tdda Documentation

Release 2.0.01

Stochastic Solutions Limited

Feb 24, 2022

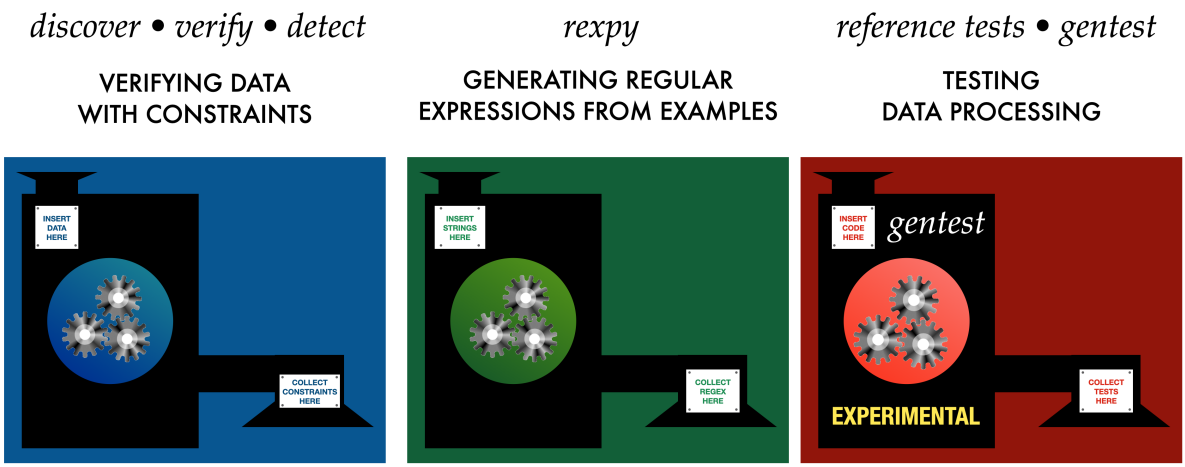
Contents

Version 2.0.01. (Installation)

The TDDA module helps with the testing of data and of code that manipulates data. It serves as a concrete implementation of the ideas discussed on the test-driven data analysis blog. When installed, the module offers a suite of command-line tools that can be used with data from any source, not just Python. It also provideds enhanced test methods for Python code, and the new *Gentest* functionality enables automatic generation of test programs for arbitrary code (not just Python code). There is also a full Python API for all functionality.

Test-driven data analysis is closely related to reproducible research, but with more of a focus on automated testing. It is best seen as overlapping and partly complementary to reproducible research.

The major components of the TDDA module are:



- **Automatic Constraint Generation and Verification:** The package includes command-line tools and API calls for
  - *discovery* of constraints that are satisified by (example) data — `tdda discover`;
  - *verification* that a dataset satisfies a set of constraints. The constraints can have been generated automatically, constructed manually, or (most commonly) consist of generated constraints that have been subsequently refined by hand — `tdda verify`;
  - *detection* of records, fields and values that fail to satisify constraints (anomaly detection) — `tdda detect`.
- **Reference Testing:** The TDDA library offers extensions to `unittest` and `pytest` for managing the testing of data analysis pipelines, where the results are typically much larger, and more complex, and more variable than for many other sorts of programs.
- **Automatic Generation of Regular Expressions from Examples:** There is command-line tool (and API) for automatically inferring regular expressions from (structured) textual data — `rexpy`. This was developed as part of constraint generation, but has broader utility.
- **Automatic Test Generation (Experimental):** From version 2.0 on, the TDDA library also includes experimental features for automatically generating tests for almost any command-line based program or script. The code to be tested can take the form of a shell script or any other command-line code, and can be written in any language or mix of languages.

CHAPTER 1

Contents

1.1 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. It also supports tables in a variety of relation databases. There is also a command-line utility for discovering and verifying constraints, and detecting failing records.
- 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.

Although the library is provided as a Python package, and can be called through its Python API, it also provides command-line tools.

1.2 Installation

If you don't need source, the simplest way to install the TDDA library is to do a normal `pip` install:

```
pip install tdda
```

If you have multiple Python installations and want to ensure you use the right one, use:

```
python -m pip install tdda
```

replacing `python` with whatever you need to use to run your target version of Python.

If you want it installed for all users and don't have write access to your Python's site packages, you should change the permissions so that you can write it, or log in as a user who does have permission, or add `sudo` before the `pip` or `python` command.

1.2.1 Upgrading

Add `--upgrade` or `-U` after `install` in the commands above to upgrade an existing installation. This is a general pattern for `pip`:

```
pip install -U tdda
```

or:

```
python -m pip install -U tdda
```

1.2.2 Source installation

The tdda project is hosted on Github at github.com/tdda/tdda.

If you want to do a source installation, that's probably all you need to know, but:

```
git clone git@github.com:tdda/tdda.git
```

or:

```
git clone https://github.com/tdda/tdda.git
```

is the command to get it.

Then:

```
python setup.py install
```

should install it.

1.2.3 Checking the installation

If all has gone well, you should be able to type:

```
tdda
```

and it will show you some help.

You should also be able to use:

```
>>> import tdda
>>> tdda.__version__
```

from your Python successfully.

Finally, you should be able to run the tests with no failures, like this for example:

```
cd tdda/tdda
python testtdda.py
Skipping Postgres tests (no driver library)
Skipping MySQL tests (no driver library)
.....................................................................................
 ↪..............sssssssss.....................................................................
 ↪........sssss.............................
----------------------------------------------------------------------
Ran 213 tests in 18.421s

OK (skipped=15)
```

Some tests will be skipped (`s`) if you don't have various libraries or haven't (yet) told TDDA about any databases you might want to use.

1.2.4 Optional Installations for using Databases, Feather Files, Pandas

Extra libraries are required to access some of the constraint-generation and verification functionality, depending on the data sources that you wish to use.

- `pandas` (required for CSV files and `feather` files)
- `feather-format` (required for `feather` files)
- `pmmif` (makes `feather` file reading and writing more robust)
- `pygresql` (required for PostgreSQL database tables)
- `MySQL-python` or `mysqlclient` or `mysql-connector-python` (required for MySQL database tables)
- `pymongo` (required for MongoDB document collections)

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

```
pip install pandas
pip install feather-format
pip install pmmif
pip install pygresql
pip install pymongo
```

and, for MySQL, **one** of:

```
pip install MySQL-python
pip install mysqlclient
pip install mysql-connector-python
```

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.

1.3 Automatic Constraint Generation, Data Verification & Anomaly Detection

The TDDA library provides support for constraint generation, verification and anomaly detection for datasets, including `.csv` files and Pandas DataFrames.

The module includes:

- *The tdda Command-line Tool* for discovering constraints in data, and for verifying data against those constraints, using the *TDDA JSON file format* (`.tdda` files).
- A Python `constraints` library 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:`)

Note: To use databases, `pandas`, `.feather` files etc. you may need to install extra optional packages. See *Optional Installations for using Databases, Feather Files, Pandas*.

1.3.1 The `tdda` Command-line Tool

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

It also provides some other functionality to help with using the tool. The following command forms are supported:

- *tdda discover* —perform constraint discovery.
- *tdda verify* — verify data against constraints.
- *tdda detect* — detect anomalies in data by checking constraints.
- `tdda examples` — generate example data and code.
- `tdda help` — show help on how to use the tool.
- `tdda test` — 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.

1.3.2 `tdda discover`

The `tdda discover` command can generate constraints for data, and save the generated constraints as a *TDDA JSON file format* file (`.tdda`).

Usage:

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

- `input` is one of:

  - a `.csv` file
  - a `-`, meaning that a `.csv` file should be read 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 if `-` is used, the constraints are written to standard output (`stdout`).

