tappy Documentation

Release 2.1

Matt Layman

September 23, 2016

Contents

tappy provides tools for working with the Test Anything Protocol (TAP) in Python.

tappy generates TAP output from your `unittest` test cases. You can use the TAP output files with a tool like the Jenkins TAP plugin or any other TAP consumer.

tappy also provides a `tappy` command line tool as a TAP consumer. This tool can read TAP files and display the results like a normal Python test runner. tappy provides other TAP consumers via Python APIs for programmatic access to TAP files.

For the curious: tappy sounds like "happy."

CHAPTER 1

Installation

tappy is available for download from PyPI. tappy is currently supported on Python 2.7, 3.3, 3.4, 3.5, and PyPy. It is continuously tested on Linux, OS X, and Windows.

```
$ pip install tap.py
```

CHAPTER 2

Documentation

2.1 TAP Producers

tappy integrates with `unittest` based test cases to produce TAP output. The producers come in three varieties: support with only the standard library, support for nose, and support for pytest.

- `TAPTestRunner` - This subclass of `unittest.TextTestRunner` provides all the functionality of `TextTestRunner` and generates TAP files or streams.
- tappy for **nose** - tappy provides a plugin (simply called `TAP`) for the **nose** testing tool.
- tappy for **pytest** - tappy provides a plugin called `tap` for the **pytest** testing tool.

By default, the producers will create one TAP file for each `TestCase` executed by the test suite. The files will use the name of the test case class with a `.tap` extension. For example:

```
class TestFoo(unittest.TestCase):

    def test_identity(self):
        """Test numeric equality as an example."""
        self.assertTrue(1 == 1)
```

The class will create a file named `TestFoo.tap` containing the following.

```
# TAP results for TestFoo
ok 1 - Test numeric equality as an example.
1..1
```

The producers also have streaming modes which bypass the default runner output and write TAP to the output stream instead of files. This is useful for piping output directly to tools that read TAP natively.

```
$ nosetests --with-tap --tap-stream tap.tests.test_parser
# TAP results for TestParser
ok 1 - test_after_hash_is_not_description (tap.tests.test_parser.TestParser)
ok 2 - The parser extracts a bail out line.
ok 3 - The parser extracts a diagnostic line.
ok 4 - The TAP spec dictates that anything less than 13 is an error.
ok 5 - test_finds_description (tap.tests.test_parser.TestParser)
ok 6 - The parser extracts a not ok line.
ok 7 - The parser extracts a test number.
ok 8 - The parser extracts an ok line.
ok 9 - The parser extracts a plan line.
ok 10 - The parser extracts a plan line containing a SKIP.
1..10
```

2.1.1 Examples

The `TAPTestRunner` works like the `TextTestRunner`. To use the runner, load test cases using the `TestLoader` and pass the tests to the run method. The sample below is the test runner used with tappy's own tests.

```
import os
import unittest

from tap import TAPTestRunner

if __name__ == '__main__':
    tests_dir = os.path.dirname(os.path.abspath(__file__))
    loader = unittest.TestLoader()
    tests = loader.discover(tests_dir)
    runner = TAPTestRunner()
    runner.set_outdir('testout')
    runner.set_format('Hi: {method_name} - {short_description}')
    runner.run(tests)
```

Running tappy with **nose** is as straightforward as enabling the plugin when calling `nosetests`.

```
$ nosetests --with-tap
...............
----------------------------------------------------------------------
Ran 15 tests in 0.020s

OK
```

The **pytest** plugin is automatically activated for **pytest** when tappy is installed. Because it is automatically activated, **pytest** users should specify an output style.

```
$ py.test --tap-files
============================ test session starts ============================
platform linux2 -- Python 2.7.6 -- py-1.4.30 -- pytest-2.7.2
rootdir: /home/matt/tappy, inifile:
plugins: tap.py
collected 94 items

tests/test_adapter.py .....
tests/test_directive.py ......
tests/test_line.py ......
tests/test_loader.py ......
tests/test_main.py .
tests/test_nose_plugin.py ......
tests/test_parser.py ..............
tests/test_pytest_plugin.py .........
tests/test_result.py .......
tests/test_rules.py ........
tests/test_runner.py .......
tests/test_tracker.py ..............

======================== 94 passed in 0.24 seconds =========================
```

The configuration options for each TAP tool are listed in the following sections.

2.1.2 TAPTestRunner

You can configure the `TAPTestRunner` from a set of class or instance methods.

