MozBase Documentation

Release 1

Mozilla Automation and Tools team

Jun 27, 2017

Contents

Mozbase is a set of easy-to-use Python packages forming a supplemental standard library for Mozilla. It provides consistency and reduces redundancy in automation and other system-level software. All of Mozilla's test harnesses use mozbase to some degree, including Talos, mochitest, reftest, Autophone, and Eideticker.

In the course of writing automated tests at Mozilla, we found that the same tasks came up over and over, regardless of the specific nature of what we were testing. We figured that consolidating this code into a set of libraries would save us a good deal of time, and so we spent some effort factoring out the best-of-breed automation code into something we named "mozbase" (usually written all in lower case except at the beginning of a sentence).

This is the main documentation for users of mozbase. There is also a project wiki page with notes on development practices and administration.

The documentation is organized by category, then by module. Figure out what you want to do then dive in!

CHAPTER 1

Managing lists of tests

We don't always want to run all tests, all the time. Sometimes a test may be broken, in other cases we only want to run a test on a specific platform or build of Mozilla. To handle these cases (and more), we created a python library to create and use test "manifests", which codify this information.

`manifestparser` — Create and manage test manifests

manifestparser lets you easily create and use test manifests, to control which tests are run under what circumstances.

What manifestparser gives you:

- manifests are ordered lists of tests
- tests may have an arbitrary number of key, value pairs
- the parser returns an ordered list of test data structures, which are just dicts with some keys. For example, a test with no user-specified metadata looks like this:

```
[{'expected': 'pass',
  'path': '/home/mozilla/mozmill/src/manifestparser/manifestparser/tests/testToolbar/
↪testBackForwardButtons.js',
  'relpath': 'testToolbar/testBackForwardButtons.js',
  'name': 'testBackForwardButtons.js',
  'here': '/home/mozilla/mozmill/src/manifestparser/manifestparser/tests',
  'manifest': '/home/mozilla/mozmill/src/manifestparser/manifestparser/tests/manifest.
↪ini',}]
```

The keys displayed here (path, relpath, name, here, and manifest) are reserved keys for manifestparser and any consuming APIs. You can add additional key, value metadata to each test.

Why have test manifests?

It is desirable to have a unified format for test manifests for testing [mozilla-central](http://hg.mozilla.org/mozilla-central), etc.

- It is desirable to be able to selectively enable or disable tests based on platform or other conditions. This should be easy to do. Currently, since many of the harnesses just crawl directories, there is no effective way of disabling a test except for removal from mozilla-central
- It is desriable to do this in a universal way so that enabling and disabling tests as well as other tasks are easily accessible to a wider audience than just those intimately familiar with the specific test framework.
- It is desirable to have other metadata on top of the test. For instance, let's say a test is marked as skipped. It would be nice to give the reason why.

Most Mozilla test harnesses work by crawling a directory structure. While this is straight-forward, manifests offer several practical advantages:

- ability to turn a test off easily: if a test is broken on m-c currently, the only way to turn it off, generally speaking, is just removing the test. Often this is undesirable, as if the test should be dismissed because other people want to land and it can't be investigated in real time (is it a failure? is the test bad? is no one around that knows the test?), then backing out a test is at best problematic. With a manifest, a test may be disabled without removing it from the tree and a bug filed with the appropriate reason:

```
[test_broken.js]
disabled = https://bugzilla.mozilla.org/show_bug.cgi?id=123456
```

- ability to run different (subsets of) tests on different platforms. Traditionally, we've done a bit of magic or had the test know what platform it would or would not run on. With manifests, you can mark what platforms a test will or will not run on and change these without changing the test.

```
[test_works_on_windows_only.js]
skip-if = os != 'win'
```

- ability to markup tests with metadata. We have a large, complicated, and always changing infrastructure. key, value metadata may be used as an annotation to a test and appropriately curated and mined. For instance, we could mark certain tests as randomorange with a bug number, if it were desirable.
- ability to have sane and well-defined test-runs. You can keep different manifests for different test runs and `[include:]` (sub)manifests as appropriate to your needs.

Manifest Format

Manifests are .ini file with the section names denoting the path relative to the manifest:

```
[foo.js]
[bar.js]
[fleem.js]
```

The sections are read in order. In addition, tests may include arbitrary key, value metadata to be used by the harness. You may also have a *[DEFAULT]* section that will give key, value pairs that will be inherited by each test unless overridden:

```
[DEFAULT]
type = restart

[lilies.js]
color = white

[daffodils.js]
color = yellow
type = other
# override type from DEFAULT
```

```
[roses.js]
color = red
```

You can also include other manifests:

```
[include:subdir/anothermanifest.ini]
```

And reference parent manifests to inherit keys and values from the DEFAULT section, without adding possible included tests.

```
[parent:../manifest.ini]
```

Manifests are included relative to the directory of the manifest with the *[include:]* directive unless they are absolute paths.

By default you can use '#' as a comment character. Comments can start a new line, or be inline.

```
[roses.js]
# a valid comment
color = red # another valid comment
```

Comment characters must be preceded by a space, or they will not be treated as comments.

```
[test1.js]
url = https://foo.com/bar#baz
```

The '#baz' anchor will not be stripped off, as it wasn't preceded by a space.

Special variable server-root

There is a special variable called *server-root* used for paths on the system. This variable is deemed a path and will be expanded into its absolute form.

Because of the inheritant nature of the key/value pairs, if one requires a system path, it must be absolute for it to be of any use in any included file.

```
[DEFAULTS]
server-root = ../data

[test1.js]
server-root = test1/data
```

Manifest Conditional Expressions

The conditional expressions used in manifests are parsed using the *ExpressionParser* class.

class `manifestparser.`**`ExpressionParser`**(*text*, *valuemapping*, *strict=False*)

A parser for a simple expression language.

The expression language can be described as follows:

```
EXPRESSION ::= LITERAL | '(' EXPRESSION ')' | '!' EXPRESSION | EXPRESSION OP␣
↪EXPRESSION
OP ::= '==' | '!=' | '<' | '>' | '<=' | '>=' | '&&' | '||'
LITERAL ::= BOOL | INT | IDENT | STRING
```

```
BOOL ::= 'true' | 'false'
INT ::= [0-9]+
IDENT ::= [a-zA-Z_]\w*
STRING ::= '"' [^"] '"' | ''' [^'] '''
```

At its core, expressions consist of booleans, integers, identifiers and. strings. Booleans are one of *true* or *false*. Integers are a series of digits. Identifiers are a series of English letters and underscores. Strings are a pair of matching quote characters (single or double) with zero or more characters inside.

Expressions can be combined with operators: the equals (==) and not equals (!=) operators compare two expressions and produce a boolean. The and (&&) and or (||) operators take two expressions and produce the logical AND or OR value of them, respectively. An expression can also be prefixed with the not (!) operator, which produces its logical negation.

Finally, any expression may be contained within parentheses for grouping.

Identifiers take their values from the mapping provided.

Consumers of this module are expected to pass in a value dictionary for evaluating conditional expressions. A common pattern is to pass the dictionary from the *mozinfo* module.

Data

Manifest Destiny gives tests as a list of dictionaries (in python terms).

- path: full path to the test
- **relpath: relative path starting from the root directory. The root directory** is typically the location of the root manifest, or the source repository. It can be specified at runtime by passing in *rootdir* to *TestManifest*. Defaults to the directory containing the test's ancestor manifest.
- name: file name of the test
- here: the parent directory of the manifest
- manifest: the path to the manifest containing the test

This data corresponds to a one-line manifest:

```
[testToolbar/testBackForwardButtons.js]
```

If additional key, values were specified, they would be in this dict as well.

Outside of the reserved keys, the remaining key, values are up to convention to use. There is a (currently very minimal) generic integration layer in manifestparser for use of all harnesses, *manifestparser.TestManifest*. For instance, if the 'disabled' key is present, you can get the set of tests without disabled (various other queries are doable as well).

Since the system is convention-based, the harnesses may do whatever they want with the data. They may ignore it completely, they may use the provided integration layer, or they may provide their own integration layer. This should allow whatever sort of logic is desired. For instance, if in yourtestharness you wanted to run only on mondays for a certain class of tests:

```
tests = []
for test in manifests.tests:
    if 'runOnDay' in test:
       if calendar.day_name[calendar.weekday(*datetime.datetime.now().
↪timetuple()[:3])].lower() == test['runOnDay'].lower():
          tests.append(test)
    else:
       tests.append(test)
```

To recap: * the manifests allow you to specify test data * the parser gives you this data * you can use it however you want or process it further as you need

Tests are denoted by sections in an .ini file (see http://hg.mozilla.org/automation/manifestparser/file/tip/manifestparser/tests/mozmill-example.ini).

Additional manifest files may be included with an *[include:]* directive:

```
[include:path-to-additional-file.manifest]
```

The path to included files is relative to the current manifest.

The *[DEFAULT]* section contains variables that all tests inherit from.

Included files will inherit the top-level variables but may override in their own *[DEFAULT]* section.

manifestparser Architecture

There is a two- or three-layered approach to the manifestparser architecture, depending on your needs:

1. ManifestParser: this is a generic parser for .ini manifests that facilitates the *[include:]* logic and the inheritence of metadata. Despite the internal variable being called *self.tests* (an oversight), this layer has nothing in particular to do with tests.

2. TestManifest: this is a harness-agnostic integration layer that is test-specific. TestManifest faciliates *skip-if* logic.

3. Optionally, a harness will have an integration layer than inherits from TestManifest if more harness-specific customization is desired at the manifest level.

See the source code at https://github.com/mozilla/mozbase/tree/master/manifestparser and https://github.com/mozilla/mozbase/blob/master/manifestparser/manifestparser.py in particular.

Filtering Manifests

After creating a *TestManifest* object, all manifest files are read and a list of test objects can be accessed via *TestManifest.tests*. However this list contains all test objects, whether they should be run or not. Normally they need to be filtered down only to the set of tests that should be run by the test harness.

To do this, a test harness can call *TestManifest.active_tests*:

```
tests = manifest.active_tests(exists=True, disabled=True, **tags)
```

By default, *active_tests* runs the filters found in `DEFAULT_FILTERS`. It also accepts two convenience arguments:

1. *exists*: if True (default), filter out tests that do not exist on the local file system.
2. *disabled*: if True (default), do not filter out tests containing the 'disabled' key (which can be set by *skip-if* manually).

This works for simple cases, but there are other built-in filters, or even custom filters that can be applied to the *TestManifest*. To do so, add the filter to *TestManifest.filters*:

```
from manifestparser.filters import subsuite
import mozinfo

filters = [subsuite('devtools')]
tests = manifest.active_tests(filters=filters, **mozinfo.info)
```

A filter is a callable that accepts an iterable of test objects and a dictionary of values, and returns a new iterable of test objects. It is possible to define custom filters if the built-in ones are not enough.

class manifestparser.filters.**chunk_by_dir**(*this_chunk*, *total_chunks*, *depth*)
Basic chunking algorithm that splits directories of tests evenly at a given depth.

For example, a depth of 2 means all test directories two path nodes away from the base are gathered, then split evenly across the total number of chunks. The number of tests in each of the directories is not taken into account (so chunks will not contain an even number of tests). All test paths must be relative to the same root (typically the root of the source repository).

Parameters

- **this_chunk** – the current chunk, 1 <= this_chunk <= total_chunks
- **total_chunks** – the total number of chunks
- **depth** – the minimum depth of a subdirectory before it will be considered unique

class manifestparser.filters.**chunk_by_runtime**(*this_chunk*, *total_chunks*, *runtimes*, *default_runtime=0*)
Chunking algorithm that attempts to group tests into chunks based on their average runtimes. It keeps manifests of tests together and pairs slow running manifests with fast ones.

Parameters

- **this_chunk** – the current chunk, 1 <= this_chunk <= total_chunks
- **total_chunks** – the total number of chunks
- **runtimes** – dictionary of test runtime data, of the form {<test path>: <average runtime>}
- **default_runtime** – value in seconds to assign tests that don't exist in the runtimes file

class manifestparser.filters.**chunk_by_slice**(*this_chunk*, *total_chunks*, *disabled=False*)
Basic chunking algorithm that splits tests evenly across total chunks.

Parameters

- **this_chunk** – the current chunk, 1 <= this_chunk <= total_chunks
- **total_chunks** – the total number of chunks
- **disabled** – Whether to include disabled tests in the chunking algorithm. If False, each chunk contains an equal number of non-disabled tests. If True, each chunk contains an equal number of tests (default False)

manifestparser.filters.**enabled**(*tests*, *values*)
Removes all tests containing the *disabled* key. This filter can be added by passing *disabled=False* into *active_tests*.

manifestparser.filters.**exists**(*tests*, *values*)
Removes all tests that do not exist on the file system. This filter is added by default, but can be removed by passing *exists=False* into *active_tests*.

manifestparser.filters.**fail_if**(*tests*, *values*)
Sets expected to 'fail' on all tests containing the *fail-if* tag and whose condition is True. This filter is added by default.

class manifestparser.filters.**pathprefix**(*paths*)
Removes tests that don't start with any of the given test paths.

