autotest Documentation

Release 0.16.0

Autotest Team

August 28, 2015

Contents

Welcome! This is the Autotest documentation.

Autotest is a framework for fully automated testing.

It is designed primarily to test the Linux kernel, though it is useful for many other functions such as qualifying new hardware.

Contents:

CHAPTER 1

Autotest Client Test Documentation

The Autotest Client Tests are the most common type of tests (the other being Server tests).

Contents:

1.1 Linux Distribution Detection

Autotest has a facility that lets tests determine quite precisely the distribution they're running on.

This is done through the implementation and registration of probe classes.

Those probe classes can check for given characteristics of the running operating system, such as the existence of a release file, its contents or even the existence of a binary that is exclusive to a distribution (such as package managers).

1.1.1 Quickly detecting the Linux Distribution

The *autotest.client.shared.distro* module provides many APIs, but the simplest one to use is the *detect()*.

Its usage is quite straighforward:

```
>>> from autotest.client.shared import distro
>>> detected_distro = distro.detect()
```

The returned distro can be the result of a probe validating the distribution detection, or the not so useful *UNKNOWN_DISTRO*.

To access the relevant data on a *LinuxDistro*, simply use the attributes:

- name
- version
- release
- arch

Example:

```
>>> detected_distro = distro.detect()
>>> print detected_distro.name
redhat
```

1.1.2 The unknown Linux Distribution

When the detection mechanism can't precily detect the Linux Distribution, it will still return a *LinuxDistro* instance, but a special one that contains special values for its name, version, etc.

autotest.client.shared.distro.**UNKNOWN_DISTRO = <LinuxDistro: name=unknown, version=0, release=0, arch=unk**
The distribution that is used when the exact one could not be found

1.1.3 Writing a Linux Distribution Probe

The easiest way to write a probe for your target Linux Distribution is to make use of the features of the *Probe* class.

Even if you do plan to use the features documented here, keep in mind that all probes should inherit from *Probe* and provide a basic interface.

Checking the Distrution name only

The most trivial probe is one that checks the existence of a file and returns the distribution name:

```
class RedHatProbe(Probe):
    CHECK_FILE = '/etc/redhat-release'
    CHECK_FILE_DISTRO_NAME = 'redhat'
```

To make use of a probe, it's necessary to register it:

```
>>> from autotest.client.shared import distro
>>> distro.register_probe(RedHatProbe)
```

And that's it. This is a valid example, but will give you nothing but the distro name.

You should usually aim for more information, such as the version numbers.

Checking the Distribution name and version numbers

If you want to also detect the distro version numbers (and you should), then it's possible to use the *Probe.CHECK_VERSION_REGEX* feature of the *Probe* class.

Probe.**CHECK_VERSION_REGEX = None**
A regular expresion that will be run on the file pointed to by CHECK_FILE_EXISTS

If your regex has two or more groups, that is, it will look for and save references to two or more string, it will consider the second group to be the LinuxDistro.release number.

Probe Scores

To increase the accuracy of the probe results, it's possible to register a score for a probe. If a probe wants to, it can register a score for itself.

Probes that return a score will be given priority over probes that don't.

The score should be based on the number of checks that ran during the probe to account for its accuracy.

Probes should not be given a higher score because their checks look more precise than everyone else's.

Registering your own probes

Not only the probes that ship with autotest can be used, but your custom probe classes can be added to the detection system.

To do that simply call the function *register_probe()*:

autotest.client.shared.distro.**register_probe**(*probe_class*)
Register a probe to be run during autodetection

Now, remember that for things to happen smootlhy your registered probe must be a subclass of *Probe*.

1.1.4 API Reference

LinuxDistro

class autotest.client.shared.distro.**LinuxDistro**(*name*, *version*, *release*, *arch*)
Simple collection of information for a Linux Distribution

Probe

class autotest.client.shared.distro.**Probe**
Probes the machine and does it best to confirm it's the right distro

CHECK_FILE = None
Points to a file that can determine if this machine is running a given Linux Distribution. This servers a first check that enables the extra checks to carry on.

CHECK_FILE_CONTAINS = None
Sets the content that should be checked on the file pointed to by CHECK_FILE_EXISTS. Leave it set to *None* (its default) to check only if the file exists, and not check its contents

CHECK_FILE_DISTRO_NAME = None
The name of the Linux Distribution to be returned if the file defined by CHECK_FILE_EXISTS exist.

CHECK_VERSION_REGEX = None
A regular expresion that will be run on the file pointed to by CHECK_FILE_EXISTS

check_name_for_file()
Checks if this class will look for a file and return a distro

The conditions that must be true include the file that identifies the distro file being set (*CHECK_FILE*) and the name of the distro to be returned (*CHECK_FILE_DISTRO_NAME*)

check_name_for_file_contains()
Checks if this class will look for text on a file and return a distro

The conditions that must be true include the file that identifies the distro file being set (*CHECK_FILE*), the text to look for inside the distro file (*CHECK_FILE_CONTAINS*) and the name of the distro to be returned (*CHECK_FILE_DISTRO_NAME*)

check_release()
Checks if this has the conditions met to look for the release number

check_version()
Checks if this class will look for a regex in file and return a distro

get_distro()
Returns the *LinuxDistro* this probe detected

name_for_file()
Get the distro name if the *CHECK_FILE* is set and exists

name_for_file_contains()
Get the distro if the *CHECK_FILE* is set and has content

release()
Returns the release of the distro

version()
Returns the version of the distro

register_probe()

autotest.client.shared.distro.**register_probe**(*probe_class*)
Register a probe to be run during autodetection

detect()

autotest.client.shared.distro.**detect**()
Attempts to detect the Linux Distribution running on this machine

Returns the detected *LinuxDistro* or *UNKNOWN_DISTRO*

Return type *LinuxDistro*

CHAPTER 2

Autotest Shared Definitions

As some of Autotest's functionality is being split into external components and source code repositories, a new namespace, `autotest.shared` contains definitions that is supposed be common and thus shared among these.

Contents:

2.1 Frontend

Basic definitions for the frontend.

Note that the frontend is broader in scope and functionality than the rpc server. Another way to put that is the rpc server is a subset of the frontend.

`autotest.shared.frontend.`**`AFE_SERVICE_NAME`** **= 'afe'**
The name of the "AFE" service, used when accessing that service

`autotest.shared.frontend.`**`TKO_SERVICE_NAME`** **= 'tko'**
The name of the "TKO" service, used when accessing that service

`autotest.shared.frontend.`**`AFE_URL_PREFIX`** **= 'afe/server/'**
Prefix applied to all AFE URLs. This information is useful if requests are coming through apache, and you need this app to coexist with others

`autotest.shared.frontend.`**`TKO_URL_PREFIX`** **= 'new_tko/server/'**
Prefix applied to the TKO server frontend

2.2 RPC

Basic definitions for the rpc services.

`autotest.shared.rpc.`**`DEFAULT_PATH`** **= '/'**
RPC path to use for unknown service

`autotest.shared.rpc.`**`AFE_PATH`** **= 'afe/server/rpc/'**
RPC path for the AFE service

`autotest.shared.rpc.`**`TKO_PATH`** **= 'new_tko/server/rpc/'**
RPC path for the TKO service

`autotest.shared.rpc.`**`PATHS`** **= {'afe': 'afe/server/rpc/', 'tko': 'new_tko/server/rpc/'}**
The service available on a regular Autotest RPC server and their RPC PATHS

CHAPTER 3

RPC Server

The Autotest RPC Server, also known as the frontend, is a Django based application that provides:

- The Database Objects (defined by Django `Models`)
- A remoting interface using the JSON-RPC protocol
- The `Administration Web Interface` that Django gives us for free

We'll start by taking a look at the Database the Models and the database structure that they generate.

Contents:

3.1 Models

The Database Models play a major role in the RPC server. The most important things they do:

- Define and create the database structure on the Autotest Relational Database
- Provide a object like uniform API for the Database entries

Note: For historical reasons, the RPC server is composed of two different applications, AFE and TKO. Because of that, the models are also defined in two different modules.

These may soon be united into a single application, specially their model definition. For now, keep in mind that the model you are looking for may be in one of two different places.

3.1.1 Model Logic

Autotest extends the base Django Database models with some custom logic.

`ModelWithInvalid`

3.1.2 AFE Models

AFE stands for Autotest Front End. It's an application that provides access to the core of Autotest definitions, such as Hosts, Tests, Jobs, etc.

For the classes that inherit from `django.db.models.Model` some of the attributes documented here are instances from one of the many `django.db.models.fields` classes and will be mapped into a field on the relational database.

AtomicGroup

Job

Label

Drone

DroneSet

User

Host

HostAttribute

Test

TestParameter

Profiler

AclGroup

Kernel

ParameterizedJob

ParameterizedJobProfiler

ParameterizedJobProfilerParameter

ParameterizedJobParameter

Job

3.1.3 AFE Exceptions

Besides persistence, Models also provide some logic. And as such, some custom error conditions exist.

3.1.4 TKO Models

TKO is the autotest application dedicated to storing and querying test results.

Machine

Kernel

Patch

Status

Job

JobKeyval

Test

CHAPTER 4

client Package

4.1 autotest_local Module

class autotest.client.autotest_local.**AutotestLocalApp**
Autotest local app runs tests locally

Point it to a control file and let it rock

main()

parse_cmdline()

usage()

4.2 base_sysinfo Module

class autotest.client.base_sysinfo.**base_sysinfo**(*job_resultsdir*)
Bases: object

deserialize(*serialized*)

log_after_each_iteration(**args*, ***dargs*)

log_after_each_test(**args*, ***dargs*)

log_before_each_iteration(**args*, ***dargs*)

log_before_each_test(**args*, ***dargs*)

log_per_reboot_data(**args*, ***dargs*)

log_test_keyvals(*test_sysinfodir*)
Logging hook called by log_after_each_test to collect keyval entries to be written in the test keyval.

serialize()

class autotest.client.base_sysinfo.**command**(*cmd*, *logf=None*, *log_in_keyval=False*, *compress_log=False*)
Bases: *autotest.client.base_sysinfo.loggable*

run(*logdir*)

class autotest.client.base_sysinfo.**logfile**(*path*, *logf=None*, *log_in_keyval=False*)
Bases: *autotest.client.base_sysinfo.loggable*

run(*logdir*)

class autotest.client.base_sysinfo.**loggable**(*logf*, *log_in_keyval*)
Bases: object

Abstract class for representing all things "loggable" by sysinfo.

readline(*logdir*)

4.3 base_utils Module

DO NOT import this file directly - import client/bin/utils.py, which will mix this in

Convenience functions for use by tests or whomever.

Note that this file is mixed in by utils.py - note very carefully the precedence order defined there

autotest.client.base_utils.**append_path**(*oldpath*, *newpath*)
append newpath to oldpath

autotest.client.base_utils.**avgtime_print**(*dir*)
Calculate some benchmarking statistics. Input is a directory containing a file called 'time'. File contains one-per-line results of /usr/bin/time. Output is average Elapsed, User, and System time in seconds,

and average CPU percentage.

autotest.client.base_utils.**cat_file_to_cmd**(*file*, *command*, *ignore_status=0*, *return_output=False*)
equivalent to 'cat file | command' but knows to use zcat or bzcat if appropriate

autotest.client.base_utils.**check_for_kernel_feature**(*feature*)

autotest.client.base_utils.**check_glibc_ver**(*ver*)

autotest.client.base_utils.**check_kernel_ver**(*ver*)

autotest.client.base_utils.**count_cpus**()
number of CPUs in the local machine according to /proc/cpuinfo

autotest.client.base_utils.**cpu_has_flags**(*flags*)
Check if a list of flags are available on current CPU info

Parameters **flags** – A *list* of cpu flags that must exists

on the current CPU. :type flags: *list* :returns: *bool* True if all the flags were found or False if not :rtype: *list*

autotest.client.base_utils.**cpu_online_map**()
Check out the available cpu online map

autotest.client.base_utils.**difflist**(*list1*, *list2*)
returns items in list2 that are not in list1

autotest.client.base_utils.**disk_block_size**(*path*)
Return the disk block size, in bytes

autotest.client.base_utils.**dump_object**(*object*)
Dump an object's attributes and methods

kind of like dir()

autotest.client.base_utils.**environ**(*env_key*)
return the requested environment variable, or '' if unset

autotest.client.base_utils.**extract_all_time_results**(*results_string*)
Extract user, system, and elapsed times into a list of tuples

autotest.client.base_utils.**extract_tarball**(*tarball*)
Returns the directory extracted by the tarball.

autotest.client.base_utils.**extract_tarball_to_dir**(*tarball*, *dir*)
Extract a tarball to a specified directory name instead of whatever the top level of a tarball is - useful for versioned directory names, etc

autotest.client.base_utils.**file_contains_pattern**(*file*, *pattern*)
Return true if file contains the specified egrep pattern

autotest.client.base_utils.**force_copy**(*src*, *dest*)
Replace dest with a new copy of src, even if it exists

autotest.client.base_utils.**force_link**(*src*, *dest*)
Link src to dest, overwriting it if it exists

autotest.client.base_utils.**freespace**(*path*)
Return the disk free space, in bytes

autotest.client.base_utils.**get_cc**()

autotest.client.base_utils.**get_cpu_arch**()
Work out which CPU architecture we're running on

autotest.client.base_utils.**get_cpu_family**()

autotest.client.base_utils.**get_cpu_info**()
Reads /proc/cpuinfo and returns a list of file lines

Returns *list* of lines from /proc/cpuinfo file

Return type *list*

autotest.client.base_utils.**get_cpu_stat**(*key*)
Get load per cpu from /proc/stat :return: list of values of CPU times

autotest.client.base_utils.**get_cpu_vendor**()

autotest.client.base_utils.**get_cpu_vendor_name**()
Get the current cpu vendor name

Returns string 'intel' or 'amd' or 'power7' depending

on the current CPU architecture. :rtype: *string*

autotest.client.base_utils.**get_current_kernel_arch**()
Get the machine architecture

autotest.client.base_utils.**get_disks**()

autotest.client.base_utils.**get_file_arch**(*filename*)

autotest.client.base_utils.**get_hwclock_seconds**(*utc=True*)
Return the hardware clock in seconds as a floating point value. Use Coordinated Universal Time if utc is True, local time otherwise. Raise a ValueError if unable to read the hardware clock.

autotest.client.base_utils.**get_loaded_modules**()

autotest.client.base_utils.**get_modules_dir**()
Return the modules dir for the running kernel version

autotest.client.base_utils.**get_os_vendor**()
Try to guess what's the os vendor.

autotest.client.base_utils.**get_systemmap**()
Return the full path to System.map

Ahem. This is crap. Pray harder. Bad Martin.

autotest.client.base_utils.**get_uptime**()

Returns return the uptime of system in secs in float

in error case return 'None'

autotest.client.base_utils.**get_vmlinux**()
Return the full path to vmlinux

Ahem. This is crap. Pray harder. Bad Martin.

autotest.client.base_utils.**grep**(*pattern*, *file*)
This is mainly to fix the return code inversion from grep Also handles compressed files.

returns 1 if the pattern is present in the file, 0 if not.

autotest.client.base_utils.**hash_file**(*filename*, *size=None*, *method='md5'*)
Calculate the hash of filename. If size is not None, limit to first size bytes. Throw exception if something is wrong with filename. Can be also implemented with bash one-liner (assuming size%1024==0): dd if=filename bs=1024 count=size/1024 | sha1sum -

Parameters

- **filename** – Path of the file that will have its hash calculated.
- **method** – Method used to calculate the hash. Supported methods: * md5 * sha1

Returns Hash of the file, if something goes wrong, return None.

autotest.client.base_utils.**human_format**(*number*)

autotest.client.base_utils.**list_grep**(*list*, *pattern*)
True if any item in list matches the specified pattern.

autotest.client.base_utils.**load_module**(*module_name*)

autotest.client.base_utils.**locate**(*pattern*, *root='/home/docs/checkouts/readthedocs.org/user_builds/autotest/checkouts*

autotest.client.base_utils.**module_is_loaded**(*module_name*)

autotest.client.base_utils.**pickle_load**(*filename*)

autotest.client.base_utils.**ping_default_gateway**()
Ping the default gateway.

autotest.client.base_utils.**prepend_path**(*newpath*, *oldpath*)
prepend newpath to oldpath

autotest.client.base_utils.**print_to_tty**(*string*)
Output string straight to the tty

autotest.client.base_utils.**process_is_alive**(*name_pattern*)
'pgrep name' misses all python processes and also long process names. 'pgrep -f name' gets all shell commands with name in args. So look only for command whose initial pathname ends with name. Name itself is an egrep pattern, so it can use | etc for variations.

autotest.client.base_utils.**running_config**()
Return path of config file of the currently running kernel

autotest.client.base_utils.**running_os_full_version**()

autotest.client.base_utils.**running_os_ident**()

autotest.client.base_utils.**running_os_release**()

autotest.client.base_utils.**set_power_state**(*state*)
Set the system power state to 'state'.

autotest.client.base_utils.**set_wake_alarm**(*alarm_time*)
Set the hardware RTC-based wake alarm to 'alarm_time'.

autotest.client.base_utils.**standby**()
Power-on suspend (S1)

autotest.client.base_utils.**suspend_to_disk**()
Suspend the system to disk (S4)

autotest.client.base_utils.**suspend_to_ram**()
Suspend the system to RAM (S3)

autotest.client.base_utils.**sysctl**(*key*, *value=None*)
Generic implementation of sysctl, to read and write.

Parameters

- **key** – A location under /proc/sys
- **value** – If not None, a value to write into the sysctl.

Returns The single-line sysctl value as a string.

autotest.client.base_utils.**sysctl_kernel**(*key*, *value=None*)
(Very) partial implementation of sysctl, for kernel params

autotest.client.base_utils.**to_seconds**(*time_string*)
Converts a string in M+:SS.SS format to S+.SS

autotest.client.base_utils.**unload_module**(*module_name*)
Removes a module. Handles dependencies. If even then it's not possible to remove one of the modules, it will trhow an error.CmdError exception.

Parameters **module_name** – Name of the module we want to remove.

autotest.client.base_utils.**unmap_url_cache**(*cachedir*, *url*, *expected_hash*, *method='md5'*)
Downloads a file from a URL to a cache directory. If the file is already at the expected position and has the expected hash, let's not download it again.

Parameters

- **cachedir** – Directory that might hold a copy of the file we want to download.
- **url** – URL for the file we want to download.
- **expected_hash** – Hash string that we expect the file downloaded to have.
- **method** – Method used to calculate the hash string (md5, sha1).

autotest.client.base_utils.**where_art_thy_filehandles**()
Dump the current list of filehandles

4.4 bkr_proxy Module

bkr_proxy - class used to talk to beaker

class autotest.client.bkr_proxy.**BkrProxy**(*recipe_id*, *labc_url=None*)
Bases: object

get_recipe()

recipe_abort()

recipe_stop()

recipe_upload_file(*localfile*, *remotepath=''*)

result_upload_file(*task_id*, *result_id*, *localfile*, *remotepath=''*)

task_abort(*task_id*)

task_result(*task_id*, *result_type*, *result_path*, *result_score*, *result_summary*)

task_start(*task_id*, *kill_time=0*)

task_stop(*task_id*)

task_upload_file(*task_id*, *localfile*, *remotepath=''*)

update_watchdog(*task_id*, *kill_time*)

exception autotest.client.bkr_proxy.**BkrProxyException**(*text*)
Bases: exceptions.Exception

autotest.client.bkr_proxy.**copy_data**(*data*, *dest*, *header=None*, *use_put=None*)
Copy data to a destination

To aid in debugging, copy a file locally to verify the contents. Attempts to write the same data that would otherwise be sent remotely.

Parameters

- **data** – data string to copy
- **dest** – destination path
- **header** – header info item to return
- **use_put** – dictionary of items for PUT method

Returns nothing or header info if requested

autotest.client.bkr_proxy.**copy_local**(*data*, *dest*, *use_put=None*)
Copy data locally to a file

To aid in debugging, copy a file locally to verify the contents. Attempts to write the same data that would otherwise be sent remotely.

Parameters

- **data** – encoded data string to copy locally
- **dest** – local file path
- **use_put** – chooses to write in binary or text

Returns nothing

autotest.client.bkr_proxy.**copy_remote**(*data*, *dest*, *use_put=None*)
Copy data to a remote server using http calls POST or PUT

Using http POST and PUT methods, copy data over http. To use PUT method, provide a dictionary of values to be populated in the Content-Range and Content-Length headers. Otherwise default is to use POST method.

Traps on HTTPError 500 and 400

Parameters

- **data** – encoded data string to copy remotely
- **dest** – remote server URL
- **use_put** – dictionary of items if using PUT method

Returns html header info for post processing

autotest.client.bkr_proxy.**make_path_bkrcache**(*r*)
Converts a recipe id into an internal path for cache'ing recipe

Parameters **r** – recipe id

Returns a path to the internal recipe cache file

autotest.client.bkr_proxy.**make_path_cmdlog**(*r*)
Converts a recipe id into an internal path for logging purposes

Parameters **r** – recipe id

Returns a path to the internal command log

autotest.client.bkr_proxy.**make_path_log**(*r*, *t=None*, *i=None*)
Converts id into a beaker path to log file

Given a recipe id, a task id, and/or result id, translate them into the proper beaker path to the log file. Depending on which log file is needed, provide the appropriate params. Note the dependency, a result id needs a task id and recipe id, while a task id needs a recipe id.

Parameters

- **r** – recipe id
- **t** – task id
- **i** – result id

Returns a beaker path of the task's result file

autotest.client.bkr_proxy.**make_path_recipe**(*r*)
Converts a recipe id into a beaker path

Parameters **r** – recipe id

Returns a beaker path to the recipe id

autotest.client.bkr_proxy.**make_path_result**(*r*, *t*)
Converts task id into a beaker path to result file

Given a recipe id and a task id, translate them into the proper beaker path to the result file.

Parameters

- **r** – recipe id
- **t** – task id

Returns a beaker path of the task's result file

autotest.client.bkr_proxy.**make_path_status**(*r*, *t=None*)
Converts id into a beaker path to status file

Given a recipe id and/or a task id, translate them into the proper beaker path to the status file. Recipe only, returns the path to the recipe's status, whereas including a task returns the path to the task's status.

Parameters

- **r** – recipe id

- **t** – task id

Returns a beaker path of the recipe's/task's status file

autotest.client.bkr_proxy.**make_path_watchdog**(*r*)
Converts a recipe id into a beaker path for the watchdog

Parameters **r** – recipe id

Returns a beaker path of the recipe's watchdog file

4.5 bkr_xml Module

module to parse beaker xml recipe

class autotest.client.bkr_xml.**BeakerXMLParser**
Bases: object

Handles parsing of beaker job xml

handle_recipe(*recipe_node*)

handle_recipes(*recipe_nodes*)

handle_task(*recipe*, *task_node*)

handle_task_param(*task*, *param_node*)

handle_task_params(*task*, *param_nodes*)

handle_tasks(*recipe*, *task_nodes*)

parse_from_file(*file_name*)

parse_xml(*xml*)
Returns dict, mapping hostname to recipe

class autotest.client.bkr_xml.**Recipe**
Bases: object

class autotest.client.bkr_xml.**Task**
Bases: object

Simple record to store task properties

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

autotest.client.bkr_xml.**xml_attr**(*node*, *key*, *default=None*)

autotest.client.bkr_xml.**xml_get_nodes**(*node*, *tag*)

4.6 client_logging_config Module

class autotest.client.client_logging_config.**ClientLoggingConfig**(*use_console=True*)
Bases: *autotest.client.shared.logging_config.LoggingConfig*

add_debug_file_handlers(*log_dir*, *log_name=None*)

configure_logging(*results_dir=None*, *verbose=False*)

4.7 cmdparser Module

Autotest command parser

copyright Don Zickus <dzickus@redhat.com> 2011

class autotest.client.cmdparser.**CmdParserLoggingConfig**(*use_console=True*)
Bases: *autotest.client.shared.logging_config.LoggingConfig*

Used with the sole purpose of providing convenient logging setup for the KVM test auxiliary programs.

configure_logging(*results_dir=None*, *verbose=False*)

class autotest.client.cmdparser.**CommandParser**
Bases: object

A client-side command wrapper for the autotest client.

COMMAND_LIST = ['help', 'list', 'run', 'fetch', 'bootstrap']

bootstrap(*args*, *options*)
Bootstrap autotest by fetching the control file first and pass it back

Currently this relies on a harness to retrieve the file

fetch(*args*, *options*)
fetch a remote control file or packages

classmethod help()
List the commands and their usage strings.

:param args is not used here.

classmethod list_tests()
List the available tests for users to choose from

parse_args(*args*, *options*)
Process a client side command.

Parameters **args** – Command line args.

run(*args*, *options*)
Wrap args with a path and send it back to autotest.

4.8 common Module

4.9 config Module

The Job Configuration

The job configuration, holding configuration variable supplied to the job.

The config should be viewed as a hierarchical namespace. The elements of the hierarchy are separated by periods (.) and where multiple words are required at a level they should be separated by underscores (_). Please no StudlyCaps.

For example: boot.default_args

class autotest.client.config.**config**(*job*)
Bases: object

The BASIC job configuration

Properties:

job The job object for this job

config The job configuration dictionary

get(*name*)

set(*name*, *value*)

4.10 cpuset Module

autotest.client.cpuset.**abbrev_list**(*vals*)
Condense unsigned (0,4,5,6,7,10) to '0,4-7,10'.

autotest.client.cpuset.**all_drive_names**()

autotest.client.cpuset.**avail_mbytes**(*parent=''*)

autotest.client.cpuset.**available_exclusive_mem_nodes**(*parent_container*)

autotest.client.cpuset.**container_bytes**(*name*)

autotest.client.cpuset.**container_exists**(*name*)

autotest.client.cpuset.**container_mbytes**(*name*)

autotest.client.cpuset.**cpus_path**(*container_name*)

autotest.client.cpuset.**cpuset_attr**(*container_name*, *attr*)

autotest.client.cpuset.**create_container_directly**(*name*, *mbytes*, *cpus*)

autotest.client.cpuset.**create_container_via_memcg**(*name*, *parent*, *bytes*, *cpus*)

autotest.client.cpuset.**create_container_with_mbytes_and_specific_cpus**(*name*, *mbytes*, *cpus=None*, *root=''*, *io={}*, *move_in=True*, *timeout=0*)
Create a cpuset container and move job's current pid into it Allocate the list "cpus" of cpus to that container

name = arbitrary string tag mbytes = reqested memory for job in megabytes cpus = list of cpu indices to associate with the cpuset

defaults to all cpus avail with given root

root = the parent cpuset to nest this new set within '': unnested top-level container

io = arguments for proportional IO containers move_in = True: Move current process into the new container now. timeout = must be 0: persist until explicitly deleted.

autotest.client.cpuset.**create_container_with_specific_mems_cpus**(*name*, *mems*, *cpus*)

autotest.client.cpuset.**delete_leftover_test_containers**()

autotest.client.cpuset.**discover_container_style**()

autotest.client.cpuset.**full_path**(*container_name*)

autotest.client.cpuset.**get_boot_numa**()

autotest.client.cpuset.**get_cpus**(*container_name*)

autotest.client.cpuset.**get_mem_nodes**(*container_name*)

autotest.client.cpuset.**get_tasks**(*container_name*)

autotest.client.cpuset.**inner_containers_of**(*parent*)

autotest.client.cpuset.**io_attr**(*container_name*, *attr*)

autotest.client.cpuset.**mbytes_per_mem_node**()

autotest.client.cpuset.**memory_path**(*container_name*)

autotest.client.cpuset.**mems_path**(*container_name*)

autotest.client.cpuset.**move_self_into_container**(*name*)

autotest.client.cpuset.**move_tasks_into_container**(*name*, *tasks*)

autotest.client.cpuset.**my_available_exclusive_mem_nodes**()

autotest.client.cpuset.**my_container_name**()

autotest.client.cpuset.**my_lock**(*lockname*)

autotest.client.cpuset.**my_mem_nodes**()

autotest.client.cpuset.**my_unlock**(*lockfile*)

autotest.client.cpuset.**need_fake_numa**()

autotest.client.cpuset.**need_mem_containers**()

autotest.client.cpuset.**node_avail_kbytes**(*node*)

autotest.client.cpuset.**nodes_avail_mbytes**(*nodes*)

autotest.client.cpuset.**rangelist_to_set**(*rangelist*)

autotest.client.cpuset.**release_container**(*container_name=None*)

autotest.client.cpuset.**remove_empty_prio_classes**(*prios*)

autotest.client.cpuset.**set_io_controls**(*container_name, disks=[], ioprio_classes=[2], io_shares=[95], io_limits=[0]*)

autotest.client.cpuset.**tasks_path**(*container_name*)

autotest.client.cpuset.**unpath**(*container_path*)

4.11 fsdev_disks Module

autotest.client.fsdev_disks.**finish_fsdev**(*force_cleanup=False*)

This method can be called from the test file to optionally restore all the drives used by the test to a standard ext2 format. Note that if use_fsdev_lib() was invoked with 'reinit_disks' not set to True, this method does nothing. Note also that only fsdev "server-side" dynamic control files should ever set force_cleanup to True.

class autotest.client.fsdev_disks.**fsdev_disks**(*job*)

Disk drive handling class used for file system development

config_sched_tunables(*desc_file*)

get_fsdev_mgr()

load_sched_tunable_values(*val_file*)

set_sched_tunables(*disks*)
Given a list of disks in the format returned by get_disk_list() above, set the I/O scheduler values on all the disks to the values loaded earlier by load_sched_tunables().

set_tunable(*disk*, *name*, *path*, *val*)
Given a disk name, a path to a tunable value under _TUNE_PATH and the new value for the parameter, set the value and verify that the value has been successfully set.

autotest.client.fsdev_disks.**get_disk_list**(*std_mounts_only=True*, *get_all_disks=False*)
Get a list of dictionaries with information about disks on this system.

Parameters

- **std_mounts_only** – Whether the function should return only disks that have a mount point defined (True) or even devices that doesn't (False).
- **get_all_disks** – Whether the function should return only partitioned disks (False) or return every disk, regardless of being partitioned or not (True).

Returns List of dictionaries with disk information (see more below).

The 'disk_list' array returned by get_disk_list() has an entry for each disk drive we find on the box. Each of these entries is a map with the following 3 string values:

'device' disk device name (i.e. the part after /dev/) 'mountpt' disk mount path 'tunable' disk name for setting scheduler tunables (/sys/block/sd??)

The last value is an integer that indicates the current mount status of the drive:

'mounted' 0 = not currently mounted

1 = mounted r/w on the expected path

-1 = mounted readonly or at an unexpected path

When the 'std_mounts_only' argument is True we don't include drives mounted on 'unusual' mount points in the result. If a given device is partitioned, it will return all partitions that exist on it. If it's not, it will return the device itself (ie, if there are /dev/sdb1 and /dev/sdb2, those will be returned but not /dev/sdb. if there is only a /dev/sdc, that one will be returned).

autotest.client.fsdev_disks.**match_fs**(*disk*, *dev_path*, *fs_type*, *fs_makeopt*)
Matches the user provided fs_type and fs_makeopt with the current disk.

autotest.client.fsdev_disks.**mkfs_all_disks**(*job*, *disk_list*, *fs_type*, *fs_makeopt*, *fs_mnt_opt*)
Prepare all the drives in 'disk_list' for testing. For each disk this means unmounting any mount points that use the disk, running mkfs with 'fs_type' as the file system type and 'fs_makeopt' as the 'mkfs' options, and finally remounting the freshly formatted drive using the flags in 'fs_mnt_opt'.

autotest.client.fsdev_disks.**prepare_disks**(*job*, *fs_desc*, *disk1_only=False*, *disk_list=None*)
Prepare drive(s) to contain the file system type / options given in the description line 'fs_desc'. When 'disk_list' is not None, we prepare all the drives in that list; otherwise we pick the first available data drive (which is usually hdc3) and prepare just that one drive.

Args:

fs_desc: A partition.FsOptions instance describing the test -OR- a

legacy string describing the same in '/' separated format: 'fstype / mkfs opts / mount opts / short name'.

disk1_only: Boolean, defaults to False. If True, only test the first disk.

disk_list: A list of disks to prepare. If None is given we default to asking get_disk_list().

Returns: (mount path of the first disk, short name of the test, list of disks) OR (None, '', None) if no fs_desc was given.

autotest.client.fsdev_disks.**prepare_fsdev**(*job*)
Called from the test file to get the necessary drive(s) ready; return a pair of values: the absolute path to the first drive's mount point plus the complete disk list (which is useful for tests that need to use more than one drive).

autotest.client.fsdev_disks.**restore_disks**(*job*, *restore=False*, *disk_list=None*)
Restore ext2 on the drives in 'disk_list' if 'restore' is True; when disk_list is None, we do nothing.

autotest.client.fsdev_disks.**use_fsdev_lib**(*fs_desc*, *disk1_only*, *reinit_disks*)
Called from the control file to indicate that fsdev is to be used.

autotest.client.fsdev_disks.**wipe_disks**(*job*, *disk_list*)
Wipe all of the drives in 'disk_list' using the 'wipe' functionality in the filesystem class.

4.12 fsdev_mgr Module

This module defines the BaseFsdevManager Class which provides an implementation of the 'fsdev' helper API; site specific extensions to any of these methods should inherit this class.

class autotest.client.fsdev_mgr.**BaseFsdevManager**
Bases: object

check_mount_point(*part_name*, *mount_point*)

Parameters

- **part_name** – A partition name such as 'sda3' or similar.
- **mount_point** – A mount point such as '/usr/local' or an empty string if no mount point is known.

Returns The expected mount point for part_name or a false value (None or '') if the client should not mount this partition.

include_partition(*part_name*)

map_drive_name(*part_name*)

use_partition(*part_name*)

Parameters **part_name** – A partition name such as 'sda3' or similar.

Returns bool, should we use this partition for testing?

class autotest.client.fsdev_mgr.**FsdevManager**
Bases: *autotest.client.fsdev_mgr.BaseFsdevManager*

autotest.client.fsdev_mgr.**SiteFsdevManager**
alias of *BaseFsdevManager*

4.13 fsinfo Module

This module gives the mkfs creation options for an existing filesystem.

tune2fs or xfs_growfs is called according to the filesystem. The results, filesystem tunables, are parsed and mapped to corresponding mkfs options.

autotest.client.fsinfo.**compare_features**(*needed_feature*, *current_feature*)
Compare two ext* feature lists.

autotest.client.fsinfo.**convert_conf_opt**(*default_opt*)

autotest.client.fsinfo.**ext_mkfs_options**(*tune2fs_dict*, *mkfs_option*)
Map the tune2fs options to mkfs options.

autotest.client.fsinfo.**ext_tunables**(*dev*)
Call tune2fs -l and parse the result.

autotest.client.fsinfo.**match_ext_options**(*fs_type*, *dev*, *needed_options*)
Compare the current ext* filesystem tunables with needed ones.

autotest.client.fsinfo.**match_mkfs_option**(*fs_type*, *dev*, *needed_options*)
Compare the current filesystem tunables with needed ones.

autotest.client.fsinfo.**match_xfs_options**(*dev*, *needed_options*)
Compare the current ext* filesystem tunables with needed ones.

autotest.client.fsinfo.**merge_ext_features**(*conf_feature*, *user_feature*)

autotest.client.fsinfo.**opt_string2dict**(*opt_string*)
Breaks the mkfs.ext* option string into dictionary.

autotest.client.fsinfo.**parse_mke2fs_conf**(*fs_type*, *conf_file='/etc/mke2fs.conf'*)
Parses mke2fs config file for default settings.

autotest.client.fsinfo.**xfs_mkfs_options**(*tune2fs_dict*, *mkfs_option*)
Maps filesystem tunables to their corresponding mkfs options.

autotest.client.fsinfo.**xfs_tunables**(*dev*)
Call xfs_grow -n to get filesystem tunables.

4.14 harness Module

The harness interface

The interface between the client and the server when hosted.

class autotest.client.harness.**harness**(*job*)
Bases: object

The NULL server harness

Properties:

job The job object for this job

run_abort()
A run within this job is aborting. It all went wrong

run_complete()
A run within this job is completing (all done)

run_pause()
A run within this job is completing (expect continue)

run_reboot()
A run within this job is performing a reboot (expect continue following reboot)

run_start()
A run within this job is starting

run_test_complete()
A test run by this job is complete. Note that if multiple tests are run in parallel, this will only be called when all of the parallel runs complete.

setup(*job*)

job The job object for this job

test_status(*status*, *tag*)
A test within this job is completing

test_status_detail(*code*, *subdir*, *operation*, *status*, *tag*, *optional_fields*)
A test within this job is completing (detail)

autotest.client.harness.**select**(*which*, *job*, *harness_args*)

4.15 harness_ABAT Module

4.16 harness_autoserv Module

class autotest.client.harness_autoserv.**AutoservFetcher**(*package_manager*, *job_harness*)
Bases: *autotest.client.shared.base_packages.RepositoryFetcher*

fetch_pkg_file(*filename*, *dest_path*)

class autotest.client.harness_autoserv.**harness_autoserv**(*job*, *harness_args*)
Bases: *autotest.client.harness.harness*

The server harness for running from autoserv

Properties:

job The job object for this job

fetch_package(*pkg_name*, *dest_path*)
Request a package from the remote autoserv.

Parameters

- **pkg_name** – The name of the package, as generally used by the client.shared.packages infrastructure.
- **dest_path** – The path the package should be copied to.

run_start()

run_test_complete()
A test run by this job is complete, signal it to autoserv and wait for it to signal to continue

test_status(*status*, *tag*)
A test within this job is completing

4.17 harness_beaker Module

The harness interface The interface between the client and beaker lab controller.

exception autotest.client.harness_beaker.**HarnessException**(*text*)
Bases: exceptions.Exception

autotest.client.harness_beaker.**get_beaker_code**(*at_code*)

class autotest.client.harness_beaker.**harness_beaker**(*job*, *harness_args*)
Bases: *autotest.client.harness.harness*

bootstrap(*fetchdir*)
How to kickstart autotest when you have no control file? You download the beaker XML, convert it to a control file and pass it back to autotest. Much like bootstrapping.. :-)

convert_task_to_control(*fetchdir*, *control*, *task*)
Tasks are really just: # yum install $TEST # cd /mnt/tests/$TEST # make run

Convert that into a test module with a control file

find_recipe(*recipes_dict*)

get_processed_tests()

get_recipe_from_LC()

get_test_name(*task*)

init_recipe_from_beaker()

init_task_params(*task*)

kill_watchdog()

parse_args(*args*, *is_bootstrap*)

parse_quickcmd(*args*)

run_abort()
A run within this job is aborting. It all went wrong

run_complete()
A run within this job is completing (all done)

run_pause()
A run within this job is completing (expect continue)

run_reboot()
A run within this job is performing a reboot (expect continue following reboot)

run_start()
A run within this job is starting

run_test_complete()
A test run by this job is complete. Note that if multiple tests are run in parallel, this will only be called when all of the parallel runs complete.

setupInitSymlink()

start_watchdog(*heartbeat*)

tear_down()
called from complete and abort. clean up and shutdown

test_status(*status*, *tag*)
A test within this job is completing

test_status_detail(*code*, *subdir*, *operation*, *status*, *tag*, *optional_fields*)
A test within this job is completing (detail)

upload_recipe_files()

upload_result_files(*task_id*, *resultid*, *subdir*)

upload_task_files(*task_id*, *subdir*)

watchdog_loop(*heartbeat*)

write_processed_tests(*subdir*, *t_id='0'*)

4.18 harness_simple Module

The simple harness interface

class autotest.client.harness_simple.**harness_simple**(*job*, *harness_args*)
Bases: *autotest.client.harness.harness*

The simple server harness

Properties:

job The job object for this job

test_status(*status*, *tag*)
A test within this job is completing

4.19 harness_standalone Module

The standalone harness interface

The default interface as required for the standalone reboot helper.

class autotest.client.harness_standalone.**harness_standalone**(*job*, *harness_args*)
Bases: *autotest.client.harness.harness*

The standalone server harness

Properties:

job The job object for this job

4.20 job Module

The main job wrapper

This is the core infrastructure.