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.

1.3.3 `tdda verify`

The `tdda verify` command is used to validate data from various sources, against constraints from a *TDDA JSON file format* constraints file.

Usage:

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

- `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 a `.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.

1.3.4 tdda detect

The `tdda detect` command is used to detect anomalies on data, by checking against constraints from a *TDDA JSON file format* constraints file.

Usage:

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

- `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.

1.3.5 Constraints for CSV Files and Pandas DataFrames

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`

1.3.6 Constraints for Databases

When a database table is used with the any `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
```

The following example will use the file `.tdda_db_conn_postgres` from your home directory (see *Database Connection Files*), providing all of the default parameters for the database connection.

```
tdda discover postgres:mytable
tdda discover postgres:myschema.mytable
```

For MongoDB, document collections are used instead of database tables, and a document can be referred to at any level in the collection structure. Only scalar properties are used for constraint discovery and verification (and any deeper nested structure is ignored). For example:

```
tdda discover mongodb:mydocument
tdda discover mongodb:subcollection.mysubdocument
```

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 (where *DBTYPE* is the name of the kind of database server).

To use constraints for databases, you must have an appropriate DB-API (PEP-0249) driver library installed within your Python environment.

These are:

- For PostgreSQL: `pygresql` or `PyGreSQL`
- For MySQL: `MySQL-python`, `mysqlclient` or `mysql-connector-python`
- For SQLite: `sqlite3`
- For MongoDB: `pymongo`

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 an appropriate `DBTYPE:` (one of the supported kinds of database server, such as `postgres`).

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,
}
```

Some additional notes:

- 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.
- For MySQL (in a `.tdda_db_conn_mysql` file), the `schema` parameter **must** be specified, as there is no built-in default for it to use.
- For Microsoft Windows, the connector file should have the same name as for Unix, beginning with a dot, even though this form of filename is not otherwise commonly used on Windows.

1.3.7 TDDA JSON file format

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

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

(continues on next page)

(continued from previous page)

```
        ...
    }
}
```

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
}
```

1.3.8 Constraints Examples

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

To copy these constraints examples, run the command:

```
tdda examples constraints [directory]
```

If `directory` is not supplied, `constraints_examples` will be used.

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

```
tdda examples
```

which will create a number of separate subdirectories.

1.4 Rexpy

1.4.1 The `rexpy` command

```
rexpy [FLAGS] [inputfile [outputfile]]
```

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

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

Optional `FLAGS` may be used to modify Rexpy's behaviour:

-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]+)$
```

-q, --quote Display the resulting regular expressions as double-quoted, escaped strings, in a form broadly suitable for use in Unix shells, JSON, and string literals in many programming languages. e.g.

```
^[A-Z]+\-[0-9]+$
```

becomes

```
"^[A-Z]+\\-[0-9]+$"
```

--portable Product maximally portable regular expressions (e.g. `[0-9]` rather than `\d`). (This is the default.)

--grep Same as `--portable`

--java Produce Java-style regular expressions (e.g. `\p{Digit}`)

--posix Produce POSIX-compilant regular expressions (e.g. `[[:digit:]]` rather than `\d`).

--perl Produce Perl-style regular expressions (e.g. `\d`)

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

-d, --dot, -., --period Allow dot to be treated as a letter. Mostly useful for matching identifiers.

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

-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

1.4.2 Rexpy Examples

TDDA rexpy is supplied with a set of examples.

To copy the rexpy examples, run the command:

```
tdda examples rexpy [directory]
```

If `directory` is not supplied, `rexpy_examples` will be used.

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

```
tdda examples
```

which will create a number of separate subdirectories.

1.5 Gentest: Automatic Test Generation for Unix & Linux Commands/Scripts

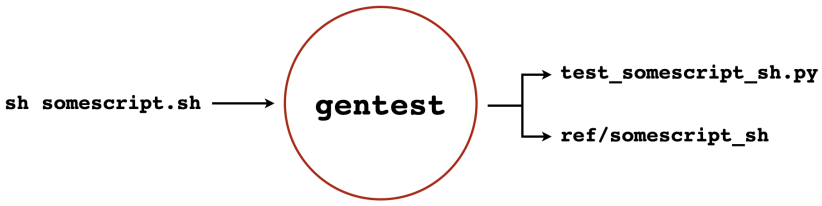
Gentest automatically generates tests for shell scripts and command-line programs on Unix and Linux.

It can currently test

- files created
- screen output (to `stdout` and `stderr`)
- exit codes and error states

and has some ability to handle run-to-run variation that occur in correctly running code.

The general model is shown below:



As shown, in simple cases, Gentest simply takes a command as input, and generates a Python test program, together with any required reference outputs in a subdirectory. The test script runs the command and executes a number of tests to see if it behaves as expected.

1.5.1 The Big Idea

The key assumption Gentest makes is that the code you give it is running correctly when you run the `tdda gentest` command. The tests Gentest creates don't really test that code is *correct*; merely that it its behaviour is consistent and doesn't generate error states. This is what we mean when we talk about *reference tests*: test that processes with known, believed correct *reference* outputs continue to operate as expected.

"Consistent" doesn't need to mean *identical every time*. Gentest runs code more than once, and tries to cater for variations it sees and things that look like non-portable aspects of your environment.

1.5.2 Running Gentest

The simplest way to run Gentest is to use the wizard, which will prompt you with a series of questions:

```
$ tdda gentest
```

In simple cases, where you want the default behaviour (see below) you can add the command that you want to test to skip the wizard:

```
$ tdda gentest 'command to test'
```

The command can be anything that you can run from the command line—a simple Unix command, a shell script, or a compiled or interpreted program, optionally with parameters.

The full syntax (which is mostly used programmatically, or by copying what Gentest writes into its output files, when you want to regenerate a set of tests after changing the code) is:

```
$ tdda gentest [FLAGS] 'command to test' [test_output.py [FILES]]
```

See *Gentest Parameters and Options* for full details of available flags and parameters.

1.5.3 Gentest Examples

TDDA gentest is supplied with a set of examples.

To copy the gentest examples, run the command:

```
tdda examples gentest [directory]
```

If `directory` is not supplied, `gentest_examples` will be used.

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

```
tdda examples
```

which will create a number of separate subdirectories.

The first few examples are rather trivial. Go straight to the worked Python Examples or R Examples if you want more excitement and realism straight away.

1.5.4 Example 1: Hey, cats! (not using Wizard)

We start with a purely illustrative example—using Gentest to create a test that checks the behaviour of the `cat` command on a known file:

```
$ echo 'Hey, cats!' > hey
$ tdda gentest 'cat hey'

Running command 'cat hey' to generate output (run 1 of 2).
Saved (non-empty) output to stdout to /home/tdda/tmp/ref/cat_hey/STDOUT.
Saved (empty) output to stderr to /home/tdda/tmp/ref/cat_hey/STDERR.

Running command 'cat hey' to generate output (run 2 of 2).
Saved (non-empty) output to stdout to /home/tdda/tmp/ref/cat_hey/2/STDOUT.
Saved (empty) output to stderr to /home/tdda/tmp/ref/cat_hey/2/STDERR.

Test script written as /home/tdda/tmp/test_cat_hey.py
Command execution took: 0.012s

SUMMARY:

Directory to run in:        /home/tdda/tmp
Shell command:              cat hey
Test script generated:      test_cat_hey.py
Reference files (none):
Check stdout:               yes (was 'Hey, cats!\n')
Check stderr:               yes (was empty)
Expected exit code:         0
Clobbering permitted:       yes
Number of times script ran: 2

$ python test_cat_hey.py
```

(continues on next page)

(continued from previous page)

```
....
----------------------------------------------------------------------
Ran 4 tests in 0.008s

OK
```

For this illustration, we first create a file called `hey` containing the text `Hey, cats!`, which we will feed to the `cat` command (to display it in the terminal).