Parameters **paths** – A list of test paths to filter on

manifestparser.filters.**run_if**(*tests*, *values*)
Sets disabled on all tests containing the *run-if* tag and whose condition is False. This filter is added by default.

manifestparser.filters.**skip_if**(*tests*, *values*)
Sets disabled on all tests containing the *skip-if* tag and whose condition is True. This filter is added by default.

class manifestparser.filters.**subsuite**(*name=None*)
If *name* is None, removes all tests that have a *subsuite* key. Otherwise removes all tests that do not have a subsuite matching *name*.

It is possible to specify conditional subsuite keys using: subsuite = foo,condition

where 'foo' is the subsuite name, and 'condition' is the same type of condition used for skip-if. If the condition doesn't evaluate to true, the subsuite designation will be removed from the test.

Parameters **name** – The name of the subsuite to run (default None)

class manifestparser.filters.**tags**(*tags*)
Removes tests that don't contain any of the given tags. This overrides InstanceFilter's __eq__ method, so multiple instances can be added. Multiple tag filters is equivalent to joining tags with the AND operator.

To define a tag in a manifest, add a *tags* attribute to a test or DEFAULT section. Tests can have multiple tags, in which case they should be whitespace delimited. For example:

[test_foobar.html] tags = foo bar

Parameters **tags** – A tag or list of tags to filter tests on

manifestparser.filters.**DEFAULT_FILTERS**
By default `active_tests()` will run the *skip_if()*, *run_if()* and *fail_if()* filters.

For example, suppose we want to introduce a new key called *timeout-if* that adds a 'timeout' property to a test if a certain condition is True. The syntax in the manifest files will look like this:

```
[test_foo.py]
timeout-if = 300, os == 'win'
```

The value is <timeout>, <condition> where condition is the same format as the one in *skip-if*. In the above case, if os == 'win', a timeout of 300 seconds will be applied. Otherwise, no timeout will be applied. All we need to do is define the filter and add it:

```
from manifestparser.expression import parse
import mozinfo

def timeout_if(tests, values):
    for test in tests:
        if 'timeout-if' in test:
            timeout, condition = test['timeout-if'].split(',', 1)
            if parse(condition, **values):
                test['timeout'] = timeout
        yield test

tests = manifest.active_tests(filters=[timeout_if], **mozinfo.info)
```

Creating Manifests

manifestparser comes with a console script, *manifestparser create*, that may be used to create a seed manifest structure from a directory of files. Run *manifestparser help create* for usage information.

Copying Manifests

To copy tests and manifests from a source:

```
manifestparser [options] copy from_manifest to_directory -tag1 -tag2 `key1=value1␣
↪key2=value2 ...
```

Updating Tests

To update the tests associated with with a manifest from a source directory:

```
manifestparser [options] update manifest from_directory -tag1 -tag2 `key1=value1␣
↪`key2=value2 ...
```

Usage example

Here is an example of how to create manifests for a directory tree and update the tests listed in the manifests from an external source.

Creating Manifests

Let's say you want to make a series of manifests for a given directory structure containing *.js* test files:

```
testing/mozmill/tests/firefox/
testing/mozmill/tests/firefox/testAwesomeBar/
testing/mozmill/tests/firefox/testPreferences/
testing/mozmill/tests/firefox/testPrivateBrowsing/
testing/mozmill/tests/firefox/testSessionStore/
testing/mozmill/tests/firefox/testTechnicalTools/
testing/mozmill/tests/firefox/testToolbar/
testing/mozmill/tests/firefox/restartTests
```

You can use *manifestparser create* to do this:

```
$ manifestparser help create
Usage: manifestparser.py [options] create directory <directory> <...>

     create a manifest from a list of directories

Options:
  -p PATTERN, `pattern=PATTERN
                        glob pattern for files
  -i IGNORE, `ignore=IGNORE
                        directories to ignore
  -w IN_PLACE, --in-place=IN_PLACE
                        Write .ini files in place; filename to write to
```

We only want *.js* files and we want to skip the *restartTests* directory. We also want to write a manifest per directory, so I use the *–in-place* option to write the manifests:

```
manifestparser create . -i restartTests -p '*.js' -w manifest.ini
```

This creates a manifest.ini per directory that we care about with the JS test files:

```
testing/mozmill/tests/firefox/manifest.ini
testing/mozmill/tests/firefox/testAwesomeBar/manifest.ini
testing/mozmill/tests/firefox/testPreferences/manifest.ini
testing/mozmill/tests/firefox/testPrivateBrowsing/manifest.ini
testing/mozmill/tests/firefox/testSessionStore/manifest.ini
testing/mozmill/tests/firefox/testTechnicalTools/manifest.ini
testing/mozmill/tests/firefox/testToolbar/manifest.ini
```

The top-level *manifest.ini* merely has *[include:]* references to the sub manifests:

```
[include:testAwesomeBar/manifest.ini]
[include:testPreferences/manifest.ini]
[include:testPrivateBrowsing/manifest.ini]
[include:testSessionStore/manifest.ini]
[include:testTechnicalTools/manifest.ini]
[include:testToolbar/manifest.ini]
```

Each sub-level manifest contains the (*.js*) test files relative to it.

Updating the tests from manifests

You may need to update tests as given in manifests from a different source directory. *manifestparser update* was made for just this purpose:

```
Usage: manifestparser [options] update manifest directory -tag1 -tag2 `key1=value1 --
↪key2=value2 ...

    update the tests as listed in a manifest from a directory
```

To update from a directory of tests in *~/mozmill/src/mozmill-tests/firefox/* run:

```
manifestparser update manifest.ini ~/mozmill/src/mozmill-tests/firefox/
```

Tests

manifestparser includes a suite of tests.

test_manifest.txt is a doctest that may be helpful in figuring out how to use the API. Tests are run via *mach python-test testing/mozbase/manifestparser*.

Bugs

Please file any bugs or feature requests at

https://bugzilla.mozilla.org/enter_bug.cgi?product=Testing&component=ManifestParser

Or contact jhammel @mozilla.org or in #ateam on irc.mozilla.org

CLI

Run *manifestparser help* for usage information.

To create a manifest from a set of directories:

```
manifestparser [options] create directory <directory> <...> [create-options]
```

To output a manifest of tests:

```
manifestparser [options] write manifest <manifest> <...> -tag1 -tag2 --key1=value1 --
↪key2=value2 ...
```

To copy tests and manifests from a source:

```
manifestparser [options] copy from_manifest to_manifest -tag1 -tag2 `key1=value1␣
↪key2=value2 ...
```

To update the tests associated with with a manifest from a source directory:

```
manifestparser [options] update manifest from_directory -tag1 -tag2 --key1=value1 --
↪key2=value2 ...
```

Design Considerations

Contrary to some opinion, manifestparser.py and the associated .ini format were not magically plucked from the sky but were descended upon through several design considerations.

- test manifests should be ordered. While python 2.6 and greater has a ConfigParser that can use an ordered dictionary, it is a requirement that we support python 2.4 for the build + testing environment. To that end, a *read_ini* function was implemented in manifestparser.py that should be the equivalent of the .ini dialect used by ConfigParser.
- the manifest format should be easily human readable/writable. While there was initially some thought of using JSON, there was pushback that JSON was not easily editable. An ideal manifest format would degenerate to a line-separated list of files. While .ini format requires an additional *[]* per line, and while there have been complaints about this, hopefully this is good enough.
- python does not have an in-built YAML parser. Since it was undesirable for manifestparser.py to have any dependencies, YAML was dismissed as a format.
- we could have used a proprietary format but decided against it. Everyone knows .ini and there are good tools to deal with it. However, since read_ini is the only function that transforms a manifest to a list of key, value pairs, while the implications for changing the format impacts downstream code, doing so should be programmatically simple.
- there should be a single file that may easily be transported. Traditionally, test harnesses have lived in mozilla-central. This is less true these days and it is increasingly likely that more tests will not live in mozilla-central going forward. So *manifestparser.py* should be highly consumable. To this end, it is a single file, as appropriate to mozilla-central, which is also a working python package deployed to PyPI for easy installation.

Historical Reference

Date-ordered list of links about how manifests came to be where they are today:

```
* https://wiki.mozilla.org/Auto-tools/Projects/UniversalManifest
* http://alice.nodelman.net/blog/post/2010/05/
* http://alice.nodelman.net/blog/post/universal-manifest-for-unit-tests-a-proposal/
* https://elvis314.wordpress.com/2010/07/05/improving-personal-hygiene-by-adjusting-
↪mochitests/
* https://elvis314.wordpress.com/2010/07/27/types-of-data-we-care-about-in-a-manifest/
```

```
* https://bugzilla.mozilla.org/show_bug.cgi?id=585106
* http://elvis314.wordpress.com/2011/05/20/converting-xpcshell-from-listing-
↪directories-to-a-manifest/
* https://bugzilla.mozilla.org/show_bug.cgi?id=616999
* https://developer.mozilla.org/en/Writing_xpcshell-based_unit_tests#Adding_your_
↪tests_to_the_xpcshell_manifest
```

CHAPTER 2

Getting information on the system under test

It's often necessary to get some information about the system we're testing, for example to turn on or off some platform specific behaviour.

`mozinfo` — Get system information

Throughout mozmill and other Mozilla python code, checking the underlying platform is done in many different ways. The various checks needed lead to a lot of copy+pasting, leaving the reader to wonder....is this specific check necessary for (e.g.) an operating system? Because information is not consolidated, checks are not done consistently, nor is it defined what we are checking for.

mozinfo proposes to solve this problem. mozinfo is a bridge interface, making the underlying (complex) plethora of OS and architecture combinations conform to a subset of values of relevance to Mozilla software. The current implementation exposes relevant keys and values such as: `os`, `version`, `bits`, and `processor`. Additionally, the service pack in use is available on the windows platform.

API Usage

mozinfo is a python package. Downloading the software and running `python setup.py develop` will allow you to do `import mozinfo` from python. mozinfo.py is the only file contained is this package, so if you need a single-file solution, you can just download or call this file through the web.

The top level attributes (`os`, `version`, `bits`, `processor`) are available as module globals:

```
if mozinfo.os == 'win': ...
```

In addition, mozinfo exports a dictionary, `mozinfo.info`, that contain these values. mozinfo also exports:

- `choices`: a dictionary of possible values for os, bits, and processor
- `main`: the console_script entry point for mozinfo
- `unknown`: a singleton denoting a value that cannot be determined

`unknown` has the string representation `"UNKNOWN"`. `unknown` will evaluate as `False` in python:

```
if not mozinfo.os: ... # unknown!
```

Command Line Usage

mozinfo comes with a command line program, `mozinfo` which may be used to diagnose one's current system.

Example output:

```
os: linux
version: Ubuntu 10.10
bits: 32
processor: x86
```

Three of these fields, os, bits, and processor, have a finite set of choices. You may display the value of these choices using `mozinfo --os`, `mozinfo --bits`, and `mozinfo --processor`. `mozinfo --help` documents command-line usage.

`mozinfo.`**`update`**(*new_info*)
Update the info.

Parameters **`new_info`** – Either a dict containing the new info or a path/url to a json file containing the new info.

`mozinfo.`**`find_and_update_from_json`**(**dirs*)
Find a mozinfo.json file, load it, and update the info with the contents.

Parameters **`dirs`** – Directories in which to look for the file. They will be searched after first looking in the root of the objdir if the current script is being run from a Mozilla objdir.

Returns the full path to mozinfo.json if it was found, or None otherwise.

class `mozinfo.`**`StringVersion`**(*vstring*)
A string version that can be compared with comparison operators.

`moznetwork` — Get network information

moznetwork is a very simple module designed for one task: getting the network address of the current machine.

Example usage:

```
import moznetwork

try:
    ip = moznetwork.get_ip()
    print "The external IP of your machine is '%s'" % ip
except moznetwork.NetworkError:
    print "Unable to determine IP address of machine"
    raise
```

`moznetwork.`**`get_ip`**()
Provides an available network interface address, for example "192.168.1.3".

A *NetworkError* exception is raised in case of failure.

mozversion — Get application information

mozversion provides version information such as the application name and the changesets that it has been built from. This is commonly used in reporting or for conditional logic based on the application under test.

