Sample Tester Documentation

Release 0.11.1

Victor Chudnovsky

May 24, 2019

Contents:

The sample tester allows defining tests once and applying them to semantically identical executables (typically runnable code samples) instantiated in multiple languages and environments.

CHAPTER 1

Installation

(optional) Activate your preferred virtual environment:

```
. PATH/TO/YOUR/VENV/bin/activate
```

Install the necessary packages:

```
pip install pyyaml  # needs to be installed before sample-tester
pip install sample-tester
```

This will put the command `sample-tester` in your path.

CHAPTER 2

Defining tests

To execute a test, you will need:

1. A "test plan", defined via one or more `*.yaml` files. Here's an example:

Listing 1: language.test.yaml

```
test:
  suites:
  - name:  "Language samples test"
    setup:     # can have yaml and/or code, just as in the cases below
      - code:
          log('In setup "hi"')
    teardown:  # can have yaml and/or code, just as in the cases below
      - code:
          log('In teardown bye')
    cases:

    - name: "A test defined via yaml directives"
      spec:
      - call:
          sample: "language_analyze_sentiment_text"
          params:
            content:
              literal: "happy happy smile hope"
      - assert_success: [] # try assert_failure to see how failure looks
      - assert_contains:
          - message: "Score is very positive"
          - literal: "score: 0.8"
      - assert_contains:
          - message: "Magnitude is very positive"
          - literal: "magnitude: 0.8"
      - assert_not_contains:
          - message: "Random message"
          - literal: "The rain in Spain falls mainly in the plain"
```

See the *Testplan* page for information on the yaml directives available in the testplan, and how to use them directly via embedded Python code.

2. A "manifest", defined via one or more `*.manifest.yaml` files. Here's an example:

Listing 2: language.manifest.yaml

```
version: 2
sets:
- environment: java
  invocation: "{jar} -D{class} {path} @args"
  path: "examples/mock-samples/java/"
  __items__:
  - class: AnalyzeSentiment
    jar: "./do_java"
    chdir: "examples/mock-samples/java/"
    path: "language-v1/AnalyzeSentiment"
    sample: "language_analyze_sentiment_text"
- environment: python
  bin: "python3"
  path: "examples/mock-samples/python/"
  __items__:
  - path: "language-v1/analyze_sentiment_request_language_sentiment_text.py"
    sample: "language_analyze_sentiment_text"
- environment: bash
  # notice: no "bin:" because artifacts are already executable
  path: "examples/mock-samples/sh/"
  __items__:
  - path: "language-v1/analyze_sentiment.sh"
    sample: "language_analyze_sentiment_text"
```

See the *Manifest file format* page for an explanation of the manifest.

2.1 Testplan

One of the inputs to sample-tester is the "testplan", which outlines how to run the samples and what checks to perform.

1. The testplan can be spread over any number of `TESTPLAN.yaml` files.
2. You can have any number of test suites.
3. Each test suite can have `setup`, `teardown`, and `cases` sections.
4. The `cases` section is a list of test cases. For _each_ test case, `setup` is executed before running the test case and `teardown` is executed after.
5. `setup`, `teardown` and each `cases[...].spec` is a list of directives and arguments. The directives can be any of the following YAML directives:
   - `log`: print the arguments, printf style
   - `uuid`: return a uuid (if called from yaml, assign it to the variable names as an argument)
   - `shell`: run in the shell the command specified in the argument
   - `call`: call the artifact named in the argument; error if the call fails
   - `call_may_fail`: call the artifact named in the argument; do not error even if the call fails
   - `assert_contains`: require the given variable to contain a string; abort the test case otherwise

- `assert_not_contains`: require the given variable to not contain a string; abort the test case otherwise
- `assert_success`: require that the exit code of the last `call_may_fail` was 0; abort the test case otherwise. If the preceding call was a just a `call`, it would have already failed on a non-zero exit code.
- `assert_failure`: require that the exit code of the last `call_may_fail` or `call` was NOT 0; abort the test case otherwise. Note, though, that if we're executing this after just a `call`, it must have succeeded so this assertion will fail.
- `env`: assign the value of an environment variable to a test case variable
- `extract_match`: extrack regex matches into local variables
- `code`: execute the argument as a chunk of Python code. The other directives above are available as Python calls with the names above. In addition, the following functions are available inside Python `code` only:
  - `fail`: mark the test as having failed, but continue executing
  - `abort`: mark the test as having failed and stop executing
  - `assert_that`: if the condition in the first argument is false, abort the test case