Copyright Andy Whitcroft, Martin J. Bligh 2006

exception autotest.client.job.**NotAvailableError**
Bases: *autotest.client.shared.error.AutotestError*

exception autotest.client.job.**StepError**
Bases: *autotest.client.shared.error.AutotestError*

class autotest.client.job.**base_client_job**(*control*, *options*, *drop_caches=True*, *extra_copy_cmdline=None*)
Bases: *autotest.client.shared.base_job.base_job*

The client-side concrete implementation of base_job.

Optional properties provided by this implementation: control bootloader harness

add_repository(*repo_urls*)
Adds the repository locations to the job so that packages can be fetched from them when needed. The repository list needs to be a string list Ex: job.add_repository(['http://blah1','http://blah2'])

add_sysinfo_command(*command*, *logfile=None*, *on_every_test=False*)

add_sysinfo_logfile(*file*, *on_every_test=False*)

barrier(**args*, ***kwds*)
Create a barrier object

complete(*status*)
Write pending TAP reports, clean up, and exit

config_get(*name*)

config_set(*name*, *value*)

control_get()

control_set(*control*)

cpu_count()

disable_external_logging()

disable_warnings(*warning_type*)

enable_external_logging()

enable_warnings(*warning_type*)

end_reboot(*subdir*, *kernel*, *patches*, *running_id=None*)

end_reboot_and_verify(*expected_when*, *expected_id*, *subdir*, *type='src'*, *patches=[]*)
Check the passed kernel identifier against the command line and the running kernel, abort the job on missmatch.

filesystem(**args*, ***dargs*)
Same as partition

@deprecated: Use partition method instead

handle_persistent_option(*options*, *option_name*)
Select option from command line or persistent state. Store selected option to allow standalone client to continue after reboot with previously selected options. Priority: 1. explicitly specified via command line 2. stored in state file (if continuing job '-c') 3. default is None

harness_select(*which*, *harness_args*)

install_pkg(*name*, *pkg_type*, *install_dir*)
This method is a simple wrapper around the actual package installation method in the Packager class. This is used internally by the profilers, deps and tests code. name : name of the package (ex: sleeptest, dbench etc.) pkg_type : Type of the package (ex: test, dep etc.) install_dir : The directory in which the source is actually

untarred into. (ex: client/profilers/<name> for profilers)

kernel(*base_tree*, *results_dir=''*, *tmp_dir=''*, *leave=False*)
Summon a kernel object

monitor_disk_usage(*max_rate*)
Signal that the job should monitor disk space usage on / and generate a warning if a test uses up disk space at a rate exceeding 'max_rate'.

Parameters:

max_rate - the maximium allowed rate of disk consumption during a test, in MB/hour, or 0 to indicate no limit.

next_step(*fn*, **args*, ***dargs*)
Create a new step and place it after any steps added while running the current step but before any steps added in previous steps

next_step_append(*fn*, **args*, ***dargs*)
Define the next step and place it at the end

next_step_prepend(*fn*, **args*, ***dargs*)
Insert a new step, executing first

noop(*text*)

parallel(**args*, ***dargs*)
Run tasks in parallel

partition(*device*, *loop_size=0*, *mountpoint=None*)
Work with a machine partition

param device e.g. /dev/sda2, /dev/sdb1 etc...

param mountpoint Specify a directory to mount to. If not specified autotest tmp directory will be used.

param loop_size Size of loopback device (in MB). Defaults to 0.

return A L{client.partition.partition} object

quit()

reboot(*tag=<object object>*)

reboot_setup()

relative_path(*path*)
Return a patch relative to the job results directory

require_gcc()
Test whether gcc is installed on the machine.

run_group(*function*, *tag=None*, ***dargs*)
Run a function nested within a group level.

function: Callable to run.

tag: An optional tag name for the group. If None (default) function.__name__ will be used.

****dargs:** Named arguments for the function.

run_test(**args*, ***dargs*)
Summon a test object and run it.

:param url A url that identifies the test to run. :param tag An optional keyword argument that will be added to the

test and subdir name.

:param subdir_tag An optional keyword argument that will be added to the subdir name.

Returns True if the test passes, False otherwise.

run_test_detail(**args*, ***dargs*)
Summon a test object and run it, returning test status.

:param url A url that identifies the test to run. :param tag An optional keyword argument that will be added to the

test and subdir name.

:param subdir_tag An optional keyword argument that will be added to the subdir name.

Returns Test status

@see: client/shared/error.py, exit_status

setup_dep(*deps*)
Set up the dependencies for this test. deps is a list of libraries required for this test.

setup_dirs(*results_dir*, *tmp_dir*)

start_reboot()

step_engine()
The multi-run engine used when the control file defines step_init.

Does the next step.

xen(*base_tree*, *results_dir=''*, *tmp_dir=''*, *leave=False*, *kjob=None*)
Summon a xen object

class autotest.client.job.**disk_usage_monitor**(*logging_func*, *device*, *max_mb_per_hour*)

start()

stop()

classmethod watch(**monitor_args*, ***monitor_dargs*)
Generic decorator to wrap a function call with the standard create-monitor -> start -> call -> stop idiom.

class autotest.client.job.**job**(*control*, *options*, *drop_caches=True*, *extra_copy_cmdline=None*)
Bases: *autotest.client.job.base_client_job*

autotest.client.job.**runjob**(*control*, *drop_caches*, *options*)
Run a job using the given control file.

This is the main interface to this module.

@see base_job.__init__ for parameter info.

autotest.client.job.**site_job**
alias of *base_client_job*

class autotest.client.job.**status_indenter**(*job*)
Bases: *autotest.client.shared.base_job.status_indenter*

Provide a status indenter that is backed by job._record_prefix.

decrement()

increment()

indent

4.21 kernel Module

class autotest.client.kernel.**BootableKernel**(*job*)
Bases: object

add_to_bootloader(*args=''*)

autotest.client.kernel.**auto_kernel**(*job*, *path*, *subdir*, *tmp_dir*, *build_dir*, *leave=False*)
Create a kernel object, dynamically selecting the appropriate class to use based on the path provided.

class autotest.client.kernel.**kernel**(*job*, *base_tree*, *subdir*, *tmp_dir*, *build_dir*, *leave=False*)
Bases: *autotest.client.kernel.BootableKernel*

Class for compiling kernels.

Data for the object includes the src files used to create the kernel, patches applied, config (base + changes), the build directory itself, and logged output

Properties:

job Backpointer to the job object we're part of

autodir Path to the top level autotest dir (see global_config.ini, session COMMON/autotest_top_path)

src_dir <tmp_dir>/src/

build_dir <tmp_dir>/linux/

config_dir <results_dir>/config/

log_dir <results_dir>/debug/

results_dir <results_dir>/results/

apply_patches(*local_patches*)
apply the list of patches, in order

autodir = ''

boot(*args=''*, *ident=True*)
install and boot this kernel, do not care how just make it happen.

build(**args*, ***dargs*)

build_timed(*threads*, *timefile='/dev/null'*, *make_opts=''*, *output='/dev/null'*)
time the bulding of the kernel

clean(**args*, ***dargs*)

config(**args*, ***dargs*)

extract(**args*, ***dargs*)

extraversion(*tag*, *append=True*)

get_kernel_build_arch(*arch=None*)
Work out the current kernel architecture (as a kernel arch)

get_kernel_build_ident()

get_kernel_build_release()

get_kernel_build_ver()
Check Makefile and .config to return kernel version

get_kernel_tree(*base_tree*)
Extract/link base_tree to self.build_dir

get_patches(*patches*)
fetch the patches to the local src_dir

install(**args*, ***dargs*)

kernelexpand(*kernel*)

mkinitrd(**args*, ***dargs*)

patch(**args*, ***dargs*)

pickle_dump(*filename*)
dump a pickle of ourself out to the specified filename

we can't pickle the backreference to job (it contains fd's), nor would we want to. Same for logfile (fd's).

set_build_image(*image*)

set_build_target(*build_target*)

set_cross_cc(*target_arch=None*, *cross_compile=None*, *build_target='bzImage'*)
Set up to cross-compile. This is broken. We need to work out what the default compile produces, and if not, THEN set the cross compiler.

autotest.client.kernel.**preprocess_path**(*path*)

class autotest.client.kernel.**rpm_kernel**(*job*, *rpm_package*, *subdir*)
Bases: *autotest.client.kernel.BootableKernel*

Class for installing a binary rpm kernel package

boot(*args=''*, *ident=True*)
install and boot this kernel

build(**args*, ***dargs*)
Dummy function, binary kernel so nothing to build.

install(**args*, ***dargs*)

class autotest.client.kernel.**rpm_kernel_suse**(*job*, *rpm_package*, *subdir*)
Bases: *autotest.client.kernel.rpm_kernel*

Class for installing openSUSE/SLE rpm kernel package

add_to_bootloader(*tag='dummy'*, *args=''*)
Set parameters of this kernel in bootloader

install()

autotest.client.kernel.**rpm_kernel_vendor**(*job*, *rpm_package*, *subdir*)

autotest.client.kernel.**tee_output_logdir_mark**(*fn*)

4.22 kernel_config Module

autotest.client.kernel_config.**apply_overrides**(*orig_file*, *changes_file*, *output_file*)

autotest.client.kernel_config.**config_by_name**(*name*, *s*)

autotest.client.kernel_config.**diff_configs**(*old*, *new*)

autotest.client.kernel_config.**feature_enabled**(*feature*, *config*)
Verify whether a given kernel option is enabled.

Parameters

- **feature** – Kernel feature, such as "CONFIG_DEFAULT_UIMAGE".
- **config** – Config file path, such as /tmp/config.

class autotest.client.kernel_config.**kernel_config**(*job*, *build_dir*, *config_dir*, *orig_file*, *overrides*, *defconfig=False*, *name=None*, *make=None*)

Bases: object

Build directory must be ready before init'ing config.

Stages:

1. Get original config file
2. Apply overrides
3. **Do 'make oldconfig' to update it to current source code** (gets done implicitly during the process)

You may specifiy the defconfig within the tree to build, or a custom config file you want, or None, to get machine's default config file from the repo.

config_record(*name*)
Copy the current .config file to the config.<name>[.<n>]

update_config(*old_config*, *new_config=None*)

autotest.client.kernel_config.**modules_needed**(*config*)

4.23 kernel_versions Module

autotest.client.kernel_versions.**is_release_candidate**(*version*)

autotest.client.kernel_versions.**is_released_kernel**(*version*)

autotest.client.kernel_versions.**version_choose_config**(*version*, *candidates*)

autotest.client.kernel_versions.**version_encode**(*version*)

autotest.client.kernel_versions.**version_len**(*version*)

autotest.client.kernel_versions.**version_limit**(*version*, *n*)

4.24 kernelexpand Module

Program and API used to expand kernel versions, trying to match them with the URL of the correspondent package on kernel.org or a mirror. Example:

$ ./kernelexpand.py 3.1 http://www.kernel.org/pub/linux/kernel/v3.x/linux-3.1.tar.bz2

author Andy Whitcroft (apw@shadowen.org)

copyright IBM 2008

@license: GPL v2 @see: Inspired by kernelexpand by Martin J. Bligh, 2003

autotest.client.kernelexpand.**decompose_kernel**(*kernel*)

autotest.client.kernelexpand.**decompose_kernel_2x_once**(*kernel*)
Generate the parameters for the patches (2.X version):

full => full kernel name base => all but the matches suffix minor => 2.n.m major => 2.n minor-prev => 2.n.m-1

Parameters **kernel** – String representing a kernel version to be expanded.

autotest.client.kernelexpand.**decompose_kernel_post_2x_once**(*kernel*)
Generate the parameters for the patches (post 2.X version):

full => full kernel name base => all but the matches suffix minor => o.n.m major => o.n minor-prev => o.n.m-1

Parameters **kernel** – String representing a kernel version to be expanded.

autotest.client.kernelexpand.**expand_classic**(*kernel*, *mirrors*)

autotest.client.kernelexpand.**get_mappings_2x**()

autotest.client.kernelexpand.**get_mappings_post_2x**()

autotest.client.kernelexpand.**mirror_kernel_components**(*mirrors*, *components*)

autotest.client.kernelexpand.**select_kernel_components**(*components*)

autotest.client.kernelexpand.**url_accessible**(*url*)

4.25 kvm_control Module

Utilities useful to client control files that test KVM.

autotest.client.kvm_control.**get_kvm_arch**()
Get the kvm kernel module to be loaded based on the CPU architecture

Raises error.TestError if no vendor name or cpu flags are found

Returns 'kvm_amd' or 'kvm_intel' or 'kvm_power7'

Return type *string*

autotest.client.kvm_control.**load_kvm**()
Loads the appropriate KVM kernel modules depending on the current CPU architecture

Returns 0 on success or 1 on failure

Return type *int*

autotest.client.kvm_control.**unload_kvm**()
Unloads the current KVM kernel modules (if loaded)

Returns 0 on success or 1 on failure

Return type *int*

4.26 local_host Module

This file contains the implementation of a host object for the local machine.

class autotest.client.local_host.**LocalHost**(**args*, ***dargs*)
Bases: *autotest.client.shared.hosts.base_classes.Host*

list_files_glob(*path_glob*)
Get a list of files on a remote host given a glob pattern path.

run(*command*, *timeout=3600*, *ignore_status=False*, *stdout_tee=<object object>*, *stderr_tee=<object object>*, *stdin=None*, *args=()*)
@see shared.hosts.Host.run()

symlink_closure(*paths*)
Given a sequence of path strings, return the set of all paths that can be reached from the initial set by following symlinks.

Parameters **paths** – sequence of path strings.

Returns a sequence of path strings that are all the unique paths that can be reached from the given ones after following symlinks.

wait_up(*timeout=None*)

4.27 lv_utils Module

Utility for taking shapshots from existing logical volumes or creates such.

author Plamen Dimitrov

copyright Intra2net AG 2012

@license: GPL v2

param vg_name Name of the volume group.

param lv_name Name of the logical volume.

param lv_size Size of the logical volume as string in the form "#G" (for example 30G).

param lv_snapshot_name Name of the snapshot with origin the logical volume.

param lv_snapshot_size Size of the snapshot with origin the logical volume also as "#G".

param ramdisk_vg_size Size of the ramdisk virtual group.

param ramdisk_basedir Base directory for the ramdisk sparse file.

param ramdisk_sparse_filename Name of the ramdisk sparse file.

Sample ramdisk params: ramdisk_vg_size = "40000" ramdisk_basedir = "/tmp" ramdisk_sparse_filename = "virtual_hdd"

Sample general params: vg_name='autotest_vg', lv_name='autotest_lv', lv_size='1G', lv_snapshot_name='autotest_sn', lv_snapshot_size='1G'

The ramdisk volume group size is in MB.

autotest.client.lv_utils.**lv_check**(*vg_name*, *lv_name*)
Check whether provided logical volume exists.

autotest.client.lv_utils.**vg_check**(*vg_name*)
Check whether provided volume group exists.

autotest.client.lv_utils.**vg_list**()
List available volume groups.

autotest.client.lv_utils.**vg_ramdisk_cleanup**(*ramdisk_filename*, *vg_ramdisk_dir*, *vg_name*, *loop_device*)
Inline cleanup function in case of test error.

4.28 optparser Module

Autotest client/local option parser

class autotest.client.optparser.**AutotestLocalOptionParser**
Bases: optparse.OptionParser

Default autotest option parser

4.29 os_dep Module

autotest.client.os_dep.**command**(*cmd*)

autotest.client.os_dep.**commands**(**cmds*)

autotest.client.os_dep.**header**(*hdr*)

autotest.client.os_dep.**headers**(**hdrs*)

autotest.client.os_dep.**libraries**(**libs*)

autotest.client.os_dep.**library**(*lib*)

4.30 parallel Module

Parallel execution management

autotest.client.parallel.**fork_nuke_subprocess**(*tmp*, *pid*)

autotest.client.parallel.**fork_start**(*tmp*, *l*)

autotest.client.parallel.**fork_waitfor**(*tmp*, *pid*)

autotest.client.parallel.**fork_waitfor_timed**(*tmp*, *pid*, *timeout*)
Waits for pid until it terminates or timeout expires. If timeout expires, test subprocess is killed.

4.31 partition Module

APIs to write tests and control files that handle partition creation, deletion and formatting.

copyright Google 2006-2008

author Martin Bligh (mbligh@google.com)

class autotest.client.partition.**FsOptions**(*fstype*, *fs_tag*, *mkfs_flags=None*, *mount_options=None*)
Bases: object

A class encapsulating a filesystem test's parameters.

fs_tag

fstype

mkfs_flags

mount_options

autotest.client.partition.**filesystems**()
Return a list of all available filesystems

autotest.client.partition.**filter_partition_list**(*partitions*, *devnames*)
Pick and choose which partition to keep.

filter_partition_list accepts a list of partition objects and a list of strings. If a partition has the device name of the strings it is returned in a list.

Parameters

- **partitions** – A list of L{partition} objects
- **devnames** – A list of devnames of the form '/dev/hdc3' that specifies which partitions to include in the returned list.

Returns A list of L{partition} objects specified by devnames, in the order devnames specified

autotest.client.partition.**get_iosched_path**(*device_name*, *component*)

autotest.client.partition.**get_mount_info**(*partition_list*)
Picks up mount point information about the machine mounts. By default, we try to associate mount points with UUIDs, because in newer distros the partitions are uniquely identified using them.

autotest.client.partition.**get_partition_list**(*job*, *min_blocks=0*, *filter_func=None*, *exclude_swap=True*, *open_func=<built-in function open>*)
Get a list of partition objects for all disk partitions on the system.

Loopback devices and unnumbered (whole disk) devices are always excluded.

Parameters

- **job** – The job instance to pass to the partition object constructor.
- **min_blocks** – The minimum number of blocks for a partition to be considered.
- **filter_func** – A callable that returns True if a partition is desired. It will be passed one parameter: The partition name (hdc3, etc.). Some useful filter functions are already defined in this module.
- **exclude_swap** – If True any partition actively in use as a swap device will be excluded.
- **__open** – Reserved for unit testing.

Returns A list of L{partition} objects.

autotest.client.partition.**get_unmounted_partition_list**(*root_part*, *job=None*, *min_blocks=0*, *filter_func=None*, *exclude_swap=True*, *open_func=<built-in function open>*)
Return a list of partition objects that are not mounted.

Parameters

- **root_part** – The root device name (without the '/dev/' prefix, example 'hda2') that will be filtered from the partition list.

  Reasoning: in Linux /proc/mounts will never directly mention the root partition as being mounted on / instead it will say that /dev/root is mounted on /. Thus require this argument to filter out the root_part from the ones checked to be mounted.
- **min_blocks, filter_func, exclude_swap, open_func**(*job,*) – Forwarded to get_partition_list().

Returns List of L{partition} objects that are not mounted.

autotest.client.partition.**is_linux_fs_type**(*device*)
Checks if specified partition is type 83

Parameters **device** – the device, e.g. /dev/sda3

Returns False if the supplied partition name is not type 83 linux, True otherwise

autotest.client.partition.**is_valid_disk**(*device*)
Checks if a disk is valid

Parameters **device** – e.g. /dev/sda, /dev/hda

autotest.client.partition.**is_valid_partition**(*device*)
Checks if a partition is valid

Parameters **device** – e.g. /dev/sda1, /dev/hda1

autotest.client.partition.**list_mount_devices**()

autotest.client.partition.**list_mount_points**()

autotest.client.partition.**parallel**(*partitions*, *method_name*, **args*, ***dargs*)
Run a partition method (with appropriate arguments) in parallel, across a list of partition objects

class autotest.client.partition.**partition**(*job*, *device*, *loop_size=0*, *mountpoint=None*)
Bases: object

Class for handling partitions and filesystems

fsck(*args='-fy'*, *record=True*)
Run filesystem check

Parameters **args** – arguments to filesystem check tool. Default is "-n" which works on most tools.

get_fsck_exec()
Return the proper mkfs executable based on self.fstype

get_io_scheduler(*device_name*)

get_io_scheduler_list(*device_name*)

get_mountpoint(*open_func=<built-in function open>*, *filename=None*)
Find the mount point of this partition object.

Parameters

- **open_func** – the function to use for opening the file containing the mounted partitions information
- **filename** – where to look for the mounted partitions information (default None which means it will search /proc/mounts and/or /etc/mtab)

Returns a string with the mount point of the partition or None if not mounted

mkfs(*fstype=None*, *args=''*, *record=True*)
Format a partition to filesystem type

Parameters

- **fstype** – the filesystem type, e.g.. "ext3", "ext2"
- **args** – arguments to be passed to mkfs command.
- **record** – if set, output result of mkfs operation to autotest output

mkfs_exec(*fstype*)
Return the proper mkfs executable based on fs

mount(*mountpoint=None*, *fstype=None*, *args=''*, *record=True*)
Mount this partition to a mount point

Parameters

- **mountpoint** – If you have not provided a mountpoint to partition object or want to use a different one, you may specify it here.
- **fstype** – Filesystem type. If not provided partition object value will be used.
- **args** – Arguments to be passed to "mount" command.
- **record** – If True, output result of mount operation to autotest output.

run_test(*test*, ***dargs*)

run_test_on_partition(*test*, *mountpoint_func*, ***dargs*)
Executes a test fs-style (umount,mkfs,mount,test)

Here we unmarshal the args to set up tags before running the test. Tests are also run by first umounting, mkfsing and then mounting before executing the test.

Parameters

- **test** – name of test to run
- **mountpoint_func** – function to return mount point string

set_fs_options(*fs_options*)
Set filesystem options

param fs_options A L{FsOptions} object

set_io_scheduler(*device_name*, *name*)

setup_before_test(*mountpoint_func*)
Prepare a partition for running a test. Unmounts any filesystem that's currently mounted on the partition, makes a new filesystem (according to this partition's filesystem options) and mounts it where directed by mountpoint_func.

Parameters **mountpoint_func** – A callable that returns a path as a string, given a partition instance.

unmount(*ignore_status=False*, *record=True*)
Umount this partition.

It's easier said than done to umount a partition. We need to lock the mtab file to make sure we don't have any locking problems if we are umounting in paralllel.

If there turns out to be a problem with the simple umount we end up calling umount_force to get more aggressive.

Parameters

- **ignore_status** – should we notice the umount status
- **record** – if True, output result of umount operation to autotest output

unmount_force()
Kill all other jobs accessing this partition. Use fuser and ps to find all mounts on this mountpoint and unmount them.

Returns true for success or false for any errors

wipe()
Delete all files of a given partition filesystem.

autotest.client.partition.**partname_to_device**(*part*)
Converts a partition name to its associated device

autotest.client.partition.**run_test_on_partitions**(*job*, *test*, *partitions*, *mountpoint_func*, *tag*, *fs_opt*, *do_fsck=True*, ***dargs*)
Run a test that requires multiple partitions. Filesystems will be made on the partitions and mounted, then the test will run, then the filesystems will be unmounted and optionally fsck'd.

Parameters

- **job** – A job instance to run the test
- **test** – A string containing the name of the test
- **partitions** – A list of partition objects, these are passed to the test as partitions=
- **mountpoint_func** – A callable that returns a mountpoint given a partition instance
- **tag** – A string tag to make this test unique (Required for control files that make multiple calls to this routine with the same value of 'test'.)
- **fs_opt** – An FsOptions instance that describes what filesystem to make
- **do_fsck** – include fsck in post-test partition cleanup.
- **dargs** – Dictionary of arguments to be passed to job.run_test() and eventually the test

autotest.client.partition.**unmount_partition**(*device*)
Unmount a mounted partition

Parameters **device** – e.g. /dev/sda1, /dev/hda1

class autotest.client.partition.**virtual_partition**(*file_img*, *file_size*)
Handles block device emulation using file images of disks. It's important to note that this API can be used only if we have the following programs present on the client machine:

•sfdisk

•losetup

•kpartx

destroy()
Removes the virtual partition from /dev/mapper, detaches the image file from the loopback device and removes the image file.

autotest.client.partition.**wipe_filesystem**(*job*, *mountpoint*)

4.32 profiler Module

class autotest.client.profiler.**profiler**(*job*)

initialize(**args*, ***dargs*)

preserve_srcdir = False

report(*test*)

setup(**args*, ***dargs*)

start(*test*)

stop(*test*)

supports_reboot = False

4.33 setup Module

autotest.client.setup.**get_data_files**()

autotest.client.setup.**get_filelist**()

autotest.client.setup.**get_package_data**()

autotest.client.setup.**get_package_dir**()

autotest.client.setup.**get_packages**()

autotest.client.setup.**get_scripts**()

autotest.client.setup.**run**()

4.34 setup_job Module

autotest.client.setup_job.**init_test**(*options*, *testdir*)
Instantiate a client test object from a given test directory.

:param options Command line options passed in to instantiate a setup_job which associates with this test.

:param testdir The test directory. :return: A test object or None if failed to instantiate.

autotest.client.setup_job.**load_all_client_tests**(*options*)
Load and instantiate all client tests.

This function is inspired from runtest() on client/shared/test.py.

Parameters **options** – an object passed in from command line OptionParser. See all options defined on client/autotest.

Returns a tuple containing the list of all instantiated tests and a list of tests that failed to instantiate.

class autotest.client.setup_job.**setup_job**(*options*)
Bases: *autotest.client.job.job*

setup_job is a job which runs client test setup() method at server side.

This job is used to pre-setup client tests when development toolchain is not available at client.

autotest.client.setup_job.**setup_test**(*client_test*)
Direct invoke test.setup() method.

Returns A boolean to represent success or not.

autotest.client.setup_job.**setup_tests**(*options*)
Load and instantiate all client tests.

This function is inspired from runtest() on client/shared/test.py.

Parameters **options** – an object passed in from command line OptionParser. See all options defined on client/autotest.

4.35 setup_modules Module

Module used to create the autotest namespace for single dir use case.

Autotest programs can be used and developed without requiring it to be installed system-wide. In order for the code to see the library namespace:

from autotest.client.shared import error from autotest.server import hosts ...

Without system wide install, we need some hacks, that are performed here.

author John Admanski (jadmanski@google.com)

autotest.client.setup_modules.**import_module**(*module*, *from_where*)
Equivalent to 'from from_where import module'.

Parameters

- **module** – Module name.
- **from_where** – Package from where the module is being imported.

Returns The corresponding module.

autotest.client.setup_modules.**setup**(*base_path*, *root_module_name='autotest'*)
Setup a library namespace, with the appropriate top root module name.

Perform all the necessary setup so that all the packages at 'base_path' can be imported via "import root_module_name.package".

Parameters

- **base_path** – Base path for the module.
- **root_module_name** – Top level name for the module.

4.36 sysinfo Module

4.37 test Module

autotest.client.test.**runtest**(*job*, *url*, *tag*, *args*, *dargs*)

class autotest.client.test.**test**(*job*, *bindir*, *outputdir*)
Bases: *autotest.client.shared.test.base_test*

configure_crash_handler()

Configure the crash handler by:

- Setting up core size to unlimited
- Putting an appropriate crash handler on /proc/sys/kernel/core_pattern
- Create files that the crash handler will use to figure which tests are active at a given moment

The crash handler will pick up the core file and write it to self.debugdir, and perform analysis on it to generate a report. The program also outputs some results to syslog.

If multiple tests are running, an attempt to verify if we still have the old PID on the system process table to determine whether it is a parent of the current test execution. If we can't determine it, the core file and the report file will be copied to all test debug dirs.

crash_handler_report()
If core dumps are found on the debugdir after the execution of the test, let the user know.

4.38 test_config Module

Wrapper around ConfigParser to manage testcases configuration.

author rsalveti@linux.vnet.ibm.com (Ricardo Salveti de Araujo)

class autotest.client.test_config.**config_loader**(*cfg*, *tmpdir='/tmp'*, *raise_errors=False*)
Base class of the configuration parser

check(*section*)
Check if the config file has valid values

check_parameter(*param_type*, *parameter*)
Check if a option has a valid value

get(*section*, *option*, *default=None*)
Get the value of a option.

Section of the config file and the option name. You can pass a default value if the option doesn't exist.

Parameters

- **section** – Configuration file section.
- **option** – Option we're looking after.

@default: In case the option is not available and raise_errors is set to False, return the default.

remove(*section*, *option*)
Remove an option.

save()
Save the configuration file with all modifications

set(*section*, *option*, *value*)
Set an option.

This change is not persistent unless saved with 'save()'.

4.39 utils Module

Convenience functions for use by tests or whomever.

NOTE: this is a mixin library that pulls in functions from several places Note carefully what the precendece order is

There's no really good way to do this, as this isn't a class we can do inheritance with, just a collection of static methods.

4.40 xen Module

class autotest.client.xen.**xen**(*job*, *base_tree*, *results_dir*, *tmp_dir*, *build_dir*, *leave=False*, *kjob=None*)
Bases: *autotest.client.kernel.kernel*

add_to_bootloader(*tag='autotest'*, *args=''*)
add this kernel to bootloader, taking an optional parameter of space separated parameters e.g.: kernel.add_to_bootloader('mykernel', 'ro acpi=off')

build(*make_opts=''*, *logfile=''*, *extraversion='autotest'*)
build xen

make_opts additional options to make, if any

build_timed(**args*, ***kwds*)

config(*config_file*, *config_list=None*)

fix_up_xen_kernel_makefile(*kernel_dir*)
Fix up broken EXTRAVERSION in xen-ified Linux kernel Makefile

get_xen_build_ver()
Check Makefile and .config to return kernel version

get_xen_kernel_build_ver()
Check xen buildconfig for current kernel version

install(*tag=''*, *prefix='/'*, *extraversion='autotest'*)
make install in the kernel tree

log(*msg*)

4.41 Subpackages

4.41.1 net Package

basic_machine Module

common Module

net_tc Module

Convenience methods for use to manipulate traffic control settings.

see http://linux.die.net/man/8/tc for details about traffic controls in linux.

Example

try: import autotest.common as common

except ImportError:

import common

from autotest.client.net.net_tc import * from autotest.client.net.net_utils import *

class mock_netif(object):

def __init__(self, name): self._name = name

def get_name(self): return self._name

netem_qdisc = netem() netem_qdisc.add_param('loss 100%')

ack_filter = u32filter() ack_filter.add_rule('match ip protocol 6 0xff') ack_filter.add_rule('match u8 0x10 0x10 at nexthdr+13') ack_filter.set_dest_qdisc(netem_qdisc)

root_qdisc = prio() root_qdisc.get_class(2).set_leaf_qdisc(netem_qdisc) root_qdisc.add_filter(ack_filter)

lo_if = mock_netif('lo')

root_qdisc.setup(lo_if)

run test here ... root_qdisc.restore(lo_if)

class autotest.client.net.net_tc.**classful_qdisc**(*handle*)
Bases: *autotest.client.net.net_tc.qdisc*

add_class(*child_class*)

add_filter(*filter*)

classful = True

restore(*netif*)

setup(*netif*)

class autotest.client.net.net_tc.**classless_qdisc**(*handle*)
Bases: *autotest.client.net.net_tc.qdisc*

classful = False

class autotest.client.net.net_tc.**netem**(*handle=300*)
Bases: *autotest.client.net.net_tc.classless_qdisc*

add_param(*param*)

name = 'netem'

setup(*netif*)

autotest.client.net.net_tc.**new_handle**()

class autotest.client.net.net_tc.**pfifo**(*handle=200*)
Bases: *autotest.client.net.net_tc.classless_qdisc*

name = 'pfifo'

setup(*netif*)

class autotest.client.net.net_tc.**prio**(*handle=100*, *bands=3*)
Bases: *autotest.client.net.net_tc.classful_qdisc*

get_class(*band*)

name = 'prio'

setup(*netif*)

class autotest.client.net.net_tc.**qdisc**(*handle*)
Bases: object

get_handle()

get_parent_class()

id()

restore(*netif*)

set_parent_class(*parent_class*)

setup(*netif*)

tc_cmd(*tc_conf*)

class autotest.client.net.net_tc.**tcclass**(*handle*, *minor*, *leaf_qdisc=None*)
Bases: object

add_child(*child_class*)

get_leaf_qdisc()

get_minor()

get_parent_class()

id()

restore(*netif*)

set_leaf_qdisc(*leaf_qdisc*)

set_parent_class(*parent_class*)

setup(*netif*)

class autotest.client.net.net_tc.**tcfilter**
Bases: object

conf_command = 'cmd'

conf_device = 'dev'

conf_flowid = 'flowid'

conf_name = 'name'

conf_params = 'params'

conf_parent = 'parent'

conf_priority = 'priority'

conf_protocol = 'protocol'

conf_qdiscid = 'qdiscid'

conf_rules = 'cmd'

conf_type = 'filtertype'

get_dest_qdisc()

get_handle()

get_parent_qdisc()

get_priority()

get_protocol()

restore(*netif*)

set_dest_qdisc(*dest_qdisc*)

set_handle(*handle*)

set_parent_qdisc(*parent_qdisc*)

set_priority(*priority*)

set_protocol(*protocol*)

setup(*netif*)

tc_cmd(*tc_conf*)

class autotest.client.net.net_tc.**u32filter**
Bases: *autotest.client.net.net_tc.tcfilter*

add_rule(*rule*)

filtertype = 'u32'

restore(*netif*)

setup(*netif*)

net_utils Module

Convenience functions for use by network tests or whomever.

This library is to release in the public repository.

autotest.client.net.net_utils.**bond**()

class autotest.client.net.net_utils.**bonding**
Bases: object

This class implements bonding interface abstraction.

AB_MODE = 1

AD_MODE = 2

NO_MODE = 0

disable()

enable()

get_active_interfaces()

get_mii_status()

get_mode()

get_slave_interfaces()

is_bondable()

is_enabled()

wait_for_state_change()
Wait for bonding state change.

Wait up to 90 seconds to successfully ping the gateway. This is to know when LACP state change has converged. (0 seconds is 3x lacp timeout, use by protocol)

class autotest.client.net.net_utils.**ethernet**
Bases: object

Provide ethernet packet manipulation methods.

CHECKSUM_LEN = 4

ETH_LLDP_DST_MAC = '01:80:C2:00:00:0E'

ETH_PACKET_MAX_SIZE = 1518

ETH_PACKET_MIN_SIZE = 64

ETH_TYPE_8021Q = 33024

ETH_TYPE_ARP = 2054

ETH_TYPE_CDP = 8192

ETH_TYPE_IP = 2048

ETH_TYPE_IP6 = 34525

ETH_TYPE_LLDP = 35020

ETH_TYPE_LOOPBACK = 36864

FRAME_KEY_DST_MAC = 'dst'

FRAME_KEY_PAYLOAD = 'payload'

FRAME_KEY_PROTO = 'proto'

FRAME_KEY_SRC_MAC = 'src'

HDR_LEN = 14

static **mac_binary_to_string**(*hwaddr*)
Converts a MAC address byte string to text string.

Converts a MAC byte string 'xxxxxxxxxxxx' to a text string 'aa:aa:aa:aa:aa:aa'

Args: hwaddr: a byte string containing the MAC address to convert.

Returns: A text string.

static **mac_string_to_binary**(*hwaddr*)
Converts a MAC address text string to byte string.

Converts a MAC text string from a text string 'aa:aa:aa:aa:aa:aa' to a byte string 'xxxxxxxxxxxx'

Args: hwaddr: a text string containing the MAC address to convert.

Returns: A byte string.

static **pack**(*dst*, *src*, *protocol*, *payload*)
Pack a frame in a byte string.

Args: dst: destination mac in byte string format src: src mac address in byte string format protocol: short in network byte order payload: byte string payload data

Returns: An ethernet frame with header and payload in a byte string.

static **unpack**(*raw_frame*)
Unpack a raw ethernet frame.

Returns:

None on error

{ 'dst' [byte string,] 'src' : byte string, 'proto' : short in host byte order, 'payload' : byte string

}

autotest.client.net.net_utils.**ethernet_packet**()

autotest.client.net.net_utils.**netif**(*name*)

autotest.client.net.net_utils.**network**()

class autotest.client.net.net_utils.**network_interface**(*name*)
Bases: object

DISABLE = False

ENABLE = True

add_maddr(*maddr*)

del_maddr(*maddr*)

disable_loopback()

disable_promisc()

down()

enable_loopback()

enable_promisc()

flush()

get_advertised_link_modes()

get_carrier()

get_driver()

get_hwaddr()

get_ipaddr()

get_name()

get_speed()

get_stats()

get_stats_diff(*orig_stats*)

get_supported_link_modes()

get_wakeon()

is_autoneg_advertised()

is_autoneg_on()

is_down()

is_full_duplex()

is_loopback_enabled()

is_pause_autoneg_on()

is_rx_pause_on()

is_rx_summing_on()

is_scatter_gather_on()

is_tso_on()

is_tx_pause_on()

is_tx_summing_on()

parse_ethtool(*field*, *match*, *option=''*, *next_field=''*)

recv(*len*)

restore()

send(*buf*)

set_hwaddr(*hwaddr*)

set_ipaddr(*ipaddr*)

up()

wait_for_carrier(*timeout=60*)

class autotest.client.net.net_utils.**network_utils**
Bases: object

disable_ip_local_loopback(*ignore_status=False*)

enable_ip_local_loopback(*ignore_status=False*)

get_ip_local(*query_ip*, *netmask='24'*)
Get ip address in local system which can communicate with query_ip.

Parameters **query_ip** – IP of client which wants to communicate with autotest machine.