Note that mozversion can report the version of remote devices (e.g. Firefox OS) but it requires the mozdevice dependency in that case. You can require it along with mozversion by using the extra *device* dependency:

```
pip install mozversion[device]
```

API Usage

mozversion.**get_version**(*binary=None*, *sources=None*, *host=None*, *device_serial=None*, *adb_host=None*, *adb_port=None*)

Returns the application version information as a dict. You can specify a path to the binary of the application or an Android APK file (to get version information for Firefox for Android). If this is omitted then the current directory is checked for the existance of an application.ini file. If not found and that the binary path was not specified, then it is assumed the target application is a remote Firefox OS instance.

Parameters

- **binary** – Path to the binary for the application or Android APK file
- **sources** – Path to the sources.xml file (Firefox OS)
- **host** – Host address of remote Firefox OS instance (not used with ADB)
- **device_serial** – Serial identifier of Firefox OS device (ADB)
- **adb_host** – Host address of ADB server
- **adb_port** – Port of ADB server

Examples

Firefox:

```
import mozversion

version = mozversion.get_version(binary='/path/to/firefox-bin')
for (key, value) in sorted(version.items()):
    if value:
        print '%s: %s' % (key, value)
```

Firefox for Android:

```
version = mozversion.get_version(binary='path/to/firefox.apk')
print version['application_changeset'] # gets hg revision of build
```

FirefoxOS:

```
version = mozversion.get_version(sources='path/to/sources.xml', dm_type='adb')
print version['gaia_changeset'] # gets gaia git revision
```

Command Line Usage

mozversion comes with a command line program, `mozversion` which may be used to get version information from an application.

Usage:

```
mozversion [options]
```

Options

—binary

This is the path to the target application binary or .apk. If this is omitted then the current directory is checked for the existance of an application.ini file. If not found, then it is assumed the target application is a remote Firefox OS instance.

—sources

The path to the sources.xml that accompanies the target application (Firefox OS only). If this is omitted then the current directory is checked for the existance of a sources.xml file.

Examples

Firefox:

```
$ mozversion --binary=/path/to/firefox-bin
application_buildid: 20131205075310
application_changeset: 39faf812aaec
application_name: Firefox
application_repository: http://hg.mozilla.org/releases/mozilla-release
application_version: 26.0
platform_buildid: 20131205075310
platform_changeset: 39faf812aaec
platform_repository: http://hg.mozilla.org/releases/mozilla-release
```

Firefox for Android:

```
$ mozversion --binary=/path/to/firefox.apk
```

Firefox OS:

```
$ mozversion --sources=/path/to/sources.xml
application_buildid: 20140106040201
application_changeset: 14ac61461f2a
application_name: B2G
application_repository: http://hg.mozilla.org/mozilla-central
application_version: 29.0a1
build_changeset: 59605a7c026ff06cc1613af3938579b1dddc6cfe
device_firmware_date: 1380051975
device_firmware_version_incremental: 139
device_firmware_version_release: 4.0.4
device_id: msm7627a
gaia_changeset: 9a222ac02db176e47299bb37112ae40aeadbeca7
```

```
gaia_date: 1389005812
gecko_changeset: 3a2d8af198510726b063a217438fcf2591f4dfcf
platform_buildid: 20140106040201
platform_changeset: 14ac61461f2a
platform_repository: http://hg.mozilla.org/mozilla-central
```

CHAPTER 3

Set up and running

Activities under this domain include installing the software, creating a profile (a set of configuration settings), running a program in a controlled environment such that it can be shut down safely, and correctly handling the case where the system crashes.

`mozfile` — File utilities for use in Mozilla testing

mozfile is a convenience library for taking care of some common file-related tasks in automated testing, such as extracting files or recursively removing directories.

`mozfile.`**`extract`**(*src*, *dest=None*)
Takes in a tar or zip file and extracts it to dest

If dest is not specified, extracts to os.path.dirname(src)

Returns the list of top level files that were extracted

`mozfile.`**`extract_tarball`**(*src*, *dest*)
extract a .tar file

`mozfile.`**`extract_zip`**(*src*, *dest*)
extract a zip file

`mozfile.`**`move`**(*src*, *dst*)
Move a file or directory path.

This is a replacement for shutil.move that works better under windows, retrying operations on some known errors due to various things keeping a handle on file paths.

`mozfile.`**`remove`**(*path*)
Removes the specified file, link, or directory tree.

This is a replacement for shutil.rmtree that works better under windows. It does the following things:

- check path access for the current user before trying to remove

- retry operations on some known errors due to various things keeping a handle on file paths - like explorer, virus scanners, etc. The known errors are errno.EACCES and errno.ENOTEMPTY, and it will retry up to 5 five times with a delay of (failed_attempts * 0.5) seconds between each attempt.

Note that no error will be raised if the given path does not exists.

Parameters **path** – path to be removed

mozinstall — Install and uninstall Gecko-based applications

mozinstall is a small python module with several convenience methods useful for installing and uninstalling a gecko-based application (e.g. Firefox) on the desktop.

Simple example

```
import mozinstall
import tempfile

tempdir = tempfile.mkdtemp()
firefox_dmg = 'firefox-38.0a1.en-US.mac.dmg'
install_folder = mozinstall.install(src=firefox_dmg, dest=tempdir)
binary = mozinstall.get_binary(install_folder, 'Firefox')
# from here you can execute the binary directly
# ...
mozinstall.uninstall(install_folder)
```

API Documentation

mozinstall.**is_installer**(*src*)

Tests if the given file is a valid installer package.

Supported types: Linux: tar.gz, tar.bz2 Mac: dmg Windows: zip, exe

On Windows pefile will be used to determine if the executable is the right type, if it is installed on the system.

Parameters **src** – Path to the install file.

mozinstall.**install**(*src*, *dest*)

Install a zip, exe, tar.gz, tar.bz2 or dmg file, and return the path of the installation folder.

Parameters

- **src** – Path to the install file
- **dest** – Path to install to (to ensure we do not overwrite any existent files the folder should not exist yet)

mozinstall.**get_binary**(*path*, *app_name*)

Find the binary in the specified path, and return its path. If binary is not found throw an InvalidBinary exception.

Parameters

- **path** – Path within to search for the binary
- **app_name** – Application binary without file extension to look for

mozinstall.**uninstall**(*install_folder*)
Uninstalls the application in the specified path. If it has been installed via an installer on Windows, use the uninstaller first.

Parameters **install_folder** – Path of the installation folder

exception mozinstall.**InstallError**
Thrown when installation fails. Includes traceback if available.

exception mozinstall.**InvalidBinary**
Thrown when the binary cannot be found after the installation.

exception mozinstall.**InvalidSource**
Thrown when the specified source is not a recognized file type.

Supported types: Linux: tar.gz, tar.bz2 Mac: dmg Windows: zip, exe

mozprofile — Create and modify Mozilla application profiles

Mozprofile is a python tool for creating and managing profiles for Mozilla's applications (Firefox, Thunderbird, etc.). In addition to creating profiles, mozprofile can install addons and set preferences Mozprofile can be utilized from the command line or as an API.

The preferred way of setting up profile data (addons, permissions, preferences etc) is by passing them to the *profile* constructor.

Addons

exception mozprofile.addons.**AddonFormatError**
Exception for not well-formed add-on manifest files

class mozprofile.addons.**AddonManager**(*profile*, *restore=True*)
Handles all operations regarding addons in a profile including: installing and cleaning addons

classmethod **addon_details**(*addon_path*)
Returns a dictionary of details about the addon.

Parameters **addon_path** – path to the add-on directory or XPI

Returns:

```
{'id':      u'rainbow@colors.org', # id of the addon
 'version': u'1.4',                # version of the addon
 'name':    u'Rainbow',            # name of the addon
 'unpack':  False }                # whether to unpack the addon
```

clean()
Clean up addons in the profile.

classmethod **download**(*url*, *target_folder=None*)
Downloads an add-on from the specified URL to the target folder

Parameters

- **url** – URL of the add-on (XPI file)
- **target_folder** – Folder to store the XPI file in

get_addon_path(*addon_id*)
Returns the path to the installed add-on

Parameters **addon_id** – id of the add-on to retrieve the path from

classmethod get_amo_install_path(*query*)
Get the addon xpi install path for the specified AMO query.

Parameters **query** – query-documentation

install_addons(*addons=None*, *manifests=None*)
Installs all types of addons

Parameters

- **addons** – a list of addon paths to install
- **manifest** – a list of addon manifests to install

install_from_manifest(*filepath*)
Installs addons from a manifest :param filepath: path to the manifest of addons to install

install_from_path(*path*, *unpack=False*)
Installs addon from a filepath, url or directory of addons in the profile.

Parameters

- **path** – url, path to .xpi, or directory of addons
- **unpack** – whether to unpack unless specified otherwise in the install.rdf

classmethod is_addon(*addon_path*)
Checks if the given path is a valid addon

Parameters **addon_path** – path to the add-on directory or XPI

remove_addon(*addon_id*)
Remove the add-on as specified by the id

Parameters **addon_id** – id of the add-on to be removed

Addons may be installed individually or from a manifest.

Example:

```
from mozprofile import FirefoxProfile

# create new profile to pass to mozmill/mozrunner
profile = FirefoxProfile(addons=["adblock.xpi"])
```

Command Line Interface

Creates and/or modifies a Firefox profile. The profile can be modified by passing in addons to install or preferences to set. If no profile is specified, a new profile is created and the path of the resulting profile is printed.

class mozprofile.cli.**MozProfileCLI**(*args=['-b', 'latex', '-D', 'language=en', '-d', '_build/doctrees', '.', '_build/latex'], add_options=None*)
The Command Line Interface for mozprofile.

preferences()
profile preferences

profile(*restore=False*)
create the profile

profile_args()
arguments to instantiate the profile class

profile_class
alias of Profile

mozprofile.cli.**cli**(*args=['-b', 'latex', '-D', 'language=en', '-d', '_build/doctrees', '.', '_build/latex']*)
Handles the command line arguments for mozprofile via sys.argv

The profile to be operated on may be specified with the --profile switch. If a profile is not specified, one will be created in a temporary directory which will be echoed to the terminal:

```
(mozmill)> mozprofile
/tmp/tmp4q1iEU.mozrunner
(mozmill)> ls /tmp/tmp4q1iEU.mozrunner
user.js
```

To run mozprofile from the command line enter: mozprofile --help for a list of options.

Permissions

add permissions to the profile

exception mozprofile.permissions.**MissingPrimaryLocationError**
No primary location defined in locations file.

exception mozprofile.permissions.**MultiplePrimaryLocationsError**
More than one primary location defined.

exception mozprofile.permissions.**DuplicateLocationError**(*url*)
Same location defined twice.

exception mozprofile.permissions.**BadPortLocationError**(*given_port*)
Location has invalid port value.

exception mozprofile.permissions.**LocationsSyntaxError**(*lineno*, *err=None*)
Signifies a syntax error on a particular line in server-locations.txt.

class mozprofile.permissions.**Location**(*scheme*, *host*, *port*, *options*)
Represents a location line in server-locations.txt.

isEqual(*location*)
compare scheme://host:port, but ignore options

class mozprofile.permissions.**ServerLocations**(*filename=None*, *add_callback=None*)
Iterable collection of locations. Use provided functions to add new locations, rather that manipulating _locations directly, in order to check for errors and to ensure the callback is called, if given.

read(*filename*, *check_for_primary=True*)
Reads the file and adds all valid locations to the self._locations array.

Parameters

- **filename** – in the format of server-locations.txt
- **check_for_primary** – if True, a MissingPrimaryLocationError exception is raised if no primary is found

The only exception is that the port, if not defined, defaults to 80 or 443.

FIXME: Shouldn't this default to the protocol-appropriate port? Is there any reason to have defaults at all?

class mozprofile.permissions.**Permissions**(*profileDir*, *locations=None*)
Allows handling of permissions for mozprofile

clean_db()
Removed permissions added by mozprofile.

network_prefs(*proxy=None*)
take known locations and generate preferences to handle permissions and proxy returns a tuple of prefs, user_prefs

pac_prefs(*user_proxy=None*)
return preferences for Proxy Auto Config. originally taken from http://dxr.mozilla.org/mozilla-central/source/build/automation.py.in

write_db(*locations*)
write permissions to the sqlite database

You can set permissions by creating a `ServerLocations` object that you pass to the `Profile` constructor. Hosts can be added to it with `add_host(host, port).port` can be 0.