- `set_stream` - Enable streaming mode to send TAP output directly to the output stream. Use the `set_stream` instance method.

```
runner = TAPTestRunner()
runner.set_stream(True)
```

- `set_outdir` - The `TAPTestRunner` gives the user the ability to set the output directory. Use the `set_outdir` class method.

```
TAPTestRunner.set_outdir('/my/output/directory')
```

- `set_combined` - TAP results can be directed into a single output file. Use the `set_combined` class method to store the results in `testresults.tap`.

```
TAPTestRunner.set_combined(True)
```

- `set_format` - Use the `set_format` class method to change the format of result lines. `{method_name}` and `{short_description}` are available options.

```
TAPTestRunner.set_format('{method_name}: {short_description}')
```

- `set_header` - Turn off or on the test case header output. The default is `True` (ie, the header is displayed.) Use the `set_header` instance method.

```
runner = TAPTestRunner()
runner.set_header(False)
```

2.1.3 nose TAP Plugin

Note: To use this plugin, install it with `pip install nose-tap`.

The **nose** TAP plugin is configured from command line flags.

- `--with-tap` - This flag is required to enable the plugin.
- `--tap-stream` - Enable streaming mode to send TAP output directly to the output stream.
- `--tap-combined` - Store test results in a single output file in `testresults.tap`.
- `--tap-outdir` - The **nose** TAP plugin also supports an optional output directory when you don't want to store the `.tap` files wherever you executed `nosetests`.

  Use `--tap-outdir` followed by a directory path to store the files in a different place. The directory will be created if it does not exist.
- `--tap-format` - Provide a different format for the result lines. `{method_name}` and `{short_description}` are available options. For example, `'{method_name}: {short_description}'`.

2.1.4 pytest TAP Plugin

Note: To use this plugin, install it with `pip install pytest-tap`.

The **pytest** TAP plugin is configured from command line flags. Since **pytest** automatically activates the TAP plugin, the plugin does nothing by default. Users must enable a TAP output mode (via `--tap-stream|files|combined`) or the plugin will take no action.

- `--tap-stream` - Enable streaming mode to send TAP output directly to the output stream.
- `--tap-files` - Store test results in individual test files. One test file is created for each test case.
- `--tap-combined` - Store test results in a single output file in `testresults.tap`.
- `--tap-outdir` - The **pytest** TAP plugin also supports an optional output directory when you don't want to store the `.tap` files wherever you executed `py.test`.

  Use `--tap-outdir` followed by a directory path to store the files in a different place. The directory will be created if it does not exist.

2.1.5 Python and TAP

The TAP specification is open-ended on certain topics. This section clarifies how tappy interprets these topics.

The specification indicates that a test line represents a "test point" without explicitly defining "test point." tappy assumes that each test line is **per test method**. TAP producers in other languages may output test lines **per assertion**, but the unit of work in the Python ecosystem is the test method (i.e. `unittest`, nose, and pytest all report per method by default).

tappy does not permit setting the plan. Instead, the plan is a count of the number of test methods executed. Python test runners execute all test methods in a suite, regardless of any errors encountered. Thus, the test method count should be an accurate measure for the plan.

2.2 TAP Consumers

2.2.1 tappy Tool

The `tappy` command line tool is a TAP consumer. The tool accepts TAP files or directories containing TAP files and provides a standard Python `unittest` style summary report. Check out `tappy -h` for the complete list of options. You can also use the tool's shorter alias of `tap`.

```
$ tappy *.tap
.................F................................
======================================================================
FAIL: <file=TestParser.tap>
- The parser extracts a bail out line.
----------------------------------------------------------------------


----------------------------------------------------------------------
Ran 51 tests in 0.002s

FAILED (failures=1)
```

TAP Stream

`tappy` can read a TAP stream directly STDIN. This permits any TAP producer to pipe its results to `tappy` without generating intermediate output files. `tappy` will read from STDIN when no arguments are provided or when a dash character is the only argument.

Here is an example of `nosetests` piping to `tappy`:

```
$ nosetests --with-tap --tap-stream 2>&1 | tappy
.....................................................................
.........................................
---------------------------------------------------------------------
Ran 117 tests in 0.003s

OK
```

In this example, `nosetests` puts the TAP stream on STDERR so it must be redirected to STDOUT because the Unix pipe expects input on STDOUT.

`tappy` can use redirected input from a shell.

```
$ tappy < TestAdapter.tap
........
---------------------------------------------------------------------
Ran 8 tests in 0.000s

OK
```

This final example shows `tappy` consuming TAP from Perl's test tool, `prove`. The example includes the optional dash character.

```
$ prove t/array.t -v | tappy -
............
---------------------------------------------------------------------
Ran 12 tests in 0.001s

OK
```

2.2.2 API

In addition to a command line interface, tappy enables programmatic access to TAP files for users to create their own TAP consumers. This access comes in two forms:

1. A `Loader` class which provides a `load` method to load a set of TAP files into a `unittest.TestSuite`. The `Loader` can receive files or directories.

```
>>> loader = Loader()
>>> suite = loader.load(['foo.tap', 'bar.tap', 'baz.tap'])
```

2. A `Parser` class to provide a lower level interface. The `Parser` can parse a file via `parse_file` and return parsed lines that categorize the file contents.

```
>>> parser = Parser()
>>> for line in parser.parse_file('foo.tap'):
...     # Do whatever you want with the processed line.
...     pass
```

The API specifics are listed below.

class tap.loader.**Loader**
Load TAP lines into unittest-able objects.

load(*files*)
Load any files found into a suite.

