tdda Documentation

Release 1.0.13

Stochastic Solutions Limited

Feb 28, 2019

Contents

Version 1.0.13

CONTENTS

CHAPTER 1

Overview

The `tdda` package provides Python support for test-driven data analysis (see 1-page summary with references, or the blog)

- The *`tdda.referencetest`* library is used to support the creation of *reference tests*, based on either `unittest` or `pytest`.
- The `tdda.constraints` library is used to *discover* constraints from a (Pandas) DataFrame, write them out as JSON, and to *verify* that datasets meet the constraints in the constraints file. There is also a command-line utility for discovering and verifying constraints.
- The *`tdda.rexpy`* library is a tool for automatically inferring regular expressions from a column in a Pandas DataFrame or from a (Python) list of examples. There is also a command-line utility for Rexpy.

CHAPTER 2

Microsoft Windows Configuration

The TDDA library makes use of some non-ASCII characters in its output. In order for these to be displayed correctly on Windows systems, a suitable font must be used.

Fonts that are known to support these characters on Windows include:

- `NSimSun`
- `MS Gothic`
- `SimSun-ExtB`

Fonts that are known not to support these characters on Windows include:

- `Consolas`
- `Courier New`
- `Lucida Console`
- `Lucida Sans Typewriter`

The font for a Command Prompt window can be set through the window's `Properties`.

Alternatively, the `--ascii` flag can be used when using verify or detect functionality.

CHAPTER 3

Command Line Tools

The `tdda` package includes two command-line utilities:

3.1 tdda

The `tdda` command provides a command-line interface for constraint discovery and verification, for a variety of data sources.

It also provides a simple way to get access to example data and tests.

See Constraint *Command-line Tool*.

3.2 rexpy

The `rexpy` command provides a command-line interface for Regular Expression generation from data examples.

See Rexpy *Command-line Tool*.

CHAPTER 4

Reference Tests

The *referencetest* module provides support for unit tests, allowing them to easily compare test results against saved "known to be correct" reference results.

This is typically useful for testing software that produces any of the following types of output:

- a CSV file
- a text file (for example: HTML, JSON, logfiles, graphs, tables, etc)
- a string
- a Pandas DataFrame.

The main features are:

- If the comparison between a string and a file fails, the actual string is written to a file and a `diff` command is suggested for seeing the differences between the actual output and the expected output.
- There is support for CSV files, allowing fine control over how the comparison is to be performed. This includes:
  - the ability to select which columns to compare (and which to exclude from the comparison).
  - the ability to compare metadata (types of fields) as well as values.
  - the ability to specify the precision (as number of decimal places) for the comparison of floating-point values.
  - clear reporting of where the differences are, if the comparison fails.
- There is support for ignoring lines within the strings/files that contain particular patterns or regular expressions. This is typically useful for filtering out things like version numbers and timestamps that vary in the output from run to run, but which do not indicate a problem.
- There is support for re-writing the reference output with the actual output. This, obviously, should be used only after careful checking that the new output is correct, either because the previous output was in fact wrong, or because the intended behaviour has changed.
- It allows you to group your reference results into different *kinds*. This means you can keep different kinds of reference result files in different locations. It also means that you can selectively choose to only regenerate

particular kinds of reference results, if they need to be updated because they turned out to have been wrong or if the intended behaviour has changed. Kinds are strings.

4.1 Prerequisites

- `pandas` optional, required for CSV file support, see http://pandas.pydata.org.
- `pytest` optional, required for tests based on pytest rather than unittest, see http://docs.pytest.org.

These can be installed with:

```
pip install pandas
pip install pytest
```

The module provides interfaces for this to be called from unit-tests based on either the standard Python `unittest` framework, or on `pytest`.

4.2 Simple Examples

Simple unittest example:

For use with `unittest`, the *`ReferenceTest`* API is provided through the *`ReferenceTestCase`* class. This is an extension to the standard `unittest.TestCase` class, so that the `ReferenceTest` methods can be called directly from `unittest` tests.

This example shows how to write a test for a function that generates a CSV file:

```
from tdda.referencetest import ReferenceTestCase, tag
import my_module

class MyTest(ReferenceTestCase):
    @tag
    def test_my_csv_file(self):
        result = my_module.produce_a_csv_file(self.tmp_dir)
        self.assertCSVFileCorrect(result, 'result.csv')

MyTest.set_default_data_location('testdata')

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

To run the test:

```
python mytest.py
```

The test is tagged with `@tag`, meaning that it will be included if you run the tests with the `--tagged` option flag to specify that only tagged tests should be run:

```
python mytest.py --tagged
```

The first time you run the test, it will produce an error unless you have already created the expected ("reference") results. You can create the reference results automatically

```
python mytest.py --write-all
```

Having generated the reference results, you should carefully examine the files it has produced in the data output location, to check that they are as expected.

Simple pytest example:

For use with `pytest`, the *ReferenceTest* API is provided through the *referencepytest* module. This is a module that can be imported directly from `pytest` tests, allowing them to access *ReferenceTest* methods and properties.

This example shows how to write a test for a function that generates a CSV file:

```
from tdda.referencetest import referencepytest, tag
import my_module

@tag
def test_my_csv_function(ref):
    resultfile = my_module.produce_a_csv_file(ref.tmp_dir)
    ref.assertCSVFileCorrect(resultfile, 'result.csv')

referencepytest.set_default_data_location('testdata')
```

You also need a `conftest.py` file, to define the fixtures and defaults:

```
import pytest
from tdda.referencetest import referencepytest

def pytest_addoption(parser):
    referencepytest.addoption(parser)

def pytest_collection_modifyitems(session, config, items):
    referencepytest.tagged(config, items)

@pytest.fixture(scope='module')
def ref(request):
    return referencepytest.ref(request)

referencepytest.set_default_data_location('testdata')
```

To run the test:

```
pytest
```

The test is tagged with `@tag`, meaning that it will be included if you run the tests with the `--tagged` option flag to specify that only tagged tests should be run:

```
pytest --tagged
```

The first time you run the test, it will produce an error unless you have already created the expected ("reference") results. You can create the reference results automatically:

```
pytest --write-all -s
```

Having generated the reference results, you should examine the files it has produced in the data output location, to check that they are as expected.

4.3 Methods and Functions

class tdda.referencetest.referencetest.**ReferenceTest**(*assert_fn*)

The *ReferenceTest* class provides support for comparing results against a set of reference "known to be correct" results.

The functionality provided by this class can be used with:

- the standard Python `unittest` framework, using the *ReferenceTestCase* class. This is a subclass of, and therefore a drop-in replacement for, `unittest.TestCase`. It extends that class with all of the methods from the *ReferenceTest* class.
- the `pytest` framework, using the *referencepytest* module. This module provides all of the methods from the *ReferenceTest* class, exposed as functions that can be called directly from tests in a `pytest` suite.

In addition to the various test-assertion methods, the module also provides some useful instance variables. All of these can be set explicitly in test setup code, using the *set_defaults()* class method:

tmp_dir The location where temporary files can be written to. It defaults to a system-specific temporary area.

print_fn The function to use to display information while running tests, which should have the same signature as Python3's standard print function (the `__future__` print function in Python2).

verbose Boolean verbose flag, to control reporting of errors while running tests. Reference tests tend to take longer to run than traditional unit tests, so it is often useful to be able to see information from failing tests as they happen, rather than waiting for the full report at the end.

all_fields_except(*exclusions*)

Helper function, for using with *check_data*, *check_types* and *check_order* parameters to assertion functions for Pandas DataFrames. It returns the names of all of the fields in the DataFrame being checked, apart from the ones given.

exclusions is a list of field names.

assertCSVFileCorrect(*actual_path*, *ref_csv*, *kind='csv'*, *csv_read_fn=None*, *check_data=None*, *check_types=None*, *check_order=None*, *condition=None*, *sortby=None*, *precision=None*, ***kwargs*)

Check that a CSV file matches a reference one.

***actual_path*:** Actual CSV file.

***ref_csv*:** Name of reference CSV file. The location of the reference file is determined by the configuration via *set_data_location()*.

***kind*:** (Optional) reference kind (a string; see above), used to locate the reference CSV file.

***csv_read_fn*:** (Optional) function to read a CSV file to obtain a pandas DataFrame. If `None`, then a default CSV loader is used.

The default CSV loader function is a wrapper around Pandas `pd.read_csv()`, with default options as follows:

- `index_col` is `None`
- `infer_datetime_format` is `True`
- `quotechar` is `"`
- `quoting` is `csv.QUOTE_MINIMAL`
- `escapechar` is `\` (backslash)

- `na_values` are the empty string, `"NaN"`, and `"NULL"`
- `keep_default_na` is `False`

***check_data*:** (Optional) restriction of fields whose values should be compared. Possible values are:

- `None` or `True` (to apply the comparison to *all* fields; this is the default).
- `False` (to skip the comparison completely)
- a list of field names (to check only these fields)
- a function taking a `DataFrame` as its single parameter, and returning a list of field names to check.

***check_types*:** (Optional) restriction of fields whose types should be compared. See *check_data* (above) for possible values.

***check_order*:** (Optional) restriction of fields whose (relative) order should be compared. See *check_data* (above) for possible values.

***check_extra_cols*:** (Optional) restriction of extra fields in the actual dataset which, if found, will cause the check to fail. See *check_data* (above) for possible values.

***sortby*:** (Optional) specification of fields to sort by before comparing.

- `None` or `False` (do not sort; this is the default)
- `True` (to sort on all fields based on their order in the reference datasets; you probably don't want to use this option)
- a list of field names (to sort on these fields, in order)
- a function taking a `DataFrame` (which will be the reference data frame) as its single parameter, and returning a list of field names to sort on.

***condition*:** (Optional) filter to be applied to datasets before comparing. It can be `None`, or can be a function that takes a *DataFrame* as its single parameter and returns a vector of booleans (to specify which rows should be compared).

***precision*:** (Optional) number of decimal places to use for floating-point comparisons. Default is not to perform rounding.

*****kwargs*:** Any additional named parameters are passed straight through to the *csv_read_fn* function.

Raises `NotImplementedError` if Pandas is not available.

assertCSVFilesCorrect(*actual_paths*, *ref_csvs*, *kind='csv'*, *csv_read_fn=None*, *check_data=None*, *check_types=None*, *check_order=None*, *condition=None*, *sortby=None*, *precision=None*, ***kwargs*)

Check that a set of CSV files match corresponding reference ones.

***actual_paths*:** List of actual CSV files.

***ref_csvs*:** List of names of matching reference CSV files. The location of the reference files is determined by the configuration via *set_data_location()*.

***kind*:** (Optional) reference kind (a string; see above), used to locate the reference CSV file.

***csv_read_fn*:** (Optional) function to read a CSV file to obtain a pandas DataFrame. If `None`, then a default CSV loader is used.

The default CSV loader function is a wrapper around Pandas `pd.read_csv()`, with default options as follows:

- `index_col` is `None`
- `infer_datetime_format` is `True`
- `quotechar` is `"`
- `quoting` is `csv.QUOTE_MINIMAL`
- `escapechar` is `\` (backslash)
- `na_values` are the empty string, `"NaN"`, and `"NULL"`
- `keep_default_na` is `False`

***check_data*:** (Optional) restriction of fields whose values should be compared. Possible values are:

- `None` or `True` (to apply the comparison to *all* fields; this is the default).
- `False` (to skip the comparison completely)
- a list of field names (to check only these fields)
- a function taking a `DataFrame` as its single parameter, and returning a list of field names to check.

***check_types*:** (Optional) restriction of fields whose types should be compared. See *check_data* (above) for possible values.

***check_order*:** (Optional) restriction of fields whose (relative) order should be compared. See *check_data* (above) for possible values.

***check_extra_cols*:** (Optional) restriction of extra fields in the actual dataset which, if found, will cause the check to fail. See *check_data* (above) for possible values.

***sortby*:** (Optional) specification of fields to sort by before comparing.

- `None` or `False` (do not sort; this is the default)
- `True` (to sort on all fields based on their order in the reference datasets; you probably don't want to use this option)
- a list of field names (to sort on these fields, in order)
- a function taking a `DataFrame` (which will be the reference data frame) as its single parameter, and returning a list of field names to sort on.

***condition*:** (Optional) filter to be applied to datasets before comparing. It can be `None`, or can be a function that takes a `DataFrame` as its single parameter and returns a vector of booleans (to specify which rows should be compared).

***precision*:** (Optional) number of decimal places to use for floating-point comparisons. Default is not to perform rounding.

****kwargs*:** Any additional named parameters are passed straight through to the *csv_read_fn* function.

Raises `NotImplementedError` if Pandas is not available.

assertDataFrameCorrect(*df*, *ref_csv*, *actual_path=None*, *kind='csv'*, *csv_read_fn=None*, *check_data=None*, *check_types=None*, *check_order=None*, *condition=None*, *sortby=None*, *precision=None*, ***kwargs*)

Check that an in-memory Pandas DataFrame matches a reference one from a saved reference CSV file.

***df*:** Actual DataFrame.

***ref_csv*:** Name of reference CSV file. The location of the reference file is determined by the configuration via *set_data_location()*.

***actual_path*:** Optional parameter, giving path for file where actual DataFrame originated, used for error messages.

***kind*:** (Optional) reference kind (a string; see above), used to locate the reference CSV file.

***csv_read_fn*:** (Optional) function to read a CSV file to obtain a pandas DataFrame. If `None`, then a default CSV loader is used.

The default CSV loader function is a wrapper around Pandas `pd.read_csv()`, with default options as follows:

- `index_col` is `None`
- `infer_datetime_format` is `True`
- `quotechar` is `"`
- `quoting` is `csv.QUOTE_MINIMAL`
- `escapechar` is `\` (backslash)
- `na_values` are the empty string, `"NaN"`, and `"NULL"`
- `keep_default_na` is `False`

***check_data*:** (Optional) restriction of fields whose values should be compared. Possible values are:

- `None` or `True` (to apply the comparison to *all* fields; this is the default).
- `False` (to skip the comparison completely)
- a list of field names (to check only these fields)
- a function taking a `DataFrame` as its single parameter, and returning a list of field names to check.

***check_types*:** (Optional) restriction of fields whose types should be compared. See *check_data* (above) for possible values.

***check_order*:** (Optional) restriction of fields whose (relative) order should be compared. See *check_data* (above) for possible values.

***check_extra_cols*:** (Optional) restriction of extra fields in the actual dataset which, if found, will cause the check to fail. See *check_data* (above) for possible values.

***sortby*:** (Optional) specification of fields to sort by before comparing.

- `None` or `False` (do not sort; this is the default)
- `True` (to sort on all fields based on their order in the reference datasets; you probably don't want to use this option)
- a list of field names (to sort on these fields, in order)
- a function taking a `DataFrame` (which will be the reference data frame) as its single parameter, and returning a list of field names to sort on.

***condition*:** (Optional) filter to be applied to datasets before comparing. It can be `None`, or can be a function that takes a *DataFrame* as its single parameter and returns a vector of booleans (to specify which rows should be compared).

***precision*:** (Optional) number of decimal places to use for floating-point comparisons. Default is not to perform rounding.

Raises `NotImplementedError` if Pandas is not available.

assertDataFramesEqual(*df*, *ref_df*, *actual_path=None*, *expected_path=None*, *check_data=None*, *check_types=None*, *check_order=None*, *condition=None*, *sortby=None*, *precision=None*)

Check that an in-memory Pandas *DataFrame* matches an in-memory reference one.

***df*:** Actual *DataFrame*.

***ref_df*:** Expected *DataFrame*.

***actual_path*:** (Optional) path for file where actual DataFrame originated, used for error messages.

***expected_path*:** (Optional) path for file where expected DataFrame originated, used for error messages.

***check_data*:** (Optional) restriction of fields whose values should be compared. Possible values are:

- `None` or `True` (to apply the comparison to *all* fields; this is the default).
- `False` (to skip the comparison completely)
- a list of field names (to check only these fields)
- a function taking a `DataFrame` as its single parameter, and returning a list of field names to check.

***check_types*:** (Optional) restriction of fields whose types should be compared. See *check_data* (above) for possible values.

***check_order*:** (Optional) restriction of fields whose (relative) order should be compared. See *check_data* (above) for possible values.

***check_extra_cols*:** (Optional) restriction of extra fields in the actual dataset which, if found, will cause the check to fail. See *check_data* (above) for possible values.

***sortby*:** (Optional) specification of fields to sort by before comparing.

- `None` or `False` (do not sort; this is the default)
- `True` (to sort on all fields based on their order in the reference datasets; you probably don't want to use this option)
- a list of field names (to sort on these fields, in order)
- a function taking a `DataFrame` (which will be the reference data frame) as its single parameter, and returning a list of field names to sort on.

***condition*:** (Optional) filter to be applied to datasets before comparing. It can be `None`, or can be a function that takes a *DataFrame* as its single parameter and returns a vector of booleans (to specify which rows should be compared).

***precision*:** (Optional) number of decimal places to use for floating-point comparisons. Default is not to perform rounding.

Raises `NotImplementedError` if Pandas is not available.

assertFileCorrect(*actual_path*, *ref_path*, *kind=None*, *lstrip=False*, *rstrip=False*, *ignore_substrings=None*, *ignore_patterns=None*, *ignore_lines=None*, *preprocess=None*, *max_permutation_cases=0*)

Check that a file matches the contents from a reference text file.

***actual_path*:** A path for a text file.

***ref_path*:** The name of the reference file. The location of the reference file is determined by the configuration via *set_data_location()*.

***kind*:** The reference *kind*, used to locate the reference file.

***lstrip*:** If set to `True`, lines are left-stripped before the comparison is carried out.

***rstrip*:** If set to `True`, lines are right-stripped before the comparison is carried out.

***ignore_substrings*:** An optional list of substrings; lines containing any of these substrings will be ignored in the comparison.

***ignore_patterns*:** An optional list of regular expressions; lines will be considered to be the same if they only differ in substrings that match one of these regular expressions. The expressions must not contain parenthesised groups, and should only include explicit anchors if they need to refer to the whole line.

ignore_lines An optional list of substrings; lines containing any of these substrings will be completely removed before carrying out the comparison. This is the means by which you would exclude 'optional' content.

***preprocess*:** An optional function that takes a list of strings and preprocesses it in some way; this function will be applied to both the actual and expected.

***max_permutation_cases*:** An optional number specifying the maximum number of permutations allowed; if the actual and expected lists differ only in that their lines are permutations of each other, and the number of such permutations does not exceed this limit, then the two are considered to be identical.

This should be used for unstructured data such as logfiles, etc. For CSV files, use *assertCSVFileCorrect()* instead.

assertFilesCorrect(*actual_paths*, *ref_paths*, *kind=None*, *lstrip=False*, *rstrip=False*, *ignore_substrings=None*, *ignore_patterns=None*, *ignore_lines=None*, *preprocess=None*, *max_permutation_cases=0*)

Check that a collection of files matche the contents from matching collection of reference text files.

***actual_paths*:** A list of paths for text files.

***ref_paths*:** A list of names of the matching reference files. The location of the reference files is determined by the configuration via *set_data_location()*.

***kind*:** The reference *kind*, used to locate the reference files. All the files must be of the same kind.

***lstrip*:** If set to `True`, lines are left-stripped before the comparison is carried out.

***rstrip*:** If set to `True`, lines are right-stripped before the comparison is carried out.

***ignore_substrings*:** An optional list of substrings; lines containing any of these substrings will be ignored in the comparison.

***ignore_patterns*:** An optional list of regular expressions; lines will be considered to be the same if they only differ in substrings that match one of these regular expressions. The expressions must not contain parenthesised groups, and should only include explicit anchors if they need to refer to the whole line.

ignore_lines An optional list of substrings; lines containing any of these substrings will be completely removed before carrying out the comparison. This is the means by which you would exclude 'optional' content.

***preprocess*:** An optional function that takes a list of strings and preprocesses it in some way; this function will be applied to both the actual and expected.

***max_permutation_cases*:** An optional number specifying the maximum number of permutations allowed; if the actual and expected lists differ only in that their lines are permutations of each other, and the number of such permutations does not exceed this limit, then the two are considered to be identical.

This should be used for unstructured data such as logfiles, etc. For CSV files, use *assertCSVFileCorrect()* instead.

assertStringCorrect(*string*, *ref_path*, *kind=None*, *lstrip=False*, *rstrip=False*, *ignore_substrings=None*, *ignore_patterns=None*, *ignore_lines=None*, *preprocess=None*, *max_permutation_cases=0*)

Check that an in-memory string matches the contents from a reference text file.

***string*:** The actual string.

***ref_path*:** The name of the reference file. The location of the reference file is determined by the configuration via *set_data_location()*.

***kind*:** The reference *kind*, used to locate the reference file.

***lstrip*:** If set to `True`, both strings are left-stripped before the comparison is carried out. Note: the stripping is on a per-line basis.

***rstrip*:** If set to `True`, both strings are right-stripped before the comparison is carried out. Note: the stripping is on a per-line basis.

***ignore_substrings*:** An optional list of substrings; lines containing any of these substrings will be ignored in the comparison.

***ignore_patterns*:** An optional list of regular expressions; lines will be considered to be the same if they only differ in substrings that match one of these regular expressions. The expressions must not contain parenthesised groups, and should only include explicit anchors if they need to refer to the whole line.

ignore_lines An optional list of substrings; lines containing any of these substrings will be completely removed before carrying out the comparison. This is the means by which you would exclude 'optional' content.

***preprocess*:** An optional function that takes a list of strings and preprocesses it in some way; this function will be applied to both the actual and expected.

***max_permutation_cases*:** An optional number specifying the maximum number of permutations allowed; if the actual and expected lists differ only in that their lines are permutations of each other, and the number of such permutations does not exceed this limit, then the two are considered to be identical.

set_data_location(*location*, *kind=None*)

Declare the filesystem location for reference files of a particular kind. Typically you would subclass *ReferenceTestCase* and pass in these locations though its __init__ method when constructing an instance of ReferenceTestCase as a superclass.

If calls to *assertFileCorrect()* (etc) are made for kinds of reference data that hasn't had their location defined explicitly, then the default location is used. This is the location declared for the `None` *kind* and this default **must** be specified.

This method overrides any global defaults set from calls to the `ReferenceeTest.set_default_data_location()` class-method.

If you haven't even defined the `None` default, and you make calls to *assertFileCorrect()* (etc) using relative pathnames for the reference data files, then it can't check correctness, so it will raise an exception.

classmethod set_default_data_location(*location*, *kind=None*)

Declare the default filesystem location for reference files of a particular kind. This sets the location for all instances of the class it is called on. Subclasses will inherit this default (unless they explicitly override it).

To set the location globally for all tests in all classes within an application, call this method on the *ReferenceTest* class.

The instance method *set_data_location()* can be used to set the per-kind data locations for an individual instance of a class.

If calls to *assertFileCorrect()* (etc) are made for kinds of reference data that hasn't had their location defined explicitly, then the default location is used. This is the location declared for the `None` *kind* and this default **must** be specified.

If you haven't even defined the `None` default, and you make calls to *assertFileCorrect()* (etc) using relative pathnames for the reference data files, then it can't check correctness, so it will raise an exception.

classmethod set_defaults(***kwargs*)

Set default parameters, at the class level. These defaults will apply to all instances of the class.

The following parameters can be set:

***verbose*: Sets the boolean verbose flag globally, to control reporting of errors while running tests. Reference tests tend to take longer to run than traditional unit tests, so it is often useful to be able to see information from failing tests as they happen, rather than waiting for the full report at the end. Verbose is set to `True` by default.

***print_fn*: Sets the print function globally, to specify** the function to use to display information while running tests. The function have the same signature as Python3's standard print function (the `__future__` print function in Python2), a default print function is used which writes unbuffered to `sys.stdout`.

***tmp_dir*:** Sets the tmp_dir property globally, to specify the directory where temporary files are written. Temporary files are created whenever a text file check fails and a 'preprocess' function has been specified. It's useful to be able to see the contents of the files after preprocessing has taken place, so preprocessed versions of the files are written to this directory, and their pathnames are included in the failure messages. If not explicitly set by *set_defaults()*, the environment variable *TDDA_FAIL_DIR* is used, or, if that is not defined, it defaults to */tmp*, *c:temp* or whatever `tempfile.gettempdir()` returns, as appropriate.

classmethod set_regeneration(*kind=None*, *regenerate=True*)

Set the regeneration flag for a particular kind of reference file, globally, for all instances of the class.

If the regenerate flag is set to `True`, then the framework will regenerate reference data of that kind, rather than comparing.

All of the regeneration flags are set to False by default.

`tdda.referencetest.referencetest.`**tag**(*test*)

Decorator for tests, so that you can specify you only want to run a tagged subset of tests, with the -1 or –tagged option.

4.4 `unittest` Framework Support

This module provides the *ReferenceTestCase* class, which extends the standard `unittest.TestCase` test-case class, augmenting it with methods for checking correctness of files against reference data.

It also provides a main() function, which can be used to run (and regenerate) reference tests which have been implemented using subclasses of ReferenceTestCase.

For example:

```
from tdda.referencetest import ReferenceTestCase
import my_module

class TestMyClass(ReferenceTestCase):
    def test_my_csv_function(self):
        result = my_module.my_csv_function(self.tmp_dir)
        self.assertCSVFileCorrect(result, 'result.csv')

    def test_my_pandas_dataframe_function(self):
        result = my_module.my_dataframe_function()
        self.assertDataFrameCorrect(result, 'result.csv')

    def test_my_table_function(self):
        result = my_module.my_table_function()
        self.assertStringCorrect(result, 'table.txt', kind='table')

    def test_my_graph_function(self):
        result = my_module.my_graph_function()
        self.assertStringCorrect(result, 'graph.txt', kind='graph')

TestMyClass.set_default_data_location('testdata')

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

4.4.1 Tagged Tests

If the tests are run with the `--tagged` or `-1` (the digit one) command-line option, then only tests that have been decorated with `referencetest.tag`, are run. This is a mechanism for allowing only a chosen subset of tests to be run, which is useful during development. The `@tag` decorator can be applied to either test classes or test methods.

