scriptharness Documentation

Release 0.1.0a0

Aki Sasaki

February 09, 2016

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

Install

```
# This will automatically bring in all requirements.
pip install scriptharness

# To do a full install with docs/testing requirements,
pip install -r requirements.txt
```

CHAPTER 5

Running unit tests

5.1 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
```

5.2 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
```

5.2.1 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
```

5.2.2 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.

5.2.3 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.

5.2.4 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`.

5.2.5 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.

5.2.6 scriptharness package

Submodules

scriptharness.actions module

scriptharness.commands module

scriptharness.config module

scriptharness.errorlists module

scriptharness.exceptions module

scriptharness.log module

scriptharness.os module

scriptharness.process module

scriptharness.script module

scriptharness.status module

scriptharness.structures module

scriptharness.unicode module

scriptharness.version module

Module contents

5.2.7 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 Output-Parser*.)
  - 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

5.2.8 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.

Install

```
# This will automatically bring in all requirements.
pip install scriptharness

# To do a full install with docs/testing requirements,
pip install -r requirements.txt
```

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 6

Indices and tables

- genindex
- modindex
- search