory(__file__)
def select():
```

(continues on next page)

(continued from previous page)

```
    return squint.Select('example.csv')


@pytest.mark.mandatory
def test_column_names(select):
    required_names = {'A', 'B', 'C'}
    validate(select.fieldnames, required_names)


def test_a(select):
    requirement = {'x', 'y', 'z'}
    validate(select('A'), requirement)


# ...add more tests here...


if __name__ == '__main__':
    import sys
    sys.exit(pytest.main(sys.argv))
```

```
#!/usr/bin/env python3
import pytest
import sqlite3
from datatest import (
    validate,
    accepted,
    working_directory,
    Missing,
    Extra,
    Invalid,
    Deviation,
)


@pytest.fixture(scope='session')
def connection():
    with working_directory(__file__):
        conn = sqlite3.connect('example.sqlite3')
    yield conn
    conn.close()


@pytest.fixture(scope='function')
def cursor(connection):
    cur = connection.cursor()
    yield cur
    cur.close()


@pytest.mark.mandatory
def test_column_names(cursor):
    cursor.execute('SELECT * FROM mytable LIMIT 0;')
    column_names = [item[0] for item in cursor.description]
    required_names = {'A', 'B', 'C'}
    validate(column_names, required_names)
```

(continues on next page)

(continued from previous page)

```


def test_a(cursor):
    cursor.execute('SELECT A FROM mytable;')
    requirement = {'x', 'y', 'z'}
    validate(cursor, requirement)


# ...add more tests here...


if __name__ == '__main__':
    import sys
    sys.exit(pytest.main(sys.argv))
```

Pandas

Pandas (integrated)

Squint

SQL

```
#!/usr/bin/env python3
import pandas as pd
import datatest as dt
from datatest import (
    Missing,
    Extra,
    Invalid,
    Deviation,
)


@dt.working_directory(__file__)
def setUpModule():
    global df
    df = pd.read_csv('example.csv')


class TestMyData(dt.DataTestCase):
    @dt.mandatory
    def test_column_names(self):
        required_names = {'A', 'B', 'C'}
        self.assertValid(df.columns, required_names)

    def test_a(self):
        requirement = {'x', 'y', 'z'}
        self.assertValid(df['A'], requirement)

    # ...add more tests here...


if __name__ == '__main__':
    from datatest import main
    main()
```

```python
#!/usr/bin/env python3
import pandas as pd
import datatest as dt
from datatest import (
    Missing,
    Extra,
    Invalid,
    Deviation,
)


@dt.working_directory(__file__)
def setUpModule():
    global df
    df = pd.read_csv('example.csv')
    dt.register_accessors()  # Register pandas accessors.


class TestMyData(dt.DataTestCase):
    @dt.mandatory
    def test_column_names(self):
        required_names = {'A', 'B', 'C'}
        df.columns.validate(required_names)

    def test_a(self):
        requirement = {'x', 'y', 'z'}
        df['A'].validate(requirement)

    # ...add more tests here...


if __name__ == '__main__':
    from datatest import main
    main()
```

```python
#!/usr/bin/env python3
import squint
from datatest import (
    DataTestCase,
    mandatory,
    working_directory,
    Missing,
    Extra,
    Invalid,
    Deviation,
)


@working_directory(__file__)
def setUpModule():
    global select
    select = squint.Select('example.csv')


class TestMyData(DataTestCase):
    @mandatory
    def test_column_names(self):
```

(continues on next page)

(continued from previous page)

```
        required_names = {'A', 'B', 'C'}
        self.assertValid(select.fieldnames, required_names)

    def test_a(self):
        requirement = {'x', 'y', 'z'}
        self.assertValid(select('A'), requirement)

    # ...add more tests here...


if __name__ == '__main__':
    from datatest import main
    main()
```

```
#!/usr/bin/env python3
import sqlite3
from datatest import (
    DataTestCase,
    mandatory,
    working_directory,
    Missing,
    Extra,
    Invalid,
    Deviation,
)


@working_directory(__file__)
def setUpModule():
    global connection
    connection = sqlite3.connect('example.sqlite3')


def tearDownModule():
    connection.close()


class MyTest(DataTestCase):
    def setUp(self):
        cursor = connection.cursor()
        self.addCleanup(cursor.close)

        self.cursor = cursor

    @mandatory
    def test_column_names(self):
        self.cursor.execute('SELECT * FROM mytable LIMIT 0;')
        column_names = [item[0] for item in self.cursor.description]
        required_names = {'A', 'B', 'C'}
        self.assertValid(column_names, required_names)

    def test_a(self):
        self.cursor.execute('SELECT A FROM mytable;')
        requirement = {'x', 'y', 'z'}
        self.assertValid(self.cursor, requirement)

```

(continues on next page)

(continued from previous page)

```
43
44 if __name__ == '__main__':
45     from datatest import main
46     main()
```

2. Adapt the Sample Code to Suit Your Data

After copying the sample code into your own file, begin adapting it to suit your data:

1. Change the fixture to use your data (instead of "example.csv").
2. Update the set in `test_column_names()` to require the names your data should contain (instead of "A", "B", and "C").
3. Rename `test_a()` and change it to check values in one of the columns in your data.
4. Add more tests appropriate for your own data requirements.

3. Refactor Your Tests as They Grow

As your tests grow, look to structure them into related groups. Start by creating separate classes to contain groups of related test cases. And as you develop more and more classes, create separate modules to hold groups of related classes. If you are using `pytest`, move your fixtures into a `conftest.py` file.

1.2.3 How to Run Tests

Pytest

If you have a pytest style script named `test_mydata.py`, you can run it by typing the following at the command line:

```
pytest test_mydata.py
```

You invoke pytest just as you would in any other circumstance—see pytest's standard Usage and Invocations for full details.

Unittest

If you have a unittest style script named `test_mydata.py`, you can run it by typing the following at the command line:

```
python -m datatest test_mydata.py
```

Datatest includes a unittest-style test runner that facilitates incremental testing. It runs tests in declaration order (i.e., by line-number) and supports the *@mandatory* decorator.

1.2.4 How to Validate Column Names

To validate the column names in a data source, simply pass the names themselves as the *data* argument when calling `validate()`. You control the method of validation with the *requirement* you provide.

Column Requirements

Exist in Any Order

Using a `set` requirement, we can check that column names exist but allow them to appear in any order:

```
column_names = ...
validate(column_names, {'A', 'B', 'C'})
```

A Subset/Superset of Required Columns

Using `validate.subset()`/`validate.superset()`, we can check that column names are a subset or superset of the required names:

```
column_names = ...
validate.subset(column_names, {'A', 'B', 'C', 'D', 'E'})
```

Defined in a Specific Order

Using a `list` requirement, we can check that column names exist and that they appear in a specified order:

```
column_names = ...
validate(column_names, ['A', 'B', 'C'])
```

Matches Custom Formats

Sometimes we don't care exactly what the column names are but we want to check that they conform to a specific format. To do this, we can define a helper function that performs any arbitrary comparison we want. Below, we check that column names are all written in uppercase letters:

```
def isupper(x):  # <- helper function
    return x.isupper()

column_names = ...
validate(column_names, isupper)
```

In addition to `isupper()`, there are other string methods that can be useful for validating specific formats: `islower()`, `isalpha()`, `isascii()`, `isidentifier()`, etc.

A More Complex Example

Below, we check that the column names start with two capital letters and end with one or more digits. The examples below demonstrate different ways of checking this format:

Regex Pattern

Helper Function

We can use *validate.regex()* to check that column names match a regular expression pattern. The pattern matches strings that start with two capital letters (`^[A-Z]{2}`) and end with one or more digits (`\d+$`):

```
column_names = ...
msg = 'Must have two capital letters followed by digits.'
validate.regex(column_names, r'^[A-Z]{2}\d+$', msg=msg)
```

This example performs the same validation as the Regex Pattern version, but uses slicing and string methods to implement the same requirement:

```
def two_letters_plus_digits(x):
    """Must have two capital letters followed by digits."""
    first_two_chars = x[:2]
    remaining_chars = x[2:]

    if not first_two_chars.isalpha():
        return False
    if not first_two_chars.isupper():
        return False
    return remaining_chars.isdigit()

column_names = ...
validate(column_names, two_letters_plus_digits)
```

Examples Using Various Data Sources

csv.reader()

```
import csv
from datatest import validate

with open('mydata.csv', newline='') as csvfile:
    reader = csv.reader(csvfile)

    header_row = next(reader)
    validate(header_row, {'A', 'B', 'C'})
```

csv.DictReader()

```
import csv
from datatest import validate

with open('mydata.csv', newline='') as csvfile:
    reader = csv.DictReader(csvfile)

    validate(reader.fieldnames, {'A', 'B', 'C'})
```

Pandas

```
import pandas as pd
import datatest as dt

df = pd.read_csv('mydata.csv')
dt.validate(df.columns, {'A', 'B', 'C'})
```

Pandas (Integrated)

```
import pandas as pd
import datatest as dt

dt.register_accessors()

df = pd.read_csv('mydata.csv')
df.columns.validate({'A', 'B', 'C'})
```

Squint

```
import squint
from datatest import validate

select = squint.Select('mydata.csv')
validate(select.fieldnames, {'A', 'B', 'C'})
```

Database

If you're using a DBAPI2 compatible connection (see **PEP 249**), you can get a table's column names using the **cursor.description** attribute:

```
import sqlite3
from datatest import validate

connection = sqlite3.connect('mydata.sqlite3')
cursor = connection.cursor()

cursor.execute('SELECT * FROM mytable LIMIT 0;')
```

(continues on next page)

(continued from previous page)

```
column_names = [item[0] for item in cursor.description]
validate(column_names, {'A', 'B', 'C'})
```

Above, we select all columns (`SELECT *`) from our table but limit the result to zero rows (`LIMIT 0`). Executing this query populates `cursor.description` even though no records are returned. We take the column name from each item in description (`item[0]`) and perform our validation.

1.2.5 How to Customize Differences

When using a helper function for validation, datatest's default behavior is to produce *Invalid* differences when the function returns False. But you can customize this behavior by returning a difference object instead of False. The returned difference is used in place of an automatically generated one.

Default Behavior

In the following example, the helper function checks that text values are upper case and have no extra whitespace. If the values are good, the function returns `True`, if the values are bad it returns `False`:

```
from datatest import validate


def wellformed(x):  # <- Helper function.
    """Must be upercase and no extra whitespace."""
    return x == ' '.join(x.split()) and x.isupper()

data = [
    'CAPE GIRARDEAU',
    'GREENE ',
    'JACKSON',
    'St. Louis',
]

validate(data, wellformed)
```

Each time the helper function returns `False`, an *Invalid* difference is created:

```
Traceback (most recent call last):
  File "example.py", line 15, in <module>
    validate(data, wellformed)
ValidationError: Must be upercase and no extra whitespace. (2 differences): [
    Invalid('GREENE '),
    Invalid('St. Louis'),
]
```

Custom Differences

In this example, the helper function returns a custom `BadWhitespace` or `NotUpperCase` difference for each bad value:

```
from datatest import validate, Invalid


class BadWhitespace(Invalid):
    """For strings with leading, trailing, or irregular whitespace."""


class NotUpperCase(Invalid):
    """For strings that aren't upper case."""


def wellformed(x):  # <- Helper function.
    """Must be upercase and no extra whitespace."""
    if x != ' '.join(x.split()):
        return BadWhitespace(x)
    if not x.isupper():
        return NotUpperCase(x)
    return True


data = [
    'CAPE GIRARDEAU',
    'GREENE ',
    'JACKSON',
    'St. Louis',
]

validate(data, wellformed)
```

These differences are use in the ValidationError:

```
Traceback (most recent call last):
  File "example.py", line 15, in <module>
    validate(data, wellformed)
ValidationError: Must be upercase and no extra whitespace. (2 differences): [
    BadWhitespace('GREENE '),
    NotUpperCase('St. Louis'),
]
```

Caution: Typically, you should try to **stick with existing differences** in your data tests. Only create a custom subclass when its meaning is evident and doing so helps your data preparation workflow.

Don't add a custom class when it doesn't benefit your testing process. At best, you're doing extra work for no added benefit. And at worst, an ambiguous or needlessly complex subclass can cause more problems than it solves.

If you need to resolve ambiguity in a validation, you can split the check into multiple calls. Below, we perform the same check demonstrated earlier using two *`validate()`* calls:

```
from datatest import validate

data = [
    'CAPE GIRARDEAU',
    'GREENE ',
```

```
    'JACKSON',
    'St. Louis',
]

def no_irregular_whitespace(x):  # <- Helper function.
    """Must have no irregular whitespace."""
    return x == ' '.join(x.split())

validate(data, no_irregular_whitespace)


def is_upper_case(x):  # <- Helper function.
    """Must be upper case."""
    return x.isupper()

validate(data, is_upper_case)
```

1.2.6 How to Validate Data Types

To check that data is of a particular type, call `validate()` with a type as the *requirement* argument (see *Predicates*).

Simple Type Checking

In the following example, we use the `float` type as the *requirement*. The elements in *data* are considered valid if they are float instances:

```
from datatest import validate

data = [0.0, 1.0, 2.0]
validate(data, float)
```

In this example, we use the `str` type as the *requirement*. The elements in *data* are considered valid if they are strings:

```
from datatest import validate

data = ['a', 'b', 'c']
validate(data, str)
```

Using a Tuple of Types

You can also use a **predicate tuple** to test the types contained in tuples. The elements in *data* are considered valid if the tuples contain a number followed by a string:

```
from numbers import Number
from datatest import validate

data = [(0.0, 'a'), (1.0, 'b'), (2, 'c'), (3, 'd')]
validate(data, (Number, str))
```

In the example above, the `Number` base class is used to check for numbers of any type (`int`, `float`, `complex`, `Decimal`, etc.).

Checking Pandas Types

Type Inference and Conversion

Import the `pandas` package:

```
>>> import pandas as pd
```

INFERENCE

When a column's values are all integers (`1`, `2`, and `3`), then Pandas infers an integer dtype:

```
>>> pd.Series([1, 2, 3])
0    1
1    2
2    3
dtype: int64
```

When a column's values are a mix of integers (`1` and `3`) and floating point numbers (`2.0`), then Pandas will infer a floating point dtype—notice that the original integers have been coerced into float values:

```
>>> pd.Series([1, 2.0, 3])
0    1.0
1    2.0
2    3.0
dtype: float64
```

When certain non-numeric types are present, `'three'`, then pandas will use a generic "object" dtype:

```
>>> pd.Series([1, 2.0, 'three'])
0        1
1        2
2    three
dtype: object
```

CONVERSION

When a dtype is specified, `dtype=float`, Pandas will attempt to convert values into the given type. Here, the integers are explicitly converted into float values:

```
>>> pd.Series([1, 2, 3], dtype=float)
0    1.0
1    2.0
2    3.0
dtype: float64
```

In this example, integers and floating point numbers are converted into string values, `dtype=str`:

```
>>> pd.Series([1, 2.0, 3], dtype=str)
0      1
1    2.0
2      3
dtype: object
```

When a value cannot be converted into a specified type, an error is raised:

```
>>> pd.Series([1, 2.0, 'three'], dtype=int)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "~/myproject/venv/lib64/python3.8/site-packages/pandas/core/series.py", line
  327, in __init__
    data = sanitize_array(data, index, dtype, copy, raise_cast_failure=True)
  File "~/myproject/venv/lib64/python3.8/site-packages/pandas/core/construction.py",
   line 447, in sanitize_array
    subarr = _try_cast(data, dtype, copy, raise_cast_failure)
  File "~/myproject/venv/lib64/python3.8/site-packages/pandas/core/construction.py",
   line 555, in _try_cast
    maybe_cast_to_integer_array(arr, dtype)
  File "~/myproject/venv/lib64/python3.8/site-packages/pandas/core/dtypes/cast.py",
  line 1674, in maybe_cast_to_integer_array
    casted = np.array(arr, dtype=dtype, copy=copy)
ValueError: invalid literal for int() with base 10: 'three'
```

SEE ALSO

For more details, see the Pandas documentation regarding object conversion.

Check the types for each row of elements within a `DataFrame`:

Passing

Failing

```
import pandas as pd
import datatest as dt

dt.register_accessors()

df = pd.DataFrame(data={'A': ['foo', 'bar', 'baz', 'qux'],
                        'B': [10, 20, 30, 40]})

df.validate((str, int))
```

```
import pandas as pd
import datatest as dt

dt.register_accessors()

df = pd.DataFrame(data={'A': ['foo', 'bar', 'baz', 'qux'],
                        'B': [10, 20, 'x', 'y']})

df.validate((str, int))
```

```
Traceback (most recent call last):
  File "example.py", line 9, in <module>
    df.validate((str, int))
datatest.ValidationError: does not satisfy `(str, int)` (2 differences): [
    Invalid(('baz', 'x')),
    Invalid(('qux', 'y')),
]
```

Check the type of each element, one column at a time:

Passing

Failing

```
import pandas as pd
import datatest as dt

dt.register_accessors()

df = pd.DataFrame(data={'A': ['foo', 'bar', 'baz', 'qux'],
                        'B': [10, 20, 30, 40]})

df['A'].validate(str)
df['B'].validate(int)
```

```
import pandas as pd
import datatest as dt

dt.register_accessors()

df = pd.DataFrame(data={'A': ['foo', 'bar', 'baz', 'qux'],
                        'B': [10, 20, 'x', 'y']})

df['A'].validate(str)
df['B'].validate(int)
```

```
Traceback (most recent call last):
  File "example.py", line 10, in <module>
    df['B'].validate(int)
datatest.ValidationError: does not satisfy `int` (2 differences): [
    Invalid('x'),
    Invalid('y'),
]
```

Check the `dtypes` of the columns themselves (not the elements they contain):

Passing

Failing

```
import pandas as pd
import numpy as np
import datatest as dt

dt.register_accessors()

df = pd.DataFrame(data={'A': ['foo', 'bar', 'baz', 'qux'],
                        'B': [10, 20, 30, 40]})

required = {
    'A': np.dtype(object),
    'B': np.dtype(int),
}
df.dtypes.validate(required)
```

```
import pandas as pd
import numpy as np
import datatest as dt

dt.register_accessors()
```

(continues on next page)

(continued from previous page)

```
df = pd.DataFrame(data={'A': ['foo', 'bar', 'baz', 'qux'],
                        'B': [10, 20, 'x', 'y']})

required = {
    'A': np.dtype(object),
    'B': np.dtype(int),
}
df.dtypes.validate(required)
```

```
Traceback (most recent call last):
  File "example.py", line 14, in <module>
    df.dtypes.validate(required)
datatest.ValidationError: does not satisfy `dtype('int64')` (1 difference): {
    'B': Invalid(dtype('O'), expected=dtype('int64')),
}
```

NumPy Types

Type Inference and Conversion

Import the `numpy` package:

```
>>> import numpy as np
```

INFERENCE

When a column's values are all integers (`1`, `2`, and `3`), then NumPy infers an integer dtype:

```
>>> a = np.array([1, 2, 3])
>>> a
array([1, 2, 3])
>>> a.dtype
dtype('int64')
```

When a column's values are a mix of integers (`1` and `3`) and floating point numbers (`2.0`), then NumPy will infer a floating point dtype—notice that the original integers have been coerced into float values:

```
>>> a = np.array([1, 2.0, 3])
>>> a
array([1., 2., 3.])
>>> a.dtype
dtype('float64')
```

When given a string, `'three'`, NumPy will infer a unicode text dtype. This is different than how Pandas handles the situation. Notice that all of the values are converted to text:

```
>>> a = np.array([1, 2.0, 'three'])
>>> a
array(['1', '2.0', 'three'], dtype='<U32')
>>> a.dtype
dtype('<U32')
```

When certain non-numeric types are present, e.g. `{4}`, then Numpy will use a generic "object" dtype. In this case, the values maintain their original types—no conversion takes place:

```
>>> a = np.array([1, 2.0, 'three', {4}])
>>> a
array([1, 2.0, 'three', {4}], dtype=object)
>>> a.dtype
dtype('O')
```

CONVERSION

When a dtype is specified, `dtype=float`, NumPy will attempt to convert values into the given type. Here, the integers are explicitly converted into float values:

```
>>> a = np.array([1, 2, 3], dtype=float)
>>> a
array([1., 2., 3.])
>>> a.dtype
dtype('float64')
```

In this example, integers and floating point numbers are converted into unicode text values, `dtype=str`:

```
>>> a = np.array([1, 2.0, 3], dtype=str)
>>> a
array(['1', '2.0', '3'], dtype='<U3')
>>> a.dtype
dtype('<U3')
```

When a value cannot be converted into a specified type, an error is raised:

```
>>> a = np.array([1, 2.0, 'three'], dtype=int)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'three'
```

For more details on NumPy types see:

- https://numpy.org/doc/stable/reference/arrays.scalars.html
- https://numpy.org/doc/stable/reference/arrays.dtypes.html

With *Predicate* matching, you can use Python's built-in `str`, `int`, `float`, and `complex` to validate types in NumPy arrays.

Check the type of each element in a one-dimentional array:

Passing

Failing

```
import numpy as np
import datatest as dt

a = np.array([1.0, 2.0, 3.0])

dt.validate(a, float)
```

```
import numpy as np
import datatest as dt

a = np.array([1.0, 2.0, frozenset({3})])
```

(continues on next page)

(continued from previous page)

```
dt.validate(a, float)
```

```
Traceback (most recent call last):
  File "example.py", line 6, in <module>
    dt.validate(a, float)
datatest.ValidationError: does not satisfy `float` (1 difference): [
    Invalid(frozenset({3})),
]
```

Check the types for each row of elements within a two-dimentional array:

Passing

Failing

```
import numpy as np
import datatest as dt

a = np.array([(1.0, 12.25),
              (2.0, 33.75),
              (3.0, 101.5)])

dt.validate(a, (float, float))
```

```
import numpy as np
import datatest as dt

a = np.array([(1.0, 12.25),
              (2.0, 33.75),
              (frozenset({3}), 101.5)])

dt.validate(a, (float, float))
```

```
Traceback (most recent call last):
  File "example.py", line 8, in <module>
    dt.validate(a, (float, float))
datatest.ValidationError: does not satisfy `(float, float)` (1 difference): [
    Invalid((frozenset({3}), 101.5)),
]
```

Check the `dtype` of an array itself (not the elements it contains):

Passing

Failing

```
import numpy as np
import datatest as dt

a = np.array([(1.0, 12.25),
              (2.0, 33.75),
              (3.0, 101.5)])

dt.validate(a.dtype, np.dtype(float))
```

```
import numpy as np
import datatest as dt

a = np.array([(1.0, 12.25),
              (2.0, 33.75),
              (frozenset({3}), 101.5)])

dt.validate(a.dtype, np.dtype(float))
```

```
Traceback (most recent call last):
  File "example.py", line 8, in <module>
    dt.validate(a.dtype, np.dtype(float))
datatest.ValidationError: does not satisfy `dtype('float64')` (1 difference): [
    Invalid(dtype('O')),
]
```

Structured Arrays

If you can define your structured array directly, there's little need to validate the types it contains (unless it's an "object" dtype that could countain multiple types). But you may want to check the types in a structured array if it was constructed indirectly or was passed in from another source.

Check the types for each row of elements within a two-dimentional structured array:

Passing

Failing

```
import numpy as np
import datatest as dt

a = np.array([(1, 'x'),
              (2, 'y'),
              (3, 'z')],
             dtype='int, object')

dt.validate(a, (int, str))
```

```
import numpy as np
import datatest as dt

a = np.array([(1, 'x'),
              (2, 'y'),
              (3, 4.0)],
             dtype='int, object')

dt.validate(a, (int, str))
```

```
Traceback (most recent call last):
  File "example.py", line 9, in <module>
    dt.validate(a, (int, str))
datatest.ValidationError: does not satisfy `(int, str)` (1 difference): [
    Invalid((3, 4.0)),
]
```

You can also validate types with greater precision using NumPy's very specific dtypes (`np.uint32`, `np.float64`, etc.). or you can use NumPy's broader, generic types, like `np.character`, `np.integer`, `np.floating`, etc.:

Passing

Failing

```
import numpy as np
import datatest as dt

a = np.array([(1, 12.25),
              (2, 33.75),
              (3, 101.5)],
             dtype='int32, float32')

dt.validate(a, (np.integer, np.floating))
```

```
import numpy as np
import datatest as dt

a = np.array([(1, 12.25),
              (2, 33.75),
              (3, 101.5)],
             dtype='int32, object')

dt.validate(a, (np.integer, np.floating))
```

Since the "object" dtype was used for the second column of elements, the original type was unchanged. And although they are `float` objects, they aren't *NumPy* floating point objects. Since this is the case, all of the rows fail validation:

```
Traceback (most recent call last):
  File "example.py", line 9, in <module>
    dt.validate(a, (np.integer, np.floating))
datatest.ValidationError: does not satisfy `(integer, floating)` (3 differences): [
    Invalid((1, 12.25)),
    Invalid((2, 33.75)),
    Invalid((3, 101.5)),
]
```

Check the `dtype` values of a structured array itself (not the elements it contains):

Passing

Failing

```
import numpy as np
import datatest as dt

a = np.array([(1, 'x'),
              (2, 'y'),
              (3, 'z')],
             dtype='int, object')

data = [a.dtype[x] for x in a.dtype.names]
requirement = [np.dtype(int), np.dtype(object)]
dt.validate(data, requirement)
```

```
import numpy as np
import datatest as dt

a = np.array([(1, 'x'),
              (2, 'y'),
              (3, 'z')],
             dtype='int, str')

data = [a.dtype[x] for x in a.dtype.names]
requirement = [np.dtype(int), np.dtype(object)]
dt.validate(data, requirement)
```

```
Traceback (most recent call last):
  File "example.py", line 11, in <module>
    dt.validate(data, requirement)
datatest.ValidationError: does not match required sequence (1 difference): [
    Invalid(dtype('<U'), expected=dtype('O')),
]
```

1.2.7 How to Validate Date and Time Strings

To validate date and time formats, we can define a helper function that uses strftime codes to check for matching strings.

In the following example, we use the code `%Y-%m-%d` to check for dates that match the pattern YYYY-MM-DD:

```
from datetime import datetime
from datatest import validate


def strftime_format(format):
    def func(value):
        try:
            datetime.strptime(value, format)
        except ValueError:
            return False
        return True
    func.__doc__ = f'should use date format {format}'
    return func


data = ['2020-02-29', '03-17-2021', '2021-02-29', '2021-04-01']
validate(data, strftime_format('%Y-%m-%d'))
```

Date strings that don't match the required format are flagged as *Invalid*:

```
Traceback (most recent call last):
  File "example.py", line 17, in <module>
    validate(data, strftime_format('%Y-%m-%d'))
datatest.ValidationError: should use date format %Y-%m-%d (2 differences): [
    Invalid('03-17-2021'),
    Invalid('2021-02-29'),
]
```

Above, the date `03-17-2021` is invalid because it's not well-formed and `2021-02-29` is invalid because 2021 is not a leap-year so the last day of February is the 28th—there is no February 29th in that calendar year.

Strftime Codes for Common Formats

You can use the following **format codes** with the function defined earlier to validate many common date and time formats (e.g., `strftime_format('%d %B %Y')`):

format codes	description	example
`%Y-%m-%d`	YYYY-MM-DD	2021-03-17
`%m/%d/%Y`	MM/DD/YYYY	3/17/2021
`%d/%m/%Y`	DD/MM/YYYY	17/03/2021
`%d.%m.%Y`	DD.MM.YYYY	17.03.2021
`%d %B %Y`	DD Month YYYY	17 March 2021
`%b %d, %Y`	Mnth DD, YYYY	Mar 17, 2021
`%a %b %d %H:%M:%S %Y`	WkDay Mnth DD H:M:S YYYY	Wed Mar 17 19:42:50 2021
`%I:%M %p`	12-hour time	7:42 PM[1]
`%H:%M:%S`	24-hour time with seconds	19:42:50

In Python's `datetime` module, see strftime() and strptime() Format Codes for all supported codes.

1.2.8 How to Validate Date and Time Objects

Equality Validation

You can compare `date` and `datetime` objects just as you would any other quantitative value:

```
from datetime import date, datetime
from datatest import validate

validate(date(2020, 12, 25), date(2020, 12, 25))

validate(datetime(2020, 12, 25, 9, 30), datetime(2020, 12, 25, 9, 30))
```

Compare mappings of date objects:

```
from datetime import date
from datatest import validate

data = {
    'A': date(2020, 12, 24),
    'B': date(2020, 12, 25),
    'C': date(2020, 12, 26),
}

requirement = {
    'A': date(2020, 12, 24),
    'B': date(2020, 12, 25),
    'C': date(2020, 12, 26),
}

validate(data, requirement)
```

[1] The code `%p` expects the system locale's equivalent of AM or PM. For example, the locale `en_US` uses "AM" and "PM" while the locale `de_DE` uses "am" and "pm".

Interval Validation

We can use *validate.interval()* to check that date and datetime values are within a given range.

Check that dates fall within the month of December 2020:

```
from datetime import date
from datatest import validate

data = [
    date(2020, 12, 4),
    date(2020, 12, 11),
    date(2020, 12, 18),
    date(2020, 12, 25),
]

validate.interval(data, min=date(2020, 12, 1), max=date(2020, 12, 31))
```

Check that dates are one week old or newer:

```
from datetime import date, timedelta
from datatest import validate

data = {
    'A': date(2020, 12, 24),
    'B': date(2020, 12, 25),
    'C': date(2020, 12, 26),
}

one_week_ago = date.today() - timedelta(days=7)
validate.interval(data, min=one_week_ago, msg='one week old or newer')
```

Failures and Acceptances

When validation fails, a Deviation is generated containing a `timedelta` that represents the difference between two dates or times. You can accept time differences with *accepted.tolerance()*.

Failed Equality

Failure

Acceptance