If the tests are run with the `--istagged` or `-0` (the digit zero) command-line option, then no tests are run; instead, the framework reports the full module names of any test classes that have been decorated with `@tag`, or which contain any tests that have been decorated with `@tag`.

For example:

```
from tdda.referencetest import ReferenceTestCase, tag
import my_module

class TestMyClass1(ReferenceTestCase):
    @tag
    def test_a(self):
        ...

    def test_b(self):
        ...

@tag
class TestMyClass2(ReferenceTestCase):
    def test_x(self):
        ...
```

(continues on next page)

(continued from previous page)

```
    def test_y(self):
        ...
```

If run with `python mytests.py --tagged`, only the tagged tests are run (`TestMyClass1.test_a`, `TestMyClass2.test_x` and `TestMyClass2.test_y`).

4.4.2 Regeneration of Results

When its main is run with `--write-all` or `--write` (or `-W` or `-w` respectively), it causes the framework to regenerate reference data files. Different kinds of reference results can be regenerated by passing in a comma-separated list of *kind* names immediately after the `--write` option. If no list of *kind* names is provided, then all test results will be regenerated.

To regenerate all reference results (or generate them for the first time)

```
pytest -s --write-all
```

To regenerate just a particular kind of reference (e.g. table results)

```
python my_tests.py --write table
```

To regenerate a number of different kinds of reference (e.g. both table and graph results)

```
python my_tests.py --write table graph
```

4.4.3 `unittest` Integration Details

class tdda.referencetest.referencetestcase.**ReferenceTestCase**(**args*, ***kwargs*)

Wrapper around the *ReferenceTest* class to allow it to operate as a test-case class using the `unittest` testing framework.

The `ReferenceTestCase` class is a mix-in of `unittest.TestCase` and *ReferenceTest*, so it can be used as the base class for unit tests, allowing the tests to use any of the standard `unittest` *assert* methods, and also use any of the `referencetest` *assert* extensions.

static main()

Wrapper around the `unittest.main()` entry point.

This is the same as the `main()` function, and is provided just as a convenience, as it means that tests using the `ReferenceTestCase` class only need to import that single class on its own.

tag()

Decorator for tests, so that you can specify you only want to run a tagged subset of tests, with the -1 or –tagged option.

class tdda.referencetest.referencetestcase.**TaggedTestLoader**(*check*)

Subclass of TestLoader, which strips out any non-tagged tests.

getTestCaseNames(*testCaseClass*)

Return a sorted sequence of method names found within testCaseClass

loadTestsFromModule(**args*, ***kwargs*)

Return a suite of all test cases contained in the given module

loadTestsFromName(**args*, ***kwargs*)
Return a suite of all test cases given a string specifier.

The name may resolve either to a module, a test case class, a test method within a test case class, or a callable object which returns a TestCase or TestSuite instance.

The method optionally resolves the names relative to a given module.

loadTestsFromNames(**args*, ***kwargs*)
Return a suite of all test cases found using the given sequence of string specifiers. See 'loadTestsFromName()'.

loadTestsFromTestCase(**args*, ***kwargs*)
Return a suite of all test cases contained in testCaseClass

tdda.referencetest.referencetestcase.**main**()
Wrapper around the unittest.main() entry point.

4.5 pytest Framework Support

This provides all of the methods in the *ReferenceTest* class, in a way that allows them to be used as pytest fixtures.

This allows these functions to be called from tests running from the pytest framework.

For example:

```
import my_module

def test_my_csv_function(ref):
    resultfile = my_module.my_csv_function(ref.tmp_dir)
    ref.assertCSVFileCorrect(resultfile, 'result.csv')

def test_my_pandas_dataframe_function(ref):
    resultframe = my_module.my_dataframe_function()
    ref.assertDataFrameCorrect(resultframe, 'result.csv')

def test_my_table_function(ref):
    result = my_module.my_table_function()
    ref.assertStringCorrect(result, 'table.txt', kind='table')

def test_my_graph_function(ref):
    result = my_module.my_graph_function()
    ref.assertStringCorrect(result, 'graph.txt', kind='graph')

class TestMyClass:
    def test_my_other_table_function(ref):
        result = my_module.my_other_table_function()
        ref.assertStringCorrect(result, 'table.txt', kind='table')
```

with a conftest.py containing:

```
from tdda.referencetest.pytestconfig import (pytest_addoption,
                                             pytest_collection_modifyitems,
                                             set_default_data_location,
                                             ref)

set_default_data_location('testdata')
```

This configuration enables the additional command-line options, and also provides a `ref` fixture, as an instance of the `ReferenceTest` class. Of course, for brevity, if you prefer, you can use

    from tdda.referencetest.pytestconfig import *

rather than importing the four individual items if you are not customising anything yourself, but that is less flexible.

This example also sets a default data location which will apply to all reference fixtures. This means that any tests that use `ref` will automatically be able to locate their "expected results" reference data files.

4.5.1 Reference Fixtures

The default configuration provides a single fixture, `ref`.

To configure a large suite of tests so that tests do not all have to share a single common reference-data location, you can set up additional reference fixtures, configured differently. For example, to set up a fixure `ref_special`, whose reference data is stored in `../specialdata`, you could include:

```
@pytest.fixture(scope='module')
def ref_special(request):
    r = referencepytest.ref(request)
    r.set_data_location('../specialdata')
    return r
```

Tests can use this additional fixture:

```
import my_special_module

def test_something(ref_special):
    result = my_special_module.something()
    ref_special.assertStringCorrect(resultfile, 'something.csv')
```

4.5.2 Tagged Tests

If the tests are run with the `--tagged` command-line option, then only tests that have been decorated with `referencetest.tag`, are run. This is a mechanism for allowing only a chosen subset of tests to be run, which is useful during development. The `@tag` decorator can be applied to test functions, test classes and test methods.

If the tests are run with the `--istagged` command-line option, then no tests are run; instead, the framework reports the full module names of any test classes or functions that have been decorated with `@tag`, or classes which contain any test methods that have been decorated with `@tag`.

For example:

```
from tdda.referencetest import tag

@tag
def test_a(ref):
    assert 'a' + 'a' == 'aa'

def test_b(ref):
    assert 'b' * 2 == 'bb'

@tag
class TestMyClass:
    def test_x(self):
```

(continues on next page)

(continued from previous page)

```
        list('xxx') == ['x', 'x', 'x']

    def test_y(self):
        'y'.upper() == 'Y'
```

If run with `pytest --tagged`, only the tagged tests are run (`test_a`, `TestMyClass.test_x` and `TestMyClass.test_y`).

4.5.3 Regeneration of Results

When `pytest` is run with `--write-all` or `--write`, it causes the framework to regenerate reference data files. Different kinds of reference results can be regenerated by passing in a comma-separated list of *kind* names immediately after the `--write` option. If no list of *kind* names is provided, then all test results will be regenerated.

If the `-s` option is also provided (to disable `pytest` output capturing), it will report the names of all the files it has regenerated.

To regenerate all reference results (or generate them for the first time)

```
pytest -s --write-all
```

To regenerate just a particular kind of reference (e.g. table results)

```
pytest -s --write table
```

To regenerate a number of different kinds of reference (e.g. both table and graph results)

```
pytest -s --write table graph
```

4.5.4 `pytest` Integration Details

In addition to all of the methods from *ReferenceTest*, the following functions are provided, to allow easier integration with the `pytest` framework.

Typically your test code would not need to call any of these methods directly (apart from *set_default_data_location()*), as they are all enabled automatically if you import the default `ReferenceTest` configuration into your `conftest.py` file:

```
from tdda.referencetest.pytestconfig import *
```

`tdda.referencetest.referencepytest.`**`addoption`**(*parser*)

Support for the –write and –write-all command-line options.

A test's `conftest.py` file should declare extra options by defining a `pytest_addoption` function which should just call this.

It extends pytest to include **–write** and **–write-all** option flags which can be used to control regeneration of reference results.

`tdda.referencetest.referencepytest.`**`ref`**(*request*)

Support for dependency injection via a `pytest` fixture.

A test's conftest.py should define a fixture function for injecting a *ReferenceTest* instance, which should just call this function.

This allows tests to get access to a private instance of that class.

tdda.referencetest.referencepytest.**set_default_data_location**(*location*, *kind=None*)

This provides a mechanism for setting the default reference data location in the *ReferenceTest* class.

It takes the same parameters as *tdda.referencetest.referencetest.ReferenceTest.set_default_data_location()*.

If you want the same data locations for all your tests, it can be easier to set them with calls to this function, rather than having to set them explicitly in each test (or using *set_data_location()* in your `@pytest.fixture` *ref* definition in your `conftest.py` file).

tdda.referencetest.referencepytest.**set_defaults**(***kwargs*)

This provides a mechanism for setting default attributes in the *ReferenceTest* class.

It takes the same parameters as *tdda.referencetest.referencetest.ReferenceTest.set_defaults()*, and can be used for setting parameters such as the *tmp_dir* property.

If you want the same defaults for all your tests, it can be easier to set them with a call to this function, rather than having to set them explicitly in each test (or in your `@pytest.fixture` *ref* definition in your `conftest.py` file).

tdda.referencetest.referencepytest.**tagged**(*config*, *items*)

Support for @tag to mark tests to be run with –tagged or reported with –istagged.

It extends pytest to recognize the `--tagged` and `--istagged` command-line flags, to restrict testing to tagged tests only.

4.6 Examples

The *tdda.referencetest* module includes a set of examples, for both `unittest` and `pytest`.

To copy these examples to your own *referencetest-examples* subdirectory (or to a location of your choice), run the command:

```
tdda examples referencetest [mydirectory]
```

Alternatively, you can copy all examples using the following command:

```
tdda examples
```

which will create three separate sub-directories.

CHAPTER 5

Constraints

The `constraints` module provides support for constraint generation, verification and anomaly detection for datasets, including CSV files and Pandas DataFrames.

The module includes:

- A *Command-line Tool* for discovering constraints in data from various sources, and for verifying data against those constraints, using the `.tdda` *TDDA JSON file format*.
- A Python library `constraints` containing classes that implement constraint discovery and validation, for use from within other Python programs.
- Python implementations of constraint discovery, verification and and anomaly detection for a number of data sources:
  - CSV files
  - Pandas and R DataFrames saved as `.feather` files
  - PostgreSQL database tables (`postgres:`)
  - MySQL database tables (`mysql:`)
  - SQLite database tables (`sqlite:`)
  - MongoDB document collections (`mongodb`)

5.1 Python Prerequisites

- `numpy` and `pandas` (required for CSV files and `feather` files)
- `feather-format` (required for `feather` files)
- `pygresql` (required for PostgreSQL database tables)
- `mysql` (required for MySQL database tables)

These can be installed with (some/all of):

```
pip install numpy
pip install pandas
pip install feather-format
pip install pygresql
pip install mysql-python
```

The `sqlite3` module is provided by default as part of the standard Python libraries, so SQLite database tables can be used without having to explicitly install it.

To install `feather-format` on Windows, you will need to install `cython` as a prerequisite, which might also require you to install the Microsoft Visual C++ compiler for python, from http://aka.ms/vcpython27.

5.2 Command-line Tool

The `tdda` command-line utility provides a tool for discovering constraints in data and saving them as a `.tdda` file using the *TDDA JSON file format*, and also for verifying constraints in data against a previously prepared `.tdda` file.