Preferences

user preferences

exception mozprofile.prefs.**PreferencesReadError**
read error for prefrences files

class mozprofile.prefs.**Preferences**(*prefs=None*)
assembly of preferences from various sources

add(*prefs*, *cast=False*)

Parameters

- **prefs** –
- **cast** – whether to cast strings to value, e.g. '1' -> 1

add_file(*path*)
a preferences from a file

Parameters **path** –

classmethod cast(*value*)
interpolate a preference from a string from the command line or from e.g. an .ini file, there is no good way to denote what type the preference value is, as natively it is a string

•integers will get cast to integers

•true/false will get cast to True/False

•anything enclosed in single quotes will be treated as a string with the ''s removed from both sides

classmethod read(*path*)
read preferences from a file

classmethod read_ini(*path*, *section=None*)
read preferences from an .ini file

classmethod read_json(*path*)
read preferences from a JSON blob

classmethod read_prefs(*path*, *pref_setter='user_pref'*, *interpolation=None*)
Read preferences from (e.g.) prefs.js

Parameters

- **path** – The path to the preference file to read.
- **pref_setter** – The name of the function used to set preferences in the preference file.
- **interpolation** – If provided, a dict that will be passed to str.format to interpolate preference values.

classmethod write(*_file*, *prefs*, *pref_string='user_pref(%s, %s);'*)
write preferences to a file

Preferences can be set in several ways:

- using the API: You can make a dictionary with the preferences and pass it to the `Profile` constructor. You can also add more preferences with the `Profile.set_preferences` method.
- using a JSON blob file: `mozprofile --preferences myprefs.json`
- using a `.ini` file: `mozprofile --preferences myprefs.ini`
- via the command line: `mozprofile --pref key:value --pref key:value [...]`

When setting preferences from an `.ini` file or the `--pref` switch, the value will be interpolated as an integer or a boolean (`true`/`false`) if possible.

Profile

class mozprofile.profile.**Profile**(*profile=None*, *addons=None*, *addon_manifests=None*, *preferences=None*, *locations=None*, *proxy=None*, *restore=True*)
Handles all operations regarding profile.

Creating new profiles, installing add-ons, setting preferences and handling cleanup.

The files associated with the profile will be removed automatically after the object is garbage collected:

```
profile = Profile()
print profile.profile  # this is the path to the created profile
del profile
# the profile path has been removed from disk
```

cleanup() is called under the hood to remove the profile files. You can ensure this method is called (even in the case of exception) by using the profile as a context manager:

```
with Profile() as profile:
    # do things with the profile
    pass
# profile.cleanup() has been called here
```

clean_preferences()
Removed preferences added by mozrunner.

cleanup()
Cleanup operations for the profile.

classmethod clone(*path_from*, *path_to=None*, ***kwargs*)
Instantiate a temporary profile via cloning - path: path of the basis to clone - kwargs: arguments to the profile constructor

exists()
returns whether the profile exists or not

pop_preferences(*filename*)
pop the last set of preferences added returns True if popped

reset()
reset the profile to the beginning state

set_persistent_preferences(*preferences*)
Adds preferences dict to profile preferences and save them during a profile reset

set_preferences(*preferences*, *filename='user.js'*)
Adds preferences dict to profile preferences

summary(*return_parts=False*)
returns string summarizing profile information. if return_parts is true, return the (Part_name, value) list of tuples instead of the assembled string

class mozprofile.profile.**FirefoxProfile**(*profile=None*, *addons=None*, *addon_manifests=None*, *preferences=None*, *locations=None*, *proxy=None*, *restore=True*)
Specialized Profile subclass for Firefox

class mozprofile.profile.**MetroFirefoxProfile**(*profile=None*, *addons=None*, *addon_manifests=None*, *preferences=None*, *locations=None*, *proxy=None*, *restore=True*)
Specialized Profile subclass for Firefox Metro

class mozprofile.profile.**ThunderbirdProfile**(*profile=None*, *addons=None*, *addon_manifests=None*, *preferences=None*, *locations=None*, *proxy=None*, *restore=True*)
Specialized Profile subclass for Thunderbird

Resources

Other Mozilla programs offer additional and overlapping functionality for profiles. There is also substantive documentation on profiles and their management.

- ProfileManager: XULRunner application for managing profiles. Has a GUI and CLI.
- python-profilemanager: python CLI interface similar to ProfileManager
- profile documentation

mozprocess — Launch and manage processes

Mozprocess is a process-handling module that provides some additional features beyond those available with python's subprocess:

- better handling of child processes, especially on Windows
- the ability to timeout the process after some absolute period, or some period without any data written to stdout/stderr
- the ability to specify output handlers that will be called for each line of output produced by the process
- the ability to specify handlers that will be called on process timeout and normal process termination

Running a process

mozprocess consists of two classes: ProcessHandler inherits from ProcessHandlerMixin.

Let's see how to run a process. First, the class should be instanciated with at least one argument which is a command (or a list formed by the command followed by its arguments). Then the process can be launched using the *run()* method. Finally the *wait()* method will wait until end of execution.

```
from mozprocess import processhandler

# under Windows replace by command = ['dir', '/a']
command = ['ls', '-l']
p = processhandler.ProcessHandler(command)
print("execute command: %s" % p.commandline)
p.run()
p.wait()
```

Note that using *ProcessHandler* instead of *ProcessHandlerMixin* will print the output of executed command. The attribute *commandline* provides the launched command.

Collecting process output

Let's now consider a basic shell script that will print numbers from 1 to 5 waiting 1 second between each. This script will be used as a command to launch in further examples.

proc_sleep_echo.sh:

```
#!/bin/sh

for i in 1 2 3 4 5
do
    echo $i
    sleep 1
done
```

If you are running under Windows, you won't be able to use the previous script (unless using Cygwin). So you'll use the following script:

proc_sleep_echo.bat:

```
@echo off
FOR %%A IN (1 2 3 4 5) DO (
    ECHO %%A
    REM if you have TIMEOUT then use it instead of PING
    REM TIMEOUT /T 1 /NOBREAK
    PING -n 2 127.0.0.1 > NUL
)
```

Mozprocess allows the specification of custom output handlers to gather process output while running. ProcessHandler will by default write all outputs on stdout. You can also provide (to ProcessHandler or ProcessHandlerMixin) a function or a list of functions that will be used as callbacks on each output line generated by the process.

In the following example the command's output will be stored in a file *output.log* and printed in stdout:

```
import sys
from mozprocess import processhandler

fd = open('output.log', 'w')
```

```
def tostdout(line):
    sys.stdout.write("<%s>\n" % line)

def tofile(line):
    fd.write("<%s>\n" % line)

# under Windows you'll replace by 'proc_sleep_echo.bat'
command = './proc_sleep_echo.sh'
outputs = [tostdout, tofile]

p = processhandler.ProcessHandlerMixin(command, processOutputLine=outputs)
p.run()
p.wait()

fd.close()
```

The process output can be saved (*obj = ProcessHandler(..., storeOutput=True)*) so as it is possible to request it (*obj.output*) at any time. Note that the default value for *stroreOutput* is *True*, so it is not necessary to provide it in the parameters.

```
import time
import sys
from mozprocess import processhandler

command = './proc_sleep_echo.sh' # Windows: 'proc_sleep_echo.bat'

p = processhandler.ProcessHandler(command, storeOutput=True)
p.run()
for i in xrange(10):
    print(p.output)
    time.sleep(0.5)
p.wait()
```

In previous example, you will see the *p.output* list growing.

Execution

Status

It is possible to query the status of the process via *poll()* that will return None if the process is still running, 0 if it ended without failures and a negative value if it was killed by a signal (Unix-only).

```
import time
import signal
from mozprocess import processhandler

command = './proc_sleep_echo.sh'
p = processhandler.ProcessHandler(command)
p.run()
time.sleep(2)
print("poll status: %s" % p.poll())
time.sleep(1)
p.kill(signal.SIGKILL)
print("poll status: %s" % p.poll())
```

Timeout

A timeout can be provided to the *run()* method. If the process last more than timeout seconds, it will be stopped.

After execution, the property *timedOut* will be set to True if a timeout was reached.

It is also possible to provide functions (*obj = ProcessHandler[Mixin](..., onTimeout=functions)*) that will be called if the timeout was reached.

```
from mozprocess import processhandler

def ontimeout():
    print("REACHED TIMEOUT")

command = './proc_sleep_echo.sh' # Windows: 'proc_sleep_echo.bat'
functions = [ontimeout]
p = processhandler.ProcessHandler(command, onTimeout=functions)
p.run(timeout=2)
p.wait()
print("timedOut = %s" % p.timedOut)
```

By default the process will be killed on timeout but it is possible to prevent this by setting *kill_on_timeout* to *False*.

```
p = processhandler.ProcessHandler(command, onTimeout=functions, kill_on_timeout=False)
p.run(timeout=2)
p.wait()
print("timedOut = %s" % p.timedOut)
```

In this case, no output will be available after the timeout, but the process will still be running.

Waiting

It is possible to wait until the process exits as already seen with the method *wait()*, or until the end of a timeout if given. Note that in last case the process is still alive after the timeout.

```
command = './proc_sleep_echo.sh' # Windows: 'proc_sleep_echo.bat'
p = processhandler.ProcessHandler(command)
p.run()
p.wait(timeout=2)
print("timedOut = %s" % p.timedOut)
p.wait()
```

Killing

You can request to kill the process with the method *kill*. f the parameter "ignore_children" is set to False when the process handler class is initialized, all the process's children will be killed as well.

Except on Windows, you can specify the signal with which to kill method the process (e.g.: *kill(signal.SIGKILL)*).

```
import time
from mozprocess import processhandler

command = './proc_sleep_echo.sh' # Windows: 'proc_sleep_echo.bat'
p = processhandler.ProcessHandler(command)
p.run()
time.sleep(2)
p.kill()
```

End of execution

You can provide a function or a list of functions to call at the end of the process using the initilization parameter *onFinish*.

```
from mozprocess import processhandler

def finish():
    print("Finished!!")

command = './proc_sleep_echo.sh' # Windows: 'proc_sleep_echo.bat'

p = processhandler.ProcessHandler(command, onFinish=finish)
p.run()
p.wait()
```

Child management

Consider the following scripts:

proc_child.sh:

```
#!/bin/sh
for i in a b c d e
do
    echo $i
    sleep 1
done
```

proc_parent.sh:

```
#!/bin/sh
./proc_child.sh
for i in 1 2 3 4 5
do
    echo $i
    sleep 1
done
```

For windows users consider:

proc_child.bat:

```
@echo off
FOR %%A IN (a b c d e) DO (
    ECHO %%A
    REM TIMEOUT /T 1 /NOBREAK
    PING -n 2 127.0.0.1 > NUL
)
```

proc_parent.bat:

```
@echo off
call proc_child.bat
```

```
FOR %%A IN (1 2 3 4 5) DO (
    ECHO %%A
    REM TIMEOUT /T 1 /NOBREAK
    PING -n 2 127.0.0.1 > NUL
)
```

For processes that launch other processes, mozprocess allows you to get child running status, wait for child termination, and kill children.

Ignoring children

By default the *ignore_children* option is False. In that case, killing the main process will kill all its children at the same time.

```
import time
from mozprocess import processhandler

def finish():
    print("Finished")

command = './proc_parent.sh'
p = processhandler.ProcessHandler(command, ignore_children=False, onFinish=finish)
p.run()
time.sleep(2)
print("kill")
p.kill()
```

If *ignore_children* is set to *True*, killing will apply only to the main process that will wait children end of execution before stoping (join).

```
import time
from mozprocess import processhandler

def finish():
    print("Finished")

command = './proc_parent.sh'
p = processhandler.ProcessHandler(command, ignore_children=True, onFinish=finish)
p.run()
time.sleep(2)
print("kill")
p.kill()
```

API Documentation

class mozprocess.**ProcessHandlerMixin**(*cmd*, *args=None*, *cwd=None*, *env=None*, *ignore_children=False*, *kill_on_timeout=True*, *processOutputLine=()*, *processStderrLine=()*, *onTimeout=()*, *onFinish=()*, ***kwargs*)

A class for launching and manipulating local processes.

Parameters

- **cmd** – command to run. May be a string or a list. If specified as a list, the first element will be interpreted as the command, and all additional elements will be interpreted as arguments to that command.

- **args** – list of arguments to pass to the command (defaults to None). Must not be set when *cmd* is specified as a list.
- **cwd** – working directory for command (defaults to None).
- **env** – is the environment to use for the process (defaults to os.environ).
- **ignore_children** – causes system to ignore child processes when True, defaults to False (which tracks child processes).
- **kill_on_timeout** – when True, the process will be killed when a timeout is reached. When False, the caller is responsible for killing the process. Failure to do so could cause a call to wait() to hang indefinitely. (Defaults to True.)
- **processOutputLine** – function or list of functions to be called for each line of output produced by the process (defaults to an empty list).
- **processStderrLine** – function or list of functions to be called for each line of error output - stderr - produced by the process (defaults to an empty list). If this is not specified, stderr lines will be sent to the *processOutputLine* callbacks.
- **onTimeout** – function or list of functions to be called when the process times out.
- **onFinish** – function or list of functions to be called when the process terminates normally without timing out.
- **kwargs** – additional keyword args to pass directly into Popen.