```
from datetime import date
from datatest import validate

validate(date(2020, 12, 27), date(2020, 12, 25))
```

```
Traceback (most recent call last):
  File "example.py", line 4, in <module>
    validate(date(2020, 12, 27), date(2020, 12, 25))
datatest.ValidationError: does not satisfy `datetime.date(2020, 12, 25)` (1␣
↪difference): [
    Deviation(timedelta(days=+2), date(2020, 12, 25)),
]
```

```
from datetime import date, timedelta
from datatest import validate, accepted

with accepted.tolerance(timedelta(days=2)):  # accepts ±2 days
    validate(date(2020, 12, 27), date(2020, 12, 25))
```

Failed Interval

Failure

Acceptance

```
from datetime import date
from datatest import validate

data = [
    date(2020, 11, 26),
    date(2020, 12, 11),
    date(2020, 12, 25),
    date(2021, 1, 4),
]

validate.interval(data, min=date(2020, 12, 1), max=date(2020, 12, 31))
```

```
Traceback (most recent call last):
  File "example.py", line 11, in <module>
    validate.interval(data, min=date(2020, 12, 1), max=date(2020, 12, 31))
datatest.ValidationError: elements `x` do not satisfy `datetime.date(2020,
12, 1) <= x <= datetime.date(2020, 12, 31)` (2 differences): [
    Deviation(timedelta(days=-5), date(2020, 12, 1)),
    Deviation(timedelta(days=+4), date(2020, 12, 31)),
]
```

```
from datetime import date, timedelta
from datatest import validate, accepted

data = [
    date(2020, 11, 26),
    date(2020, 12, 11),
    date(2020, 12, 25),
    date(2021, 1, 4),
]

with accepted.tolerance(timedelta(days=5)):  # accepts ±5 days
    validate.interval(data, min=date(2020, 12, 1), max=date(2020, 12, 31))
```

Failed Mapping Equalities

Failure

Acceptance

```
from datetime import datetime
from datatest import validate

remote_log = {
    'job165': datetime(2020, 11, 27, hour=3, minute=29, second=55),
    'job382': datetime(2020, 11, 27, hour=3, minute=36, second=33),
    'job592': datetime(2020, 11, 27, hour=3, minute=42, second=14),
    'job720': datetime(2020, 11, 27, hour=3, minute=50, second=7),
    'job826': datetime(2020, 11, 27, hour=3, minute=54, second=12),
}

master_log = {
    'job165': datetime(2020, 11, 27, hour=3, minute=28, second=1),
    'job382': datetime(2020, 11, 27, hour=3, minute=36, second=51),
    'job592': datetime(2020, 11, 27, hour=3, minute=40, second=18),
    'job720': datetime(2020, 11, 27, hour=3, minute=49, second=39),
    'job826': datetime(2020, 11, 27, hour=3, minute=55, second=20),
}

validate(remote_log, master_log)
```

```
Traceback (most recent call last):
  File "example.py", line 20, in <module>
    validate(remote_log, master_log)
datatest.ValidationError: does not satisfy mapping requirements (5 differences): {
    'job165': Deviation(timedelta(seconds=+114), datetime(2020, 11, 27, 3, 28, 1)),
    'job382': Deviation(timedelta(seconds=-18), datetime(2020, 11, 27, 3, 36, 51)),
    'job592': Deviation(timedelta(seconds=+116), datetime(2020, 11, 27, 3, 40, 18)),
    'job720': Deviation(timedelta(seconds=+28), datetime(2020, 11, 27, 3, 49, 39)),
    'job826': Deviation(timedelta(seconds=-68), datetime(2020, 11, 27, 3, 55, 20)),
}
```

```
from datetime import datetime, timedelta
from datatest import validate, accepted

remote_log = {
    'job165': datetime(2020, 11, 27, hour=3, minute=29, second=55),
    'job382': datetime(2020, 11, 27, hour=3, minute=36, second=33),
    'job592': datetime(2020, 11, 27, hour=3, minute=42, second=14),
    'job720': datetime(2020, 11, 27, hour=3, minute=50, second=7),
    'job826': datetime(2020, 11, 27, hour=3, minute=54, second=12),
}

master_log = {
    'job165': datetime(2020, 11, 27, hour=3, minute=28, second=1),
    'job382': datetime(2020, 11, 27, hour=3, minute=36, second=51),
    'job592': datetime(2020, 11, 27, hour=3, minute=40, second=18),
    'job720': datetime(2020, 11, 27, hour=3, minute=49, second=39),
    'job826': datetime(2020, 11, 27, hour=3, minute=55, second=20),
}

```

(continues on next page)

(continued from previous page)

```
with accepted.tolerance(timedelta(seconds=120)):  # accepts ±120 seconds
    validate(remote_log, master_log)
```

Note: The *Deviation*'s repr (its printable representation) does some tricks with `datetime` objects to improve readability and help users understand their data errors. Compare the representations of the following datetime objects against their representations when included inside a Deviation:

```
>>> from datetime import timedelta, date
>>> from datatest import Deviation

>>> timedelta(days=2)
datetime.timedelta(days=2)

>>> date(2020, 12, 25)
datetime.date(2020, 12, 25)

>>> Deviation(timedelta(days=2), date(2020, 12, 25))
Deviation(timedelta(days=+2), date(2020, 12, 25))
```

And below, we see a negative-value timedelta with a particularly surprising native repr. The Deviation repr modifies this to be more readable:

```
>>> from datetime import timedelta, datetime
>>> from datatest import Deviation

>>> timedelta(seconds=-3)
datetime.timedelta(days=-1, seconds=86397)

>>> datetime(2020, 12, 25, 9, 30)
datetime.datetime(2020, 12, 25, 9, 30)

>>> Deviation(timedelta(seconds=-3), datetime(2020, 12, 25, 9, 30))
Deviation(timedelta(seconds=-3), datetime(2020, 12, 25, 9, 30))
```

While the timedelta reprs do differ, they *are* legitimate constructors for creating objects of equal value:

```
>>> from datetime import timedelta

>>> timedelta(days=+2) == timedelta(days=2)
True

>>> timedelta(seconds=-3) == timedelta(days=-1, seconds=86397)
True
```

1.2.9 How to Validate File Names

Sometimes you need to make sure that files are well organized and conform to some specific naming scheme. To validate the files names in a directory, simply pass the names themselves as the *data* argument when calling `validate()`. You then control the method of validation with the *requirement* you provide.

---

pathlib Basics

While there are multiple ways to get file names stored on disk, examples on this page use the Standard Library's `pathlib` module. If you're not familiar with pathlib, please review some basics examples before continuing:

These examples assume the following directory structure:

```
├── file1.csv
├── file2.csv
├── file3.xlsx
└── directory1/
    ├── file4.csv
    └── file5.xlsx
```

Import the `Path` class:

```
>>> from pathlib import Path
```

Get a list of file and directory names from the current directory:

```
>>> [str(p) for p in Path('.').iterdir()]
['file1.csv', 'file2.csv', 'file3.xlsx', 'directory1']
```

Filter the results to just files, no directories, using an `if` clause:

```
>>> [str(p) for p in Path('.').iterdir() if p.is_file()]
['file1.csv', 'file2.csv', 'file3.xlsx']
```

Get a list of path names ending in ".csv" from the current directory using `glob`-style pattern matching:

```
>>> [str(p) for p in Path('.').glob('*.csv')]
['file1.csv', 'file2.csv']
```

Get a list of CSV paths from the current directory and all subdirectories:

```
>>> [str(p) for p in Path('.').rglob('*.csv')]  # <- Using "recursive glob".
['file1.csv', 'file2.csv', 'directory1/file4.csv']
```

Get a list of CSV names from the current directory and all subdirectories using `p.name` instead of `str(p)` (excludes directory name):

```
>>> [p.name for p in Path('.').rglob('*.csv')]  # <- p.name excludes directory
['file1.csv', 'file2.csv', 'file4.csv']
```

Get a list of file and directory paths from the parent directory using the special name `..`:

```
>>> [str(p) for p in Path('..').iterdir()]
[ <parent directory names here> ]
```

Lowercase

Check that file names are lowercase:

```
from pathlib import Path
from datatest import validate, working_directory


with working_directory(__file__):
    file_names = (str(p) for p in Path('.').iterdir() if p.is_file())

def islower(x):
    return x.islower()

validate(file_names, islower, msg='should be lowercase')
```

Lowercase Without Spaces

Check that file names are lowercase and don't use spaces:

```
from pathlib import Path
from datatest import validate, working_directory


with working_directory(__file__):
    file_names = (str(p) for p in Path('.').iterdir() if p.is_file())

msg = 'Should be lowercase with no spaces.',
validate.regex(file_names, r'[a-z0-9_.\-]+', msg=msg)
```

Not Too Long

Check that the file names aren't too long (25 characters or less):

```
from pathlib import Path
from datatest import validate, working_directory


with working_directory(__file__):
    file_names = (str(p) for p in Path('.').iterdir() if p.is_file())

def not_too_long(x):
    """Path names should be 25 characters or less."""
    return len(x) <= 25

validate(file_names, not_too_long)
```

Check for CSV Type

Check that files are CSVs files:

```
from pathlib import Path
from datatest import validate, working_directory


with working_directory(__file__):
    file_names = (str(p) for p in Path('.').iterdir() if p.is_file())

def is_csv(x):
    return x.lower().endswith('.csv')

validate(file_names, is_csv, msg='should be CSV file')
```

Multiple Files Types

Check that files are CSV, Excel, or DBF file types:

```
from pathlib import Path
from datatest import validate, working_directory


with working_directory(__file__):
    file_names = (str(p) for p in Path('.').iterdir() if p.is_file())

def tabular_formats(x):  # <- Helper function.
    """Should be CSV, Excel, or DBF files."""
    suffix = Path(x).suffix
    return suffix.lower() in {'.csv', '.xlsx', '.xls', '.dbf'}

validate(file_names, tabular_formats)
```

Specific Files Exist

Using *validate.superset()*, check that the list of file names includes a given set of required files:

```
from pathlib import Path
from datatest import validate, working_directory


with working_directory(__file__):
    file_names = (str(p) for p in Path('.').iterdir() if p.is_file())

validate.superset(file_names, {'readme.txt', 'license.txt', 'config.ini'})
```

Includes Date

Check that file names begin with a date in YYYYMMDD format (e.g., 20201103_data.csv):

```
from pathlib import Path
from datatest import validate, working_directory


with working_directory(__file__):
    file_names = (p.name for p in Path('.').iterdir() if p.is_file())

msg = 'Should have date prefix followed by an underscore (YYYYMMDD_).'
validate.regex(file_names, r'^\d{4}\d{2}\d{2}_.+', msg=msg)
```

You can change the regex pattern to match another naming scheme of your choice. See the following examples for ideas:

description	regex pattern	example
date prefix	`^\d{4}-\d{2}-\d{2}_.+`	2020-11-03_data.csv
date prefix (no hyphen)	`^\d{4}\d{2}\d{2}_.+`	20201103_data.csv
date suffix	`.+_\d{4}-\d{2}-\d{2}.+$`	data_2020-11-03.csv
date suffix (no hyphen)	`.+_\d{4}\d{2}\d{2}.+$`	data_20201103.csv

See Also

- *How to Test File Properties*

1.2.10 How to Test File Properties

In some cases, you might need to check the properties of several files at once. This can be accomplished by loading the properties into a DataFrame or other object using a fixture.

Example

Pandas

dict-of-lists

```
import datetime
import os
import pathlib
import pytest
import pandas as pd
import datatest as dt


def get_properties(file_path):
    """Accepts a pathlib.Path and returns a dict of file properties."""
    stats = file_path.stat()

    size_in_mb = stats.st_size / 1024 / 1024  # Convert bytes to megabytes.

    return {
```

(continues on next page)

(continued from previous page)

```
        'path': str(file_path),
        'name': file_path.name,
        'modified_date': datetime.date.fromtimestamp(stats.st_mtime),
        'size': round(size_in_mb, 2),
        'readable': os.access(file_path, os.R_OK),
        'writable': os.access(file_path, os.W_OK),
    }


@pytest.fixture(scope='session')
@dt.working_directory(__file__)
def df():
    directory = '.'  # Current directory.
    pattern = '*.csv'  # Matches CSV files.
    paths = (p for p in pathlib.Path(directory).glob(pattern) if p.is_file())
    dict_records = (get_properties(p) for p in paths)
    df = pd.DataFrame.from_records(dict_records)
    df = df.set_index(['path'])
    return df


def test_filename(df):
    def is_lower_case(x):  # <- Helper function.
        return x.islower()

    msg = 'Must be lowercase.'
    dt.validate(df['name'], is_lower_case, msg=msg)


def test_freshness(df):
    one_week_ago = datetime.date.today() - datetime.timedelta(days=7)
    msg = 'Must be no older than one week.'
    dt.validate.interval(df['modified_date'], min=one_week_ago, msg=msg)


def test_filesize(df):
    msg = 'Must be 1 MB or less in size.'
    dt.validate.interval(df['size'], max=1.0, msg=msg)


def test_permissions(df):
    msg = 'Must have read and write permissions.'
    dt.validate(df[['readable', 'writable']], (True, True), msg=msg)


if __name__ == '__main__':
    import sys
    sys.exit(pytest.main(sys.argv))
```

```
import datetime
import os
import pathlib
import pytest
import collections
from datatest import validate, working_directory

```

(continues on next page)

(continued from previous page)

```

def get_properties(file_path):
    """Accepts a pathlib.Path and returns a dict of file properties."""
    stats = file_path.stat()

    size_in_mb = stats.st_size / 1024 / 1024  # Convert bytes to megabytes.

    return {
        'path': str(file_path),
        'name': file_path.name,
        'modified_date': datetime.date.fromtimestamp(stats.st_mtime),
        'size': round(size_in_mb, 2),
        'readable': os.access(file_path, os.R_OK),
        'writable': os.access(file_path, os.W_OK),
    }


@pytest.fixture(scope='session')
@working_directory(__file__)
def files_info():
    directory = '.'  # Current directory.
    pattern = '*.csv'  # Matches CSV files.
    paths = (p for p in pathlib.Path(directory).glob(pattern) if p.is_file())
    dict_of_lists = collections.defaultdict(list)
    for path in paths:
        properties_dict = get_properties(path)
        for k, v in properties_dict.items():
            dict_of_lists[k].append(v)
    return dict_of_lists


def test_filename(files_info):
    def is_lower_case(x):  # <- Helper function.
        return x.islower()

    msg = 'Must be lowercase.'
    validate(files_info['name'], is_lower_case, msg=msg)


def test_freshness(files_info):
    data = dict(zip(files_info['path'], files_info['modified_date']))

    one_week_ago = datetime.date.today() - datetime.timedelta(days=7)

    msg = 'Must be no older than one week.'
    validate.interval(data, min=one_week_ago, msg=msg)


def test_filesize(files_info):
    data = dict(zip(files_info['path'], files_info['size']))

    msg = 'Must be 1 MB or less in size.'
    validate.interval(data, max=1.0, msg=msg)


def test_permissions(files_info):
    values = zip(files_info['readable'], files_info['writable'])
```

(continues on next page)

(continued from previous page)