It also provides some other functionality to help with using the tool. It takes commands in the following forms:

- *tdda discover* to perform constraint discovery.
- *tdda verify* to verify data against constraints.
- *tdda detect* to detect anomalies in data by checking constraints.
- `tdda examples` to copy example data and code where you can see them.
- `tdda help` to show help on how to use the tool.
- `tdda test` to run the TDDA library's internal tests.

See *Examples* for more detail on the code and data examples that are included as part of the `tdda` package.

See *Tests* for more detail on the `tdda` package's own tests, used to test that the package is installed and configured correctly.

5.2.1 *tdda discover*

Discover TDDA constraints for data from various sources, and save the generated constraints as a *TDDA JSON file format* file.

Usage:

```
tdda discover [FLAGS] input [constraints.tdda]
```

where

- *input* is one of:
  - a CSV file
  - a `-`, meaning it will read a csv file from standard input
  - a `feather` file containing a DataFrame, with extension `.feather`
  - a database table
- *constraints.tdda*, if provided, specifies the name of a file to which the generated constraints will be written.

If no constraints output file is provided, or is −, the generated constraints are written to standard output.

Optional flags are:

- `-r` or `--rex` to include regular expression generation
- `-R` or `--norex` to exclude regular expression generation

See *Constraints for CSV Files and Pandas DataFrames* for details of how a CSV file is read.

See *Constraints for Databases* for details of how database tables are accessed.

5.2.2 *tdda verify*

Verify data from various sources, against constraints from a *TDDA JSON file format* constraints file.

Usage:

```
tdda verify [FLAGS] input [constraints.tdda]
```

where:

- *input* is one of:
  - a csv file
  - a −, meaning it will read a csv file from standard input
  - a `feather` file containing a DataFrame, with extension `.feather`
  - a database table
- *constraints.tdda*, if provided, is a JSON *.tdda* file constaining constraints.

If no constraints file is provided and the input is a CSV or feather file, a constraints file with the same path as the input file, but with a *.tdda* extension, will be used.

For database tables, the constraints file parameter is mandatory.

Optional flags are:

- **`-a`, `--all`** Report all fields, even if there are no failures
- **`-f`, `--fields`** Report only fields with failures
- **`-7`, `--ascii`** Report in ASCII form, without using special characters.
- **`--epsilon E`** Use this value of epsilon for fuzziness in comparing numeric values.
- **`--type_checking strict|sloppy`** By default, type-checking is sloppy, meaning that when checking type constraints, all numeric types are considered to be equivalent. With strict typing, `int` is considered different from `real`.

See *Constraints for CSV Files and Pandas DataFrames* for details of how a CSV file is read.

See *Constraints for Databases* for details of how database tables are accessed.

5.2.3 *tdda detect*

Detect anomalies on data from various sources, by checking against constraints from a *TDDA JSON file format* constraints file.

Usage:

```
tdda detect [FLAGS] input constraints.tdda output
```

where:

- *input* is one of:
  - a csv file name
  - a −, meaning it will read a csv file from standard input
  - a `feather` file containing a DataFrame, with extension `.feather`
  - a database table
- *constraints.tdda*, is a JSON *.tdda* file constaining constraints.
- *output* is one of:
  - a csv file to be created containing failing records
  - a −, meaning it will write the csv file containing failing records to standard output
  - a `feather` file with extension `.feather`, to be created containing a DataFrame of failing records

If no constraints file is provided and the input is a CSV or feather file, a constraints file with the same path as the input file, but with a *.tdda* extension, will be used.

Optional flags are:

- **`-a`, `--all`** Report all fields, even if there are no failures
- **`-f`, `--fields`** Report only fields with failures
- **`-7`, `--ascii`** Report in ASCII form, without using special characters.
- **`--epsilon E`** Use this value of epsilon for fuzziness in comparing numeric values.
- **`--type_checking strict|sloppy`** By default, type-checking is sloppy, meaning that when checking type constraints, all numeric types are considered to be equivalent. With strict typing, `int` is considered different from `real`.
- **`--write-all`** Include passing records in the output.
- **`--per-constraint`** Write one column per failing constraint, as well as the `n_failures` total column for each row.
- **`--output-fields FIELD1 FIELD2 ...`** Specify original columns to write out. If used with no field names, all original columns will be included.
- **`--index`** Include a row-number index in the output file. The row number is automatically included if no output fields are specified. Rows are usually numbered from 1, unless the (feather) input file already has an index.

If no records fail any of the constraints, then no output file is created (and if the output file already exists, it is deleted).

See *Constraints for CSV Files and Pandas DataFrames* for details of how a CSV file is read.

See *Constraints for Databases* for details of how database tables are accessed.

5.3 Constraints for CSV Files and Pandas DataFrames

The *tdda.constraints.pd.constraints* module provides an implementation of TDDA constraint discovery and verification for Pandas DataFrames.

This allows it to be used for data in CSV files, or for Pandas or R DataFrames saved as Feather files.

The top-level functions are:

***discover_df()*:** Discover constraints from a Pandas DataFrame.

***verify_df()*:** Verify (check) a Pandas DataFrame, against a set of previously discovered constraints.

***detect_df()*:** Verify (check) a Pandas DataFrame, against a set of previously discovered constraints, and generate an output dataset containing information about input rows which failed any of the constraints.

tdda.constraints.pd.constraints.**discover_df**(*df*, *inc_rex=False*, *df_path=None*)

Automatically discover potentially useful constraints that characterize the Pandas DataFrame provided.

Input:

***df*:** any Pandas DataFrame.

***inc_rex*:** If `True`, include discovery of regular expressions for string fields, using rexpy (default: `False`).

***df_path*:** The path from which the dataframe was loaded, if any.

Possible return values:

- *DatasetConstraints* object
- `None` — (if no constraints were found).

This function goes through each column in the DataFrame and, where appropriate, generates constraints that describe (and are satisified by) this dataframe.

Assuming it generates at least one constraint for at least one field it returns a *tdda.constraints.base.DatasetConstraints* object.

This includes a `fields` attribute, keyed on the column name.

The returned *DatasetConstraints* object includes a `to_json()` method, which converts the constraints into JSON for saving as a tdda constraints file. By convention, such JSON files use a `.tdda` extension.

The JSON constraints file can be used to check whether other datasets also satisfy the constraints.

The kinds of constraints (potentially) generated for each field (column) are:

type: the (coarse, TDDA) type of the field. One of `bool`, `int`, `real`, `string` or `date`.

min: for non-string fields, the minimum value in the column. Not generated for all-null columns.

max: for non-string fields, the maximum value in the column. Not generated for all-null columns.

min_length: For string fields, the length of the shortest string(s) in the field. N.B. In Python2, this assumes the strings are encoded in UTF-8, and an error may occur if this is not the case. String length counts unicode characters, not bytes.

max_length: For string fields, the length of the longest string(s) in the field. N.B. In Python2, this assumes the strings are encoded in UTF-8, and an error may occur if this is not the case. String length counts unicode characters, not bytes.

sign: If all the values in a numeric field have consistent sign, a sign constraint will be written with a value chosen from:

- `positive` — For all values `v` in field: `v > 0`
- `non-negative` — For all values `v` in field: `v >= 0`
- `zero` — For all values `v` in field: `v == 0`

- `non-positive` — For all values `v` in field: `v <= 0`
- `negative` — For all values `v` in field: `v < 0`
- `null` — For all values `v` in field: `v is null`

`max_nulls`: The maximum number of nulls allowed in the field.

- If the field has no nulls, a constraint will be written with `max_nulls` set to zero.
- If the field has a single null, a constraint will be written with `max_nulls` set to one.
- If the field has more than 1 null, no constraint will be generated.

`no_duplicates`: For string fields (only, for now), if every non-null value in the field is different, this constraint will be generated (with value `True`); otherwise no constraint will be generated. So this constraint indicates that all the **non-null** values in a string field are distinct (unique).

`allowed_values`: For string fields only, if there are `MAX_CATEGORIES` or fewer distinct string values in the dataframe, an AllowedValues constraint listing them will be generated. `MAX_CATEGORIES` is currently "hard-wired" to 20.

Example usage:

```
import pandas as pd
from tdda.constraints import discover_df

df = pd.DataFrame({'a': [1, 2, 3], 'b': ['one', 'two', pd.np.NaN]})
constraints = discover_df(df)
with open('example_constraints.tdda', 'w') as f:
    f.write(constraints.to_json())
```

See *simple_generation.py* in the *Examples* for a slightly fuller example.

`tdda.constraints.pd.constraints.`**`verify_df`**(*df*, *constraints_path*, *epsilon=None*, *type_checking=None*, *report='all'*, ***kwargs*)

Verify that (i.e. check whether) the Pandas DataFrame provided satisfies the constraints in the JSON `.tdda` file provided.

Mandatory Inputs:

***df*:** A Pandas DataFrame, to be checked.

***constraints_path*:** The path to a JSON `.tdda` file (possibly generated by the discover_df function, below) containing constraints to be checked.

Optional Inputs:

***epsilon*:** When checking minimum and maximum values for numeric fields, this provides a tolerance. The tolerance is a proportion of the constraint value by which the constraint can be exceeded without causing a constraint violation to be issued.

For example, with epsilon set to 0.01 (i.e. 1%), values can be up to 1% larger than a max constraint without generating constraint failure, and minimum values can be up to 1% smaller that the minimum constraint value without generating a constraint failure. (These are modified, as appropriate, for negative values.)

If not specified, an *epsilon* of 0 is used, so there is no tolerance.

NOTE: A consequence of the fact that these are proportionate is that min/max values of zero do not have any tolerance, i.e. the wrong sign always generates a failure.

***type_checking*:** `strict` or `sloppy`. Because Pandas silently, routinely and automatically "promotes" integer and boolean columns to reals and objects respectively if they contain nulls, strict

type checking can be problematical in Pandas. For this reason, `type_checking` defaults to `sloppy`, meaning that type changes that could plausibly be attributed to Pandas type promotion will not generate constraint values.

If this is set to strict, a Pandas `float` column `c` will only be allowed to satisfy a an `int` type constraint if:

```
c.dropnulls().astype(int) == c.dropnulls()
```

Similarly, Object fields will satisfy a `bool` constraint only if:

```
c.dropnulls().astype(bool) == c.dropnulls()
```

***report*:** `all` or `fields`. This controls the behaviour of the `__str__()` method on the resulting *PandasVerification* object (but not its content).

The default is `all`, which means that all fields are shown, together with the verification status of each constraint for that field.

If report is set to `fields`, only fields for which at least one constraint failed are shown.

Returns:

PandasVerification object.

This object has attributes:

- *passes* — Number of passing constriants
- *failures* — Number of failing constraints

It also has a *to_frame()* method for converting the results of the verification to a Pandas DataFrame, and a `__str__()` method to print both the detailed and summary results of the verification.

Example usage:

```
import pandas as pd
from tdda.constraints import verify_df

df = pd.DataFrame({'a': [0, 1, 2, 10, pd.np.NaN],
                   'b': ['one', 'one', 'two', 'three', pd.np.NaN]})
v = verify_df(df, 'example_constraints.tdda')

print('Constraints passing: %d\n' % v.passes)
print('Constraints failing: %d\n' % v.failures)
print(str(v))
print(v.to_frame())
```

See *simple_verification.py* in the *Examples* for a slightly fuller example.

`tdda.constraints.pd.constraints.`**`detect_df`**(*df*, *constraints_path*, *epsilon=None*, *type_checking=None*, *outpath=None*, *write_all=False*, *per_constraint=False*, *output_fields=None*, *index=False*, *in_place=False*, *rownumber_is_index=True*, *boolean_ints=False*, *report='records'*, ***kwargs*)

Check the records from the Pandas DataFrame provided, to detect records that fail any of the constraints in the JSON `.tdda` file provided. This is anomaly detection.

Mandatory Inputs:

***df*:** A Pandas DataFrame, to be checked.

***constraints_path*:** The path to a JSON `.tdda` file (possibly generated by the discover_df function, below) containing constraints to be checked.

Optional Inputs:

***epsilon*:** When checking minimum and maximum values for numeric fields, this provides a tolerance. The tolerance is a proportion of the constraint value by which the constraint can be exceeded without causing a constraint violation to be issued.

For example, with epsilon set to 0.01 (i.e. 1%), values can be up to 1% larger than a max constraint without generating constraint failure, and minimum values can be up to 1% smaller that the minimum constraint value without generating a constraint failure. (These are modified, as appropriate, for negative values.)

If not specified, an *epsilon* of 0 is used, so there is no tolerance.