We run `tdda gentest 'cat hey'` to generate the test. It is a good idea to enclose the command in single quotes, and this may be necessary if it includes spaces or special characters. If the command itself uses single quotes, normal shell rules apply and they will need to be escaped. (It's generally easier to use the wizard in such cases.)

Gentest then runs the command we specified a number of times—2 by default.

Gentest finishes by displaying a summary of what it did. It names the script automatically, based on a sanitized version of the command—in this case `test_hey_cats.py`

Finally, we run the test script generated, and in this case it reports that it has run four tests, all of which passed. In this case, the four tests (which you can obviously see by looking in the test script) are:

- check that the output written to the screen `Hey, cats!` (on `stdout`) was as expected.
- check that there was no output written to `stderr`, since no output was written to `stderr` when it ran the code to generate the tests
- check that the exit code returned by the command was `0`. (On Unix and Linux systems, programs return a numeric code, which should be zero if the program completes normally, and a (small) non-zero number if it finishes abnormally, with different codes indicating different issues.
- check that the program did not crash.

1.5.5 The Generated Test Code

We'll walk through the core of the generated test code briefly. Obviously, you can look at all of it: it's right there in `test_hey_cats.py`, but won't bother with the boilerplate.

The core of the generated code is a test class (subclassing `tdda.ReferenceTestCase`, which inherits from `unittest.TestCase`), with a `setUpClass` class method that runs the command and assigns class attributes for the `output` to `stdout`, the `error` output to `stderr`, any exeception (`exc`) that occurred, the exit code (`exit_code`) from the command and the wall-clock time it took to run (`duration`, in seconds):

```
@classmethod
def setUpClass(cls):

    (cls.output,
     cls.error,
     cls.exception,
     cls.exit_code,
     cls.duration) = exec_command(cls.command, cls.cwd)
```

When the code doesn't generate any files that need to be checked, there's then a single test for each of the four checks mentioned above:

- First, there was no exception (in which case, `self.exc` will be `None`):

```
def test_no_exception(self):
    self.assertIsNone(self.exception)
```

- Then comes the check that the exit code was zero:

```
def test_exit_code(self):
    self.assertEqual(self.exit_code, 0)
```

- After this, there's a test to check the normal output to `sys.stdout`. The reference output has been saved to `./ref/cat_hey/STDOUT`, and `self.refdir` is set (further up in the script) to `./ref/cat_hey/` (conceptually):

```
def test_stdout(self):
    self.assertStringCorrect(self.output,
                             os.path.join(self.refdir, 'STDOUT'))
```

  The `assertStringCorrect` method, with no extra parameters, compares the first value (the output, stored in self.output, as a literal string, in this case) with the reference output, in the file. But the method does more than that, including accepting extra parameters to control the comparison, capturing output to file when the strings don't match, and re-writing the reference if so instructed.

- The last is the counterpart the `stdout` check, this time instead checking `stderr`. Since there was no output to stderr, in this case it's actually just checking that there was no output:

```
def test_stderr(self):
    self.assertStringCorrect(self.error,
                             os.path.join(self.refdir, 'STDERR'))
```

1.5.6 Test Failures

Let's look at what happens in a few error and test cases. Using the same test code as before, let's change the output by changing what's in the file `hey`:

```
$ echo 'Ho, cats!' > hey
$ python test_cat_hey.py
...    1 line is different, starting at line 1
Expected file /home/tdda/tmp/ref/cat_hey/STDOUT
Compare raw with:
    diff /var/folders/tx/z752bv1x6qx8swpncq8qg5mm0000gp/T/actual-raw-STDOUT /home/
↪tdda/tmp/ref/cat_hey/STDOUT

Compare post-processed with:
    diff /var/folders/tx/z752bv1x6qx8swpncq8qg5mm0000gp/T/actual-STDOUT /var/folders/
↪tx/z752bv1x6qx8swpncq8qg5mm0000gp/T/expected-STDOUT

F
======================================================================
FAIL: test_stdout (__main__.TestCAT_)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_cat_hey.py", line 48, in test_stdout
    os.path.join(self.refdir, 'STDOUT'))
AssertionError: False is not true : 1 line is different, starting at line 1
Expected file /home/tdda/tmp/ref/cat_hey/STDOUT
Compare raw with:
    diff /var/folders/tx/z752bv1x6qx8swpncq8qg5mm0000gp/T/actual-raw-STDOUT /home/
↪tdda/tmp/ref/cat_hey/STDOUT

Compare post-processed with:
```

(continues on next page)

(continued from previous page)

```
    diff /var/folders/tx/z752bv1x6qx8swpncq8qg5mm0000gp/T/actual-STDOUT /var/folders/
↪tx/z752bv1x6qx8swpncq8qg5mm0000gp/T/expected-STDOUT


----------------------------------------------------------------------
Ran 4 tests in 0.010s

FAILED (failures=1)
```

The output is a bit verbose, and when there are failures, these tend to be shown twice (as here), but the main things to note are:

The failure that has occured is with `STDOUT`, i.e. the messages sent to the screen on the normal output stream (as opposed to the error stream).

Gentest tells you how to examine the differences between the actual output and the output it expected with using a `diff` command. The actual output has been saved to a file called `actual-raw-STDOUT` in a temporary directory, and Gentest stored the output it expected for `STDOUT` in `ref/cat_hey/STDOUT`. If you have a visual diff tool (such as `opendiff` on the Mac, or `meld` on Linux, or `vdiff` on Unix), you can probably just replace `diff` with your preferred tool.

If we run the command suggested, we get:

```
$ diff /var/folders/tx/z752bv1x6qx8swpncq8qg5mm0000gp/T/actual-raw-STDOUT /home/tdda/
↪tmp/ref/cat_hey/STDOUT
1c1
< Ho, cats!
---
> Hey, cats!
```

Here we can see that the change from Hey to Ho has been picked up. (You may see a claim that one of the files doesn't end with a newline; that's a bug.)

Gentest also suggests a command for comparing post-processed versions of output. This isn't relevant in this case, but in cases later (when Gentest decides that the output is not completely fixed) where this is useful.

1.5.7 Updating the reference outputs if the new behaviour is correct

If the new behaviour is in fact correct, there are several ways to update the test with the new results.