Any directories are walked and their files are added as TAP files.

Returns A unittest.TestSuite instance

load_suite_from_file(*filename*)
Load a test suite with test lines from the provided TAP file.

Returns A unittest.TestSuite instance

load_suite_from_stdin()
Load a test suite with test lines from the TAP stream on STDIN.

Returns A unittest.TestSuite instance

class tap.parser.**Parser**
A parser for TAP files and lines.

parse_file(*filename*)
Parse a TAP file to an iterable of tap.line.Line objects.

This is a generator method that will yield an object for each parsed line. The file given by *filename* is assumed to exist.

parse_stdin()
Parse a TAP stream from standard input.

Note: this has the side effect of closing the standard input filehandle after parsing.

parse_text(*text*)
Parse a string containing one or more lines of TAP output.

parse(*fh*)
Generate tap.line.Line objects, given a file-like object *fh*.

fh may be any object that implements both the iterator and context management protocol (i.e. it can be used in both a "with" statement and a "for...in" statement.)

Trailing whitespace and newline characters will be automatically stripped from the input lines.

parse_line(*text*)
Parse a line into whatever TAP category it belongs.

Line Categories

The parser returns Line instances. Each line contains different properties depending on its category.

class tap.line.**Result**(*ok*, *number=None*, *description=''*, *directive=None*, *diagnostics=None*)
Information about an individual test line.

category

Returns test

ok
Get the ok status.

Return type bool

number
Get the test number.

Return type int

description
Get the description.

skip
Check if this test was skipped.

Return type bool

todo
Check if this test was a TODO.

Return type bool

class tap.line.**Plan**(*expected_tests*, *directive=None*)
A plan line to indicate how many tests to expect.

category

Returns plan

expected_tests
Get the number of expected tests.

Return type int

skip
Check if this plan should skip the file.

Return type bool

class tap.line.**Diagnostic**(*text*)
A diagnostic line (i.e. anything starting with a hash).

category

Returns diagnostic

text
Get the text.

class tap.line.**Bail**(*reason*)
A bail out line (i.e. anything starting with 'Bail out!').

category

Returns bail

reason
Get the reason.

class tap.line.**Version**(*version*)
A version line (i.e. of the form 'TAP version 13').

category

Returns version

version
Get the version number.

Return type int

class tap.line.**Unknown**

A line that represents something that is not a known TAP line.

This exists for the purpose of a Null Object pattern.

category

Returns unknown

2.3 TAP Syntax Highlighter for Pygments

Pygments contains an extension for syntax highlighting of TAP files. Any project that uses Pygments, like Sphinx, can take advantage of this feature.

This highlighter was initially implemented in tappy. Since the highlighter was merged into the upstream Pygments project, tappy is no longer a requirement to get TAP syntax highlighting.

Below is an example usage for Sphinx.

```
.. code-block:: tap

   1..2
   ok 1 - A passing test.
   not ok 2 - A failing test.
```

2.4 Contributing

tappy should be easy to contribute to. If anything is unclear about how to contribute, please submit an issue on GitHub so that we can fix it!

2.4.1 How

Fork tappy on GitHub and submit a pull request when you're ready.

2.4.2 Setup

tappy uses a standard Python toolchain. While many setups are possible, the following should get you started quickly. At minimum, you'll need virtualenv and pip to begin.

```
$ # Start from the root of a tappy clone.
$ virtualenv venv                          # Create your virtual environment.
$ source venv/bin/activate                 # Activate it.
(venv)$ pip install -r requirements-dev.txt  # Install developer tools.
(venv)$ pip install -e .                   # Install tappy in editable mode.
(venv)$ nosetests                          # Run the test suite.
```

The commands above show how to get a tappy clone configured. If you've executed those commands and the test suite passes, you should be ready to develop.

2.4.3 Guidelines

1. Code should follow PEP 8 style. Please run it through `pep8` to check.
2. Please try to conform with any conventions seen in the code for consistency.
3. Make sure your change works against master! (Bonus points for unit tests.)

2.5 Alternatives

tappy is not the only project that can produce TAP output for Python. While tappy is a capable TAP producer and consumer, other projects might be a better fit for you. The following comparison lists some other Python TAP tools and lists some of the biggest differences compared to tappy.