Returns IP address which can communicate with query_ip

list()

process_mpstat(*mpstat_out*, *sample_count*, *loud=True*)
Parses mpstat output of the following two forms: 02:10:17 0 0.00 0.00 0.00 0.00 0.00 0.00 0.00 100.00 1012.87 02:10:13 PM 0 0.00 0.00 0.00 0.00 0.00 0.00 0.00 100.00 1019.00

reset(*ignore_status=False*)

start(*ignore_status=False*)

stop(*ignore_status=False*)

class autotest.client.net.net_utils.**raw_socket**(*iface_name*)
Bases: object

This class implements an raw socket abstraction.

ETH_P_ALL = 3

SOCKET_TIMEOUT = 1

close()
Close the raw socket

open(*protocol=None*)
Opens the raw socket to send and receive.

Args: protocol : short in host byte order. None if ALL

recv(*timeout*)
Synchroneous receive.

Receives one packet from the interface and returns its content in a string. Wait up to timeout for the packet if timeout is not 0. This function filters out all the packets that are less than the minimum ethernet packet size (60+crc).

Args:

timeout: max time in seconds to wait for the read to complete. '0', wait for ever until a valid packet is received

Returns:

packet: None no packet was received a binary string containing the received packet.

time_left: amount of time left in timeout

recv_from(*dst_mac*, *src_mac*, *protocol*)
Receive an ethernet frame that matches the dst, src and proto.

Filters all received packet to find a matching one, then unpack it and present it to the caller as a frame.

Waits up to self._socket_timeout for a matching frame before returning.

Args: dst_mac: 'byte string'. None do not use in filter. src_mac: 'byte string'. None do not use in filter. protocol: short in host byte order. None do not use in filter.

Returns:

ethernet frame: { 'dst' [byte string,]

'src' : byte string, 'proto' : short in host byte order, 'payload' : byte string

}

send(*packet*)
Send an ethernet packet.

send_to(*dst_mac*, *src_mac*, *protocol*, *payload*)
Send an ethernet frame.

Send an ethernet frame, formating the header.

Args: dst_mac: 'byte string' src_mac: 'byte string' protocol: short in host byte order payload: 'byte string'

set_socket_timeout(*timeout*)
Set the timeout use by recv_from.

Args: timeout: time in seconds

socket()

socket_timeout()
Get the timeout use by recv_from

net_utils_mock Module

Set of Mocks and stubs for network utilities unit tests.

Implement a set of mocks and stubs use to implement unit tests for the network libraries.

class autotest.client.net.net_utils_mock.**netif_stub**(*iface*, *cls*, *name*, **args*, ***kwargs*)
Bases: *autotest.client.shared.test_utils.mock.mock_class*

wait_for_carrier(*timeout*)

autotest.client.net.net_utils_mock.**netutils_netif**(*iface*)

class autotest.client.net.net_utils_mock.**network_interface_mock**(*iface='some_name'*, *test_init=False*)
Bases: *autotest.client.net.net_utils.network_interface*

get_driver()

get_ipaddr()

is_down()

is_loopback_enabled()

wait_for_carrier(*timeout=1*)

autotest.client.net.net_utils_mock.**os_open**(**args*, ***kwarg*)

class autotest.client.net.net_utils_mock.**os_stub**(*symbol*, ***kwargs*)
Bases: *autotest.client.shared.test_utils.mock.mock_function*

open(**args*, ***kwargs*)

read(**args*, ***kwargs*)

readval = ''

class autotest.client.net.net_utils_mock.**socket_stub**(*iface*, *cls*, *name*, **args*, ***kwargs*)
Bases: *autotest.client.shared.test_utils.mock.mock_class*

Class use to mock sockets.

bind(*arg*)

close()

recv(*size*)

send(*buf*)

settimeout(*timeout*)

socket(*family*, *type*)

4.41.2 profilers Package

profilers Package

class autotest.client.profilers.**profilers**(*job*)
Bases: *autotest.client.shared.profiler_manager.profiler_manager*

load_profiler(*profiler*, *args*, *dargs*)

Subpackages

blktrace Package

blktrace Module Autotest profiler for blktrace blktrace - generate traces of the i/o traffic on block devices

class autotest.client.profilers.blktrace.blktrace.**blktrace**(*job*)
Bases: *autotest.client.profiler.profiler*

get_device(*test*)

initialize(***dargs*)

report(*test*)

setup(*tarball='blktrace.tar.bz2'*, ***dargs*)

start(*test*)

stop(*test*)

version = 2

catprofile Package

catprofile Module Sets up a subprocses to cat a file on a specified interval

Defaults options: job.profilers.add('catprofile', ['/proc/meminfo','/proc/uptime'],

outfile=monitor, interval=1)

class autotest.client.profilers.catprofile.catprofile.**catprofile**(*job*)
Bases: *autotest.client.profiler.profiler*

initialize(*filenames=['/proc/meminfo', '/proc/slabinfo'], outfile='monitor', interval=1, **dargs*)

report(*test*)

start(*test*)

stop(*test*)

version = 1

cmdprofile Package

cmdprofile Module Sets up a subprocess to run any generic command in the background every few seconds (by default the interval is 60 secs)

class autotest.client.profilers.cmdprofile.cmdprofile.**cmdprofile**(*job*)
Bases: *autotest.client.profiler.profiler*

initialize(*cmds=['ps'], interval=60, outputfile='cmdprofile', outputfiles=None, **dargs*)

start(*test*)

stop(*test*)

supports_reboot = True

version = 2

cpistat Package

cpistat Module Uses perf_events to count cycles and instructions

Defaults options: job.profilers.add('cpistat', interval=1)

class autotest.client.profilers.cpistat.cpistat.**cpistat**(*job*)
Bases: *autotest.client.profiler.profiler*

initialize(*interval=1*, ***dargs*)

start(*test*)

stop(*test*)

version = 1

ftrace Package

ftrace Module Function tracer profiler for autotest.

author David Sharp (dhsharp@google.com)

class autotest.client.profilers.ftrace.ftrace.**ftrace**(*job*)
Bases: *autotest.client.profiler.profiler*

ftrace profiler for autotest. It builds ftrace from souce and runs trace-cmd with configurable parameters.

@see: git://git.kernel.org/pub/scm/linux/kernel/git/rostedt/trace-cmd.git

initialize(*tracepoints*, *buffer_size_kb=1408*, ***kwargs*)
Initialize ftrace profiler.

Parameters

- **tracepoints** – List containing a mix of tracpoint names and (tracepoint name, filter) tuples. Tracepoint names are as accepted by trace-cmd -e, eg "syscalls", or "syscalls:sys_enter_read". Filters are as accepted by trace-cmd -f, eg "((sig >= 10 && sig < 15) || sig == 17)"
- **buffer_size_kb** – Set the size of the ring buffer (per cpu).

static join_command(*cmd*)
Shell escape the command for BgJob. grmbl.

Parameters **cmd** – Command list.

mountpoint = '/sys/kernel/debug'

setup(*tarball='trace-cmd.tar.bz2'*, ***kwargs*)
Build and install trace-cmd from source.

The tarball was obtained by checking the git repo at 09-14-2010, removing the Documentation and the .git folders, and compressing it.

Parameters

- **tarball** – Path to trace-cmd tarball.
- ****kwargs** – Dictionary with additional parameters.

start(*test*)
Start ftrace profiler

Parameters **test** – Autotest test in which the profiler will operate on.

stop(*test*)
Stop ftrace profiler.

Parameters **test** – Autotest test in which the profiler will operate on.

tracing_dir = '/sys/kernel/debug/tracing'

version = 1

inotify Package

inotify Module inotify logs filesystem activity that may be directly or indirectly caused by the test that is running. It requires the inotify-tools package, more specifically, the inotifywait tool.

Heavily inspired / shamelessly copied from the kvm_stat profiler.

copyright Red Hat 2013

author Cleber Rosa <cleber@redhat.com>

class autotest.client.profilers.inotify.inotify.**inotify**(*job*)
Bases: *autotest.client.profiler.profiler*

Profiler based on inotifywait from inotify-tools

initialize(*paths=[]*)

report(*test*)

start(*test*)

stop(*test*)

version = 1

iostat Package

iostat Module Run iostat with a default interval of 1 second.

class autotest.client.profilers.iostat.iostat.**iostat**(*job*)
Bases: *autotest.client.profiler.profiler*

initialize(*interval=1*, *options=''*, ***dargs*)

report(*test*)

start(*test*)

stop(*test*)

version = 2

kvm_stat Package

kvm_stat Module kvm_stat prints statistics generated by the kvm module. It depends on debugfs. If no debugfs is mounted, the profiler will try to mount it so it's possible to proceed.

copyright Red Hat 2010

author Lucas Meneghel Rodrigues (lmr@redhat.com)

class autotest.client.profilers.kvm_stat.kvm_stat.**kvm_stat**(*job*)
Bases: *autotest.client.profiler.profiler*

kvm_stat based profiler. Consists on executing kvm_stat -l during a given test execution, redirecting its output to a file on the profile dir.

initialize(***dargs*)
Gets path of kvm_stat and verifies if debugfs needs to be mounted.

report(*test*)
Report function. Does nothing as there's no postprocesing needed.

Parameters **test** – Autotest test on which this profiler will operate on.

start(*test*)
Starts kvm_stat subprocess.

Parameters **test** – Autotest test on which this profiler will operate on.

stop(*test*)
Stops profiler execution by sending a SIGTERM to kvm_stat process.

Parameters **test** – Autotest test on which this profiler will operate on.

version = 1

lockmeter Package

lockmeter Module Lockstat is the basic tool used to control the kernel's Lockmeter functionality: e.g., turning the kernel's data gathering on or off, and retrieving that data from the kernel so that Lockstat can massage it and produce printed reports. See http://oss.sgi.com/projects/lockmeter for details.

NOTE: if you get compile errors from config.h, referring you to a FAQ, you might need to do 'cat < /dev/null > /usr/include/linux/config.h'. But read the FAQ first.

class autotest.client.profilers.lockmeter.lockmeter.**lockmeter**(*job*)
Bases: *autotest.client.profiler.profiler*

initialize(***dargs*)

report(*test*)

setup(*tarball='lockstat-1.4.11.tar.bz2'*)

start(*test*)

stop(*test*)

version = 1

lttng Package

lttng Module Trace kernel events with Linux Tracing Toolkit (lttng). You need to install the lttng patched kernel in order to use the profiler.

Examples: job.profilers.add('lttng', tracepoints = None): enable all trace points. job.profilers.add('lttng', tracepoints = []): disable all trace points. job.profilers.add('lttng', tracepoints = ['kernel_arch_syscall_entry',

'kernel_arch_syscall_exit'])

will only trace syscall events.

Take a look at /proc/ltt for the list of the tracing events currently supported by lttng and their output formats.

To view the collected traces, copy results/your-test/profiler/lttng to a machine that has Linux Tracing Toolkit Viewer (lttv) installed:

test$ scp -r results/your-test/profiler/lttng user@localmachine:/home/tmp/

Then you can examine the traces either in text mode or in GUI: localmachine$ lttv -m textDump -t /home/tmp/lttng

or localmachine$ lttv-gui -t /home/tmp/lttng &

class autotest.client.profilers.lttng.lttng.**lttng**(*job*)
Bases: *autotest.client.profiler.profiler*

initialize(*outputsize=1048576*, *tracepoints=None*, ***dargs*)

setup(*tarball='ltt-control-0.51-12082008.tar.gz'*, ***dargs*)

start(*test*)

stop(*test*)

version = 1

mpstat Package

mpstat Module Sets up a subprocess to run mpstat on a specified interval, default 1 second

class autotest.client.profilers.mpstat.mpstat.**mpstat**(*job*)
Bases: *autotest.client.profiler.profiler*

initialize(*interval=1*, ***dargs*)

report(*test*)

start(*test*)

stop(*test*)

version = 1

oprofile Package

oprofile Module OProfile is a system-wide profiler for Linux systems, capable of profiling all running code at low overhead. OProfile is released under the GNU GPL.

It consists of a kernel driver and a daemon for collecting sample data, and several post-profiling tools for turning data into information.

More Info: http://oprofile.sourceforge.net/ Will need some libaries to compile. Do 'apt-get build-dep oprofile'

class autotest.client.profilers.oprofile.oprofile.**oprofile**(*job*)
Bases: *autotest.client.profiler.profiler*

initialize(*vmlinux=None*, *events=[]*, *others=None*, *local=None*, ***dargs*)

report(*test*)

setup(*tarball='oprofile-0.9.4.tar.bz2'*, *local=None*, **args*, ***dargs*)

setup_done = False

start(*test*)

stop(*test*)

version = 7

perf Package

perf Module perf is a tool included in the linux kernel tree that supports functionality similar to oprofile and more.

@see: http://lwn.net/Articles/310260/

class autotest.client.profilers.perf.perf.**perf**(*job*)
Bases: *autotest.client.profiler.profiler*

initialize(*events=['cycles', 'instructions'], trace=False, **dargs*)

report(*test*)

start(*test*)

stop(*test*)

version = 1

powertop Package

powertop Module What's eating the battery life of my laptop? Why isn't it many more hours? Which software component causes the most power to be burned? These are important questions without a good answer... until now.

class autotest.client.profilers.powertop.powertop.**powertop**(*job*)
Bases: *autotest.client.profiler.profiler*

preserve_srcdir = True

report(*test*)

setup(**args*, ***dargs*)

start(*test*)

stop(*test*)

version = 1

readprofile Package

readprofile Module readprofile - a tool to read kernel profiling information

The readprofile command uses the /proc/profile information to print ascii data on standard output. The output is organized in three columns: the first is the number of clock ticks, the second is the name of the C function in the kernel where those many ticks occurred, and the third is the normalized 'load' of the procedure, calculated as a ratio between the number of ticks and the length of the procedure. The output is filled with blanks to ease readability.

class autotest.client.profilers.readprofile.readprofile.**readprofile**(*job*)
Bases: *autotest.client.profiler.profiler*

initialize(***dargs*)

report(*test*)

setup(*tarball='util-linux-2.12r.tar.bz2'*)

start(*test*)

stop(*test*)

version = 1

sar Package

sar Module Sets up a subprocess to run sar from the sysstat suite

Default options: sar -A -f

class autotest.client.profilers.sar.sar.**sar**(*job*)
Bases: *autotest.client.profiler.profiler*

The sar command writes to standard output the contents of selected cumulative activity counters in the operating system. This profiler executes sar and redirects its output in a file located in the profiler results dir.

initialize(*interval=1*, ***dargs*)
Set sar interval and verify what flags the installed sar supports.

Parameters **interval** – Interval used by sar to produce system data.

report (*test*)
Report function. Convert the binary sar data to text.

Parameters **test** – Autotest test on which this profiler will operate on.

start (*test*)
Starts sar subprocess.

Parameters **test** – Autotest test on which this profiler will operate on.

stop (*test*)
Stops profiler execution by sending a SIGTERM to sar process.

Parameters **test** – Autotest test on which this profiler will operate on.

version = 1

systemtap Package

systemtap Module Autotest systemtap profiler.

class autotest.client.profilers.systemtap.systemtap.**systemtap** (*job*)
Bases: *autotest.client.profiler.profiler*

Tracing test process using systemtap tools.

initialize (***dargs*)

report (*test*)

start (*test*)

stop (*test*)

version = 1

vmstat Package

vmstat Module Runs vmstat X where X is the interval in seconds

Defaults options: job.profilers.add('vmstat', interval=1)

class autotest.client.profilers.vmstat.vmstat.**vmstat** (*job*)
Bases: *autotest.client.profiler.profiler*

initialize (*interval=1*, ***dargs*)

report (*test*)

start (*test*)

stop (*test*)

version = 1

4.41.3 shared Package

autotemp Module

Autotest tempfile wrapper for mkstemp (known as tempfile here) and mkdtemp (known as tempdir).

This wrapper provides a mechanism to clean up temporary files/dirs once they are no longer need.

Files/Dirs will have a unique_id prepended to the suffix and a _autotmp_ tag appended to the prefix.

It is required that the unique_id param is supplied when a temp dir/file is created.

class autotest.client.shared.autotemp.**tempdir**(*suffix=''*, *unique_id=None*, *prefix=''*, *dir=None*)

Bases: object

A wrapper for tempfile.mkdtemp

@var name: The name of the temporary dir. :return: A tempdir object example usage:

b = autotemp.tempdir(unique_id='exemdir') b.name # your directory b.clean() # clean up after yourself

clean()

Remove the temporary dir that was created. This is also called by the destructor.

class autotest.client.shared.autotemp.**tempfile**(*unique_id*, *suffix=''*, *prefix=''*, *dir=None*, *text=False*)

Bases: object

A wrapper for tempfile.mkstemp

Parameters **unique_id** – required, a unique string to help identify what part of code created the tempfile.

@var name: The name of the temporary file. @var fd: the file descriptor of the temporary file that was created. :return: a tempfile object example usage:

t = autotemp.tempfile(unique_id='fig') t.name # name of file t.fd # file descriptor t.fo # file object t.clean() # clean up after yourself

clean()

Remove the temporary file that was created. This is also called by the destructor.

barrier Module

base_barrier Module

exception autotest.client.shared.base_barrier.**BarrierAbortError**

Bases: *autotest.client.shared.error.BarrierError*

Special BarrierError raised when an explicit abort is requested.

class autotest.client.shared.base_barrier.**barrier**(*hostid*, *tag*, *timeout=None*, *port=None*, *listen_server=None*)

Bases: object

Multi-machine barrier support.

Provides multi-machine barrier mechanism. Execution stops until all members arrive at the barrier.

When a barrier is forming the master node (first in sort order) in the set accepts connections from each member of the set. As they arrive they indicate the barrier they are joining and their identifier (their hostname or IP address and optional tag). They are then asked to wait. When all members are present the master node then checks that each member is still responding via a ping/pong exchange. If this is successful then everyone has checked in at the barrier. We then tell everyone they may continue via a rlse message.

Where the master is not the first to reach the barrier the client connects will fail. Client will retry until they either succeed in connecting to master or the overall timeout is exceeded.

As an example here is the exchange for a three node barrier called 'TAG'

MASTER CLIENT1 CLIENT2 <————-TAG C1————- ————wait————>

[...]

<————-TAG C2———————— ————wait———————>

[...]

————ping————> <————pong———— ————ping————
——> <————-pong————————-

—— BARRIER conditions MET ——

————rlse————> ————rlse————————>

Note that once the last client has responded to pong the barrier is implicitly deemed satisifed, they have all acknowledged their presence. If we fail to send any of the rlse messages the barrier is still a success, the failed host has effectively broken 'right at the beginning' of the post barrier execution window.

In addition, there is another rendezvous, that makes each slave a server and the master a client. The connection process and usage is still the same but allows barriers from machines that only have a one-way connection initiation. This is called rendezvous_servers.

For example:

if ME == SERVER: server start

b = job.barrier(ME, 'server-up', 120) b.rendezvous(CLIENT, SERVER)

if ME == CLIENT: client run

b = job.barrier(ME, 'test-complete', 3600) b.rendezvous(CLIENT, SERVER)

if ME == SERVER: server stop

Any client can also request an abort of the job by setting abort=True in the rendezvous arguments.

rendezvous(**hosts*, ***dargs*)

rendezvous_servers(*masterid*, **hosts*, ***dargs*)

autotest.client.shared.base_barrier.**get_host_from_id**(*hostid*)

class autotest.client.shared.base_barrier.**listen_server**(*address=''*, *port=11922*)
Bases: object

Manages a listening socket for barrier.

Can be used to run multiple barrier instances with the same listening socket (if they were going to listen on the same port).

Attributes:

Attr address Address to bind to (string).

Attr port Port to bind to.

Attr socket Listening socket object.

close()
Close the listening socket.

base_check_version Module

class autotest.client.shared.base_check_version.**base_check_python_version**

PYTHON_BIN_GLOB_STRINGS = ['/usr/bin/python2*', '/usr/local/bin/python2*']

extract_version(*path*)

find_desired_python()
Returns the path of the desired python interpreter.

restart()

base_job Module

class autotest.client.shared.base_job.**TAPReport**(*enable*, *resultdir=None*, *global_filename='status'*)

Bases: object

Deal with TAP reporting for the Autotest client.

job_statuses = {'END GOOD': True, 'GOOD': True, 'NOSTATUS': False, 'WARN': False, 'START': True, 'ERROR

record(*log_entry*, *indent*, *log_files*)
Append a job-level status event to self._reports_container. All events will be written to TAP log files at the end of the test run. Otherwise, it's impossilble to determine the TAP plan.

Parameters

- **log_entry** – A string status code describing the type of status entry being recorded. It must pass log.is_valid_status to be considered valid.
- **indent** – Level of the log_entry to determine the operation if log_entry.operation is not given.
- **log_files** – List of full path of files the TAP report will be written to at the end of the test.

record_keyval(*path*, *dictionary*, *type_tag=None*)
Append a key-value pairs of dictionary to self._keyval_container in TAP format. Once finished write out the keyval.tap file to the file system.

If type_tag is None, then the key must be composed of alphanumeric characters (or dashes + underscores). However, if type-tag is not null then the keys must also have "{type_tag}" as a suffix. At the moment the only valid values of type_tag are "attr" and "perf".

Parameters

- **path** – The full path of the keyval.tap file to be created
- **dictionary** – The keys and values.
- **type_tag** – The type of the values

classmethod tap_ok(*success*, *counter*, *message*)
return a TAP message string.

Parameters

- **success** – True for positive message string.
- **counter** – number of TAP line in plan.
- **message** – additional message to report in TAP line.

write()
Write the TAP reports to files.

class autotest.client.shared.base_job.**base_job**(**args*, ***dargs*)
Bases: object

An abstract base class for the various autotest job classes.

Property autodir The top level autotest directory.

Property clientdir The autotest client directory.

Property serverdir The autotest server directory. [OPTIONAL]

Property resultdir The directory where results should be written out. [WRITABLE]

Property pkgdir The job packages directory. [WRITABLE]

Property tmpdir The job temporary directory. [WRITABLE]

Property testdir The job test directory. [WRITABLE]

Property customtestdir The custom test directory. [WRITABLE]

Property site_testdir The job site test directory. [WRITABLE]

Property bindir The client bin/ directory.

Property configdir The client config/ directory.

Property profdir The client profilers/ directory.

Property toolsdir The client tools/ directory.

Property conmuxdir The conmux directory. [OPTIONAL]

Property control A path to the control file to be executed. [OPTIONAL]

Property hosts A set of all live Host objects currently in use by the job. Code running in the context of a local client can safely assume that this set contains only a single entry.

Property machines A list of the machine names associated with the job.

Property user The user executing the job.

Property tag A tag identifying the job. Often used by the scheduler to give a name of the form NUMBER-USERNAME/HOSTNAME.

Property args A list of additional miscellaneous command-line arguments provided when starting the job.

Property last_boot_tag The label of the kernel from the last reboot. [OPTIONAL,PERSISTENT]

Property automatic_test_tag A string which, if set, will be automatically added to the test name when running tests.

Property default_profile_only A boolean indicating the default value of profile_only used by test.execute. [PERSISTENT]

Property drop_caches A boolean indicating if caches should be dropped before each test is executed.

Property drop_caches_between_iterations A boolean indicating if caches should be dropped before each test iteration is executed.

Property run_test_cleanup A boolean indicating if test.cleanup should be run by default after a test completes, if the run_cleanup argument is not specified. [PERSISTENT]

Property num_tests_run The number of tests run during the job. [OPTIONAL]

Property num_tests_failed The number of tests failed during the job. [OPTIONAL]

Property bootloader An instance of the boottool class. May not be available on job instances where access to the bootloader is not available (e.g. on the server running a server job). [OPTIONAL]

Property harness An instance of the client test harness. Only available in contexts where client test execution happens. [OPTIONAL]

Property logging An instance of the logging manager associated with the job.

Property profilers An instance of the profiler manager associated with the job.

Property sysinfo An instance of the sysinfo object. Only available in contexts where it's possible to collect sysinfo.

Property warning_manager A class for managing which types of WARN messages should be logged and which should be suppressed. [OPTIONAL]

Property warning_loggers A set of readable streams that will be monitored for WARN messages to be logged. [OPTIONAL]

Abstract methods:

_find_base_directories [CLASSMETHOD] Returns the location of autodir, clientdir and serverdir

_find_resultdir Returns the location of resultdir. Gets a copy of any parameters passed into base_job.__init__. Can return None to indicate that no resultdir is to be used.

_get_status_logger Returns a status_logger instance for recording job status logs.

autodir

automatic_test_tag

bindir

clientdir

configdir

conmuxdir

customtestdir

default_profile_only

get_state(*name*, *default*=<*object object*>)
Returns the value associated with a particular name.

Parameters

- **name** – The name the value was saved with.
- **default** – A default value to return if no state is currently associated with var.

Returns A deep copy of the value associated with name. Note that this explicitly returns a deep copy to avoid problems with mutable values; mutations are not persisted or shared.

Raises `KeyError` when no state is associated with var and a default value is not provided.

last_boot_tag

pkgdir

pop_execution_context()
Reverse the effects of the previous push_execution_context call.

Raises `IndexError` when the stack of contexts is empty.

profdir

push_execution_context(*resultdir*)
Save off the current context of the job and change to the given one.

In practice method just changes the resultdir, but it may become more extensive in the future. The expected use case is for when a child job needs to be executed in some sort of nested context (for example the way parallel_simple does). The original context can be restored with a pop_execution_context call.

Parameters **resultdir** – The new resultdir, relative to the current one.

record(*status_code*, *subdir*, *operation*, *status=''*, *optional_fields=None*)
Record a job-level status event.

Logs an event noteworthy to the Autotest job as a whole. Messages will be written into a global status log file, as well as a subdir-local status log file (if subdir is specified).

Parameters

- **status_code** – A string status code describing the type of status entry being recorded. It must pass log.is_valid_status to be considered valid.
- **subdir** – A specific results subdirectory this also applies to, or None. If not None the subdirectory must exist.
- **operation** – A string describing the operation that was run.
- **status** – An optional human-readable message describing the status entry, for example an error message or "completed successfully".
- **optional_fields** – An optional dictionary of additional named fields to be included with the status message. Every time timestamp and localtime entries are generated with the current time and added to this dictionary.

record_entry(*entry*, *log_in_subdir=True*)
Record a job-level status event, using a status_log_entry.

This is the same as self.record but using an existing status log entry object rather than constructing one for you.

Parameters

- **entry** – A status_log_entry object
- **log_in_subdir** – A boolean that indicates (when true) that subdir logs should be written into the subdirectory status log file.

resultdir

run_test_cleanup

serverdir

set_state(*name*, *value*)
Saves the value given with the provided name.

Parameters

- **name** – The name the value should be saved with.
- **value** – The value to save.

site_testdir

tag

testdir

tmpdir

toolsdir

use_sequence_number

class autotest.client.shared.base_job.**job_directory**(*path*, *is_writable=False*)
Bases: object

Represents a job.*dir directory.

exception JobDirectoryException
Bases: *autotest.client.shared.error.AutotestError*

Generic job_directory exception superclass.

exception job_directory.**MissingDirectoryException**(*path*)
Bases: autotest.client.shared.base_job.JobDirectoryException

Raised when a directory required by the job does not exist.

exception job_directory.**UncreatableDirectoryException**(*path*, *error*)
Bases: autotest.client.shared.base_job.JobDirectoryException

Raised when a directory required by the job is missing and cannot be created.

exception job_directory.**UnwritableDirectoryException**(*path*)
Bases: autotest.client.shared.base_job.JobDirectoryException

Raised when a writable directory required by the job exists but is not writable.

static job_directory.**property_factory**(*attribute*)
Create a job.*dir -> job._*dir.path property accessor.

Parameters **attribute** – A string with the name of the attribute this is exposed as. '_'+attribute must then be attribute that holds either None or a job_directory-like object

Returns A read-only property object that exposes a job_directory path

class autotest.client.shared.base_job.**job_state**
Bases: object

A class for managing explicit job and user state, optionally persistent.

The class allows you to save state by name (like a dictionary). Any state stored in this class should be picklable and deep copyable. While this is not enforced it is recommended that only valid python identifiers be used as names. Additionally, the namespace 'stateful_property' is used for storing the valued associated with properties constructed using the property_factory method.

NO_DEFAULT = <object object>

PICKLE_PROTOCOL = 2

discard(**args*, ***dargs*)
If namespace.name is a defined value, deletes it.

Parameters

- **namespace** (*string*) – The namespace that the property should be stored in.
- **name** (*string*) – The name the value was saved with.

discard_namespace(**args*, ***dargs*)
Delete all defined namespace.* names.

Parameters **namespace** (*string*) – The namespace to be cleared.

get (**args*, ***dargs*)
Returns the value associated with a particular name.

Parameters

- **namespace** (*string*) – The namespace that the property should be stored in.
- **name** (*string*) – The name the value was saved with.
- **default** (*object*) – A default value to return if no state is currently associated with var.

Returns A deep copy of the value associated with name. Note that this explicitly returns a deep copy to avoid problems with mutable values; mutations are not persisted or shared.

Raises `KeyError` raised when no state is associated with var and a default value is not provided.

has (**args*, ***dargs*)
Return a boolean indicating if namespace.name is defined.

Parameters

- **namespace** (*string*) – The namespace that the property should be stored in.
- **name** (*string*) – The name the value was saved with.

Returns True if the given name is defined in the given namespace and False otherwise.

Return type bool

static property_factory (*state_attribute*, *property_attribute*, *default*, *namespace='global_properties'*)
Create a property object for an attribute using self.get and self.set.

Parameters

- **state_attribute** – A string with the name of the attribute on job that contains the job_state instance.
- **property_attribute** – A string with the name of the attribute this property is exposed as.
- **default** – A default value that should be used for this property if it is not set.
- **namespace** – The namespace to store the attribute value in.

Returns A read-write property object that performs self.get calls to read the value and self.set calls to set it.

read_from_file (*file_path*, *merge=True*)
Read in any state from the file at file_path.

When `merge=True`, any state specified only in-memory will be preserved. Any state specified on-disk will be set in-memory, even if an in-memory setting already exists.

Parameters

- **file_path** (*string*) – The path where the state should be read from. It must exist but it can be empty.
- **merge** (*bool*) – If true, merge the on-disk state with the in-memory state. If false, replace the in-memory state with the on-disk state.

Warning: This method is intentionally concurrency-unsafe. It makes no attempt to control concurrent access to the file at `file_path`.

set(**args*, ***dargs*)
Saves the value given with the provided name.

Parameters

- **namespace** (*string*) – The namespace that the property should be stored in.
- **name** (*string*) – The name the value was saved with.
- **value** – The value to save.

set_backing_file(*file_path*)
Change the path used as the backing file for the persistent state.

When a new backing file is specified if a file already exists then its contents will be added into the current state, with conflicts between the file and memory being resolved in favor of the file contents. The file will then be kept in sync with the (combined) in-memory state. The syncing can be disabled by setting this to None.

Parameters **file_path** (*string*) – A path on the filesystem that can be read from and written to, or None to turn off the backing store.

write_to_file(*file_path*)
Write out the current state to the given path.

Warning: This method is intentionally concurrency-unsafe. It makes no attempt to control concurrent access to the file at file_path.

Parameters **file_path** (*string*) – The path where the state should be written out to. Must be writable.

class autotest.client.shared.base_job.**status_indenter**
Bases: object

Abstract interface that a status log indenter should use.

decrement()
Decrease indentation by one level.

increment()
Increase indentation by one level.

indent

class autotest.client.shared.base_job.**status_log_entry**(*status_code*, *subdir*, *operation*, *message*, *fields*, *timestamp=None*)
Bases: object

Represents a single status log entry.

BAD_CHAR_REGEX = <_sre.SRE_Pattern object>

LOCALTIME_FIELD = 'localtime'

RENDERED_NONE_VALUE = '----'

TIMESTAMP_FIELD = 'timestamp'

is_end()
Indicates if this status log is the end of a nested block.

Returns A boolean indicating if this entry ends a nested block.

is_start()
Indicates if this status log is the start of a new nested block.

Returns A boolean indicating if this entry starts a new nested block.

classmethod parse(*line*)
Parse a status log entry from a text string.

This method is the inverse of render; it should always be true that parse(entry.render()) produces a new status_log_entry equivalent to entry.

Returns A new status_log_entry instance with fields extracted from the given status line. If the line is an extra message line then None is returned.

render()
Render the status log entry into a text string.

Returns A text string suitable for writing into a status log file.

class autotest.client.shared.base_job.**status_logger**(*job*, *indenter*, *global_filename='status'*, *subdir_filename='status'*, *record_hook=None*, *tap_writer=None*)
Bases: object

Represents a status log file. Responsible for translating messages into on-disk status log lines.

Property global_filename The filename to write top-level logs to.

Property subdir_filename The filename to write subdir-level logs to.

record_entry(*log_entry*, *log_in_subdir=True*)
Record a status_log_entry into the appropriate status log files.

Parameters

- **log_entry** – A status_log_entry instance to be recorded into the status logs.
- **log_in_subdir** – A boolean that indicates (when true) that subdir logs should be written into the subdirectory status log file.

render_entry(*log_entry*)
Render a status_log_entry as it would be written to a log file.

Parameters **log_entry** – A status_log_entry instance to be rendered.

Returns The status log entry, rendered as it would be written to the logs (including indentation).

autotest.client.shared.base_job.**with_backing_file**(*method*)
A decorator to perform a lock-read-*-write-unlock cycle.

When applied to a method, this decorator will automatically wrap calls to the method in a lock-and-read before the call followed by a write-and-unlock. Any operation that is reading or writing state should be decorated with this method to ensure that backing file state is consistently maintained.

autotest.client.shared.base_job.**with_backing_lock**(*method*)
A decorator to perform a lock-*-unlock cycle.

When applied to a method, this decorator will automatically wrap calls to the method in a backing file lock and before the call followed by a backing file unlock.

base_packages Module

This module defines the BasePackageManager Class which provides an implementation of the packaging system API providing methods to fetch, upload and remove packages. Site specific extensions to any of these methods should

inherit this class.

class autotest.client.shared.base_packages.**BasePackageManager**(*pkgmgr_dir*, *hostname=None*, *repo_urls=None*, *upload_paths=None*, *do_locking=True*, *run_function=<function run>*, *run_function_args=[]*, *run_function_dargs={}*)

Bases: object

add_repository(*repo*)

compare_checksum(*pkg_path*, *repo_url*)
Calculate the checksum of the file specified in pkg_path and compare it with the checksum in the checksum file Return True if both match else return False. pkg_path : The full path to the package file for which the

checksum is being compared

repo_url : The URL to fetch the checksum from

compute_checksum(*pkg_path*)
Compute the MD5 checksum for the package file and return it. pkg_path : The complete path for the package file

fetch_pkg(*pkg_name*, *dest_path*, *repo_url=None*, *use_checksum=False*, *install=False*)
Fetch the package into dest_dir from repo_url. By default repo_url is None and the package is looked in all the repositories specified. Otherwise it fetches it from the specific repo_url. pkg_name : name of the package (ex: test-sleeptest.tar.bz2,

dep-gcc.tar.bz2, kernel.1-1.rpm)

repo_url : the URL of the repository where the package is located. dest_path : complete path of where the package will be fetched to. use_checksum : This is set to False to fetch the packages.checksum file

so that the checksum comparison is bypassed for the checksum file itself. This is used internally by the packaging system. It should be ignored by externals callers of this method who use it fetch custom packages.

install [install path has unique name and destination requirements] that vary based on the fetcher that is used. So call them here as opposed to install_pkg.

get_fetcher(*url*)

get_mirror_list(*repo_urls*)
Stub function for site specific mirrors.

Returns: Priority ordered list

get_package_name(*url*, *pkg_type*)
Extract the group and test name for the url. This method is currently used only for tests.

static get_tarball_name(*name*, *pkg_type*)
Converts a package name and type into a tarball name.

Parameters

- **name** – The name of the package
- **pkg_type** – The type of the package

Returns A tarball filename for that specific type of package

install_pkg(*name*, *pkg_type*, *fetch_dir*, *install_dir*, *preserve_install_dir=False*, *repo_url=None*)
Remove install_dir if it already exists and then recreate it unless preserve_install_dir is specified as True. Fetch the package into the pkg_dir. Untar the package into install_dir The assumption is that packages are of the form : <pkg_type>.<pkg_name>.tar.bz2 name : name of the package type : type of the package fetch_dir : The directory into which the package tarball will be

fetched to.

install_dir : the directory where the package files will be untarred to repo_url : the url of the repository to fetch the package from.

static parse_tarball_name(*tarball_name*)
Coverts a package tarball name into a package name and type.

Parameters **tarball_name** – The filename of the tarball

Returns (name, pkg_type) where name is the package name and pkg_type is the package type.

remove_checksum(*pkg_name*)
Remove the checksum of the package from the packages checksum file. This method is called whenever a package is removed from the repositories in order clean its corresponding checksum. pkg_name : The name of the package to be removed

remove_pkg(*pkg_name*, *remove_path=None*, *remove_checksum=False*)
Remove the package from the specified remove_path pkg_name : name of the package (ex: test-sleeptest.tar.bz2,

dep-gcc.tar.bz2)

remove_path : the location to remove the package from.

remove_pkg_file(*filename*, *pkg_dir*)
Remove the file named filename from pkg_dir

repo_check(*repo*)
Check to make sure the repo is in a sane state: ensure we have at least XX amount of free space Make sure we can write to the repo

tar_package(*pkg_name*, *src_dir*, *dest_dir*, *include_string=None*, *exclude_string=None*)
Create a tar.bz2 file with the name 'pkg_name' say test-blah.tar.bz2.

Includes the files specified in include_string, and excludes the files specified on the exclude string, while tarring the source. Returns the destination tarball path.

Parameters

- **pkg_name** – Package name.
- **src_dir** – Directory that contains the data to be packaged.
- **dest_dir** – Directory that will hold the destination tarball.
- **include_string** – Pattern that represents the files that will be added to the tar package.
- **exclude_string** – Pattern that represents the files that should be excluded from the tar package. It could be either a string or a list.

untar_pkg(*tarball_path*, *dest_dir*)
Untar the package present in the tarball_path and put a ".checksum" file in the dest_dir containing the checksum of the tarball. This method assumes that the package to be untarred is of the form <name>.tar.bz2

untar_required(*tarball_path*, *dest_dir*)
Compare the checksum of the tarball_path with the .checksum file in the dest_dir and return False if it matches. The untar of the package happens only if the checksums do not match.

update_checksum(*pkg_path*)
Update the checksum of the package in the packages' checksum file. This method is called whenever a package is fetched just to be sure that the checksums in the local file are the latest. pkg_path : The complete path to the package file.

upkeep(*custom_repos=None*)
Clean up custom upload/download areas

upload_pkg(*pkg_path*, *upload_path=None*, *update_checksum=False*, *timeout=300*)

upload_pkg_dir(*dir_path*, *upload_path*)
Upload a full directory. Depending on the upload path, the appropriate method for that protocol is called. Currently this copies the whole tmp package directory to the target directory. This assumes that the web server is running on the same machine where the method is being called from. The upload_path's files are basically served by that web server.

upload_pkg_file(*file_path*, *upload_path*)
Upload a single file. Depending on the upload path, the appropriate method for that protocol is called. Currently this simply copies the file to the target directory (but can be extended for other protocols) This assumes that the web server is running on the same machine where the method is being called from. The upload_path's files are basically served by that web server.

upload_pkg_parallel(*pkg_path*, *upload_path*, *update_checksum=False*)
Uploads to a specified upload_path or to all the repos. Also uploads the checksum file to all the repos. pkg_path : The complete path to the package file upload_path : the absolute path where the files are copied to.

if set to 'None' assumes 'all' repos

update_checksum [If set to False, the checksum file is not] going to be updated which happens by default. This is necessary for custom packages (like custom kernels and custom tests) that get uploaded which do not need to be part of the checksum file and bloat it.

class autotest.client.shared.base_packages.**GitFetcher**(*package_manager*, *repository_url*)
Bases: *autotest.client.shared.base_packages.RepositoryFetcher*

A git based repository fetcher

fetch_pkg_file(*filename*, *dest_path*)
Fetch a package file and save it to the given destination path

git is an SCM, you can download the test directly. No need to fetch a bz2'd tarball file. However 'filename' is <type>-<name>.tar.bz2 break this up and only fetch <name>.