Here is an informative instance of a sample testfile:

```
test:
  suites:
  - name:  "Language samples test"
    setup:     # can have yaml and/or code, just as in the cases below
      - code:
          log('In setup "hi"')
    teardown:  # can have yaml and/or code, just as in the cases below
      - code:
          log('In teardown bye')
    cases:

    - name: "A test defined via yaml directives"
      spec:
      - call:
          sample: "language_analyze_sentiment_text"
          params:
            content:
              literal: "happy happy smile hope"
      - assert_success: [] # try assert_failure to see how failure looks
      - assert_contains:
          - message: "Score is very positive"
          - literal: "score: 0.8"
      - assert_contains:
          - message: "Magnitude is very positive"
          - literal: "magnitude: 0.8"
      - assert_not_contains:
          - message: "Random message"
          - literal: "The rain in Spain falls mainly in the plain"
# Above is the typical usage

    - name: "A test defined via 'code'"
      spec:
      - code: |
          out = call("language_analyze_sentiment_text", content="happy happy smile␣
↪hope")
          assert_success("that should have worked", "well")
          import re
```

(continues on next page)

(continued from previous page)

```
        score_found = re.search('score: ([0123456789.]+)', out)
        assert_that(score_found is not None, 'score matches regexp')
        score = float(score_found.group(1))
        assert_that(score > 0.7, 'score is high')

        magnitude_found = re.search('magnitude: ([0123456789.]+)', out)
        assert_that(magnitude_found is not None, 'magnitude matches regexp')
        magnitude = float(magnitude_found.group(1))
        assert_that(magnitude > 0.7, 'magnitude is high')

        assert_not_contains("random message", "the rain in Spain")

  - name: "A test defined via 'code', with explicit calls to specific samples"
    spec:
    - code: |
        _, out = shell("python3 examples/mock-samples/python/language-v1/analyze_
↪sentiment_request_language_sentiment_text.py -content='happy happy smile hope'")

    # You can interleave yaml and code!
    - assert_success:
      - "that should have worked {}"
      - well

    - code: |
        import re

        score_found = re.search('score: ([0123456789.]+)', out)  # TODO: Can this␣
↪be negative?
        assert_that(score_found is not None, 'score matches regexp')
        score = float(score_found.group(1))
        assert_that(score > 0.7, 'score is high')
        home = env('HOME')
        log('home directory: {}'.format(home))

        magnitude_found = re.search('magnitude: ([0123456789.]+)', out)
        assert_that(magnitude_found is not None, 'magnitude matches regexp')
        magnitude = float(magnitude_found.group(1))
        assert_that(magnitude > 0.7, 'magnitude is high')
```

This test plan has three equivalent representations of the same test, one with canonical artifact paths in the declarative style (using YAML directives), the second with canonical artifact paths in the imperative style (using a `code` block), and the third using absolute artifact paths in the imperative style (which you would rarely use, since th point of this tool is to not have to hardcode different paths to semantically identical samples).

Unless you specify explicit paths to each sample (which means your test plan cannot run for different languages/environments simultaneously), you will need one or more manifest files (`*.manifest.yaml`) listing the path and identifiers for each sample in each language/environment. . Refer to the *Manifest file format* page for an explanation of the structure of the `*.manifest.yaml` files.

2.2 Manifest file format

A manifest file is a YAML file that associates each artifact (sample) of interest on disk with a series of tags that can be used to uniquely identify that artifact. Right now both versions "1" and "2" of the manifest file format are supported; version "2" is a superset of version "1".

The fundamental unit in a manifest is the “item”, which is a collection of tag name/value pairs; each unit should correspond to exactly one artifact on disk.

Since a lot of the artifacts will share part or all of some tags (for example, the initial directory components, or the binary used to execute them), “items” are grouped into “sets”. Each set may define its own tag name/value pairs. These pairs are applied to each of the items inside the set as follows:

1. If the item does not define a given tag name, then the tag name/value pair in its set is applied to the item.
2. If the item does define a given tag name, then the corresponding tag value specified in the set is prepended to the corresponding value specified in the item. This is particularly useful in the case of paths: the set may define the common path for all of its items, and each item specifies its unique trailing directories and filename.

In manifest version “2”, tag values can include references to other tags: the value of tag “A” can reference the value of tag “B” by enclosing the name of tag “B” in curly brackets: `{TAG_B_NAME}`. For example:

```
name: Zoe
greeting: "Hello, {name}!
```

will define the same sets of tags as

```
name: Zoe
greeting: "Hello, Zoe!"
```

While tags can be referenced arbitrarily deep, no reference can form a loop (ie a tag directly or indirectly including itself).

