scriptharness Documentation

Release 0.1.0a0

Aki Sasaki

August 18, 2015

Contents

Scriptharness is a framework for writing scripts. There are three core principles: full logging, flexible configuration, and modular actions. The goal of *full logging* is to be able to debug problems purely through the log. The goal of *flexible configuration* is to make each script useful in a variety of contexts and environments. The goals of *modular actions* are a) faster development feedback loops and b) different workflows for different usage requirements.

CHAPTER 1

Full logging

Many scripts log. However, logging can happen sporadically, and it's generally acceptable to run a number of actions silently (e.g., `os.chdir()` will happily change directories with no indication in the log). In *full logging*, the goal is to be able to debug bustage purely through the log.

At the outset, the user can add a generic logging wrapper to any method with minimal fuss. As scriptharness matures, there will be more customized wrappers to use as drop-in replacements for previously-non-logging methods.

CHAPTER 2

Flexible configuration

Many scripts use some sort of configuration, whether hardcoded, in a file, or through the command line. A family of scripts written by the same author(s) may have similar configuration options and patterns, but often times they vary wildly from script to script.

By offering a standard way of accepting configuration options, and then exporting that config to a file for later debugging or replication, scriptharness makes things a bit neater and cleaner and more familiar between scripts.

By either disallowing runtime configuration changes, or by explicitly logging them, scriptharness removes some of the guesswork when debugging bustage.

CHAPTER 3

Modular actions

Scriptharness actions allow for:

- faster development feedback loops. No need to rerun the entirety of a long-running script when trying to debug a single action inside that script.
- different workflows for different usage requirements, such as running standalone versus running in cloud infrastructure

This is in the same spirit of other frameworks that allow for discrete targets, tasks, or actions: make, maven, ansible, and many more.

CHAPTER 4

Running unit tests

4.1 Linux and OS X

```
# By default, this will look for python 2.7 + 3.{3,4,5}.
# You can run |tox -e ENV| to run a specific env, e.g. |tox -e py27|
pip install tox
tox
# alternately, ./run_tests.sh
```

4.2 Windows

```
# By default, this will look for python 2.7 + 3.4
# You can run |tox -c tox_win.ini -e ENV| to run a specific env, e.g. |tox -c tox_win.ini -e py27|
pip install tox
tox -c win.ini
```

4.2.1 Quickstart

Here's an example script, quickstart.py.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# This file is formatted slightly differently for readability in ReadTheDocs.
"""python-scriptharness quickstart example.

This file can be found in the examples/ directory of the source at
https://github.com/scriptharness/python-scriptharness
"""
from __future__ import absolute_import, division, print_function, \
                       unicode_literals
import scriptharness
import scriptharness.commands

"""First, define functions for all actions.  Each action MUST have a function
defined.  The function should be named the same as the action.  (If the
action has a `-` in it, replace it with an `_`; e.g. an action named
`upload-to-s3` would call the `upload_to_s3()` function.  Each action function
will take a single argument, `context`.
```

```
Each action function should be idempotent, and able to run standalone.
In this example, `package` may require that the steps in `build` ran at
some point before `package` is run, but we can't assume that happened in
the same script run.  It could have happened yesterday, or three weeks ago,
and `package` should still be able to run.  If you need to save state
between actions, consider saving state to disk.
"""
def clobber(context):
    """Clobber the source"""
    context.logger.info("log message from clobber")

def pull(context):
    """Pull source"""
    context.logger.info("log message from pull")

def build(context):
    """Build source"""
    context.logger.info("log message from build")
    if context.config.get("new_argument"):
        context.logger.info("new_argument is set to %s",
                            context_config['new_argument'])

def package(context):
    """Package source"""
    context.logger.info("log message from package")
    scriptharness.commands.run(
        ['python', '-c',
         "from __future__ import print_function; print('hello world!')"]
    )

def upload(context):
    """Upload packages"""
    context.logger.info("log message from upload")

def notify(context):
    """Notify watchers"""
    context.logger.info("log message from notify")


if __name__ == '__main__':
    """Enable logging to screen + artifacts/log.txt.  Not required, but
    without it the script will run silently.
    """
    scriptharness.prepare_simple_logging("artifacts/log.txt")

    """Define actions.  All six actions are available to run, but if the
    script is run without any action commandline options, only the
    enabled actions will run.

    If default_actions is specified, it MUST be a subset of all_actions
    (the first list), and any actions in default_actions will be enabled
    by default (the others will be disabled).  If default_actions isn't
    specified, all the actions are enabled.

    Each action MUST have a function defined (see above).
    """
    actions = scriptharness.get_actions_from_list(
        ["clobber", "pull", "build", "package", "upload", "notify"],
```

```
        default_actions=["pull", "build", "package"]
    )

    """Create a commandline argument parser, with default scriptharness
    argument options pre-populated.
    """
    template = scriptharness.get_config_template(all_actions=actions)

    """Add new commandline argument(s)
    https://docs.python.org/dev/library/argparse.html#argparse.ArgumentParser.add_argument
    """
    template.add_argument("--new-argument", action='store',
                          help="help message for --new-argument")

    """Create the Script object.  If ``get_script()`` is called a second time,
    it will return the same-named script object.  (`name` in get_script()
    defaults to "root".  We'll explore running multiple Script objects within
    the same script in the not-distant future.)

    When this Script object is created, it will parse all commandline
    arguments sent to the script.  So it doesn't matter that this script
    (quickstart.py) didn't have the --new-argument option until one line
    above; the Script object will parse it and store the new_argument
    value in its config.
    """
    script = scriptharness.get_script(actions=actions, template=template)

    """This will run the script.
    Essentially, it will go through the list of actions, and if the action
    is enabled, it will run the associated function.
    """
    script.run()
```

output

If you run this without any arguments, you might get output like this:

```
$ ./quickstart.py
00:00:00     INFO - Starting at 2015-06-21 00:00 PDT.
00:00:00     INFO - Enabled actions:
00:00:00     INFO -  pull, build, package
00:00:00     INFO - {'new_argument': None,
00:00:00     INFO -  'scriptharness_artifact_dir': '/src/python-scriptharness/docs/artifacts',
00:00:00     INFO -  'scriptharness_base_dir': '/src/python-scriptharness/docs',
00:00:00     INFO -  'scriptharness_work_dir': '/src/python-scriptharness/docs/build'}
00:00:00     INFO - Creating directory /src/python-scriptharness/docs/artifacts
00:00:00     INFO - Already exists.
00:00:00     INFO - ### Skipping action clobber
00:00:00     INFO - ### Running action pull
00:00:00     INFO - log message from pull
00:00:00     INFO - ### Action pull: finished successfully
00:00:00     INFO - ### Running action build
00:00:00     INFO - log message from build
00:00:00     INFO - ### Action build: finished successfully
00:00:00     INFO - ### Running action package
00:00:00     INFO - log message from package
00:00:00     INFO - Running command: ['python', '-c', "from __future__ import print_function; print(
```

```
00:00:00     INFO - Copy/paste: python -c "from __future__ import print_function; print('hello world
00:00:00     INFO -  hello world!
00:00:00     INFO - ### Action package: finished successfully
00:00:00     INFO - ### Skipping action upload
00:00:00     INFO - ### Skipping action notify
00:00:00     INFO - Done.
```

First, it announced it's starting the script. Next, it outputs the running config, also saving it to the file `artifacts/localconfig.json`. Then it logs each action as it runs enabled actions and skips disabled actions. Finally, it announces 'Done.'.

The same output is written to the file `artifacts/log.txt`.

--actions

You can change which actions are run via the `--actions` option:

```
$ ./quickstart.py --actions package upload notify
00:00:05     INFO - Starting at 2015-06-21 00:00 PDT.
00:00:05     INFO - Enabled actions:
00:00:05     INFO -  package, upload, notify
00:00:05     INFO - {'new_argument': None,
00:00:05     INFO -  'scriptharness_artifact_dir': '/src/python-scriptharness/docs/artifacts',
00:00:05     INFO -  'scriptharness_base_dir': '/src/python-scriptharness/docs',
00:00:05     INFO -  'scriptharness_work_dir': '/src/python-scriptharness/docs/build'}
00:00:05     INFO - Creating directory /src/python-scriptharness/docs/artifacts
00:00:05     INFO - Already exists.
00:00:05     INFO - ### Skipping action clobber
00:00:05     INFO - ### Skipping action pull
00:00:05     INFO - ### Skipping action build
00:00:05     INFO - ### Running action package
00:00:05     INFO - log message from package
00:00:05     INFO - Running command: ['python', '-c', "from __future__ import print_function; print(
00:00:05     INFO - Copy/paste: python -c "from __future__ import print_function; print('hello world
00:00:05     INFO -  hello world!
00:00:05     INFO - ### Action package: finished successfully
00:00:05     INFO - ### Running action upload
00:00:05     INFO - log message from upload
00:00:05     INFO - ### Action upload: finished successfully
00:00:05     INFO - ### Running action notify
00:00:05     INFO - log message from notify
00:00:05     INFO - ### Action notify: finished successfully
00:00:05     INFO - Done.
```

For more, see *Enabling and Disabling Actions*.

--list-actions

If you want to list which actions are available, and which are enabled by default, use the `--list-actions` option:

```
$ ./quickstart.py --list-actions
  clobber ['all']
* pull ['all']
* build ['all']
* package ['all']
  upload ['all']
  notify ['all']
```

--dump-config

You can change the `new_argument` value in the config via the `--new-argument` option that the script added. Also, if you just want to see what the config is without running anything, you can use the `--dump-config` option:

```
$ ./quickstart.py --new-argument foo --dump-config
00:00:14     INFO - Dumping config:
00:00:14     INFO - {'new_argument': 'foo',
00:00:14     INFO -  'scriptharness_artifact_dir': '/src/python-scriptharness/docs/artifacts',
00:00:14     INFO -  'scriptharness_base_dir': '/src/python-scriptharness/docs',
00:00:14     INFO -  'scriptharness_work_dir': '/src/python-scriptharness/docs/build'}
00:00:14     INFO - Creating directory /src/python-scriptharness/docs/artifacts
00:00:14     INFO - Already exists.
```

--help

You can always use the `--help` option:

```
$ ./quickstart.py --help
usage: quickstart.py [-h] [--opt-config-file CONFIG_FILE]
                     [--config-file CONFIG_FILE] [--dump-config]
                     [--actions ACTION [ACTION ...]]
                     [--skip-actions ACTION [ACTION ...]]
                     [--add-actions ACTION [ACTION ...]] [--list-actions]
                     [--action-group {none,all}] [--new-argument NEW_ARGUMENT]

optional arguments:
  -h, --help            show this help message and exit
  --opt-config-file CONFIG_FILE, --opt-cfg CONFIG_FILE
                        Specify optional config files/urls
  --config-file CONFIG_FILE, --cfg CONFIG_FILE, -c CONFIG_FILE
                        Specify required config files/urls
  --dump-config         Log the built configuration and exit.
  --actions ACTION [ACTION ...]
                        Specify the actions to run.
  --skip-actions ACTION [ACTION ...]
                        Specify the actions to skip.
  --add-actions ACTION [ACTION ...]
                        Specify the actions to add to the default set.
  --list-actions        List all actions (default prepended with '*') and
                        exit.
  --action-group {none,all}
                        Specify the action group to use.
  --new-argument NEW_ARGUMENT
                        help message for --new-argument
```

4.2.2 Enabling and Disabling Actions

--action-group

Some actions are enabled by default and others are disabled by default, based on the script. However, sometimes the set of default actions are biased towards developers, or a production environment, and are not the ideal set of default actions for another environment.

Action groups allow for defining other sets of defaults. For example, there could be a *development*, *staging*, or *production* action group for that environment. These would have to be defined in the script.

Consider the following action groups.

Action	development	production
clobber	no	yes
pull	no	yes
prepare-dev-env	yes	no
build	yes	yes
package	yes	yes
upload	no	yes
notify	no	yes

Running the script with `--action-group development` would enable the `prepare-dev-env`, `build`, and `package` actions, while `--action-group production` would enable all actions except for `prepare-dev-env`.

There are also the built-in `all` and `none` groups, that enable all and disable all actions, respectively.

`--actions`

The `--actions` option takes a number of action names as arguments. Those actions will be enabled; all others will be disabled.

`--actions` and `--action-group` are incompatible. Currently `--actions` will override `--action-group` and is not an error.

For an example, see *–actions* in the quickstart.

`--add-actions`

The `--add-actions` option adds a set of actions to the set of already enabled actions. In the above example, `--action-group development --add-actions notify` would enable the `prepare-dev-env`, `build`, `package`, and `notify` actions.

`--skip-actions`

The `--skip-actions` option removes a set of actions from the set of already enabled actions. In the above example, `--action-group development --skip-actions package` would enable the `prepare-dev-env` and `build` actions.

4.2.3 Configuration

Configuration Overview

The runtime configuration of a Script is built from several layers.

- There is a ConfigTemplate that can have default values for certain config variables. These defaults are the basis of the config dict. (See *ConfigTemplates* for more details on ConfigTemplate).
- The script can define an `initial_config` dict that is laid on top of the ConfigTemplate defaults, so any shared config variables are overwritten by the `initial_config`.
- The ConfigTemplate.get_parser() method generates an `argparse.ArgumentParser`. This parser parses the commandline options.

- If the commandline options specify any files via the `--config-file` option, then those files are read, and the contents are overlaid on top of the config. The first file specified will be overlaid first, then the second, and so on.
- If the commandline options specify any *optional* config files via the `--opt-config-file` option, and *if those files exist*, then each existing file is read and the contents are overlaid on top of the config.
- Finally, any other commandline options are overlaid on top of the config.

After the config is built, the script logs the config, and saves it to a `localconfig.json` file. This file can be inspected or reused for a later script run.

ConfigTemplates

It's very powerful to be able to build a configuration dict that can hold any key value pairs, but it's non-trivial for users to verify if their config is valid or if there are options that they're not taking advantage of.

To make the config more well-defined, we have the ConfigTemplate. The ConfigTemplate is comprised of ConfigVariable objects, and is based on the `argparse.ArgumentParser`, but with these qualities:

- The ConfigTemplate can keep track of all config variables, including ones that aren't available as commandline options. The option-less config variables must be specified via default, config file, or `initial_config`.
- The templates can be added together, via ConfigTemplate.update().
- Each ConfigVariable self-validates, and the ConfigTemplate makes sure there are no conflicting commandline options.
- There is a ConfigTemplate.remove_option() method to remove a commandline option from the corresponding ConfigVariable. This may be needed if you want to add two config templates together, but they both have a `-f` commandline option specified, for example.
- The ConfigTemplate.validate_config() method validates the built configuration. Each ConfigVariable can define whether they're required, whether they require or are incompatible with other variables (`required_vars` and `incompatible_vars`), and each can define their own `validate_cb` callback function.
- There is a ConfigTemplate.add_argument() for those who want to maintain argparse syntax.

Parent parsers are supported, to group commandline options in the `--help` output. Subparsers are not currently supported, though it may be possible to replace the ConfigTemplate.parser with a subparser-enabled parser at the expense of validation and the ability to ConfigTemplate.update().

When supporting downstream scripts, it's best to keep each ConfigTemplate modular. It's easy to combine them via ConfigTemplate.update(), but less trivial to remove functionality. The action config template, for instance, can be added to the base config template right before running parse_args().

LoggingDict and ReadOnlyDict

Each Script has a config dict. By default, this dict is a LoggingDict, which logs any changes made to the config.

For example, if the config looked like:

```
{
    "foo": 1,
    "bar": [2, 3, 4],
    "baz": {
        "z": 5,
        "y": 6,
        "x": 7,
```

```
    },
}
```

then updating the config might log:

```
00:11:22  INFO - root.config['baz'] update: y now 8
```

Alternatively, someone could change the script class to StrictScript, which uses ReadOnlyDict. Once the ReadOnlyDict is locked, it cannot be modified.

By either explicitly logging any changes to the config, and/or preventing any changes to the config, it's easier to debug any unexpected behavior.

4.2.4 Scripts and Actions

Scripts and Phases

The Script is generally what one would think of as the script itself: it parses the commandline arguments and runs each enabled Action. There's the possibility of enabling running multiple Scripts in parallel at some point.

It's possible to add callbacks, called listeners, to the Script. These get triggered in phases. The list of phases are in ALL_PHASES; the phases that allow listeners are in LISTENER_PHASES.

- The `PRE_RUN` phase is first, before any actions are run.
- The `PRE_ACTION` phase happens before every enabled action, but a listener can be added to a subset of those actions if desired.
- The `ACTION` phase is when the enabled Action is run. No listener can be added to the `ACTION` phase.
- The `POST_ACTION` phase happens after every enabled action, but a listener can be added to a subset of those actions if desired.
- The `POST_RUN` phase happens after all enabled actions are run.
- The `POST_FATAL` phase happens after a ScriptHarnessFatal exception is raised, but before the script exits.

Contexts

Each listener or Action function is passed a Context. The Context is a `namedtuple` with the following properties:

- script (Script): the Script calling the function
- config (dict): by default this is a LoggingDict
- logger (logging.Logger): the logger for the Script
- action (Action): this is only defined during the `RUN_ACTION`, `PRE_ACTION`, and `POST_ACTION` phases; it is `None` in the other phases.
- phase (str): this will be one of `PRE_RUN`, `POST_RUN`, `PRE_ACTION`, `POST_ACTION`, or `POST_FATAL`, depending on which phase we're in.

The logger and config (and to a lesser degree, the script and action) objects are all available to each function called for convenience and consistency.

Actions

Each action can be enabled or disabled via commandline options (see *Enabling and Disabling Actions*). By default they look for a function with the same name as the action name, with - replaced by _. However, any function or method may be specified as the Action.function.

When run, the Action calls the Action.function with a Context. The function should raise ScriptHarnessError on error, or ScriptHarnessFatal on fatal error.

Afterwards, the Action.history contains the `return_value`, status, `start_time`, and `end_time`.

4.2.5 Commands

Command and run()

The Command object simply takes an external command and runs it, logging stdout and stderr as each message arrives. The main benefits of using Command are logging and timeouts. Command takes two timeouts: `output_timeout`, which is how long the command can go without outputting anything before timing out, and `max_timeout`, which is the total amount of time that can elapse from the start of the command.

(The command is run via `subprocess.Popen` and timeouts are monitored via the *multiprocessing* module.)

After the command is run, it runs the `detect_error_cb` callback function to determine whether the command was run successfully.

The process of creating and running a Command is twofold: Command.__init__() and Command.run(). As a shortcut, there is a run() function that will do both steps for you.

ParsedCommand and parse()

Ideally, external command output would be for humans only, and the exit code would be meaningful. In practice, this is not always the case. Exit codes aren't always helpful or even meaningful, and sometimes critical information is buried in a flood of output.

ParsedCommand takes the output of a command and parses it for matching substrings or regular expressions, using *ErrorLists and OutputParser* to determine the log level of a line of output. Because it subclasses Command, ParsedCommand also has built-in `output_timeout` and `max_timeout` support.

As with Command and run(), ParsedCommand has a shortcut function, parse().

ErrorLists and OutputParser

The ErrorList object describes which lines of output are of special interest. It's a class for better validation.

An example error_list:

```
[
    {
        "regex": re.compile("^Error: not actually an error!"),
        level=-1
    }, {
        "regex": re.compile("^Error:"),
        "level": logging.ERROR,
        "pre_context_lines": 5,
        "post_context_lines": 5
    }, {
```