Parameters

- **filename** (*string*) – The filename of the package file to fetch.
- **dest_path** (*string*) – Destination path to download the file to.

git_archive_cmd_pattern = 'git archive –remote=%s -o %s %s'

install_pkg_post(*filename*, *fetch_dir*, *install_dir*, *preserve_install_dir=False*)

class autotest.client.shared.base_packages.**HttpFetcher**(*package_manager*, *repository_url*)
Bases: *autotest.client.shared.base_packages.RepositoryFetcher*

Repository Fetcher using HTTP

fetch_pkg_file (*filename*, *dest_path*)
Fetch a package file from a package repository.

Parameters

- **filename** (*string*) – The filename of the package file to fetch.
- **dest_path** (*string*) – Destination path to download the file to.

Raises PackageFetchError if the fetch failed

wget_cmd_pattern = 'wget –connect-timeout=15 -nv %s -O %s'

class autotest.client.shared.base_packages.**LocalFilesystemFetcher** (*package_manager*, *repository_url*)
Bases: *autotest.client.shared.base_packages.RepositoryFetcher*

fetch_pkg_file (*filename*, *dest_path*)

class autotest.client.shared.base_packages.**RepositoryFetcher** (*package_manager*, *repository_url*)
Bases: object

Base class with common functionality for repository fetchers

fetch_pkg_file (*filename*, *dest_path*)
Fetch a package file from a package repository.

Parameters

- **filename** (*string*) – The filename of the package file to fetch.
- **dest_path** (*string*) – Destination path to download the file to.

Raises PackageFetchError if the fetch failed

install_pkg_post (*filename*, *fetch_dir*, *install_dir*, *preserve_install_dir=False*)
Fetcher specific post install

Parameters

- **filename** (*string*) – The filename of the package to install
- **fetch_dir** (*string*) – The fetched path of the package
- **install_dir** (*string*) – The path to install the package to

@preserve_install_dir: Preserve the install directory

install_pkg_setup (*name*, *fetch_dir*, *install*)
Install setup for a package based on fetcher type.

Parameters

- **name** (*string*) – The filename to be munged
- **fetch_dir** (*string*) – The destination path to be munged
- **install** (*boolean*) – Whether this is be called from the install path or not

Returns tuple with (name, fetch_dir)

url = None

autotest.client.shared.base_packages.**check_diskspace**(*repo*, *min_free=None*)
Check if the remote directory over at the pkg repo has available diskspace

If the amount of free space is not supplied, it is taken from the global configuration file, section [PACKAGES], key 'mininum_free_space'. The unit used are in SI, that is, 1 GB = 10**9 bytes.

Parameters **repo** (*string*) – a remote package repo URL

Param min_free mininum amount of free space, in GB (10**9 bytes)

Raises

- **error.RepoUnknownError** – general repository error condition
- **error.RepoDiskFullError** – repository does not have at least the requested amount of free disk space.

autotest.client.shared.base_packages.**check_write**(*repo*)
Checks that the remote repository directory is writable

Parameters **repo** (*string*) – a remote package repo URL

Raises error.RepoWriteError repository write error

autotest.client.shared.base_packages.**create_directory**(*repo*)
Create a directory over at the remote repository

Parameters **repo** (*string*) – the repo URL containing the remote directory path

Returns a CmdResult object or None

autotest.client.shared.base_packages.**has_pbzip2**()
Check if parallel bzip2 is available on this system.

Returns True if pbzip2 is available, False otherwise

autotest.client.shared.base_packages.**parse_ssh_path**(*repo*)
Parse an SSH url

Parameters **repo** (*string*) – a repo uri like ssh://xx@xx/path/to/

Returns tuple with (host, remote_path)

autotest.client.shared.base_packages.**repo_run_command**(*repo*, *cmd*, *ignore_status=False*, *cd=True*)
Run a command relative to the repo path

This is basically a utils.run() wrapper that sets itself in a repo directory if it is appropriate, so parameters such as cmd and ignore_status are passed along to it.

Parameters

- **repo** (*string*) – a repository url
- **cmd** (*string*) – the command to be executed. This is passed along to utils.run()
- **ignore_status** (*boolean*) – do not raise an exception, no matter what the exit code of the command is.
- **cd** (*boolean*) – wether to change the working directory to the repo directory before running the specified command.

Returns a CmdResult object or None

Raises CmdError the exit code of the command execution was not 0

autotest.client.shared.base_packages.**trim_custom_directories**(*repo*, *older_than_days=None*)

Remove old files from the remote repo directory

The age of the files, if not provided by the older_than_days parameter is taken from the global configuration file, at section [PACKAGES], configuration item 'custom_max_age'.

Parameters **repo** (*string*) – a remote package repo URL

base_syncdata Module

class autotest.client.shared.base_syncdata.**SessionData**(*hosts*, *timeout*)

Bases: object

close()

is_finished()

set_finish()

timeout()

class autotest.client.shared.base_syncdata.**SyncData**(*masterid*, *hostid*, *hosts*, *session_id=None*, *listen_server=None*, *port=13234*, *tmpdir=None*)

Bases: object

Provides data synchronization between hosts.

Transferred data is pickled and sent to all destination points. If there is no listen server it will create a new one. If multiple hosts wants to communicate with each other, then communications are identified by session_id.

close()

single_sync(*data=None*, *timeout=60*, *session_id=None*)

sync(*data=None*, *timeout=60*, *session_id=None*)

Synchronize data between hosts.

timeout()

class autotest.client.shared.base_syncdata.**SyncListenServer**(*address=''*, *port=13234*, *tmpdir=None*)

Bases: object

close()

Close SyncListenServer thread.

Close all open connection with clients and listen server.

class autotest.client.shared.base_syncdata.**TempDir**(*tmpdir=None*)

Bases: *autotest.client.shared.autotemp.tempdir*

TempDir class is tempdir for predefined tmpdir.

clean()

Should not delete predefined tmpdir.

autotest.client.shared.base_syncdata.**net_recv_object**(*sock*, *timeout=60*)

Receive python object over network.

Parameters

- **ip_addr** – ipaddres of waiter for data.

- **obj** – object to send

Returns object from network

autotest.client.shared.base_syncdata.**net_send_object**(*sock*, *obj*)
Send python object over network.

Parameters

- **ip_addr** – ipaddres of waiter for data.
- **obj** – object to send

boottool Module

boottool client-side module.

This module provides an API for client side tests that need to manipulate boot entries. It's based on the rewrite of boottool, now python and grubby based. It aims to be keep API compatibility with the older version, except from XEN support which has been removed. We'll gladly accept patches that provide full coverage for this mode/feature.

Copyright 2009 Google Inc. Copyright 2012 Red Hat, Inc.

Released under the GPL v2

class autotest.client.shared.boottool.**boottool**(*path=None*)
Bases: *autotest.client.tools.boottool.Grubby*

Client site side boottool wrapper.

Inherits all functionality from boottool(.py) CLI app (lazily).

check_version Module

class autotest.client.shared.check_version.**check_python_version**
Bases: *autotest.client.shared.check_version.site_check_python_version*, *autotest.client.shared.base_check_version.base_check_python_version*

class autotest.client.shared.check_version.**site_check_python_version**

common Module

control_data Module

class autotest.client.shared.control_data.**ControlData**(*vars*, *path*, *raise_warnings=False*)
Bases: object

set_attr(*attr*, *val*, *raise_warnings=False*)

set_author(*val*)

set_dependencies(*val*)

set_doc(*val*)

set_experimental(*val*)

set_name(*val*)

set_run_verify(*val*)

set_sync_count(*val*)

set_test_category(*val*)

set_test_class(*val*)

set_test_parameters(*val*)

set_test_type(*val*)

set_time(*val*)

exception autotest.client.shared.control_data.**ControlVariableException**
Bases: exceptions.Exception

autotest.client.shared.control_data.**parse_control**(*path*, *raise_warnings=False*)

distro Module

This module provides the client facilities to detect the Linux Distribution it's running under.

This is a replacement for the get_os_vendor() function from the utils module.

class autotest.client.shared.distro.**LinuxDistro**(*name*, *version*, *release*, *arch*)
Bases: object

Simple collection of information for a Linux Distribution

class autotest.client.shared.distro.**Probe**
Bases: object

Probes the machine and does it best to confirm it's the right distro

CHECK_FILE = None
Points to a file that can determine if this machine is running a given Linux Distribution. This servers a first check that enables the extra checks to carry on.

CHECK_FILE_CONTAINS = None
Sets the content that should be checked on the file pointed to by CHECK_FILE_EXISTS. Leave it set to *None* (its default) to check only if the file exists, and not check its contents

CHECK_FILE_DISTRO_NAME = None
The name of the Linux Distribution to be returned if the file defined by CHECK_FILE_EXISTS exist.

CHECK_VERSION_REGEX = None
A regular expresion that will be run on the file pointed to by CHECK_FILE_EXISTS

check_name_for_file()
Checks if this class will look for a file and return a distro

The conditions that must be true include the file that identifies the distro file being set (*CHECK_FILE*) and the name of the distro to be returned (*CHECK_FILE_DISTRO_NAME*)

check_name_for_file_contains()
Checks if this class will look for text on a file and return a distro

The conditions that must be true include the file that identifies the distro file being set (*CHECK_FILE*), the text to look for inside the distro file (*CHECK_FILE_CONTAINS*) and the name of the distro to be returned (*CHECK_FILE_DISTRO_NAME*)

check_release()
Checks if this has the conditions met to look for the release number

check_version()
Checks if this class will look for a regex in file and return a distro

get_distro()
Returns the *LinuxDistro* this probe detected

name_for_file()
Get the distro name if the *CHECK_FILE* is set and exists

name_for_file_contains()
Get the distro if the *CHECK_FILE* is set and has content

release()
Returns the release of the distro

version()
Returns the version of the distro

autotest.client.shared.distro.**register_probe**(*probe_class*)
Register a probe to be run during autodetection

autotest.client.shared.distro.**detect**()
Attempts to detect the Linux Distribution running on this machine

Returns the detected *LinuxDistro* or *UNKNOWN_DISTRO*

Return type *LinuxDistro*

distro_def Module

This module defines a structure and portable format for relevant information on Linux Distributions in such a way that information about known distros can be packed and distributed.

Please note that this module deals with Linux Distributions not necessarily installed on the running system.

autotest.client.shared.distro_def.**save**(*linux_distro*, *path*)
Saves the linux_distro to an external file format

Parameters

- **linux_distro** (DistroDef) – an *DistroDef* instance
- **path** (*str*) – the location for the output file

Returns None

autotest.client.shared.distro_def.**load**(*path*)
Loads the distro from an external file

Parameters **path** (*str*) – the location for the input file

Returns an *DistroDef* instance

Return type *DistroDef*

autotest.client.shared.distro_def.**load_from_tree**(*name*, *version*, *release*, *arch*, *package_type*, *path*)
Loads a DistroDef from an installable tree

Parameters

- **name** (*str*) – a short name that precisely distinguishes this Linux Distribution among all others.

- **version** (*str*) – the major version of the distribution. Usually this is a single number that denotes a large development cycle and support file.
- **release** (*str*) – the release or minor version of the distribution. Usually this is also a single number, that is often omitted or starts with a 0 when the major version is initially release. It's ofter associated with a shorter development cycle that contains incremental a collection of improvements and fixes.
- **arch** (*str*) – the main target for this Linux Distribution. It's common for some architectures to ship with packages for previous and still compatible architectures, such as it's the case with Intel/AMD 64 bit architecture that support 32 bit code. In cases like this, this should be set to the 64 bit architecture name.
- **package_type** (*str*) – one of the available package info loader types
- **path** (*str*) – top level directory of the distro installation tree files

class autotest.client.shared.distro_def.**SoftwarePackage**(*name*, *version*, *release*, *checksum*, *arch*)

Bases: object

Definition of relevant information on a software package

class autotest.client.shared.distro_def.**DistroDef**(*name*, *version*, *release*, *arch*)

Bases: *autotest.client.shared.distro.LinuxDistro*

More complete information on a given Linux Distribution

software_packages = None

All the software packages that ship with this Linux distro

software_packages_type = None

A simple text that denotes the software type that makes this distro

autotest.client.shared.distro_def.**DISTRO_PKG_INFO_LOADERS = {'deb': <class 'autotest.client.shared.distro_**

the type of distro that will determine what loader will be used

enum Module

Generic enumeration support.

class autotest.client.shared.enum.**Enum**(**names*, ***kwargs*)

Bases: object

Utility class to implement Enum-like functionality.

```
>>> e = Enum('String one', 'String two')
>>> e.STRING_ONE
0
>>> e.STRING_TWO
1
>>> e.choices()
[(0, 'String one'), (1, 'String two')]
>>> e.get_value('String one')
0
>>> e.get_string(0)
'String one'
```

```
>>> e = Enum('Hello', 'Goodbye', string_values=True)
>>> e.HELLO, e.GOODBYE
('Hello', 'Goodbye')
```

```
>>> e = Enum('One', 'Two', start_value=1)
>>> e.ONE
1
>>> e.TWO
2
```

choices()
Return choice list suitable for Django model choices.

static get_attr_name(*string*)

get_string(*value*)
Given a value, get the string name for it.

get_value(*name*)
Convert a string name to it's corresponding value. If a value is passed in, it is returned.

error Module

Internal global error types

autotest.client.shared.error.**format_error**()

autotest.client.shared.error.**context_aware**(*fn*)
A decorator that must be applied to functions that call context().

autotest.client.shared.error.**context**(*s=''*, *log=None*)
Set the context for the currently executing function and optionally log it.

Parameters

- **s** – A string. If not provided, the context for the current function will be cleared.
- **log** – A logging function to pass the context message to. If None, no function will be called.

autotest.client.shared.error.**get_context**()
Return the current context (or None if none is defined).

autotest.client.shared.error.**exception_context**(*e*)
Return the context of a given exception (or None if none is defined).

exception autotest.client.shared.error.**AutoservHostIsShuttingDownError**
Bases: *autotest.client.shared.error.AutoservHostError*

Host is shutting down

exception autotest.client.shared.error.**AutoservShutdownError**
Bases: *autotest.client.shared.error.AutoservRebootError*

Error occurred during shutdown of machine

exception autotest.client.shared.error.**AutoservHardwareRepairRequiredError**
Bases: *autotest.client.shared.error.AutoservError*

Exception class raised during repairs to indicate that a hardware repair is going to be necessary.

exception autotest.client.shared.error.**RepoWriteError**
Bases: *autotest.client.shared.error.PackagingError*

Raised when packager cannot write to a repo's desitnation

exception autotest.client.shared.error.**AutoservUnsupportedError**
Bases: *autotest.client.shared.error.AutoservError*

Error raised when you try to use an unsupported optional feature

exception autotest.client.shared.error.**CmdError**(*command*, *result_obj*, *additional_text=None*)
Bases: *autotest.client.shared.error.TestError*

Indicates that a command failed, is fatal to the test unless caught.

exception autotest.client.shared.error.**AutotestError**
Bases: exceptions.Exception

The parent of all errors deliberately thrown within the client code.

exception autotest.client.shared.error.**RepoDiskFullError**
Bases: *autotest.client.shared.error.PackagingError*

Raised when the destination for packages is full

exception autotest.client.shared.error.**AutoservRebootError**
Bases: *autotest.client.shared.error.AutoservError*

Error occurred while rebooting a machine

exception autotest.client.shared.error.**TestWarn**
Bases: *autotest.client.shared.error.TestBaseException*

Indicates that bad things (may) have happened, but not an explicit failure.

exit_status = 'WARN'

exception autotest.client.shared.error.**PackageInstallError**
Bases: *autotest.client.shared.error.PackagingError*

Raised when there is an error installing the package

exception autotest.client.shared.error.**HostInstallProfileError**
Bases: *autotest.client.shared.error.JobError*

Indicates the machine failed to have a profile assigned.

exception autotest.client.shared.error.**PackageError**
Bases: *autotest.client.shared.error.TestError*

Indicates an error trying to perform a package operation.

exception autotest.client.shared.error.**AutotestHostRunError**(*description*, *result_obj*)
Bases: *autotest.client.shared.error.HostRunErrorMixIn*, *autotest.client.shared.error.AutotestError*

exception autotest.client.shared.error.**UnhandledTestFail**(*unhandled_exception*)
Bases: *autotest.client.shared.error.TestFail*

Indicates an unhandled fail in a test.

exception autotest.client.shared.error.**BarrierAbortError**
Bases: *autotest.client.shared.error.BarrierError*

Indicate that the barrier was explicitly aborted by a member.

exception autotest.client.shared.error.**AutoservSubcommandError**(*func*, *exit_code*)
Bases: *autotest.client.shared.error.AutoservError*

Indicates an error while executing a (forked) subcommand

exception autotest.client.shared.error.**NetCommunicationError**
Bases: *autotest.client.shared.error.JobError*

Indicate that network communication was broken.

exception autotest.client.shared.error.**PackageRemoveError**
Bases: *autotest.client.shared.error.PackagingError*

Raised when there is an error removing the package

exception autotest.client.shared.error.**UnhandledTestError**(*unhandled_exception*)
Bases: *autotest.client.shared.error.TestError*

Indicates an unhandled error in a test.

exception autotest.client.shared.error.**DataSyncError**
Bases: *autotest.client.shared.error.NetCommunicationError*

Indicates problem during synchronization data over network.

exception autotest.client.shared.error.**AutoservHostError**
Bases: *autotest.client.shared.error.AutoservError*

Error reaching a host

exception autotest.client.shared.error.**TestBaseException**
Bases: *autotest.client.shared.error.AutotestError*

The parent of all test exceptions.

exit_status = 'NEVER_RAISE_THIS'

exception autotest.client.shared.error.**TestNAError**
Bases: *autotest.client.shared.error.TestBaseException*

Indictates that the test is Not Applicable. Should be thrown when various conditions are such that the test is inappropriate.

exit_status = 'TEST_NA'

exception autotest.client.shared.error.**AutoservHardwareHostError**
Bases: *autotest.client.shared.error.AutoservHostError*

Found hardware problems with the host

exception autotest.client.shared.error.**AutoservError**
Bases: exceptions.Exception

exception autotest.client.shared.error.**AutoservSSHTimeout**
Bases: *autotest.client.shared.error.AutoservError*

SSH experienced a connection timeout

exception autotest.client.shared.error.**InstallError**
Bases: *autotest.client.shared.error.JobError*

Indicates an installation error which Terminates and fails the job.

exception autotest.client.shared.error.**AutoservDiskFullHostError**(*path*, *want_gb*, *free_space_gb*)
Bases: *autotest.client.shared.error.AutoservHostError*

Not enough free disk space on host

exception autotest.client.shared.error.**AutoservInstallError**
Bases: *autotest.client.shared.error.AutoservError*

Error occurred while installing autotest on a host

exception autotest.client.shared.error.**TestError**
Bases: *autotest.client.shared.error.TestBaseException*

Indicates that something went wrong with the test harness itself.

exit_status = 'ERROR'

exception autotest.client.shared.error.**AutoservVirtError**
Bases: *autotest.client.shared.error.AutoservError*

Vitualization related error

exception autotest.client.shared.error.**BarrierError**
Bases: *autotest.client.shared.error.JobError*

Indicates an error happened during a barrier operation.

exception autotest.client.shared.error.**AutotestRunError**
Bases: *autotest.client.shared.error.AutotestError*

Indicates a problem running server side control files.

exception autotest.client.shared.error.**RepoError**
Bases: *autotest.client.shared.error.PackagingError*

Raised when a repo isn't working in some way

exception autotest.client.shared.error.**PackagingError**
Bases: *autotest.client.shared.error.AutotestError*

Abstract error class for all packaging related errors.

exception autotest.client.shared.error.**RepoUnknownError**
Bases: *autotest.client.shared.error.PackagingError*

Raised when packager cannot write to a repo's desitnation

exception autotest.client.shared.error.**UnhandledJobError**(*unhandled_exception*)
Bases: *autotest.client.shared.error.JobError*

Indicates an unhandled error in a job.

exception autotest.client.shared.error.**TestFail**
Bases: *autotest.client.shared.error.TestBaseException*

Indicates that the test failed, but the job will not continue.

exit_status = 'FAIL'

exception autotest.client.shared.error.**JobError**
Bases: *autotest.client.shared.error.AutotestError*

Indicates an error which terminates and fails the whole job (ABORT).

exception autotest.client.shared.error.**AutoservRunError**(*description*, *result_obj*)
Bases: *autotest.client.shared.error.HostRunErrorMixIn*, *autotest.client.shared.error.AutoservError*

exception autotest.client.shared.error.**PackageFetchError**
Bases: *autotest.client.shared.error.PackagingError*

Raised when there is an error fetching the package

exception autotest.client.shared.error.**PackageUploadError**
Bases: *autotest.client.shared.error.PackagingError*

Raised when there is an error uploading the package

exception autotest.client.shared.error.**AutoservHardwareRepairRequestedError**
Bases: *autotest.client.shared.error.AutoservError*

Exception class raised from Host.repair_full() (or overrides) when software repair fails but it successfully managed to request a hardware repair (by notifying the staff, sending mail, etc)

exception autotest.client.shared.error.**HostRunErrorMixIn**(*description*, *result_obj*)
Bases: exceptions.Exception

Indicates a problem in the host run() function raised from client code. Should always be constructed with a tuple of two args (error description (str), run result object). This is a common class mixed in to create the client and server side versions of it.

exception autotest.client.shared.error.**HarnessError**
Bases: *autotest.client.shared.error.JobError*

Indicates problem with the harness.

exception autotest.client.shared.error.**AutoservNotMountedHostError**
Bases: *autotest.client.shared.error.AutoservHostError*

Found unmounted partitions that should be mounted

exception autotest.client.shared.error.**AutoservSshPermissionDeniedError**(*description*, *result_obj*)
Bases: *autotest.client.shared.error.AutoservRunError*

Indicates that a SSH permission denied error was encountered.

exception autotest.client.shared.error.**HostInstallTimeoutError**
Bases: *autotest.client.shared.error.JobError*

Indicates the machine failed to be installed after the predetermined timeout.

exception autotest.client.shared.error.**AutoservSshPingHostError**
Bases: *autotest.client.shared.error.AutoservHostError*

SSH ping failed

exception autotest.client.shared.error.**AutotestTimeoutError**
Bases: *autotest.client.shared.error.AutotestError*

This exception is raised when an autotest test exceeds the timeout parameter passed to run_timed_test and is killed.

git Module

Code that helps to deal with content from git repositories

class autotest.client.shared.git.**GitRepoHelper**(*uri*, *branch='master'*, *lbranch='master'*, *commit=None*, *destination_dir=None*, *base_uri=None*)
Bases: object

Helps to deal with git repos, mostly fetching content from a repo

checkout(*branch=None*, *commit=None*)
Performs a git checkout for a given branch and start point (commit)

Parameters

- **branch** – Remote branch name.
- **commit** – Specific commit hash.

execute()
Performs all steps necessary to initialize and download a git repo.

This includes the init, fetch and checkout steps in one single utility method.

fetch(*uri*)
Performs a git fetch from the remote repo

get_top_commit()
Returns the topmost commit id for the current branch.

Returns Commit id.

get_top_tag()
Returns the topmost tag for the current branch.

Returns Tag.

git_cmd(*cmd*, *ignore_status=False*)
Wraps git commands.

Parameters

- **cmd** – Command to be executed.
- **ignore_status** – Whether we should suppress error.CmdError exceptions if the command did return exit code !=0 (True), or not suppress them (False).

init()
Initializes a directory for receiving a verbatim copy of git repo

This creates a directory if necessary, and either resets or inits the repo

autotest.client.shared.git.**get_repo**(*uri*, *branch='master'*, *lbranch='master'*, *commit=None*, *destination_dir=None*, *base_uri=None*)
Utility function that retrieves a given git code repository.

Parameters

- **uri** (*string*) – git repository url
- **branch** (*string*) – git remote branch
- **destination_dir** (*string*) – path of a dir where to save downloaded code
- **commit** (*string*) – specific commit to download
- **lbranch** (*string*) – git local branch name, if different from remote
- **uri** – a closer, usually local, git repository url from where to fetch content first from

host_protections Module

host_queue_entry_states Module

This module contains the status enums for use by HostQueueEntrys in the database. It is a stand alone module as these status strings are needed from various disconnected pieces of code that should not depend on everything that autotest.frontend.afe.models depends on such as RPC clients.

iscsi Module

Basic iscsi support for Linux host with the help of commands iscsiadm and tgtadm.

This include the basic operates such as login and get device name by target name. And it can support the real iscsi access and emulated iscsi in localhost then access it.

class autotest.client.shared.iscsi.**Iscsi**(*params*, *root_dir='/tmp'*)
Bases: object

Basic iscsi support class. Will handle the emulated iscsi export and access to both real iscsi and emulated iscsi device.

cleanup()
Clean up env after iscsi used.

delete_target()
Delete target from host.

export_target()
Export target in localhost for emulated iscsi

get_device_name()
Get device name from the target name.

get_target_id()
Get target id from image name. Only works for emulated iscsi device

logged_in()
Check if the session is login or not.

login()
Login session for both real iscsi device and emulated iscsi. Include env check and setup.

logout()
Logout from target.

portal_visible()
Check if the portal can be found or not.

autotest.client.shared.iscsi.**iscsi_discover**(*portal_ip*)
Query from iscsi server for available targets

Parameters **portal_ip** – Ip for iscsi server

autotest.client.shared.iscsi.**iscsi_get_nodes**()
Get the iscsi nodes

autotest.client.shared.iscsi.**iscsi_get_sessions**()
Get the iscsi sessions activated

autotest.client.shared.iscsi.**iscsi_login**(*target_name*)
Login to a target with the target name

Parameters **target_name** – Name of the target

autotest.client.shared.iscsi.**iscsi_logout**(*target_name=None*)
Logout from a target. If the target name is not set then logout all targets.

Params target_name Name of the target.

iso9660 Module

Basic ISO9660 file-system support.

This code does not attempt (so far) to implement code that knows about ISO9660 internal structure. Instead, it uses commonly available support either in userspace tools or on the Linux kernel itself (via mount).

autotest.client.shared.iso9660.**iso9660**(*path*)
Checks the avaiable tools on a system and chooses class accordingly

This is a convinience function, that will pick the first avaialable iso9660 capable tool.

Parameters **path** (*str*) – path to an iso9660 image file

Returns an instance of any iso9660 capable tool

Return type *Iso9660IsoInfo*, *Iso9660IsoRead*, *Iso9660Mount* or None

class autotest.client.shared.iso9660.**Iso9660IsoInfo**(*path*)
Bases: autotest.client.shared.iso9660.BaseIso9660

Represents a ISO9660 filesystem

This implementation is based on the cdrkit's isoinfo tool

read(*path*)

class autotest.client.shared.iso9660.**Iso9660IsoRead**(*path*)
Bases: autotest.client.shared.iso9660.BaseIso9660

Represents a ISO9660 filesystem

This implementation is based on the libcdio's iso-read tool

close()

copy(*src*, *dst*)

read(*path*)

class autotest.client.shared.iso9660.**Iso9660Mount**(*path*)
Bases: autotest.client.shared.iso9660.BaseIso9660

Represents a mounted ISO9660 filesystem.

close()
Perform umount operation on the temporary dir

Return type None

copy(*src*, *dst*)

Parameters

- **src** (*str*) – source
- **dst** (*str*) – destination

Return type None

read(*path*)
Read data from path

Parameters **path** (*str*) – path to read data

Returns data content

Return type str

jsontemplate Module

Python implementation of json-template.

JSON Template is a minimal and powerful templating language for transforming a JSON dictionary to arbitrary text.

To use this module, you will typically use the Template constructor, and catch various exceptions thrown. You may also want to use the FromFile/FromString methods, which allow Template constructor options to be embedded in the template string itself.

Other functions are exposed for tools which may want to process templates.

exception autotest.client.shared.jsontemplate.**Error**
Bases: exceptions.Exception

Base class for all exceptions in this module.

Thus you can "except jsontemplate.Error: to catch all exceptions thrown by this module.

exception autotest.client.shared.jsontemplate.**CompilationError**
Bases: *autotest.client.shared.jsontemplate.Error*

Base class for errors that happen during the compilation stage.

exception autotest.client.shared.jsontemplate.**EvaluationError**(*msg*, *original_exception=None*)
Bases: *autotest.client.shared.jsontemplate.Error*

Base class for errors that happen when expanding the template.

This class of errors generally involve the data dictionary or the execution of the formatters.

exception autotest.client.shared.jsontemplate.**BadFormatter**
Bases: *autotest.client.shared.jsontemplate.CompilationError*

A bad formatter was specified, e.g. {variable|BAD}

exception autotest.client.shared.jsontemplate.**BadPredicate**
Bases: *autotest.client.shared.jsontemplate.CompilationError*

A bad predicate was specified, e.g. {.BAD?}

exception autotest.client.shared.jsontemplate.**MissingFormatter**
Bases: *autotest.client.shared.jsontemplate.CompilationError*

Raised when formatters are required, and a variable is missing a formatter.

exception autotest.client.shared.jsontemplate.**ConfigurationError**
Bases: *autotest.client.shared.jsontemplate.CompilationError*

Raised when the Template options are invalid and it can't even be compiled.

exception autotest.client.shared.jsontemplate.**TemplateSyntaxError**
Bases: *autotest.client.shared.jsontemplate.CompilationError*

Syntax error in the template text.

exception autotest.client.shared.jsontemplate.**UndefinedVariable**(*msg*, *original_exception=None*)
Bases: *autotest.client.shared.jsontemplate.EvaluationError*

The template contains a variable not defined by the data dictionary.

autotest.client.shared.jsontemplate.**CompileTemplate**(*template_str*, *builder=None*, *meta='{}'*, *format_char='|'*, *more_formatters=<function <lambda>>*, *more_predicates=<function <lambda>>*, *default_formatter='str'*)

Compile the template string, calling methods on the 'program builder'.

Args:

template_str: The template string. It should not have any compilation options in the header – those are parsed by FromString/FromFile

builder: The interface of _ProgramBuilder isn't fixed. Use at your own risk.

meta: The metacharacters to use, e.g. '{}', '[]'.

more_formatters:

Something that can map format strings to formatter functions. One of:

- A plain dictionary of names -> functions e.g. {'html': cgi.escape}
- A higher-order function which takes format strings and returns formatter functions. Useful for when formatters have parsed arguments.
- A FunctionRegistry instance for the most control. This allows formatters which takes contexts as well.

more_predicates: Like more_formatters, but for predicates.

default_formatter: The formatter to use for substitutions that are missing a formatter. The 'str' formatter the "default default" – it just tries to convert the context value to a string in some unspecified manner.

Returns: The compiled program (obtained from the builder)

Raises: The various subclasses of CompilationError. For example, if default_formatter=None, and a variable is missing a formatter, then MissingFormatter is raised.

This function is public so it can be used by other tools, e.g. a syntax checking tool run before submitting a template to source control.

autotest.client.shared.jsontemplate.**FromString**(*s*, *more_formatters=<function <lambda>>*, *_constructor=None*)

Like FromFile, but takes a string.

autotest.client.shared.jsontemplate.**FromFile**(*f*, *more_formatters=<function <lambda>>*, *_constructor=None*)

Parse a template from a file, using a simple file format.

This is useful when you want to include template options in a data file, rather than in the source code.

The format is similar to HTTP or E-mail headers. The first lines of the file can specify template options, such as the metacharacters to use. One blank line must separate the options from the template body.

Example:

default-formatter: none meta: {{}} format-char: : <blank line required> Template goes here: {{variable:html}}

Args: f: A file handle to read from. Caller is responsible for opening and closing it.

class autotest.client.shared.jsontemplate.**Template**(*template_str*, *builder=None*, *undefined_str=None*, ***compile_options*)

Bases: object

Represents a compiled template.

Like many template systems, the template string is compiled into a program, and then it can be expanded any number of times. For example, in a web app, you can compile the templates once at server startup, and use the expand() method at request handling time. expand() uses the compiled representation.

There are various options for controlling parsing – see CompileTemplate. Don't go crazy with metacharacters. {}, [], {{}} or <> should cover nearly any circumstance, e.g. generating HTML, CSS XML, JavaScript, C programs, text files, etc.

expand(**args*, ***kwargs*)

Expands the template with the given data dictionary, returning a string.

This is a small wrapper around render(), and is the most convenient interface.

Args: The JSON data dictionary. Like the builtin dict() constructor, it can take a single dictionary as a positional argument, or arbitrary keyword arguments.

Returns: The return value could be a str() or unicode() instance, depending on the the type of the template string passed in, and what the types the strings in the dictionary are.

render(*data_dict*, *callback*)

Low level method to expands the template piece by piece.

Args: data_dict: The JSON data dictionary. callback: A callback which should be called with each expanded token.

Example: You can pass 'f.write' as the callback to write directly to a file handle.

tokenstream(*data_dict*)

Yields a list of tokens resulting from expansion.

This may be useful for WSGI apps. NOTE: In the current implementation, the entire expanded template must be stored memory.

NOTE: This is a generator, but JavaScript doesn't have generators.

autotest.client.shared.jsontemplate.**expand**(*template_str*, *dictionary*, ***kwargs*)

Free function to expands a template string with a data dictionary.

This is useful for cases where you don't care about saving the result of compilation (similar to re.match('.*', s) vs DOT_STAR.match(s))

kernel_versions Module

autotest.client.shared.kernel_versions.**is_release_candidate**(*version*)

autotest.client.shared.kernel_versions.**is_released_kernel**(*version*)

autotest.client.shared.kernel_versions.**version_choose_config**(*version*, *candidates*)

autotest.client.shared.kernel_versions.**version_encode**(*version*)

autotest.client.shared.kernel_versions.**version_len**(*version*)

autotest.client.shared.kernel_versions.**version_limit**(*version*, *n*)

log Module

autotest.client.shared.log.**is_failure**(*status*)

autotest.client.shared.log.**is_valid_status**(*status*)

autotest.client.shared.log.**log_and_ignore_errors**(*msg*)
A decorator for wrapping functions in a 'log exception and ignore' try-except block.

autotest.client.shared.log.**record**(*fn*)
Generic method decorator for logging calls under the assumption that return=GOOD, exception=FAIL. The method determines parameters as:

> subdir = self.subdir if it exists, or None operation = "class name"."method name" status = None on GOOD, str(exception) on FAIL

The object using this method must have a job attribute for the logging to actually occur, otherwise the logging will silently fail.

Logging can explicitly be disabled for a call by passing a logged=False parameter

logging_config Module

class autotest.client.shared.logging_config.**AllowBelowSeverity**(*level*)
Bases: logging.Filter

Allows only records less severe than a given level (the opposite of what the normal logging level filtering does.

filter(*record*)

class autotest.client.shared.logging_config.**LoggingConfig**(*use_console=True*)
Bases: object

add_console_handlers()

add_debug_file_handlers(*log_dir*, *log_name=None*)

add_file_handler(*file_path*, *level=10*, *log_dir=None*)

add_stream_handler(*stream*, *level=10*)

configure_logging(*use_console=True*, *verbose=False*)

console_formatter = <logging.Formatter object>

file_formatter = <logging.Formatter object>

classmethod get_autotest_root()

classmethod get_server_log_dir()

classmethod get_timestamped_log_name(*base_name*)

global_level = 10

stderr_level = 40

stdout_level = 20

class autotest.client.shared.logging_config.**TestingConfig**(*use_console=True*)
Bases: *autotest.client.shared.logging_config.LoggingConfig*

add_file_handler(**args*, ***kwargs*)

add_stream_handler(**args*, ***kwargs*)

configure_logging(**kwargs)

logging_manager Module

class autotest.client.shared.logging_manager.**FdRedirectionLoggingManager**
Bases: *autotest.client.shared.logging_manager.LoggingManager*

A simple extension of LoggingManager to use FdRedirectionStreamManagers, so that managed streams have their underlying FDs redirected.

STREAM_MANAGER_CLASS
alias of _FdRedirectionStreamManager

start_logging()

undo_redirect()

class autotest.client.shared.logging_manager.**LoggingFile**(*prefix=''*, *level=10*, *logger=<logging.RootLogger object>*)
Bases: object

File-like object that will receive messages pass them to the logging infrastructure in an appropriate way.

flush()

isatty()

write(*data*)
" Writes data only if it constitutes a whole line. If it's not the case, store it in a buffer and wait until we have a complete line. :param data - Raw data (a string) that will be processed.

writelines(*lines*)
" Writes itertable of lines

Parameters **lines** – An iterable of strings that will be processed.

class autotest.client.shared.logging_manager.**LoggingManager**
Bases: object

Manages a stack of logging configurations, allowing clients to conveniently add and remove logging destinations. Also keeps a list of StreamManagers to easily direct streams into the logging module.

STREAM_MANAGER_CLASS
alias of _StreamManager

logging_config_object = None

manage_stderr()

manage_stdout()

manage_stream(*stream*, *level*, *stream_setter*)
Tells this manager to manage the given stream. All data written to the stream will be directed to the logging module instead. Must be called before start_logging().

Parameters

- **stream** – stream to manage
- **level** – level to log data written to this stream
- **stream_setter** – function to set the stream to a new object

redirect(*filename*)
Redirect output to the specified file

redirect_to_stream(*stream*)
Redirect output to the given stream

restore()
Same as undo_redirect(). For backwards compatibility with fd_stack.

start_logging()
Begin capturing output to the logging module.

stop_logging()
Restore output to its original state.

tee_redirect(*filename*, *level=None*)
Tee output to the specified file

tee_redirect_debug_dir(*debug_dir*, *log_name=None*, *tag=None*)
Tee output to a full new set of debug logs in the given directory.

tee_redirect_to_stream(*stream*)
Tee output to the given stream

undo_redirect()
Undo the last redirection (that hasn't yet been undone).