2.3 Tags for sample-tester

Some manifest tags are of special interest to the sample test runner:

- `sample`: The unique ID for the sample.
- `path`: The path to the sample source code on disk.
- `environment`: A label used to group samples that share the same programming language or execution environment. In particular, artifacts with the same `sample` but different `environments` are taken to represent the same conceptual sample, but implemented in the different languages/environments; this allows a test specification to refer to the `samples` only and sample-tester will then run that test for each of the `environments` available.
- `invocation`: The command line to use to run the sample. The invocation typically makes use of two features for flexibility:
  - manifest tag inclusion: By including a `{TAG_NAME}`, `invocation` (just like any tag) can include the value of another tag.
  - tester argument substitution: By including a `@args` literal, the `invocation` tag can specify where to insert the sample parameters as determined by the sample-tester from the test plan file.

  Thus, the following would be the typical usage for Java, where each sample item in the manifest includes a `class_name` tag and a `jar` tag:

  ```
  invocation: "java {jar} -D{class_name} -Dexec.arguments='@args'"
  ```

- (deprecated) `bin`: The executable used to run the sample. The sample `path` and arguments are appended to the value of this tag to form the command line that the tester runs.

Advanced usage: you can tell sample-tester to use different key names than the ones above. For example, to use keys `some_name`, `how_to_call`, and `switch_path` instead of `sample`, `invocation`, and `chdir`, respectively, you would simply specify this flag when calling sample-tester:

```
-c tag:some_name:how_to_call,switch_path
```

Here's a typical manifest file:

```
version: 2
sets:
- environment: java
  invocation: "{jar} -D{class} {path} @args"
  path: "examples/mock-samples/java/"
  __items__:
  - class: AnalyzeSentiment
    jar: "./do_java"
    chdir: "examples/mock-samples/java/"
    path: "language-v1/AnalyzeSentiment"
    sample: "language_analyze_sentiment_text"
- environment: python
  bin: "python3"
  path: "examples/mock-samples/python/"
  __items__:
  - path: "language-v1/analyze_sentiment_request_language_sentiment_text.py"
    sample: "language_analyze_sentiment_text"
- environment: bash
  # notice: no "bin:" because artifacts are already executable
  path: "examples/mock-samples/sh/"
  __items__:
  - path: "language-v1/analyze_sentiment.sh"
    sample: "language_analyze_sentiment_text"
```

CHAPTER 3

Running tests

To run the tests you have *defined*, do the following:

1. Prepare your environment. For example, to run tests against Google APIs, ensure you have credentials set up:

```
export GOOGLE_APPLICATION_CREDENTIALS=/path/to/your/creds.json
```

2. Run the tester, specifying your manifest (any number of `*.manifest.yaml` files) and test plan (any number of other `*.yaml` files):

```
sample-tester examples/convention-tag/language.test.yaml examples/convention-tag/
↪language.manifest.yaml
```

3.1 Command-line flags

3.1.1 Basic usage

```
sampletester TEST.yaml [TEST.yaml ...] [MANIFEST.manifest.yaml ...]
             [--envs=REGEX] [--suites=REGEX] [--cases=REGEX]
             [--fail-fast]
```

where:

- there can be any number of `TEST.yaml` *testplan* files
- there can be any number of `MANIFEST.manifest.yaml` *manifest* files
- `--envs`, `--suites`, and `--cases` are Python-style regular expressions (beware shell-escapes!) to select which environments, suites, and cases to run, based on their names. All the environemnts, suites, or cases will be selected to run by default if the corresponding flag is not set. Note that if an environment is not selected, its suites are not selected regardless of `--suites`; if a suite is not selected, its testcases are not selected regardless of `--cases`.

- `--fail-fast` makes execution stop as soon as a failing test case is encountered, without executing any remaining test cases.

Controlling the output

In all cases, `sample-tester` exits with a non-zero code if there were any errors in the flags, test config, or test execution.

In addition, by default `sampletester` prints the status of test cases to stdout. This output is controlled by the following flags:

- `--verbosity` (`-v`): controls how much output to show for passing tests. The default is a "summary" view, but "quiet" (no output) and "detailed" (full case output) options are available.
- `--suppress_failures` (`-f`): Overrides the default behavior of showing output for failing test cases, regardless of the `--verbosity` setting
- `--xunit=FILE` outputs a test summary in xUnit format to `FILE` (use `-` for stdout).

3.1.2 Advanced usage

The tester uses a "convention" to match sample names in the testplan to actual, specific files on disk for given languages and environments. Each convention may choose to take some set-up arguments. You can specify an alternate convention and/or convention arguments via the flag `--convention=CONVENTION:ARG,ARGS`. The default convention is `tag:sample`, which uses the `sample` key in the manifest files. To use, say, the `target` key in the manifest, simply pass `--convention=tag:target`.

If you want to define an additional convention, refer to the documention in the repo on how to do so. If you do have such an additional convention defined, you may use the `--convention` flag to select it and give it any desired arguments, as above.