```
        "substr": "Obscure error #94382",
        "explanation":
            "This is a fatal program error."
        "exception": ScriptHarnessFatal
    }
]
```

Any output line that matches the first regex will be ignored (discarded), because level is negative. Because the list is matched in order, the more specific regex is placed before the more general 2nd regex. If the order were reversed, the more specific regex would never match anything. The second regex sets the level to logging.ERROR for this line, and 5 lines above and 5 lines below this message. (See *OutputBuffer and context lines*.)

The final substring has an explanation that will be logged immediately after the matching line, to explain vague error messages. Because it has a defined *exception*, it will raise.

ParsedCommand sends its output to the OutputParser object, which passes it on to the ErrorList. It keeps track of the number of errors and warnings, as well as handling any context line buffering through the OutputBuffer.

OutputBuffer and context lines

Sometimes there's an obvious error message line, like `make: *** [all] Error 2`, but it's not very helpful without the log context around the line. For those ErrorLists, we can use `pre_context_lines` and `post_context_lines` for the number of lines before and after the matching line, respectively. So if we wanted to mark the 10 lines above the `make: *** [all] Error 2` as errors, as well, then we can do so.

(Long long ago, I would buffer *all* the output of certain commands, notably Visual Studio output, when I either wanted to

- separate threaded logs into easier-to-read unthreaded logs-per-component, or
- search back up above some line, like the first `make` line above `make: *** [all] Error 2`, so we wouldn't have to hardcode some number of `pre_context_lines` and guess how much context is needed.

For the moment, however, we only have `pre_context_lines` and `post_context_lines`.)

The OutputBuffer holds the buffered output for `pre_context_lines`, and keeps track of how many lines in the future will need to be marked at which level for `post_context_lines`.

If multiple lines match, and a line of output is marked as multiple levels, the highest level will win. E.g., `logging.CRITICAL` will beat `logging.ERROR`, which will beat `logging.WARNING`, etc.

Output, get_output(), and get_text_output()

Sometimes you need to manipulate the output from a command, not just log it or perform general error parsing. There's `subprocess.check_output()`, but that doesn't log or have full timeout support.

Enter Output. This also inherits Command, but because Output.run() is a completely different method than Command.run(), it has its own timeout implementation. (It does still support both `output_timeout` and `max_timeout`.) It redirects STDOUT and STDERR to temp files.

Much like Command has its helper run() function, Output has *two* helper functions: get_output() and get_text_output(). The former yields the Output object, and the caller can either access the `NamedTemporaryFile` Output.stdout and Output.stderr objects, or use the Output.get_output() method. Because of this, it is suitable for binary or lengthy output. get_text_output() will get the STDOUT contents for you, log them, and return them to you.

4.2.6 scriptharness package

Submodules

scriptharness.actions module

The goals of *modular actions* are:

- faster development feedback loops, and
- different workflows for different usage requirements.

scriptharness.actions.**LOGGER_NAME**
str

logging.Logger name to use

scriptharness.actions.**STRINGS**
dict

strings for actions. In the future these may be in a function to allow for localization.

class scriptharness.actions.**Action**(*name*, *action_groups=None*, *function=None*, *enabled=True*)
Bases: object

Basic Action object.

name
str

This is the action name, for logging.

enabled
bool

Enabled actions will run. Disabled actions will log the skip_message and not run.

strings
dict

Strings for action-specific log messages.

logger_name
str

The logger name for logging calls inside this object.

function
function

This is the function to call in run_function().

history
dict

History of the action (return_value, status, start_time, end_time).

run(*context*)
Run the action.

This sets self.history timestamps and status.

Parameters **context** (Context) – the context from the calling Script.

Returns **status** – one of SUCCESS, ERROR, or FATAL.

Return type int

Raises *scriptharness.exceptions.ScriptHarnessFatal* – when the function raises ScriptHarnessFatal, run() re-raises.

run_function(*context*)
Run self.function. Called from run() for subclassing purposes.

This sets self.history['return_value'] for posterity.

Parameters **context** (Context) – the context from the calling Script (passed from run()).

scriptharness.actions.**get_function_by_name**(*function_name*)
If function isn't passed to Action, find the function with the same name

This searches in sys.modules['__main__'] and globals() for the function.

Parameters **function_name** (*str*) – The name of the function to find.

Returns the function found.

Return type *function*

Raises scriptharness.exceptions.ScriptHarnesException – if the function is not found or not callable.

scriptharness.commands module

Commands, largely through subprocess.

scriptharness.commands.**LOGGER_NAME**
str

default logging.Logger name.

scriptharness.commands.**STRINGS**
dict

Strings for logging.

class scriptharness.commands.**Command**(*command*, *logger=None*, *detect_error_cb=None*, ***kwargs*)
Bases: object

Basic command: run and log output. Stdout and stderr are interleaved depending on the timing of the message. Because we're logging output, we're expecting text/non-binary output only. For binary output, use the scriptharness.commands.Output object.

command
list or string

The command to send to subprocess.Popen

logger
logging.Logger

logger to log with.

detect_error_cb
function

this function determines whether the command was successful.

history
dict

This dictionary holds the timestamps and status of the command.

kwargs
dict

These kwargs will be passed to subprocess.Popen, except for the optional 'output_timeout' and 'timeout', which are processed by Command. *output_timeout* is how long a command can run without outputting anything to the screen/log. *timeout* is how long the command can run, total.

strings
dict

Strings to log.

add_line(*line*)
Log the output. Here for subclassing.

Parameters **line** (*str*) – a line of output

finish_process()
Here for subclassing.

static **fix_env**(*env*)
Windows environments are fiddly.

Parameters **env** (*dict*) – the environment we'll be passing to subprocess.Popen.

log_env(*env*)
Log environment variables. Here for subclassing.

Parameters **env** (*dict*) – the environment we'll be passing to subprocess.Popen.

log_start()
Log the start of the command, also checking for the existence of cwd if defined.

Raises *scriptharness.exceptions.ScriptHarnessException* – if cwd is defined and doesn't exist.

run()
Run the command.

Raises **scriptharness.exceptions.ScriptHarnessError on error** –

class scriptharness.commands.**Output**(**args*, ***kwargs*)
Bases: *scriptharness.commands.Command*

Run the command and capture stdout and stderr to separate files. The output can be binary or text.

strings
dict

Strings to log.

stdout
NamedTemporaryFile

file to log stdout to

stderr
NamedTemporaryFile

file to log stderr to

+ all of the attributes in scriptharness.commands.Command

cleanup()
Best effort cleanup of stdout and stderr temp files.

finish_process()
Close the filehandles.

get_output(*handle_name=u'stdout'*, *text=True*)
Get output from file. This reads the output into memory, so this is not appropriate for large amounts of output.

Parameters

- **handle_name** (*Optional["stdout" or "stderr"]*) – the handle to read from. Defaults to "stdout"
- **text** (*Optional[bool]*) – whether the output is text. If so, run output through to_unicode() and rstrip(). Defaults to True.

run()

class scriptharness.commands.**ParsedCommand**(*command*, *error_list=None*, *parser=None*, ***kwargs*)
Bases: *scriptharness.commands.Command*

Parse each line of output for errors.

This class could have easily subclassed both OutputParser and Command; that may have been slightly cleaner. However, people have subclassed OutputParser in mozharness for various purposes; keeping the two objects separate may encourage that behavior.

add_line(*line*)
Send the line to the parser.

Parameters **line** (*str*) – a line of output

scriptharness.commands.**check_output**(*command*, *logger_name=u'scriptharness.commands.check_output'*, *level=20*, *log_output=True*, ***kwargs*)
Wrap subprocess.check_output with logging

Parameters

- **command** (*str or list*) – The command to run.
- **logger_name** (*Optional[str]*) – the logger name to log with.
- **level** (*Optional[int]*) – the logging level to log with. Defaults to logging.INFO
- **log_output** (*Optional[bool]*) – When true, log the output of the command. Defaults to True.
- ****kwargs** – sent to *subprocess.check_output()*

scriptharness.commands.**detect_errors**(*command*)
Very basic detect_errors_cb for Command.

This looks in the command.history for return_value. If this is set to 0 or other null value other than None, the command is successful. Otherwise it's unsuccessful.

Parameters **command** (*Command obj*) –

scriptharness.commands.**detect_parsed_errors**(*command*)
Very basic detect_errors_cb for ParsedCommand.

This looks in the command.history for num_errors. If this is set to 0, the command is successful. Otherwise it's unsuccessful.

Parameters **command** (*Command obj*) –

scriptharness.commands.**get_output**(**args*, ***kwds*)

Run command and yield the Output cmd object. The stdout and stderr file paths can be retrieved through cmd.stdout and cmd.stderr, respectively.

The output is not logged, and is written as byte data, so this can work for both binary or text. If text, get_text_output is preferred for full logging, unless the output is either sensitive in nature or so verbose that logging it would be more harmful than useful. Also, if text, most likely the consumer will want to pass the output through scriptharness.unicode.to_unicode().

Parameters

- **command** (*list or str*) – the command to use in subprocess.Popen
- **halt_on_failure** (*Optional[bool]*) – raise ScriptHarnessFatal on error if True. Default: False
- ****kwargs** – kwargs to send to scriptharness.commands.Output

Yields cmd (scriptharness.commands.Output)

Raises *scriptharness.exceptions.ScriptHarnessFatal* – when halt_on_failure is True and we hit an error or timeout.

scriptharness.commands.**get_text_output**(*command*, *level=20*, ***kwargs*)

Run command and return the raw stdout from that command. Because we log the output, we're assuming the output is text.

Parameters

- **command** (*list or str*) – command for subprocess.Popen
- **level** (*int*) – logging level
- ****kwargs** – kwargs to send to scriptharness.commands.Output

Returns **output** – the stdout from the command.

Return type text

scriptharness.commands.**parse**(*command*, ***kwargs*)

Shortcut for running a ParsedCommand.

Not entirely sure if this should also catch ScriptHarnessFatal, as those are explicitly trying to kill the script.

Parameters

- **command** (*list or str*) – Command line to run.
- ****kwargs** – kwargs for run/ParsedCommand.

Returns command exit code (int)

Raises *scriptharness.exceptions.ScriptHarnessFatal* – on fatal error

scriptharness.commands.**run**(*command*, *cmd_class=<class 'scriptharness.commands.Command'>*, *halt_on_failure=False*, **args*, ***kwargs*)

Shortcut for running a Command.

Not entirely sure if this should also catch ScriptHarnessFatal, as those are explicitly trying to kill the script.

Parameters

- **command** (*list or str*) – Command line to run.

- **cmd_class** (*Optional[Command subclass]*) – the class to instantiate. Defaults to scriptharness.commands.Command.
- **halt_on_failure** (*Optional[bool]*) – raise ScriptHarnessFatal on error if True. Default: False
- ****kwargs** – kwargs for subprocess.Popen.

Returns command exit code (int)

Raises *scriptharness.exceptions.ScriptHarnessFatal* – on fatal error

scriptharness.config module

The goal of *flexible configuration* is to make each script useful in a variety of contexts and environments.

scriptharness.config.**LOGGER_NAME**
str

logging.getLogger name

scriptharness.config.**OPTION_REGEX**
re.compile

regular expression to validate a commandline option

scriptharness.config.**VALID_ARGPARSE_ACTIONS**
tuple

for validating the ConfigVariable action

scriptharness.config.**STRINGS**
dict

strings for ConfigVariable

scriptharness.config.**DEFAULT_CONFIG_DEFINITION**
dict

Config definition to create the default ConfigTemplate for all scriptharness scripts.

class scriptharness.config.**ConfigTemplate**(*config_dict*)
Bases: object

Short for Config Template Definition, or CTD. Because scriptharness scripts can take any arbitrary configuration variables or commandline options from various locations, it's difficult to tell what requires what, what's optional, and what's extraneous.

By allowing the developer to create a config template definition, we can check for config well-formedness.

config_variables
dict

a name to ConfigVariable dictionary

parser
argparse.ArgumentParser

this is the commandline parser.

add_argument(**args*, ***kwargs*)
Helper method to make ConfigTemplate usage more similar to ArgumentParser.

add_variable(*definition*, *name=None*)
Add a variable to the config template definition.

See scriptharness.config.ConfigVariable for the definition format.

Parameters

- **name** (*str*) – the variable name. This maps to argparse's *dest*
- **definition** (*dict or ConfigVariable*) – a ConfigVariable or the definition of the config variable.

all_options
Build and return set of all commandline options

Returns **options** – all commandline options

Return type set

defaults()
Get the defaults for all the variables, even the non-commandline ones.

Returns name to default value.

Return type dict

get_parser(***kwargs*)
Create and populate the argparse.ArgumentParser for commandline parsing.

Parameters ****kwargs** – keyword arguments to send to argparse.ArgumentParser.

Returns the commandline parser for this Config Template

Return type argparse.ArgumentParser

items()
Have ConfigTemplate act more like a dict.

Returns self.config_variables.items()

remove_option(*option*)
Remove a commandline option from the ConfigTemplate.

Because we can add templates together, we may sometimes encounter conflicting commandline options. This method allows us to remove those options from one or both templates.

Parameters **option** (*str*) – The commandline option to remove.

update(*config_dict*)
Update self with a new config_dict

Parameters

- **config_dict** (*dict*) – A dict of ConfigVariables or dicts.
- **strict** (*Optional[bool]*) – When True, throw an exception when there's a conflicting variable.

validate_config(*config*)
Validate a config dict against each ConfigVariable.validate_config check.

Parameters **config** (*dict*) – the config dictionary to validate.

Raises *scriptharness.exceptions.ScriptHarnessException* – on error.

class scriptharness.config.**ConfigVariable**(*name*, *definition*)

Bases: object

This object defines what a single config variable looks like.

The variable is overridable from the commandline when when self.definition['options'] is defined. Otherwise the variable is only script-level and config-file-level settable.

The definition will look like this:

```
{
  # argparse-specific, for argparse.ArgumentParser.add_argument
  # if 'options' is not set, these will be ignored.
  'options': ['--foo', '-f'],
  'action': 'store',  # (None, 'store', 'store_const', 'store_true',
                      #  'store_false', 'append', 'append_const',
                      #  'count', 'help', 'version', 'parsers')
                      # defaults to 'store'

  # argparse-related
  # if 'options' is set, these will be used with
  # argparse.ArgumentParser.add_argument; otherwise they're here for
  # the non-commandline-config.
  'help': 'help string',  # not sure whether this should be required
                          # or highly recommended.
  'required': True,
  'default': 'bar',
  'parent_parser': 'parent',  # this is for argparse --help sorting
  'type': str,  # a python type
  'choices': [],  # enum / list of choices

  # Not related to argparse
  'validate_cb': None,  # optional, function to validate the
                        # config.  This function should take the args
                        # (name, parsed_args) and return a list of
                        # error message strings.
  'incompatible_vars': [],  # names of incompatible vars if this var
                            # is set
  'required_vars': [],  # names of other vars that are required to be
                        # set if this var is set
  'optional_vars': [],  # names of other vars that are optionally
                        # used in relation to this var.  This is purely
                        # informational.
}
```

name

str

the name of the variable. This corresponds to the argparse *dest*, or the config dict key.

definition

dict

the config definition for this variable. See above for the format.

add_argument(*parser*)

If self.definition['options'] is set, add the appropriate argument to the parser.

Parameters **parser** (*argparse.ArgumentParser*) – the parser to add the argument to.

Returns on success.

Return type argparse.Action

Raises `ScriptHarnessException` – on argparse.ArgumentParser.add_argument error.

validate_config(*config*)
Once we build the config, we can validate it by sending the built config to each of these methods.

Parameters **config** (*dict*) – the config built from build_config()

Returns **messages** – any error messages, if applicable.

Return type list of strings

scriptharness.config.**action_config_template**(*all_actions*)
Create an action option parser from the action list.

Actions to run are specified as the argparse.REMAINDER options.

Parameters **all_actions** (*iterable*) – this is either all Action objects for the script, or a data structure of pairs of action_name:enabled to pass to iterate_pairs().

Returns with action options

Return type *ConfigTemplate*

scriptharness.config.**build_config**(*template*, *parsed_args*, *initial_config=None*)
Build a configuration dict from the parser and initial config.

The configuration is built in this order:

•template defaults

•initial_config

•parsed_args.config_files, in order

•parsed_args.opt_config_files, in order, if they exist

•non-default parser args (cmdln_args)

So the commandline args can override everything else, as long as there are options to do so. (Commandline args will need to be a subset of the parser args). The final configuration file can override everything but the commandline args, and its config isn't restricted as a subset of the parser options.

Parameters

- **parser** (*ArgumentParser*) – the parser used to parse_args()
- **parsed_args** (*argparse Namespace*) – the results of parse_args()
- **initial_config** (*Optional[dict]*) – initial configuration to set before commandline args

scriptharness.config.**download_url**(*url*, *path=None*, *timeout=None*)
Download a url to a path.

Parameters

- **url** (*str*) – the url to download
- **path** (*Optional[str]*) – the path to write the contents to.
- **timeout** (*Optional[float]*) – how long to wait before timing out.

Returns **path** – the path to the downloaded file.

Return type str

Raises *scriptharness.exceptions.ScriptHarnessException* – if there are download issues, or if we can't write to path.

scriptharness.config.**get_config_template**(*template=None*, *all_actions=None*, *definition=None*)

Create a script ConfigTemplate.

If template is not defined, it will take the definition (defaults to DEFAULT_CONFIG_DEFINITION) and create a new ConfigTemplate. Otherwise it uses template.

If all_actions is defined, it will add an action ConfigTemplate to the template.

Parameters

- **template** (*Optional[ConfigTemplate]*) – the ConfigTemplate to optionally append the action_template to. Defaults to None.
- **all_actions** (*Optional[list]*) – list of actions to generate an action ConfigTemplate. Defaults to None.
- **definition** (*Optional[dict]*) – config definition to prepopulate the ConfigTemplate with. Defaults to DEFAULT_CONFIG_DEFINITION.

Returns ConfigTemplate

scriptharness.config.**get_filename_from_url**(*url*)

Determine the filename of a file from its url.

Parameters **url** (*str*) – the url to parse

Returns **name** – the name of the file

Return type str

scriptharness.config.**get_list_actions_string**(*action_name*, *enabled*, *groups=None*)

Build a string for –list-actions output.

Parameters

- **action_name** (*str*) – name of the action
- **enabled** (*bool*) – whether the action is enabled by default
- **groups** (*Optional[list]*) – a list of action_group names that the action belongs to. Defaults to None.

Returns **string** – a line of –list-actions output.

Return type str

scriptharness.config.**is_url**(*resource*)

Is it a url?

Note: This function will return False for *file://* strings

Parameters **resource** (*str*) – possible url

Returns True if it's a url, False otherwise.

Return type bool

scriptharness.config.**parse_args**(*template*, *cmdln_args=None*, ***kwargs*)

Parse the commandline args.

Parameters

- **template** (ConfigTemplate) – specify the config template to use
- **cmdln_args** (*Optional[list]*) – override the commandline args with these

- ****kwargs** – sent to ConfigTemplate.get_parser() if parser is a ConfigTemplate

Returns tuple(ArgumentParser, parsed_args)

scriptharness.config.**parse_config_file**(*path*)
Read a config file and return a dictionary. For now, only support json.

Parameters **path** (*str*) – path or url to config file.

Returns **config** – the parsed json dict.

Return type dict

Raises *scriptharness.exceptions.ScriptHarnessException* – if the path is unreadable or not valid json.

scriptharness.config.**update_dirs**(*config*, *max_depth=2*)
Directory paths for the script are defined in config. Absolute paths help avoid chdir issues.

scriptharness_base_dir (or any other directory path, or any config value) can be overridden during build_config(). Defining the directory paths as formattable strings is configurable but not overly complex.

Any key in *config* named scriptharness_SOMETHING_dir will be % formatted with the other dirs as the replacement dictionary.

Parameters **config** (*dict*) – the config to parse for scriptharness_SOMETHING_dir keys.

scriptharness.config.**validate_config_definition**(*name*, *definition*)
Validate the ConfigVariable definition's well-formedness.

Parameters

- **name** (*str*) – the name of the variable
- **definition** (*dict*) – the definition to validate

Raises ScriptHarnessException – if there are any error messages

scriptharness.errorlists module

Error lists are used to parse output in scriptharness.log.OutputParser.

Each line of output is matched against each substring or regular expression in the error list. On a match, we determine the 'level' of that line. Levels are ints, and match the levels in the python logging module. Negative levels are ignored.

class scriptharness.errorlists.**ErrorList**(*error_list*, *strict=True*)
Bases: list

Error lists, to describe how to parse output. In object form for better validation.

An example error_list:

```
    [
        {
            "regex": re.compile("^Error: not actually an error!"),
            level=-1
        }, {
            "regex": re.compile("^Error:"),
            "level": logging.ERROR,
            "pre_context_lines": 5,
            "post_context_lines": 5
        }, {
            "substr": "Obscure error #94382",
```