If any subprocesses have been launched since the redirection was performed, they must have ended by the time this is called. Otherwise, this will hang waiting for the logging subprocess to end.

class autotest.client.shared.logging_manager.**SortingLoggingFile**(*prefix=''*, *level_list=[('ERROR', 40), ('WARN', 30), ('INFO', 20), ('DEBUG', 10)]*, *logger=<logging.RootLogger object>*)
Bases: *autotest.client.shared.logging_manager.LoggingFile*

File-like object that will receive messages and pass them to the logging infrastructure. It decides where to pass each line by applying a regex to it and seeing which level it matched.

autotest.client.shared.logging_manager.**configure_logging**(*logging_config*, ***kwargs*)
Configure the logging module using the specific configuration object, which should be an instance of logging_config.LoggingConfig (usually of a subclass). Any keyword args will be passed to the object's configure_logging() method.

Every entry point should call this method at application startup.

autotest.client.shared.logging_manager.**do_not_report_as_logging_caller**(*func*)
Decorator to annotate functions we will tell logging not to log.

autotest.client.shared.logging_manager.**get_logging_manager**(*manage_stdout_and_stderr=False*, *redirect_fds=False*)
Create a LoggingManager that's managing sys.stdout and sys.stderr.

Every entry point that wants to capture stdout/stderr and/or use LoggingManager to manage a stack of destinations should call this method at application startup.

magic Module

Library used to determine a file MIME type by its magic number, it doesn't have any external dependencies. Based on work of Jason Petrone (jp_py@jsnp.net), adapted to autotest.

Command Line Usage: Running as 'python magic.py file_path' will print a mime string (or just a description) of the file present on file_path.

API Usage: magic.guess_type(file_path) - Returns a description of what the file on path 'file' contains. This function name was chosen due to a similar function on python standard library 'mimetypes'.

@license: GPL v2 :copyright: Jason Petrone (jp_py@jsnp.net) 2000 :copyright: Lucas Meneghel Rodrigues (lmr@redhat.com) 2010 @see: http://www.jsnp.net/code/magic.py

class autotest.client.shared.magic.**MagicLoggingConfig**(*use_console=True*)
Bases: *autotest.client.shared.logging_config.LoggingConfig*

configure_logging(*results_dir=None*, *verbose=False*)

class autotest.client.shared.magic.**MagicTest**(*offset*, *t*, *op*, *value*, *msg*, *mask=None*)
Bases: object

Compile a magic database entry so it can be compared with data read from files.

compare(*data*)
Compare data read from the file with the expected data for this particular mime type register.

Parameters **data** – Data read from the file.

test(*data*)
Compare data read from file with self.value if operator is '='.

Parameters **data** – Data read from the file.

Returns None if no match between data and expected value string. Else, print matching mime type information.

autotest.client.shared.magic.**guess_type**(*filename*)
Guess the mimetype of a file based on its filename.

Parameters **filename** – File name.

Returns Mimetype string or description, when appropriate mime not available.

mail Module

Notification email library.

Aims to replace a bunch of different email module wrappers previously used.

class autotest.client.shared.mail.**EmailNotificationManager**(*module='scheduler'*)
Bases: object

Email notification facility, for use in things like the autotest scheduler.

This facility can use values defined in the autotest settings (global_config.ini) to conveniently send notification emails to the admin of an autotest module.

enqueue_admin(*subject*, *message*)
Enqueue an email to the test grid admin.

enqueue_exception_admin(*reason*)
Enqueue an email containing an exception to the test grid admin.

send(*to_string*, *subject*, *body*)
Send emails to the addresses listed in to_string.

to_string is split into a list which can be delimited by any of: ';', ',', ':' or any whitespace

send_admin(*subject*, *body*)
Send an email to this grid admin.

send_queued_admin()
Send all queued emails to the test grid admin.

set_module(*module*)
Change the name of the module we're notifying for.

autotest.client.shared.mail.**send**(*from_address*, *to_addresses*, *cc_addresses*, *subject*, *body*, *smtp_info*, *html=None*)
Send out an email.

Args: from_address: The email address to put in the "From:" field. to_addresses: Either a single string or an iterable of

strings to put in the "To:" field of the email.

cc_addresses: Either a single string of an iterable of strings to put in the "Cc:" field of the email.

subject: The email subject. body: The body of the email. there's no special

handling of encoding here, so it's safest to stick to 7-bit ASCII text.

smtp_info: Dictionary with SMTP info. html: Optional HTML content of the message.

mock Module

class autotest.client.shared.mock.**Mock**(*spec=None*, *side_effect=None*, *return_value=sentinel.DEFAULT*, *wraps=None*, *name=None*, *spec_set=None*, *parent=None*, *_spec_state=None*, *_new_name=''*, *_new_parent=None*, ***kwargs*)
Bases: autotest.client.shared.mock.CallableMixin, autotest.client.shared.mock.NonCallableM

Create a new *Mock* object. *Mock* takes several optional arguments that specify the behaviour of the Mock object:

- *spec*: This can be either a list of strings or an existing object (a class or instance) that acts as the specification for the mock object. If you pass in an object then a list of strings is formed by calling dir on the object (excluding unsupported magic attributes and methods). Accessing any attribute not in this list will raise an *AttributeError*.

  If *spec* is an object (rather than a list of strings) then *mock.__class__* returns the class of the spec object. This allows mocks to pass *isinstance* tests.

- *spec_set*: A stricter variant of *spec*. If used, attempting to *set* or get an attribute on the mock that isn't on the object passed as *spec_set* will raise an *AttributeError*.

- *side_effect*: A function to be called whenever the Mock is called. See the *side_effect* attribute. Useful for raising exceptions or dynamically changing return values. The function is called with the same arguments as the mock, and unless it returns *DEFAULT*, the return value of this function is used as the return value.

  Alternatively *side_effect* can be an exception class or instance. In this case the exception will be raised when the mock is called.

  If *side_effect* is an iterable then each call to the mock will return the next value from the iterable. If any of the members of the iterable are exceptions they will be raised instead of returned.

- *return_value*: The value returned when the mock is called. By default this is a new Mock (created on first access). See the *return_value* attribute.
- *wraps*: Item for the mock object to wrap. If *wraps* is not None then calling the Mock will pass the call through to the wrapped object (returning the real result). Attribute access on the mock will return a Mock object that wraps the corresponding attribute of the wrapped object (so attempting to access an attribute that doesn't exist will raise an *AttributeError*).

  If the mock has an explicit *return_value* set then calls are not passed to the wrapped object and the *return_value* is returned instead.
- *name*: If the mock has a name then it will be used in the repr of the mock. This can be useful for debugging. The name is propagated to child mocks.

Mocks can also be called with arbitrary keyword arguments. These will be used to set attributes on the mock after it is created.

class autotest.client.shared.mock.**MagicMock**(**args*, ***kw*)
Bases: autotest.client.shared.mock.MagicMixin, *autotest.client.shared.mock.Mock*

MagicMock is a subclass of Mock with default implementations of most of the magic methods. You can use MagicMock without having to configure the magic methods yourself.

If you use the *spec* or *spec_set* arguments then *only* magic methods that exist in the spec will be created.

Attributes and the return value of a *MagicMock* will also be *MagicMocks*.

mock_add_spec(*spec*, *spec_set=False*)
Add a spec to a mock. *spec* can either be an object or a list of strings. Only attributes on the *spec* can be fetched as attributes from the mock.

If *spec_set* is True then only attributes on the spec can be set.

autotest.client.shared.mock.**patch**(*target*, *new=sentinel.DEFAULT*, *spec=None*, *create=False*, *spec_set=None*, *autospec=None*, *new_callable=None*, ***kwargs*)
patch acts as a function decorator, class decorator or a context manager. Inside the body of the function or with statement, the *target* is patched with a *new* object. When the function/with statement exits the patch is undone.

If *new* is omitted, then the target is replaced with a *MagicMock*. If *patch* is used as a decorator and *new* is omitted, the created mock is passed in as an extra argument to the decorated function. If *patch* is used as a context manager the created mock is returned by the context manager.

target should be a string in the form *'package.module.ClassName'*. The *target* is imported and the specified object replaced with the *new* object, so the *target* must be importable from the environment you are calling *patch* from. The target is imported when the decorated function is executed, not at decoration time.

The *spec* and *spec_set* keyword arguments are passed to the *MagicMock* if patch is creating one for you.

In addition you can pass *spec=True* or *spec_set=True*, which causes patch to pass in the object being mocked as the spec/spec_set object.

new_callable allows you to specify a different class, or callable object, that will be called to create the *new* object. By default *MagicMock* is used.

A more powerful form of *spec* is *autospec*. If you set *autospec=True* then the mock with be created with a spec from the object being replaced. All attributes of the mock will also have the spec of the corresponding attribute of the object being replaced. Methods and functions being mocked will have their arguments checked and will raise a *TypeError* if they are called with the wrong signature. For mocks replacing a class, their return value (the 'instance') will have the same spec as the class.

Instead of *autospec=True* you can pass *autospec=some_object* to use an arbitrary object as the spec instead of the one being replaced.

By default *patch* will fail to replace attributes that don't exist. If you pass in *create=True*, and the attribute doesn't exist, patch will create the attribute for you when the patched function is called, and delete it again afterwards. This is useful for writing tests against attributes that your production code creates at runtime. It is off by by default because it can be dangerous. With it switched on you can write passing tests against APIs that don't actually exist!

Patch can be used as a *TestCase* class decorator. It works by decorating each test method in the class. This reduces the boilerplate code when your test methods share a common patchings set. *patch* finds tests by looking for method names that start with *patch.TEST_PREFIX*. By default this is *test*, which matches the way *unittest* finds tests. You can specify an alternative prefix by setting *patch.TEST_PREFIX*.

Patch can be used as a context manager, with the with statement. Here the patching applies to the indented block after the with statement. If you use "as" then the patched object will be bound to the name after the "as"; very useful if *patch* is creating a mock object for you.

patch takes arbitrary keyword arguments. These will be passed to the *Mock* (or *new_callable*) on construction.

patch.dict(...), *patch.multiple(...)* and *patch.object(...)* are available for alternate use-cases.

autotest.client.shared.mock.**call**

A tuple for holding the results of a call to a mock, either in the form *(args, kwargs)* or *(name, args, kwargs)*.

If args or kwargs are empty then a call tuple will compare equal to a tuple without those values. This makes comparisons less verbose:

```
_Call(('name', (), {})) == ('name',)
_Call(('name', (1,), {})) == ('name', (1,))
_Call(((), {'a': 'b'})) == ({'a': 'b'},)
```

The *_Call* object provides a useful shortcut for comparing with call:

```
_Call(((1, 2), {'a': 3})) == call(1, 2, a=3)
_Call(('foo', (1, 2), {'a': 3})) == call.foo(1, 2, a=3)
```

If the _Call has no name then it will match any name.

autotest.client.shared.mock.**create_autospec**(*spec*, *spec_set=False*, *instance=False*, *_parent=None*, *_name=None*, ***kwargs*)

Create a mock object using another object as a spec. Attributes on the mock will use the corresponding attribute on the *spec* object as their spec.

Functions or methods being mocked will have their arguments checked to check that they are called with the correct signature.

If *spec_set* is True then attempting to set attributes that don't exist on the spec object will raise an *AttributeError*.

If a class is used as a spec then the return value of the mock (the instance of the class) will have the same spec. You can use a class as the spec for an instance object by passing *instance=True*. The returned mock will only be callable if instances of the mock are callable.

create_autospec also takes arbitrary keyword arguments that are passed to the constructor of the created mock.

class autotest.client.shared.mock.**NonCallableMock**(*spec=None*, *wraps=None*, *name=None*, *spec_set=None*, *parent=None*, *_spec_state=None*, *_new_name=''*, *_new_parent=None*, ***kwargs*)

Bases: autotest.client.shared.mock.Base

A non-callable version of *Mock*

assert_any_call(**args*, ***kwargs*)

assert the mock has been called with the specified arguments.

The assert passes if the mock has *ever* been called, unlike *assert_called_with* and *assert_called_once_with* that only pass if the call is the most recent one.

assert_called_once_with(*_mock_self*, **args*, ***kwargs*)
assert that the mock was called exactly once and with the specified arguments.

assert_called_with(*_mock_self*, **args*, ***kwargs*)
assert that the mock was called with the specified arguments.

Raises an AssertionError if the args and keyword args passed in are different to the last call to the mock.

assert_has_calls(*calls*, *any_order=False*)
assert the mock has been called with the specified calls. The *mock_calls* list is checked for the calls.

If *any_order* is False (the default) then the calls must be sequential. There can be extra calls before or after the specified calls.

If *any_order* is True then the calls can be in any order, but they must all appear in *mock_calls*.

attach_mock(*mock*, *attribute*)
Attach a mock as an attribute of this one, replacing its name and parent. Calls to the attached mock will be recorded in the *method_calls* and *mock_calls* attributes of this one.

call_args

call_args_list

call_count

called

configure_mock(***kwargs*)
Set attributes on the mock through keyword arguments.

Attributes plus return values and side effects can be set on child mocks using standard dot notation and unpacking a dictionary in the method call:

```
>>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> mock.configure_mock(**attrs)
```

mock_add_spec(*spec*, *spec_set=False*)
Add a spec to a mock. *spec* can either be an object or a list of strings. Only attributes on the *spec* can be fetched as attributes from the mock.

If *spec_set* is True then only attributes on the spec can be set.

mock_calls

reset_mock()
Restore the mock object to its initial state.

return_value

side_effect

class autotest.client.shared.mock.**NonCallableMagicMock**(**args*, ***kw*)
Bases: autotest.client.shared.mock.MagicMixin, *autotest.client.shared.mock.NonCallableMock*

A version of *MagicMock* that isn't callable.

mock_add_spec(*spec*, *spec_set=False*)
Add a spec to a mock. *spec* can either be an object or a list of strings. Only attributes on the *spec* can be fetched as attributes from the mock.

If *spec_set* is True then only attributes on the spec can be set.

autotest.client.shared.mock.**mock_open**(*mock=None*, *read_data=''*)
A helper function to create a mock to replace the use of *open*. It works for *open* called directly or used as a context manager.

The *mock* argument is the mock object to configure. If *None* (the default) then a *MagicMock* will be created for you, with the API limited to methods or attributes available on standard file handles.

read_data is a string for the *read* method of the file handle to return. This is an empty string by default.

class autotest.client.shared.mock.**PropertyMock**(*spec=None*, *side_effect=None*, *return_value=sentinel.DEFAULT*, *wraps=None*, *name=None*, *spec_set=None*, *parent=None*, *_spec_state=None*, *_new_name=''*, *_new_parent=None*, ***kwargs*)
Bases: *autotest.client.shared.mock.Mock*

A mock intended to be used as a property, or other descriptor, on a class. *PropertyMock* provides *__get__* and *__set__* methods so you can specify a return value when it is fetched.

Fetching a *PropertyMock* instance from an object calls the mock, with no args. Setting it calls the mock with the value being set.

openvswitch Module

class autotest.client.shared.openvswitch.**OpenVSwitch**(*tmpdir*, *db_path=None*, *db_socket=None*, *db_pidfile=None*, *ovs_pidfile=None*, *db-schema=None*, *install_prefix=None*)
Bases: *autotest.client.shared.openvswitch.OpenVSwitchSystem*

OpenVSwtich class.

clean()

init_db()

init_new()
Create new dbfile without any configuration.

start_ovs_vswitchd()

class autotest.client.shared.openvswitch.**OpenVSwitchControl**
Bases: object

Class select the best matches control class for installed version of OpenVSwitch.

OpenVSwtich parameters are described in man ovs-vswitchd.conf.db

add_br(*br_name*)

add_port(*br_name*, *port_name*)

add_port_tag(*port_name*, *tag*)

add_port_trunk(*port_name*, *trunk*)

br_exist(*br_name*)

check_port_in_br(*br_name*, *port_name*)

static convert_version_to_int(*version*)

Parameters **version** – (int) Converted from version string 1.4.0 => int 140

del_br(*br_name*)

del_port(*br_name*, *port_name*)

classmethod get_version()
Get version of installed OpenVSwtich.

Returns Version of OpenVSwtich.

list_br()

set_vlanmode(*port_name*, *vlan_mode*)

status()

class autotest.client.shared.openvswitch.**OpenVSwitchControlCli**
Bases: *autotest.client.shared.openvswitch.OpenVSwitchControl*, *autotest.client.shared.utils.VersionableClass*

Class select the best matches control class for installed version of OpenVSwitch.

class autotest.client.shared.openvswitch.**OpenVSwitchControlCli_140**
Bases: *autotest.client.shared.openvswitch.OpenVSwitchControlCli*, *autotest.client.shared.utils.VersionableClass*

Don't use this class directly. This class is automatically selected by OpenVSwitchControl.

add_br(*br_name*)

add_fake_br(*br_name*, *parent*, *vlan*)

add_port(*br_name*, *port_name*)

add_port_tag(*port_name*, *tag*)

add_port_trunk(*port_name*, *trunk*)

Parameters **trunk** – list of vlans id.

br_exist(*br_name*)

del_br(*br_name*)

del_port(*br_name*, *port_name*)

classmethod is_right_version(*version*)
Check condition for select control class.

Parameters **version** – version of OpenVSwtich

list_br()

list_ports(*br_name*)

ovs_vsctl(*parmas*, *ignore_status=False*)

port_to_br(*port_name*)
Return bridge which contain port.

Parameters **port_name** – Name of port.

Returns Bridge name or None if there is no bridge which contain port.

set_vlanmode(*port_name*, *vlan_mode*)

status()

class autotest.client.shared.openvswitch.**OpenVSwitchControlDB**
Bases: *autotest.client.shared.openvswitch.OpenVSwitchControl*, *autotest.client.shared.utils.VersionableClass*

Class select the best matches control class for installed version of OpenVSwitch.

class autotest.client.shared.openvswitch.**OpenVSwitchControlDB_140**
Bases: *autotest.client.shared.openvswitch.OpenVSwitchControlDB*, *autotest.client.shared.utils.VersionableClass*

Don't use this class directly. This class is automatically selected by OpenVSwitchControl.

classmethod is_right_version(*version*)
Check condition for select control class.

Parameters **version** – version of OpenVSwtich

class autotest.client.shared.openvswitch.**OpenVSwitchSystem**(*db_path=None*, *db_socket=None*, *db_pidfile=None*, *ovs_pidfile=None*, *dbschema=None*, *install_prefix=None*)
Bases: *autotest.client.shared.openvswitch.OpenVSwitchControlCli*, *autotest.client.shared.openvswitch.OpenVSwitchControlDB*

OpenVSwtich class.

check()

check_db_daemon()
Check if OVS daemon is started correctly.

check_db_file()
Check if db_file exists.

check_db_socket()
Check if db socket exists.

check_switch_daemon()
Check if OVS daemon is started correctly.

clean()
Empty cleanup function

init_system()
Create new dbfile without any configuration.

is_installed()
Check if OpenVSwitch is already installed in system on default places.

Returns Version of OpenVSwtich.

class autotest.client.shared.openvswitch.**ServiceManager**
Bases: *autotest.client.shared.openvswitch.ServiceManagerInterface*

class autotest.client.shared.openvswitch.**ServiceManagerInterface**
Bases: *autotest.client.shared.utils.VersionableClass*

classmethod get_version()
Get version of ServiceManager. :return: Version of ServiceManager.

restart(*service_name*)

start(*service_name*)

status(*service_name*)

stop(*service_name*)

class autotest.client.shared.openvswitch.**ServiceManagerSystemD**
Bases: *autotest.client.shared.openvswitch.ServiceManagerInterface*, *autotest.client.shared.utils.VersionableClass*

classmethod is_right_version(*version*)

restart(*service_name*)

start(*service_name*)

status(*service_name*)

stop(*service_name*)

class autotest.client.shared.openvswitch.**ServiceManagerSysvinit**
Bases: *autotest.client.shared.openvswitch.ServiceManagerInterface*, *autotest.client.shared.utils.VersionableClass*

classmethod is_right_version(*version*)

restart(*service_name*)

start(*service_name*)

stop(*service_name*)

packages Module

class autotest.client.shared.packages.**PackageManager**(*pkgmgr_dir*, *hostname=None*, *repo_urls=None*, *upload_paths=None*, *do_locking=True*, *run_function=<function run>*, *run_function_args=[]*, *run_function_dargs={}*)
Bases: *autotest.client.shared.base_packages.BasePackageManager*

pexpect Module

Pexpect is a Python module for spawning child applications and controlling them automatically. Pexpect can be used for automating interactive applications such as ssh, ftp, passwd, telnet, etc. It can be used to a automate setup scripts for duplicating software package installations on different servers. It can be used for automated software testing. Pexpect is in the spirit of Don Libes' Expect, but Pexpect is pure Python. Other Expect-like modules for Python require TCL and Expect or require C extensions to be compiled. Pexpect does not use C, Expect, or TCL extensions. It should work on any platform that supports the standard Python pty module. The Pexpect interface focuses on ease of use so that simple tasks are easy.

There are two main interfaces to Pexpect – the function, run() and the class, spawn. You can call the run() function to execute a command and return the output. This is a handy replacement for os.system().

For example:

```
pexpect.run('ls -la')
```

The more powerful interface is the spawn class. You can use this to spawn an external child command and then interact with the child by sending lines and expecting responses.

For example:

```
child = pexpect.spawn('scp foo myname@host.example.com:.')
child.expect ('Password:')
child.sendline (mypassword)
```

This works even for commands that ask for passwords or other input outside of the normal stdio streams.

Credits: Noah Spurrier, Richard Holden, Marco Molteni, Kimberley Burchett, Robert Stone, Hartmut Goebel, Chad Schroeder, Erick Tryzelaar, Dave Kirby, Ids vander Molen, George Todd, Noel Taylor, Nicolas D. Cesar, Alexander Gattin, Geoffrey Marshall, Francisco Lourenco, Glen Mabey, Karthik Gurusamy, Fernando Perez, Corey Minyard, Jon Cohen, Guillaume Chazarain, Andrew Ryan, Nick Craig-Wood, Andrew Stone, Jorgen Grahn (Let me know if I forgot anyone.)

Free, open source, and all that good stuff.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Pexpect Copyright (c) 2008 Noah Spurrier http://pexpect.sourceforge.net/

$Id: pexpect.py 507 2007-12-27 02:40:52Z noah $

exception autotest.client.shared.pexpect.**ExceptionPexpect**(*value*)
Bases: exceptions.Exception

Base class for all exceptions raised by this module.

get_trace()
This returns an abbreviated stack trace with lines that only concern the caller. In other words, the stack trace inside the Pexpect module is not included.

exception autotest.client.shared.pexpect.**EOF**(*value*)
Bases: *autotest.client.shared.pexpect.ExceptionPexpect*

Raised when EOF is read from a child. This usually means the child has exited.

exception autotest.client.shared.pexpect.**TIMEOUT**(*value*)
Bases: *autotest.client.shared.pexpect.ExceptionPexpect*

Raised when a read time exceeds the timeout.

class autotest.client.shared.pexpect.**spawn**(*command*, *args=[]*, *timeout=30*, *maxread=2000*, *searchwindowsize=None*, *logfile=None*, *cwd=None*, *env=None*)
Bases: object

This is the main class interface for Pexpect. Use this class to start and control child applications.

close(*force=True*)
This closes the connection with the child application. Note that calling close() more than once is valid.

This emulates standard Python behavior with files. Set force to True if you want to make sure that the child is terminated (SIGKILL is sent if the child ignores SIGHUP and SIGINT).

compile_pattern_list(*patterns*)

This compiles a pattern-string or a list of pattern-strings. Patterns must be a StringType, EOF, TIMEOUT, SRE_Pattern, or a list of those. Patterns may also be None which results in an empty list (you might do this if waiting for an EOF or TIMEOUT condition without expecting any pattern).

This is used by expect() when calling expect_list(). Thus expect() is nothing more than:

```
cpl = self.compile_pattern_list(pl)
return self.expect_list(cpl, timeout)
```

If you are using expect() within a loop it may be more efficient to compile the patterns first and then call expect_list(). This avoid calls in a loop to compile_pattern_list():

```
cpl = self.compile_pattern_list(my_pattern)
while some_condition:
   ...
   i = self.expect_list(clp, timeout)
   ...
```

eof()

This returns True if the EOF exception was ever raised.

expect(*pattern*, *timeout=-1*, *searchwindowsize=None*)

This seeks through the stream until a pattern is matched. The pattern is overloaded and may take several types. The pattern can be a StringType, EOF, a compiled re, or a list of any of those types. Strings will be compiled to re types. This returns the index into the pattern list. If the pattern was not a list this returns index 0 on a successful match. This may raise exceptions for EOF or TIMEOUT. To avoid the EOF or TIMEOUT exceptions add EOF or TIMEOUT to the pattern list. That will cause expect to match an EOF or TIMEOUT condition instead of raising an exception.

If you pass a list of patterns and more than one matches, the first match in the stream is chosen. If more than one pattern matches at that point, the leftmost in the pattern list is chosen. For example:

```
# the input is 'foobar'
index = p.expect (['bar', 'foo', 'foobar'])
# returns 1 ('foo') even though 'foobar' is a "better" match
```

Please note, however, that buffering can affect this behavior, since input arrives in unpredictable chunks. For example:

```
# the input is 'foobar'
index = p.expect (['foobar', 'foo'])
# returns 0 ('foobar') if all input is available at once,
# but returs 1 ('foo') if parts of the final 'bar' arrive late
```

After a match is found the instance attributes 'before', 'after' and 'match' will be set. You can see all the data read before the match in 'before'. You can see the data that was matched in 'after'. The re.MatchObject used in the re match will be in 'match'. If an error occurred then 'before' will be set to all the data read so far and 'after' and 'match' will be None.

If timeout is -1 then timeout will be set to the self.timeout value.

A list entry may be EOF or TIMEOUT instead of a string. This will catch these exceptions and return the index of the list entry instead of raising the exception. The attribute 'after' will be set to the exception type. The attribute 'match' will be None. This allows you to write code like this:

```
index = p.expect (['good', 'bad', pexpect.EOF, pexpect.TIMEOUT])
if index == 0:
    do_something()
elif index == 1:
    do_something_else()
elif index == 2:
    do_some_other_thing()
elif index == 3:
    do_something_completely_different()
```

instead of code like this:

```
try:
    index = p.expect (['good', 'bad'])
    if index == 0:
        do_something()
    elif index == 1:
        do_something_else()
except EOF:
    do_some_other_thing()
except TIMEOUT:
    do_something_completely_different()
```

These two forms are equivalent. It all depends on what you want. You can also just expect the EOF if you are waiting for all output of a child to finish. For example:

```
p = pexpect.spawn('/bin/ls')
p.expect (pexpect.EOF)
print p.before
```

If you are trying to optimize for speed then see expect_list().

expect_exact (*pattern_list*, *timeout=-1*, *searchwindowsize=-1*)
This is similar to expect(), but uses plain string matching instead of compiled regular expressions in 'pattern_list'. The 'pattern_list' may be a string; a list or other sequence of strings; or TIMEOUT and EOF.

This call might be faster than expect() for two reasons: string searching is faster than RE matching and it is possible to limit the search to just the end of the input buffer.

This method is also useful when you don't want to have to worry about escaping regular expression characters that you want to match.

expect_list (*pattern_list*, *timeout=-1*, *searchwindowsize=-1*)
This takes a list of compiled regular expressions and returns the index into the pattern_list that matched the child output. The list may also contain EOF or TIMEOUT (which are not compiled regular expressions). This method is similar to the expect() method except that expect_list() does not recompile the pattern list on every call. This may help if you are trying to optimize for speed, otherwise just use the expect() method. This is called by expect(). If timeout==-1 then the self.timeout value is used. If searchwindowsize==-1 then the self.searchwindowsize value is used.

expect_loop (*searcher*, *timeout=-1*, *searchwindowsize=-1*)
This is the common loop used inside expect. The 'searcher' should be an instance of searcher_re or searcher_string, which describes how and what to search for in the input.

See expect() for other arguments, return value and exceptions.

fileno ()
This returns the file descriptor of the pty for the child.

flush ()
This does nothing. It is here to support the interface for a File-like object.

getecho()
This returns the terminal echo mode. This returns True if echo is on or False if echo is off. Child applications that are expecting you to enter a password often set ECHO False. See waitnoecho().

getwinsize()
This returns the terminal window size of the child tty. The return value is a tuple of (rows, cols).

interact(*escape_character='\x1d'*, *input_filter=None*, *output_filter=None*)
This gives control of the child process to the interactive user (the human at the keyboard). Keystrokes are sent to the child process, and the stdout and stderr output of the child process is printed. This simply echos the child stdout and child stderr to the real stdout and it echos the real stdin to the child stdin. When the user types the escape_character this method will stop. The default for escape_character is ^]. This should not be confused with ASCII 27 – the ESC character. ASCII 29 was chosen for historical merit because this is the character used by 'telnet' as the escape character. The escape_character will not be sent to the child process.

You may pass in optional input and output filter functions. These functions should take a string and return a string. The output_filter will be passed all the output from the child process. The input_filter will be passed all the keyboard input from the user. The input_filter is run BEFORE the check for the escape_character.

Note that if you change the window size of the parent the SIGWINCH signal will not be passed through to the child. If you want the child window size to change when the parent's window size changes then do something like the following example:

```
import pexpect, struct, fcntl, termios, signal, sys
def sigwinch_passthrough (sig, data):
    s = struct.pack("HHHH", 0, 0, 0, 0)
    a = struct.unpack('hhhh', fcntl.ioctl(sys.stdout.fileno(), termios.TIOCGWINSZ , s))
    global p
    p.setwinsize(a[0],a[1])
p = pexpect.spawn('/bin/bash') # Note this is global and used in sigwinch_passthrough.
signal.signal(signal.SIGWINCH, sigwinch_passthrough)
p.interact()
```

isalive()
This tests if the child process is running or not. This is non-blocking. If the child was terminated then this will read the exitstatus or signalstatus of the child. This returns True if the child process appears to be running or False if not. It can take literally SECONDS for Solaris to return the right status.

isatty()
This returns True if the file descriptor is open and connected to a tty(-like) device, else False.

kill(*sig*)
This sends the given signal to the child application. In keeping with UNIX tradition it has a misleading name. It does not necessarily kill the child unless you send the right signal.

next()
This is to support iterators over a file-like object.

read(*size=-1*)
This reads at most "size" bytes from the file (less if the read hits EOF before obtaining size bytes). If the size argument is negative or omitted, read all data until EOF is reached. The bytes are returned as a string object. An empty string is returned when EOF is encountered immediately.

read_nonblocking(*size=1*, *timeout=-1*)
This reads at most size characters from the child application. It includes a timeout. If the read does not complete within the timeout period then a TIMEOUT exception is raised. If the end of file is read then an EOF exception will be raised. If a log file was set using setlog() then all data will also be written to the log file.

If timeout is None then the read may block indefinitely. If timeout is -1 then the self.timeout value is used. If timeout is 0 then the child is polled and if there was no data immediately ready then this will raise a TIMEOUT exception.

The timeout refers only to the amount of time to read at least one character. This is not effected by the 'size' parameter, so if you call read_nonblocking(size=100, timeout=30) and only one character is available right away then one character will be returned immediately. It will not wait for 30 seconds for another 99 characters to come in.

This is a wrapper around os.read(). It uses select.select() to implement the timeout.

readline(*size=-1*)
This reads and returns one entire line. A trailing newline is kept in the string, but may be absent when a file ends with an incomplete line. Note: This readline() looks for a rn pair even on UNIX because this is what the pseudo tty device returns. So contrary to what you may expect you will receive the newline as rn. An empty string is returned when EOF is hit immediately. Currently, the size argument is mostly ignored, so this behavior is not standard for a file-like object. If size is 0 then an empty string is returned.

readlines(*sizehint=-1*)
This reads until EOF using readline() and returns a list containing the lines thus read. The optional "size-hint" argument is ignored.

send(*s*)
This sends a string to the child process. This returns the number of bytes written. If a log file was set then the data is also written to the log.

sendcontrol(*char*)
This sends a control character to the child such as Ctrl-C or Ctrl-D. For example, to send a Ctrl-G (ASCII 7):

```
child.sendcontrol('g')
```

See also, sendintr() and sendeof().

sendeof()
This sends an EOF to the child. This sends a character which causes the pending parent output buffer to be sent to the waiting child program without waiting for end-of-line. If it is the first character of the line, the read() in the user program returns 0, which signifies end-of-file. This means to work as expected a sendeof() has to be called at the beginning of a line. This method does not send a newline. It is the responsibility of the caller to ensure the eof is sent at the beginning of a line.

sendintr()
This sends a SIGINT to the child. It does not require the SIGINT to be the first character on a line.

sendline(*s=''*)
This is like send(), but it adds a line feed (os.linesep). This returns the number of bytes written.

setecho(*state*)
This sets the terminal echo mode on or off. Note that anything the child sent before the echo will be lost, so you should be sure that your input buffer is empty before you call setecho(). For example, the following will work as expected:

```
p = pexpect.spawn('cat')
p.sendline ('1234') # We will see this twice (once from tty echo and again from cat).
p.expect (['1234'])
p.expect (['1234'])
p.setecho(False) # Turn off tty echo
p.sendline ('abcd') # We will set this only once (echoed by cat).
p.sendline ('wxyz') # We will set this only once (echoed by cat)
```

```
        p.expect (['abcd'])
        p.expect (['wxyz'])
```

The following WILL NOT WORK because the lines sent before the setecho will be lost:

```
        p = pexpect.spawn('cat')
        p.sendline ('1234') # We will see this twice (once from tty echo and again from cat).
        p.setecho(False) # Turn off tty echo
        p.sendline ('abcd') # We will set this only once (echoed by cat).
        p.sendline ('wxyz') # We will set this only once (echoed by cat)
        p.expect (['1234'])
        p.expect (['1234'])
        p.expect (['abcd'])
        p.expect (['wxyz'])
```

setlog(*fileobject*)
This method is no longer supported or allowed.

setmaxread(*maxread*)
This method is no longer supported or allowed. I don't like getters and setters without a good reason.

setwinsize(*r*, *c*)
This sets the terminal window size of the child tty. This will cause a SIGWINCH signal to be sent to the child. This does not change the physical window size. It changes the size reported to TTY-aware applications like vi or curses – applications that respond to the SIGWINCH signal.

terminate(*force=False*)
This forces a child process to terminate. It starts nicely with SIGHUP and SIGINT. If "force" is True then moves onto SIGKILL. This returns True if the child was terminated. This returns False if the child could not be terminated.

wait()
This waits until the child exits. This is a blocking call. This will not read any data from the child, so this will block forever if the child has unread output and has terminated. In other words, the child may have printed output then called exit(); but, technically, the child is still alive until its output is read.

waitnoecho(*timeout=-1*)
This waits until the terminal ECHO flag is set False. This returns True if the echo mode is off. This returns False if the ECHO flag was not set False before the timeout. This can be used to detect when the child is waiting for a password. Usually a child application will turn off echo mode when it is waiting for the user to enter a password. For example, instead of expecting the "password:" prompt you can wait for the child to set ECHO off:

```
        p = pexpect.spawn ('ssh user@example.com')
        p.waitnoecho()
        p.sendline(mypassword)
```

If timeout is None then this method to block forever until ECHO flag is False.

write(*s*)
This is similar to send() except that there is no return value.

writelines(*sequence*)
This calls write() for each element in the sequence. The sequence can be any iterable object producing strings, typically a list of strings. This does not add line separators There is no return value.

autotest.client.shared.pexpect.**run**(*command*, *timeout=-1*, *withexitstatus=False*, *events=None*, *extra_args=None*, *logfile=None*, *cwd=None*, *env=None*)
This function runs the given command; waits for it to finish; then returns all output as a string. STDERR is

included in output. If the full path to the command is not given then the path is searched.

Note that lines are terminated by CR/LF (rn) combination even on UNIX-like systems because this is the standard for pseudo ttys. If you set 'withexitstatus' to true, then run will return a tuple of (command_output, exitstatus). If 'withexitstatus' is false then this returns just command_output.

The run() function can often be used instead of creating a spawn instance. For example, the following code uses spawn:

```
from pexpect import *
child = spawn('scp foo myname@host.example.com:.')
child.expect ('(?i)password')
child.sendline (mypassword)
```

The previous code can be replace with the following:

```
from pexpect import *
run ('scp foo myname@host.example.com:.', events={'(?i)password': mypassword})
```

Start the apache daemon on the local machine:

```
from pexpect import *
run ("/usr/local/apache/bin/apachectl start")
```

Check in a file using SVN:

```
from pexpect import *
run ("svn ci -m 'automatic commit' my_file.py")
```

Run a command and capture exit status:

```
from pexpect import *
(command_output, exitstatus) = run ('ls -l /bin', withexitstatus=1)
```

The following will run SSH and execute 'ls -l' on the remote machine. The password 'secret' will be sent if the '(?i)password' pattern is ever seen:

```
run ("ssh username@machine.example.com 'ls -l'", events={'(?i)password':'secret\n'})
```

This will start mencoder to rip a video from DVD. This will also display progress ticks every 5 seconds as it runs. For example:

```
from pexpect import *
def print_ticks(d):
    print d['event_count'],
run ("mencoder dvd://1 -o video.avi -oac copy -ovc copy", events={TIMEOUT:print_ticks}, timeout=
```

The 'events' argument should be a dictionary of patterns and responses. Whenever one of the patterns is seen in the command out run() will send the associated response string. Note that you should put newlines in your string if Enter is necessary. The responses may also contain callback functions. Any callback is function that takes a dictionary as an argument. The dictionary contains all the locals from the run() function, so you can access the child spawn object or any other variable defined in run() (event_count, child, and extra_args are the most useful). A callback may return True to stop the current run process otherwise run() continues until the next event. A callback may also return a string which will be sent to the child. 'extra_args' is not used by directly run(). It provides a way to pass data to a callback function through run() through the locals dictionary passed to a callback.