2.5.1 pycotap

pycotap is a good tool for when you want TAP output, but you don't want extra dependencies. pycotap is a zero dependency TAP producer. It is so small that you could even embed it into your project. Check out the project homepage.

2.5.2 catapult

catapult is a TAP producer. catapult is also capable of producing TAP-Y and TAP-J which are YAML and JSON test streams that are inspired by TAP. You can find the catapult source on GitHub.

2.5.3 pytap13

pytap13 is a TAP consumer for TAP version 13. It parses a TAP stream and produces test instances that can be inspected. pytap13's homepage is on Bitbucket.

2.5.4 bayeux

bayeux is a TAP producer that is designed to work with unittest and unittest2. bayeux is on GitLab..

2.5.5 taptaptap

taptaptap is a TAP producer with a procedural style similar to Perl. It also includes a `TapWriter` class as a TAP producer. Visit the taptaptap homepage.

2.5.6 unittest-tap-reporting

unittest-tap-reporting is another zero dependency TAP producer. Check it out on GitHub.

If there are other relevant projects, please post an issue on GitHub so this comparison page can be updated accordingly.

2.6 Releases

2.6.1 Version 2.1, Released September 23, 2016

- Add `Parser.parse_text` to parse TAP provided as a string.

2.6.2 Version 2.0, Released July 31, 2016

- Remove nose plugin. The plugin moved to the `nose-tap` distribution.
- Remove pytest plugin. The plugin moved to the `pytest-tap` distribution.
- Remove Pygments syntax highlighting plugin. The plugin was merged upstream directly into the Pygments project and is available without tappy.
- Drop support for Python 2.6.

2.6.3 Version 1.9, Released March 28, 2016

- `TAPTestRunner` has a `set_header` method to enable or disable test case header ouput in the TAP stream.
- Add support for Python 3.5.
- Perform continuous integration testing on OS X.
- Drop support for Python 3.2.

2.6.4 Version 1.8, Released November 30, 2015

- The `tappy` TAP consumer can read a TAP stream directly from STDIN.
- Tracebacks are included as diagnostic output for failures and errors.
- The `tappy` TAP consumer has an alternative, shorter name of `tap`.
- The pytest plugin now defaults to no output unless provided a flag. Users dependent on the old default behavior can use `--tap-files` to achieve the same results.
- Translated into Arabic.
- Translated into Chinese.
- Translated into Japanese.
- Translated into Russian.
- Perform continuous integration testing on Windows with AppVeyor.
- Improve unit test coverage to 100%.

2.6.5 Version 1.7, Released August 19, 2015

- Provide a plugin to integrate with pytest.
- Document some viable alternatives to tappy.
- Translated into German.
- Translated into Portuguese.

2.6.6 Version 1.6, Released June 18, 2015

- `TAPTestRunner` has a `set_stream` method to stream all TAP output directly to an output stream instead of a file. results in a single output file.
- The `nosetests` plugin has an optional `--tap-stream` flag to stream all TAP output directly to an output stream instead of a file.
- tappy is now internationalized. It is translated into Dutch, French, Italian, and Spanish.
- tappy is available as a Python wheel package, the new Python packaging standard.

2.6.7 Version 1.5, Released May 18, 2015

- `TAPTestRunner` has a `set_combined` method to collect all results in a single output file.
- The `nosetests` plugin has an optional `--tap-combined` flag to collect all results in a single output file.
- `TAPTestRunner` has a `set_format` method to specify line format.
- The `nosetests` plugin has an optional `--tap-format` flag to specify line format.

2.6.8 Version 1.4, Released April 4, 2015

- Update `setup.py` to support Debian packaging. Include man page.

2.6.9 Version 1.3, Released January 9, 2015

- The `tappy` command line tool is available as a TAP consumer.
- The `Parser` and `Loader` are available as APIs for programmatic handling of TAP files and data.

2.6.10 Version 1.2, Released December 21, 2014

- Provide a syntax highlighter for Pygments so any project using Pygments (e.g., Sphinx) can highlight TAP output.

2.6.11 Version 1.1, Released October 23, 2014

- `TAPTestRunner` has a `set_outdir` method to specify where to store `.tap` files.
- The `nosetests` plugin has an optional `--tap-outdir` flag to specify where to store `.tap` files.
- tappy has backported support for Python 2.6.
- tappy has support for Python 3.2, 3.3, and 3.4.
- tappy has support for PyPy.

2.6.12 Version 1.0, Released March 16, 2014

- Initial release of tappy
- `TAPTestRunner` - A test runner for `unittest` modules that generates TAP files.
- Provides a plugin for integrating with **nose**.