- You can copy the first `diff command` (the one for the raw output) and replace the `diff` with `cp`. This overwrites the reference output and should all the failing test to pass:

  ```
  $ cp /var/folders/tx/z752bv1x6qx8swpncq8qg5mm0000gp/T/actual-raw-STDOUT /home/
  ↪tdda/tmp/ref/cat_hey/STDOUT
  $ python test_cat_hey.py
   ....
  ----------------------------------------------------------------------
  Ran 4 tests in 0.010s

  OK
  ```

- Alternatively, you can re-run the test script, adding `--write-all` This will rewrite *all* reference outputs with whatever the script produces. If you only want to update a single test, you can use the `@tag` decorator to tag it and the use the `-1` flag:

```
$ python test_cat_hey.py --write-all
..Written /home/tdda/tmp/ref/cat_hey/STDERR
.Written /home/tdda/tmp/ref/cat_hey/STDOUT
.
----------------------------------------------------------------------
Ran 4 tests in 0.010s

OK
$ python test_cat_hey.py
....
----------------------------------------------------------------------
Ran 4 tests in 0.008s

OK
```

- Finally, of course, you can simply rerun `tdda gentest` to completely regenerate the entire test script and reference outputs. Test files generated by gentest contain the command needed to regenerate the the same way as before, near the top, even if the Wizard was originally used. For example, the top of `test_cat_hey.py` is:

```
  """
  test_cat_hey.py: Automatically generated test code from tdda gentest.

  Generation command:

    tdda gentest 'cat hey' 'test_cat_hey.py' '.'
  """

In this case, this is the same we used, except that this is explicitly
specifying the name of the test script and that `Gentest` should look
to see whether any files are created in the current directory `.`
when it runs.
```

1.5.8 The Gentest Wizard

To run Gentest's wizard (recommended in most cases), use:

```
tdda gentest
```

You will then be prompted with a series of questions/directions. Most questions have suggested default answers, which you can accept by just hitting the `RETURN` (enter) key. The questions will be:

- `Enter shell command to be tested:` Here is when you give the full command you want tested, e.g. `sh example2.sh`. You don't need to quote the command, even if it has spaces or special characters, which makes it easier if you need quotes in the command or its parameters.
- `Enter name for test script [test_sh_example2_sh]:` The default name is a sanitized version of your command, and may become long if your command is complicated, so you might wish to choose a shorter name. Normally, you should start the name with `test` and give it a `.py` extension. Note: any existing file of the chosen name will be silently overwritten unless you use the `--noclobber` flag.
- `Check all files written under $(pwd)?: [y]:` Ordinarily, Gentest will watch to see whether your code writes any files in the current directory, or its subdirectories, and will use those as *reference* outputs, i.e. treat them as files to be checked. Answer `n`) if you don't want this to be done.
- `Check all files written under (gentest's) $TMPDIR?: [y]:` Also by default, gentest will look for any files written to `$TMPDIR`, if that shell variable is set, or the to the system's temporary di-

rectory (usually, `/tmp`), if it is not. Say `n` if you don't want gentest to look at files written to the temporary directory.

- `Enter other files/directories to be checked, one per line, then a blank line:` You can specify other locations gentest should watch for files. It's best not to make this a very high level directory (especially not `/`, as this will be very slow), but if there's a location your code is writing to, tell Gentest if you would like those files checked.
- `Check stdout?: [y]:` Gentest normally captures output to the screen, and checks that. On Unix and Linux systems, this output is split between ordinary output, which goes to `stdout` (file descriptor `1`) and `stderr` (file descriptor `2`), which is normally reserved for errors and warnings. If you don't want Gentest to check `stdout`, say `n`.
- `Check stderr?: [y]:` Again, if you don't want the standard error stream to be checked, say no to this question.
- `Exit code should be zero?: [y]:` All programs on Unix and Linux return an exit status, which is a numeric value between 0 and 255. Well-behaved programs return a `0` exit status to indicate success or normal functioning, and non-zero exit statuses to indicate different error conditions. Gentest normally includes a test of the exit status that fails if it is non-zero, and declines to generate tests if a non-zero exit status is returned when it is running the command for initial test generation. You can override this behaviour by sauing `n` to to this question.
- `Clobber (overwrite) previous outputs (if they exists)?: [y]:` By default, Gentest will overwrite any previous test script (of the same name) and corresponding reference outputs when run. If you say `n`, it will fail if the output test script or reference directory already exist.
- `Number of times to run script?: [2]:` As noted above, Gentest does not require that commands behave identically every time. While its capabilities are necessarily limited (as a minimally artificially intelligent™ system) Gentest attempts to recognize a limited range of correct behaviours by virtue of:
  - running tests multiple times
  - noting various aspects of the environment that may affect results.

  By default, Gentest runs scripts just twice, to gauge run-to-run variation, but results can sometimes be made more robust by increasing this number.

1.5.9 Example 2: Using the Gentest Wizard

For this example, we'll use Gentest with the following shell script. `example2.sh`, in the current directory:

```
echo "Hey, cats!"
echo
echo "This is gentest, running on `hostname`"
echo
echo "I have to say, the weather was better in Münich!"
echo
echo "Today, `date` it's proper dreich here."
echo
echo "Let's have a file as well." > FILE1
echo "Have a number: $RANDOM" >> FILE1
echo "Random number written to $PWD/FILE1"
```

You can get this script by typing `tdda examples`. This will generate a few directories in the directory you are in, including `gentest` which contains this script. Either change to that directory or copy `example2.sh` up a level.

We'll use the Gentest wizard, accepting the defaults suggestions after specifying `sh example2.sh` as the command to be tested, except that we'll ask gentest to run the script `10` times (for reasons we'll see below):

```
$ tdda gentest
Enter shell command to be tested: sh example2.sh
Enter name for test script [test_sh_example2_sh]:
Check all files written under $(pwd)?: [y]:
Check all files written under (gentest's) $TMPDIR?: [y]:
Enter other files/directories to be checked, one per line, then a blank line:

Check stdout?: [y]:
Check stderr?: [y]:
Exit code should be zero?: [y]:
Clobber (overwrite) previous outputs (if they exist)?: [y]:
Number of times to run script?: [2]: 10

Running command 'sh example2.sh' to generate output (run 1 of 10).
Saved (non-empty) output to stdout to /home/tdda/tmp/ref/sh_example2_sh/STDOUT.
Saved (empty) output to stderr to /home/tdda/tmp/ref/sh_example2_sh/STDERR.
Copied $(pwd)/FILE1 to $(pwd)/ref/sh_example2_sh/FILE1

Running command 'sh example2.sh' to generate output (run 2 of 10).
Saved (non-empty) output to stdout to /home/tdda/tmp/ref/sh_example2_sh/2/STDOUT.
Saved (empty) output to stderr to /home/tdda/tmp/ref/sh_example2_sh/2/STDERR.
Copied $(pwd)/FILE1 to $(pwd)/ref/sh_example2_sh/2/FILE1
  .
  .
  .
Running command 'sh example2.sh' to generate output (run 10 of 10).
Saved (non-empty) output to stdout to /home/tdda/tmp/ref/sh_example2_sh/STDOUT.
Saved (empty) output to stderr to /home/tdda/tmp/ref/sh_example2_sh/STDERR.

Copied $(pwd)/FILE1 to $(pwd)/ref/sh_example2_sh/10/FILE1

Test script written as /home/tdda/tmp/test_sh_example2_sh.py
Command execution took: 0.021s

SUMMARY:

Directory to run in:        /home/tdda/tmp
Shell command:              sh example2.sh
Test script generated:      test_sh_example2_sh.py
Reference files:
    $(pwd)/FILE1
Check stdout:               yes (was 9 lines)
Check stderr:               yes (was empty)
Expected exit code:         0
Clobbering permitted:       yes
Number of times script ran: 10
```

Notice that in this case:

Gentest noticed that `FILE1` was written, in the current working directory

Gentest reports that nine lines were written to stdout when the code was run.

If we run the tests, we it's most likely all five tests will pass, (though probably necessarily always, for reasons we'll discuss below):

```
$ python test_sh_example2_sh.py
.....
```

(continues on next page)

(continued from previous page)

```
----------------------------------------------------------------------
Ran 5 tests in 0.015s

OK
```

This time there is one more test than *Example 1*, because the script wrote a file (`FILE1`), for which there's now a reference test. Additionally, some of tests are more complex, to account for run-to-run variation and dependencies.

If you look at the tests generated, three of them should identical to the ones in *Example 1* (`test_no_exception`, `test_exit_code` and `test_stderr`). The other two are more interesting.