NOTE: Child processes will be tracked by default. If for any reason we are unable to track child processes and ignore_children is set to False, then we will fall back to only tracking the root process. The fallback will be logged.

commandline
the string value of the command line (command + args)

kill(*sig=None*)
Kills the managed process.

If you created the process with 'ignore_children=False' (the default) then it will also also kill all child processes spawned by it. If you specified 'ignore_children=True' when creating the process, only the root process will be killed.

Note that this does not manage any state, save any output etc, it immediately kills the process.

Parameters **sig** – Signal used to kill the process, defaults to SIGKILL (has no effect on Windows)

run(*timeout=None*, *outputTimeout=None*)
Starts the process.

If timeout is not None, the process will be allowed to continue for that number of seconds before being killed. If the process is killed due to a timeout, the onTimeout handler will be called.

If outputTimeout is not None, the process will be allowed to continue for that number of seconds without producing any output before being killed.

timedOut
True if the process has timed out.

wait(*timeout=None*)
Waits until all output has been read and the process is terminated.

If timeout is not None, will return after timeout seconds. This timeout only causes the wait function to return and does not kill the process.

Returns the process exit code value: - None if the process hasn't terminated yet - A negative number if the process was killed by signal N (Unix only) - '0' if the process ended without failures

class mozprocess.**ProcessHandler**(*cmd*, *logfile=None*, *stream=True*, *storeOutput=True*, ***kwargs*)

Convenience class for handling processes with default output handlers.

By default, all output is sent to stdout. This can be disabled by setting the *stream* argument to None.

If processOutputLine keyword argument is specified the function or the list of functions specified by this argument will be called for each line of output; the output will not be written to stdout automatically then if stream is True (the default).

If storeOutput==True, the output produced by the process will be saved as self.output.

If logfile is not None, the output produced by the process will be appended to the given file.

mozrunner — Manage remote and local gecko processes

Mozrunner provides an API to manage a gecko-based application with an arbitrary configuration profile. It currently supports local desktop binaries such as Firefox and Thunderbird, as well as Firefox OS on mobile devices and emulators.

Basic usage

The simplest way to use mozrunner, is to instantiate a runner, start it and then wait for it to finish:

```
from mozrunner import FirefoxRunner
binary = 'path/to/firefox/binary'
runner = FirefoxRunner(binary=binary)
runner.start()
runner.wait()
```

This automatically creates and uses a default mozprofile object. If you wish to use a specialized or pre-existing profile, you can create a *mozprofile* object and pass it in:

```
from mozprofile import FirefoxProfile
from mozrunner import FirefoxRunner
import os

binary = 'path/to/firefox/binary'
profile_path = 'path/to/profile'
if os.path.exists(profile_path):
    profile = FirefoxProfile.clone(path_from=profile_path)
else:
    profile = FirefoxProfile(profile=profile_path)
runner = FirefoxRunner(binary=binary, profile=profile)
runner.start()
runner.wait()
```

Handling output

By default, mozrunner dumps the output of the gecko process to standard output. It is possible to add arbitrary output handlers by passing them in via the *process_args* argument. Be careful, passing in a handler overrides the default behaviour. So if you want to use a handler in addition to dumping to stdout, you need to specify that explicitly. For example:

```
from mozrunner import FirefoxRunner

def handle_output_line(line):
    do_something(line)

binary = 'path/to/firefox/binary'
process_args = { 'stream': sys.stdout,
                 'processOutputLine': [handle_output_line] }
runner = FirefoxRunner(binary=binary, process_args=process_args)
```

Mozrunner uses *mozprocess* to manage the underlying gecko process and handle output. See the *mozprocess documentation* for all available arguments accepted by *process_args*.

Handling timeouts

Sometimes gecko can hang, or maybe it is just taking too long. To handle this case you may want to set a timeout. Mozrunner has two kinds of timeouts, the traditional *timeout*, and the *outputTimeout*. These get passed into the *runner.start()* method. Setting *timeout* will cause gecko to be killed after the specified number of seconds, no matter what. Setting *outputTimeout* will cause gecko to be killed after the specified number of seconds with no output. In both cases the process handler's *onTimeout* callbacks will be triggered.

```
from mozrunner import FirefoxRunner

def on_timeout():
    print('timed out after 10 seconds with no output!')

binary = 'path/to/firefox/binary'
process_args = { 'onTimeout': on_timeout }
runner = FirefoxRunner(binary=binary, process_args=process_args)
runner.start(outputTimeout=10)
runner.wait()
```

The *runner.wait()* method also accepts a timeout argument. But unlike the arguments to *runner.start()*, this one simply returns from the wait call and does not kill the gecko process.

```
runner.start(timeout=100)

waiting = 0
while runner.wait(timeout=1) is None:
    waiting += 1
    print("Been waiting for %d seconds so far.." % waiting)
assert waiting <= 100
```

Using a device runner

The previous examples used a GeckoRuntimeRunner. If you want to control a gecko process on a remote device, you need to use a DeviceRunner. The api is nearly identical except you don't pass in a binary, instead you create a device object. For example, for B2G (Firefox OS) emulators you might do:

```
from mozrunner import B2GEmulatorRunner

b2g_home = 'path/to/B2G'
runner = B2GEmulatorRunner(arch='arm', b2g_home=b2g_home)
```

```
runner.start()
runner.wait()
```

Device runners have a *device* object. Remember that the gecko process runs on the device. In the case of the emulator, it is possible to start the device independently of the gecko process.

```
runner.device.start() # launches the emulator (which also launches gecko)
runner.start()        # stops the gecko process, installs the profile, restarts the␣
↪gecko process
```

Runner API Documentation

Application Runners

This module contains a set of shortcut methods that create runners for commonly used Mozilla applications, such as Firefox or B2G emulator.

mozrunner.runners.**B2GDesktopRunner**(**args*, ***kwargs*)

Create a B2G desktop runner.

Parameters

- **binary** – Path to b2g desktop binary.
- **cmdargs** – Arguments to pass into binary.
- **profile** – Profile object to use.
- **env** – Environment variables to pass into the gecko process.
- **clean_profile** – If True, restores profile back to original state.
- **process_class** – Class used to launch the binary.
- **process_args** – Arguments to pass into process_class.
- **symbols_path** – Path to symbol files used for crash analysis.
- **show_crash_reporter** – allow the crash reporter window to pop up. Defaults to False.

Returns A GeckoRuntimeRunner for b2g desktop.

mozrunner.runners.**B2GDeviceRunner**(*b2g_home=None*, *adb_path=None*, *logdir=None*, *serial=None*, ***kwargs*)

Create a B2G device runner.

Parameters

- **b2g_home** – Path to root B2G repository.
- **logdir** – Path to save logfiles such as logcat.
- **serial** – Serial of device to connect to as seen in *adb devices*.
- **profile** – Profile object to use.
- **env** – Environment variables to pass into the b2g.sh process.
- **clean_profile** – If True, restores profile back to original state.
- **process_class** – Class used to launch the b2g.sh process.
- **process_args** – Arguments to pass into the b2g.sh process.

- **symbols_path** – Path to symbol files used for crash analysis.

Returns A DeviceRunner for B2G devices.

mozrunner.runners.**B2GEmulatorRunner**(*arch='arm'*, *b2g_home=None*, *adb_path=None*, *logdir=None*, *binary=None*, *no_window=None*, *resolution=None*, *sdcard=None*, *userdata=None*, ***kwargs*)

Create a B2G emulator runner.

Parameters

- **arch** – The architecture of the emulator, either 'arm' or 'x86'. Defaults to 'arm'.
- **b2g_home** – Path to root B2G repository.
- **logdir** – Path to save logfiles such as logcat and qemu output.
- **no_window** – Run emulator without a window.
- **resolution** – Screen resolution to set emulator to, e.g '800x1000'.
- **sdcard** – Path to local emulated sdcard storage.
- **userdata** – Path to custom userdata image.
- **profile** – Profile object to use.
- **env** – Environment variables to pass into the b2g.sh process.
- **clean_profile** – If True, restores profile back to original state.
- **process_class** – Class used to launch the b2g.sh process.
- **process_args** – Arguments to pass into the b2g.sh process.
- **symbols_path** – Path to symbol files used for crash analysis.

Returns A DeviceRunner for B2G emulators.

mozrunner.runners.**FennecEmulatorRunner**(*avd='mozemulator-4.3'*, *adb_path=None*, *avd_home=None*, *logdir=None*, *serial=None*, *binary=None*, *app='org.mozilla.fennec'*, ***kwargs*)

Create a Fennec emulator runner. This can either start a new emulator (which will use an avd), or connect to an already-running emulator.

Parameters

- **avd** – name of an AVD available in your environment. Typically obtained via tooltool: either 'mozemulator-4.3' or 'mozemulator-x86'. Defaults to 'mozemulator-4.3'
- **avd_home** – Path to avd parent directory
- **logdir** – Path to save logfiles such as logcat and qemu output.
- **serial** – Serial of emulator to connect to as seen in *adb devices*. Defaults to the first entry in *adb devices*.
- **binary** – Path to emulator binary. Defaults to None, which causes the device_class to guess based on PATH.
- **app** – Name of Fennec app (often org.mozilla.fennec_$USER) Defaults to 'org.mozilla.fennec'
- **cmdargs** – Arguments to pass into binary.

Returns A DeviceRunner for Android emulators.

mozrunner.runners.**FirefoxRunner**(**args*, ***kwargs*)
Create a desktop Firefox runner.

Parameters

- **binary** – Path to Firefox binary.
- **cmdargs** – Arguments to pass into binary.
- **profile** – Profile object to use.
- **env** – Environment variables to pass into the gecko process.
- **clean_profile** – If True, restores profile back to original state.
- **process_class** – Class used to launch the binary.
- **process_args** – Arguments to pass into process_class.
- **symbols_path** – Path to symbol files used for crash analysis.
- **show_crash_reporter** – allow the crash reporter window to pop up. Defaults to False.

Returns A GeckoRuntimeRunner for Firefox.

mozrunner.runners.**Runner**(**args*, ***kwargs*)
Create a generic GeckoRuntime runner.

Parameters

- **binary** – Path to binary.
- **cmdargs** – Arguments to pass into binary.
- **profile** – Profile object to use.
- **env** – Environment variables to pass into the gecko process.
- **clean_profile** – If True, restores profile back to original state.
- **process_class** – Class used to launch the binary.
- **process_args** – Arguments to pass into process_class.
- **symbols_path** – Path to symbol files used for crash analysis.
- **show_crash_reporter** – allow the crash reporter window to pop up. Defaults to False.

Returns A generic GeckoRuntimeRunner.

mozrunner.runners.**ThunderbirdRunner**(**args*, ***kwargs*)
Create a desktop Thunderbird runner.

Parameters

- **binary** – Path to Thunderbird binary.
- **cmdargs** – Arguments to pass into binary.
- **profile** – Profile object to use.
- **env** – Environment variables to pass into the gecko process.
- **clean_profile** – If True, restores profile back to original state.
- **process_class** – Class used to launch the binary.
- **process_args** – Arguments to pass into process_class.
- **symbols_path** – Path to symbol files used for crash analysis.

- **show_crash_reporter** – allow the crash reporter window to pop up. Defaults to False.

Returns A GeckoRuntimeRunner for Thunderbird.

BaseRunner

class mozrunner.base.**BaseRunner**(*app_ctx=None*, *profile=None*, *clean_profile=True*, *env=None*, *process_class=None*, *process_args=None*, *symbols_path=None*, *dump_save_path=None*, *addons=None*)

The base runner class for all mozrunner objects, both local and remote.

check_for_crashes(*dump_directory=None*, *dump_save_path=None*, *test_name=None*, *quiet=False*)

Check for possible crashes and output the stack traces.

Parameters

- **dump_directory** – Directory to search for minidump files
- **dump_save_path** – Directory to save the minidump files to
- **test_name** – Name to use in the crash output
- **quiet** – If *True* don't print the PROCESS-CRASH message to stdout

Returns Number of crashes which have been detected since the last invocation

cleanup()

Cleanup all runner state

command

Returns the command list to run.

is_running()

Checks if the process is running.

Returns True if the process is active

reset()

Reset the runner to its default state.

returncode

The returncode of the process_handler. A value of None indicates the process is still running. A negative value indicates the process was killed with the specified signal.

Raises RunnerNotStartedError

start(*debug_args=None*, *interactive=False*, *timeout=None*, *outputTimeout=None*)

Run self.command in the proper environment.

Parameters

- **debug_args** – arguments for a debugger
- **interactive** – uses subprocess.Popen directly
- **timeout** – see process_handler.run()
- **outputTimeout** – see process_handler.run()

Returns the process id

stop(*sig=None*)

Kill the process.

Parameters **sig** – Signal used to kill the process, defaults to SIGKILL (has no effect on Windows).