```
            "explanation":
                "This is a fatal program error."
            "exception": ScriptHarnessFatal
        }
    ]
```

Any output line that matches the first regex will be ignored (discarded), because level is negative. Because the list is matched in order, the more specific regex is placed before the more general 2nd regex. If the order were reversed, the more specific regex would never match anything. The second regex sets the level to logging.ERROR for this line, and 5 lines above and 5 lines below this message.

Currently undecided whether we should support modification of ErrorLists (which would require validating any new items and recalculating pre and post context_lines) or having ErrorList inherit tuple and dealing with all the renaming. Most likely the former, but until then, the supported way of modifying an ErrorList is to create a new one.

strict
bool

If True, be more strict about well-formed error_lists.

pre_context_lines
int

The max number of lines the error_list defines in pre_context_lines.

post_context_lines
int

The max number of lines the error_list defines in post_context_lines.

validate_error_list(*error_list*)
Validate an error_list. This is going to be a pain to unit test properly.

Parameters **error_list** (*list of dicts*) – an error_list.

Returns (pre_context_lines, post_context_lines) (tuple of int, int)

Raises *scriptharness.exceptions.ScriptHarnessException* – if error_list is not well-formed.

scriptharness.errorlists.**MAKE_ERROR_LIST = [{u'substr': u'No rule to make target ', u'level': 40}, {u'regex': <_sre**
Make errors. These are prime candidates to add pre_context_lines to.

scriptharness.errorlists.**SSH_ERROR_LIST = [{u'substr': u'Name or service not known', u'level': 40}, {u'substr': u**
For ssh, scp, rsync over ssh.

scriptharness.errorlists.**check_context_lines**(*context_lines*, *orig_context_lines*, *name*, *messages*)
Verifies and returns the larger int of context_lines and orig_context_lines.

Parameters

- **context_lines** (*value*) – The value of pre_context_lines or post_context_lines to validate.
- **orig_context_lines** (*int*) – The previous max int sent to check_context_lines
- **name** (*str*) – The name of the field (pre_context_lines or post_context_lines)
- **messages** (*list*) – The list of error messages so far.

Returns If context_lines is a non-int or negative, an error is appended to messages and we return orig_context_lines. Otherwise, we return the max of context_lines or orig_context_lines.

Return type int

scriptharness.errorlists.**check_ignore**(*strict*, *ignore*, *message*, *messages*)
If the level of an error_check is negative, it will be ignored. There is currently no pre_context_lines or post_context_lines support for ignored lines. When self.strict is True, append an error to messages.

This function doesn't do a whole lot anymore, other than remove the number of branches in validate_error_list.

Parameters

- **strict** (*bool*) – Whether the error-checking is strict or not.
- **ignore** (*bool*) – True when 'level' is in error_check and negative.
- **message** (*str*) – The message to append if ignore and strict.
- **messages** (*list*) – The error messages so far.

scriptharness.errorlists.**exactly_one**(*key1*, *key2*, *error_check*, *messages*)
Make sure one, and only one, of key1 and key2 are in error_check. If that's not the case, append an error message in messages.

Parameters

- **key1** (*str*) – Dictionary key.
- **key2** (*str*) – Dictionary key.
- **error_check** (*dict*) – a single item of error_list.
- **messages** (*list*) – the list of error messages so far.

Returns True if there is exactly one of the two keys in error_check.

Return type Bool

scriptharness.errorlists.**verify_unicode**(*key*, *error_check*, *messages*)
If key is in error_check, it must be of type six.text_type. If not, append an error message to messages.

Parameters

- **key** (*str*) – a dict key
- **error_check** (*dict*) – a single item of error_list
- **messages** (*list*) – The error messages so far

scriptharness.exceptions module

Scriptharness exceptions.

These exceptions are written with several things in mind:

1. the exceptions should be unicode-capable in python 2.7 (py3 gets that for free),
2. the exceptions should differentiate between user-facing exceptions and developer-facing exceptions, and
3. ScriptHarnessFatal should exit the script.

There may be more exceptions in the future, to further differentiate between errors.

exception scriptharness.exceptions.**ScriptHarnessBaseException**
Bases: exceptions.Exception

All scriptharness exceptions should inherit this exception.

However, in most cases you probably want to catch ScriptHarnessException instead.

__unicode__()
This method will become __unicode__() in py2 via the @six.python_2_unicode_compatible decorator.

exception scriptharness.exceptions.**ScriptHarnessError**
Bases: *scriptharness.exceptions.ScriptHarnessBaseException*

User-facing exception.

Scriptharness has detected an error in the running process.

Since this exception is not designed to always exit, it's best to catch these and deal with the error.

exception scriptharness.exceptions.**ScriptHarnessException**
Bases: *scriptharness.exceptions.ScriptHarnessBaseException*

There is a problem in how scriptharness is being called. All developer-facing exceptions should inherit this class.

If you want to catch all developer-facing scriptharness exceptions, catch ScriptHarnessException.

exception scriptharness.exceptions.**ScriptHarnessFatal**
Bases: exceptions.SystemExit, *scriptharness.exceptions.ScriptHarnessBaseException*

User-facing exception.

Scriptharness has detected a fatal failure in the running process. This exception should result in program termination; using try/except may result in unexpected or dangerous behavior.

exception scriptharness.exceptions.**ScriptHarnessTimeout**
Bases: *scriptharness.exceptions.ScriptHarnessException*

There was a timeout while running scriptharness.

scriptharness.log module

The goal of *full logging* is to be able to debug problems purely through the log.

scriptharness.log.**LOGGER_NAME**
str

the default name to use for logging.getLogger()

scriptharness.log.**DEFAULT_DATEFMT**
str

default logging date format

scriptharness.log.**DEFAULT_FMT**
str

default logging format

scriptharness.log.**DEFAULT_LEVEL**
int

default logging level

class scriptharness.log.**LogMethod**(*func=None*, ***kwargs*)
Bases: object

Wrapper decorator object for logging and error detection. This is here as a shortcut to wrap functions with basic logging.

default_config
dict

contains the config defaults that can be overridden via __init__ kwargs. Changing default_config directly may carry over to other decorated LogMethod functions!

__call__ (*func*, **args*, ***kwargs*)
Wrap the function call as a decorator.

When there are decorator arguments, __call__ is only called once, at decorator time. args and kwargs only show up when func is called, so we need to create and return a wrapping function.

Parameters

- **func** (function) – this is the decorated function.
- ***args** – the args from the wrapped function call.
- ****kwargs** – the kwargs from the wrapped function call.

default_config = {u'post_success_msg': u'%(func_name)s completed.', u'error_level': 40, u'post_failure_msg': u'%

post_func()
Log the success message until we get an error detection callback.

This method is split out for easier subclassing.

pre_func()
Log the function call before proceeding.

This method is split out for easier subclassing.

set_repl_dict()
Create a replacement dictionary to format strings.

The log messages in pre_func() and post_func() require some additional info. Specify that info in the replacement dictionary.

Currently, set the following:

```
func_name: self.func.__name__
*args: the args passed to self.func()
**kwargs: the kwargs passed to self.func()
```

After running self.func, we'll also set return_value.

class scriptharness.log.**OutputBuffer**(*logger*, *pre_context_lines*, *post_context_lines*)
Bases: object

Buffer output for context lines: essentially, an error_check can set the level of X lines in the past or Y lines in the future. If multiple error_checks set the level for a line, currently the higher level wins.

For instance, if a `make: *** [all] Error 2` sets the level to logging.ERROR for 10 pre_context_lines, we'll need to buffer at least 10 lines in case we hit that error. If a second error_check sets the level to logging.WARNING 5 lines above the `make: *** [all] Error 2`, the ERROR wins out, and that line is still marked as an ERROR.

This restricts the buffer size to pre_context_lines. In years past I've also ordered Visual Studio output by thread, and set the error all the way up until we match some other pattern, so the buffer had to grow to an arbitrary size. Those could be represented by separate classes/subclasses if needed.

add_line(*level*, *line*, **args*, ***kwargs*)
Add a line to the buffer.

Parameters

- **level** (*int*) – the logging level for the line.
- **line** (*str*) – the line to log
- **pre_context_lines** (*Optional[int]*) – the number of lines before this one to set to log level *level*. This defaults to 0.
- **post_context_lines** (*Optional[int]*) – the number of lines after this one to set to log level *level*. This defaults to 0.

dump_buffer()
Write all the buffered log lines to the log.

pop_buffer(*num=1*)
Pop num lines from the front of the buffer and log them at the level set for each line.

Parameters **num** (*Optional[int]*) – The number of lines to pop and log. Defaults to 1.

update_buffer_levels(*level*, *pre_context_lines*)
Set the level for each buffer line to level if it's higher than the existing level.

Parameters

- **level** (*int*) – The logging level to set the lines to
- **pre_context_lines** (*int*) – The number of lines to affect. Since these are relative to the current line, these will be counted backwards from the end of the buffer.

class scriptharness.log.**OutputParser**(*error_list*, *logger=None*, ***kwargs*)
Bases: object

Helper object to parse command output.

add_buffer(*level*, *messages*, *error_check=None*)
Add the line to self.context_buffer if it exists, otherwise log it.

Parameters

- **level** (*int*) – logging level to log the line at
- **line** (*str*) – line to log
- **error_check** (*Optional[dict]*) – the error_check in error_list that first matched line, if applicable. Defaults to None.

add_line(*line*)
parse a line and check if it matches one in *error_list*, if so then log it.

Parameters **line** (*str*) – a line of output to parse.

class scriptharness.log.**UnicodeFormatter**(*fmt=None*, *datefmt=None*)
Bases: logging.Formatter

Subclass logging.Formatter to handle unicode strings in py2.

encoding
str

defaults to utf-8.

encoding = u'utf-8'

format(*record*)

scriptharness.log.**get_console_handler**(*formatter=None*, *logger=None*, *level=20*)
Create a stream handler to add to a logger.

Parameters

- **formatter** (*Optional[logging.Formatter]*) – formatter to use for logs.
- **logger** (*Optional[logging logger]*) – logger to add the file handler to.
- **level** (*Optional[int]*) – logging level for the file.

Returns logging.StreamHandler handler. This can be added to a logger via logger.addHandler(handler)

scriptharness.log.**get_file_handler**(*path*, *level=20*, *formatter=None*, *logger=None*, *mode=u'w'*)

Create a file handler to add to a logger.

Parameters

- **path** (*str*) – the path to the logfile.
- **level** (*Optional[int]*) – logging level for the file.
- **formatter** (*Optional[logging.Formatter]*) – formatter to use for logs.
- **logger** (*Optional[logging logger]*) – logger to add the file handler to.
- **mode** (*Optional[str]*) – mode to open the file

Returns **handler** – This can be added to a logger via logger.addHandler(handler)

Return type logging.FileHandler

scriptharness.log.**get_formatter**(*fmt=u'%(asctime)s %(levelname)8s - %(message)s'*, *datefmt=u'%H:%M:%S'*)

Create a unicode-friendly formatter to add to logging handlers.

Parameters

- **fmt** (*Optional[str]*) – logging message format.
- **datefmt** (*Optional[str]*) – date format for the log message.

Returns UnicodeFormatter to add to a handler - handler.setFormatter(formatter)

scriptharness.log.**prepare_simple_logging**(*path*, *mode=u'w'*, *logger_name=u''*, *level=20*, *formatter=None*)

Create a unicode-friendly logger.

By default it'll create the root logger with a console handler; if passed a path it'll also create a file handler. Both handlers will have a unicode-friendly formatter.

This function is intended to be called a single time. If called a second time, beware creating multiple console handlers or multiple file handlers writing to the same file.

Parameters

- **path** (*Optional[str]*) – path to the file log. If this isn't set, don't create a file handler. Default ''
- **mode** (*Optional[char]*) – the mode to open the file log. Default 'w'
- **logger_name** (*Optional[str]*) – the name of the logger to use. Default ''
- **level** (*Optional[int]*) – the level to log. Default DEFAULT_LEVEL
- **formatter** (*Optional[Formatter]*) – a logging Formatter to use; to handle unicode, subclass UnicodeFormatter.

Returns logger (Logger object). This is also easily retrievable via logging.getLogger(logger_name).

scriptharness.os module

Wrapping python os and related functions.

param LOGGER_NAME the default logging.Logger name

type LOGGER_NAME str

scriptharness.os.**make_parent_dir**(*path*, ***kwargs*)
Create the parent of path if it doesn't exist.

Parameters

- **path** (*str*) – path to the file.
- ****kwargs** – These are passed to makedirs().

scriptharness.os.**makedirs**(*path*, *level=20*, *context=None*)
os.makedirs() wrapper.

Parameters

- **path** (*str*) – path to the directory
- **level** (*Optional[int]*) – the logging level to log with. Defaults to logging.INFO.

scriptharness.process module

Scriptharness multiprocessing support.

scriptharness.process.**command_subprocess**(*queue*, **args*, ***kwargs*)
Run a subprocess as a multiprocess.Process. This will open STDOUT and STDERR to the same pipe, and read lines from it. Use this with watch_command() for timeout support.

Note: This is intended for non-binary output only.

Parameters

- **queue** (*multiprocessing.Queue*) – the queue to write to
- ***args** – sent to subprocess.Popen
- ****kwargs** – sent to subprocess.Popen

scriptharness.process.**kill_proc_tree**(*pid*, *include_parent=False*, *wait=5*)
Find the children of a process and kill them; optionally also kill the process. Uses psutil, which is cross-platform and py2&3 compatible.

From http://stackoverflow.com/a/4229404

Parameters

- **pid** (*int*) – The process ID of the parent.
- **include_parent** (*Optional[bool]*) – kill the parent as well if True. Defaults to False.
- **wait** (*Optional[int]*) – How long to wait for the children and parent to die. Defaults to 5.

scriptharness.process.**kill_runner**(*runner*)
Kill the runner process and children.

Parameters **runner** (*multiprocessing.Process*) – the process to kill.

scriptharness.process.**watch_command**(*logger*, *queue*, *runner*, *add_line_cb*, *max_timeout=None*, *output_timeout=None*)

This function watches the queue of the command_subprocess process.

Usage:

```
queue = multiprocessing.Queue()
runner = multiprocessing.Process(target=command_subprocess,
                                 args=(queue,))
runner.start()
watch_command(logger, queue, runner, add_line_cb,
              output_timeout=output_timeout, max_timeout=max_timeout)
```

Parameters

- **logger** (*logging.Logger*) – the logger to use.
- **queue** (*multiprocessing.Queue*) – the queue that the runner is writing to.
- **runner** (*multiprocessing.Process*) – the runner Process to watch.
- **add_line_cb** (function) – any output lines read will be sent here.
- **max_timeout** (*Optional[int]*) – when specified, the process will be killed if it takes longer than this number of seconds. Default: None
- **output_timeout** (*Optional[int]*) – when specified, the process will be killed if it doesn't produce any output for this number of seconds. Default: None

Returns **runner.exitcode** – on non-timeout.

Return type int

Raises

- *scriptharness.exceptions.ScriptHarnessFatal* – on KeyboardInterrupt
- *scriptharness.exceptions.ScriptHarnessTimeout* – on output_timeout or max_timeout.

scriptharness.process.**watch_output**(*logger*, *runner*, *stdout*, *stderr*, *max_timeout=None*, *output_timeout=None*)

This function watches the queue of the output_subprocess process.

Usage:

```
runner = multiprocessing.Process(target=output_subprocess, args=(queue,))
runner.start()
watch_output(logger, runner, output_timeout=output_timeout,
             max_timeout=max_timeout)
```

Parameters

- **logger** (*logging.Logger*) – the logger to use.
- **runner** (*subprocess.Popen*) – the runner process to watch.
- **max_timeout** (*Optional[int]*) – when specified, the process will be killed if it takes longer than this number of seconds. Default: None
- **output_timeout** (*Optional[int]*) – when specified, the process will be killed if it doesn't produce any output for this number of seconds. Default: None

Returns **runner.exitcode** – on non-timeout.

Return type int

Raises

- *scriptharness.exceptions.ScriptHarnessFatal* – on KeyboardInterrupt
- *scriptharness.exceptions.ScriptHarnessTimeout* – on output_timeout or max_timeout.

scriptharness.script module

Scripts control the running of Actions.

scriptharness.script.**LOGGER_NAME**
str

logging.Logger name to use

scriptharness.script.**LISTENER_PHASES**
tuple

valid phases for Script.add_listener()

scriptharness.script.**ALL_PHASES**
tuple

valid phases for build_context()

scriptharness.script.**PRE_RUN**
str

the pre-run phase constant

scriptharness.script.**POST_RUN**
str

the post-run phase constant

scriptharness.script.**PRE_ACTION**
str

the pre-action phase constant

scriptharness.script.**POST_ACTION**
str

the post-action phase constant

scriptharness.script.**RUN_ACTION**
str

the run-action phase constant

class scriptharness.script.**Context**(*script*, *config*, *logger*, *action*, *phase*)
Bases: tuple

This is a namedtuple passed to each listener and action function so they can reference the config, logger, etc. easily. It contains pointers to the Script, config, logger, and phase. During action phases it also contains a pointer to the Action; during other phases, Context.action is None.

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

__getstate__()
Exclude the OrderedDict from pickling

__repr__()
Return a nicely formatted representation string

action
Alias for field number 3

config
Alias for field number 1

logger
Alias for field number 2

phase
Alias for field number 4

script
Alias for field number 0

class scriptharness.script.**Script**(*actions*, *template*, *name=u'root'*, ***kwargs*)
Bases: object

This maintains the context of the config + actions.

In general there is a single Script object per run, but the intent is to allow for parallel processing by instantiating multiple Script objects when it makes sense.

config
LoggingDict

the config for the script

actions
tuple

Action objects to run.

name
string

The name of the script

listeners
dict

Callbacks for run(). Listener functions can be set for each of LISTENER_PHASES.

logger
logging.Logger

the logger for the script

add_listener(*listener*, *phase*, *action_names=None*)
Add a callback for a specific script phase.

For pre and post_run, run at the beginning and end of the script, respectively.

For pre and post_action, run at the beginning and end of actions, respectively. If action_names are specified, only run before/after those action(s).

Parameters

- **listener** (function) – Function to call at the right time.

- **phase** (*str*) – When to run the function. Choices in LISTENER_PHASES
- **action_names** (*iterable*) – for pre/post action phase listeners, only run before/after these action(s).

build_config(*template*, *cmdln_args=None*, *initial_config=None*)
Create self.config from the parsed args.

If –dump-config is in the commandline arguments, the script will dump the config to screen and disk, and exit.

Parameters

- **template** (ConfigTemplate) – template to parse and validate the config.
- **cmdln_args** (*Optional[tuple]*) – override the commandline args
- **initial_config** (*Optional[dict]*) – initial config dict to apply.

Returns parsed_args from parse_args()

config = None

dict_to_config(*config*)
Convert the config dict to a LoggingDict.

This method is mainly here for subclassing; otherwise it could have easily stayed part of self.build_config().

end_message()
Log a message at the end of run()

Split out for subclassing; the string may end up moving elsewhere for localizability.

get_logger()
Get a logger to log messages.

This is not strictly needed, as python's logging module will keep track of these loggers.

However, if we support structured logging as well as python logging, get_logger() may return one or the other depending on config.

This method may end up moving to the scriptharness module, and tracked in ScriptManager.

Returns logging.Logger object.

log_enabled_actions()
Log enabled actions.

run()
Run all enabled actions.

run_action(*action*)
Run a specific action.

Parameters **(Action object).** (action) –

Raises

- *scriptharness.exceptions.ScriptHarnessFatal* – when the Action
- **raises ScriptHarnessFatal, this method re-raises.** –

save_config()
Save config to disk.

start_message()
Log a message at the end of __init__()

Split out for subclassing; the string may end up moving elsewhere for localizability.

verify_actions(*actions*)
Make sure actions consists of Action objects, with no duplicate names.

Then set self.actions to a namedtuple so we can find each action by name easily.

Parameters **actions** (*list of Action objects*) – these are passed from __init__().

class scriptharness.script.**StrictScript**(**args*, ***kwargs*)
Bases: *scriptharness.script.Script*

A subclass of Script that uses a ReadOnlyDict for config, and locks its attributes.

As for naming, there were the following choices:

- Locking sounds like Logging;
- ReadOnlyScript is a misnomer;
- StrictScript is a tongue-twister.

_lock
bool

Similar to the ReadOnlyDict _lock. Once set, __setattr__ will raise if any attributes are changed/set.

dict_to_config(*config*)
Set self.config to a ReadOnlyDict and lock.

pre_config_lock()
Stub method for subclassing.

run()

scriptharness.script.**build_context**(*script*, *phase*, *action=None*)
Build context for functions called by Actions.

Parameters

- **script** (Script) – The calling Script
- **phase** (*str*) – The current script phase (one of ALL_PHASES)
- **action** (*Optional[Action]*) – The active Action, if applicable.

Raises *scriptharness.exceptions.ScriptHarnessException* – if there is an invalid phase.

Returns scriptharness.script.Context namedtuple.

scriptharness.script.**enable_actions**(*parsed_args*, *action_list*)
If parsed_args has action-related options, enable/disable actions as appropriate.

Parameters

- **(argparse Namespace)** (*parsed_args*) –
- **(list of Actions)** (*action_list*) –

scriptharness.script.**save_config**(*config*, *path*)
Save the configuration file to path as json.

Parameters

- **config** (*dict*) – The config to save
- **path** (*str*) – The path to write the config to

scriptharness.status module

Statuses for Commands and Actions.

scriptharness.status.**SUCCESS**
int

Constant for Action or Command.history['status']

scriptharness.status.**ERROR**
int

Constant for Action or Command.history['status']

scriptharness.status.**FATAL**
int

Constant for Action or Command.history['status']

scriptharness.structures module

Data structures for configs.

There are two config dict models here:

- LoggingDict logs any changes to the dict or its children. When debugging, config changes will be marked in the log. This is the default model.
- ReadOnlyDict recursively locks the dictionary. This is to aid in debugging; one can assume the config hasn't changed from the moment of locking. This is the original *mozharness* model.

scriptharness.structures.**DEFAULT_LEVEL**
int

the default logging level to set

scriptharness.structures.**DEFAULT_LOGGER_NAME**
str

the default logger name to use

scriptharness.structures.**QUOTES**
tuple

the order of quotes to use for key logging

scriptharness.structures.**LOGGING_STRINGS**
dict

a dict of strings to use for logging, for easier unittesting and potentially for future localization.

scriptharness.structures.**MUTED_LOGGING_STRINGS**
dict

a dict of strings to use for logging when the values in the list/dict shouldn't be logged

scriptharness.structures.**SUPPORTED_LOGGING_TYPES**
dict

a non-logging to logging class map, e.g. dict: LoggingDict. Not currently supporting sets or collections.

class scriptharness.structures.**LockedTuple**
Bases: tuple

A tuple with its children recursively locked.

Tuples are read-only by nature, but we need to be able to recursively lock the contents of the tuple, since the tuple can contain dicts or lists.

Taken straight from mozharness.

__deepcopy__(*memo*)
Return a list on deepcopy.

class scriptharness.structures.**LoggingClass**
Bases: object

General logging methods for the Logging* classes to subclass.

level
int

the logging level for changes

logger_name
str

the logger name to use

name
str

the name of the class for logs

parent
str

the name of the parent, if applicable, for logs

ancestor_child_list(*child_list=None*)
Get the original ancestor of self, and the descending, linear list of descendents' names leading up to (and including) self.

Parameters **child_list** (*list, automatically generated*) – in a multi-level nested Logging* class, generate the list of children's names. This list will be built by prepending our name and calling ancestor_child_list() on self.parent.

Returns **(ancestor, child_list)** – for self.full_name and self.log_change support

Return type LoggingClass, list

full_name()
Get the full name of self.

This will call self.ancestor_child_list to get the original ancestor + all the names of its descendents up to and including self, then build the name from that.

Parameters

- **ancestor** (*Optional[LoggingClass]*) – specify the ancestor
- **child_list** (*Optional[list]*) – a list of descendents' names, in order

Returns **name** – the full name of self.

Return type string

items()
Return dict.items() for dicts, and enumerate(self) for lists+tuples.

This both simplifies recursively_set_parent() and silences pylint complaining that LoggingClass doesn't have an items() method.

The main negative here might be adding an attr items to non-dict data types.

level = None

log_change(*message*, *repl_dict=None*)
Log a change to self.

Parameters **message** (*str*) – The message to log.

logger_name = None

name = None

parent = None

recursively_set_parent(*name=None*, *parent=None*)
Recursively set name + parent.

If our LoggingDict is a multi-level nested Logging* instance, then seeing a log message that something in one of the Logging* instances has changed can be confusing. If we know that it's grandparent[parent][self][child] that has changed, then the log message is helpful.

For each child, set name automatically. For dicts, the name is the key. For everything else, the name is the index.

Parameters

- **name** (*Optional[str]*) – set self.name, for later logging purposes. Defaults to None.
- **parent** (*Optional[Logging*]*) – set self.parent, for logging purposes. Defaults to None.

class scriptharness.structures.**LoggingDict**(*items*, *level=20*, *muted=False*, *logger_name=u'scriptharness.data_structures'*)
Bases: *scriptharness.structures.LoggingClass*, dict

A dict that logs any changes, as do its children.

level
int

the logging level for changes

logger_name
str

the logger name to use

muted
bool

whether our logging messages are muted

strings
dict

a dict of strings to use for messages

__deepcopy__(*memo*)
Return a dict on deepcopy()

child_set_parent(*key*)
When the dict changes, we can just target the specific changed children. Very simple wrapper method.

Parameters **key** (*str*) – the dict key to the child value.

clear()

log_update(*key*, *value*)
Helper method for update(): log one key/value pair at a time.

Parameters

- **key** (*str*) – key to update
- **value** (*any*) – value to set

Returns key (str) if it doesn't exist in self, else None

pop(*key*, *default=None*)

popitem()

setdefault(*key*, *default=None*)

update(*args*)

class scriptharness.structures.**LoggingList**(*items*, *level=20*, *muted=False*, *logger_name=u'scriptharness.data_structures'*)
Bases: *scriptharness.structures.LoggingClass*, list

A list that logs any changes, as do its children.

level
int

the logging level for changes

logger_name
str

the logger name to use

muted
bool

whether our logging messages are muted

strings
dict

a dict of strings to use for messages

__deepcopy__(*memo*)
Return a list on deepcopy.

append(*item*)

child_set_parent(*position=0*)
When the list changes, we either want to change all of the children's names (which correspond to indeces) or a subset of [position:]

extend(*item*)

insert(*position*, *item*)

log_self()
Log the current list.

Since some methods insert values or rearrange them, it'll be easier to debug things if we log the list after those operations.

pop(*position=None*)

remove(*item*)

reverse()

sort(**args*, ***kwargs*)

class scriptharness.structures.**LoggingTuple**
Bases: *scriptharness.structures.LoggingClass*, tuple

A tuple whose children log any changes.

__deepcopy__(*memo*)
Return a tuple on deepcopy.

class scriptharness.structures.**ReadOnlyDict**(**args*, ***kwargs*)
Bases: dict

A dict that is lockable. When locked, any changes raise exceptions.

Slightly modified version of mozharness.base.config.ReadOnlyDict, largely for pylint.

_lock
bool

When locked, the dict is read-only and cannot be unlocked.

__deepcopy__(*memo*)
Create an unlocked ReadOnlyDict on deepcopy()

clear(**args*)

lock()
Recursively lock the dictionary.

pop(**args*)

popitem(**args*)

setdefault(**args*)

update(**args*)

scriptharness.structures.**add_logging_to_obj**(*item*, ***kwargs*)
Recursively add logging to all contents of a LoggingDict.

Any children of supported types will also have logging enabled. Currently supported:: list, tuple, dict.

Parameters **item** (*object*) – a child of a LoggingDict.

Returns A logging version of item, when applicable, or item.

scriptharness.structures.**get_strings**(*instance_type*, *muted=False*)
Get the strings for LoggingClass instance, muted or unmuted

Parameters

- **instance** (*obj*) – LoggingClass instance or 'list' or 'dict'
- **muted** (*Optional[bool]*) – return the MUTED_LOGGING_STRINGS strings if True

scriptharness.structures.**is_logging_class**(*item*)
Determine if a class is one of the Logging* classes.

Parameters **item** (*object*) – the object to check.

scriptharness.structures.**iterate_pairs**(*data*)
Iterate over pairs of a data structure.

Usage:: for key, value in iterate_pairs(data_structure):