```
    data = dict(zip(files_info['path'], values))

    msg = 'Must have read and write permissions.'
    validate(data, (True, True), msg=msg)


if __name__ == '__main__':
    import sys
    sys.exit(pytest.main(sys.argv))
```

Other Properties

To check other file properties, you can modify or add to the `get_properties()` function.

Below, we count the number of lines in each file and add a `line_count` to the dictionary of properties:

```
import datetime
import os

...
```

```
...

def get_properties(file_path):
    """Accepts a pathlib.Path and returns a dict of file properties."""
    stats = file_path.stat()

    size_in_mb = stats.st_size / 1024 / 1024  # Convert bytes to megabytes.

    with open(file_path) as fh:
        line_count = len(fh.readlines())

    return {
        'path': str(file_path),
        'name': file_path.name,
        'modified_date': datetime.date.fromtimestamp(stats.st_mtime),
        'size': round(size_in_mb, 2),
        'readable': os.access(file_path, os.R_OK),
        'writable': os.access(file_path, os.W_OK),
        'line_count': line_count,
    }

...
```

See Also

- *How to Validate File Names*
- *How to Validate Date and Time Objects*

1.2.11 How to Avoid Excel Automatic Formatting

When MS Excel opens CSV files (and many other tabular formats), its default behavior will reformat certain values as dates, strip leading zeros, convert long numbers into scientific notation, and more. There are many cases where these kinds of changes actually corrupt your data.

It is possible to control Excel's formatting behavior using its *Text Import Wizard*. But as long as other users can open and re-save your CSV files, there may be no good way to guarantee that someone else won't inadvertently corrupt your data with Excel's default auto-format behavior. In a situation like this, you can mitigate problems by avoiding values that Excel likes to auto-format.

Using the *Predicate* object below, you can check that values are "Excel safe" and receive a list of differences when values are vulnerable to inadvertent auto-formatting:

```
import re
from datatest import validate, Predicate


# Predicate to check that elements are not subject
# to Excel auto-formatting.
excel_safe = ~Predicate(re.compile(r'''^(
    # Date format character combinations.
    \d{1,2}-(?:\d{1,2}|\d{4})
    | (Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)[ \-]\d{1,2}
    | [01]?[0-9]-(Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)

    # Time conversions.
    | [01]?[0-9][ ]?(AM?|PM?)     # Twelve-hour clock.
    | \d?\d[ ]*:                  # HH (hours).
    | \d?\d[ ]*(:[ ]*\d\d?){1,2}  # HH:MM and HH:MM:SS

    # Numeric conversions.
    | 0\d+\.?\d*        # Number with leading zeros.
    | \d*\.\d*0         # Decimal point with trailing zeros.
    | \d*\.             # Trailing decimal point.
    | \d.?\d*E[+-]?\d+  # Scientific notation.
    | \d{16,}           # Numbers of 16+ digits get approximated.

    # Whitespace normalization.
    | \s.*              # Leading whitespace.
    | .*\s              # Trailing whitespace.
    | .*\s\s.*          # Irregular whitespace (new in Office 365).

    # Other conversions
    | =.+               # Spreadsheet formula.

)$''', re.VERBOSE | re.IGNORECASE), name='excel_safe')


data = [
    'AOX-18',
    'APR-23',
    'DBB-01',
    'DEC-20',
    'DNZ-33',
    'DVH-50',
]
validate(data, excel_safe)
```

In the example above, we use `excel_safe` as our *requirement*. The validation fails because our *data* contains two codes that Excel would auto-convert into date types:

```
ValidationError: does not satisfy excel_safe() (2 differences): [
    Invalid('APR-23'),
    Invalid('DEC-20'),
]
```

Fixing the Data

To address the failure, we need to change the values in *data* so they are no longer subject to Excel's auto-formatting behavior. There are a few ways to do this.

We can prefix the failing values with apostrophes (`'APR-23` and `'DEC-20`). This causes Excel to treat them as text instead of dates or numbers:

```
...

data = [
    "AOX-18",
    "'APR-23",
    "DBB-01",
    "'DEC-20",
    "DNZ-33",
    "DVH-50",
]
validate(data, excel_safe)
```

Another approach would be to change the formatting for the all of the values. Below, the hyphens in *data* have been replaced with underscores (_):

```
...

data = [
    'AOX_18',
    'APR_23',
    'DBB_01',
    'DEC_20',
    'DNZ_33',
    'DVH_50',
]
validate(data, excel_safe)
```

After making the needed changes, the validation will now pass without error.

Caution: The `excel_safe` predicate implements a blacklist approach to detect values that Excel will automatically convert. It is not guaranteed to catch everything and future versions of Excel could introduce new behaviors. If you discover auto-formatted values that are not handled by this helper function (or if you have an idea regarding a workable whitelist approach), please file an issue and we will try to improve it.

1.2.12 How to Validate Mailing Addresses (US)

CASS Certified Verification

Unfortunately, the only "real" way to validate addresses is to use a verification service or program. Simple validation checks cannot guarantee that an address is correct or deliverable. In the United States, proper address verification requires the use of CASS certified software. Several online services offer address verification but to use one you must write code to interact with that service's API. Implementing such a solution is beyond the scope of this document.

Heuristic Evaluation

Sometimes the benefits of comprehensive address verification are not enough to justify the work required to interface with a third-party service or the possible cost of a subscription fee. Simple checks for well-formedness and set membership can catch many obvious errors and omissions. This weaker form of verification can be useful in many situations.

Load Data as Text

To start, we will load our example addresses into a pandas `DataFrame`. It's important to specify `dtype=str` to prevent pandas' type inference from loading certain columns using a numeric dtype. In some data sets, ZIP Codes could be misidentified as numeric data and loading them into numeric column would strip any leading zeros—corrupting the data you're testing:

```
import pandas as pd
from datatest import validate

df = pd.read_csv('addresses.csv', dtype=str)

...
```

Our address data will look something like the following:

street	city	state	zipcode
1600 Pennsylvania Avenue NW	Washington	DC	20500
30 Rockefeller Plaza	New York	NY	10112
350 Fifth Avenue, 34th Floor	New York	NY	10118-3299
1060 W Addison St	Chicago	IL	60613
15 Central Park W Apt 7P	New York	NY	10023-7711
11 Wall St	New York	NY	10005
2400 Fulton St	San Francisco	CA	94118-4107
351 Farmington Ave	Hartford	CT	06105-6400

Street Address

Street addresses are difficult to validate with a simple check. The US Postal Service publishes addressing standards designed to account for a majority of address styles (see delivery address line). But these standards do not account for all situations.

You could build a function to check that "street" values contain commonly used suffixes, but such a test could give misleading results when checking hyphenated address ranges, grid-style addresses, and rural routes. If you are not using a third-party verification service, it may be best to simply check that the field is not empty.

The example below uses a regular expression, `\w+`, to match one or more letters or numbers:

```
...

validate.regex(df['street'], r'\w+')
```

City Name

The US Postal Service sells a regularly updated *City State Product* file. For paying customers who purchase the USPS file or for users of third-party services, "city" values can be matched against a controlled vocabulary of approved city names. As with street validation, when such resources are unavailable it's probably best to check that the field is not empty.

The example below uses a regular expression, `[A-Za-z]+`, to match one or more letters:

```
...

validate.regex(df['city'], r'[A-Za-z]+')
```

State Abbreviation

Unlike the previous fields, the set of possible state abbreviations is small and easy to check against. The set includes codes for all 50 states, the District of Columbia, US territories, associate states, and armed forces delivery codes.

In this example, we use *`validate.subset()`* to check that the values in the "state" column are members of the `state_codes` set:

```
...

state_codes = {
    'AL', 'AK', 'AZ', 'AR', 'CA', 'CO', 'CT', 'DE', 'FL', 'GA',
    'HI', 'ID', 'IL', 'IN', 'IA', 'KS', 'KY', 'LA', 'ME', 'MD',
    'MA', 'MI', 'MN', 'MS', 'MO', 'MT', 'NE', 'NV', 'NH', 'NJ',
    'NM', 'NY', 'NC', 'ND', 'OH', 'OK', 'OR', 'PA', 'RI', 'SC',
    'SD', 'TN', 'TX', 'UT', 'VT', 'VA', 'WA', 'WV', 'WI', 'WY',
    'DC', 'AS', 'GU', 'MP', 'PR', 'VI', 'FM', 'MH', 'PW',
    'AA', 'AE', 'AP',
}
validate.subset(df['state'], state_codes)
```

ZIP Code

The set of valid ZIP Codes is very large but they can be easily checked for well-formedness. Basic ZIP Codes are five digits and extended ZIP+4 Codes are nine digits (e.g., 20500 and 20500-0005).

This example uses a regex, `^\d{5}(-\d{4})?$`, to match the two possible formats:

```
...

validate.regex(df['zipcode'], r'^\d{5}(-\d{4})?$')
```

State and ZIP Code Consistency

The first digit of a ZIP Code is associated with a specific region of the country (a group of states). For example, ZIP Codes begging with "4" only occur in Indiana, Kentucky, Michigan, and Ohio. We can use these regional associations as a sanity check to make sure that our "state" and "zipcode" values are plausible and consistent.

The following example defines a helper function, `state_zip_consistency()`, to check the first digit of a ZIP Code against a set of associated state codes:

```
...

def state_zip_consistency(state_zipcode):
    """ZIP Code should be consistent with state."""
    lookup = {
        '0': {'CT', 'MA', 'ME', 'NH', 'NJ', 'NY', 'PR', 'RI', 'VT', 'VI', 'AE'},
        '1': {'DE', 'NY', 'PA'},
        '2': {'DC', 'MD', 'NC', 'SC', 'VA', 'WV'},
        '3': {'AL', 'FL', 'GA', 'MS', 'TN', 'AA'},
        '4': {'IN', 'KY', 'MI', 'OH'},
        '5': {'IA', 'MN', 'MT', 'ND', 'SD', 'WI'},
        '6': {'IL', 'KS', 'MO', 'NE'},
        '7': {'AR', 'LA', 'OK', 'TX'},
        '8': {'AZ', 'CO', 'ID', 'NM', 'NV', 'UT', 'WY'},
        '9': {'AK', 'AS', 'CA', 'GU', 'HI', 'MH', 'FM', 'MP', 'OR', 'PW', 'WA', 'AP'},
    }
    state, zipcode = state_zipcode
    first_digit = zipcode[0]
    return state in lookup[first_digit]

validate(df[['state', 'zipcode']], state_zip_consistency)
```

This check works well to detect data processing errors that might mis-align or otherwise damage "state" and "zipcode" values. But it cannot detect if ZIP Codes are assigned to the wrong states within in the same region—for example, it wouldn't be able to determine if an Indiana ZIP Code was used on an Kentucky address (since the ZIP Codes in both of these states begin with "4").

1.2.13 How to Validate Fuzzy Matches

When comparing strings of text, it can sometimes be useful to check that values are similar instead of asserting that they are exactly the same. Datatest provides options for *approximate string matching* (also called "fuzzy matching").

When checking mappings or sequences of values, you can accept approximate matches with the *accepted.fuzzy()* acceptance:

Using Acceptance

No Acceptance

```
from datatest import validate, accepted

linked_record = {
    'id165': 'Saint Louis',
    'id382': 'Raliegh',
    'id592': 'Austin',
    'id720': 'Cincinatti',
    'id826': 'Philadelphia',
}

master_record = {
    'id165': 'St. Louis',
    'id382': 'Raleigh',
    'id592': 'Austin',
    'id720': 'Cincinnati',
    'id826': 'Philadelphia',
}

with accepted.fuzzy(cutoff=0.6):
    validate(linked_record, master_record)
```

```
from datatest import validate

linked_record = {
    'id165': 'Saint Louis',
    'id382': 'Raliegh',
    'id592': 'Austin',
    'id720': 'Cincinatti',
    'id826': 'Philadelphia',
}

master_record = {
    'id165': 'St. Louis',
    'id382': 'Raleigh',
    'id592': 'Austin',
    'id720': 'Cincinnati',
    'id826': 'Philadelphia',
}

validate(linked_record, master_record)
```

```
Traceback (most recent call last):
  File "example.py", line 19, in <module>
    validate(linked_record, master_record)
datatest.ValidationError: does not satisfy mapping requirements (3 differences): {
    'id165': Invalid('Saint Louis', expected='St. Louis'),
```

(continues on next page)

(continued from previous page)

```
    'id382': Invalid('Raliegh', expected='Raleigh'),
    'id720': Invalid('Cincinatti', expected='Cincinnati'),
}
```

If variation is an inherent, natural feature of the data and does not necessarily represent a defect, it may be appropriate to use *validate.fuzzy()* instead of the acceptance shown previously:

```
from datatest import validate

linked_record = {
    'id165': 'Saint Louis',
    'id382': 'Raliegh',
    'id592': 'Austin',
    'id720': 'Cincinatti',
    'id826': 'Philadelphia',
}

master_record = {
    'id165': 'St. Louis',
    'id382': 'Raleigh',
    'id592': 'Austin',
    'id720': 'Cincinnati',
    'id826': 'Philadelphia',
}

validate.fuzzy(linked_record, master_record, cutoff=0.6)
```

That said, it's probably more appropriate to use an acceptance for this specific example.

1.2.14 How to Deal With NaN Values

IEEE 754

While the behavior of NaN values can seem strange, it's actually the result of an intentionally designed specification. The behavior was standardized in IEEE 754 (IEEE Standard for Floating-Point Arithmetic), a technical standards document first published in 1985 and implemented by many popular programming languages (including Python).

When checking certain types of data, you may encounter NaN values. Working with NaNs can be frustrating because they don't always act as one might expect.

About NaN values:

- NaN is short for "Not a Number".
- NaN values represent undefined or unrepresentable results from certain mathematical operations.
- Mathematical operations involving a NaN will either return a NaN or raise an exception.
- Comparisons involving a NaN will return False.

Checking for NaN Values

To make sure data elements do not contain NaN values, you can use a helper function:

```
from math import isnan
from datatest import validate


data = [5, 6, float('nan')]

def not_nan(x):
    """Values should not be NaN."""
    return not isnan(x)

validate(data, not_nan)
```

You can also do this using an inverted *Predicate* match:

```
from math import isnan
from datatest import validate, Predicate


data = [5, 6, float('nan')]

requirement = ~Predicate(isnan)

validate(data, requirement)
```

Accepting NaN Differences

If validation fails and returns NaN differences, you can accept them as you would any other difference:

Using Acceptance

No Acceptance

```
from math import nan
from datatest import validate, accepted, Extra


data = [5, 6, float('nan')]
requirement = {5, 6}

with accepted(Extra(nan)):
    validate(data, requirement)
```

```
from math import nan
from datatest import validate


data = [5, 6, float('nan')]
requirement = {5, 6}

validate(data, requirement)
```

```
Traceback (most recent call last):
  File "example.py", line 8, in <module>
    validate(data, requirement)
datatest.ValidationError: does not satisfy set membership (1 difference): [
    Extra(nan),
]
```

Like other values, NaNs can also be accepted as part of a list, set, or mapping of differences:

Using Acceptance

No Acceptance