NOTE: A consequence of the fact that these are proportionate is that min/max values of zero do not have any tolerance, i.e. the wrong sign always generates a failure.

***type_checking*:** `strict` or `sloppy`. Because Pandas silently, routinely and automatically "promotes" integer and boolean columns to reals and objects respectively if they contain nulls, strict type checking can be problematical in Pandas. For this reason, `type_checking` defaults to `sloppy`, meaning that type changes that could plausibly be attributed to Pandas type promotion will not generate constraint values.

If this is set to strict, a Pandas `float` column `c` will only be allowed to satisfy a an `int` type constraint if:

```
c.dropnulls().astype(int) == c.dropnulls()
```

Similarly, Object fields will satisfy a `bool` constraint only if:

```
c.dropnulls().astype(bool) == c.dropnulls()
```

***outpath*:** This specifies that the verification process should detect records that violate any constraints, and write them out to this CSV (or feather) file.

By default, only failing records are written out to file, but this can be overridden with the `write_all` parameter.

By default, the columns in the detection output file will be a boolean `ok` field for each constraint on each field, an and `n_failures` field containing the total number of constraints that failed for each row. This behavious can be overridden with the `per_constraint`, `output_fields` and `index` parameters.

***write_all*:** Include passing records in the detection output file when detecting.

***per_constraint*:** Write one column per failing constraint, as well as the `n_failures` total.

***output_fields*:** Specify original columns to write out when detecting.

If passed in as an empty list (rather than None), all original columns will be included.

***index*:** Boolean to specify whether to include a row-number index in the output file when detecting.

This is automatically enabled if no output field names are specified.

Rows are numbered from 0.

***in_place*:** Detect failing constraints by adding columns to the input DataFrame.

If `outpath` is also specified, then failing records will also be written to file.

***rownumber_is_index*:** `False` if the DataFrame originated from a CSV file (and therefore any detection output file should refer to row numbers from the file, rather than items from the DataFrame index).

***boolean_ints*:** If `True`, write out all boolean values to CSV file as integers (1 for true, and 0 for false), rather than as `true` and `false` values.

The *report* parameter from *verify_df()* can also be used, in which case a verification report will also be produced in addition to the detection results.

Returns:

PandasDetection object.

This object has a *detected()* method for obtaining the Pandas DataFrame containing the detection results.

Example usage:

```
import pandas as pd
from tdda.constraints import detect_df

df = pd.DataFrame({'a': [0, 1, 2, 10, pd.np.NaN],
                   'b': ['one', 'one', 'two', 'three', pd.np.NaN]})
v = detect_df(df, 'example_constraints.tdda')
detection_df = v.detected()
print(detection_df.to_string())
```

class tdda.constraints.pd.constraints.**PandasConstraintCalculator**(*df*)
Implementation of the Constraint Calculator methods for Pandas dataframes.

class tdda.constraints.pd.constraints.**PandasConstraintDetector**(*df*)
Implementation of the Constraint Detector methods for Pandas dataframes.

class tdda.constraints.pd.constraints.**PandasConstraintVerifier**(*df*, *epsilon=None*, *type_checking=None*)
A *PandasConstraintVerifier* object provides methods for verifying every type of constraint against a Pandas DataFrame.

class tdda.constraints.pd.constraints.**PandasConstraintDiscoverer**(*df*, *inc_rex=False*)
A *PandasConstraintDiscoverer* object is used to discover constraints on a Pandas DataFrame.

class tdda.constraints.pd.constraints.**PandasVerification**(**args*, ***kwargs*)
A *PandasVerification* object adds a *to_frame()* method to a *tdda.constraints.base.Verification* object.

This allows the result of constraint verification to be converted to a Pandas DataFrame, including columns for the field (column) name, the numbers of passes and failures and boolean columns for each constraint, with values:

- `True` — if the constraint was satified for the column
- `False` — if column failed to satisfy the constraint
- `pd.np.NaN` — if there was no constraint of this kind

This Pandas-specific implementation of constraint verification also provides methods *to_frame()* to get the overall verification result as as a Pandas DataFrame, and `detected()` to get any detection results as a a Pandas DataFrame (if the verification has been run with in `detect` mode).

to_dataframe()
Converts object to a Pandas DataFrame.

to_frame()
Converts object to a Pandas DataFrame.

class tdda.constraints.pd.constraints.**PandasDetection**(**args*, ***kwargs*)
A *PandasDetection* object adds a *detected()* method to a *PandasVerification* object.

This allows the Pandas DataFrame resulting from constraint detection to be made available.

The object also provides properties *n_passing_records* and *n_failing_records*, recording how many records passed and failed the detection process.

detected()
Returns a Pandas DataFrame containing the detection results.

If there are no failing records, and the detection was not run with the *write_all* flag set, then `None` is returned.

If a CSV file is used with the `tdda` command-line tool, it will be processed by the standard Pandas CSV file reader with the following settings:

- `index_col` is `None`
- `infer_datetime_format` is `True`
- `quotechar` is `"`
- `quoting` is `csv.QUOTE_MINIMAL`
- `escapechar` is `\\` (backslash)
- `na_values` are the empty string, `"NaN"`, and `"NULL"`
- `keep_default_na` is `False`

5.4 Constraints for Databases

When a database table is used with the `tdda` command-line tool, the table name (including an optional schema) can be preceded by `DBTYPE` chosen from `postgres`, `mysql`, `sqlite` or `mongodb`:

```
DBTYPE:schema.tablename
```

If `DBTYPE` is used, this will cause the file `.tdda_db_conn_DBTYPE` to be read from your home directory (see *Database Connection Files*), which can contain all connection parameters.

Parameters can also be provided using the following flags (which override the values in the `.tdda_db_conn_DBTYPE` file, if provided):

- **-conn FILE** Database connection file (see *Database Connection Files*)
- **-dbtype DBTYPE** Type of database
- **-db DATABASE** Name of database to connect to
- **-host HOSTNAME** Name of server to connect to
- **-port PORTNUMBER** IP port number to connect to
- **-user USERNAME** Username to connect as
- **-password PASSWORD** Password to authenticate with

If `-conn` is provided, then none of the other options are required, and the database connection details are read from the specified file.

If the database type is specified (with the `-dbtype` option, or by prefixing the table name, such as `postgres:mytable`), then a default connection file `.tdda_db_conn_DBTYPE` (in your home directory) is used, if present.

5.4.1 Database Connection Files

To use a database source, you can either specify the database type using the `--dbtype DBTYPE` option, or you can prefix the table name with `DBTYPE:`.

You can provide default values for all of the other database options in a database connection file `.tdda_db_conn_DBTYPE`, in your home directory.

Any database-related options passed in on the command line will override the default settings from the connection file.

A `tdda_db_conn_DBTYPE` file is a JSON file of the form:

```
{
    "dbtype": DBTYPE,
    "db": DATABASE,
    "host": HOSTNAME,
    "port": PORTNUMBER,
    "user": USERNAME,
    "password": PASSWORD,
    "schema": SCHEMA,
}
```

All the entries are optional.

If a `password` is provided, then care should be taken to ensure that the file has appropriate filesystem permissions so that it cannot be read by other users.

If a `schema` is provided, then it will be used as the default schema, when constraints are discovered or verified on a table name with no schema specified.

5.4.2 API

TDDA constraint discovery and verification is provided for a number of DB-API (PEP-0249) compliant databases, and also for a number of other (NoSQL) databases.

The top-level functions are:

***discover_db_table()*:** Discover constraints from a single database table.

***verify_db_table()*:** Verify (check) a single database table, against a set of previously discovered constraints.

detect_db_table(): Verify (check) a single database table, against a set of previously discovered constraints.

tdda.constraints.db.constraints.**discover_db_table**(*dbtype*, *db*, *tablename*, *inc_rex=False*)

Automatically discover potentially useful constraints that characterize the database table provided.

Input:

***dbtype*:** Type of database.

***db*:** a database object

***tablename*:** a table name

Possible return values:

- *DatasetConstraints* object
- None — (if no constraints were found).

This function goes through each column in the table and, where appropriate, generates constraints that describe (and are satisified by) this dataframe.

Assuming it generates at least one constraint for at least one field it returns a *tdda.constraints.base.DatasetConstraints* object.

This includes a 'fields' attribute, keyed on the column name.

The returned *DatasetConstraints* object includes a to_json() method, which converts the constraints into JSON for saving as a tdda constraints file. By convention, such JSON files use a '.tdda' extension.

The JSON constraints file can be used to check whether other datasets also satisfy the constraints.

The kinds of constraints (potentially) generated for each field (column) are:

***type*:** the (coarse, TDDA) type of the field. One of 'bool', 'int', 'real', 'string' or 'date'.

***min*:** for non-string fields, the minimum value in the column. Not generated for all-null columns.

***max*:** for non-string fields, the maximum value in the column. Not generated for all-null columns.

***min_length*:** For string fields, the length of the shortest string(s) in the field.

***max_length*:** For string fields, the length of the longest string(s) in the field.

***sign*:** If all the values in a numeric field have consistent sign, a sign constraint will be written with a value chosen from:

- positive — For all values *v* in field: *v > 0*
- non-negative — For all values *v* in field: *v >= 0*
- zero — For all values *v* in field: *v == 0*
- non-positive — For all values *v* in field: *v <= 0*
- negative — For all values *v* in field: *v < 0*
- null — For all values *v* in field: *v is null*

***max_nulls*:** The maximum number of nulls allowed in the field.

- If the field has no nulls, a constraint will be written with max_nulls set to zero.
- If the field has a single null, a constraint will be written with max_nulls set to one.
- If the field has more than 1 null, no constraint will be generated.

***no_duplicates*:** For string fields (only, for now), if every non-null value in the field is different, this constraint will be generated (with value True); otherwise no constraint will be generated. So this constraint indicates that all the **non-null** values in a string field are distinct (unique).

***allowed_values*:** For string fields only, if there are MAX_CATEGORIES or fewer distinct string values in the dataframe, an AllowedValues constraint listing them will be generated. MAX_CATEGORIES is currently "hard-wired" to 20.

Example usage:

```
import pgdb
from tdda.constraints import discover_db_table

dbspec = 'localhost:databasename:username:password'
tablename = 'schemaname.tablename'
db = pgdb.connect(dbspec)
constraints = discover_db_table('postgres', db, tablename)

with open('myconstraints.tdda', 'w') as f:
    f.write(constraints.to_json())
```

tdda.constraints.db.constraints.**verify_db_table**(*dbtype*, *db*, *tablename*, *constraints_path*, *epsilon=None*, *type_checking='strict'*, *testing=False*, *report='all'*, ***kwargs*)

Verify that (i.e. check whether) the database table provided satisfies the constraints in the JSON .tdda file provided.

Mandatory Inputs:

***dbtype*:** Type of database.

***db*:** A database object

***tablename*:** A database table name, to be checked.

***constraints_path*:** The path to a JSON .tdda file (possibly generated by the discover_constraints function, below) containing constraints to be checked.

Optional Inputs:

***epsilon*:** When checking minimum and maximum values for numeric fields, this provides a tolerance. The tolerance is a proportion of the constraint value by which the constraint can be exceeded without causing a constraint violation to be issued.

For example, with epsilon set to 0.01 (i.e. 1%), values can be up to 1% larger than a max constraint without generating constraint failure, and minimum values can be up to 1% smaller that the minimum constraint value without generating a constraint failure. (These are modified, as appropriate, for negative values.)

If not specified, an *epsilon* of 0 is used, so there is no tolerance.

NOTE: A consequence of the fact that these are proportionate is that min/max values of zero do not have any tolerance, i.e. the wrong sign always generates a failure.

***type_checking*:** `strict` or `sloppy`. For databases (unlike Pandas DataFrames), this defaults to 'strict'.

If this is set to sloppy, a database "real" column c will only be allowed to satisfy a an "int" type constraint.

***report*:** `all` or `fields`. This controls the behaviour of the `__str__()` method on the resulting *DatabaseVerification* object (but not its content).

The default is `all`, which means that all fields are shown, together with the verification status of each constraint for that field.

If report is set to `fields`, only fields for which at least one constraint failed are shown.

***testing*:** Boolean flag. Should only be set to `True` when being run as part of an automated test. It suppresses type-compatibility warnings.

Returns:

DatabaseVerification object.