```
    :param data: a dict, iterable-of-iterable pairs
```

scriptharness.structures.**make_immutable**(*item*)
Recursively lock all contents of a ReadOnlyDict.

Any children of supported types will also be locked. Currently supported:: list, tuple, dict.

and we locked r on a shallow level, we could still r['b'].append() or r['c']['key2'] = 'value2'. So to avoid that, we need to recursively lock r via make_immutable.

Parameters **item** (*object*) – a child of a ReadOnlyDict.

Returns A locked version of item, when applicable, or item.

scriptharness.unicode module

Scriptharness unicode compatibility.

Once scriptharness drops python 2.x support, this module can go away.

scriptharness.unicode.**to_unicode**(*obj*, *encoding=u'utf-8'*)
Encode a string as unicode in python2.

http://farmdev.com/talks/unicode/

Parameters

- **obj** (*str*) – the string to encode
- **encoding** (*Optional[str]*) – the encoding to use. Defaults to 'utf-8'.

Returns **obj** – the encoded string

Return type *unicode*

scriptharness.version module

Deal with the scriptharness version in semver format.

However, since writing this I've discovered that setuptools and sphinx don't accept all semver formatted versions. It's not clear if this will go away.

When called as a script, this will update ../version.json with the appropriate version info.

scriptharness.version.**__version__**
tuple

semver version - three integers and an optional string.

scriptharness.version.**__version_string__**
str

semver version in string format.

scriptharness.version.**get_version_string**(*version*)
Translate a version tuple into a string.

Specify the __version__ as a tuple for more precise comparisons, and translate it to __version_string__ for when that's needed.

This function exists primarily for easier unit testing.

Parameters **version** (*tuple*) – three ints and an optional string.

Returns **version_string** – the tuple translated into a string per semver.org

Return type str

scriptharness.version.**write_version**(*name=None*, *path=None*)
Write the version info to ../version.json, for setup.py

Parameters

- **name** (*Optional[str]*) – this is for the `write_version(name=__name__)` below. That's one way to both follow the `if __name__ == '__main__':` convention but also allow for full coverage without ignoring parts of the file.
- **path** (*Optional[str]*) – the path to write the version json to. Defaults to ../version.json

Module contents

Scriptharness is a python scripting harness or framework.

Scriptharness' core principles are:

- Full logging.
- Flexible configuration.
- Modular actions.

The top-level module has two main purposes:

1. to serve shortcuts for simple scripts. A single import, and a few function calls should serve for simple workflows.
2. the ScriptManager can keep track of all Script objects if and when a script requires multiple Script objects.

scriptharness.**get_script**(**args*, ***kwargs*)
This will retrieve an existing script or create one and return it.

Parameters

- **actions** (*tuple of Actions*) – When creating a new Script, this is required. When retrieving an existing script, this is ignored/optional.
- **parser** (*argparse.ArgumentParser*) – When creating a new Script, this is required. When retrieving an existing script, this is ignored/optional.
- **name** (*Optional[str]*) – The name of the script to retrieve/create. Defaults to "root". This is a keyword argument, so use name=NAME

- ****kwargs** – kwargs to pass to MANAGER.get_script(); these will be passed to Script.__init__() when creating a new Script. When retrieving an existing script, this is ignored/optional.

Returns The Script instance.

scriptharness.**get_config**(*name=u'root'*)
This will return the config from an existing script.

Parameters **name** (*Optional[str]*) – The name of the script to get the config from. Defaults to "root".

Raises *scriptharness.exceptions.ScriptHarnessException* – if there is no script of name *name*.

Returns **config** – By default scriptharness.structures.LoggingDict

Return type dict

scriptharness.**get_actions**(*all_actions*)
Build a tuple of Action objects for the script.

Parameters **all_actions** (*data structure*) – ordered mapping of action_name:enabled bool, as accepted by iterate_pairs()

Returns tuple of Action objects

scriptharness.**get_actions_from_list**(*all_actions*, *default_actions=None*)
Helper method to generate the ordered mapping for get_actions().

Parameters

- **all_actions** (*list*) – ordered list of all action names
- **default_actions** (*Optional[list]*) – actions that are enabled by default

Returns tuple of Action objects

scriptharness.**get_logger**(*name=u'root'*)
This will return the logger from an existing script.

This function isn't strictly needed, since the logging module keeps track of loggers for you. However, if/when scriptharness supports multiple parallel Script objects, and if/when scriptharness supports structured logging outside of the python logging module, this function will become more important.

Parameters **name** (*Optional[str]*) – The name of the script to get the logger from. Defaults to "root".

Raises *scriptharness.exceptions.ScriptHarnessException* – if there is no script of name *name*.

Returns logger (logging.Logger)

scriptharness.**get_config_template**(*template=None*, *all_actions=None*, *definition=None*)
Create a script ConfigTemplate.

If template is not defined, it will take the definition (defaults to DEFAULT_CONFIG_DEFINITION) and create a new ConfigTemplate. Otherwise it uses template.

If all_actions is defined, it will add an action ConfigTemplate to the template.

Parameters

- **template** (*Optional[ConfigTemplate]*) – the ConfigTemplate to optionally append the action_template to. Defaults to None.

- **all_actions** (*Optional[list]*) – list of actions to generate an action ConfigTemplate. Defaults to None.
- **definition** (*Optional[dict]*) – config definition to prepopulate the ConfigTemplate with. Defaults to DEFAULT_CONFIG_DEFINITION.

Returns ConfigTemplate

scriptharness.**prepare_simple_logging**(*path*, *mode=u'w'*, *logger_name=u''*, *level=20*, *formatter=None*)

Create a unicode-friendly logger.

By default it'll create the root logger with a console handler; if passed a path it'll also create a file handler. Both handlers will have a unicode-friendly formatter.

This function is intended to be called a single time. If called a second time, beware creating multiple console handlers or multiple file handlers writing to the same file.

Parameters

- **path** (*Optional[str]*) – path to the file log. If this isn't set, don't create a file handler. Default ''
- **mode** (*Optional[char]*) – the mode to open the file log. Default 'w'
- **logger_name** (*Optional[str]*) – the name of the logger to use. Default ''
- **level** (*Optional[int]*) – the level to log. Default DEFAULT_LEVEL
- **formatter** (*Optional[Formatter]*) – a logging Formatter to use; to handle unicode, subclass UnicodeFormatter.

Returns logger (Logger object). This is also easily retrievable via logging.getLogger(logger_name).

scriptharness.**set_action_class**(*action_class*)

By default new actions use the scriptharness.actions.Action class. Override here.

Parameters **action_class** (*class*) – use this class for new actions.

scriptharness.**set_script_class**(*script_class*)

By default new scripts use the scriptharness.script.Script class. Override here.

Parameters **script_class** (*class*) – use this class for new scripts.

4.2.7 Scriptharness 0.2.0 Release Notes

date 2015/06/21

Highlights

This release adds *Command and run()*, *ParsedCommand and parse()*, and *Output, get_output(), and get_text_output()* with output_timeout and max_timeout support. ParsedCommand supports context lines (see *OutputBuffer and context lines*).

It also adds *ConfigTemplates*, which allow for specifying what a well-formed configuration looks like for a script, as well as config validation.

What's New

- More ways to enable and disable actions. Now, in addition to `--actions`, there's `--add-actions`, `--skip-actions`, and `--action-group` to change the set of default actions to run. (See *Enabling and Disabling Actions*.)
- Added Command object with cross-platform output_timeout and max_timeout support, with a run() wrapper function for easier use. This is for running external tools with timeouts. (See *Command and run()*.)
  - Added ScriptHarnessTimeout exception
- Added ParsedCommand subclass of Command. Also added a parse() wrapper function for easier use. This is for running external tools, and parsing the output of those tools to detect errors. (See *ParsedCommand and parse()*.)
  - Added ErrorList, OutputParser objects for ParsedCommand error parsing. (See *ErrorLists and OutputParser*.)
  - Added OutputBuffer object for ParsedCommand context lines support. (See *OutputBuffer and context lines*.)
- Added Output object with cross-platform output_timeout and max_timeout support. Also added get_output(), and get_text_output() wrapper functions for easier use. This is for capturing the output of an external tool for later use. (See *Output, get_output(), and get_text_output()*.)
- Added ConfigVariable and ConfigTemplate objects for configuration definition and validation support. See *ConfigTemplates*.
- Added documentation.
- Script.actions is now a namedtuple
- test_config.py no longer hardcodes port 8001.
- Split a number of modules out.
- 100% coverage
- pylint 10.00
- Current issues are tracked on GitHub.

Note: If you've cloned python-scriptharness 0.1.0, you may need to remove the `scriptharness/commands` directory, as it will conflict with the new `scriptharness/commands.py` module.

Historical Release Notes

Scriptharness 0.2.0 Release Notes

date 2015/06/21

Highlights This release adds *Command and run()*, *ParsedCommand and parse()*, and *Output, get_output(), and get_text_output()* with output_timeout and max_timeout support. ParsedCommand supports context lines (see *OutputBuffer and context lines*).

It also adds *ConfigTemplates*, which allow for specifying what a well-formed configuration looks like for a script, as well as config validation.

What's New

- More ways to enable and disable actions. Now, in addition to `--actions`, there's `--add-actions`, `--skip-actions`, and `--action-group` to change the set of default actions to run. (See *Enabling and Disabling Actions*.)
- Added Command object with cross-platform output_timeout and max_timeout support, with a run() wrapper function for easier use. This is for running external tools with timeouts. (See *Command and run()*.)
  - Added ScriptHarnessTimeout exception
- Added ParsedCommand subclass of Command. Also added a parse() wrapper function for easier use. This is for running external tools, and parsing the output of those tools to detect errors. (See *ParsedCommand and parse()*.)
  - Added ErrorList, OutputParser objects for ParsedCommand error parsing. (See *ErrorLists and OutputParser*.)
  - Added OutputBuffer object for ParsedCommand context lines support. (See *OutputBuffer and context lines*.)
- Added Output object with cross-platform output_timeout and max_timeout support. Also added get_output(), and get_text_output() wrapper functions for easier use. This is for capturing the output of an external tool for later use. (See *Output, get_output(), and get_text_output()*.)
- Added ConfigVariable and ConfigTemplate objects for configuration definition and validation support. See *ConfigTemplates*.
- Added documentation.
- Script.actions is now a namedtuple
- test_config.py no longer hardcodes port 8001.
- Split a number of modules out.
- 100% coverage
- pylint 10.00
- Current issues are tracked on GitHub.

Note: If you've cloned python-scriptharness 0.1.0, you may need to remove the `scriptharness/commands` directory, as it will conflict with the new `scriptharness/commands.py` module.

Scriptharness 0.1.0 Release Notes

date 2015/05/25

This is the first scriptharness release.

What's New

- python 2.7, 3.2-3.5 support
- unicode support on 2.7 (3.x gets it for free)
- no more mixins
- no more query_abs_dirs()
- argparse instead of optparse
- virtualenv instead of clone-and-run

- because of virtualenv model, requests instead of urllib2
- LoggingDict to allow and log config changes
- LogMethod decorator to add simple logging to any function or method
- ScriptManager object like logging.Manager
- Action functions can be module-level functions
- multiple Script model, though running multiple Scripts is currently untested
- choice of StrictScript for ReadOnlyDict usage
- all preflight and postflight functions are listeners
- quickstart.py for faster learning curve
- readthedocs + full docstrings for faster learning curve
- 100% coverage
- pylint 10.00

Known Issues

- run_command() and get_output_from_command() are not yet ported
- test_config.py hardcodes port 8001
- 1 broken test on Windows python 2.7: cgi httpserver call downloads cgi script
- 5 disabled tests on Windows python 3.4
- windows console doesn't print or input Unicode http://bugs.python.org/issue1602
- subprocess failing in GUI applications on Windows http://bugs.python.org/issue3905
- currently only one way to enable/disable actions: –actions

4.2.8 Scriptharness

Scriptharness is a framework for writing scripts. There are three core principles: full logging, flexible configuration, and modular actions. The goal of *full logging* is to be able to debug problems purely through the log. The goal of *flexible configuration* is to make each script useful in a variety of contexts and environments. The goals of *modular actions* are a) faster development feedback loops and b) different workflows for different usage requirements.

Full logging

Many scripts log. However, logging can happen sporadically, and it's generally acceptable to run a number of actions silently (e.g., `os.chdir()` will happily change directories with no indication in the log). In *full logging*, the goal is to be able to debug bustage purely through the log.

At the outset, the user can add a generic logging wrapper to any method with minimal fuss. As scriptharness matures, there will be more customized wrappers to use as drop-in replacements for previously-non-logging methods.

Flexible configuration

Many scripts use some sort of configuration, whether hardcoded, in a file, or through the command line. A family of scripts written by the same author(s) may have similar configuration options and patterns, but often times they vary wildly from script to script.

By offering a standard way of accepting configuration options, and then exporting that config to a file for later debugging or replication, scriptharness makes things a bit neater and cleaner and more familiar between scripts.

By either disallowing runtime configuration changes, or by explicitly logging them, scriptharness removes some of the guesswork when debugging bustage.

Modular actions

Scriptharness actions allow for:

- faster development feedback loops. No need to rerun the entirety of a long-running script when trying to debug a single action inside that script.
- different workflows for different usage requirements, such as running standalone versus running in cloud infrastructure

This is in the same spirit of other frameworks that allow for discrete targets, tasks, or actions: make, maven, ansible, and many more.

Running unit tests

Linux and OS X

```
# By default, this will look for python 2.7 + 3.{3,4,5}.
# You can run |tox -e ENV| to run a specific env, e.g. |tox -e py27|
pip install tox
tox
# alternately, ./run_tests.sh
```

Windows

```
# By default, this will look for python 2.7 + 3.4
# You can run |tox -c tox_win.ini -e ENV| to run a specific env, e.g. |tox -c tox_win.ini -e py27|
pip install tox
tox -c win.ini
```

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

S

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W