```
from math import nan
from datatest import validate, accepted, Missing, Extra


data = [5, 6, float('nan')]
requirement = {5, 6, 7}

with accepted([Missing(7), Extra(nan)]):
    validate(data, requirement)
```

```
from math import nan
from datatest import validate


data = [5, 6, float('nan')]
requirement = {5, 6, 7}

validate(data, requirement)
```

```
Traceback (most recent call last):
  File "example.py", line 8, in <module>
    validate(data, requirement)
datatest.ValidationError: does not satisfy set membership (2 differences): [
    Missing(7),
    Extra(nan),
]
```

Note: The `math.nan` value is new in Python 3.5. NaN values can also be created in any Python version using `float('nan')`.

Dropping NaNs Before Validation

Sometimes it's OK to ignore NaN values entirely. If this is appropriate in your circumstance, you can simply remove all NaN records and validate the remaining data.

Pandas Example

Non-Pandas Example

If you're using Pandas, you can call the `Series.dropna()` and `DataFrame.dropna()` methods to drop records that contain NaN values:

```
import pandas as pd
from datatest import validate


source = pd.Series([1, 1, 2, 2, float('nan')])

data = source.dropna()  # Drop NaN valued elements.
requirement = {1, 2}

validate(data, requirement)
```

In this example, we use the Standard Library's `math.isnan()` in a list comprehension to drop NaN values from our data:

```
from math import isnan
from datatest import validate


source = [1, 1, 2, 2, float('nan')]

data = [x for x in source if not isnan(x)]  # Keep values if not NaN.
requirement = {1, 2}

validate(data, requirement)
```

Requiring NaN Values

If necessary, it's possible to require that NaNs appear in your *data*. But putting NaN values directly into a *requirement* can be frought with problems and should usually be avoided. The most robust way to do this is by replacing NaN values with a special token and then requiring the token.

Below, we define a custom `NanToken` object and use it to replace actual NaN values.

Pandas Example

Non-Pandas Example

If you're using Pandas, you can call the `Series.fillna()` and `DataFrame.fillna()` methods to replace NaNs with a different value:

```
import pandas as pd
from datatest import validate


class NanToken(object):
    def __repr__(self):
        return self.__class__.__name__

NanToken = NanToken()


source = pd.Series([1, 1, 2, 2, float('nan')])

data = source.fillna(NanToken)  # Replace NaNs with NanToken.
requirement = {1, 2, NanToken}

validate(data, requirement)
```

In this example, we use a list comprehension and helper function to replace NaN values in our list of data elements:

```
from math import isnan
from datatest import validate


class NanToken(object):
    def __repr__(self):
        return self.__class__.__name__

NanToken = NanToken()


def replace_nan(x):  # <- Helper function.
    if isnan(x):
        return NanToken
    return x


source = [1, 1, 2, 2, float('nan')]

data = [replace_nan(x) for x in source]  # Replace NaNs with NanToken.
requirement = {1, 2, NanToken}

validate(data, requirement)
```

A Deeper Understanding

Equality: NaN ≠ NaN

NaN values don't compare as equal to anything—even themselves:

```
>>> x = float('nan')
>>> x == x
False
```

To check if a value is NaN, it's common for modules and packages to provide a function for this purpose (e.g., `math.isnan()`, `numpy.isnan()`, `pandas.isna()`, etc.):

```
>>> import math
>>> x = float('nan')
>>> math.isnan(x)
True
```

While NaN values cannot be compared directly, they *can* be compared as part of a difference object. In fact, difference comparisons treat all NaN values as equal—even when the underlying type is different:

```
>>> import decimal, math, numpy
>>> from datatest import Invalid

>>> Invalid(math.nan) == Invalid(float('nan'))
True
>>> Invalid(math.nan) == Invalid(complex('nan'))
True
>>> Invalid(math.nan) == Invalid(decimal.Decimal('nan'))
True
```

(continues on next page)

(continued from previous page)

```
>>> Invalid(math.nan) == Invalid(numpy.nan)
True
>>> Invalid(math.nan) == Invalid(numpy.float32('nan'))
True
>>> Invalid(math.nan) == Invalid(numpy.float64('nan'))
True
```

Identity: NaN is NaN, Except When it Isn't

Some packages provide a NaN constant that can be referenced in user code (e.g., `math.nan` and `numpy.nan`). While it may be tempting to use these constants to check for matching NaN values, this approach is not reliable in practice.

To optimize performance, Numpy and Pandas must strictly manage the memory layouts of the data they contain. When `numpy.nan` is inserted into an `ndarray` or `Series`, the value is coerced into a compatible `dtype` when necessary. When a NaN's type is coerced, a separate instance is created and the ability to match using the `is` operator no longer works as you might expect:

```
>>> import pandas as pd
>>> import numpy as np

>>> np.nan is np.nan
True

>>> s = pd.Series([10, 11, np.nan])
>>> s[2]
nan
>>> s[2] is np.nan
False
```

We can verify that the types are now different:

```
>>> type(np.nan)
float
>>> type(s[2])
float64
```

Generally speaking, it is not safe to assume that NaN is NaN. This means that—for reliable validation—it's best to remove NaN records entirely or replace them with some other value.

1.2.15 How to Validate Negative Matches

Sometimes you want to check that data is **not** equal to a specific value. There are a few different ways to perform this type of negative matching.

Helper Function

One obvious way to check for a negative match is to define a helper function that checks for `!=` to a given value:

```
from datatest import validate

data = [...]

def not_bar(x):
    return x != 'bar'

validate(data, not_bar)
```

Inverted Predicate

Datatest provides a *Predicate* class for handling different kinds of matching. You can invert a Predicate's behavior using the inversion operator, `~`:

```
from datatest import validate, Predicate

data = [...]
validate(data, ~Predicate('bar'))
```

Functional Style

If you are accustomed to programming in a functional style, you could perform a negative match using `functools.partial()` and `operator.ne()`:

```
from functools import partial
from operator import ne
from datatest import validate

data = [...]
validate(data, partial(ne, 'bar'))
```

1.2.16 How to Check for Outliers

There are many techniques for detecting outliers and no single approach can work for all cases. This page describes an often useful approach based on the interquartile/*Tukey fence* method for outlier detection.

Other common methods for outlier detection are sensitive to extreme values and can perform poorly when applied to skewed distributions. The Tukey fence method is resistant to extreme values and applies to both normal and slightly skewed distributions.

You can copy the following `RequiredOutliers` class to use in your own tests:

```
from statistics import median
from datatest import validate
from datatest.requirements import adapts_mapping
from datatest.requirements import RequiredInterval


@adapts_mapping
class RequiredOutliers(RequiredInterval):
    """Require that data does not contain outliers."""
    def __init__(self, values, multiplier=2.2):
        values = sorted(values)

        if len(values) >= 2:
            midpoint = int(round(len(values) / 2.0))
            q1 = median(values[:midpoint])
            q3 = median(values[midpoint:])
            iqr = q3 - q1
            lower = q1 - (iqr * multiplier)
            upper = q3 + (iqr * multiplier)
        elif values:
            lower = upper = values[0]
        else:
            lower = upper = 0

        super().__init__(lower, upper)

...
```

In "Exploratory Data Analysis" by John W. Tukey (1977), a multiplier of 1.5 was proposed for labeling outliers and 3.0 was proposed for labeling "far out" outliers. The default *multiplier* of `2.2` is based on "Fine-Tuning Some Resistant Rules for Outlier Labeling" by Hoaglin and Iglewicz (1987).

Note: The code above relies on `statistics.median()` which is new in Python 3.4. If you are running an older version of Python, you can use the following `median()` function instead:

```
def median(iterable):
    values = sorted(iterable)
    n = len(values)
    if n == 0:
        raise ValueError('no median for empty iterable')
    i = n // 2
    if n % 2 == 1:
        return values[i]
    return (values[i - 1] + values[i]) / 2.0
```

Example Usage

The following example uses the `RequiredOutliers` class defined earlier to check for outliers in a list of values:

```
...

data = [54, 44, 42, 46, 87, 48, 56, 52]  # <- 87 is an outlier
requirement = RequiredOutliers(data, multiplier=2.2)
validate(data, requirement)
```

```
ValidationError: elements `x` do not satisfy `23.0 <= x <= 77.0` (1 difference): [
    Deviation(+10.0, 77.0),
]
```

You can also use the class to validate mappings of values as well:

```
...

data = {
    'A': [54, 44, 42, 46, 87, 48, 56, 52],  # <- 87 is an outlier
    'B': [87, 83, 60, 85, 97, 91, 95, 93],  # <- 60 is an outlier
}
requirement = RequiredOutliers(data, multiplier=2.2)
validate(data, requirement)
```

```
ValidationError: does not satisfy mapping requirements (2 differences): {
    'A': [Deviation(+10.0, 77.0)],
    'B': [Deviation(-2.0, 62.0)],
}
```

Addressing Outliers

Once potential outliers have been identified, you need to decide how best to address them—there is no single best practice for determining what to do. Potential outliers provide a starting point for further investigation.

In some cases, these extreme values are legitimate and you will want to increase the *multiplier* or explicitly accept them (see *Acceptances*). In other cases, you may determine that your data contains values from two separate distributions and the test itself needs to be restructured. Or you could discover that the values represent data processing errors or other special cases and they should be excluded altogether.

How it Works

To use this approach most effectively, it helps to understand how it works. The following example explains the technique in detail using the same data as the first example given above:

$$54 \quad 44 \quad 42 \quad 46 \quad 87 \quad 48 \quad 56 \quad 52$$

1. Determine the first and third quartiles. First, sort the values in ascending order. Then, split the data in half at its median. The first quartile (**Q1**) is the median of the lower half and the third quartile (**Q3**) is the median of the

upper half:

$$\begin{array}{cccccccccc} & & \mathbf{Q1}\ (45) & & & & \mathbf{Q3}\ (55) & & & \\ & & \downarrow & & & & \downarrow & & & \\ 42 & 44 & 46 & 48 & & 52 & 54 & 56 & 87 & \\ & & & & \uparrow & & & & & \\ & & & & \mathit{median}\ (50) & & & & & \end{array}$$

2. Get the interquartile range (**IQR**) by taking the third quartile and subtracting the first quartile from it:

$$\mathbf{IQR} = \mathbf{Q3} - \mathbf{Q1}$$

$$10 = 55 - 45$$

3. Calculate a lower and upper limit using the values determined in the previous steps:

$$\text{lower limit} = \mathbf{Q1} - (\mathbf{IQR} \times \mathbf{multiplier})$$

$$23 = 45 - (10 \times 2.2)$$

$$\text{upper limit} = \mathbf{Q3} + (\mathbf{IQR} \times \mathbf{multiplier})$$

$$77 = 55 + (10 \times 2.2)$$

5. Check that values are within the determined limits. Any value less than the lower limit (23) or greater than the upper limit (77) is considered a potential outlier. In the given data, there is one potential outlier:

$$87$$

1.2.17 How to Validate Phone Numbers

To check that phone numbers are well-formed, you can use a regular expression.

USA and Canada

```
from datatest import validate

pattern = r'^\(\d{3}\)[ ]\d{3}-\d{4}$'

data = [
    '(914) 232-9901',
    '(914) 737-9938',
    '(213) 888-7636',
    '(202) 965-2900',
    '(858) 651-5050',
]

validate.regex(data, pattern, msg='must use phone number format')
```

For other common US and Canadian formats, you can use the regex patterns:

pattern	examples
`^\(\d{3}\)[ ]\d{3}-\d{4}$`	(914) 232-9901
`^\d{3}-\d{3}-\d{4}$`	914-232-9901
`^\+?1-\d{3}-\d{3}-\d{4}$`	1-914-232-9901
	+1-914-232-9901

India

```
import re
from datatest import validate


indian_phone_format = re.compile(r'''^
    (\+91[ ])?   # Optional international code.
    (\(0\))?     # Optional trunk prefix.
    # 10 digit codes with area & number splits.
    (
        \d{10}           # xxxxxxxxxx
        | \d{5}[ ]\d{5}  # xxxxx xxxxx
        | \d{4}[ ]\d{6}  # xxxx xxxxxx
        | \d{3}[ ]\d{7}  # xxx xxxxxxx
        | \d{2}[ ]\d{8}  # xx xxxxxxxx
    )
$''', re.VERBOSE)

data = [
    '+91 (0)99999 99999',
    '+91 99999 99999',
    '9999999999',
    '99999 99999',
    '9999 999999',
    '999 9999999',
    '99 99999999',
]

validate(data, indian_phone_format, msg='must use phone number format')
```

United Kingdom

```
import re
from datatest import validate


uk_phone_format = re.compile(r'''^(
    # 10 digit NSNs (leading zero doesn't count)
    \(01\d{2}[ ]\d{2}\d\)[ ]\d{2}[ ]\d{3} # (01xx xx) xx xxx
    | \(01\d{3}\)[ ]\d{3}[ ]\d{3}         # (01xxx) xxx xxx
    | \(01\d{2}\)[ ]\d{3}[ ]\d{4}         # (01xx) xxx xxxx
    | \(02\d\)[ ]\d{4}[ ]\d{4}            # (02x) xxxx xxxx
    | 0\d{3}[ ]\d{3}[ ]\d{4}              # 0xxx xxx xxxx
    | 0\d{2}[ ]\d{4}[ ]\d{4}              # 0xx xxxx xxxx
    | 07\d{3}[ ]\d{3}[ ]\d{3}             # 07xxx xxx xxx

    # 9 digit NSNs
    | \(0169[ ]77\)[ ]\d{4}               # (0169 77) xxxx
    | \(01\d{3}\)[ ]\d{2}[ ]\d{3}         # (01xxx) xx xxx
    | 0500[ ]\d{3}[ ]\d{3}                # 0500 xxx xxx
    | 0800[ ]\d{3}[ ]\d{3}                # 0800 xxx xxx
)$''', re.VERBOSE)

data = [
    '(01257) 421 282',
```

(continues on next page)

(continued from previous page)

```
    '(01736) 759 307',
    '(0169 77) 3452',
    '0116 319 5885',
    '0191 384 6777',
    '020 8399 0617',
]

validate(data, uk_phone_format, msg='must use phone number format')
```

1.2.18 How to Re-Order Acceptances

Individual acceptances can be combined together to create new acceptances with narrower or broader criteria (see *Composability*). When acceptances are combined, their criteria are applied in an order determined by their scope. Element-wise criteria are applied first, group-wise criteria are applied second, and whole-error criteria are applied last (see *Order of Operations*).

Implicit Ordering

In this first example, we have a combined acceptance made from a whole-error acceptance, *accepted.count()*, and a group-wise acceptance, *accepted([...])*:

```
with accepted.count(4) | accepted([Missing('A'), Missing('B')]):
    ...
```

Since the *Order of Operations* specifies that whole-error acceptances are applied *after* group-wise acceptances, the `accepted.count(4)` criteria is applied last even though it's defined first.

Explicit Ordering

If you want to control this order explicitly, you can use nested `with` statements to change the default behavior:

```
with accepted([Missing('A'), Missing('B')]):
    with accepted.count(4):
        ...
```

Using nested `with` statements, the inner-most block is applied first and outer blocks are applied in order until the outer-most block is applied last. In this example, the `accepted.count(4)` is applied first because it's declared in the inner-most block.

1.2.19 How to Validate Sequences

Index Position

To check for a specific sequence, you can pass a list[1] as the *requirement* argument:

[1] The validate() function will check *data* by index position when the *requirement* is any iterable object other than a set, mapping, tuple or string. See the *Sequence Validation* section of the *validate()* documentation for full details.