This object has attributes:

- *passed* — Number of passing constriants
- *failures* — Number of failing constraints

Example usage:

```
import pgdb
from tdda.constraints import verify_db_table

dbspec = 'localhost:databasename:username:password'
tablename = 'schemaname.tablename'
db = pgdb.connect(dbspec)
v = verify_db_table('postgres' db, tablename, 'myconstraints.tdda')

print('Constraints passing:', v.passes)
print('Constraints failing: %d\n' % v.failures)
print(str(v))
```

class tdda.constraints.db.constraints.**DatabaseConstraintVerifier**(*dbtype*, *db*, *tablename*, *epsilon=None*, *type_checking='strict'*, *testing=False*)

A *DatabaseConstraintVerifier* object provides methods for verifying every type of constraint against a single database table.

class tdda.constraints.db.constraints.**DatabaseVerification**(**args*, ***kwargs*)

A *DatabaseVerification* object is the variant of the *tdda.constraints.base.Verification* object used for verification of constraints on a database table.

class tdda.constraints.db.constraints.**DatabaseConstraintDiscoverer**(*dbtype*, *db*, *tablename*, *inc_rex=False*)

A *DatabaseConstraintDiscoverer* object is used to discover constraints on a single database table.

5.5 Extension Framework

The `tdda` command-line utility provides built-in support for constraint discovery and verification for tabular data stored in CSV files, Pandas DataFrames saved in `.feather` files, and for a tables in a variety of different databases.

The utility can be extended to provide support for constraint discovery and verification for other kinds of data, via its Python extension framework.

The framework will automatically use any extension implementations that have been declared using the `TDDA_EXTENSIONS` environment variable. This should be set to a list of class names, for Python classes that extend the *ExtensionBase* base class.

The class names in the `TDDA_EXTENSIONS` environment variable should be colon-separated for Unix systems, or semicolon-separated for Microsoft Windows. To be usable, the classes must be accessible by Python (either by being installed in Pythons standard module directory, or by being included in the `PYTHONPATH` environment variable.

For example:

```
export TDDA_EXTENSIONS="mytdda.MySpecialExtension"
export PYTHONPATH="/my/python/sources:$PYTHONPATH"
```

With these in place, the `tdda` command will include constraint discovery and verification using the `MySpecialExtension` implementation class provided in the Python file `/my/python/sources/mytdda.py`.

An example of a simple extension is included with the set of standard examples. See *Examples*.

5.5.1 Extension Overview

An extension should provide:

- an implementation (subclass) of *ExtensionBase*, to provide a command-line interface, extending the `tdda` command to support a particular type of input data.
- an implementation (subclass) of *BaseConstraintCalculator*, to provide methods for computing individual constraint results.
- an implementation (subclass) of *BaseConstraintDetector*, to provide methods for generating detection results.

A typical implementation looks like:

```
from tdda.constraints.flags import discover_parser, discover_flags
from tdda.constraints.flags import verify_parser, verify_flags
from tdda.constraints.flags import detect_parser, detect_flags
from tdda.constraints.extension import ExtensionBase
from tdda.constraints.base import DatasetConstraints, Detection
from tdda.constraints.baseconstraints import (BaseConstraintCalculator,
                                              BaseConstraintVerifier,
                                              BaseConstraintDetector,
                                              BaseConstraintDiscoverer)
from tdda.rexpy import rexpy

class MyExtension(ExtensionBase):
    def applicable(self):
        ...

    def help(self, stream=sys.stdout):
        print('...', file=stream)

    def spec(self):
        return '...'

    def discover(self):
        parser = discover_parser()
        parser.add_argument(...)
        params = {}
        flags = discover_flags(parser, self.argv[1:], params)
        data = ... get data source from flags ...
        discoverer = MyConstraintDiscoverer(data, **params)
        constraints = discoverer.discover()
        results = constraints.to_json()
        ... write constraints JSON to output file
        return results
```

(continues on next page)

(continued from previous page)

```
    def verify(self):
        parser = verify_parser()
        parser.add_argument(...)
        params = {}
        flags = verify_flags(parser, self.argv[1:], params)
        data = ... get data source from flags ...
        verifier = MyConstraintVerifier(data, **params)
        constraints = DatasetConstraints(loadpath=...)
        results = verifier.verify(constraints)
        return results

    def detect(self):
        parser = detect_parser()
        parser.add_argument(...)
        params = {}
        flags = detect_flags(parser, self.argv[1:], params)
        data = ... get data source from flags ...
        detector = MyConstraintDetector(data, **params)
        constraints = DatasetConstraints(loadpath=...)
        results = detector.detect(constraints)
        return results
```

5.5.2 Extension API

class tdda.constraints.extension.**BaseConstraintCalculator**

The `BaseConstraintCalculator` class defines a default or dummy implementation of all of the methods that are required in order to implement a constraint discoverer or verifier via subclasses of the base `BaseConstraintDiscoverer` and `BaseConstraintVerifier` classes.

allowed_values_exclusions()

Get list of values to ignore when computing allowed values

calc_all_non_nulls_boolean(*colname*)

Checks whether all the non-null values in a column are boolean. Returns True of they are, and False otherwise.

This is only required for implementations where a dataset column may contain values of mixed type.

calc_max(*colname*)

Calculates the maximum (non-null) value in the named column.

calc_max_length(*colname*)

Calculates the length of the longest string(s) in the named column.

calc_min(*colname*)

Calculates the minimum (non-null) value in the named column.

calc_min_length(*colname*)

Calculates the length of the shortest string(s) in the named column.

calc_non_integer_values_count(*colname*)

Calculates the number of unique non-integer values in a column

This is only required for implementations where a dataset column may contain values of mixed type.

calc_non_null_count(*colname*)

Calculates the number of nulls in a column

calc_null_count(*colname*)
Calculates the number of nulls in a column

calc_nunique(*colname*)
Calculates the number of unique non-null values in a column

calc_rex_constraint(*colname*, *constraint*, *detect=False*)
Verify whether a given column satisfies a given regular expression constraint (by matching at least one of the regular expressions given).

Returns a 'truthy' value (typically the set of the strings that do not match any of the regular expressions) on failure, and a 'falsy' value (typically False or None or an empty set) if there are no failures. Any contents of the returned value are used in the case where detect is set, by the corresponding extension method for recording detection results.

calc_tdda_type(*colname*)
Calculates the TDDA type of a column

calc_unique_values(*colname*, *include_nulls=True*)
Calculates the set of unique values (including or excluding nulls) in a column

column_exists(*colname*)
Returns whether this column exists in the dataset

find_rexes(*colname*, *values=None*)
Generate a list of regular expressions that cover all of the patterns found in the (string) column.

get_column_names()
Returns a list containing the names of all the columns

get_nrecords()
Return total number of records

is_null(*value*)
Determine whether a value is null

to_datetime(*value*)
Convert a value to a datetime

types_compatible(*x*, *y*, *colname*)
Determine whether the types of two values are compatible

class tdda.constraints.extension.**BaseConstraintDetector**
The `BaseConstraintDetector` class defines a default or dummy implementation of all of the methods that are required in order to implement constraint detection via the a subclass of the base `BaseConstraintVerifier` class.

detect_allowed_values_constraint(*colname*, *value*, *violations*)
Detect failures for an allowed_values constraint.

detect_max_constraint(*colname*, *value*, *precision*, *epsilon*)
Detect failures for a max constraint.

detect_max_length_constraint(*colname*, *value*)
Detect failures for a max_length constraint.

detect_max_nulls_constraint(*colname*, *value*)
Detect failures for a max_nulls constraint.

detect_min_constraint(*colname*, *value*, *precision*, *epsilon*)
Detect failures for a min constraint.

detect_min_length_constraint(*colname*, *value*)
Detect failures for a min_length constraint.

detect_no_duplicates_constraint(*colname*, *value*)
Detect failures for a no_duplicates constraint.

detect_rex_constraint(*colname*, *value*, *violations*)
Detect failures for a rex constraint.

detect_sign_constraint(*colname*, *value*)
Detect failures for a sign constraint.

detect_tdda_type_constraint(*colname*, *value*)
Detect failures for a type constraint.

write_detected_records(*detect_outpath=None*, *detect_write_all=False*, *detect_per_constraint=False*, *detect_output_fields=None*, *detect_index=False*, *detect_in_place=False*, *rownumber_is_index=True*, *boolean_ints=False*, ***kwargs*)
Write out a detection dataset.

Returns a :py:class:~tdda.constraints.base.Detection object (or None).

class tdda.constraints.extension.**ExtensionBase**(*argv*, *verbose=False*)
An extension must provide a class that is based on the *ExtensionBase* class, providing implementations for its *applicable()*, *help()*, *discover()* and *verify()* methods.

applicable()
The *applicable()* method should return True if the argv property contains command-line parameters that can be used by this implementation.

For example, if the extension can handle data stored in Excel .xlsx files, then its *applicable()* method should return True if any of its parameters are filenames that have a .xlsx suffix.

detect()
The *detect()* method should implement constraint detection.

It should read constraints from a .tdda file specified on the command line, and verify these constraints on the data specified, and produce detection output.

It should use the self.argv variable to get whatever other optional or mandatory flags or parameters are required to specify the data on which the constraints are to be verified, where the output detection data should be written, and detection-specific flags.

discover()
The *discover()* method should implement constraint discovery.

It should use the self.argv variable to get whatever other optional or mandatory flags or parameters are required to specify the data from which constraints are to be discovered, and the name of the file to which the constraints are to be written.

help(*self*, *stream=sys.stdout*)
The *help()* method should document itself by writing lines to the given output stream.

This is used by the tdda command's help option.

spec()
The *spec()* method should return a short one-line string describing, briefly, how to specify the input source.

verify()
The *verify()* method should implement constraint verification.

It should read constraints from a `.tdda` file specified on the command line, and verify these constraints on the data specified.

It should use the `self.argv` variable to get whatever other optional or mandatory flags or parameters are required to specify the data on which the constraints are to be verified.

5.6 Constraints API

TDDA constraint discovery and verification, common underlying functionality.

class tdda.constraints.baseconstraints.**BaseConstraintDiscoverer**(*inc_rex=False*, ***kwargs*)

The `BaseConstraintDiscoverer` class provides a generic framework for discovering constraints.

A concrete implementation of this class is constructed by creating a mix-in subclass which inherits both from `BaseConstraintDiscover` and from a specific implementation of `BaseConstraintCalculator`.

class tdda.constraints.baseconstraints.**BaseConstraintVerifier**(*epsilon=None*, *type_checking=None*, ***kwargs*)

The `BaseConstraintVerifier` class provides a generic framework for verifying constraints.

A concrete implementation of this class is constructed by creating a mix-in subclass which inherits both from `BaseConstraintVerifier` and from specific implementations of `BaseConstraintCalculator` and `BaseConstraintDetector`.

cache_values(*colname*)
Returns the dictionary for colname from the cache, first creating it if there isn't one on entry.

detect(*constraints*, *VerificationClass=<class 'tdda.constraints.base.Verification'>*, *outpath=None*, *write_all=False*, *per_constraint=False*, *output_fields=None*, *index=False*, *in_place=False*, *rownumber_is_index=True*, *boolean_ints=False*, ***kwargs*)
Apply verifiers to a set of constraints, for detection.

Note that if there is a constraint for a field that does not exist, then it fails verification, but there are no records to detect against. Similarly if the field exists but the dataset has no records.

get_all_non_nulls_boolean(*colname*)
Looks up or caches the number of non-integer values in a real column, or calculates and caches it.

get_cached_value(*value*, *colname*, *f*)
Return cached value of colname, calculating it and caching it first, if it is not already there.

get_max(*colname*)
Looks up cached maximum of column, or calculates and caches it

get_max_length(*colname*)
Looks up cached maximum string length in column, or calculates and caches it

get_min(*colname*)
Looks up cached minimum of column, or calculates and caches it

get_min_length(*colname*)
Looks up cached minimum string length in column, or calculates and caches it

get_non_integer_values_count(*colname*)
Looks up or caches the number of non-integer values in a real column, or calculates and caches it.

get_non_null_count(*colname*)
Looks up or caches the number of non-null values in a column, or calculates and caches it

get_null_count(*colname*)
Looks up or caches the number of nulls in a column, or calculates and caches it

get_nunique(*colname*)
Looks up or caches the number of unique (distinct) values in a column, or calculates and caches it.

get_tdda_type(*colname*)
Looks up cached tdda type of a column, or calculates and caches it

get_unique_values(*colname*)
Looks up or caches the list of unique (distinct) values in a column, or calculates and caches it.

verifiers()
Returns a dictionary mapping constraint types to their callable (bound) verification methods.

verify(*constraints*, *VerificationClass=<class 'tdda.constraints.base.Verification'>*, ***kwargs*)
Apply verifiers to a set of constraints, for reporting

verify_allowed_values_constraint(*colname*, *constraint*, *detect=False*)
Verify whether a given column satisfies the constraint on allowed (string) values provided.

verify_max_constraint(*colname*, *constraint*, *detect=False*)
Verify whether a given column satisfies the maximum value constraint specified.

verify_max_length_constraint(*colname*, *constraint*, *detect=False*)
Verify whether a given (string) column satisfies the maximum length constraint specified.

verify_max_nulls_constraint(*colname*, *constraint*, *detect=False*)
Verify whether a given column satisfies the supplied constraint that it should contain no nulls.

verify_min_constraint(*colname*, *constraint*, *detect=False*)
Verify whether a given column satisfies the minimum value constraint specified.

verify_min_length_constraint(*colname*, *constraint*, *detect=False*)
Verify whether a given (string) column satisfies the minimum length constraint specified.

verify_no_duplicates_constraint(*colname*, *constraint*, *detect=False*)
Verify whether a given column satisfies the constraint supplied, that it should contain no duplicate (non-null) values.

verify_rex_constraint(*colname*, *constraint*, *detect=False*)
Verify whether a given column satisfies a given regular expression constraint (by matching at least one of the regular expressions given).

verify_sign_constraint(*colname*, *constraint*, *detect=False*)
Verify whether a given column satisfies the supplied sign constraint.

verify_tdda_type_constraint(*colname*, *constraint*, *detect=False*)
Verify whether a given column satisfies the supplied type constraint.