`autotest.client.shared.pexpect.`**`which`**(*filename*)

This takes a given filename; tries to find it in the environment path; then checks if it is executable. This returns the full path to the filename if found and executable. Otherwise this returns None.

autotest.client.shared.pexpect.**split_command_line**(*command_line*)
This splits a command line into a list of arguments. It splits arguments on spaces, but handles embedded quotes, doublequotes, and escaped characters. It's impossible to do this with a regular expression, so I wrote a little state machine to parse the command line.

pidfile Module

class autotest.client.shared.pidfile.**PidFileManager**(*label*, *results_dir*)
Bases: object

close_file(*exit_code*, *signal_code=0*)

open_file()

profiler_manager Module

exception autotest.client.shared.profiler_manager.**ProfilerNotPresentError**(*name*, **args*, ***dargs*)
Bases: *autotest.client.shared.error.JobError*

class autotest.client.shared.profiler_manager.**profiler_manager**(*job*)
Bases: object

active()
Returns True if profilers are present and started, False otherwise

add(*profiler*, **args*, ***dargs*)
Add a profiler

before_start(*test*)
Override to do any setup needed before actually starting the profilers (this function is called before calling test.before_run_once() and profilers.start() in a profiled run).

current_profilers()
Returns a set of the currently enabled profilers

delete(*profiler*)
Remove a profiler

load_profiler(*profiler*, *args*, *dargs*)
Given a name and args, loads a profiler, initializes it with the required arguments, and returns an instance of it. Raises a ProfilerNotPresentError if the module isn't found.

only()
Returns True if job is supposed to be run only with profiling turned on, False otherwise

present()
Indicates if any profilers are enabled

report(*test*)
Report on all enabled profilers

set_only(*value*)
Changes the flag which determines whether or not the job is to be run without profilers at all

start(*test*)
Start all enabled profilers

stop(*test*)
Stop all enabled profilers

progressbar Module

Basic text progress bar without fancy curses features

class autotest.client.shared.progressbar.**ProgressBar**(*minimum=0*, *maximum=100*, *width=77*, *title=''*)
Displays interactively the progress of a given task

Inspired/adapted from code.activestate.com recipe #168639

DEFAULT_WIDTH = 77

get_screen_text()
Builds the actual progress bar text

increment(*increment*, *update_screen=True*)
Increments the current amount value

update(*amount*, *update_screen=True*)
Performs sanity checks and update the current amount

update_screen()
Prints the updated text to the screen

pxssh Module

This class extends pexpect.spawn to specialize setting up SSH connections. This adds methods for login, logout, and expecting the shell prompt.

$Id: pxssh.py 487 2007-08-29 22:33:29Z noah $

exception autotest.client.shared.pxssh.**ExceptionPxssh**(*value*)
Bases: *autotest.client.shared.pexpect.ExceptionPexpect*

Raised for pxssh exceptions.

class autotest.client.shared.pxssh.**pxssh**(*timeout=30*, *maxread=2000*, *searchwindowsize=None*, *logfile=None*, *cwd=None*, *env=None*)
Bases: *autotest.client.shared.pexpect.spawn*

This class extends pexpect.spawn to specialize setting up SSH connections. This adds methods for login, logout, and expecting the shell prompt. It does various tricky things to handle many situations in the SSH login process. For example, if the session is your first login, then pxssh automatically accepts the remote certificate; or if you have public key authentication setup then pxssh won't wait for the password prompt.

pxssh uses the shell prompt to synchronize output from the remote host. In order to make this more robust it sets the shell prompt to something more unique than just $ or #. This should work on most Borne/Bash or Csh style shells.

Example that runs a few commands on a remote server and prints the result:

```
import pxssh
import getpass
try:
    s = pxssh.pxssh()
    hostname = raw_input('hostname: ')
    username = raw_input('username: ')
    password = getpass.getpass('password: ')
```

```
        s.login (hostname, username, password)
        s.sendline ('uptime')  # run a command
        s.prompt()             # match the prompt
        print s.before         # print everything before the prompt.
        s.sendline ('ls -l')
        s.prompt()
        print s.before
        s.sendline ('df')
        s.prompt()
        print s.before
        s.logout()
    except pxssh.ExceptionPxssh, e:
        print "pxssh failed on login."
        print str(e)
```

Note that if you have ssh-agent running while doing development with pxssh then this can lead to a lot of confusion. Many X display managers (xdm, gdm, kdm, etc.) will automatically start a GUI agent. You may see a GUI dialog box popup asking for a password during development. You should turn off any key agents during testing. The 'force_password' attribute will turn off public key authentication. This will only work if the remote SSH server is configured to allow password logins. Example of using 'force_password' attribute:

```
    s = pxssh.pxssh()
    s.force_password = True
    hostname = raw_input('hostname: ')
    username = raw_input('username: ')
    password = getpass.getpass('password: ')
    s.login (hostname, username, password)
```

levenshtein_distance(*a*, *b*)
This calculates the Levenshtein distance between a and b.

login(*server*, *username*, *password=''*, *terminal_type='ansi'*, *original_prompt='[#$]'*, *login_timeout=10*, *port=None*, *auto_prompt_reset=True*)
This logs the user into the given server. It uses the 'original_prompt' to try to find the prompt right after login. When it finds the prompt it immediately tries to reset the prompt to something more easily matched. The default 'original_prompt' is very optimistic and is easily fooled. It's more reliable to try to match the original prompt as exactly as possible to prevent false matches by server strings such as the "Message Of The Day". On many systems you can disable the MOTD on the remote server by creating a zero-length file called "~/.hushlogin" on the remote server. If a prompt cannot be found then this will not necessarily cause the login to fail. In the case of a timeout when looking for the prompt we assume that the original prompt was so weird that we could not match it, so we use a few tricks to guess when we have reached the prompt. Then we hope for the best and blindly try to reset the prompt to something more unique. If that fails then login() raises an ExceptionPxssh exception.

In some situations it is not possible or desirable to reset the original prompt. In this case, set 'auto_prompt_reset' to False to inhibit setting the prompt to the UNIQUE_PROMPT. Remember that pxssh uses a unique prompt in the prompt() method. If the original prompt is not reset then this will disable the prompt() method unless you manually set the PROMPT attribute.

logout()
This sends exit to the remote shell. If there are stopped jobs then this automatically sends exit twice.

prompt(*timeout=20*)
This matches the shell prompt. This is little more than a short-cut to the expect() method. This returns True if the shell prompt was matched. This returns False if there was a timeout. Note that if you called login() with auto_prompt_reset set to False then you should have manually set the PROMPT attribute to a regex pattern for matching the prompt.

set_unique_prompt()
This sets the remote prompt to something more unique than # or $. This makes it easier for the prompt() method to match the shell prompt unambiguously. This method is called automatically by the login() method, but you may want to call it manually if you somehow reset the shell prompt. For example, if you 'su' to a different user then you will need to manually reset the prompt. This sends shell commands to the remote host to set the prompt, so this assumes the remote host is ready to receive commands.

Alternatively, you may use your own prompt pattern. Just set the PROMPT attribute to a regular expression that matches it. In this case you should call login() with auto_prompt_reset=False; then set the PROMPT attribute. After that the prompt() method will try to match your prompt pattern.

synch_original_prompt()
This attempts to find the prompt. Basically, press enter and record the response; press enter again and record the response; if the two responses are similar then assume we are at the original prompt.

report Module

Module used to parse the autotest job status file and generate a JSON file.

Optionally, we can also generate reports (HTML)

exception autotest.client.shared.report.**InvalidAutotestResultDirError**(*directory*)
Bases: exceptions.Exception

exception autotest.client.shared.report.**InvalidOutputDirError**(*directory*)
Bases: exceptions.Exception

class autotest.client.shared.report.**ReportLoggingConfig**(*use_console=True*)
Bases: *autotest.client.shared.logging_config.LoggingConfig*

Used with the sole purpose of providing convenient logging setup for this program.

configure_logging(*results_dir=None*, *verbose=False*)

class autotest.client.shared.report.**ReportOptionParser**
Bases: optparse.OptionParser

autotest.client.shared.report.**generate_html_report**(*results_dir*, *relative_links=True*)
Render a job report HTML.

All CSS and javascript are inlined, for more convenience.

Parameters **results_dir** – Path to the results directory.

autotest.client.shared.report.**generate_json_file**(*results_dir*, *relative_links=True*)
Generate a JSON file with autotest job summary on a given results directory

Parameters **results_dir** – Path to the results directory.

autotest.client.shared.report.**get_info_file**(*filename*)
Gets the contents of an autotest info file.

It also and highlights the file contents with possible problems.

Parameters **filename** – Info file path.

autotest.client.shared.report.**parse_results_dir**(*results_dir*, *relative_links=True*)
Parse a top level status file and produce a dictionary with job data.

Parameters **dirname** – Autotest results directory path

Returns Dictionary with job data.

autotest.client.shared.report.**write_html_report**(*results_dir*, *report_path=None*)
Write an HTML file at report_path, with job data summary.

If no report_path specified, generate one at results_dir/job_report.html.

Parameters

- **results_dir** – Directory with test results.
- **report_path** – Path to a report file (optional).

service Module

autotest.client.shared.service.**ServiceManager**(*run=<function run>*)
Detect which init program is being used, init or systemd and return a class has methods to start/stop services.

Get the system service manager service_manager = ServiceManager()

Stating service/unit "sshd" service_manager.start("sshd")

Getting a list of available units units = service_manager.list()

Disabling and stopping a list of services services_to_disable = ['ntpd', 'httpd'] for s in services_to_disable:

service_manager.disable(s) service_manager.stop(s)

Returns SysVInitServiceManager or SystemdServiceManager

Return type _GenericServiceManager

autotest.client.shared.service.**SpecificServiceManager**(*service_name*, *run=<function run>*)
Get the specific service manager for sshd sshd = SpecificServiceManager("sshd") sshd.start() sshd.stop() sshd.reload() sshd.restart() sshd.condrestart() sshd.status() sshd.enable() sshd.disable() sshd.is_enabled()

Parameters **service_name** (*str*) – systemd unit or init.d service to manager

Returns SpecificServiceManager that has start/stop methods

Return type _SpecificServiceManager

autotest.client.shared.service.**convert_systemd_target_to_runlevel**(*target*)
Convert systemd target to runlevel.

Parameters **target** (*str*) – systemd target

Returns sys_v runlevel

Return type str

Raises ValueError when systemd target is unknown

autotest.client.shared.service.**convert_sysv_runlevel**(*level*)
Convert runlevel to systemd target.

Parameters **level** (*str or int*) – sys_v runlevel

Returns systemd target

Return type str

Raises ValueError when runlevel is unknown

autotest.client.shared.service.**get_name_of_init**(*run=<function run>*)
Determine what executable is PID 1, aka init by checking /proc/1/exe This init detection will only run once and cache the return value.

Returns executable name for PID 1, aka init

Return type str

autotest.client.shared.service.**sys_v_init_command_generator**(*command*)
Generate lists of command arguments for sys_v style inits.

Parameters **command** (*str*) – start,stop,restart, etc.

Returns list of commands to pass to utils.run or similar function

Return type *list*

autotest.client.shared.service.**sys_v_init_result_parser**(*command*)
Parse results from sys_v style commands.

Parameters **command** (*str.*) – command.

Returns different from the command.

command is status: return true if service is running. command is is_enabled: return true if service is enalbled. command is list: return a dict from service name to status. command is others: return true if operate success.

autotest.client.shared.service.**systemd_command_generator**(*command*)
Generate list of command line argument strings for systemctl. One argument per string for compatibility Popen

WARNING: If systemctl detects that it is running on a tty it will use color, pipe to $PAGER, change column sizes and not truncate unit names. Use –no-pager to suppress pager output, or set PAGER=cat in the environment. You may need to take other steps to suppress color output. See https://bugzilla.redhat.com/show_bug.cgi?id=713567

Parameters **command** (*str*) – start,stop,restart, etc.

Returns list of command and arguments to pass to utils.run or similar functions

Return type *list*

autotest.client.shared.service.**systemd_result_parser**(*command*)
Parse results from systemd style commands.

Parameters **command** (*str.*) – command.

Returns different from the command.

command is status: return true if service is running. command is is_enabled: return true if service is enalbled. command is list: return a dict from service name to status. command is others: return true if operate success.

settings Module

A singleton class for accessing global config values.

provides access to global configuration file.

class autotest.client.shared.settings.**Settings**
Bases: object

check_stand_alone_client_run()

config = None

config_file = '/home/docs/checkouts/readthedocs.org/user_builds/autotest/checkouts/0.16.0/global_config.ini'

get_section_values(*sections*)
Return a config parser object containing a single section of the global configuration, that can be later written to a file object.

Parameters **section** – Tuple with sections we want to turn into a config parser object.

Returns ConfigParser() object containing all the contents of sections.

get_value(*section*, *key*, *type=<type 'str'>*, *default=<object object>*, *allow_blank=False*)

merge_configs(*shadow_config*)

override_value(*section*, *key*, *new_value*)
Override a value from the config file with a new value.

parse_config_file()

reset_values()
Reset all values to those found in the config files (undoes all overrides).

running_stand_alone_client = False

set_config_files(*config_file='/home/docs/checkouts/readthedocs.org/user_builds/autotest/checkouts/0.16.0/global_config*
shadow_file='/home/docs/checkouts/readthedocs.org/user_builds/autotest/checkouts/0.16.0/shadow_co

shadow_file = '/home/docs/checkouts/readthedocs.org/user_builds/autotest/checkouts/0.16.0/shadow_config.ini'

exception autotest.client.shared.settings.**SettingsError**
Bases: *autotest.client.shared.error.AutotestError*

exception autotest.client.shared.settings.**SettingsValueError**
Bases: *autotest.client.shared.settings.SettingsError*

software_manager Module

Software package management library.

This is an abstraction layer on top of the existing distributions high level package managers. It supports package operations useful for testing purposes, and multiple high level package managers (here called backends). If you want to make this lib to support your particular package manager/distro, please implement the given backend class.

author Higor Vieira Alves (halves@br.ibm.com)

author Lucas Meneghel Rodrigues (lmr@redhat.com)

author Ramon de Carvalho Valle (rcvalle@br.ibm.com)

copyright IBM 2008-2009

copyright Red Hat 2009-2010

class autotest.client.shared.software_manager.**AptBackend**
Bases: *autotest.client.shared.software_manager.DpkgBackend*

Implements the apt backend for software manager.

Set of operations for the apt package manager, commonly found on Debian and Debian based distributions, such as Ubuntu Linux.

add_repo(*repo*)
Add an apt repository.

Parameters **repo** – Repository string. Example: 'deb http://archive.ubuntu.com/ubuntu/ maverick universe'

install (*name*)
Installs package [name].

Parameters **name** – Package name.

provides (*path*)
Return a list of packages that provide [path].

Parameters **path** – File path.

remove (*name*)
Remove package [name].

Parameters **name** – Package name.

remove_repo (*repo*)
Remove an apt repository.

Parameters **repo** – Repository string. Example: 'deb http://archive.ubuntu.com/ubuntu/ maverick universe'

upgrade (*name=None*)
Upgrade all packages of the system with eventual new versions.

Optionally, upgrade individual packages.

Parameters **name** (*str*) – optional parameter wildcard spec to upgrade

class autotest.client.shared.software_manager.**BaseBackend**
Bases: object

This class implements all common methods among backends.

install_what_provides (*path*)
Installs package that provides [path].

Parameters **path** – Path to file.

class autotest.client.shared.software_manager.**DpkgBackend**
Bases: *autotest.client.shared.software_manager.BaseBackend*

This class implements operations executed with the dpkg package manager.

dpkg is a lower level package manager, used by higher level managers such as apt and aptitude.

INSTALLED_OUTPUT = 'install ok installed'

PACKAGE_TYPE = 'deb'

check_installed (*name*)

list_all ()
List all packages available in the system.

list_files (*package*)
List files installed by package [package].

Parameters **package** – Package name.

Returns List of paths installed by package.

class autotest.client.shared.software_manager.**RpmBackend**
Bases: *autotest.client.shared.software_manager.BaseBackend*

This class implements operations executed with the rpm package manager.

rpm is a lower level package manager, used by higher level managers such as yum and zypper.

PACKAGE_TYPE = 'rpm'

SOFTWARE_COMPONENT_QRY = 'rpm %{NAME} %{VERSION} %{RELEASE} %{SIGMD5} %{ARCH}'

check_installed(*name*, *version=None*, *arch=None*)
Check if package [name] is installed.

Parameters

- **name** – Package name.
- **version** – Package version.
- **arch** – Package architecture.

list_all(*software_components=True*)
List all installed packages.

Parameters **software_components** – log in a format suitable for the SoftwareComponent schema

list_files(*name*)
List files installed on the system by package [name].

Parameters **name** – Package name.

class autotest.client.shared.software_manager.**SoftwareManager**
Bases: object

Package management abstraction layer.

It supports a set of common package operations for testing purposes, and it uses the concept of a backend, a helper class that implements the set of operations of a given package management tool.

class autotest.client.shared.software_manager.**SoftwareManagerLoggingConfig**(*use_console=True*)
Bases: *autotest.client.shared.logging_config.LoggingConfig*

Used with the sole purpose of providing logging setup for this program.

configure_logging(*results_dir=None*, *verbose=False*)

class autotest.client.shared.software_manager.**SystemInspector**
Bases: object

System inspector class.

This may grow up to include more complete reports of operating system and machine properties.

get_package_management()
Determine the supported package management systems present on the system. If more than one package management system installed, try to find the best supported system.

class autotest.client.shared.software_manager.**YumBackend**
Bases: *autotest.client.shared.software_manager.RpmBackend*

Implements the yum backend for software manager.

Set of operations for the yum package manager, commonly found on Yellow Dog Linux and Red Hat based distributions, such as Fedora and Red Hat Enterprise Linux.

add_repo(*url*)
Adds package repository located on [url].

Parameters **url** – Universal Resource Locator of the repository.

install(*name*)
Installs package [name]. Handles local installs.

provides (*name*)
Returns a list of packages that provides a given capability.

Parameters **name** – Capability name (eg, 'foo').

remove (*name*)
Removes package [name].

Parameters **name** – Package name (eg. 'ipython').

remove_repo (*url*)
Removes package repository located on [url].

Parameters **url** – Universal Resource Locator of the repository.

upgrade (*name=None*)
Upgrade all available packages.

Optionally, upgrade individual packages.

Parameters **name** (*str*) – optional parameter wildcard spec to upgrade

class autotest.client.shared.software_manager.**ZypperBackend**
Bases: *autotest.client.shared.software_manager.RpmBackend*

Implements the zypper backend for software manager.

Set of operations for the zypper package manager, found on SUSE Linux.

add_repo (*url*)
Adds repository [url].

Parameters **url** – URL for the package repository.

install (*name*)
Installs package [name]. Handles local installs.

Parameters **name** – Package Name.

provides (*name*)
Searches for what provides a given file.

Parameters **name** – File path.

remove (*name*)
Removes package [name].

remove_repo (*url*)
Removes repository [url].

Parameters **url** – URL for the package repository.

upgrade (*name=None*)
Upgrades all packages of the system.

Optionally, upgrade individual packages.

Parameters **name** (*str*) – Optional parameter wildcard spec to upgrade

autotest.client.shared.software_manager.**install_distro_packages** (*distro_pkg_map*, *interactive=False*)
Installs packages for the currently running distribution

This utility function checks if the currently running distro is a key in the distro_pkg_map dictionary, and if there is a list of packages set as its value.

If these conditions match, the packages will be installed using the software manager interface, thus the native packaging system if the currenlty running distro.

Parameters **distro_pkg_map** (*dict*) – mapping of distro name, as returned by utils.get_os_vendor(), to a list of package names

Returns True if any packages were actually installed, False otherwise

ssh_key Module

autotest.client.shared.ssh_key.**get_public_key**()

Return a valid string ssh public key for the user executing autoserv or autotest. If there's no DSA or RSA public key, create a DSA keypair with ssh-keygen and return it.

Returns a ssh public key

Return type str

autotest.client.shared.ssh_key.**setup_ssh_key**(*hostname*, *user*, *password*, *port*)

Setup up remote login in another server by using public key

Parameters

- **hostname** (*str*) – the server to login
- **user** (*str*) – user to login
- **password** (*str*) – password
- **port** (*int*) – port number

syncdata Module

test Module

class autotest.client.shared.test.**Subtest**

Bases: object

Collect result of subtest of main test.

clean()

Check if cleanup is defined.

For makes test fatal add before implementation of test method decorator @subtest_nocleanup

decored()

failed = 0

classmethod get_full_text_result(*format_func=None*)

Returns string with text form of result

classmethod get_result()

Returns

Result of subtests. Format:

tuple(pass/fail,function_name,call_arguments)

classmethod get_text_result(*format_func=None*)

Returns string with text form of result

classmethod has_failed()

Returns If any of subtest not pass return True.

classmethod log_append(*msg*)
Add log_append to result output.

Parameters **msg** – Test of log_append

passed = 0

result = []

static result_to_string(*result*)
Format of result dict.

result = {

'result' : "PASS" / "FAIL", 'name' : class name, 'args' : test's args, 'kargs' : test's kargs, 'output' : return of test function,

}

Parameters **result** – Result of test.

static result_to_string_debug(*result*)

Parameters **result** – Result of test.

runsubtest(*url*, **args*, ***dargs*)
Execute another autotest test from inside the current test's scope.

Parameters

- **test** – Parent test.
- **url** – Url of new test.
- **tag** – Tag added to test name.
- **args** – Args for subtest.
- **dargs** – Dictionary with args for subtest.

@iterations: Number of subtest iterations. @profile_only: If true execute one profiled run.

test()
Check if test is defined.

For makes test fatal add before implementation of test method decorator @subtest_fatal

class autotest.client.shared.test.**base_test**(*job*, *bindir*, *outputdir*)
Bases: object

after_run_once()
Called after every run_once (including from a profiled run when it's called after stopping the profilers).

analyze_perf_constraints(*constraints*)

assert_(*expr*, *msg='Assertion failed.'*)

before_run_once()
Override in tests that need it, will be called before any run_once() call including the profiling run (when it's called before starting the profilers).

cleanup()

configure_crash_handler()

crash_handler_report()

drop_caches_between_iterations()

execute(*iterations=None*, *test_length=None*, *profile_only=None*, *_get_time=<built-in function time>*, *postprocess_profiled_run=None*, *constraints=()*, **args*, ***dargs*)

This is the basic execute method for the tests inherited from base_test. If you want to implement a benchmark test, it's better to implement the run_once function, to cope with the profiling infrastructure. For other tests, you can just override the default implementation.

Parameters

- **test_length** – The minimum test length in seconds. We'll run the run_once function for a number of times large enough to cover the minimum test length.
- **iterations** – A number of iterations that we'll run the run_once function. This parameter is incompatible with test_length and will be silently ignored if you specify both.
- **profile_only** – If true run X iterations with profilers enabled. If false run X iterations and one with profiling if profiles are enabled. If None, default to the value of job.default_profile_only.
- **_get_time** – [time.time] Used for unit test time injection.
- **postprocess_profiled_run** – Run the postprocessing for the profiled run.

initialize()

network_destabilizing = False

postprocess()

postprocess_iteration()

preserve_srcdir = False

process_failed_constraints()

register_after_iteration_hook(*iteration_hook*)

This is how we expect test writers to register an after_iteration_hook. This adds the method to the list of hooks which are executed after each iteration.

Parameters **iteration_hook** – Method to run after each iteration. A valid hook accepts a single argument which is the test object.

register_before_iteration_hook(*iteration_hook*)

This is how we expect test writers to register a before_iteration_hook. This adds the method to the list of hooks which are executed before each iteration.

Parameters **iteration_hook** – Method to run before each iteration. A valid hook accepts a single argument which is the test object.

run_once_profiling(*postprocess_profiled_run*, **args*, ***dargs*)

setup()

warmup(**args*, ***dargs*)

write_attr_keyval(*attr_dict*)

write_iteration_keyval(*attr_dict*, *perf_dict*, *tap_report=None*)

write_perf_keyval(*perf_dict*)

write_test_keyval(*attr_dict*)

autotest.client.shared.test.**runtest**(*job*, *url*, *tag*, *args*, *dargs*, *local_namespace={}*, *global_namespace={}*, *before_test_hook=None*, *after_test_hook=None*, *before_iteration_hook=None*, *after_iteration_hook=None*)

autotest.client.shared.test.**subtest_fatal**(*function*)
Decorator which mark test critical. If subtest fails the whole test ends.

autotest.client.shared.test.**subtest_nocleanup**(*function*)
Decorator used to disable cleanup function.

utils Module

Convenience functions for use by tests or whomever.

NOTE: this is a mixin library that pulls in functions from several places Note carefully what the precendece order is

There's no really good way to do this, as this isn't a class we can do inheritance with, just a collection of static methods.

class autotest.client.shared.utils.**AsyncJob**(*command*, *stdout_tee=None*, *stderr_tee=None*, *verbose=True*, *stdin=None*, *stderr_level=40*, *kill_func=None*)
Bases: *autotest.client.shared.utils.BgJob*

cleanup()

get_stderr()

get_stdout()

output_prepare(*stdout_file=None*, *stderr_file=None*)

process_output(*stdout=True*, *final_read=False*)

wait_for(*timeout=None*)

class autotest.client.shared.utils.**BgJob**(*command*, *stdout_tee=None*, *stderr_tee=None*, *verbose=True*, *stdin=None*, *stderr_level=40*)
Bases: object

cleanup()

output_prepare(*stdout_file=None*, *stderr_file=None*)

process_output(*stdout=True*, *final_read=False*)
output_prepare must be called prior to calling this

class autotest.client.shared.utils.**CmdResult**(*command=''*, *stdout=''*, *stderr=''*, *exit_status=None*, *duration=0*)
Bases: object

Command execution result.

command: String containing the command line itself exit_status: Integer exit code of the process stdout: String containing stdout of the process stderr: String containing stderr of the process duration: Elapsed wall clock time running the process

class autotest.client.shared.utils.**FileFieldMonitor**(*status_file*, *data_to_read*, *mode_diff*, *continuously=False*, *contlogging=False*, *separator=' +'*, *time_step=0.1*)
Bases: object

Monitors the information from the file and reports it's values.

It gather the information at start and stop of the measurement or continuously during the measurement.

class Monitor(*master*)
Bases: threading.Thread

Internal monitor class to ensure continuous monitor of monitored file.

run()
Start monitor in thread mode

FileFieldMonitor.**get_status**()

Returns Status of monitored process average value, time of test and array of monitored values and time step of continuous run.

FileFieldMonitor.**start**()
Start value monitor.

FileFieldMonitor.**stop**()
Stop value monitor.

class autotest.client.shared.utils.**ForAll**
Bases: list

class autotest.client.shared.utils.**ForAllP**
Bases: list

Parallel version of ForAll

class autotest.client.shared.utils.**ForAllPSE**
Bases: list

Parallel version of and suppress exception.

class autotest.client.shared.utils.**InterruptedThread**(*target*, *args=()*, *kwargs={}*)
Bases: threading.Thread

Run a function in a background thread.

join(*timeout=None*, *suppress_exception=False*)
Join the thread. If target raised an exception, re-raise it. Otherwise, return the value returned by target.

Parameters

- **timeout** – Timeout value to pass to threading.Thread.join().
- **suppress_exception** – If True, don't re-raise the exception.

run()
Run target (passed to the constructor). No point in calling this function directly. Call start() to make this function run in a new thread.

class autotest.client.shared.utils.**Statistic**
Bases: object

Class to display and collect average, max and min values of a given data set.

get_average()

get_max()

get_min()

record(*value*)
Record new value to statistic.

class autotest.client.shared.utils.**SystemLoad**(*pids*, *advanced=False*, *time_step=0.1*, *cpu_cont=False*, *use_log=False*)

Bases: object

Get system and/or process values and return average value of load.

dump(*pids=[]*)

Get the status of monitoring. :param pids: List of PIDs you intend to control. Use pids=[] to control

all defined PIDs.

return

tuple([cpu load], [memory load]):

([(PID1, (PID1_cpu_meas)), (PID2, (PID2_cpu_meas)), ...], [(PID1, (PID1_mem_meas)), (PID2, (PID2_mem_meas)), ...])

PID1_cpu_meas: average_values[], test_time, cont_meas_values[[]], time_step

PID1_mem_meas: average_values[], test_time, cont_meas_values[[]], time_step

where average_values[] are the measured values (mem_free,swap,...) which are described in SystemLoad.__init__()-FileFieldMonitor. cont_meas_values[[]] is a list of average_values in the sampling times.

get_cpu_status_string(*pids=[]*)

Convert status to string array. :param pids: List of PIDs you intend to control. Use pids=[] to control

all defined PIDs.

Returns String format to table.

get_mem_status_string(*pids=[]*)

Convert status to string array. :param pids: List of PIDs you intend to control. Use pids=[] to control

all defined PIDs.

Returns String format to table.

start(*pids=[]*)

Start monitoring of the process system usage. :param pids: List of PIDs you intend to control. Use pids=[] to control

all defined PIDs.

stop(*pids=[]*)

Stop monitoring of the process system usage. :param pids: List of PIDs you intend to control. Use pids=[] to control

all defined PIDs.

class autotest.client.shared.utils.**VersionableClass**

Bases: object

VersionableClass provides class hierarchy which automatically select right version of class. Class manipulation is used for this reason. By this reason is: Advantage) Only one version is working in one process. Class is changed in

whole process.

Disadvantage) Only one version is working in one process.

Example of usage (in utils_unittest):

class FooC(object): pass

#Not implemented get_version -> not used for versioning. class VCP(FooC, VersionableClass):

def __new__(cls, *args, **kargs): VCP.master_class = VCP return super(VCP, cls).__new__(cls, *args, **kargs)

def foo(self): pass

class VC2(VCP, VersionableClass): @staticmethod def get_version():

return "get_version_from_system"

@classmethod def is_right_version(cls, version):

if version is not None:

if "version is satisfied": return True

return False

def func1(self): print "func1"

def func2(self): print "func2"

get_version could be inherited. class VC3(VC2, VersionableClass):

@classmethod def is_right_version(cls, version):

if version is not None:

if "version+1 is satisfied": return True

return False

def func2(self): print "func2_2"

class M(VCP): pass

m = M() # <- When class is constructed the right version is # automatically selected. In this case VC3 is selected.

m.func2() # call VC3.func2(m) m.func1() # call VC2.func1(m) m.foo() # call VC1.foo(m)

 # When controlled "program" version is changed then is necessary call check_repair_versions or recreate object.

m.check_repair_versions()

priority of class. (change place where is method searched first in group # of verisonable class.)

class PP(VersionableClass):

def __new__(cls, *args, **kargs): PP.master_class = PP return super(PP, cls).__new__(cls, *args, **kargs)

class PP2(PP, VersionableClass): @staticmethod def get_version():

return "get_version_from_system"

@classmethod def is_right_version(cls, version):

if version is not None:

if "version is satisfied": return True

return False

def func1(self): print "PP func1"

class N(VCP, PP): pass

n = N()

n.func1() # -> "func2"

n.set_priority_class(PP, [VCP, PP])

n.func1() # -> "PP func1"

Necessary for using: 1) Subclass of versionable class must have implemented class methods

get_version and is_right_version. These two methods are necessary for correct version section. Class without this method will be never chosen like suitable class.

2.Every class derived from master_class have to add to class definition

inheritance from VersionableClass. Direct inheritance from Versionable Class is use like a mark for manipulation with VersionableClass.

3.Master of VersionableClass have to defined class variable

cls.master_class.

classmethod check_repair_versions(*master_classes=None*)

Check version of versionable class and if version not match repair version to correct version.

Parameters **master_classes** (*list.*) – Check and repair only master_class.

classmethod get_version()

Get version of installed OpenVSwtich. Must be re-implemented for in child class.

Returns Version or None when get_version is unsuccessful.

classmethod is_right_version(*version*)

Check condition for select control class. Function must be re-implemented in new OpenVSwitchControl class. Must be re-implemented for in child class.

Parameters **version** – version of OpenVSwtich

classmethod set_priority_class(*prioritized_class*, *group_classes*)

Set class priority. Limited only for change bases class priority inside one subclass.__bases__ after that continue to another class.

autotest.client.shared.utils.**archive_as_tarball**(*source_dir*, *dest_dir*, *tarball_name=None*, *compression='bz2'*, *verbose=True*)

Saves the given source directory to the given destination as a tarball

If the name of the archive is omitted, it will be taken from the source_dir. If it is an absolute path, dest_dir will be ignored. But, if both the destination directory and tarball anem is given, and the latter is not an absolute path, they will be combined.

For archiving directory '/tmp' in '/net/server/backup' as file 'tmp.tar.bz2', simply use:

```
>>> utils_misc.archive_as_tarball('/tmp', '/net/server/backup')
```

To save the file it with a different name, say 'host1-tmp.tar.bz2' and save it under '/net/server/backup', use:

```
>>> utils_misc.archive_as_tarball('/tmp', '/net/server/backup',
                                  'host1-tmp')
```

To save with gzip compression instead (resulting in the file '/net/server/backup/host1-tmp.tar.gz'), use:

```
>>> utils_misc.archive_as_tarball('/tmp', '/net/server/backup',
                                  'host1-tmp', 'gz')
```

autotest.client.shared.utils.**args_to_dict**(*args*)

Convert autoserv extra arguments in the form of key=val or key:val to a dictionary. Each argument key is converted to lowercase dictionary key.

Args: args - list of autoserv extra arguments.

Returns: dictionary

autotest.client.shared.utils.**ask**(*question*, *auto=False*)

Raw input with a prompt that emulates logging.

Parameters

- **question** – Question to be asked
- **auto** – Whether to return "y" instead of asking the question

autotest.client.shared.utils.**compare_versions**(*ver1*, *ver2*)

Version number comparison between ver1 and ver2 strings.

```
>>> compare_tuple("1", "2")
-1
>>> compare_tuple("foo-1.1", "foo-1.2")
-1
>>> compare_tuple("1.2", "1.2a")
-1
>>> compare_tuple("1.2b", "1.2a")
1
>>> compare_tuple("1.3.5.3a", "1.3.5.3b")
-1
```

Args: ver1: version string ver2: version string

Returns:

int: 1 if ver1 > ver2

0 if ver1 == ver2

-1 if ver1 < ver2

autotest.client.shared.utils.**configure**(*extra=None*, *configure='./configure'*)

Run configure passing in the correct host, build, and target options.

Parameters

- **extra** – extra command line arguments to pass to configure
- **configure** – which configure script to use

autotest.client.shared.utils.**convert_data_size**(*size*, *default_sufix='B'*)

Convert data size from human readable units to an int of arbitrary size.

Parameters

- **size** – Human readable data size representation (string).
- **default_sufix** – Default sufix used to represent data.

Returns Int with data size in the appropriate order of magnitude.

autotest.client.shared.utils.**cpu_affinity_by_task**(*pid*, *vcpu_pid*)
This function returns the allowed cpus from the proc entry for each vcpu's through its task id for a pid(of a VM)

autotest.client.shared.utils.**create_subnet_mask**(*bits*)

autotest.client.shared.utils.**delete_pid_file_if_exists**(*program_name*, *pid_files_dir=None*)
Tries to remove <program_name>.pid from the main autotest directory.

autotest.client.shared.utils.**deprecated**(*func*)
This is a decorator which can be used to mark functions as deprecated. It will result in a warning being emitted when the function is used.

autotest.client.shared.utils.**display_data_size**(*size*)
Display data size in human readable units.

Parameters **size** (*int*) – Data size, in Bytes.

Returns Human readable string with data size.

autotest.client.shared.utils.**etraceback**(*prep*, *exc_info*)

Enhanced Traceback formats traceback into lines "prep: line

name: line"

param prep desired line preposition

param exc_info sys.exc_info of the exception

return string which contains beautifully formatted exception

autotest.client.shared.utils.**format_ip_with_mask**(*ip*, *mask_bits*)

autotest.client.shared.utils.**generate_random_string**(*length*, *ignore_str='!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'*, *convert_str=''*)
Return a random string using alphanumeric characters.

Parameters

- **length** – Length of the string that will be generated.
- **ignore_str** – Characters that will not include in generated string.
- **convert_str** – Characters that need to be escaped (prepend "").

Returns The generated random string.

autotest.client.shared.utils.**get_arch**(*run_function=<function run>*)
Get the hardware architecture of the machine. run_function is used to execute the commands. It defaults to utils.run() but a custom method (if provided) should be of the same schema as utils.run. It should return a CmdResult object and throw a CmdError exception.

autotest.client.shared.utils.**get_archive_tarball_name**(*source_dir*, *tarball_name*, *compression*)
Get the name for a tarball file, based on source, name and compression

autotest.client.shared.utils.**get_children_pids**(*ppid*)
Get all PIDs of children/threads of parent ppid param ppid: parent PID return: list of PIDs of all children/threads of ppid

autotest.client.shared.utils.**get_cpu_percentage**(*function*, **args*, ***dargs*)
Returns a tuple containing the CPU% and return value from function call.

This function calculates the usage time by taking the difference of the user and system times both before and after the function call.

autotest.client.shared.utils.**get_field**(*data*, *param*, *linestart=''*, *sep=' '*)
Parse data from string. :param data: Data to parse.

example:

data: cpu 324 345 34 5 345 cpu0 34 11 34 34 33 ^^^^ start of line params 0 1 2 3 4

Parameters

- **param** – Position of parameter after linestart marker.
- **linestart** – String to which start line with parameters.
- **sep** – Separator between parameters regular expression.

autotest.client.shared.utils.**get_file**(*src*, *dest*, *permissions=None*)
Get a file from src, which can be local or a remote URL

autotest.client.shared.utils.**get_ip_local_port_range**()

autotest.client.shared.utils.**get_num_logical_cpus_per_socket**(*run_function=<function run>*)
Get the number of cores (including hyperthreading) per cpu. run_function is used to execute the commands. It defaults to utils.run() but a custom method (if provided) should be of the same schema as utils.run. It should return a CmdResult object and throw a CmdError exception.

autotest.client.shared.utils.**get_pid_from_file**(*program_name*, *pid_files_dir=None*)
Reads the pid from <program_name>.pid in the autotest directory.

:param program_name the name of the program :return: the pid if the file exists, None otherwise.

autotest.client.shared.utils.**get_pid_path**(*program_name*, *pid_files_dir=None*)

autotest.client.shared.utils.**get_process_name**(*pid*)
Get process name from PID. :param pid: PID of process.

autotest.client.shared.utils.**get_relative_path**(*path*, *reference*)
Given 2 absolute paths "path" and "reference", compute the path of "path" as relative to the directory "reference".

:param path the absolute path to convert to a relative path :param reference an absolute directory path to which the relative

path will be computed

autotest.client.shared.utils.**get_stderr_level**(*stderr_is_expected*)

autotest.client.shared.utils.**get_stream_tee_file**(*stream*, *level*, *prefix=''*)

autotest.client.shared.utils.**get_unused_port**()
Finds a semi-random available port. A race condition is still possible after the port number is returned, if another process happens to bind it.

Returns: A port number that is unused on both TCP and UDP.

autotest.client.shared.utils.**hash**(*type*, *input=None*)

Returns an hash object of type md5 or sha1. This function is implemented in order to encapsulate hash objects in a way that is compatible with python 2.4 and python 2.6 without warnings.

Note that even though python 2.6 hashlib supports hash types other than md5 and sha1, we are artificially limiting the input values in order to make the function to behave exactly the same among both python implementations.