Returns the process return code if process was already stopped, -<signal> if process was killed (Unix only)

Raises RunnerNotStartedError

wait(*timeout=None*)
Wait for the process to exit.

Parameters **timeout** – if not None, will return after timeout seconds. Timeout is ignored if interactive was set to True.

Returns the process return code if process exited normally, -<signal> if process was killed (Unix only), None if timeout was reached and the process is still running.

Raises RunnerNotStartedError

GeckoRuntimeRunner

class mozrunner.base.**GeckoRuntimeRunner**(*binary*, *cmdargs=None*, ***runner_args*)
Bases: mozrunner.base.runner.BaseRunner

The base runner class used for local gecko runtime binaries, such as Firefox and Thunderbird.

DeviceRunner

class mozrunner.base.**DeviceRunner**(*device_class*, *device_args=None*, ***kwargs*)
Bases: mozrunner.base.runner.BaseRunner

The base runner class used for running gecko on remote devices (or emulators), such as B2G.

Device API Documentation

Generally using the device classes directly shouldn't be required, but in some cases it may be desirable.

Device

class mozrunner.devices.**Device**(*app_ctx*, *logdir=None*, *serial=None*, *restore=True*)

cleanup()
Cleanup the device.

connect()
Connects to a running device. If no serial was specified in the constructor, defaults to the first entry in *adb devices*.

install_busybox(*busybox*)
Installs busybox on the device.

Parameters **busybox** – Path to busybox binary to install.

pull_minidumps()
Saves any minidumps found in the remote profile on the local filesystem.

Returns Path to directory containing the dumps.

reboot()
Reboots the device via adb.

remote_profiles
A list of remote profiles on the device.

setup_profile(*profile*)
Copy profile to the device and update the remote profiles.ini to point to the new profile.

Parameters **profile** – mozprofile object to copy over.

Emulator

class mozrunner.devices.**Emulator**(*app_ctx*, *arch*, *resolution=None*, *sdcard=None*, *userdata=None*, *no_window=None*, *binary=None*, ***kwargs*)
Bases: mozrunner.devices.emulator.BaseEmulator

args
Arguments to pass into the emulator binary.

cleanup()
Cleans up and kills the emulator, if it was started by mozrunner.

connect()
Connects to a running device. If no serial was specified in the constructor, defaults to the first entry in *adb devices*.

mozcrash — Print stack traces from minidumps left behind by crashed processes

Gets stack traces out of processes that have crashed and left behind a minidump file using the Google Breakpad library.
mozcrash is a library for getting a stack trace out of processes that have crashed and left behind a minidump file using the Google Breakpad library.

mozcrash.**check_for_crashes**(*dump_directory*, *symbols_path=None*, *stackwalk_binary=None*, *dump_save_path=None*, *test_name=None*, *quiet=False*)
Print a stack trace for minidump files left behind by a crashing program.

dump_directory will be searched for minidump files. Any minidump files found will have *stackwalk_binary* executed on them, with *symbols_path* passed as an extra argument.

stackwalk_binary should be a path to the minidump_stackwalk binary. If *stackwalk_binary* is not set, the MINIDUMP_STACKWALK environment variable will be checked and its value used if it is not empty.

symbols_path should be a path to a directory containing symbols to use for dump processing. This can either be a path to a directory containing Breakpad-format symbols, or a URL to a zip file containing a set of symbols.

If *dump_save_path* is set, it should be a path to a directory in which to copy minidump files for safekeeping after a stack trace has been printed. If not set, the environment variable MINIDUMP_SAVE_PATH will be checked and its value used if it is not empty.

If *test_name* is set it will be used as the test name in log output. If not set the filename of the calling function will be used.

If *quiet* is set, no PROCESS-CRASH message will be printed to stdout if a crash is detected.

Returns number of minidump files found.

mozdebug — Configure and launch compatible debuggers.

This module contains a set of function to gather information about the debugging capabilities of the platform. It allows to look for a specific debugger or to query the system for a compatible/default debugger.

The following simple example looks for the default debugger on the current platform and launches a debugger process with the correct debugger-specific arguments:

```
import mozdebug

debugger = mozdebug.get_default_debugger_name()
debuggerInfo = mozdebug.get_debugger_info(debugger)

debuggeePath = "toDebug"

processArgs = [self.debuggerInfo.path] + self.debuggerInfo.args
processArgs.append(debuggeePath)

run_process(args, ...)
```

mozdebug.**get_debugger_info**(*debugger*, *debuggerArgs=None*, *debuggerInteractive=False*)

Get the information about the requested debugger.

Returns a dictionary containing the |path| of the debugger executable, if it will run in |interactive| mode, its arguments and whether it needs to escape arguments it passes to the debugged program (|requiresEscapedArgs|). If the debugger cannot be found in the system, returns |None|.

Parameters

- **debugger** – The name of the debugger.
- **debuggerArgs** – If specified, it's the arguments to pass to the debugger,

as a string. Any debugger-specific separator arguments are appended after these arguments. :param debuggerInteractive: If specified, forces the debugger to be interactive.

mozdebug.**get_default_debugger_name**(*search=1*)

Get the debugger name for the default debugger on current platform.

Parameters **search** – If specified, stops looking for the debugger if the default one is not found (|DebuggerSearch.OnlyFirst|) or keeps looking for other compatible debuggers (|DebuggerSearch.KeepLooking|).

CHAPTER 4

Serving up content to be consumed by the browser

Warning: The mozhttpd module is considered obsolete. For new code, please use wptserve which can do everything mozhttpd does and more.

mozhttpd — Simple webserver

Mozhttpd is a simple http webserver written in python, designed expressly for use in automated testing scenarios. It is designed to both serve static content and provide simple web services.

The server is based on python standard library modules such as SimpleHttpServer, urlparse, etc. The ThreadingMixIn is used to serve each request on a discrete thread.

Some existing uses of mozhttpd include Peptest, Eideticker, and Talos.

The following simple example creates a basic HTTP server which serves content from the current directory, defines a single API endpoint */api/resource/<resourceid>* and then serves requests indefinitely:

```
import mozhttpd

@mozhttpd.handlers.json_response
def resource_get(request, objid):
    return (200, { 'id': objid,
                   'query': request.query })


httpd = mozhttpd.MozHttpd(port=8080, docroot='.',
                          urlhandlers = [ { 'method': 'GET',
                                            'path': '/api/resources/([^/]+)/?',
                                            'function': resource_get } ])
print "Serving '%s' at %s:%s" % (httpd.docroot, httpd.host, httpd.port)
httpd.start(block=True)
```

class mozhttpd.**MozHttpd**(*host='127.0.0.1'*, *port=0*, *docroot=None*, *urlhandlers=None*, *path_mappings=None*, *proxy_host_dirs=False*, *log_requests=False*)

Parameters

- **host** – Host from which to serve (default 127.0.0.1)
- **port** – Port from which to serve (default 8888)
- **docroot** – Server root (default os.getcwd())
- **urlhandlers** – Handlers to specify behavior against method and path match (default None)
- **path_mappings** – A dict mapping URL prefixes to additional on-disk paths.
- **proxy_host_dirs** – Toggle proxy behavior (default False)
- **log_requests** – Toggle logging behavior (default False)

Very basic HTTP server class. Takes a docroot (path on the filesystem) and a set of urlhandler dictionaries of the form:

```
{
  'method': HTTP method (string): GET, POST, or DEL,
  'path': PATH_INFO (regular expression string),
  'function': function of form fn(arg1, arg2, arg3, ..., request)
}
```

and serves HTTP. For each request, MozHttpd will either return a file off the docroot, or dispatch to a handler function (if both path and method match).

Note that one of docroot or urlhandlers may be None (in which case no local files or handlers, respectively, will be used). If both docroot or urlhandlers are None then MozHttpd will default to serving just the local directory.

MozHttpd also handles proxy requests (i.e. with a full URI on the request line). By default files are served from docroot according to the request URI's path component, but if proxy_host_dirs is True, files are served from <self.docroot>/<host>/.

For example, the request "GET http://foo.bar/dir/file.html" would (assuming no handlers match) serve <docroot>/dir/file.html if proxy_host_dirs is False, or <docroot>/foo.bar/dir/file.html if it is True.

get_url(*path='/'*)
Returns a URL that can be used for accessing the server (e.g. http://192.168.1.3:4321/)

Parameters **path** – Path to append to URL (e.g. if path were /foobar.html you would get a URL like http://192.168.1.3:4321/foobar.html). Default is /.

start(*block=False*)
Starts the server.

If *block* is True, the call will not return. If *block* is False, the server will be started on a separate thread that can be terminated by a call to stop().

stop()
Stops the server.

If the server is not running, this method has no effect.

class mozhttpd.**Request**(*uri*, *headers*, *rfile=None*)
Details of a request.

mozhttpd.**json_response**(*func*)
Translates results of 'func' into a JSON response.

Interface

class mozhttpd.**MozHttpd**(*host='127.0.0.1'*, *port=0*, *docroot=None*, *urlhandlers=None*, *path_mappings=None*, *proxy_host_dirs=False*, *log_requests=False*)

Parameters

- **host** – Host from which to serve (default 127.0.0.1)
- **port** – Port from which to serve (default 8888)
- **docroot** – Server root (default os.getcwd())
- **urlhandlers** – Handlers to specify behavior against method and path match (default None)
- **path_mappings** – A dict mapping URL prefixes to additional on-disk paths.
- **proxy_host_dirs** – Toggle proxy behavior (default False)
- **log_requests** – Toggle logging behavior (default False)

Very basic HTTP server class. Takes a docroot (path on the filesystem) and a set of urlhandler dictionaries of the form:

```
{
  'method': HTTP method (string): GET, POST, or DEL,
  'path': PATH_INFO (regular expression string),
  'function': function of form fn(arg1, arg2, arg3, ..., request)
}
```

and serves HTTP. For each request, MozHttpd will either return a file off the docroot, or dispatch to a handler function (if both path and method match).

Note that one of docroot or urlhandlers may be None (in which case no local files or handlers, respectively, will be used). If both docroot or urlhandlers are None then MozHttpd will default to serving just the local directory.

MozHttpd also handles proxy requests (i.e. with a full URI on the request line). By default files are served from docroot according to the request URI's path component, but if proxy_host_dirs is True, files are served from <self.docroot>/<host>/.

For example, the request "GET http://foo.bar/dir/file.html" would (assuming no handlers match) serve <docroot>/dir/file.html if proxy_host_dirs is False, or <docroot>/foo.bar/dir/file.html if it is True.

get_url(*path='/'*)

Returns a URL that can be used for accessing the server (e.g. http://192.168.1.3:4321/)

Parameters **path** – Path to append to URL (e.g. if path were /foobar.html you would get a URL like http://192.168.1.3:4321/foobar.html). Default is /.

start(*block=False*)

Starts the server.

If *block* is True, the call will not return. If *block* is False, the server will be started on a separate thread that can be terminated by a call to stop().

stop()

Stops the server.

If the server is not running, this method has no effect.

CHAPTER 5

Logging and reporting

Ideally output between different types of testing system should be as uniform as possible, as well as making it easy to make things more or less verbose. We created some libraries to make doing this easy.

`mozlog` — Structured logging for test output

`mozlog` is a library designed for logging the execution and results of test harnesses. The internal data model is a stream of JSON-compatible objects, with one object per log entry. The default output format is line-based, with one JSON object serialized per line.

`mozlog` is *not* based on the stdlib logging module, although it shares several concepts with it.

One notable difference between this module and the standard logging module is the way that loggers are created. The structured logging module does not require that loggers with a specific name are singleton objects accessed through a factory function. Instead the `StructuredLogger` constructor may be used directly. However all loggers with the same name share the same internal state (the "Borg" pattern). In particular the list of handler functions is the same for all loggers with the same name.

Typically, you would only instantiate one logger object per program. Two convenience methods are provided to set and get the default logger in the program.

Logging is threadsafe, with access to handlers protected by a `threading.Lock`. However it is *not* process-safe. This means that applications using multiple processes, e.g. via the `multiprocessing` module, should arrange for all logging to happen in a single process.

Data Format

Structured loggers produce messages in a simple format designed to be compatible with the JSON data model. Each message is a single object, with the type of message indicated by the `action` key. It is intended that the set of `action` values be closed; where there are use cases for additional values they should be integrated into this module rather than extended in an ad-hoc way. The set of keys present on on all messages is:

`action` The type of the message (string).

time The timestamp of the message in ms since the epoch (int).

thread The name of the thread emitting the message (string).

pid The pid of the process creating the message (int).

source Name of the logger creating the message (string).

For each `action` there are is a further set of specific fields describing the details of the event that caused the message to be emitted:

suite_start Emitted when the testsuite starts running.