```
from datatest import validate

data = ['A', 'B', 'X', 'C', 'D']
requirement = ['A', 'B', 'C', 'D']  # <- a list
validate(data, requirement)
```

Elements in the *data* and *requirement* lists are compared by sequence position. The items at index position 0 are compared to each other, then items at index position 1 are compared to each other, and so on:

index	data	requirement	result
0	**A**	**A**	matches
1	**B**	**B**	matches
2	**X**	**C**	doesn't match
3	**C**	**D**	doesn't match
4	**D**	*no value*	doesn't match

In this example, there are three differences:

```
ValidationError: does not match required sequence (3 differences): [
    Invalid('X', expected='C'),
    Invalid('C', expected='D'),
    Extra('D'),
]
```

Using enumerate()

While the previous example works well for short lists, the error does not describe **where** in your sequence the differences occur. To get the index positions associated with any differences, you can `enumerate()` your *data* and *requirement* objects:

```
from datatest import validate

data = ['A', 'B', 'X', 'C', 'D']
requirement = ['A', 'B', 'C', 'D']
validate(enumerate(data), enumerate(requirement))
```

A required **enumerate object** is treated as a mapping. The keys for any differences will correspond to their index positions:

```
ValidationError: does not satisfy mapping requirements (3 differences): {
    2: Invalid('X', expected='C'),
    3: Invalid('C', expected='D'),
    4: Extra('D'),
}
```

Relative Order

When comparing elements by sequence position, one mis-alignment can create differences for all following elements. If this behavior is not desireable, you may want to check for *relative order* instead.

If you want to check the relative order of elements rather than their index positions, you can use `validate.order()`:

```
from datatest import validate

data = ['A', 'B', 'X', 'C', 'D']
requirement = ['A', 'B', 'C', 'D']
validate.order(data, requirement)
```

When checking for relative order, this method tries to align elements into contiguous matching subsequences. This reduces the number of non-matches:

index	data	requirement	result
0	**A**	**A**	matches
1	**B**	**B**	matches
2	**X**	*no value*	doesn't match
3	**C**	**C**	matches
4	**D**	**D**	matches

Differences are reported as two-tuples containing the index (in *data*) where the difference occurs and the non-matching value. In the earlier examples, we saw that validating by index position produced three differences. But in this example, validating the same sequences by relative order produces only one difference:

```
ValidationError: does not match required order (1 difference): [
     Extra((2, 'X')),
]
```

1.3 Reference

> *"A tool is best if it does the job required with a minimum of effort, with a minimum of complexity, and with a minimum of power."* —Peter Drucker[1]

1.3.1 Datatest Core API Reference

Validation

datatest.**validate**(*data*, *requirement*, *msg=None*)

Raise a `ValidationError` if *data* does not satisfy *requirement* or pass without error if data is valid.

This is a rich comparison function—the given *data* and *requirement* arguments can be mappings, iterables, or other objects (including objects from `pandas`, `numpy`, database cursors, and `squint`). An optional *msg* string can be provided to describe the validation.

Predicate Validation:

When *requirement* is a callable, tuple, string, or non-iterable object, it is used to construct a `Predicate` for testing elements in *data*:

[1] Drucker, Peter F. "Management: Tasks, Responsibilities, Practices", New York: Harper & Row, 1973. p. 224.

```
from datatest import validate

data = [2, 4, 6, 8]

def is_even(x):
    return x % 2 == 0

validate(data, is_even)  # <- callable used as predicate
```

If the predicate returns False, then an *Invalid* or *Deviation* difference is generated. If the predicate returns a difference object, that object is used in place of a generated difference (see *Differences*). When the predicate returns any other truthy value, an element is considered valid.

Set Validation:

When *requirement* is a set, the elements in *data* are checked for membership in the set:

```
from datatest import validate

data = ['a', 'a', 'b', 'b', 'c', 'c']

required_set = {'a', 'b', 'c'}

validate(data, required_set)  # <- tests for set membership
```

If the elements in *data* do not match the required set, then *Missing* and *Extra* differences are generated.

Sequence Validation:

When *requirement* is an iterable type other than a set, mapping, tuple or string, then *data* is validated by index position. Elements are checked for predicate matches against required objects of the same index position (both *data* and *requirement* should yield values in a predictable order):

```
from datatest import validate

data = ['A', 'B', 'C', ...]

sequence = ['A', 'B', 'C', ...]

validate(data, sequence)  # <- compare elements by position
```

For details on predicate matching, see *Predicate*.

Mapping Validation:

When *requirement* is a dictionary or other mapping, the values in *data* are checked against required objects of the same key (*data* must also be a mapping):

```
from datatest import validate

data = {'A': 1, 'B': 2, 'C': ...}

required_dict = {'A': 1, 'B': 2, 'C': ...}

validate(data, required_dict)  # <- compares values
```

If values do not satisfy the corresponding required object, then differences are generated according to each object type. If an object itself is a nested mapping, it is treated as a predicate object.

Requirement Object Validation:

When *requirement* is a subclass of `BaseRequirement`, then validation and difference generation are delegated to the *requirement* itself.

In addition to *`validate()`*'s default behavior, the following methods can be used to specify additional validation behaviors.

`predicate`(*data*, *requirement*, *msg=None*)

Use *requirement* to construct a *`Predicate`* and check elements in *data* for matches (see *predicate validation* for more details).

`regex`(*data*, *requirement*, *flags=0*, *msg=None*)

Require that string values match a given regular expression (also see Regular Expression Syntax):

```
from datatest import validate

data = ['46532', '43206', '60632']

validate.regex(data, r'^\d{5}$')
```

The example above is roughly equivalent to:

```
import re
from datatest import validate

data = ['46532', '43206', '60632']

validate(data, re.compile(r'^\d{5}$'))
```

`approx`(*data*, *requirement*, *places=None*, *msg=None*, *delta=None*)

Require that numeric values are approximately equal. The given *requirement* can be a single element or a mapping.

Values compare as equal if their difference rounded to the given number of decimal places (default 7) equals zero, or if the difference between values is less than or equal to a given *delta*:

```
from datatest import validate

data = {'A': 1.3125, 'B': 8.6875}

requirement = {'A': 1.31, 'B': 8.69}

validate.approx(data, requirement, places=2)
```

It is appropriate to use *`validate.approx()`* when checking for nominal values—where some deviation is considered an intrinsic feature of the data. But when deviations represent an undesired-but-acceptible variation, *`accepted.tolerance()`* would be more fitting.

`fuzzy`(*data*, *requirement*, *cutoff=0.6*, *msg=None*)

Require that strings match with a similarity greater than or equal to *cutoff* (default `0.6`).

Similarity measures are determined using `SequenceMatcher.ratio()` from the Standard Library's `difflib` module. The values range from `1.0` (exactly the same) to `0.0` (completely different).

```
from datatest import validate

data = {
    'MO': 'Saint Louis',
```

(continues on next page)

(continued from previous page)

```
    'NY': 'New York',  # <- does not meet cutoff
    'OH': 'Cincinatti',
}

requirement = {
    'MO': 'St. Louis',
    'NY': 'New York City',
    'OH': 'Cincinnati',
}

validate.fuzzy(data, requirement, cutoff=0.8)
```

interval (*data*, *min=None*, *max=None*, *msg=None*)

Require that values are within the defined interval:

```
from datatest import validate

data = [5, 10, 15, 20]  # <- 20 outside of interval

validate.interval(data, 5, 15)
```

Require that values are greater than or equal to *min* (omitting *max* creates a left-bounded interval):

```
from datatest import validate

data = [5, 10, 15, 20]

validate.interval(data, min=5)
```

Require that values are less than or equal to *max* (omitting *min* creates a right-bounded interval):

```
from datatest import validate

data = [5, 10, 15, 20]

validate.interval(data, max=20)
```

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

Check that the set of elements in *data* matches the set of elements in *requirement* (applies *set validation* using a *requirement* of any iterable type).

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

Check that the set of elements in *data* is a subset of the set of elements in *requirement* (i.e., that every element of *data* is also a member of *requirement*).

```
from datatest import validate

data = ['A', 'B', 'C']

requirement = {'A', 'B', 'C', 'D'}

validate.subset(data, requirement)
```

> **Attention:** Since version 0.10.0, the semantics of `subset()` have been inverted. To mitigate problems for users upgrading from 0.9.6, this method issues a warning.

To ignore this warning you can add the following lines to your code:

```
import warnings
warnings.filterwarnings('ignore', message='subset and superset warning')
```

And for pytest users, you can add the following to the beginning of a test script:

```
pytestmark = pytest.mark.filterwarnings('ignore:subset and superset warning
↪')
```

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

Check that the set of elements in *data* is a superset of the set of elements in *requirement* (i.e., that members of *data* include all elements of *requirement*).

```
from datatest import validate

data = ['A', 'B', 'C', 'D']

requirement = {'A', 'B', 'C'}

validate.superset(data, requirement)
```

Attention: Since version 0.10.0, the semantics of *superset()* have been inverted. To mitigate problems for users upgrading from 0.9.6, this method issues a warning.

To ignore this warning you can add the following lines to your code:

```
import warnings
warnings.filterwarnings('ignore', message='subset and superset warning')
```

And for pytest users, you can add the following to the beginning of a test script:

```
pytestmark = pytest.mark.filterwarnings('ignore:subset and superset warning
↪')
```

unique(*data*, *msg=None*)

Require that elements in *data* are unique:

```
from datatest import validate

data = [1, 2, 3, ...]

validate.unique(data)
```

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

Check that elements in *data* match the relative order of elements in *requirement*:

```
from datatest import validate

data = ['A', 'C', 'D', 'E', 'F', ...]

required_order = ['A', 'B', 'C', 'D', 'E', ...]

validate.order(data, required_order)
```

If elements do not match the required order, *Missing* and *Extra* differences are raised. Each difference will contain a two-tuple whose first value is the index of the position in *data* where the difference occurs

and whose second value is the non-matching element itself.

In the given example, *data* is missing `'B'` at index 1 and contains an extra `'F'` at index 4:

The validation fails with the following error:

```
ValidationError: does not match required order (2 differences): [
    Missing((1, 'B')),
    Extra((4, 'F')),
]
```

Notice there are no differences for `'C'`, `'D'`, and `'E'` because their relative order matches the *requirement*—even though their index positions are different.

Note: Calling *validate()* or its methods will either raise an exception or pass without error. To get an explicit True/False return value, use the *valid()* function instead.

---

datatest.**valid**(*data*, *requirement*)

Return True if *data* satisfies *requirement* else return False.

See *validate()* for supported *data* and *requirement* values and detailed validation behavior.

exception datatest.**ValidationError**(*differences*, *description=None*)

This exception is raised when data validation fails.

differences

A collection of "difference" objects to describe elements in the data under test that do not satisfy the requirement.

description

An optional description of the failed requirement.

Differences

class datatest.**BaseDifference**

The base class for "difference" objects—all other difference classes are derived from this base.

args

The tuple of arguments given to the difference constructor. Some difference (like *Deviation*) expect a certain number of arguments and assign a special meaning to the elements of this tuple, while others are called with only a single value.

class datatest.**Missing**(*value*)

Created when *value* is missing from the data under test.

In the following example, the required value `'A'` is missing from the data under test:

```
data = ['B', 'C']

requirement = {'A', 'B', 'C'}
```

(continues on next page)

(continued from previous page)

```
datatest.validate(data, requirement)
```

Running this example raises the following error:

```
ValidationError: does not satisfy set membership (1 difference): [
    Missing('A'),
]
```

class datatest.**Extra**(*value*)

Created when *value* is unexpectedly found in the data under test.

In the following example, the value `'C'` is found in the data under test but it's not part of the required values:

```
data = ['A', 'B', 'C']

requirement = {'A', 'B'}

datatest.validate(data, requirement)
```

Running this example raises the following error:

```
ValidationError: does not satisfy set membership (1 difference): [
    Extra('C'),
]
```

class datatest.**Invalid**(*invalid*, *expected=<no value>*)

Created when a value does not satisfy a function, equality, or regular expression requirement.

In the following example, the value `9` does not satisfy the required function:

```
data = [2, 4, 6, 9]

def is_even(x):
    return x % 2 == 0

datatest.validate(data, is_even)
```

Running this example raises the following error:

```
ValidationError: does not satisfy is_even() (1 difference): [
    Invalid(9),
]
```

invalid

The invalid value under test.

expected

The expected value (optional).

class datatest.**Deviation**(*deviation*, *expected*)

Created when a quantative value deviates from its expected value.

In the following example, the dictionary item `'C': 33` does not satisfy the required item `'C': 30`:

```
data = {'A': 10, 'B': 20, 'C': 33}

requirement = {'A': 10, 'B': 20, 'C': 30}
```

(continues on next page)

(continued from previous page)

```
datatest.validate(data, requirement)
```

Running this example raises the following error:

```
ValidationError: does not satisfy mapping requirement (1 difference): {
    'C': Deviation(+3, 30),
}
```

deviation
Quantative deviation from expected value.

expected
The expected value.

Acceptances

Acceptances are context managers that operate on a *ValidationError*'s collection of differences.

datatest.**accepted**(*obj*, *msg=None*, *scope=None*)
Returns a context manager that accepts differences that match *obj* without triggering a test failure. The given *obj* can be a difference class, a difference instance, or a collection of instances.

When *obj* is a difference class, differences are accepted if they are instances of the class. When *obj* is a difference instance or collection of instances, then differences are accepted if they compare as equal to one of the accepted instances.

If given, the *scope* can be `'element'`, `'group'`, or `'whole'`. An element-wise scope will accept all differences that have a match in *obj*. A group-wise scope will accept one difference per match in *obj* per group. A whole-error scope will accept one difference per match in *obj* over the ValidationError as a whole.

If unspecified, *scope* will default to `'element'` if *obj* is a single element and `'group'` if *obj* is a collection of elements. If *obj* is a mapping, the scope is limited to the group of differences associated with a given key (which effectively treats whole-error scopes the same as group-wise scopes).

Accepted Type:

When *obj* is a class (*Missing*, *Extra*, *Deviation*, *Invalid*, etc.), differences are accepted if they are instances of the class.

The following example accepts all instances of the `Missing` class:

```
from datatest import validate, accepted, Missing

data = ['A', 'B']

requirement = {'A', 'B', 'C'}

with accepted(Missing):
    validate(data, requirement)
```

Without this acceptance, the validation would have failed with the following error:

```
ValidationError: does not satisfy set membership (1 difference): [
    Missing('C'),
]
```

Accepted Difference:

When *obj* is an instance, differences are accepted if they match the instance exactly.

The following example accepts all differences that match `Extra('D')`:

```
from datatest import validate, accepted, Extra

data = ['A', 'B', 'C', 'D']

requirement = {'A', 'B', 'C'}

with accepted(Extra('D')):
    validate(data, requirement)
```

Without this acceptance, the validation would have failed with the following error:

```
ValidationError: does not satisfy set membership (1 difference): [
    Extra('D'),
]
```

Accepted Collection:

When *obj* is a collection of difference instances, then an error's differences are accepted if they match an instance in the given collection:

```
from datatest import validate, accepted, Missing, Extra

data = ['x', 'y', 'q']

requirement = {'x', 'y', 'z'}

known_issues = accepted([
    Extra('q'),
    Missing('z'),
])

with known_issues:
    validate(data, requirement)
```

A dictionary of acceptances can accept groups of differences by matching key:

```
from datatest import validate, accepted, Missing, Extra

data = {
    'A': ['x', 'y', 'q'],
    'B': ['x', 'y'],
}

requirement = {'x', 'y', 'z'}

known_issues = accepted({
    'A': [Extra('q'), Missing('z')],
    'B': [Missing('z')],
})

with known_issues:
    validate(data, requirement)
```

keys(*predicate*, *msg=None*)

Returns a context manager that accepts differences whose associated keys satisfy the given *predicate* (see

Predicates for details).

The following example accepts differences associated with the key `'B'`:

```
from datatest import validate, accepted

data = {'A': 'x', 'B': 'y'}

requirement = 'x'

with accepted.keys('B'):
    validate(data, requirement)
```

Without this acceptance, the validation would have failed with the following error:

```
ValidationError: does not satisfy 'x' (1 difference): {
    'B': Invalid('y'),
}
```

args(*predicate*, *msg=None*)

Returns a context manager that accepts differences whose `args` satisfy the given *predicate* (see *Predicates* for details).

The example below accepts differences that contain the value `'y'`:

```
from datatest import validate, accepted

data = {'A': 'x', 'B': 'y'}

requirement = 'x'

with accepted.args('y'):
    validate(data, requirement)
```

Without this acceptance, the validation would have failed with the following error:

```
ValidationError: does not satisfy 'x' (1 difference): {
    'B': Invalid('y'),
}
```

tolerance(*tolerance*, */*, *msg=None*)

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

Accepts quantitative differences within a given *tolerance* without triggering a test failure:

```
from datatest import validate, accepted

data = {'A': 45, 'B': 205}

requirement = {'A': 50, 'B': 200}

with accepted.tolerance(5):
    validate(data, requirement)
```

The example above accepts differences within a tolerance of ±5. Without this acceptance, the validation would have failed with the following error:

```
ValidationError: does not satisfy mapping requirements (2 differences): {
    'A': Deviation(-5, 50),
```

(continues on next page)

(continued from previous page)

```
    'B': Deviation(+5, 200),
}
```

Specifying different *lower* and *upper* bounds:

```
with accepted.tolerance(-2, 7):  # <- tolerance from -2 to +7
    validate(..., ...)
```

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

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

Accepts percentages of error within a given *tolerance* without triggering a test failure:

```
from datatest import validate, accepted

data = {'A': 47, 'B': 318}

requirement = {'A': 50, 'B': 300}

with accepted.percent(0.06):
    validate(data, requirement)
```

The example above accepts differences within a tolerance of ±6%. Without this acceptance, the validation would have failed with the following error:

```
ValidationError: does not satisfy mapping requirements (2 differences): {
    'A': Deviation(-3, 50),
    'B': Deviation(+18, 300),
}
```

Specifying different *lower* and *upper* bounds:

```
with accepted.percent(-0.02, 0.01):  # <- tolerance from -2% to +1%
    validate(..., ...)
```

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

fuzzy (*cutoff=0.6*, *msg=None*)

Returns a context manager that accepts invalid strings that match their expected value with a similarity greater than or equal to *cutoff* (default 0.6). Similarity measures are determined using `SequenceMatcher.ratio()` from the Standard Library's `difflib` module. The values range from `1.0` (exactly the same) to `0.0` (completely different).

The following example accepts string differences that match with a ratio of `0.6` or greater:

```
from datatest import validate, accepted

data = {'A': 'aax', 'B': 'bbx'}

requirement = {'A': 'aaa', 'B': 'bbb'}

with accepted.fuzzy(cutoff=0.6):
    validate(data, requirement)
```

Without this acceptance, the validation would have failed with the following error:

```
ValidationError: does not satisfy mapping requirements (2 differences): {
    'A': Invalid('aax', expected='aaa'),
    'B': Invalid('bbx', expected='bbb'),
}
```

count(*number*, *msg=None*, *scope=None*)

Returns a context manager that accepts up to a given *number* of differences without triggering a test failure. If the count of differences exceeds the given *number*, the test case will fail with a *ValidationError* containing the remaining differences.

The following example accepts up to 2 differences:

```
from datatest import validate, accepted

data = ['A', 'B', 'A', 'C']

requirement = 'A'

with accepted.count(2):
    validate(data, requirement)
```

Without this acceptance, the validation would have failed with the following error:

```
ValidationError: does not satisfy 'A' (2 differences): [
    Invalid('B'),
    Invalid('C'),
]
```

Composability

Acceptances can be combined to create new acceptances with modified behavior.

The `&` operator can be used to create an *intersection* of acceptance criteria. In the following example, *accepted(Missing)* and *accepted.count(5)* are combined into a single acceptance that accepts up to five Missing differences:

```
from datatest import validate, accepted

with accepted(Missing) & accepted.count(5):
    validate(..., ...)
```

The `|` operator can be used to create *union* of acceptance criteria. In the following example, *accepted.tolerance()* and *accepted.percent()* are combined into a single acceptance that accepts Deviations of ±10 as well as Deviations of ±5%:

```
from datatest import validate, accepted

with accepted.tolerance(10) | accepted.percent(0.05):
    validate(..., ...)
```

And composed acceptances, themselves, can be composed to define increasingly specific criteria:

```
from datatest import validate, accepted

five_missing = accepted(Missing) & accepted.count(5)
```

(continues on next page)

(continued from previous page)

```
minor_deviations = accepted.tolerance(10) | accepted.percent(0.05)

with five_missing | minor_deviations:
    validate(..., ...)
```

Order of Operations

Acceptance composition uses the following order of operations—shown from highest precedence to lowest precedence. Operations with the same precedence level (appearing in the same cell) are evaluated from left to right.

Order	Operation	Description
1	`()`	Parentheses
2	`&`	Bitwise AND (intersection)
3	`|`	Bitwise OR (union)
4	*`accepted(...)`* *`accepted.keys(...)`* *`accepted.args(...)`* *`accepted.tolerance(...)`* *`accepted.percent(...)`* *`accepted.fuzzy(...)`*	Element-wise acceptances
5	*`accepted([...])`*	Group-wise acceptances
6	*`accepted.count(...)`*	Whole-error acceptances

Predicates

Datatest can use *`Predicate`* objects for validation and to define certain acceptances.

class `datatest.`**Predicate**(*obj*, *name=None*)

A Predicate is used like a function of one argument that returns `True` when applied to a matching value and `False` when applied to a non-matching value. The criteria for matching is determined by the *obj* type used to define the predicate:

obj type	matches when
function	the result of `function(value)` tests as True
type	value is an instance of the type
`re.compile(pattern)`	value matches the regular expression pattern
`True`	value is truthy (`bool(value)` returns True)
`False`	value is falsy (`bool(value)` returns False)
str or non-container	value is equal to the object
set	value is a member of the set
tuple of predicates	tuple of values satisfies corresponding tuple of predicates—each according to their type
`...` (Ellipsis literal)	(used as a wildcard, matches any value)

Example matches:

obj example	value	matches
`def is_even(x):` `  return x % 2 == 0`	`4`	Yes
	`9`	No
`float`	`1.0`	Yes
	`1`	No
`re.compile('[bc]ake')`	`'bake'`	Yes
	`'cake'`	Yes
	`'fake'`	No
`True`	`'x'`	Yes
	`''`	No
`False`	`''`	Yes
	`'x'`	No
`'foo'`	`'foo'`	Yes
	`'bar'`	No
`{'A', 'B'}`	`'A'`	Yes
	`'C'`	No
`('A', float)`	`('A', 1.0)`	Yes
	`('A', 2)`	No
`('A', ...)` Uses ellipsis wildcard.	`('A', 'X')`	Yes
	`('A', 'Y')`	Yes
	`('B', 'X')`	No

Example code:

```
>>> pred = Predicate({'A', 'B'})
>>> pred('A')
True
```

(continues on next page)

(continued from previous page)

```
>>> pred('C')
False
```

Predicate matching behavior can also be inverted with the inversion operator (~). Inverted Predicates return `False` when applied to a matching value and `True` when applied to a non-matching value:

```
>>> pred = ~Predicate({'A', 'B'})
>>> pred('A')
False
>>> pred('C')
True
```

If the *name* argument is given, a `__name__` attribute is defined using the given value:

```
>>> pred = Predicate({'A', 'B'}, name='a_or_b')
>>> pred.__name__
'a_or_b'
```

If the *name* argument is omitted, the object will not have a `__name__` attribute:

```
>>> pred = Predicate({'A', 'B'})
>>> pred.__name__
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    pred.__name__
AttributeError: 'Predicate' object has no attribute '__name__'
```

1.3.2 Data Handling API Reference

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.

You can use Python's `__file__` constant to load data relative to a file's current directory:

```
from datatest import working_directory
import pandas as pd

with working_directory(__file__):
    my_df = pd.read_csv('myfile.csv')
```

This context manager can also be used as a decorator:

```
from datatest import working_directory
import pandas as pd

@working_directory(__file__)
def my_df():
    return pd.read_csv('myfile.csv')
```

Tip: Take care when using pytest's fixture finalization in combination with "session" or "module" level fixtures. In these cases, you should use *working_directory()* as a context manager—not as a decorator.

In the first example below, the original working directory is restored immediately when the `with` statement ends. But in the second example, the original directory isn't restored until after the entire session is finished (not usually what you want):

```
# Correct:

@pytest.fixture(scope='session')
def connection():
    with working_directory(__file__):
        conn = ...  # Establish database connection.
    yield conn
    conn.close()
```

```
# Wrong:

@pytest.fixture(scope='session')
@working_directory(__file__)
def connection():
    conn = ...  # Establish database connection.
    yield conn
    conn.close()
```

When a fixture does not require finalization or if the fixture is short-lived (e.g., a function-level fixture) then either form is acceptible.

Pandas Accessors

Datatest provides an optional extension accessor for integrating validation directly with pandas objects.

datatest.**register_accessors**()

Register the `validate` accessor for tighter `pandas` integration. This provides an alternate syntax for validating DataFrame, Series, Index, and MultiIndex objects.

After calling *register_accessors()*, you can use "validate" as a method:

```
import pandas as pd
import datatest as dt

df = pd.read_csv('example.csv')

dt.validate(df['A'], {'x', 'y', 'z'})  # <- Validate column 'A'.

dt.register_accessors()
df['A'].validate({'x', 'y', 'z'})  # <- Validate 'A' using accessor syntax.
```

Accessor Equivalencies

Below, you can compare the accessor syntax against the equivalent non-accessor syntax:

Accessor Syntax

Non-accessor Syntax

```
import datatest as dt
dt.register_accessors()
...

df.columns.validate({'A', 'B', 'C'})       # Index

df['A'].validate({'x', 'y', 'z'})          # Series

df['C'].validate.interval(10, 30)          # Series

df[['A', 'C']].validate((str, int))        # DataFrame
```

```
import datatest as dt

...

dt.validate(df.columns, {'A', 'B', 'C'})   # Index

dt.validate(df['A'], {'x', 'y', 'z'})      # Series

dt.validate.interval(df['C'], 10, 30)      # Series

dt.validate(df[['A', 'C']], (str, int))    # DataFrame
```

Here is the full list of accessor equivalencies:

Accessor Expression	Equivalent Non-accessor Expression
`obj.validate(requirement)`	*`validate(obj, requirement)`*
`obj.validate.predicate(requirement)`	*`validate.predicate(obj, requirement)`*
`obj.validate.regex(requirement)`	*`validate.regex(obj, requirement)`*
`obj.validate.approx(requirement)`	*`validate.approx(obj, requirement)`*
`obj.validate.fuzzy(requirement)`	*`validate.fuzzy(obj, requirement)`*
`obj.validate.interval(min, max)`	*`validate.interval(obj, min, max)`*
`obj.validate.set(requirement)`	*`validate.set(obj, requirement)`*
`obj.validate.subset(requirement)`	*`validate.subset(obj, requirement)`*
`obj.validate.superset(requirement)`	*`validate.superset(obj, requirement)`*
`obj.validate.unique()`	*`validate.unique(obj)`*
`obj.validate.order(requirement)`	*`validate.order(obj, requirement)`*

RepeatingContainer

class datatest.**RepeatingContainer**(*iterable*)

A container that repeats attribute lookups, method calls, operations, and expressions on the objects it contains. When an action is performed, it is forwarded to each object in the container and a new RepeatingContainer is returned with the resulting values.

In the following example, a RepeatingContainer with two strings is created. A method call to `upper()` is forwarded to the individual strings and a new RepeatingContainer is returned that contains the uppercase values:

```
>>> repeating = RepeatingContainer(['foo', 'bar'])
>>> repeating.upper()
RepeatingContainer(['FOO', 'BAR'])
```

A RepeatingContainer is an iterable and its individual items can be accessed through sequence unpacking or iteration. Below, the individual objects are unpacked into the variables `x` and `y`:

```
>>> repeating = RepeatingContainer(['foo', 'bar'])
>>> repeating = repeating.upper()
>>> x, y = repeating  # <- Unpack values.
>>> x
'FOO'
>>> y
'BAR'
```

If the RepeatingContainer was created with a dict (or other mapping), then iterating over it will return a sequence of `(key, value)` tuples. This sequence can be used as-is or used to create another dict:

```
>>> repeating = RepeatingContainer({'a': 'foo', 'b': 'bar'})
>>> repeating = repeating.upper()
>>> list(repeating)
[('a', 'FOO'), ('b', 'BAR')]
>>> dict(repeating)
{'a': 'FOO', 'b': 'BAR'}
```

Validating RepeatingContainer Results

When comparing the *data under test* against a set of similarly-shaped *reference data*, it's common to perform the same operations on both data sources. When queries and selections become more complex, this duplication can grow cumbersome. But duplication can be mitigated by using a *RepeatingContainer* object.

A RepeatingContainer is compatible with many types of objects—`pandas.DataFrame`, `squint.Select`, etc.

In the following example, a RepeatingContainer is created with two objects. Then, an operation is forwarded to each object in the group. Finally, the results are unpacked and validated:

With Pandas

With Squint

Below, the indexing and method calls `...[['A', 'C']].groupby('A').sum()` are forwarded to each `pandas.DataFrame` and the results are returned inside a new RepeatingContainer:

```
import datatest as dt
import pandas as pd

compare = RepeatingContainer([
```

(continues on next page)

(continued from previous page)

```
    pd.read_csv('data_under_test.csv'),
    pd.read_csv('reference_data.csv'),
])

result = compare[['A', 'C']].groupby('A').sum()

data, requirement = result
dt.validate(data, requirement)
```

Below, the method calls `...({'A': 'C'}).sum()` are forwarded to each `squint.Select` and the results are returned inside a new RepeatingContainer:

```
from datatest import validate
from squint import Select

compare = RepeatingContainer([
    Select('data_under_test.csv'),
    Select('reference_data.csv'),
])

result = compare({'A': 'C'}).sum()

data, requirement = result
validate(data, requirement)
```

The example above can be expressed even more concisely using Python's asterisk unpacking (`*`) to unpack the values directly inside the *`validate()`* call itself:

With Pandas

With Squint

```
import datatest as dt
import pandas as pd

compare = RepeatingContainer([
    pd.read_csv('data_under_test.csv'),
    pd.read_csv('reference_data.csv'),
])

dt.validate(*compare[['A', 'C']].groupby('A').sum())
```

```
from datatest import validate
from squint import Select

compare = RepeatingContainer([
    Select('data_under_test.csv'),
    Select('reference_data.csv'),
])

validate(*compare({'A': 'C'}).sum())
```

1.3.3 Unittest Support

Datatest can be used together with the `unittest` package from the Python Standard Library. For a quick introduction, see:

- *Automated Data Testing: Unittest*
- *Unittest Samples*

DataTestCase

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

This optional wrapper class provides an interface that is consistent with established unittest conventions. This class extends `unittest.TestCase` with methods for asserting validity and accepting differences. In addition, familiar methods and attributes (like `setUp()`, `maxDiff`, assertions etc.) are also available.

VALIDATION METHODS

The assertion methods wrap *`validate()`* and its methods:

```
from datatest import DataTestCase

class MyTest(DataTestCase):
    def test_mydata(self):
        data = ...
        requirement = ...
        self.assertValid(data, requirement)
```

assertValid(*data*, *requirement*, *msg=None*)
Wrapper for *`validate()`*.

assertValidPredicate(*data*, *requirement*, *msg=None*)
Wrapper for *`validate.predicate()`*.

assertValidRegex(*data*, *requirement*, *flags=0*, *msg=None*)
Wrapper for *`validate.regex()`*.

assertValidApprox(*data*, *requirement*, *places=None*, *msg=None*, *delta=None*)
Wrapper for *`validate.approx()`*.

assertValidFuzzy(*data*, *requirement*, *cutoff=0.6*, *msg=None*)
Wrapper for *`validate.fuzzy()`*.

assertValidInterval(*data*, *min=None*, *max=None*, *msg=None*)
Wrapper for *`validate.interval()`*.

assertValidSet(*data*, *requirement*, *msg=None*)
Wrapper for *`validate.set()`*.

assertValidSubset(*data*, *requirement*, *msg=None*)
Wrapper for *`validate.subset()`*.

assertValidSuperset(*data*, *requirement*, *msg=None*)
Wrapper for *`validate.superset()`*.

assertValidUnique(*data*, *msg=None*)
Wrapper for *`validate.unique()`*.

assertValidOrder(*data*, *sequence*, *msg=None*)
Wrapper for *`validate.order()`*.

ACCEPTANCE METHODS

The acceptance methods wrap *accepted()* and its methods:

```
from datatest import DataTestCase

class MyTest(DataTestCase):
    def test_mydata(self):
        data = ...
        requirement = ...
        with self.accepted(Missing):
            self.assertValid(data, requirement)
```

accepted(*obj*, *msg=None*, *scope=None*)
Wrapper for *accepted()*.

acceptedKeys(*predicate*, *msg=None*)
Wrapper for *accepted.keys()*.

acceptedArgs(*predicate*, *msg=None*)
Wrapper for *accepted.args()*.

acceptedTolerance(*tolerance*, */*, *msg=None*)
acceptedTolerance(*lower*, *upper*, *msg=None*)
Wrapper for *accepted.tolerance()*.

acceptedPercent(*tolerance*, */*, *msg=None*)
acceptedPercent(*lower*, *upper*, *msg=None*)
Wrapper for *accepted.percent()*.

acceptedFuzzy(*cutoff=0.6*, *msg=None*)
Wrapper for *accepted.fuzzy()*.

acceptedCount(*number*, *msg=None*, *scope=None*)
Wrapper for *accepted.count()*.

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.

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):
        ...
```

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 `datatest.runner.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 `datatest.main.DataTestProgram`

See the Package Index for a full list of classes and objects.

1.4 Discussion

"The right information cannot be extracted from the wrong data." —Russell Ackoff[1]

1.4.1 Organizing a Test Suite

Unlike unit testing of software, it's oftentimes not possible to check data properties as independent "units" in isolation. Later tests often depend on the success of earlier ones. For example, it's not useful to try to check the datatype of an "account_id" column if there's no column of that name. And it might not be useful to sum the values in an "accounts_payable" column when the associated account IDs contain invalid datatypes.

Typically, data tests should be run sequentially where broader, general features are tested first and specific details are tested later (after their prerequisite tests have passed). This approach is called "top-down, incremental testing". You can use the following list as a rough guide of which features to check before others.

[1] Ackoff, Russell L. "Ackoff's Best", New York: John Wiley & Sons, Inc., 1999. p. 172.

Order to Check Features

1. data is accessible (by loading a file or connecting to a data source via a fixture)
2. names of tables or worksheets (if applicable)
3. names of columns
4. categorical columns: controlled vocabulary, set membership, etc.
5. foreign-keys (if applicable)
6. well-formedness of text values: date formats, phone numbers, etc.
7. datatypes: int, float, datetime, etc.
8. constraints: uniqueness, minimum and maximum values, etc.
9. accuracy of quantitative columns: compare sums, counts, or averages against known-good values
10. internal consistency, cross-column comparisons, etc.

1.4.2 Tips and Tricks for Data Testing

This document is intended for users who are already familiar with datatest and its features. It's a grab-bag of ideas and patterns you can use in your own code.

Using methodcaller()

It's common to have helper functions that simply call a method on an object. For example:

```
from datatest import validate

data = [...]

def is_upper(x):  # <- Helper function.
    return x.isupper()

validate(data, is_upper)
```

In the case above, our helper function calls the `isupper()` method. But instead of defining a function for this, we can simply use `operator.methodcaller()`:

```
from datatest import validate
from operator import methodcaller

data = [...]
validate(data, methodcaller('isupper'))
```

RepeatingContainer and Argument Unpacking

The *RepeatingContainer* class is useful for comparing similarly shaped data sources:

```
from datatest import validate, working_directory
import pandas as pd

with working_directory(__file__):
    compare = RepeatingContainer([
        pd.read_csv('data_under_test.csv'),
        pd.read_csv('reference_data.csv'),
    ])
```

You can use *iterable unpacking* to get the individual results for validation:

```
...

result = compare[['A', 'C']].groupby('A').sum()
data, requirement = result  # Unpack result items.

dt.validate(data, requirement)
```

But you can make this even more succinct by unpacking the arguments directly inside the *validate()* call itself—via the asterisk prefix, *:

```
...

validate(*compare[['A', 'C']].groupby('A').sum())
```

Failure Message Handling

If validation fails and you've provided a *msg* argument, its value is used in the error message. But if no *msg* is given, then *validate()* will automatically generate its own message.

This example includes a *msg* value so the error message reads `Values should be even numbers.`:

```
from datatest import validate

data = [2, 4, 5, 6, 8, 10, 11]

def is_even(x):
    """Should be even."""
    return x % 2 == 0

msg = 'Values should be even numbers.'
validate(data, is_even, msg=msg)
```

```
Traceback (most recent call last):
  File "example.py", line 10, in <module>
    validate(data, is_even, msg=msg)
datatest.ValidationError: Values should be even numbers. (2 differences): [
    Invalid(5),
    Invalid(11),
]
```

1. Docstring Message

If validation fails but no *msg* was given, then the first line of the *requirement*'s docstring is used. For this reason, it's useful to start your docstrings with *normative* language, e.g., "must be…", "should have…", "needs to…", "requires…", etc.

In the following example, the docstring `Should be even.` is used in the error:

```
from datatest import validate

def is_even(x):
    """Should be even."""
    return x % 2 == 0

data = [2, 4, 5, 6, 8, 10, 11]
validate(data, is_even)
```

```
Traceback (most recent call last):
  File "example.py", line 8, in <module>
    validate(data, is_even)
datatest.ValidationError: Should be even. (2 differences): [
    Invalid(5),
    Invalid(11),
]
```

2. __name__ Message

If validation fails but there's no *msg* and no docstring, then the *requirement*'s `__name__` will be used to construct a message.

In this example, the function's name, `is_even`, is used in the error:

```
from datatest import validate

def is_even(x):
    return x % 2 == 0

data = [2, 4, 5, 6, 8, 10, 11]
validate(data, is_even)
```

```
Traceback (most recent call last):
  File "example.py", line 7, in <module>
    validate(data, is_even)
datatest.ValidationError: does not satisfy is_even() (2 differences): [
    Invalid(5),
    Invalid(11),
]
```

3. __repr__() Message

If validation fails but there's no *msg*, no docstring, and no `__name__`, then the *requirement*'s representation is used (i.e., the result of its `__repr__()` method).

Template Tests

If you need to integrate multiple published datasets, you might want to prepare a template test suite. The template can serve as a starting point for validating each datatset. In the template scripts, you can prompt users for action by explicitly failing with an instruction message:

pytest

unittest

We can use `pytest.fail()` to fail with a message to the user:

```
...

def test_state(data):
    pytest.fail('Set requirement to appropriate state abbreviation.')
    validate(data['state'], requirement='XX')
```

```
================================== FAILURES ===================================
_____________________________________ test_state ______________________________________

    def test_state(data):
>       pytest.fail('Set requirement to appropriate state abbreviation.')
E       Failed: Set requirement to appropriate state abbreviation.

example.py:24: Failed
============================ short test summary info ============================
FAILED example.py::test_state - Failed: Set requirement to appropriate state...
========================= 1 failed, 41 passed in 0.14s ==========================
```

When you see this failure, you can remove the `pytest.fail()` line and add the needed state abbreviation:

```
...

def test_state(data):
    validate(data['state'], requirement='CA')
```

We can use `TestCase.fail()` to fail with a message to the user (`DataTestCase` inherits from `unittest.TestCase` and has access to all of its parent's methods):

```
...

class MyTest(DataTestCase):
    def test_state(self):
        self.fail('Set requirement to appropriate state abbreviation.')
        self.assertValid(self.data['state'], requirement='XX')
```

```
======================================================================
FAIL: test_state (example.MyTest)
----------------------------------------------------------------------
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
  File "example.py", line 25, in test_state
    self.fail('Set requirement to appropriate state abbreviation.')
AssertionError: Set requirement to appropriate state abbreviation.

----------------------------------------------------------------------
Ran 42 tests in 0.130s

FAILED (failures=1)
```

When you see this failure, you can remove the `self.fail()` line and add the needed state abbreviation:

```
...

class MyTest(DataTestCase):
    def test_state(self):
        self.assertValid(self.data['state'], requirement='CA')
```

Using a Function Factory

If you find yourself writing multiple helper functions that perform similar actions, consider writing a *function factory* instead. A function factory is a function that makes other functions.

Using a Factory

No Factory

In the following example, the `ends_with()` function makes helper functions:

```
from datatest import validate

def ends_with(suffix):  # <- Helper function factory.
    suffix = suffix.lower()
    def helper(x):
        return x.lower().endswith(suffix)
    helper.__doc__ = f'should end with {suffix!r}'
    return helper

data1 = [...]
validate(data1, ends_with('.csv'))

data2 = [...]
validate(data2, ends_with('.txt'))

data3 = [...]
validate(data3, ends_with('.ini'))
```

Instead of a factory, this example uses separate helper functions:

```
from datatest import validate

def ends_with_csv(x):  # <- Helper function.
    """should end with '.csv'"""
    return x.lower().endswith('.csv')

def ends_with_txt(x):  # <- Helper function.
    """should end with '.txt'"""
```

(continues on next page)

(continued from previous page)

```
    return x.lower().endswith('.txt')

def ends_with_ini(x):  # <- Helper function.
    """should end with '.ini'"""
    return x.lower().endswith('.ini')

data1 = [...]
validate(data1, ends_with_csv)

data2 = [...]
validate(data2, ends_with_txt)

data3 = [...]
validate(data3, ends_with_ini)
```

Lambda Expressions

It's common to use simple helper functions like the one below:

```
from datatest import validate

def is_even(n):  # <- Helper function.
    return n % 2 == 0

data = [...]
validate(data, is_even)
```

If your helper function is a single statement, you could also write it as a lambda expression:

```
from datatest import validate

data = [...]
validate(data, lambda n: n % 2 == 0)
```

But you should be careful with lambdas because they don't have names or docstrings. If the validation fails without an explicit *msg* value, the default message can't provide any useful context—it would read "does not satisfy <lambda>".

So if you use a lambda, it's good practice to provide a *msg* argument, too:

```
from datatest import validate

data = [...]
validate(data, lambda n: n % 2 == 0, msg='shoud be even')
```

Skip Missing Test Fixtures (pytest)

Usually, you want a test to fail if a required fixture is unavailable. But in some cases, you may want to skip tests that rely on one fixture while continuing to run tests that rely on other fixtures.

To do this, you can call `pytest.skip()` from within the fixture itself. Tests that rely on this fixture will be automatically skipped when the data source is unavailable:

```
import pytest
import pandas as pd
```

(continues on next page)

(continued from previous page)

```
from datatest import working_directory

@pytest.fixture(scope='module')
@working_directory(__file__)
def data_source1():
    file_path = 'data_source1.csv'
    try:
        return pd.read_csv(file_path)
    except FileNotFoundError:
        pytest.skip(f'cannot find {file_path}')

...
```

Test Configuration With conftest.py (pytest)

To share the same fixture with multiple test modules, you can move the fixture function into a separate file named `conftest.py`. See more from the pytest docs: Sharing Fixture Functions.

1.4.3 Data Preparation

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.

Test Driven Data-Wrangling

When data is messy, poorly structured, or uses an incompatible format, it's oftentimes not possible to prepare it using an automated process. There are a multitude of ways for messy data to counfound a processing system or schema. Dealing with data like this requires a data-wrangling approach where users are actively involved with making decisions and judgment calls about cleaning and formatting the data.

A well-structured suite of data tests can serve as a template to guide the data-wrangling process. Using a quick edit-test cycle, users can:

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

[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)

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

P

R

S

T

U

V

W