Parameters **input** – Optional input string that will be used to update the hash.

autotest.client.shared.utils.**import_site_class**(*path*, *module*, *classname*, *baseclass*, *modulefile=None*)

Try to import site specific class from site specific file if it exists

Args: path: full filename of the source file calling this (ie __file__) module: full module name classname: class name to be loaded from site file baseclass: base class object to return when no site file present or

to mixin when site class exists but is not inherited from baseclass

modulefile: module filename

Returns: baseclass if site specific class does not exist, the site specific class if it exists and is inherited from baseclass or a mixin of the site specific class and baseclass when the site specific class exists and is not inherited from baseclass

Raises: ImportError if the site file exists but imports fails

autotest.client.shared.utils.**import_site_function**(*path*, *module*, *funcname*, *dummy*, *modulefile=None*)

Try to import site specific function from site specific file if it exists

Args: path: full filename of the source file calling this (ie __file__) module: full module name funcname: function name to be imported from site file dummy: dummy function to return in case there is no function to import modulefile: module filename

Returns: site specific function object or dummy

Raises: ImportError if the site file exists but imports fails

autotest.client.shared.utils.**import_site_module**(*path*, *module*, *dummy=None*, *modulefile=None*)

Try to import the site specific module if it exists.

:param path full filename of the source file calling this (ie __file__) :param module full module name :param dummy dummy value to return in case there is no symbol to import :param modulefile module filename

Returns site specific module or dummy

:raise ImportError if the site file exists but imports fails

autotest.client.shared.utils.**import_site_symbol**(*path*, *module*, *name*, *dummy=None*, *modulefile=None*)

Try to import site specific symbol from site specific file if it exists

:param path full filename of the source file calling this (ie __file__) :param module full module name :param name symbol name to be imported from the site file :param dummy dummy value to return in case there is no symbol to import :param modulefile module filename

Returns site specific symbol or dummy

:raise ImportError if the site file exists but imports fails

autotest.client.shared.utils.**interactive_download**(*url*, *output_file*, *title=''*, *chunk_size=102400*)

Interactively downloads a given file url to a given output file

Parameters

- **url** (*string*) – URL for the file to be download
- **output_file** (*string*) – file name or absolute path on which to save the file to
- **title** (*string*) – optional title to go along the progress bar
- **chunk_size** (*integer*) – amount of data to read at a time

autotest.client.shared.utils.**ip_to_long**(*ip*)

autotest.client.shared.utils.**is_url**(*path*)
Return true if path looks like a URL

autotest.client.shared.utils.**join_bg_jobs**(*bg_jobs*, *timeout=None*)
Joins the bg_jobs with the current thread.

Returns the same list of bg_jobs objects that was passed in.

autotest.client.shared.utils.**log_last_traceback**(*msg=None*, *log=<function error>*)
Writes last traceback into specified log. :param msg: Override the default message. ["Original traceback"] :param log: Where to log the traceback [logging.error]

autotest.client.shared.utils.**long_to_ip**(*number*)

autotest.client.shared.utils.**make**(*extra=''*, *make='make'*, *timeout=None*, *ignore_status=False*)
Run make, adding MAKEOPTS to the list of options.

Parameters **extra** – extra command line arguments to pass to make.

autotest.client.shared.utils.**matrix_to_string**(*matrix*, *header=None*)
Return a pretty, aligned string representation of a nxm matrix.

This representation can be used to print any tabular data, such as database results. It works by scanning the lengths of each element in each column, and determining the format string dynamically.

Parameters

- **matrix** – Matrix representation (list with n rows of m elements).
- **header** – Optional tuple or list with header elements to be displayed.

autotest.client.shared.utils.**merge_trees**(*src*, *dest*)
Merges a source directory tree at 'src' into a destination tree at 'dest'. If a path is a file in both trees than the file in the source tree is APPENDED to the one in the destination tree. If a path is a directory in both trees then the directories are recursively merged with this function. In any other case, the function will skip the paths that cannot be merged (instead of failing).

autotest.client.shared.utils.**normalize_hostname**(*alias*)

autotest.client.shared.utils.**nuke_pid**(*pid*, *signal_queue=(15, 9)*)

autotest.client.shared.utils.**nuke_subprocess**(*subproc*)

autotest.client.shared.utils.**open_write_close**(*filename*, *data*)

autotest.client.shared.utils.**pid_is_alive**(*pid*)
True if process pid exists and is not yet stuck in Zombie state. Zombies are impossible to move between cgroups, etc. pid can be integer, or text of integer.

autotest.client.shared.utils.**program_is_alive**(*program_name*, *pid_files_dir=None*)
Checks if the process is alive and not in Zombie state.

:param program_name the name of the program :return: True if still alive, False otherwise

autotest.client.shared.utils.**read_file**(*filename*)

autotest.client.shared.utils.**read_keyval**(*path*)

Read a key-value pair format file into a dictionary, and return it. Takes either a filename or directory name as input. If it's a directory name, we assume you want the file to be called keyval.

autotest.client.shared.utils.**read_one_line**(*filename*)

autotest.client.shared.utils.**run**(*command*, *timeout=None*, *ignore_status=False*, *stdout_tee=None*, *stderr_tee=None*, *verbose=True*, *stdin=None*, *stderr_is_expected=None*, *args=()*)

Run a command on the host.

Parameters

- **command** – the command line string.
- **timeout** – time limit in seconds before attempting to kill the running process. The run() function will take a few seconds longer than 'timeout' to complete if it has to kill the process.
- **ignore_status** – do not raise an exception, no matter what the exit code of the command is.
- **stdout_tee** – optional file-like object to which stdout data will be written as it is generated (data will still be stored in result.stdout).
- **stderr_tee** – likewise for stderr.
- **verbose** – if True, log the command being run.
- **stdin** – stdin to pass to the executed process (can be a file descriptor, a file object of a real file or a string).
- **args** – sequence of strings of arguments to be given to the command inside " quotes after they have been escaped for that; each element in the sequence will be given as a separate command argument

Returns a CmdResult object

Raises CmdError the exit code of the command execution was not 0

autotest.client.shared.utils.**run_bg**(**args*, ***dargs*)

Function deprecated. Please use BgJob class instead.

autotest.client.shared.utils.**run_parallel**(*commands*, *timeout=None*, *ignore_status=False*, *stdout_tee=None*, *stderr_tee=None*)

Behaves the same as run() with the following exceptions:

•commands is a list of commands to run in parallel.

•ignore_status toggles whether or not an exception should be raised on any error.

Returns a list of CmdResult objects

class autotest.client.shared.utils.**run_randomly**(*run_sequentially=False*)

add(**args*, ***dargs*)

run(*fn*)

autotest.client.shared.utils.**safe_rmdir**(*path*, *timeout=10*)
Try to remove a directory safely, even on NFS filesystems.

Sometimes, when running an autotest client test on an NFS filesystem, when not all filedescriptors are closed, NFS will create some temporary files, that will make shutil.rmtree to fail with error 39 (directory not empty). So let's keep trying for a reasonable amount of time before giving up.

Parameters

- **path** (*string*) – Path to a directory to be removed.
- **timeout** (*int*) – Time that the function will try to remove the dir before giving up (seconds)

Raises OSError, with errno 39 in case after the timeout shutil.rmtree could not successfuly complete. If any attempt to rmtree fails with errno different than 39, that exception will be just raised.

autotest.client.shared.utils.**set_ip_local_port_range**(*lower*, *upper*)

autotest.client.shared.utils.**sh_escape**(*command*)
Escape special characters from a command so that it can be passed as a double quoted (" ") string in a (ba)sh command.

Args: command: the command string to escape.

Returns: The escaped command string. The required englobing double quotes are NOT added and so should be added at some point by the caller.

See also: http://www.tldp.org/LDP/abs/html/escapingsection.html

autotest.client.shared.utils.**signal_pid**(*pid*, *sig*)
Sends a signal to a process id. Returns True if the process terminated successfully, False otherwise.

autotest.client.shared.utils.**signal_program**(*program_name*, *sig=15*, *pid_files_dir=None*)
Sends a signal to the process listed in <program_name>.pid

:param program_name the name of the program :param sig signal to send

autotest.client.shared.utils.**strip_unicode**(*input*)

autotest.client.shared.utils.**system**(*command*, *timeout=None*, *ignore_status=False*, *verbose=True*)
Run a command

Parameters

- **timeout** – timeout in seconds
- **ignore_status** – if ignore_status=False, throw an exception if the command's exit code is non-zero if ignore_status=True, return the exit code.
- **verbose** – if True, log the command being run.

Returns exit status of command (note, this will always be zero unless ignore_status=True)

autotest.client.shared.utils.**system_output**(*command*, *timeout=None*, *ignore_status=False*, *retain_output=False*, *args=()*, *verbose=True*)
Run a command and return the stdout output.

Parameters

- **command** – command string to execute.

- **timeout** – time limit in seconds before attempting to kill the running process. The function will take a few seconds longer than 'timeout' to complete if it has to kill the process.
- **ignore_status** – do not raise an exception, no matter what the exit code of the command is.
- **retain_output** – set to True to make stdout/stderr of the command output to be also sent to the logging system
- **args** – sequence of strings of arguments to be given to the command inside " quotes after they have been escaped for that; each element in the sequence will be given as a separate command argument
- **verbose** – if True, log the command being run.

Returns a string with the stdout output of the command.

autotest.client.shared.utils.**system_output_parallel**(*commands*, *timeout=None*, *ignore_status=False*, *retain_output=False*)

autotest.client.shared.utils.**system_parallel**(*commands*, *timeout=None*, *ignore_status=False*)

This function returns a list of exit statuses for the respective list of commands.

autotest.client.shared.utils.**unmap_url**(*srcdir*, *src*, *destdir='.'*)

Receives either a path to a local file or a URL. returns either the path to the local file, or the fetched URL

unmap_url('/usr/src', 'foo.tar', '/tmp') = '/usr/src/foo.tar'

unmap_url('/usr/src', 'http://site/file', '/tmp') = '/tmp/file' (after retrieving it)

autotest.client.shared.utils.**update_version**(*srcdir*, *preserve_srcdir*, *new_version*, *install*, **args*, ***dargs*)

Make sure srcdir is version new_version

If not, delete it and install() the new version.

In the preserve_srcdir case, we just check it's up to date, and if not, we rerun install, without removing srcdir

autotest.client.shared.utils.**urlopen**(*url*, *data=None*, *timeout=5*)

Wrapper to urllib2.urlopen with timeout addition.

autotest.client.shared.utils.**urlretrieve**(*url*, *filename*, *data=None*, *timeout=300*)

Retrieve a file from given url.

autotest.client.shared.utils.**write_keyval**(*path*, *dictionary*, *type_tag=None*, *tap_report=None*)

Write a key-value pair format file out to a file. This uses append mode to open the file, so existing text will not be overwritten or reparsed.

If type_tag is None, then the key must be composed of alphanumeric characters (or dashes+underscores). However, if type-tag is not null then the keys must also have "{type_tag}" as a suffix. At the moment the only valid values of type_tag are "attr" and "perf".

Parameters

- **path** – full path of the file to be written
- **dictionary** – the items to write
- **type_tag** – see text above

autotest.client.shared.utils.**write_one_line**(*filename*, *line*)

autotest.client.shared.utils.**write_pid**(*program_name*, *pid_files_dir=None*)
Try to drop <program_name>.pid in the main autotest directory.

Args: program_name: prefix for file name

utils_cgroup Module

Helpers for cgroup testing.

copyright 2011 Red Hat Inc.

author Lukas Doktor <ldoktor@redhat.com>

class autotest.client.shared.utils_cgroup.**Cgroup**(*module*, *_client*)
Bases: object

Cgroup handling class.

cgclassify_cgroup(*pid*, *cgroup*)
Classify pid into cgroup

Parameters

- **pid** – pid of the process
- **cgroup** – cgroup name

cgdelete_all_cgroups()
Delete all cgroups in the module

cgdelete_cgroup(*cgroup*, *recursive=False*)
Delete desired cgroup.

Params cgroup desired cgroup

:params force:If true, sub cgroup can be deleted with parent cgroup

cgexec(*cgroup*, *cmd*, *args=''*)
Execute command in desired cgroup

Param cgroup: Desired cgroup

Param cmd: Executed command

Param args: Executed command's parameters

cgset_property(*prop*, *value*, *pwd=None*, *check=True*, *checkprop=None*)
Sets the property value by cgset command

Param prop: property name (file)

Param value: desired value

Parameters

- **pwd** – cgroup directory
- **check** – check the value after setup / override checking value
- **checkprop** – override prop when checking the value

get_cgroup_index(*cgroup*)
Get cgroup's index in cgroups

Param cgroup: cgroup name

Returns index of cgroup

get_cgroup_name (*pwd=None*)
Get cgroup's name

Param pwd: cgroup name

Returns cgroup's name

get_pids (*pwd=None*)
Get all pids in cgroup

Params pwd: cgroup directory

Returns all pids(list)

get_property (*prop*, *pwd=None*)
Gets the property value :param prop: property name (file) :param pwd: cgroup directory :return: [] values or None when FAILED

initialize (*modules*)
Initializes object for use.

Parameters **modules** – Array of all available cgroup modules.

is_cgroup (*pid*, *pwd*)
Checks if the 'pid' process is in 'pwd' cgroup :param pid: pid of the process :param pwd: cgroup directory :return: 0 when is 'pwd' member

is_root_cgroup (*pid*)
Checks if the 'pid' process is in root cgroup (WO cgroup) :param pid: pid of the process :return: 0 when is 'root' member

mk_cgroup (*pwd=None*, *cgroup=None*)
Creates new temporary cgroup :param pwd: where to create this cgroup (default: self.root) :param cgroup: desired cgroup name :return: last cgroup index

mk_cgroup_cgcreate (*pwd=None*, *cgroup=None*)
Make a cgroup by cgcreate command

Params cgroup: Maked cgroup name

Returns last cgroup index

refresh_cgroups ()
Refresh all cgroups path.

rm_cgroup (*pwd*)
Removes cgroup.

Parameters **pwd** – cgroup directory.

set_cgroup (*pid*, *pwd=None*)
Sets cgroup membership :param pid: pid of the process :param pwd: cgroup directory

set_property (*prop*, *value*, *pwd=None*, *check=True*, *checkprop=None*)
Sets the property value :param prop: property name (file) :param value: desired value :param pwd: cgroup directory :param check: check the value after setup / override checking value :param checkprop: override prop when checking the value

set_property_h (*prop*, *value*, *pwd=None*, *check=True*, *checkprop=None*)
Sets the one-line property value concerning the K,M,G postfix :param prop: property name (file) :param value: desired value :param pwd: cgroup directory :param check: check the value after setup / override checking value :param checkprop: override prop when checking the value

set_root_cgroup(*pid*)
Resets the cgroup membership (sets to root) :param pid: pid of the process :return: 0 when PASSED

smoke_test()
Smoke test Module independent basic tests

test(*cmd*)
Executes cgroup_client.py with cmd parameter.

Parameters **cmd** – command to be executed

Returns subprocess.Popen() process

class autotest.client.shared.utils_cgroup.**CgroupModules**(*mountdir=None*)
Bases: object

Handles the list of different cgroup filesystems.

get_pwd(*module*)
Returns the mount directory of 'module' :param module: desired module (memory, ...) :return: mount directory of 'module' or None

init(*_modules*)

Checks the mounted modules and if necessary mounts them into tmp mountdir.

Parameters **_modules** – Desired modules.'memory','cpu,cpuset'...

Returns Number of initialized modules.

autotest.client.shared.utils_cgroup.**all_cgroup_delete**()
Clear all cgroups in system

autotest.client.shared.utils_cgroup.**cgconfig_condrestart**()
Condrestart cgconfig service

autotest.client.shared.utils_cgroup.**cgconfig_exists**()
Check if cgconfig is available on the host or perhaps systemd is used

autotest.client.shared.utils_cgroup.**cgconfig_is_running**()
Check cgconfig service status

autotest.client.shared.utils_cgroup.**cgconfig_restart**()
Restart cgconfig service

autotest.client.shared.utils_cgroup.**cgconfig_start**()
Stop cgconfig service

autotest.client.shared.utils_cgroup.**cgconfig_stop**()
Start cgconfig service

autotest.client.shared.utils_cgroup.**get_all_controllers**()
Get all controllers used in system

Returns all used controllers(controller_list)

autotest.client.shared.utils_cgroup.**get_cgroup_mountpoint**(*controller*)
Get desired controller's mountpoint

@controller: Desired controller :return: controller's mountpoint

autotest.client.shared.utils_cgroup.**get_load_per_cpu**(*_stats=None*)
Gather load per cpu from /proc/stat :param _stats: previous values :return: list of diff/absolute values of CPU times [SUM, CPU1, CPU2, ...]

autotest.client.shared.utils_cgroup.**resolve_task_cgroup_path**(*pid*, *controller*)
Resolving cgroup mount path of a particular task

Params pid : process id of a task for which the cgroup path required

Params controller: takes one of the controller names in controller list

Returns resolved path for cgroup controllers of a given pid

autotest.client.shared.utils_cgroup.**service_cgconfig_control**(*action*)
Cgconfig control by action.

If cmd executes successfully, return True, otherwise return False. If the action is status, return True when it's running, otherwise return False. To check if the cgconfig stuff is available, use action "exists".

@ param action: start|stop|status|restart|condrestart

utils_koji Module

class autotest.client.shared.utils_koji.**KojiClient**(*cmd=None*)
Bases: object

Stablishes a connection with the build system, either koji or brew.

This class provides convenience methods to retrieve information on packages and the packages themselves hosted on the build system. Packages should be specified in the KojiPgkSpec syntax.

CMD_LOOKUP_ORDER = ['/usr/bin/brew', '/usr/bin/koji']

CONFIG_MAP = {'/usr/bin/brew': '/etc/brewkoji.conf', '/usr/bin/koji': '/etc/koji.conf'}

get_default_command()
Looks up for koji or brew "binaries" on the system

Systems with plain koji usually don't have a brew cmd, while systems with koji, have *both* koji and brew utilities. So we look for brew first, and if found, we consider that the system is configured for brew. If not, we consider this is a system with plain koji.

Returns either koji or brew command line executable path, or None

get_pkg_base_url()
Gets the base url for packages in Koji

get_pkg_info(*pkg*)
Returns information from Koji on the package

Parameters **pkg** (KojiPkgSpec) – information about the package, as a KojiPkgSpec instance

Returns information from Koji about the specified package

get_pkg_rpm_file_names(*pkg*, *arch=None*)
Gets the file names for the RPM packages specified in pkg

Parameters

- **pkg** (KojiPkgSpec) – a package specification
- **arch** (*string*) – packages built for this architecture, but also including architecture independent (noarch) packages

get_pkg_rpm_info(*pkg*, *arch=None*)
Returns a list of information on the RPM packages found on koji

Parameters

- **pkg** (KojiPkgSpec) – a package specification
- **arch** (*string*) – packages built for this architecture, but also including architecture independent (noarch) packages

get_pkg_rpm_names(*pkg*, *arch=None*)
Gets the names for the RPM packages specified in pkg

Parameters

- **pkg** (KojiPkgSpec) – a package specification
- **arch** (*string*) – packages built for this architecture, but also including architecture independent (noarch) packages

get_pkg_urls(*pkg*, *arch=None*)
Gets the urls for the packages specified in pkg

Parameters

- **pkg** (KojiPkgSpec) – a package specification
- **arch** (*string*) – packages built for this architecture, but also including architecture independent (noarch) packages

get_pkgs(*pkg*, *dst_dir*, *arch=None*)
Download the packages

Parameters

- **pkg** (KojiPkgSpec) – a package specification
- **dst_dir** (*string*) – the destination directory, where the downloaded packages will be saved on
- **arch** (*string*) – packages built for this architecture, but also including architecture independent (noarch) packages

get_scratch_base_url()
Gets the base url for scratch builds in Koji

get_scratch_pkg_urls(*pkg*, *arch=None*)
Gets the urls for the scratch packages specified in pkg

Parameters

- **pkg** (KojiScratchPkgSpec) – a scratch package specification
- **arch** (*string*) – packages built for this architecture, but also including architecture independent (noarch) packages

get_scratch_pkgs(*pkg*, *dst_dir*, *arch=None*)
Download the packages from a scratch build

Parameters

- **pkg** (KojiScratchPkgSpec) – a scratch package specification
- **dst_dir** (*string*) – the destination directory, where the downloaded packages will be saved on
- **arch** (*string*) – packages built for this architecture, but also including architecture independent (noarch) packages

get_session_options()
Filter only options necessary for setting up a cobbler client session

Returns only the options used for session setup

is_command_valid()
Checks if the currently set koji command is valid

Returns True or False

is_config_valid()
Checks if the currently set koji configuration is valid

Returns True or False

is_pkg_spec_build_valid(*pkg*)
Checks if build is valid on Koji

Parameters **pkg** – a Pkg instance

is_pkg_spec_tag_valid(*pkg*)
Checks if tag is valid on Koji

Parameters **pkg** (KojiPkgSpec) – a package specification

is_pkg_valid(*pkg*)
Checks if this package is altogether valid on Koji

This verifies if the build or tag specified in the package specification actually exist on the Koji server

Returns True or False

read_config(*check_is_valid=True*)
Reads options from the Koji configuration file

By default it checks if the koji configuration is valid

Parameters **check_valid** (*boolean*) – whether to include a check on the configuration

Raise ValueError

Returns None

class autotest.client.shared.utils_koji.**KojiDirIndexParser**
Bases: HTMLParser.HTMLParser

Parser for HTML directory index pages, specialized to look for RPM links

handle_starttag(*tag*, *attrs*)
Handle tags during the parsing

This just looks for links ('a' tags) for files ending in .rpm

class autotest.client.shared.utils_koji.**KojiPkgSpec**(*text=''*, *tag=None*, *build=None*, *package=None*, *subpackages=[]*)
Bases: object

A package specification syntax parser for Koji

This holds information on either tag or build, and packages to be fetched from koji and possibly installed (features external do this class).

New objects can be created either by providing information in the textual format or by using the actual parameters for tag, build, package and sub- packages. The textual format is useful for command line interfaces and configuration files, while using parameters is better for using this in a programatic fashion.

The following sets of examples are interchangeable. Specifying all packages part of build number 1000:

```
>>> from kvm_utils import KojiPkgSpec
>>> pkg = KojiPkgSpec('1000')
```

```
>>> pkg = KojiPkgSpec(build=1000)
```

Specifying only a subset of packages of build number 1000:

```
>>> pkg = KojiPkgSpec('1000:kernel,kernel-devel')
```

```
>>> pkg = KojiPkgSpec(build=1000,
                      subpackages=['kernel', 'kernel-devel'])
```

Specifying the latest build for the 'kernel' package tagged with 'dist-f14':

```
>>> pkg = KojiPkgSpec('dist-f14:kernel')
```

```
>>> pkg = KojiPkgSpec(tag='dist-f14', package='kernel')
```

Specifying the 'kernel' package using the default tag:

```
>>> kvm_utils.set_default_koji_tag('dist-f14')
>>> pkg = KojiPkgSpec('kernel')
```

```
>>> pkg = KojiPkgSpec(package='kernel')
```

Specifying the 'kernel' package using the default tag:

```
>>> kvm_utils.set_default_koji_tag('dist-f14')
>>> pkg = KojiPkgSpec('kernel')
```

```
>>> pkg = KojiPkgSpec(package='kernel')
```

If you do not specify a default tag, and give a package name without an explicit tag, your package specification is considered invalid:

```
>>> print kvm_utils.get_default_koji_tag()
None
>>> print kvm_utils.KojiPkgSpec('kernel').is_valid()
False
```

```
>>> print kvm_utils.KojiPkgSpec(package='kernel').is_valid()
False
```

SEP = ':'

describe()
Describe this package specification, in a human friendly way

Returns package specification description

describe_invalid()
Describes why this is not valid, in a human friendly way

is_valid()
Checks if this package specification is valid.

Being valid means that it has enough and not conflicting information. It does not validate that the packages specified actually existe on the Koji server.

Returns True or False

parse(*text*)
Parses a textual representation of a package specification

Parameters **text** (*string*) – textual representation of a package in koji

to_text()
Return the textual representation of this package spec

The output should be consumable by parse() and produce the same package specification.

We find that it's acceptable to put the currently set default tag as the package explicit tag in the textual definition for completeness.

Returns package specification in a textual representation

class autotest.client.shared.utils_koji.**KojiScratchPkgSpec**(*text=''*, *user=None*, *task=None*, *subpackages=[]*)

Bases: object

A package specification syntax parser for Koji scratch builds

This holds information on user, task and subpackages to be fetched from koji and possibly installed (features external do this class).

New objects can be created either by providing information in the textual format or by using the actual parameters for user, task and subpackages. The textual format is useful for command line interfaces and configuration files, while using parameters is better for using this in a programatic fashion.

This package definition has a special behaviour: if no subpackages are specified, all packages of the chosen architecture (plus noarch packages) will match.

The following sets of examples are interchangeable. Specifying all packages from a scratch build (whose task id is 1000) sent by user jdoe:

```
>>> from kvm_utils import KojiScratchPkgSpec
>>> pkg = KojiScratchPkgSpec('jdoe:1000')
```

```
>>> pkg = KojiScratchPkgSpec(user=jdoe, task=1000)
```

Specifying some packages from a scratch build whose task id is 1000, sent by user jdoe:

```
>>> pkg = KojiScratchPkgSpec('jdoe:1000:kernel,kernel-devel')
```

```
>>> pkg = KojiScratchPkgSpec(user=jdoe, task=1000,
                             subpackages=['kernel', 'kernel-devel'])
```

SEP = ':'

parse(*text*)
Parses a textual representation of a package specification

Parameters **text** (*string*) – textual representation of a package in koji

class autotest.client.shared.utils_koji.**RPMFileNameInfo**(*filename*)
Simple parser for RPM based on information present on the filename itself

get_arch()
Returns just the architecture as present on the RPM filename

get_filename_without_arch()
Returns the filename without the architecture

This also excludes the RPM suffix, that is, removes the leading arch and RPM suffix.

get_filename_without_suffix()
Returns the filename without the default RPM suffix

get_nvr_info()
Returns a dictionary with the name, version and release components

If koji is not installed, this returns None

autotest.client.shared.utils_koji.**get_default_koji_tag**()

autotest.client.shared.utils_koji.**set_default_koji_tag**(*tag*)
Sets the default tag that will be used

utils_memory Module

autotest.client.shared.utils_memory.**drop_caches**()
Writes back all dirty pages to disk and clears all the caches.

autotest.client.shared.utils_memory.**freememtotal**()

autotest.client.shared.utils_memory.**get_buddy_info**(*chunk_sizes*, *nodes='all'*, *zones='all'*)
Get the fragement status of the host. It use the same method to get the page size in buddyinfo. 2^chunk_size * page_size The chunk_sizes can be string make up by all orders that you want to check splited with blank or a mathematical expression with '>', '<' or '='. For example: The input of chunk_size could be: "0 2 4" And the return will be: {'0': 3, '2': 286, '4': 687} if you are using expression: ">=9" the return will be: {'9': 63, '10': 225}

Parameters

- **chunk_size** (*string*) – The order number shows in buddyinfo. This is not the real page size.
- **nodes** (*string*) – The numa node that you want to check. Default value is all
- **zones** (*string*) – The memory zone that you want to check. Default value is all

Returns A dict using the chunk_size as the keys

Return type dict

autotest.client.shared.utils_memory.**get_huge_page_size**()

autotest.client.shared.utils_memory.**get_num_huge_pages**()

autotest.client.shared.utils_memory.**memtotal**()

autotest.client.shared.utils_memory.**node_size**()

autotest.client.shared.utils_memory.**numa_nodes**()

autotest.client.shared.utils_memory.**read_from_meminfo**(*key*)

autotest.client.shared.utils_memory.**read_from_numa_maps**(*pid*, *key*)
Get the process numa related info from numa_maps. This function only use to get the numbers like anon=1.

Parameters

- **pid** (*String*) – Process id
- **key** (*String*) – The item you want to check from numa_maps

Returns A dict using the address as the keys

Return type dict

autotest.client.shared.utils_memory.**read_from_smaps**(*pid*, *key*)
Get specific item value from the smaps of a process include all sections.

Parameters

- **pid** (*String*) – Process id
- **key** (*String*) – The item you want to check from smaps

Returns The value of the item in kb

Return type int

autotest.client.shared.utils_memory.**read_from_vmstat**(*key*)

Get specific item value from vmstat

Parameters **key** (*String*) – The item you want to check from vmstat

Returns The value of the item

Return type int

autotest.client.shared.utils_memory.**rounded_memtotal**()

autotest.client.shared.utils_memory.**set_num_huge_pages**(*num*)

version Module

Based on work from Douglas Creager <dcreager@dcreager.net>

Gets the current version number. If possible, this is the output of "git describe", modified to conform to the versioning scheme that setuptools uses. If "git describe" returns an error (most likely because we're in an unpacked copy of a release tarball, rather than in a git working copy), then we fall back on reading the contents of the RELEASE-VERSION file.

To use this script, simply import it your setup.py file, and use the results of get_version() as your package version:

from autotest.client.shared import version

setup(version=get_version(), . . .

)

This will automatically update the RELEASE-VERSION file, if necessary. Note that the RELEASE-VERSION file should *not* be checked into git; please add it to your top-level .gitignore file.

You'll probably want to distribute the RELEASE-VERSION file in your sdist tarballs; to do this, just create a MANIFEST.in file that contains the following line:

include RELEASE-VERSION

autotest.client.shared.version.**get_version**(*abbrev=4*)

Subpackages

backports Package

backports Package This module contains backported functions that are not present on Python 2.4 but are standard in more recent versions.

autotest.client.shared.backports.**all**(*iterable*)

From http://stackoverflow.com/questions/3785433/python-backports-for-some-methods :codeauthor: Tim Pietzcker http://stackoverflow.com/users/20670/tim-pietzcker licensed under cc-wiki with attribution required

autotest.client.shared.backports.**any**(*iterable*)
From http://stackoverflow.com/questions/3785433/python-backports-for-some-methods :codeauthor: Tim Pietzcker http://stackoverflow.com/users/20670/tim-pietzcker licensed under cc-wiki with attribution required

autotest.client.shared.backports.**bin**(*number*)
Adapted from http://code.activestate.com/recipes/576847/ :codeauthor: Vishal Sapre :license: MIT

A foolishly simple look-up method of getting binary string from an integer This happens to be faster than all other ways!!!

autotest.client.shared.backports.**next**(**args*)
Retrieve the next item from the iterator by calling its next() method. If default is given, it is returned if the iterator is exhausted, otherwise StopIteration is raised. New in version 2.6.

Parameters

- **iterator** (*iterator*) – the iterator
- **default** (*object*) – the value to return if the iterator raises StopIteration

Returns The object returned by iterator.next()

Return type object

Subpackages

collections Package

collections Package

OrderedDict Module Backport of OrderedDict() class that runs on Python 2.4, 2.5, 2.6, 2.7 and pypy. Passes Python2.7's test suite and incorporates all the latest updates.

Obtained from: http://code.activestate.com/recipes/576693-ordered-dictionary-for-py24/

class autotest.client.shared.backports.collections.OrderedDict.**OrderedDict**(**args*, ***kwds*)
Bases: dict

Dictionary that remembers insertion order

http://code.activestate.com/recipes/576693-ordered-dictionary-for-py24/ :codeauthor: Raymond Hettinger :license: MIT

clear() → None. Remove all items from od.

copy() → a shallow copy of od

classmethod fromkeys(*S*[, *v*]) → New ordered dictionary with keys from S
and values equal to v (which defaults to None).

items() → list of (key, value) pairs in od

iteritems()
od.iteritems -> an iterator over the (key, value) items in od

iterkeys() → an iterator over the keys in od

itervalues()
od.itervalues -> an iterator over the values in od

keys() → list of keys in od

pop (*k*[, *d*]) → v, remove specified key and return the corresponding value.

If key is not found, d is returned if given, otherwise KeyError is raised.

popitem () → (k, v), return and remove a (key, value) pair. Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault (*k*[, *d*]) → od.get(k,d), also set od[k]=d if k not in od

update (*E*, ***F*) → None. Update od from dict/iterable E and F. If E is a dict instance, does: for k in E: od[k] = E[k] If E has a .keys() method, does: for k in E.keys(): od[k] = E[k] Or if E is an iterable of items, does: for k, v in E: od[k] = v In either case, this is followed by: for k, v in F.items(): od[k] = v

values () → list of values in od

viewitems () → a set-like object providing a view on od's items

viewkeys () → a set-like object providing a view on od's keys

viewvalues () → an object providing a view on od's values

defaultdict Module Backport of the defaultdict module, obtained from: http://code.activestate.com/recipes/523034-emulate-collectionsdefaultdict/

class autotest.client.shared.backports.collections.defaultdict.**defaultdict**(*default_factory=None*, **a*, ***kw*)

Bases: dict

collections.defaultdict is a handy shortcut added in Python 2.5 which can be emulated in older versions of Python. This recipe tries to backport defaultdict exactly and aims to be safe to subclass and extend without worrying if the base class is in C or is being emulated.

http://code.activestate.com/recipes/523034-emulate-collectionsdefaultdict/ :codeauthor: Jason Kirtland :license: PSF

Changes: * replaced self.items() with self.iteritems() to fix Pickle bug as recommended by Aaron Lav * reformated with autopep8

copy ()

namedtuple Module This module contains a backport for collections.namedtuple obtained from http://code.activestate.com/recipes/500261-named-tuples/

autotest.client.shared.backports.collections.namedtuple.**namedtuple**(*typename*, *field_names*, *verbose=False*, *rename=False*)

Returns a new subclass of tuple with named fields.

```
>>> Point = namedtuple('Point', 'x y')
>>> Point.__doc__                   # docstring for the new class
'Point(x, y)'
>>> p = Point(11, y=22)             # instantiate with positional args or keywords
>>> p[0] + p[1]                     # indexable like a plain tuple
33
>>> x, y = p                        # unpack like a regular tuple
```

```
    >>> x, y
    (11, 22)
    >>> p.x + p.y                      # fields also accessible by name
    33
    >>> d = p._asdict()                # convert to a dictionary
    >>> d['x']
    11
    >>> Point(**d)                     # convert from a dictionary
    Point(x=11, y=22)
    >>> p._replace(x=100)              # _replace() is like str.replace() but targets named fields
    Point(x=100, y=22)
```

http://code.activestate.com/recipes/500261-named-tuples/ :codeauthor: Raymond Hettinger :license: PSF

Changes: * autopep8 reformatting

simplejson Package

simplejson Package

decoder Module

encoder Module

ordered_dict Module

scanner Module

tool Module

hosts Package

hosts Package This is a convenience module to import all available types of hosts.

Implementation details: You should 'import hosts' instead of importing every available host module.

base_classes Module This module defines the base classes for the Host hierarchy.

Implementation details: You should import the "hosts" package instead of importing each type of host.

Host: a machine on which you can run programs

class autotest.client.shared.hosts.base_classes.**Host**(**args*, ***dargs*)
Bases: object

This class represents a machine on which you can run programs.

It may be a local machine, the one autoserv is running on, a remote machine or a virtual machine.

Implementation details: This is an abstract class, leaf subclasses must implement the methods listed here. You must not instantiate this class but should instantiate one of those leaf subclasses.

When overriding methods that raise NotImplementedError, the leaf class is fully responsible for the implementation and should not chain calls to super. When overriding methods that are a NOP in Host, the subclass should chain calls to super(). The criteria for fitting a new method into one category or the other should be:

1. If two separate generic implementations could reasonably be concatenated, then the abstract implementation should pass and subclasses should chain calls to super.
2. If only one class could reasonably perform the stated function (e.g. two separate run() implementations cannot both be executed) then the method should raise NotImplementedError in Host, and the implementor should NOT chain calls to super, to ensure that only one implementation ever gets executed.

DEFAULT_REBOOT_TIMEOUT = 1800

HARDWARE_REPAIR_REQUEST_THRESHOLD = 4

HOURS_TO_WAIT_FOR_RECOVERY = 2.5

WAIT_DOWN_REBOOT_TIMEOUT = 840

WAIT_DOWN_REBOOT_WARNING = 540

check_diskspace(*path*, *gb*)
Raises an error if path does not have at least gb GB free.

:param path The path to check for free disk space. :param gb A floating point number to compare with a granularity

of 1 MB.

1000 based SI units are used.

:raise AutoservDiskFullHostError if path has less than gb GB free.

check_partitions(*root_part*, *filter_func=None*)
Compare the contents of /proc/partitions with those of /proc/mounts and raise exception in case unmounted partitions are found

root_part: in Linux /proc/mounts will never directly mention the root partition as being mounted on / instead it will say that /dev/root is mounted on /. Thus require this argument to filter out the root_part from the ones checked to be mounted

filter_func: unnary predicate for additional filtering out of partitions required to be mounted

Raise: error.AutoservHostError if unfiltered unmounted partition found

cleanup()

cleanup_kernels(*boot_dir='/boot'*)
Remove any kernel image and associated files (vmlinux, system.map, modules) for any image found in the boot directory that is not referenced by entries in the bootloader configuration.

Parameters **boot_dir** – boot directory path string, default '/boot'

close()

disable_ipfilters()
Allow all network packets in and out of the host.

enable_ipfilters()
Re-enable the IP filters disabled from disable_ipfilters()

erase_dir_contents(*path*, *ignore_status=True*, *timeout=3600*)
Empty a given directory path contents.

get_arch()
Get the hardware architecture of the remote machine.

get_autodir()

get_boot_id(*timeout=60*)
Get a unique ID associated with the current boot.

Should return a string with the semantics such that two separate calls to Host.get_boot_id() return the same string if the host did not reboot between the two calls, and two different strings if it has rebooted at least once between the two calls.

:param timeout The number of seconds to wait before timing out.

Returns A string unique to this boot or None if not available.

get_cmdline()
Get the kernel command line of the remote machine.

get_file(*source*, *dest*, *delete_dest=False*)

get_kernel_ver()
Get the kernel version of the remote machine.

get_meminfo()
Get the kernel memory info (/proc/meminfo) of the remote machine and return a dictionary mapping the various statistics.

get_num_cpu()
Get the number of CPUs in the host according to /proc/cpuinfo.

get_open_func(*use_cache=True*)
Defines and returns a function that may be used instead of built-in open() to open and read files. The returned function is implemented by using self.run('cat <file>') and may cache the results for the same filename.

:param use_cache Cache results of self.run('cat <filename>') for the same filename

Returns a function that can be used instead of built-in open()

get_tmp_dir()

get_wait_up_processes()
Gets the list of local processes to wait for in wait_up.

install(*installableObject*)

is_shutting_down()
Indicates is a machine is currently shutting down.

is_up()

job = None

list_files_glob(*glob*)
Get a list of files on a remote host given a glob pattern path.

log_kernel()
Helper method for logging kernel information into the status logs. Intended for cases where the "current" kernel is not really defined and we want to explicitly log it. Does nothing if this host isn't actually associated with a job.

log_reboot(*reboot_func*)
Decorator for wrapping a reboot in a group for status logging purposes. The reboot_func parameter should be an actual function that carries out the reboot.

machine_install()

path_exists(*path*)
Determine if path exists on the remote machine.

reboot()

reboot_followup(**args*, ***dargs*)

reboot_setup(**args*, ***dargs*)

record(**args*, ***dargs*)
Helper method for recording status logs against Host.job that silently becomes a NOP if Host.job is not available. The args and dargs are passed on to Host.job.record unchanged.

repair_filesystem_only()
perform file system repairs only

repair_full()

repair_full_disk(*mountpoint*)

repair_software_only()
perform software repairs only

repair_with_protection(*protection_level*)
Perform the maximal amount of repair within the specified protection level.