- **tests** A dict of test ids keyed by group. Groups are any logical grouping of tests, for example a manifest, directory or tag. For convenience, a list of test ids can be used instead. In this case all tests will automatically be placed in the 'default' group name. Test ids can either be strings or lists of strings (an example of the latter is reftests where the id has the form [test_url, ref_type, ref_url]). Test ids are assumed to be unique within a given testsuite. In cases where the test list is not known upfront an empty dict or list may be passed (dict).
- **run_info** An optional dictionary describing the properties of the build and test environment. This contains the information provided by *mozinfo*, plus a boolean `debug` field indicating whether the build under test is a debug build.

suite_end Emitted when the testsuite is finished and no more results will be produced.

test_start Emitted when a test is being started.

- **test** A unique id for the test (string or list of strings).
- **path** Optional path to the test relative to some base (typically the root of the source tree). Mainly used when `test` id is not a path (string).

test_status Emitted for a test which has subtests to record the result of a single subtest.

- **test** The same unique id for the test as in the `test_start` message.
- **subtest** Name of the subtest (string).
- **status** Result of the test (string enum; `PASS`, `FAIL`, `TIMEOUT`, `NOTRUN`)
- **expected** Expected result of the test. Omitted if the expected result is the same as the actual result (string enum, same as `status`).

test_end Emitted to give the result of a test with no subtests, or the status of the overall file when there are subtests.

- **test** The same unique id for the test as in the `test_start` message.
- **status** Either result of the test (if there are no subtests) in which case (string enum `PASS`, `FAIL`, `TIMEOUT`, `CRASH`, `ASSERT`, `SKIP`) or the status of the overall file where there are subtests (string enum `OK`, `ERROR`, `TIMEOUT`, `CRASH`, `ASSERT`, `SKIP`).
- **expected** The expected status, or omitted if the expected status matches the actual status (string enum, same as `status`).

process_output Output from a managed subprocess.

- `process` pid of the subprocess.
- `command` Command used to launch the subprocess.
- `data` Data output by the subprocess.

log General human-readable logging message, used to debug the harnesses themselves rather than to provide input to other tools.

- **level** Level of the log message (string enum `CRITICAL`, `ERROR`, `WARNING`, `INFO`, `DEBUG`).

message Text of the log message.

Testsuite Protocol

When used for testsuites, the following structured logging messages must be emitted:

- One `suite_start` message before any `test_*` messages
- One `test_start` message per test that is run
- One `test_status` message per subtest that is run. This might be zero if the test type doesn't have the notion of subtests.
- One `test_end` message per test that is run, after the `test_start` and any `test_status` messages for that same test.
- One `suite_end` message after all `test_*` messages have been emitted.

The above mandatory events may be interspersed with `process_output` and `log` events, as required.

Subtests

The purpose of subtests is to deal with situations where a single test produces more than one result, and the exact details of the number of results is not known ahead of time. For example consider a test harness that loads JavaScript-based tests in a browser. Each url loaded would be a single test, with corresponding `test_start` and `test_end` messages. If there can be more than one JS-defined test on a page, however, it it useful to track the results of those tests seperately. Therefore each of those tests is a subtest, and one `test_status` message must be generated for each subtest result.

Subtests must have a name that is unique within their parent test.

Whether or not a test has subtests changes the meaning of the `status` property on the test itself. When the test does not have any subtests, this property is the actual test result such as `PASS` or `FAIL` . When a test does have subtests, the test itself does not have a result as-such; it isn't meaningful to describe it as having a `PASS` result, especially if the subtests did not all pass. Instead this property is used to hold information about whether the test ran without error. If no errors were detected the test must be given the status `OK`. Otherwise the test may get the status `ERROR` (for e.g. uncaught JS exceptions), `TIMEOUT` (if no results were reported in the allowed time) or `CRASH` (if the test caused the process under test to crash).

StructuredLogger Objects

mozlog.structuredlog.**set_default_logger**(*default_logger*)

Sets the default logger to logger.

It can then be retrieved with *get_default_logger()*

Note that *setup_logging()* will set a default logger for you, so there should be no need to call this function if you're using setting up logging that way (recommended).

Parameters **default_logger** – The logger to set to default.

mozlog.structuredlog.**get_default_logger**(*component=None*)

Gets the default logger if available, optionally tagged with component name. Will return None if not yet set

Parameters **component** – The component name to tag log messages with

class mozlog.structuredlog.**StructuredLogger**(*name*, *component=None*)

add_handler(*handler*)
Add a handler to the current logger

critical(**args*, ***kwargs*)
Log a message with level CRITICAL

Parameters

- **message** – The string message to log
- **exc_info** – Either a boolean indicating whether to include a traceback derived from sys.exc_info() or a three-item tuple in the same format as sys.exc_info() containing exception information to log.

debug(**args*, ***kwargs*)
Log a message with level DEBUG

Parameters

- **message** – The string message to log
- **exc_info** – Either a boolean indicating whether to include a traceback derived from sys.exc_info() or a three-item tuple in the same format as sys.exc_info() containing exception information to log.

error(**args*, ***kwargs*)
Log a message with level ERROR

Parameters

- **message** – The string message to log
- **exc_info** – Either a boolean indicating whether to include a traceback derived from sys.exc_info() or a three-item tuple in the same format as sys.exc_info() containing exception information to log.

handlers
A list of handlers that will be called when a message is logged from this logger

info(**args*, ***kwargs*)
Log a message with level INFO

Parameters

- **message** – The string message to log
- **exc_info** – Either a boolean indicating whether to include a traceback derived from sys.exc_info() or a three-item tuple in the same format as sys.exc_info() containing exception information to log.

process_output(**args*, ***kwargs*)
Log output from a managed process.

Parameters

- **process** – A unique identifier for the process producing the output (typically the pid)
- **data** – The output to log
- **command** – A string representing the full command line used to start the process.

remove_handler(*handler*)
Remove a handler from the current logger

suite_end(**args*, ***kwargs*)
Log a suite_end message

suite_start(**args*, ***kwargs*)
Log a suite_start message

Parameters

- **tests** (*dict*) – Test identifiers that will be run in the suite, keyed by group name.
- **run_info** (*dict*) – Optional information typically provided by mozinfo.
- **version_info** (*dict*) – Optional target application version information provided by mozversion.
- **device_info** (*dict*) – Optional target device information provided by mozdevice.

test_end(**args*, ***kwargs*)
Log a test_end message indicating that a test completed. For tests with subtests this indicates whether the overall test completed without errors. For tests without subtests this indicates the test result directly.

Parameters

- **test** – Identifier of the test that produced the result.
- **status** – Status string indicating the test result
- **expected** – Status string indicating the expected test result.
- **message** – String containing a message associated with the result.
- **stack** – a stack trace encountered during test execution.
- **extra** – suite-specific data associated with the test result.

test_start(**args*, ***kwargs*)
Log a test_start message

Parameters

- **test** – Identifier of the test that will run.
- **path** – Path to test relative to some base (typically the root of the source tree).

test_status(**args*, ***kwargs*)
Log a test_status message indicating a subtest result. Tests that do not have subtests are not expected to produce test_status messages.

Parameters

- **test** – Identifier of the test that produced the result.
- **subtest** – Name of the subtest.
- **status** – Status string indicating the subtest result
- **expected** – Status string indicating the expected subtest result.
- **message** – String containing a message associated with the result.
- **stack** – a stack trace encountered during test execution.
- **extra** – suite-specific data associated with the test result.

warning(**args*, ***kwargs*)
Log a message with level WARNING

Parameters

- **message** – The string message to log

- **exc_info** – Either a boolean indicating whether to include a traceback derived from sys.exc_info() or a three-item tuple in the same format as sys.exc_info() containing exception information to log.

class mozlog.structuredlog.**StructuredLogFileLike**(*logger*, *level=u'info'*, *prefix=None*)

Wrapper for file-like objects to redirect writes to logger instead. Each call to *write* becomes a single log entry of type *log*.

When using this it is important that the callees i.e. the logging handlers do not themselves try to write to the wrapped file as this will cause infinite recursion.

Parameters

- **logger** – *StructuredLogger* to which to redirect the file write operations.
- **level** – log level to use for each write.
- **prefix** – String prefix to prepend to each log entry.

ProxyLogger Objects

Since *mozlog.structuredlog.get_default_logger()* return None when the default logger is not initialized, it is not possible to directly use it at the module level.

With ProxyLogger, it is possible to write the following code:

```
from mozlog import get_proxy_logger

LOG = get_proxy_logger('component_name')


def my_function():
    LOG.info('logging with a module level object')
```

Note: mozlog still needs to be initialized before the first call occurs to a ProxyLogger instance, for example with *mozlog.commandline.setup_logging()*.

---

mozlog.proxy.**get_proxy_logger**(*component=None*)

Returns a *ProxyLogger* for the given component.

class mozlog.proxy.**ProxyLogger**(*component=None*)

A ProxyLogger behaves like a *mozlog.structuredlog.StructuredLogger*.

Each method and attribute access will be forwarded to the underlying StructuredLogger.

RuntimeError will be raised when the default logger is not yet initialized.

Handlers

A handler is a callable that is called for each log message produced and is responsible for handling the processing of that message. The typical example of this is a `StreamHandler` which takes a log message, invokes a formatter which converts the log to a string, and writes it to a file.

class mozlog.handlers.**BaseHandler**(*inner*)

A base handler providing message handling facilities to derived classes.

Parameters **inner** – A handler-like callable that may receive messages from a log user.

handle_message(*topic*, *cmd*, **args*)
Handles a message for the given topic by calling a subclass-defined callback for the command.

Parameters

- **topic** – The topic of the broadcasted message. Handlers opt-in to receiving messages by identifying a topic when calling register_message_handlers.
- **command** – The command to issue. This is a string that corresponds to a callback provided by the target.
- **arg** – Arguments to pass to the identified message callback, if any.

class mozlog.handlers.**StreamHandler**(*stream*, *formatter*)
Handler for writing to a file-like object

Parameters

- **stream** – File-like object to write log messages to
- **formatter** – formatter to convert messages to string format

class mozlog.handlers.**LogLevelFilter**(*inner*, *level*)
Handler that filters out messages with action of log and a level lower than some specified level.

Parameters

- **inner** – Handler to use for messages that pass this filter
- **level** – Minimum log level to process

class mozlog.handlers.**BufferHandler**(*inner*, *message_limit=100*, *buffered_actions=None*)
Handler that maintains a circular buffer of messages based on the size and actions specified by a user.

Parameters

- **inner** – The underlying handler used to emit messages.
- **message_limit** – The maximum number of messages to retain for context. If None, the buffer will grow without limit.
- **buffered_actions** – The set of actions to include in the buffer rather than log directly.

Formatters

Formatters are callables that take a log message, and return either a string representation of that message, or `None` if that message should not appear in the output. This allows formatters to both exclude certain items and create internal buffers of the output so that, for example, a single string might be returned for a `test_end` message indicating the overall result of the test, including data provided in the `test_status` messages.

Formatter modules are written so that they can take raw input on stdin and write formatted output on stdout. This allows the formatters to be invoked as part of a command line for post-processing raw log files.

class mozlog.formatters.base.**BaseFormatter**
Base class for implementing non-trivial formatters.

Subclasses are expected to provide a method for each action type they wish to handle, each taking a single argument for the test data. For example a trivial subclass that just produces the id of each test as it starts might be:

```
class StartIdFormatter(BaseFormatter);
    def test_start(data):
```

```
        #For simplicity in the example pretend the id is always a string
        return data["test"]
```

class mozlog.formatters.unittest.**UnittestFormatter**
Formatter designed to produce output in a format like that used by the unittest module in the standard library.

class mozlog.formatters.xunit.**XUnitFormatter**
Formatter that produces XUnit-style XML output.

The tree is created in-memory so this formatter may be problematic with very large log files.

Note that the data model isn't a perfect match. In particular XUnit assumes that each test has a unittest-style class name and function name, which isn't the case for us. The implementation currently replaces path names with something that looks like class names, but this doesn't work for test types that actually produce class names, or for test types that have multiple components in their test id (e.g. reftests).

class mozlog.formatters.html.**HTMLFormatter**
Formatter that produces a simple HTML-formatted report.

class mozlog.formatters.machformatter.**MachFormatter**(*start_time=None*, *write_interval=False*, *write_times=True*, *terminal=None*, *disable_colors=False*)

class mozlog.formatters.tbplformatter.**TbplFormatter**(*compact=False*)
Formatter that formats logs in the legacy formatting format used by TBPL This is intended to be used to preserve backward compatibility with existing tools hand-parsing this format.

Processing Log Files

The mozlog.reader module provides utilities for working with structured log files.

class mozlog.reader.**LogHandler**
Base class for objects that act as log handlers. A handler is a callable that takes a log entry as the only argument.