Exclusions from Comparisons

When generating this documentation, the test for *stdout* came out like this:

```
def test_stdout(self):
    substrings = [
        '/home/tdda/tmp',
        '11 Feb 2022 16:47:37',
        'tdda.local',
    ]
    self.assertStringCorrect(self.output,
                             os.path.join(self.refdir, 'STDOUT'),
                             ignore_substrings=substrings)
```

A list of `substrings` has been generated and these have been passed into `self.assertStringCorrect` as `ignore_substrings`. The effect of this is that any lines in the reference file (`ref/sh_example2_sh/STDOUT`) containing any of these three strings are not compared to the corresponding lines in the actual output from the command.

- The first line is considered a poor choice for comparison, not because different from run-to-run, but because the test was running in the directory `/home/tdda/tmp`. While not certain, it seems likely (to `Gentest`; or in less anthropomorphic terms, to Gentest's authors) that this line is in going to reflect the directory in which the code was run, rather than being a hard-wired output that is always `/home/tdda/tmp`. If you were using Gentest for real, and saw that, and knew that in fact this *is* a non-varying path, or that the code will only work in that particular directory, it would be sensible to remove it from list of substrings for exclusion.
- The second line, `11 Feb 2022 16:47:37` is ignored for on two grounds. First, depending on how long Gentest took to run, it may have been different for different runs among the 10. But even if it wasn't, this is a time within the window of times when the test was run. Again, Gentest "assumes" that this is a current timestamp, that will be different if the test is run at different times, and indeed, that this time will probably never occur again. Of course, this maybe wrong. It could be that the code contains that hard-wired string, perhaps marking the solomn occasion on which this every documentation was generated. And again, if you inspect the code Gentest has generated and see an excluded timestamp, or datestamp, that you think should be checked, you should of course remove it to force the check.
- Finally, `tdda.local` is excluded from checking because the machine the code was running on reports its hostname as `tdda.local`, so again, Gentest "considers it likely" that this string is host-dependent, rather than fixed.

Variable Text Comparisons

The last test is the one that might be thought to justify the claim that Gentest is *minimally artificially intelligent*:

```
def test_FILE1(self):
    patterns = [
        r'^Have a number: [0-9]{4,5}$',
    ]
    self.assertFileCorrect(os.path.join(self.cwd, 'FILE1'),
                           os.path.join(self.refdir, 'FILE1'),
                           ignore_patterns=patterns)
```

You probably noticed that the script we were testing uses the variable `$RANDOM`, which generates a pseudo-random number in the range 0 to 32,767. This generates two challenges, one of which Gentest rose to fully in this case, and the other of which it only partially matched.

The first challenge is simply that output isn't consistent. Gentest noticed that, and characterized the line using a regular expression that describes the line. Unlike `ignore_lines`, which ignores whole lines containing a given string, `ignore_patterns` is more precise: it still drops the line from comparison, but only if both the actual and the expected output lines match the regular expression.

Gentest uses Rexpy to generate the patterns for lines that vary between runs.

In this case, the regular expression generated specified that the pattern is four or 5 digits; this is pretty good, and most of the numbers generated by `$RANDOM` are 4-or 5 digits———about 97% of them in fact. But, if we run the test enough times, it will fail, when a number under 1,000 is generated. Similarly, when you run the code, you might not get `[0-9]{4,5}` as the regular expression. If you're unlucky, you might get `[0-9]{5}` (about 2.6% chance if run it 10 times, but about a 48% chance if you use the default value of 2). Conversely, if you're very lucky, you might get `[0-9]{1,5}`, in which case it should always pass.

In any case, this illustrates that Gentest's minimal artificial intelligence only goes so far, and it's a good idea to look at the generated tests, and in this case, ideally adapt the regular expression to `[0-9]{1,5}`.

If you write this to `fail.sh` (it's also in the gentest examples directory, if you run `tdda examples`):

```
#!/bin/bash
e=0
while [ $e -eq 0 ]
do
    python test_sh_example2_sh.py
    e=$?
done
```

Eventually, this will produce something like:

```
======================================================================
FAIL: test_FILE1 (__main__.TestSH_EXAMPLE2)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_sh_example2_sh.py", line 64, in test_FILE1
    ignore_patterns=patterns)
AssertionError: False is not true : 1 line is different, starting at line 2
Compare with:
    diff /Users/njr/tmp/FILE1 /Users/njr/tmp/ref/sh_example2_sh/FILE1

Note exclusions:
    ignore_patterns:
        ^Have a number: [0-9]{4,5}$

----------------------------------------------------------------------
Ran 5 tests in 0.017s
```

Notice how it's highlighting the `ignore_pattern`. And here's the diff (for the run for the documentation):

```
$ diff /Users/njr/tmp/FILE1 /Users/njr/tmp/ref/sh_example2_sh/FILE1
2c2
< Have a number: 748
---
> Have a number: 29047
```

1.5.10 R Examples for Gentest

Code and Data

These scripts and datasets are closely based on examples provided by the United States Environmental Protection Agency data at https://www.epa.gov/caddis-vol4/download-r-scripts-and-sample-data.

In order to use them, you need a functioning installation of R that can be invoked with the command `Rscript`. R also needs to have the packages `gam` and `bio.infer` installed. Those can be installed using the script `00-install-packages.R`.

The scripts have been:

- renamed with numbers to indicate the order in which to run them
- modified to run the setup scripts `0-set-variables.R`, required for them to work
- Scripts 3 and 4 have been modifed to write the plots they produce to PostScript files.

R Example 1: EPA Weighted Average Temperature Tolerances

```
tdda gentest 'Rscript 1-compute-weighted-average-tolerance-values.R' one
```

- This generates a test for running the script `1-compute-weighted-average-tolerance-values.R`, This script computes weighted average temperature tolerances for three taxas and prints them to the screen.
- The `one` is a shorter name to use for the test script. If we don't specify it, a rather long filename, based on a sanitized version of the command, will be used. (We could have specified `test_one.py` too, but `one` is enough.)
- The script should generate four tests in `test_one.py`.
- Run the test by typing:

  ```
  python test_one.py
  ```

  This assumes that the command `python` runs a suitable Python 3 with access to the TDDA library. Replace `python` with `python3` or whatever you need to run the tests under a target version on python
- If all went well, the output will be something like:

  ```
  python test_one.py
  ....
  ----------------------------------------------------------------------
  Ran 4 tests in 0.237s

  OK
  ```
- You should now have a directory `ref/one`, which will contain two files. STDOUT should contain the output that R produces to the screen, which includes a lot of startup chatter, the commands it ran, and the output.

R Example 2: A PDF Plot

The second script from the EPA generates a triptych of graphs. The code on the website displays the graphs as a pop-up, but we've modifed the code to write the graphs out as a PDF, which is rather easier to test.

To run the second example, you can either follow a similar receipe to the last, typing:

```
tdda gentest 'Rscript 2-compute-cumulative-percentiles.R' two
```

or use Genest's wizard, by just typing:

```
tdda gentest
```

- At the first prompt put in the command to run the second script:

  ```
  Enter name for test script [test_Rscript_2_compute_cumulative_percentiles_R]: two
  ```

- Accept the deafiults for everything else, just hitting the `RETURN` (enter) key until it stops

Gentest should generate `test_two.py`, and a new subdirectory `two` of the *ref* directory.