5.6.1 Underlying API Classes

Classes for representing individual constraints.

class tdda.constraints.base.**DatasetConstraints**(*per_field_constraints=None*, *load-path=None*)
Container for constraints pertaining to a dataset. Currently only supports per-field constraints.

initialize_from_dict(*in_constraints*)
Initializes this object from a dictionary in_constraints. Currently, the only key used from in_constraints is fields.

The value of in_constraints['fields'] is expected to be a dictionary, keyed on field name, whose values are the constraints for that field.

They constraints are keyed on the kind of constraint, and should contain either a single value (a scalar or a list), or a dictionary of keyword arguments for the constraint initializer.

load(*path*)
Builds a DatasetConstraints object from a json file

sort_fields(*fields=None*)
Sorts the field constraints within the object by field order, by default by alphabetical order.

If a list of field names is provided, then the fields will appear in that given order (with any additional fields appended at the end).

to_dict(*tddafile=None*)
Converts the constraints in this object to a dictionary.

to_json(*tddafile=None*)
Converts the constraints in this object to JSON. The resulting JSON is returned.

class tdda.constraints.base.**FieldConstraints**(*name=None*, *constraints=None*)
Container for constraints on a field.

to_dict_value()
Returns a pair consisting of the name supplied, or the stored name, and an ordered dictionary keyed on constraint kind with the value specifying the constraint. For simple constraints, the value is a base type; for more complex constraints with several components, the value will itself be an (ordered) dictionary.

The ordering is all to make the JSON file get written in a sensible order, rather than being a jumbled mess.

class tdda.constraints.base.**MultiFieldConstraints**(*names=None*, *constraints=None*)
Container for constraints on a pairs (or higher numbers) of fields

to_dict_value()

Returns a pair consisting of

- a comma-separated list of the field names
- an ordered dictionary keyed on constraint kind with the value specifying the constraint.

For simple constraints, the value is a base type; for more complex Constraints with several components, the value will itself be an (ordered) dictionary.

The ordering is all to make the JSON file get written in a sensible order, rather than being a jumbled mess.

class tdda.constraints.base.**Constraint**(*kind*, *value*, ***kwargs*)
Base container for a single constraint. All specific constraint types (should) subclass this.

check_validity(*name*, *value*, **valids*)
Check that the value of a constraint is allowed. If it isn't, then the TDDA file is not valid.

class tdda.constraints.base.**MinConstraint**(*value*, *precision=None*)
Constraint specifying the minimum allowed value in a field.

class tdda.constraints.base.**MaxConstraint**(*value*, *precision=None*)
Constraint specifying the maximum allowed value in a field.

class tdda.constraints.base.**SignConstraint**(*value*)
Constraint specifying allowed sign of values in a field. Used only for numeric fields (`real`, `int`, `bool`), and normally used in addition to Min and Max constraints.

Possible values are `positive`, `non-negative`, `zero`, `non-positive`, `negative` and `null`.

class tdda.constraints.base.**TypeConstraint**(*value*)
Constraint specifying the allowed (TDDA) type of a field. This can be a single value, chosen from:

- `bool`
- `int`
- `real`
- `string`
- `date`

or a list of such values, most commonly `['int', 'real']`, sometimes used because of Pandas silent and automatic promotion of integer fields to floats if nulls are present.)

class tdda.constraints.base.**MaxNullsConstraint**(*value*)
Constraint on the maximum number of nulls allowed in a field. Usually 0 or 1. (The constraint generator only generates 0 and 1, but the verifier will verify and number.)

class tdda.constraints.base.**NoDuplicatesConstraint**(*value=True*)
Constraint specifying that non dupicate non-null values are allowed in a field.

Currently only generated for string fields, though could be used more broadly.

class tdda.constraints.base.**AllowedValuesConstraint**(*value*)
Constraint restricting the allowed values in a field to an explicity list.

Currently only used for string fields.

When generating constraints, this code will only generate such a constraint if there are no more than `MAX_CATEGORIES` (= 20 at the time of writing, but check above in case this comment rusts) different values in the field.

class tdda.constraints.base.**MinLengthConstraint**(*value*)
Constraint restricting the minimum length of strings in a string field.

Generated instead of a `MinConstraint` by this generation code, but can be used in conjunction with a `MinConstraint`.

class tdda.constraints.base.**MaxLengthConstraint**(*value*)
Constraint restricting the maximum length of strings in a string field.

Generated instead of a `MaxConstraint` by this generation code, but can be used in conjunction with a `MinConstraint`.

class tdda.constraints.base.**LtConstraint**(*value*)
Constraint specifying that the first field of a pair should be (strictly) less than the second, where both are non-null.

class tdda.constraints.base.**LteConstraint**(*value*)
Constraint specifying that the first field of a pair should be no greater than the second, where both are non-null.

class tdda.constraints.base.**EqConstraint**(*value*)
Constraint specifying that two fields should have identical values where they are both non-null.

class tdda.constraints.base.**GtConstraint**(*value*)
Constraint specifying that the first field of a pair should be (strictly) greater than the second, where both are non-null.

class tdda.constraints.base.**GteConstraint**(*value*)
Constraint specifying that the first field of a pair should be greater than or equal to the second, where both are non-null.

class tdda.constraints.base.**RexConstraint**(*value*)
Constraint restricting a string field to match (at least) one of the regular expressions in a list given.

class tdda.constraints.base.**Verification**(*constraints*, *report='all'*, *ascii=False*, *detect=False*, *detect_outpath=None*, *detect_write_all=False*, *detect_per_constraint=False*, *detect_output_fields=None*, *detect_index=False*, *detect_in_place=False*, ***kwargs*)
Container for the result of a constraint verification for a dataset in the context of a given set of constraints.

5.7 TDDA JSON file format

A `.tdda` file is a JSON file containing a single JSON object of the form:

```
{
    "fields": {
        field-name: field-constraints,
        ...
    }
}
```

Each `field-constraints` item is a JSON object containing a property for each included constraint:

```
{
    "type": one of int, real, bool, string or date
    "min": minimum allowed value,
    "max": maximum allowed value,
    "min_length": minimum allowed string length (for string fields),
    "max_length": maximum allowed string length (for string fields),
    "max_nulls": maximum number of null values allowed,
    "sign": one of positive, negative, non-positive, non-negative,
    "no_duplicates": true if the field values must be unique,
    "values": list of distinct allowed values,
    "rex": list of regular expressions, to cover all cases
}
```

5.8 Examples

CHAPTER 6

Rexpy

Rexpy infers regular expressions on a line-by-line basis from text data examples.

To run the rexpy tool:

```
tdda rexpy [inputfile]
```

6.1 Command-line Tool

Usage:

```
rexpy [FLAGS] [input file [output file]]
```

If input file is provided, it should contain one string per line; otherwise lines will be read from standard input.

If output file is provided, regular expressions found will be written to that (one per line); otherwise they will be printed.

FLAGS are optional flags. Currently:

```
-h, --header      Discard first line, as a header.

-?, --help        Print this usage information and exit (without error)

-g, --group       Generate capture groups for each variable fragment
                  of each regular expression generated, i.e. surround
                  variable components with parentheses
                      e.g.     '^([A-Z]+)\-([0-9]+)$'
                      becomes  '^[A-Z]+\-[0-9]+$'

-u, --underscore  Allow underscore to be treated as a letter.
                  Mostly useful for matching identifiers
                  Also allow -_.

-d, --dot         Allow dot to be treated as a letter.
```

(continues on next page)

(continued from previous page)

```
                  Mostly useful for matching identifiers.
                  Also -. --period.

-m, --minus       Allow minus to be treated as a letter.
                  Mostly useful for matching identifiers.
                  Also --hyphen or --dash.

-v, --version     Print the version number.

-V, --verbose     Set verbosity level to 1

-VV, --Verbose    Set verbosity level to 2

-vlf, --variable  Use variable length fragments

-flf, --fixed     Use fixed length fragments
```

6.1.1 Python API

The *tdda.rexpy.rexpy* module also provides a Python API, to allow discovery of regular expressions to be incorporated into other Python programs.

class tdda.rexpy.rexpy.**Coverage**

Container for coverage information.

Attributes:

- `n`: number of matches
- `n_unique`: number matches, deduplicating strings
- `incr`: number of new (unique) matches for this regex
- `incr_uniq`: number of new (unique) deduplicated matches for this regex
- `index`: index of this regex in original list returned.

class tdda.rexpy.rexpy.**Extractor**(*examples*, *extract=True*, *tag=False*, *extra_letters=None*, *full_escape=False*, *remove_empties=False*, *strip=False*, *variableLengthFrags=False*, *specialize=False*, *max_patterns=None*, *min_diff_strings_per_pattern=1*, *min_strings_per_pattern=1*, *verbose=0*)

Regular expression 'extractor'.

Given a set of examples, this tries to construct a useful regular expression that characterizes them; failing which, a list of regular expressions that collectively cover the cases.

Results are stored in `self.results` once extraction has occurred, which happens by default on initialization, but can be invoked manually.

The examples may be given as a list or as a dictionary: if a dictionary, the values are assumed to be string frequencies.

Verbose is usually 0 or `False`. It can be to `True` or 1 for various extra output, and to higher numbers for even more verbose output. The highest level currently used is 2.

aligned_parts(*parts*)

Given a list of parts, each consisting of the fragments from a set of partially aligned patterns, show them aligned, and in a somewhat ambigous, numbered, fairly human-readable, compact form.

analyse_groups(*pattern*, *examples*)
Analyse the contents of each group (fragment) in pattern across the examples it matches.

Return zip of

- the characters in each group
- the strings in each group
- the run-length encoded fine classes in each group
- the run-length encoded characters in each group
- the group itself

all indexed on the (zero-based) group number.

batch_extract(*examples*)
Find regular expressions for a batch of examples (as given).

clean(*examples*)
Compute length of each string and count number of examples of each length.

coarse_classify(*s*)
Classify each character in a string into one of the coarse categories

coarse_classify_char(*c*)
Classify character into one of the coarse categories

coverage(*dedup=False*)
Get a list of frequencies for each regular expression, i.e the number of the (stripped) input strings it matches. The list is in the same order as the regular expressions in `self.results.rex`.

If `dedup` is set to `True`, shows only the number of distinct (stripped) input examples matches

extract()
Actually perform the regular expression 'extraction'.

find_non_matches()
Returns all example strings that do not match any of the regular expressions in results.

fine_class(*c*)
Map a character in coarse class 'C' (AlphaNumeric) to a fine class.

full_incremental_coverage(*dedup=False*, *debug=False*)
Returns an ordered dictionary of regular expressions, sorted by the number of new examples they match/explain, from most to fewest, with ties broken by pattern sort order. The values in the results dictionary are the numbers of (new) examples matched.

If `dedup` is set to `True`, frequencies are ignored.

Each result is a `Coverage` object with the following attributes:

n: number of examples matched including duplicates

n_uniq: number of examples matched, excluding duplicates

incr: number of previously unmatched examples matched, including duplicates

incr_uniq: number of previously unmatched examples matched, excluding duplicates

incremental_coverage(*dedup=False*, *debug=False*)
Returns an ordered dictionary of regular expressions, sorted by the number of new examples they match/explain, from most to fewest, with ties broken by pattern sort order. The values in the results dictionary are the numbers of (new) examples matched.