Subclasses are expected to provide a method for each action type they wish to handle, each taking a single argument for the test data. For example a trivial subclass that just produces the id of each test as it starts might be:

```
class StartIdHandler(LogHandler):
    def test_start(data):
        #For simplicity in the example pretend the id is always a string
        return data["test"]
```

mozlog.reader.**each_log**(*log_iter*, *action_map*)
Call a callback for each item in an iterable containing structured log entries

Parameters

- **log_iter** – Iterator returning structured log entries
- **action_map** – Dictionary mapping action name to callback function. Log items with actions not in this dictionary will be skipped.

mozlog.reader.**handle_log**(*log_iter*, *handler*)
Call a handler for each item in a log, discarding the return value

mozlog.reader.**imap_log**(*log_iter*, *action_map*)
Create an iterator that will invoke a callback per action for each item in a iterable containing structured log entries

Parameters

- **log_iter** – Iterator returning structured log entries
- **action_map** – Dictionary mapping action name to callback function. Log items with actions not in this dictionary will be skipped.

mozlog.reader.**read**(*log_f*, *raise_on_error=False*)
Return a generator that will return the entries in a structured log file. Note that the caller must not close the file whilst the generator is still in use.

Parameters

- **log_f** – file-like object containing the raw log entries, one per line
- **raise_on_error** – boolean indicating whether ValueError should be raised for lines that cannot be decoded.

Integration with argparse

The *mozlog.commandline* module provides integration with the *argparse* module to provide uniform logging-related command line arguments to programs using *mozlog*. Each known formatter gets a command line argument of the form `--log-{name}`, which takes the name of a file to log to with that format, or – to indicate stdout.

mozlog.commandline.**TEXT_FORMATTERS = ('raw', 'mach')**
a subset of formatters for non test harnesses related applications

mozlog.commandline.**add_logging_group**(*parser*, *include_formatters=None*)
Add logging options to an argparse ArgumentParser or optparse OptionParser.

Each formatter has a corresponding option of the form –log-{name} where {name} is the name of the formatter. The option takes a value which is either a filename or "-" to indicate stdout.

Parameters

- **parser** – The ArgumentParser or OptionParser object that should have logging options added.
- **include_formatters** – List of formatter names that should be included in the option group. Default to None, meaning all the formatters are included. A common use of this option is to specify *TEXT_FORMATTERS* to include only the most useful formatters for a command line tool that is not related to test harnesses.

mozlog.commandline.**setup_handlers**(*logger*, *formatters*, *formatter_options*, *allow_unused_options=False*)
Add handlers to the given logger according to the formatters and options provided.

Parameters

- **logger** – The logger configured by this function.
- **formatters** – A dict of {formatter, [streams]} to use in handlers.
- **formatter_options** – a dict of {formatter: {option: value}} to to use when configuring formatters.

mozlog.commandline.**setup_logging**(*logger*, *args*, *defaults=None*, *formatter_defaults=None*, *allow_unused_options=False*)
Configure a structuredlogger based on command line arguments.

The created structuredlogger will also be set as the default logger, and can be retrieved with `get_default_logger()`.

Parameters

- **logger** – A StructuredLogger instance or string name. If a string, a new StructuredLogger instance will be created using *logger* as the name.
- **args** – A dictionary of {argument_name:value} produced from parsing the command line arguments for the application
- **defaults** – A dictionary of {formatter name: output stream} to apply when there is no logging supplied on the command line. If this isn't supplied, reasonable defaults are chosen (coloured mach formatting if stdout is a terminal, or raw logs otherwise).
- **formatter_defaults** – A dictionary of {option_name: default_value} to provide to the formatters in the absence of command line overrides.

Return type *StructuredLogger*

Simple Examples

Log to stdout:

```
from mozlog import structuredlog
from mozlog import handlers, formatters
logger = structuredlog.StructuredLogger("my-test-suite")
logger.add_handler(handlers.StreamHandler(sys.stdout,
                                          formatters.JSONFormatter()))
logger.suite_start(["test-id-1"])
logger.test_start("test-id-1")
logger.info("This is a message with action='LOG' and level='INFO'")
logger.test_status("test-id-1", "subtest-1", "PASS")
logger.test_end("test-id-1", "OK")
logger.suite_end()
```

Populate an `argparse.ArgumentParser` with logging options, and create a logger based on the value of those options, defaulting to JSON output on stdout if nothing else is supplied:

```
import argparse
from mozlog import commandline

parser = argparse.ArgumentParser()
# Here one would populate the parser with other options
commandline.add_logging_group(parser)

args = parser.parse_args()
logger = commandline.setup_logging("testsuite-name", args, {"raw": sys.stdout})
```

Count the number of tests that timed out in a testsuite:

```
from mozlog import reader

count = 0

def handle_test_end(data):
    global count
    if data["status"] == "TIMEOUT":
        count += 1
```

```
reader.each_log(reader.read("my_test_run.log"),
                {"test_end": handle_test_end})

print count
```

More Complete Example

This example shows a complete toy testharness set up to used structured logging. It is avaliable as structured_example.py:

```
import argparse
import sys
import traceback
import types

from mozlog import commandline, get_default_logger


class TestAssertion(Exception):
    pass


def assert_equals(a, b):
    if a != b:
        raise TestAssertion("%r not equal to %r" % (a, b))


def expected(status):
    def inner(f):
        def test_func():
            f()
        test_func.__name__ = f.__name__
        test_func._expected = status
        return test_func
    return inner


def test_that_passes():
    assert_equals(1, int("1"))


def test_that_fails():
    assert_equals(1, int("2"))


def test_that_has_an_error():
    assert_equals(2, 1 + "1")


@expected("FAIL")
def test_expected_fail():
    assert_equals(2 + 2, 5)


class TestRunner(object):
```

```
    def __init__(self):
        self.logger = get_default_logger(component='TestRunner')

    def gather_tests(self):
        for item in globals().itervalues():
            if isinstance(item, types.FunctionType) and item.__name__.startswith(
↪"test_"):
                yield item.__name__, item

    def run(self):
        tests = list(self.gather_tests())

        self.logger.suite_start(tests=[name for name, func in tests])
        self.logger.info("Running tests")
        for name, func in tests:
            self.run_test(name, func)
        self.logger.suite_end()

    def run_test(self, name, func):
        self.logger.test_start(name)
        status = None
        message = None
        expected = func._expected if hasattr(func, "_expected") else "PASS"
        try:
            func()
        except TestAssertion as e:
            status = "FAIL"
            message = e.message
        except:
            status = "ERROR"
            message = traceback.format_exc()
        else:
            status = "PASS"
        self.logger.test_end(name, status=status, expected=expected, message=message)


def get_parser():
    parser = argparse.ArgumentParser()
    return parser


def main():
    parser = get_parser()
    commandline.add_logging_group(parser)

    args = parser.parse_args()

    logger = commandline.setup_logging("structured-example", args, {"raw": sys.stdout}
↪)

    runner = TestRunner()
    try:
        runner.run()
    except:
        logger.critical("Error during test run:\n%s" % traceback.format_exc())
```

```
if __name__ == "__main__":
    main()
```

Each global function with a name starting `test_` represents a test. A passing test returns without throwing. A failing test throws a `TestAssertion` exception via the `assert_equals()` function. Throwing anything else is considered an error in the test. There is also a `expected()` decorator that is used to annotate tests that are expected to do something other than pass.

The main entry point to the test runner is via that `main()` function. This is responsible for parsing command line arguments, and initiating the test run. Although the test harness itself does not provide any command line arguments, the `ArgumentParser` object is populated by `commandline.add_logging_group()`, which provides a generic set of structured logging arguments appropriate to all tools producing structured logging.

The values of these command line arguments are used to create a `mozlog.StructuredLogger` object populated with the specified handlers and formatters in `commandline.setup_logging()`. The third argument to this function is the default arguments to use. In this case the default is to output raw (i.e. JSON-formatted) logs to stdout.

The main test harness is provided by the `TestRunner` class. This class is responsible for scheduling all the tests and logging all the results. It is passed the `logger` object created from the command line arguments. The `run()` method starts the test run. Before the run is started it logs a `suite_start` message containing the id of each test that will run, and after the testrun is done it logs a `suite_end` message.

Individual tests are run in the `run_test()` method. For each test this logs a `test_start` message. It then runs the test and logs a `test_end` message containing the test name, status, expected status, and any informational message about the reason for the result. In this test harness there are no subtests, so the `test_end` message has the status of the test and there are no `test_status` messages.

Example Output

When run without providing any command line options, the raw structured log messages are sent to stdout:

```
$ python structured_example.py

{"source": "structured-example", "tests": ["test_that_has_an_error", "test_that_fails
↪", "test_expected_fail", "test_that_passes"], "thread": "MainThread", "time":␣
↪1401446682787, "action": "suite_start", "pid": 18456}
{"source": "structured-example", "thread": "MainThread", "time": 1401446682787,
↪"action": "log", "message": "Running tests", "level": "INFO", "pid": 18456}
{"source": "structured-example", "test": "test_that_has_an_error", "thread":
↪"MainThread", "time": 1401446682787, "action": "test_start", "pid": 18456}
{"status": "ERROR", "thread": "MainThread", "pid": 18456, "source": "structured-
↪example", "test": "test_that_has_an_error", "time": 1401446682788, "action": "test_
↪end", "message": "Traceback (most recent call last):\n  File \"structured_example.
↪py\", line 61, in run_test\n    func()\n  File \"structured_example.py\", line 31,␣
↪in test_that_has_an_error\n    assert_equals(2, 1 + \"1\")\nTypeError: unsupported␣
↪operand type(s) for +: 'int' and 'str'\n", "expected": "PASS"}
{"source": "structured-example", "test": "test_that_fails", "thread": "MainThread",
↪"time": 1401446682788, "action": "test_start", "pid": 18456}
{"status": "FAIL", "thread": "MainThread", "pid": 18456, "source": "structured-example
↪", "test": "test_that_fails", "time": 1401446682788, "action": "test_end", "message
↪": "1 not equal to 2", "expected": "PASS"}
{"source": "structured-example", "test": "test_expected_fail", "thread": "MainThread",
↪ "time": 1401446682788, "action": "test_start", "pid": 18456}
{"status": "FAIL", "thread": "MainThread", "pid": 18456, "source": "structured-example
↪", "test": "test_expected_fail", "time": 1401446682788, "action": "test_end",
↪"message": "4 not equal to 5"}
{"source": "structured-example", "test": "test_that_passes", "thread": "MainThread",
↪"time": 1401446682788, "action": "test_start", "pid": 18456}
```

```
{"status": "PASS", "source": "structured-example", "test": "test_that_passes", "thread
↪": "MainThread", "time": 1401446682789, "action": "test_end", "pid": 18456}
{"action": "suite_end", "source": "structured-example", "pid": 18456, "thread":
↪"MainThread", "time": 1401446682789}
```

The structured logging module provides a number of command line options:

```
$ python structured_example.py --help

usage: structured_example.py [-h] [--log-unittest LOG_UNITTEST]
                             [--log-raw LOG_RAW] [--log-html LOG_HTML]
                             [--log-xunit LOG_XUNIT]
                             [--log-mach LOG_MACH]

optional arguments:
  -h, --help            show this help message and exit

Output Logging:
  Options for logging output. Each option represents a possible logging
  format and takes a filename to write that format to, or '-' to write to
  stdout.

  --log-unittest LOG_UNITTEST
                        Unittest style output
  --log-raw LOG_RAW     Raw structured log messages
  --log-html LOG_HTML   HTML report
  --log-xunit LOG_XUNIT
                        xUnit compatible XML
  --log-mach LOG_MACH   Human-readable output
```

In order to get human-readable output on stdout and the structured log data to go to the file `structured.log`, we would run:

```
$ python structured_example.py --log-mach=- --log-raw=structured.log

0:00.00 SUITE_START: MainThread 4
0:01.00 LOG: MainThread INFO Running tests
0:01.00 TEST_START: MainThread test_that_has_an_error
0:01.00 TEST_END: MainThread Harness status ERROR, expected PASS. Subtests passed 0/0.
↪ Unexpected 1
0:01.00 TEST_START: MainThread test_that_fails
0:01.00 TEST_END: MainThread Harness status FAIL, expected PASS. Subtests passed 0/0.␣
↪Unexpected 1
0:01.00 TEST_START: MainThread test_expected_fail
0:02.00 TEST_END: MainThread Harness status FAIL. Subtests passed 0/0. Unexpected 0
0:02.00 TEST_START: MainThread test_that_passes
0:02.00 TEST_END: MainThread Harness status PASS. Subtests passed 0/0. Unexpected 0
0:02.00 SUITE_END: MainThread
```

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index

G

H

I

J

K

L

M

N

P

R

S

T

U

W

X