In this case, whether the tasts pass when you run them will depend on timing and luck: You may get this:

```
$ python test_two.py
.....
----------------------------------------------------------------------
Ran 5 tests in 0.258s

OK
```

or you may get something more like:

```
$ python test_two.py
..2 lines are different, starting at line 5
Compare with:
    diff /home/tdda/tmp/gentest_examples/r-examples/plots2.pdf /home/tdda/tmp/gentest_
↪examples/r-examples/ref/two/plots2.pdf

F..
======================================================================
FAIL: test_plots2_pdf (__main__.Test)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/tdda/tmp/gentest_examples/r-examples/test_two.py", line 52, in test_
↪plots2_pdf
    self.assertFileCorrect(os.path.join(self.cwd, 'plots2.pdf'),
AssertionError: False is not true : 2 lines are different, starting at line 5
Compare with:
    diff /home/tdda/tmp/gentest_examples/r-examples/plots2.pdf /home/tdda/tmp/gentest_
↪examples/r-examples/ref/two/plots2.pdf


----------------------------------------------------------------------
Ran 5 tests in 0.288s

FAILED (failures=1)
```

The reason it may pass or fail is that R's PDF writer write a timestamp into the PDF file, accurate to the second. If the timing is such that the timestamp the same, to the second, for each of the two trial runs Gentest does by default, it will

see two identical PDF files and assume they're always the same. But by the time you run the test, the time will almost certainly be later, and a slightly different PDF will be generated.

In the in which the trial PDFs were identical, the test code Gentest will generate for write the PDF file will look something like this:

```
def test_plots2_pdf(self):
    self.assertFileCorrect(os.path.join(self.cwd, 'plots2.pdf'),
                           os.path.join(self.refdir, 'plots2.pdf'))
```

If, however, the PDF generation happened at different times during Gentest's, trial runs, it will see different PDFs and generate a better test:

```
def test_plots2_pdf(self):
    patterns = [
        r'^/[A-Z][a-z]+[A-Z][a-z]{3} \(D:[0-9]{14}\)$',
    ]
    self.assertFileCorrect(os.path.join(self.cwd, 'plots2.pdf'),
                           os.path.join(self.refdir, 'plots2.pdf'),
                           ignore_patterns=patterns)
```

It can be hard to see, because PDFs are technically binary files, though the often mostly consist of text and can be treated more-or-less like text if you are careful. In fact, the sort of thing they contain is this:

```
/CreationDate (D:20220220134310)
/ModDate (D:20220220134310)
```

Fairly obviously these are date stamps: `2022-20-22T12:43:10`, written as a 14-digit string.

In the better version of the test, which Gentest generates if its trial runs produce differing outputs, it decides to ignore lines that match the regular expression shown, which both of these do.

- If you get this failure, you have a few options. First, Gentest suggests a `diff` command you can use to examine the differences. In fact, the diff command is quite just to say that the files are binary and differ. This is technically true, though some `diff` tools can be persuaded to show the differences anyway. Even if you can's see them, what you can definitely do it open the two files (the diff command includes their full paths) and look at them to see whether they look the same. Hopefully they will.
- In terms of correcting it, the simplest thing to do is to increase the number of times the test is run. Depending on how fast your machine is, the odds are that if you increase the number of runs even to 3, the first and last will probably run at different times (to the *second*), and if you increase it to 10, this will almost certainly be true.
- Alternatively, you can edit the test yourself, but this will require you to write one or more exclusion patterns, as Gentest did. They can, however, be simpler. One possibiity is to use `ignore_substrings`, which ignores lines that contain the subtrings given:

  ```
  def test_plots2_pdf(self):
      ignores = [
          '/CreationDate',
          '/ModDate'
      ]
      self.assertFileCorrect(os.path.join(self.cwd, 'plots2.pdf'),
                             os.path.join(self.refdir, 'plots2.pdf'),
                             ignore_substrings=ignores)
  ```

- In future, Gentest will probably allow you to specify a time two wait between invocations of the test command, which would be another way to fix the problem.

1.5.11 Gentest Parameters and Options

The full syntax for gentest is:

```
tdda gentest [FLAGS] 'command to test' [testfile.py [directories and files]]
```

where

- `command to test` is a runnable shell command. It is normally sensible to enclose it in single quotes, which will prevent most shell expansion (wildcard _globbing_ etc.) from happening, protect spaces etc. If you need single quotes in the comman you want to run, you either need to escape those carefully or use the wizard instead. Commands can be almost anything, including
  - plain shell command such as `cat hey`
  - compiled programs such as `someprogram param1 param2`
  - interpreted programs such as `python script param1 param2 param3`
  - shell scripts such as `sh foo.sh` or `. foo.sh`
  - pipelines such as `grep pattern file | sort | uniq`
  - `make clean all` (Plain `make` would rarely be a good choice if the `Makefile` contains dependency checking, because most of the point of `make` is run different things different times, though if you use `--no-stdout --no-stderr` (see below) to suppress checking screen output, it might be reasonable.)
- `testfile.py` is the name of the Python test script to write. If not specified, or specified as `-`, a name of the general form `test_sanitized_command.py` will be used, where santized command is derived from the command, removing problematical characters. **NOTE:** Unless you use `--no-clobber`, no checking is done: new files will simply overwrite old ones. If you use Gentest twice in same directory with commands that Gentest santizes the same way, the second will overwrite the first.
- `directories and files` is a list of directories and files to be monitored for changes while the script it run. By default, the twodirectories checked are the current directory `.` and the system temporary directory, as specified by `TMPDIR`, if set, failing which `/tmp`.

The `FLAGS` (switches, options) available are:

`-?`, `--?`, `-h`, `--help` Show Gentest's help message

`-m MAX_FILES`, `--maxfiles MAX_FILES` Maximum number of files for Gentest to track.

`-r`, `--relative-paths` Show relative paths wherever possible'

`-n ITERATIONS`, `--iterations` Number of times to run the command (default 2)

`-O`, `--no-stdout` Do not generate a test checking output to `stdout`

`-E`, `--no-stderr` Do not generate a test checking output to `stderr`

`-Z`, `--nonzeroexit` Do not require exit status to be 0

`-C`, `--no-clobber` Do not overwrite existing test script or reference directory

1.6 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.

1.6.1 Prerequisites

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

1.6.2 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):
```

(continues on next page)

(continued from previous page)

```
    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.

1.6.3 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.

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.

assertBinaryFileCorrect(*actual_path*, *ref_path*, *kind=None*)

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

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

***ref_path*:** The name of the reference binary 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.

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`

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

It also accepts the `check_data`, `check_types`, `check_order`, `check_extra_cols`, `sortby`, `condition` and `precision` optional parameters described in *assertDataFramesEqual()*.

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`

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

It also accepts the `check_data`, `check_types`, `check_order`, `check_extra_cols`, `sortby`, `condition` and `precision` optional parameters described in *assertDataFramesEqual()*.

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`

It also accepts the `check_data`, `check_types`, `check_order`, `check_extra_cols`, `sortby`, `condition` and `precision` optional parameters described in *assertDataFramesEqual()*.

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*, *remove_lines=None*, *ignore_lines=None*, *preprocess=None*, *max_permutation_cases=0*, *encoding=None*)

Check that a text 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()*.

It also accepts the `kind`, `lstrip`, `rstrip`, `ignore_substrings`, `ignore_patterns`, `remove_lines`, `preprocess` and `max_permutation_cases` optional parameters described in *assertStringCorrect()*.

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

The *ignore_lines* parameter exists for backwards compatibility as an alias for *remove_lines*.

The *assertFileCorrect()* method can be used as an alias for *assertTextFileCorrect()*, retained for backwards compatibility.

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

Check that a collection of text 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()*.

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

It also accepts the `kind`, `lstrip`, `rstrip`, `ignore_substrings`, `ignore_patterns`, `remove_lines`, `preprocess` and `max_permutation_cases` optional parameters described in *assertStringCorrect()*.