If `dedup` is set to `True`, frequencies are ignored.

merge_fixed_omnipresent_at_pos(*patterns*)
Find unusual columns in fixed positions relative to ends. Align those, split and recurse

merge_fixed_only_present_at_pos(*patterns*)
Find unusual columns in fixed positions relative to ends. Align those Split and recurse

n_examples(*dedup=False*)
Returns the total number of examples used by rexpy. If `dedup` is set to `True`, this the number of different examples, otherwise it is the "raw" number of examples. In all cases, examples have been stripped.

refine_groups(*pattern*, *examples*)
Refine the categories for variable run-length-encoded patterns provided by narrowing the characters in the groups.

rle2re(*rles*, *tagged=False*, *as_re=True*)
Convert run-length-encoded code string to regular expression

rle_fc_c(*s*, *pattern*, *rlefc_in*, *rlec_in*)

Convert a string, matching a 'C'-(fragment) pattern, to

- a run-length encoded sequence of fine classes
- a run-length encoded sequence of characters

Given inputs:

s — a string representing the actual substring of an example that matches a pattern fragment described by pattern

`pattern` — a VRLE of coarse classes

`rlefc_in` — a VRLE of fine classes, or None, or False

`rlec_in` — a VRLE of characters, or None, or False

Returns new rlefc and rlec, each of which is:

`False`, if the string doesn't match the corresponding input VRLE

a possibly expanded VRLE, if it does match, or would match if expanded (by allowing more of fewer repetitions).

run_length_encode_coarse_classes(*s*)
Returns run-length encoded coarse classification

sample(*nPerLength*)
Sample strings for potentially faster induction. Only used if over a hundred million distinct strings are given. For now.

specialize(*patterns*)
Check all the catpure groups in each patterns and simplify any that are sufficiently low frequency.

vrle2re(*vrles*, *tagged=False*, *as_re=True*)
Convert variable run-length-encoded code string to regular expression

vrle2refrags(*vrles*)
Convert variable run-length-encoded code string to regular expression and list of fragments

class tdda.rexpy.rexpy.**Fragment**
Container for a fragment.

Attributes:

- `re`: the regular expression for the fragment
- `group`: True if it forms a capture group (i.e. is not constant)

`tdda.rexpy.rexpy.`**`capture_group`**(*s*)
Places parentheses around s to form a capure group (a tagged piece of a regular expression), unless it is already a capture group.

`tdda.rexpy.rexpy.`**`cre`**(*rex*)
Compiled regular expression Memoized implementation.

`tdda.rexpy.rexpy.`**`expand_or_falsify_vrle`**(*rle*, *vrle*, *fixed=False*, *variableLength=False*)

Given a run-length encoded sequence (e.g. `[('A', 3), ('B', 4)]`)

and (usually) a variable run-length encoded sequence (e.g. `[('A', 2, 3), ('B', 1, 2)]`)

expand the VRLE to include the case of the RLE, if they can be consistent.

If they cannot, return False.

If vrle is None, this indicates it hasn't been found yet, so rle is simply expanded to a VRLE.

If vrle is False, this indicates that a counterexample has already been found, so False is returned again.

If variableLength is set to True, patterns will be merged even if it is a different length from the vrle, as long as the overlapping part is consistent.

`tdda.rexpy.rexpy.`**`extract`**(*examples*, *tag=False*, *encoding=None*, *as_object=False*, *extra_letters=None*, *full_escape=False*, *remove_empties=False*, *strip=False*, *variableLengthFrags=False*, *max_patterns=None*, *min_diff_strings_per_pattern=1*, *min_strings_per_pattern=1*, *verbose=0*)
Extract regular expression(s) from examples and return them.

Normally, examples should be unicode (i.e. `str` in Python3, and `unicode` in Python2). However, encoded strings can be passed in provided the encoding is specified.

Results will always be unicode.

If as_object is set, the extractor object is returned, with results in .results.rex; otherwise, a list of regular expressions, as unicode strings is returned.

`tdda.rexpy.rexpy.`**`get_omnipresent_at_pos`**(*fragFreqCounters*, *n*, ***kwargs*)
Find patterns in `fragFreqCounters` for which the frequency is `n`.

fragFreqCounters is a dictionary (usually keyed on 'fragments') of whose values are dictionaries mapping positions to frequencies.

For example:

```
{
    ('a', 1, 1, 'fixed'): {1: 7, -1: 7, 3: 4},
    ('b', 1, 1, 'fixed'): {2: 6, 3: 4},
}
```

This indicates that the pattern `('a', 1, 1, 'fixed')` has frequency 7 at positions 1 and -1, and frequency 4 at position 3, while pattern `('b', 1, 1, 'fixed')` has frequency 6 at position 2 and 4 at position 3.

With n set to 7, this returns:

```
[
    (('a', 1, 1, 'fixed'), -1)
```

(continues on next page)

(continued from previous page)

```
    (('a', 1, 1, 'fixed'), 1),
]
```

(sorted on pos; each pos really should occur at most once.)

tdda.rexpy.rexpy.**get_only_present_at_pos**(*fragFreqCounters*, **args*, ***kwargs*)
Find patterns in fragFreqCounters that, when present, are always at the same position.

`fragFreqCounters` is a dictionary (usually keyed on `fragments`) of whose values are dictionaries mapping positions to frequencies.

For example:

```
{
    ('a', 1, 1, 'fixed'): {1: 7, -1: 7, 3: 4},
    ('b', 1, 1, 'fixed'): {2: 6},
}
```

This indicates that the

- pattern `('a', 1, 1, 'fixed')` has frequency 7 at positions 1 and -1, and frequency 4 at position 3;
- pattern `('b', 1, 1, 'fixed')` has frequency 6 at position 2 (only)

So this would return:

```
[
    (('b', 1, 1, 'fixed'), 2)
]
```

(sorted on `pos`; each `pos` really should occur at most once.)

tdda.rexpy.rexpy.**left_parts**(*patterns*, *fixed*)
patterns is a list of patterns each consisting of a list of frags.

fixed is a list of `(fragment, position)` pairs, sorted on position, specifying points at which to split the patterns.

This function returns a list of lists of pattern fragments, split at each fixed position.

tdda.rexpy.rexpy.**length_stats**(*patterns*)
Given a list of patterns, returns named tuple containing

all_same_length: boolean, True if all patterns are the same length

max_length: length of the longest pattern in patterns

tdda.rexpy.rexpy.**matrices2incremental_coverage**(*patterns*, *matrix*, *deduped*, *indexes*, *example_freqs*, *dedup=False*)
Find patterns, in order of # of matches, and pull out freqs. Then set overlapping matches to zero and repeat. Returns ordered dict, sorted by incremental match rate, with number of (previously unaccounted for) strings matched.

tdda.rexpy.rexpy.**pdextract**(*cols*)
Extract regular expression(s) from the Pandas column (`Series`) object or list of Pandas columns given.

All columns provided should be string columns (i.e. of type np.dtype('O'), possibly including null values, which will be ignored.

Example use:

```
import pandas as pd
from tdda.rexpy import pdextract

df = pd.DataFrame({'a3': ["one", "two", pd.np.NaN],
                   'a45': ['three', 'four', 'five']})

re3 = pdextract(df['a3'])
re45 = pdextract(df['a45'])
re345 = pdextract([df['a3'], df['a45']])
```

This should result in:

```
re3   = '^[a-z]{3}$'
re5   = '^[a-z]{3}$'
re345 = '^[a-z]{3}$'
```

tdda.rexpy.rexpy.**rex_coverage**(*patterns*, *example_freqs*, *dedup=False*)

Given a list of regular expressions and a dictionary of examples and their frequencies, this counts the number of times each pattern matches a an example.

If `dedup` is set to `True`, the frequencies are ignored, so that only the number of keys is returned.

tdda.rexpy.rexpy.**rex_full_incremental_coverage**(*patterns*, *example_freqs*, *dedup=False*, *debug=False*)

Returns an ordered dictionary containing, keyed on terminated regular expressions, from patterns, sorted in decreasing order of incremental coverage, i.e. with the pattern matching the most first, followed by the one matching the most remaining examples etc.

If `dedup` is set to `True`, the ordering ignores duplicate examples; otherise, duplicates help determine the sort order.

Each entry in the dictionary returned is a `Coverage` object with the following attributes:

n: number of examples matched including duplicatesb

n_uniq: number of examples matched, excluding duplicates

incr: number of previously unmatched examples matched, including duplicates

incr_uniq: number of previously unmatched examples matched, excluding duplicates

tdda.rexpy.rexpy.**rex_incremental_coverage**(*patterns*, *example_freqs*, *dedup=False*, *debug=False*)

Given a list of regular expressions and a dictionary of examples and their frequencies, this computes their incremental coverage, i.e. it produces an ordered dictionary, sorted from the "most useful" patterns (the one that matches the most examples) to the least useful. Usefulness is defined as "matching the most previously unmatched patterns". The dictionary entries are the number of (new) matches for the pattern.

If `dedup` is set to `True`, the frequencies are ignored when computing match rate; if set to false, patterns get credit for the nmultiplicity of examples they match.

Ties are broken by lexical order of the (terminated) patterns.

For example, given patterns p1, p2, and p3, and examples e1, e2 and e3, with a match profile as follows (where the numbers are multiplicities)

example	p1	p2	p3
e1	2	2	0
e2	0	3	3
e3	1	0	0
e4	0	0	4
e5	1	0	1
TOTAL	4	4	8

If `dedup` is `False` this would produce:

```
OrderedDict(
    (p3, 8),
    (p1, 3),
    (p2, 0)
)
```

because:

- p3 matches the most, with 8
- Of the strings unmatched by p3, p1 accounts for 3 (e1 x 2 and e3 x 1) whereas p2 accounts for no new strings.

With `dedup` set to `True`, the matrix transforms to

example	p1	p2	p3
e1	1	1	0
e2	0	1	1
e3	1	0	0
e4	0	0	1
e5	1	0	1
TOTAL	3	2	3

So p1 and p3 are tied.

If we assume the p1 sorts before p3, the result would then be:

```
OrderedDict(
    (p1, 3),
    (p3, 2),
    (p2, 0)
)
```

tdda.rexpy.rexpy.**right_parts**(*patterns*, *fixed*)

patterns is a list of patterns each consisting of a list of frags.

fixed is a list of `(fragment, pos)` pairs where position specifies the position from the right, i.e a position that can be indexed as `-position`.

Fixed should be sorted, increasing on position, i.e. sorted from the right-most pattern. The positions specify points at which to split the patterns.

This function returns a list of lists of pattern fragments, split at each fixed position.

tdda.rexpy.rexpy.**run_length_encode**(*s*)

Return run-length-encoding of string s, e.g.:

```
'CCC-BB-A' --> (('C', 3), ('-', 1), ('B', 2), ('-', 1), ('A', 1))
```

tdda.rexpy.rexpy.**signature**(*rle*)
Return the sequence of characters in a run-length encoding (i.e. the signature).

Also works with variable run-length encodings

tdda.rexpy.rexpy.**terminate_patterns_and_sort**(*patterns*)
Given a list of regular expressions, this terminates any that are not and returns them in sorted order. Also returns a list of the original indexes of the results.

tdda.rexpy.rexpy.**to_vrles**(*rles*)
Convert a list of run-length encodings to a list of variable run-length encodings, one for each common signature.

For example, given inputs of:

```
    (('C', 2),)
    (('C', 3),)
and (('C', 2), ('.', 1))
```

this would return:

```
    (('C', 2, 3),)
and (('C', 2, 2), ('.', 1, 1))
```

6.2 Examples

The *tdda.rexpy* module includes a set of examples.

To copy these examples to your own *rexpy-examples* subdirectory (or to a location of your choice), run the command:

```
tdda examples rexpy [mydirectory]
```

CHAPTER 7

Tests

The TDDA package includes a set of unit-tests, for testing that the package is correctly installed and configured, and does not include any regressions.

To run these tests:

```
tdda test
```

The output should look something like:

```
.....................................................................s....
.........................s.........................
----------------------------------------------------------------------
Ran 120 tests in 1.849s

OK (skipped=2)
```

Some tests may be skipped, if they depend on modules that are not installed in your local environment (for instance, for testing TDDA database functionality for databases for which you do not have drivers installed).

The overall test status should always be **OK**.

CHAPTER 8

Examples

The TDDA package includes embedded examples of code and data.

To copy these examples to a directory of your choice (or, if you don't specify a location, then to the current directory), run:

```
tdda examples [mydirectory]
```

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Python Module Index

t

Index

D

E

F

G

T

V

W