The *assertFilesCorrect()* metohd can be used as an alias for *assertTextFilesCorrect()*, retained for backwards compatibility.

assertStringCorrect(*string*, *ref_path*, *kind=None*, *lstrip=False*, *rstrip=False*, *ignore_substrings=None*, *ignore_patterns=None*, *remove_lines=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 should only include explicit anchors if they need to refer to the whole line. Only the matched expression within the line is ignored; any text to the left or right of the matched expression must either be **exactly** the same on both sides, or be ignorable.

remove_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.

The *ignore_lines* parameter exists for backwards compatibility as an alias for *remove_lines*.

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

Check that a text 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()*.

It also accepts the `kind`, `lstrip`, `rstrip`, `ignore_substrings`, `ignore_patterns`, `remove_lines`, `preprocess` and `max_permutation_cases` optional parameters described in *assertStringCorrect()*.

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

The *ignore_lines* parameter exists for backwards compatibility as an alias for *remove_lines*.

The *assertFileCorrect()* method can be used as an alias for *assertTextFileCorrect()*, retained for backwards compatibility.

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

Check that a collection of text 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()*.

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

It also accepts the `kind`, `lstrip`, `rstrip`, `ignore_substrings`, `ignore_patterns`, `remove_lines`, `preprocess` and `max_permutation_cases` optional parameters described in *assertStringCorrect()*.

The *assertFilesCorrect()* metohd can be used as an alias for *assertTextFilesCorrect()*, retained for backwards compatibility.

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 *assertTextFileCorrect()* (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 *ReferenceTest.set_default_data_location()* class-method.

If you haven't even defined the `None` default, and you make calls to *assertTextFileCorrect()* (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 *assertTextFileCorrect()* (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 *assertTextFileCorrect()* (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, 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.

1.6.4 `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()
```

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):
        ...

    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`).

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
```

`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(*module=None*, *argv=None*, ***kw*)

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*, *printer=None*)

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.

1.6.5 `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.

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')
```

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):
        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`).

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
```

`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.

1.6.6 Reference Test Examples

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

To copy these examples, run the command:

```
tdda examples referencetest [directory]
```

If directory is not supplied referencetest-examples will be used.

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

```
tdda examples
```

which will create a number of separate subdirectories.

1.7 TDDA's Constraints API

1.7.1 tdda.constraints

tdda.constraints.**discover_db_table**(*dbtype*, *db*, *tablename*, *inc_rex=False*, *seed=None*)
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.

Regular Expression constraints are not (currently) generated for fields in database tables.

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.**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
```

(continues on next page)

(continued from previous page)

```
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))
```

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

For detection of failures from verification of constraints, but not yet implemented for database tables.

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:

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

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

***tdda.constraints.detect_db_table()*:** For detection of failing records in a single database table, but not yet implemented for databases.

class tdda.constraints.db.constraints.**DatabaseConstraintCalculator**(*tablename*, *testing=False*)

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*, *seed=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=None*)
Determine whether the types of two values are compatible

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*, *seed=None*)
A *DatabaseConstraintDiscoverer* object is used to discover constraints on a single database table.

1.7.2 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*.

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)
```

(continues on next page)

(continued from previous page)

```
    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

    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
```

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.

1.7.3 Constraints API

TDDA constraint discovery and verification, common underlying functionality.

class tdda.constraints.baseconstraints.**BaseConstraintDiscoverer**(*inc_rex=False*, *seed=None*, ***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.

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(*raw=False*)

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*, *comment=None*)
Constraint specifying the minimum allowed value in a field.

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

class tdda.constraints.base.**SignConstraint**(*value*, *comment=None*)
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*, *comment=None*)
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*, *comment=None*)
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*, *comment=None*)
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*, *comment=None*)
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*, *comment=None*)
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*, *comment=None*)
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.

1.8 TDDA's API for Rexpy

1.8.1 tdda.rexpy

Python API

The *tdda.rexpy.rexpy* module 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*, *size=None*, *seed=None*, *dialect='portable'*, *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 of strings, a integer-valued, string-keyed dictionary or a function.

- If it's a list, each string in the list is an example string
- It it's a dictionary (or counter), each string is to be used, and the values are taken as frequencies (should be non-negative)
- If it's a function, it should be as specified below (see the definition of example_check_function)

size can be provided as:

- a Size() instance, to control various sizes within rexpy
- None (the default), in which case rexpy's defaults are used
- False or 0, which means don't use sampling

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_fragments(*vrle*, *v_id*)
Analyse the contents of each fragment in vrle across the examples it matches.

Return zip of

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

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

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

build_tree_inner(*vrles*, *fulls=None*)
Turn the VRLEs into a tree based on the different initial fragments.

check_for_failures(*rexes*, *maxExamples*)
This method is the default check_fn (See the definition of example_check_function below.)

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_bad_patterns(*freqs*)
Given freqs, a list of frequencies (for the corresponding indexed RE) identify indexes to patterns that:

- have too few strings
- cause too many patterns to be returned

NOTE: min_diff_strings_per_pattern is currently ignored.

Returns set of indices for deletion

find_frag_sep_frag_repeated(*tree*, *existing=None*, *results=None*)
This specifically looks for patterns in the tree constructed by self.build_tree of the general form

A ABA ABABA

etc., where A and B are both fragments. A common example is that A is recognizably a pattern and B is recognizably a separator. So, for example:

HM MH-RT QY-TR-BF QK-YT-IU-QP

all fit

[A-Z]{2}(-[A-Z])*

which, as fragments, would be A = (C, 2, 2) and B = ('.', 1, 1).

The tree is currently not recording or taking account of frequency in the fragments. We might want to do that. Or we might leave that as a job for whatever is going to consolidate the different branches of the tree that match the repeating patterns returned.

find_non_matches(*rexes*)
Returns all example strings that do not match any of the regular expressions in results, together with their frequencies.

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

fragment2re(*fragment*, *tagged=False*, *as_re=True*, *output=False*)
Convert fragment to RE.

If output is set, this is for final output, and should be in the specified dialect (if any).

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 in the sort order.

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_fragments(*vrle*, *v_id*)
Refine the categories for variable-run-length-encoded pattern (vrle) provided by narrowing the characters in each fragment.

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(*n*)
Sample self.all_examples for potentially faster induction.

sample_examples(*examples*, *n*)
Sample examples provided for potentially faster induction.

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*, *output=False*)
Convert variable run-length-encoded code string to regular expression

If output is set, this is for final output, and should be in the specified dialect (if any).

vrle2refrags(*vrles*, *output=False*)
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)

class tdda.rexpy.rexpy.**IDCounter**
Rather Like a counter, but also assigns a numeric ID (from 1) to each key and actually builds the counter on that.

Use .add to increment an existing key's count, or to initialize it to (by default) 1.

Get the key's ID with .ids[key] or .keys.get(key).

add(*key*, *freq=1*)
Adds the given key, counting it and ensuring it has an id.

Returns the id.

getitem(*key*)
Gets the count for the key

class tdda.rexpy.rexpy.**PRNGState**(*n*)
Seeds the Python PRNG and after captures its state.

restore() cam be used to set them back to the captured state.

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.**escaped_bracket**(*chars*, *dialect=None*, *inner=False*)
Construct a regular expression Bracket (character class), obeying the special regex rules for escaping these:

- Characters do not, in general need to be escaped
- If there is a close bracket ("]") it mst be the first character
- If there is a hyphen ("-") it must be the last character
- If there is a carat ("^"), it must not be the first character
- If there is a backslash, it's probably best to escape it. Some implementations don't require this, but it will rarely do any harm, and most implementation understand at least some escape sequences ("w", "W", "d", "s" etc.), so escaping seems prudent.

However, javascript and ruby do not follow the unescaped "]" as the first character rule, so if either of these dialects is specified, the "]" will be escaped (but still placed in the first position.

If inner is set to True, the result is returned without brackets.

tdda.rexpy.rexpy.**example_check_function**(*rexes*, *maxN=None*)
CHECK FUNCTIONS This is an example check function

A check function takes a list of regular expressions (as strings) and optionally, a maximum number of (different) strings to return.

It should return two things:

- An Examples object (importing that class from rexpy.py) containing strings that don't match any of the regular expressions in the list provided. (If the list is empty, all strings are candidates to be returned.)
- a list of how many strings matched each regular expression provided (in the same order).

If maxN is None, it should return all strings that fail to match; if it is a number, that is the maximum number of (distinct) failures to return. The function *is* expected to return all failures, however, if there are fewer than maxN failures (i.e., it's not OK if maxN is 20 to return just 1 failiing string if actually 5 different strings fail.)

Examples: The examples object is initialized with a list of (distinct) failing strings, and optionally a corresponding list of their frequencies. If no frequencies are provided, all frequencies will be set to 1 when the Examples object is initialized.

The regular expression match frequencies are used to eliminate low-frequency or low-ranked regular expressions. It is not essential that the values cover all candidate strings; it is enough to give frequencies for those strings tested before maxN failures are generated.

(Normally, the regular expressions provided will be exclusive, i.e. at most one will match, so it's also fine only to test a string against regular expressions until a match is found. . . you don't need to test against other patterns in case the string also matches more than one.)

`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*, *size=None*, *seed=None*, *dialect='portable'*, *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},
```

(continues on next page)

(continued from previous page)

```
    ('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)
    (('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*, *examples*, *sort_on_deduped=False*)
Find patterns, in (descending) 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.**nvl**(*v*, *w*)
This function is used as syntactic sugar for replacing null values.

tdda.rexpy.rexpy.**pdextract**(*cols*, *seed=None*)
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 numpy as np
import pandas as pd
from tdda.rexpy import pdextract

df = pd.DataFrame({'a3': ["one", "two", 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*, *examples*, *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*, *examples*, *sort_on_deduped=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*, *examples*, *sort_on_deduped=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.**rexpy_streams**(*in_path=None*, *out_path=None*, *skip_header=False*, *quote=False*, ***kwargs*)

in_path is None: to read inputs from stdin path to file: to read inputs from file at in_path list of strings: to use those strings as the inputs

out_path is: None: to write outputs to stdout path to file: to write outputs from file at out_path False: to return the strings as a list

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))
```

1.9 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.

1.10 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`.

1.11 Examples

The TDDA package includes embedded examples of code and data. To copy these examples, run:

```
tdda examples
```

which will create a number of subdirectories of the current directory, currently:

- `constraints_examples`
- `rexpy_examples`
- `referencetest_examples`
- `gentest_examples`

1.12 Recent Changes

1.12.1 This Version

- **2.0** Addition of Gentest—functionality for automatically generating Python test code for any command-line program

- **2.0** Major overhaul of documentation.
  - More descriptive documentation
  - Better (though incomplete) separation between user code (particularly the command-line utilities `tdda gentest`, `tdda discover`, `tdda verify`, `tdda detect` and `rexpy`).
  - Add more external links to resources and fix those that had rusted
  - Improve the CSS to make the documentation render better on tdda.readthedocs.io
  - Adopt a customized version of the readthedocs theme for the documentation everywhere, so that what you see if you build the documentation locally should be more similar to what you see at tdda.readthedocs.io
- **2.0** Significant changes to the algorithm used by Rexpy. Should now be faster, but potentially more stochastic.
- **2.0** Rexpy can now generate many different flavours of regular expressions.
- **2.0. Planned Deprecation** We plan to move from using `.feather` files to `.parquet` files in the 2.1 release, ad which point `.feather` files will immediately be deprecated.

1.12.2 Older Versions

- Reference test exercises added.
- Escaping of special characters for regular expressions is now done in a way that is uniform across Python2, Python pre-3.7, and Python 3.7+.
- JSON is now generated the same for Python2 and PYthon3 (no blank lines at the end of lines, and UTF8-encoded).
- Fixed issue with `tdda test` command not working properly in the previous version, to self-test an installation.
- Added new option flag `--interleave` for *tdda detect*. This causes the `_ok` detection fields to be interleaved with the original fields that they refer to in the resulting detection dataset, rather than all appearing together at the far right hand side. This option was actually present in the previous release, but not sufficiently documented.
- Fix for the `--write-all` parameter for *`tdda.referencetest`* result regeneration, which had regressed slightly in the previous version.
- Improved reporting of differences for text files in *`tdda.referencetest`* when the *actual* results do not match the *expected* file contents. Now fully takes account of the `ignore` and `remove` parameters.
- The `ignore_patterns` parameter in *`assertTextFileCorrect()`* (and others) in *`tdda.referencetest`* now causes only the portion of a line that matches the regular expressions to be ignored; anything else on the line (before or after the part that matches a regular expression) must be **identical** in the *actual* and *expected* results. This means that you are specifying the part of the line that is allowed to differ, rather than marking an entire line to be ignored. This is a change in functionality, but is what had always been intended. For fuller control (and to get the previous behaviour), you can anchor the expressions with `^.*(...).*$`, and then they will apply to the entire line.
- The `ignore_patterns` parameter in *`tdda.referencetest`* can now accept grouped subexpressions in regular expressions. This allows use of alternations, which were previously not supported.
- The `ignore_substrings` parameter in *`assertTextFileCorrect()`* (and others) *`tdda.referencetest`* now only matches lines in the *expected* file (where you have full control over what will appear there), not in the *actual* file. This fixes a problem with differences being masked (and not reported as problems) if the *actual* happened to include unexpected matching content on lines other than where intended.
- The *`tdda.constraints`* package is now more resilient against unexpected type mismatches. Previously, if the type didn't match, then in some circumstances exceptions would be (incorrectly) raised for other constraints, rather than failures.

- The *tdda.constraints* package now supports Python `datetime.date` fields in Pandas DataFrames, in addition to the existing support of `datetime.datetime`.
- The *tdda.constraints* Python API now provides support for in-memory constraints, by allowing Python dictionaries to be passed in to `verify_df()` and `detect_df()`, as an alternative to passing in a `.tdda` filename. This allows an application using the library to store its constraints however it wants to, rather than having to use the filesystem (e.g. storing it online and fetching with an HTTP `GET`).
- The *tdda.constraints* package can now access MySQL databases using the mysql.connector driver, in addition to the MySQLdb and mysqlclient drivers.
- The `tdda.rexpy` tool can now *quote* the regular expressions it produces, with the new `--quote` option flag. This makes it easier to copy the expressions to use them on the command line, or embed them in strings in many programming languages.
- The Python API now allows you to `import tdda` and then refer to its subpackages via `tdda.referencetest`, `tdda.constraints` or `tdda.rexpy`. Previously you had to explicitly import each submodule separately.

CHAPTER 2

Resources

- Talks & Filmed Tutorials about TDDA etc (Nick Radcliffe)
- TDDA Library (PyCon DE, Eberhard Hansis, 2019)
- Tutorial Video Screencasts on Exercises
- Tutorials YouTube Channel
- Paper: Automatic Constraint Generation and Verification
- 1-page summary of ideas
- Quick-reference Guide / Cheat Sheet
- TDDA Blog
- Twitter tdda0
- Slack (mail/DM on twitter for invitation)
- Source Repository (Github)

CHAPTER 3

Indexes and Search

- genindex
- modindex
- search

Python Module Index

t

Index

D

E

F

G

H

I

L

M

N

P

R

S

T

V

W