Parameters **protection_level** – the protection level to use for limiting repairs, a host_protections.Protection

request_hardware_repair()
Should somehow request (send a mail?) for hardware repairs on this machine. The implementation can either return by raising the special error.AutoservHardwareRepairRequestedError exception or can try to wait until the machine is repaired and then return normally.

run(*command*, *timeout=3600*, *ignore_status=False*, *stdout_tee=<object object>*, *stderr_tee=<object object>*, *stdin=None*, *args=()*)
Run a command on this host.

Parameters

- **command** – the command line string
- **timeout** – time limit in seconds before attempting to kill the running process. The run() function will take a few seconds longer than 'timeout' to complete if it has to kill the process.
- **ignore_status** – do not raise an exception, no matter what the exit code of the command is.
- **stdout_tee/stderr_tee** – where to tee the stdout/stderr
- **stdin** – stdin to pass (a string) to the executed command
- **args** – sequence of strings to pass as arguments to command by quoting them in " and escaping their contents if necessary

Returns a utils.CmdResult object

Raises AutotestHostRunError the exit code of the command execution was not 0 and ignore_status was not enabled

run_output(*command*, **args*, ***dargs*)

send_file(*source*, *dest*, *delete_dest=False*)

set_autodir()

setup()

start_loggers()
Called to start continuous host logging.

stop_loggers()
Called to stop continuous host logging.

symlink_closure(*paths*)
Given a sequence of path strings, return the set of all paths that can be reached from the initial set by following symlinks.

Parameters **paths** – sequence of path strings.

Returns a sequence of path strings that are all the unique paths that can be reached from the given ones after following symlinks.

sysrq_reboot()

verify()

verify_connectivity()

verify_hardware()

verify_software()

wait_down(*timeout=None*, *warning_timer=None*, *old_boot_id=None*)

wait_for_restart(*timeout=1800*, *down_timeout=840*, *down_warning=540*, *log_failure=True*, *old_boot_id=None*, ***dargs*)
Wait for the host to come back from a reboot. This is a generic implementation based entirely on wait_up and wait_down.

wait_up(*timeout=None*)

common Module

test_utils Package

config_change_validation Module Module for testing config file changes.

author Kristof Katus and Plamen Dimitrov

copyright Intra2net AG 2012

@license: GPL v2

autotest.client.shared.test_utils.config_change_validation.**assert_config_change**(*actual_result*, *expected_result*)
Wrapper of the upper method returning boolean true if no config changes were detected.

autotest.client.shared.test_utils.config_change_validation.**assert_config_change_dict**(*actual_r*, *expected_*)
Calculates unexpected line changes.

The arguments actual_result and expected_results are of the same data structure type: Dict[file_path] –> (adds, removes), where adds = [added_line, ...] and removes = [removed_line, ...].

The return value has the following structure: Dict[file_path] –> (unexpected_adds,

not_present_adds, unexpected_removes, not_present_removes)

autotest.client.shared.test_utils.config_change_validation.**del_temp_file_copies**(*file_paths*)
Deletes all the provided files

autotest.client.shared.test_utils.config_change_validation.**extract_config_changes**(*file_paths*, *compared_file_p*
Extracts diff information based on the new and temporarily saved old config files

Returns a dictionary of file path and corresponding diff information key-value pairs.

autotest.client.shared.test_utils.config_change_validation.**get_temp_file_path**(*file_path*)
Generates a temporary filename

autotest.client.shared.test_utils.config_change_validation.**make_temp_file_copies**(*file_paths*)
Creates temporary copies of the provided files

autotest.client.shared.test_utils.config_change_validation.**parse_unified_diff_output**(*lines*)
Parses the unified diff output of two files

Returns a pair of adds and removes, where each is a list of trimmed lines

autotest.client.shared.test_utils.config_change_validation.**print_change_diffs**(*change_diffs*)
Pretty prints the output of the evaluate_config_changes function

functools_24 Module

autotest.client.shared.test_utils.functools_24.**compose**(**args*)
autotest.client.shared.test_utils.functools_24.**fastcut**(**sargs*, ***skw*)

mock Module

exception autotest.client.shared.test_utils.mock.**CheckPlaybackError**
Bases: exceptions.Exception

Raised when mock playback does not match recorded calls.

class autotest.client.shared.test_utils.mock.**SaveDataAfterCloseStringIO**(*buf=''*)
Bases: StringIO.StringIO

Saves the contents in a final_data property when close() is called.

Useful as a mock output file object to test both that the file was closed and what was written.

Properties:

final_data: Set to the StringIO's getvalue() data when close() is called. None if close() has not been called.

close()

final_data = None

exception autotest.client.shared.test_utils.mock.**StubNotFoundError**
Bases: exceptions.Exception

Raised when god is asked to unstub an attribute that was not stubbed

class autotest.client.shared.test_utils.mock.**anything_comparator**
Bases: *autotest.client.shared.test_utils.mock.argument_comparator*

is_satisfied_by(*parameter*)

class autotest.client.shared.test_utils.mock.**argument_comparator**
Bases: object

is_satisfied_by(*parameter*)

class autotest.client.shared.test_utils.mock.**base_mapping**(*symbol*, *return_obj*, **args*, ***dargs*)

Bases: object

match(**args*, ***dargs*)

class autotest.client.shared.test_utils.mock.**equality_comparator**(*value*)

Bases: *autotest.client.shared.test_utils.mock.argument_comparator*

is_satisfied_by(*parameter*)

class autotest.client.shared.test_utils.mock.**function_any_args_mapping**(*symbol*, *return_val*, **args*, ***dargs*)

Bases: *autotest.client.shared.test_utils.mock.function_mapping*

A mock function mapping that doesn't verify its arguments.

match(**args*, ***dargs*)

class autotest.client.shared.test_utils.mock.**function_mapping**(*symbol*, *return_val*, **args*, ***dargs*)

Bases: *autotest.client.shared.test_utils.mock.base_mapping*

and_raises(*error*)

and_return(*return_obj*)

class autotest.client.shared.test_utils.mock.**is_instance_comparator**(*cls*)

Bases: *autotest.client.shared.test_utils.mock.argument_comparator*

is_satisfied_by(*parameter*)

class autotest.client.shared.test_utils.mock.**is_string_comparator**

Bases: *autotest.client.shared.test_utils.mock.argument_comparator*

is_satisfied_by(*parameter*)

class autotest.client.shared.test_utils.mock.**mask_function**(*symbol*, *original_function*, *default_return_val=None*, *record=None*, *playback=None*)

Bases: *autotest.client.shared.test_utils.mock.mock_function*

run_original_function(**args*, ***dargs*)

class autotest.client.shared.test_utils.mock.**mock_class**(*cls*, *name*, *default_ret_val=None*, *record=None*, *playback=None*)

Bases: object

class autotest.client.shared.test_utils.mock.**mock_function**(*symbol*, *default_return_val=None*, *record=None*, *playback=None*)

Bases: object

expect_any_call()

Like expect_call but don't give a hoot what arguments are passed.

expect_call(**args*, ***dargs*)

class autotest.client.shared.test_utils.mock.**mock_god**(*debug=False*, *fail_fast=True*, *ut=None*)

Bases: object

NONEXISTENT_ATTRIBUTE = <object object>

check_playback()
Report any errors that were encounterd during calls to __method_playback().

create_mock_class(*cls*, *name*, *default_ret_val=None*)
Given something that defines a namespace cls (class, object, module), and a (hopefully unique) name, will create a mock_class object with that name and that possesses all the public attributes of cls. default_ret_val sets the default_ret_val on all methods of the cls mock.

create_mock_class_obj(*cls*, *name*, *default_ret_val=None*)

create_mock_function(*symbol*, *default_return_val=None*)
create a mock_function with name symbol and default return value of default_ret_val.

mock_io()
Mocks and saves the stdout & stderr output

mock_up(*obj*, *name*, *default_ret_val=None*)
Given an object (class instance or module) and a registration name, then replace all its methods with mock function objects (passing the orignal functions to the mock functions).

set_fail_fast(*fail_fast*)

stub_class(*namespace*, *symbol*)

stub_class_method(*cls*, *symbol*)

stub_function(*namespace*, *symbol*)

stub_function_to_return(*namespace*, *symbol*, *object_to_return*)
Stub out a function with one that always returns a fixed value.

:param namespace The namespace containing the function to stub out. :param symbol The attribute within the namespace to stub out. :param object_to_return The value that the stub should return whenever

it is called.

stub_with(*namespace*, *symbol*, *new_attribute*)

unmock_io()
Restores the stdout & stderr, and returns both output strings

unstub(*namespace*, *symbol*)

unstub_all()

class autotest.client.shared.test_utils.mock.**regex_comparator**(*pattern*, *flags=0*)
Bases: *autotest.client.shared.test_utils.mock.argument_comparator*

is_satisfied_by(*parameter*)

mock_demo Module

mock_demo_MUT Module

autotest.client.shared.test_utils.mock_demo_MUT.**do_create_stuff**()

unittest Module Python unit testing framework, based on Erich Gamma's JUnit and Kent Beck's Smalltalk testing framework.

This module contains the core framework classes that form the basis of specific test cases and suites (TestCase, TestSuite etc.), and also a text-based utility class for running the tests and reporting the results

(TextTestRunner).

Simple usage:

import unittest

class IntegerArithmenticTestCase(unittest.TestCase):

def testAdd(self): ## test method names begin 'test*' self.assertEqual((1 + 2), 3) self.assertEqual(0 + 1, 1)

def testMultiply(self): self.assertEqual((0 * 10), 0) self.assertEqual((5 * 8), 40)

if __name__ == '__main__': unittest.main()

Further information is available in the bundled documentation, and from

http://docs.python.org/library/unittest.html

Copyright (c) 1999-2003 Steve Purcell Copyright (c) 2003-2009 Python Software Foundation Copyright (c) 2009 Garrett Cooper This module is free software, and you may redistribute it and/or modify it under the same terms as Python itself, so long as this copyright message and disclaimer are retained in their original form.

IN NO EVENT SHALL THE AUTHOR BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS CODE, EVEN IF THE AUTHOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE AUTHOR SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE CODE PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, AND THERE IS NO OBLIGATION WHATSOEVER TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

Garrett: This module was backported using source from r71263 with fixes noted in Issue 5771.

class autotest.client.shared.test_utils.unittest.**TestResult**
Bases: object

Holder for test result information.

Test results are automatically managed by the TestCase and TestSuite classes, and do not need to be explicitly manipulated by writers of tests.

Each instance holds the total number of tests run, and collections of failures and errors that occurred among those test runs. The collections contain tuples of (testcase, exceptioninfo), where exceptioninfo is the formatted traceback of the error that occurred.

addError(*test*, *err*)
Called when an error has occurred. 'err' is a tuple of values as returned by sys.exc_info().

addExpectedFailure(*test*, *err*)
Called when an expected failure/error occurred.

addFailure(*test*, *err*)
Called when an error has occurred. 'err' is a tuple of values as returned by sys.exc_info().

addSkip(*test*, *reason*)
Called when a test is skipped.

addSuccess(*test*)
Called when a test has completed successfully

addUnexpectedSuccess(*test*)
Called when a test was expected to fail, but succeed.

startTest(*test*)
Called when the given test is about to be run

stop()
Indicates that the tests should be aborted

stopTest(*test*)
Called when the given test has been run

wasSuccessful()
Tells whether or not this result was a success

class autotest.client.shared.test_utils.unittest.**TestCase**(*methodName='runTest'*)
Bases: object

A class whose instances are single test cases.

By default, the test code itself should be placed in a method named 'runTest'.

If the fixture may be used for many test cases, create as many test methods as are needed. When instantiating such a TestCase subclass, specify in the constructor arguments the name of the test method that the instance is to execute.

Test authors should subclass TestCase for their own tests. Construction and deconstruction of the test's environment ('fixture') can be implemented by overriding the 'setUp' and 'tearDown' methods respectively.

If it is necessary to override the __init__ method, the base class __init__ method must always be called. It is important that subclasses should not change the signature of their __init__ method, since instances of the classes are instantiated automatically by parts of the framework in order to be run.

addTypeEqualityFunc(*typeobj*, *function*)
Add a type specific assertEqual style function to compare a type.

This method is for use by TestCase subclasses that need to register their own type equality functions to provide nicer error messages.

Args:

typeobj: The data type to call this function on when both values are of the same type in assertEqual().

function: The callable taking two arguments and an optional msg= argument that raises self.failureException with a useful error message when the two arguments are not equal.

assertAlmostEqual(*first*, *second*, *places=7*, *msg=None*)
Fail if the two objects are unequal as determined by their difference rounded to the given number of decimal places (default 7) and comparing to zero.

Note that decimal places (from zero) are usually not the same as significant digits (measured from the most significant digit).

assertAlmostEquals(*first*, *second*, *places=7*, *msg=None*)
Fail if the two objects are unequal as determined by their difference rounded to the given number of decimal places (default 7) and comparing to zero.

Note that decimal places (from zero) are usually not the same as significant digits (measured from the most significant digit).

assertDictContainsSubset(*expected*, *actual*, *msg=None*)
Checks whether actual is a superset of expected.

assertDictEqual(*d1*, *d2*, *msg=None*)

assertEqual(*first*, *second*, *msg=None*)
Fail if the two objects are unequal as determined by the '==' operator.

assertEquals(*first*, *second*, *msg=None*)
Fail if the two objects are unequal as determined by the '==' operator.

assertFalse(*expr*, *msg=None*)
Fail the test if the expression is true.

assertGreater(*a*, *b*, *msg=None*)
Just like self.assertTrue(a > b), but with a nicer default message.

assertGreaterEqual(*a*, *b*, *msg=None*)
Just like self.assertTrue(a >= b), but with a nicer default message.

assertIn(*member*, *container*, *msg=None*)
Just like self.assertTrue(a in b), but with a nicer default message.

assertIs(*expr1*, *expr2*, *msg=None*)
Just like self.assertTrue(a is b), but with a nicer default message.

assertIsNone(*obj*, *msg=None*)
Same as self.assertTrue(obj is None), with a nicer default message.

assertIsNot(*expr1*, *expr2*, *msg=None*)
Just like self.assertTrue(a is not b), but with a nicer default message.

assertIsNotNone(*obj*, *msg=None*)
Included for symmetry with assertIsNone.

assertLess(*a*, *b*, *msg=None*)
Just like self.assertTrue(a < b), but with a nicer default message.

assertLessEqual(*a*, *b*, *msg=None*)
Just like self.assertTrue(a <= b), but with a nicer default message.

assertListEqual(*list1*, *list2*, *msg=None*)
A list-specific equality assertion.

Args: list1: The first list to compare. list2: The second list to compare. msg: Optional message to use on failure instead of a list of

differences.

assertMultiLineEqual(*first*, *second*, *msg=None*)
Assert that two multi-line strings are equal.

assertNotAlmostEqual(*first*, *second*, *places=7*, *msg=None*)
Fail if the two objects are equal as determined by their difference rounded to the given number of decimal places (default 7) and comparing to zero.

Note that decimal places (from zero) are usually not the same as significant digits (measured from the most significant digit).

assertNotAlmostEquals(*first*, *second*, *places=7*, *msg=None*)
Fail if the two objects are equal as determined by their difference rounded to the given number of decimal places (default 7) and comparing to zero.

Note that decimal places (from zero) are usually not the same as significant digits (measured from the most significant digit).

assertNotEqual(*first*, *second*, *msg=None*)
Fail if the two objects are equal as determined by the '==' operator.

assertNotEquals(*first*, *second*, *msg=None*)
Fail if the two objects are equal as determined by the '==' operator.

assertNotIn(*member*, *container*, *msg=None*)
Just like self.assertTrue(a not in b), but with a nicer default message.

assertRaises(*excClass*, *callableObj=None*, **args*, ***kwargs*)
Fail unless an exception of class excClass is thrown by callableObj when invoked with arguments args and keyword arguments kwargs. If a different type of exception is thrown, it will not be caught, and the test case will be deemed to have suffered an error, exactly as for an unexpected exception.

If called with callableObj omitted or None, will return a context object used like this:

```
with self.assertRaises(some_error_class):
    do_something()
```

assertRaisesRegexp(*expected_exception*, *expected_regexp*, *callable_obj=None*, **args*, ***kwargs*)
Asserts that the message in a raised exception matches a regexp.

Args: expected_exception: Exception class expected to be raised. expected_regexp: Regexp (re pattern object or string) expected

to be found in error message.

callable_obj: Function to be called. args: Extra args. kwargs: Extra kwargs.

assertRegexpMatches(*text*, *expected_regex*, *msg=None*)

assertSameElements(*expected_seq*, *actual_seq*, *msg=None*)
An unordered sequence specific comparison.

Raises with an error message listing which elements of expected_seq are missing from actual_seq and vice versa if any.

assertSequenceEqual(*seq1*, *seq2*, *msg=None*, *seq_type=None*)
An equality assertion for ordered sequences (like lists and tuples).

For the purposes of this function, a valid orderd sequence type is one which can be indexed, has a length, and has an equality operator.

Args: seq1: The first sequence to compare. seq2: The second sequence to compare. seq_type: The expected datatype of the sequences, or None if no

datatype should be enforced.

msg: Optional message to use on failure instead of a list of differences.

assertSetEqual(*set1*, *set2*, *msg=None*)
A set-specific equality assertion.

Args: set1: The first set to compare. set2: The second set to compare. msg: Optional message to use on failure instead of a list of

differences.

For more general containership equality, assertSameElements will work with things other than sets. This uses ducktyping to support different types of sets, and is optimized for sets specifically (parameters must support a difference method).

assertTrue(*expr*, *msg=None*)
Fail the test unless the expression is true.

assertTupleEqual(*tuple1*, *tuple2*, *msg=None*)
A tuple-specific equality assertion.

Args: tuple1: The first tuple to compare. tuple2: The second tuple to compare. msg: Optional message to use on failure instead of a list of

differences.

assert_(*expr*, *msg=None*)
Fail the test unless the expression is true.

countTestCases()

debug()
Run the test without collecting errors in a TestResult

defaultTestResult()

fail(*msg=None*)
Fail immediately, with the given message.

failIf(**args*, ***kwargs*)

failIfAlmostEqual(**args*, ***kwargs*)

failIfEqual(**args*, ***kwargs*)

failUnless(**args*, ***kwargs*)

failUnlessAlmostEqual(**args*, ***kwargs*)

failUnlessEqual(**args*, ***kwargs*)

failUnlessRaises(**args*, ***kwargs*)

failureException
alias of `AssertionError`

id()

longMessage = False

run(*result=None*)

setUp()
Hook method for setting up the test fixture before exercising it.

shortDescription()
Returns both the test method name and first line of its docstring.

If no docstring is given, only returns the method name.

This method overrides unittest.TestCase.shortDescription(), which only returns the first line of the docstring, obscuring the name of the test upon failure.

skipTest(*reason*)
Skip this test.

tearDown()
Hook method for deconstructing the test fixture after testing it.

class autotest.client.shared.test_utils.unittest.**TestSuite**(*tests=()*)
Bases: object

A test suite is a composite test consisting of a number of TestCases.

For use, create an instance of TestSuite, then add test case instances. When all tests have been added, the suite can be passed to a test runner, such as TextTestRunner. It will run the individual test cases in the order in which they were added, aggregating the results. When subclassing, do not forget to call the base class constructor.

addTest(*test*)

addTests(*tests*)

countTestCases()

debug()
Run the tests without collecting errors in a TestResult

run(*result*)

class autotest.client.shared.test_utils.unittest.**ClassTestSuite**(*tests*, *class_collected_from*)
Bases: *autotest.client.shared.test_utils.unittest.TestSuite*

Suite of tests derived from a single TestCase class.

id()

run(*result*)

shortDescription()

class autotest.client.shared.test_utils.unittest.**TextTestRunner**(*stream=<open file '<stderr>', mode 'w'>*, *descriptions=1*, *verbosity=1*)
Bases: object

A test runner class that displays results in textual form.

It prints out the names of tests as they are run, errors as they occur, and a summary of the results at the end of the test run.

run(*test*)
Run the given test case or test suite.

class autotest.client.shared.test_utils.unittest.**TestLoader**
Bases: object

This class is responsible for loading tests according to various criteria and returning them wrapped in a TestSuite

classSuiteClass
alias of *ClassTestSuite*

getTestCaseNames(*testCaseClass*)
Return a sorted sequence of method names found within testCaseClass

loadTestsFromModule(*module*)
Return a suite of all tests cases contained in the given module

loadTestsFromName(*name*, *module=None*)
Return a suite of all tests cases given a string specifier.

The name may resolve either to a module, a test case class, a test method within a test case class, or a callable object which returns a TestCase or TestSuite instance.

The method optionally resolves the names relative to a given module.

loadTestsFromNames(*names*, *module=None*)
Return a suite of all tests cases found using the given sequence of string specifiers. See 'loadTestsFromName()'.

loadTestsFromTestCase(*testCaseClass*)
Return a suite of all tests cases contained in testCaseClass

sortTestMethodsUsing()
cmp(x, y) -> integer

Return negative if x<y, zero if x==y, positive if x>y.

suiteClass
alias of *TestSuite*

testMethodPrefix = 'test'

class autotest.client.shared.test_utils.unittest.**FunctionTestCase**(*testFunc*, *setUp=None*, *tearDown=None*, *description=None*)
Bases: *autotest.client.shared.test_utils.unittest.TestCase*

A test case that wraps a test function.

This is useful for slipping pre-existing test functions into the unittest framework. Optionally, set-up and tidy-up functions can be supplied. As with TestCase, the tidy-up ('tearDown') function will always be called if the set-up ('setUp') function ran successfully.

id()

runTest()

setUp()

shortDescription()

tearDown()

autotest.client.shared.test_utils.unittest.**main**
alias of TestProgram

exception autotest.client.shared.test_utils.unittest.**SkipTest**
Bases: exceptions.Exception

Raise this exception in a test to skip it.

Usually you can use TestResult.skip() or one of the skipping decorators instead of raising this directly.

autotest.client.shared.test_utils.unittest.**skip**(*reason*)
Unconditionally skip a test.

autotest.client.shared.test_utils.unittest.**skipIf**(*condition*, *reason*)
Skip a test if the condition is true.

autotest.client.shared.test_utils.unittest.**skipUnless**(*condition*, *reason*)
Skip a test unless the condition is true.

autotest.client.shared.test_utils.unittest.**expectedFailure**(*func*)

autotest.client.shared.test_utils.unittest.**getTestCaseNames**(*testCaseClass*, *prefix*, *sortUsing=<built-in function cmp>*)

autotest.client.shared.test_utils.unittest.**makeSuite**(*testCaseClass*, *prefix='test'*, *sortUsing=<built-in function cmp>*, *suiteClass=<class 'autotest.client.shared.test_utils.unittest.TestSuite'>*)

autotest.client.shared.test_utils.unittest.**findTestCases**(*module*, *prefix='test'*, *sortUsing=<built-in function cmp>*, *suiteClass=<class 'autotest.client.shared.test_utils.unittest.TestSuite'>*)

4.41.4 tools Package

JUnit_api Module

class autotest.client.tools.JUnit_api.**errorType**(*message=None*, *type_=None*, *valueOf_=None*)

Bases: autotest.client.tools.JUnit_api.GeneratedsSuper

The error message. e.g., if a java exception is thrown, the return value of getMessage()The type of error that occurred. e.g., if a java execption is thrown the full class name of the exception.

build(*node*)

buildAttributes(*node*, *attrs*, *already_processed*)

buildChildren(*child_*, *node*, *nodeName_*, *fromsubclass_=False*)

export(*outfile*, *level*, *namespace_=''*, *name_='errorType'*, *namespacedef_=''*)

exportAttributes(*outfile*, *level*, *already_processed*, *namespace_=''*, *name_='errorType'*)

exportChildren(*outfile*, *level*, *namespace_=''*, *name_='errorType'*, *fromsubclass_=False*)

exportLiteral(*outfile*, *level*, *name_='errorType'*)

exportLiteralAttributes(*outfile*, *level*, *already_processed*, *name_*)

exportLiteralChildren(*outfile*, *level*, *name_*)

static factory(**args_*, ***kwargs_*)

get_message()

get_type()

get_valueOf_()

hasContent_()

set_message(*message*)

set_type(*type_*)

set_valueOf_(*valueOf_*)

subclass = None

superclass = None

class autotest.client.tools.JUnit_api.**failureType**(*message=None*, *type_=None*, *valueOf_=None*)

Bases: autotest.client.tools.JUnit_api.GeneratedsSuper

The message specified in the assertThe type of the assert.

build(*node*)

buildAttributes(*node*, *attrs*, *already_processed*)

buildChildren(*child_*, *node*, *nodeName_*, *fromsubclass_=False*)

export(*outfile*, *level*, *namespace_=''*, *name_='failureType'*, *namespacedef_=''*)

exportAttributes(*outfile*, *level*, *already_processed*, *namespace_=''*, *name_='failureType'*)

exportChildren(*outfile*, *level*, *namespace_=''*, *name_='failureType'*, *fromsubclass_=False*)

exportLiteral(*outfile*, *level*, *name_='failureType'*)

exportLiteralAttributes(*outfile*, *level*, *already_processed*, *name_*)

exportLiteralChildren(*outfile*, *level*, *name_*)

static factory(**args_*, ***kwargs_*)

get_message()

get_type()

get_valueOf_()

hasContent_()

set_message(*message*)

set_type(*type_*)

set_valueOf_(*valueOf_*)

subclass = None

superclass = None

class autotest.client.tools.JUnit_api.**propertiesType**(*property=None*)

Bases: autotest.client.tools.JUnit_api.GeneratedsSuper

add_property(*value*)

build(*node*)

buildAttributes(*node*, *attrs*, *already_processed*)

buildChildren(*child_*, *node*, *nodeName_*, *fromsubclass_=False*)

export(*outfile*, *level*, *namespace_=''*, *name_='propertiesType'*, *namespacedef_=''*)

exportAttributes(*outfile*, *level*, *already_processed*, *namespace_=''*, *name_='propertiesType'*)

exportChildren(*outfile*, *level*, *namespace_=''*, *name_='propertiesType'*, *fromsubclass_=False*)

exportLiteral(*outfile*, *level*, *name_='propertiesType'*)

exportLiteralAttributes(*outfile*, *level*, *already_processed*, *name_*)

exportLiteralChildren(*outfile*, *level*, *name_*)

static factory(**args_*, ***kwargs_*)

get_property()

hasContent_()

insert_property(*index*, *value*)

set_property(*property*)

subclass = None

superclass = None

class autotest.client.tools.JUnit_api.**propertyType**(*name=None*, *value=None*)
Bases: autotest.client.tools.JUnit_api.GeneratedsSuper

build(*node*)

buildAttributes(*node*, *attrs*, *already_processed*)

buildChildren(*child_*, *node*, *nodeName_*, *fromsubclass_=False*)

export(*outfile*, *level*, *namespace_=''*, *name_='propertyType'*, *namespacedef_=''*)

exportAttributes(*outfile*, *level*, *already_processed*, *namespace_=''*, *name_='propertyType'*)

exportChildren(*outfile*, *level*, *namespace_=''*, *name_='propertyType'*, *fromsubclass_=False*)

exportLiteral(*outfile*, *level*, *name_='propertyType'*)

exportLiteralAttributes(*outfile*, *level*, *already_processed*, *name_*)

exportLiteralChildren(*outfile*, *level*, *name_*)

static **factory**(**args_*, ***kwargs_*)

get_name()

get_value()

hasContent_()

set_name(*name*)

set_value(*value*)

subclass = None

superclass = None

class autotest.client.tools.JUnit_api.**system_err**
Bases: autotest.client.tools.JUnit_api.GeneratedsSuper

Data that was written to standard error while the test was executed

build(*node*)

buildAttributes(*node*, *attrs*, *already_processed*)

buildChildren(*child_*, *node*, *nodeName_*, *fromsubclass_=False*)

export(*outfile*, *level*, *namespace_=''*, *name_='system-err'*, *namespacedef_=''*)

exportAttributes(*outfile*, *level*, *already_processed*, *namespace_=''*, *name_='system-err'*)

exportChildren(*outfile*, *level*, *namespace_=''*, *name_='system-err'*, *fromsubclass_=False*)

exportLiteral(*outfile*, *level*, *name_='system-err'*)

exportLiteralAttributes(*outfile*, *level*, *already_processed*, *name_*)

exportLiteralChildren(*outfile*, *level*, *name_*)

static **factory**(**args_*, ***kwargs_*)

hasContent_()

subclass = None

superclass = None

class autotest.client.tools.JUnit_api.**system_out**
Bases: autotest.client.tools.JUnit_api.GeneratedsSuper

Data that was written to standard out while the test was executed

build(*node*)

buildAttributes(*node*, *attrs*, *already_processed*)

buildChildren(*child_*, *node*, *nodeName_*, *fromsubclass_=False*)

export(*outfile*, *level*, *namespace_=''*, *name_='system-out'*, *namespacedef_=''*)

exportAttributes(*outfile*, *level*, *already_processed*, *namespace_=''*, *name_='system-out'*)

exportChildren(*outfile*, *level*, *namespace_=''*, *name_='system-out'*, *fromsubclass_=False*)

exportLiteral(*outfile*, *level*, *name_='system-out'*)

exportLiteralAttributes(*outfile*, *level*, *already_processed*, *name_*)

exportLiteralChildren(*outfile*, *level*, *name_*)

static factory(**args_*, ***kwargs_*)

hasContent_()

subclass = None

superclass = None

class autotest.client.tools.JUnit_api.**testcaseType**(*classname=None*, *name=None*, *time=None*, *error=None*, *failure=None*)
Bases: autotest.client.tools.JUnit_api.GeneratedsSuper

Name of the test methodFull class name for the class the test method is in.Time taken (in seconds) to execute the test

build(*node*)

buildAttributes(*node*, *attrs*, *already_processed*)

buildChildren(*child_*, *node*, *nodeName_*, *fromsubclass_=False*)

export(*outfile*, *level*, *namespace_=''*, *name_='testcaseType'*, *namespacedef_=''*)

exportAttributes(*outfile*, *level*, *already_processed*, *namespace_=''*, *name_='testcaseType'*)

exportChildren(*outfile*, *level*, *namespace_=''*, *name_='testcaseType'*, *fromsubclass_=False*)

exportLiteral(*outfile*, *level*, *name_='testcaseType'*)

exportLiteralAttributes(*outfile*, *level*, *already_processed*, *name_*)

exportLiteralChildren(*outfile*, *level*, *name_*)

static factory(**args_*, ***kwargs_*)

get_classname()

get_error()

get_failure()

get_name()

get_time()

hasContent_()

set_classname(*classname*)

set_error(*error*)

set_failure(*failure*)

set_name(*name*)

set_time(*time*)

subclass = None

superclass = None

class autotest.client.tools.JUnit_api.**testsuite**(*tests=None*, *errors=None*, *name=None*, *timestamp=None*, *hostname=None*, *time=None*, *failures=None*, *properties=None*, *testcase=None*, *system_out=None*, *system_err=None*, *extensiontype_=None*)

Bases: autotest.client.tools.JUnit_api.GeneratedsSuper

Contains the results of exexuting a testsuiteFull class name of the test for non-aggregated testsuite documents. Class name without the package for aggregated testsuites documentswhen the test was executed. Timezone may not be specified.Host on which the tests were executed. 'localhost' should be used if the hostname cannot be determined.The total number of tests in the suiteThe total number of tests in the suite that failed. A failure is a test which the code has explicitly failed by using the mechanisms for that purpose. e.g., via an assertEqualsThe total number of tests in the suite that errorrd. An errored test is one that had an unanticipated problem. e.g., an unchecked throwable; or a problem with the implementation of the test.Time taken (in seconds) to execute the tests in the suite

add_testcase(*value*)

build(*node*)

buildAttributes(*node*, *attrs*, *already_processed*)

buildChildren(*child_*, *node*, *nodeName_*, *fromsubclass_=False*)

export(*outfile*, *level*, *namespace_=''*, *name_='testsuite'*, *namespacedef_=''*)

exportAttributes(*outfile*, *level*, *already_processed*, *namespace_=''*, *name_='testsuite'*)

exportChildren(*outfile*, *level*, *namespace_=''*, *name_='testsuite'*, *fromsubclass_=False*)

exportLiteral(*outfile*, *level*, *name_='testsuite'*)

exportLiteralAttributes(*outfile*, *level*, *already_processed*, *name_*)

exportLiteralChildren(*outfile*, *level*, *name_*)

static factory(**args_*, ***kwargs_*)

get_errors()

get_extensiontype_()

get_failures()

get_hostname()

get_name()

get_properties()

get_system_err()

get_system_out()

get_testcase()

get_tests()

get_time()

get_timestamp()

hasContent_()

insert_testcase(*index*, *value*)

set_errors(*errors*)

set_extensiontype_(*extensiontype_*)

set_failures(*failures*)

set_hostname(*hostname*)

set_name(*name*)

set_properties(*properties*)

set_system_err(*system_err*)

set_system_out(*system_out*)

set_testcase(*testcase*)

set_tests(*tests*)

set_time(*time*)

set_timestamp(*timestamp*)

subclass = None

superclass = None

validate_ISO8601_DATETIME_PATTERN(*value*)

class autotest.client.tools.JUnit_api.**testsuiteType**(*tests=None*, *errors=None*, *name=None*, *timestamp=None*, *hostname=None*, *time=None*, *failures=None*, *properties=None*, *testcase=None*, *system_out=None*, *system_err=None*, *id=None*, *package=None*)

Bases: *autotest.client.tools.JUnit_api.testsuite*

Derived from testsuite/@name in the non-aggregated documentsStarts at '0' for the first testsuite and is incremented by 1 for each following testsuite

build(*node*)

buildAttributes(*node*, *attrs*, *already_processed*)

buildChildren(*child_*, *node*, *nodeName_*, *fromsubclass_=False*)

export(*outfile*, *level*, *namespace_='*', *name_='testsuiteType'*, *namespacedef_='*')

exportAttributes(*outfile*, *level*, *already_processed*, *namespace_='*', *name_='testsuiteType'*)

exportChildren(*outfile*, *level*, *namespace_=''*, *name_='testsuiteType'*, *fromsubclass_=False*)

exportLiteral(*outfile*, *level*, *name_='testsuiteType'*)

exportLiteralAttributes(*outfile*, *level*, *already_processed*, *name_*)

exportLiteralChildren(*outfile*, *level*, *name_*)

static **factory**(**args_*, ***kwargs_*)

get_id()

get_package()

hasContent_()

set_id(*id*)

set_package(*package*)

subclass = None

superclass
alias of *testsuite*

class autotest.client.tools.JUnit_api.**testsuites**(*testsuite=None*)
Bases: autotest.client.tools.JUnit_api.GeneratedsSuper

Contains an aggregation of testsuite results

add_testsuite(*value*)

build(*node*)

buildAttributes(*node*, *attrs*, *already_processed*)

buildChildren(*child_*, *node*, *nodeName_*, *fromsubclass_=False*)

export(*outfile*, *level*, *namespace_=''*, *name_='testsuites'*, *namespacedef_=''*)

exportAttributes(*outfile*, *level*, *already_processed*, *namespace_=''*, *name_='testsuites'*)

exportChildren(*outfile*, *level*, *namespace_=''*, *name_='testsuites'*, *fromsubclass_=False*)

exportLiteral(*outfile*, *level*, *name_='testsuites'*)

exportLiteralAttributes(*outfile*, *level*, *already_processed*, *name_*)

exportLiteralChildren(*outfile*, *level*, *name_*)

static **factory**(**args_*, ***kwargs_*)

get_testsuite()

hasContent_()

insert_testsuite(*index*, *value*)

set_testsuite(*testsuite*)

subclass = None

superclass = None

boottool Module

A boottool clone, but written in python and relying mostly on grubby[1].

[1] - http://git.fedorahosted.org/git/?p=grubby.git

class autotest.client.tools.boottool.**Grubby**(*path=None*, *opts=None*)
Bases: object

Grubby wrapper

This class calls the grubby binary for most commands, but also adds some functionality that is not really suited to be included in int, such as boot-once.

SUPPORTED_BOOTLOADERS = ('lilo', 'grub2', 'grub', 'extlinux', 'yaboot', 'elilo')

add_args(*kernel*, *args*)
Add cmdline arguments for the specified kernel.

Parameters

- **kernel** – can be a position number (index) or title
- **args** – argument to be added to the current list of args

add_kernel(*path*, *title='autoserv'*, *root=None*, *args=None*, *initrd=None*, *default=False*, *position='end'*)
Add a kernel entry to the bootloader (or replace if one exists already with the same title).

Parameters

- **path** – string path to the kernel image file
- **title** – title of this entry in the bootloader config
- **root** – string of the root device
- **args** – string with cmdline args
- **initrd** – string path to the initrd file
- **default** – set to True to make this entry the default one (default False)
- **position** – where to insert the new entry in the bootloader config file (default 'end', other valid input 'start', or # of the title)
- **xen_hypervisor** – xen hypervisor image file (valid only when xen mode is enabled)

arch_probe()
Get the system architecture

This is much simpler version then the original boottool version, that does not attempt to filter the result of the command / system call that returns the archicture.

Returns string with system archicteture, such as x86_64, ppc64, etc

boot_once(*title=None*)
Configures the bootloader to boot an entry only once

This is not implemented by grubby, but directly implemented here, via the 'boot_once_<bootloader>' method.

boot_once_elilo(*entry_index*)
Implements boot once for machines with kernel >= 2.6

This manipulates EFI variables via the interface available at /sys/firmware/efi/vars

boot_once_grub(*entry_index*)
Implements the boot once feature for the grub bootloader

boot_once_grub2(*entry_index*)
Implements the boot once feature for the grub2 bootloader

Caveat: this assumes the default set is of type "saved", and not a numeric value.

boot_once_yaboot(*entry_title*)
Implements the boot once feature for the yaboot bootloader

bootloader_probe()
Get the bootloader name that is detected on this machine

This module performs the same action as client side boottool.py get_type() method, but with a better name IMHO.

Returns name of detected bootloader

default()
Get the default entry index.

This module performs the same action as client side boottool.py get_default() method, but with a better name IMHO.

Returns an integer with the the default entry.

get_architecture()
Get the system architecture

This is much simpler version then the original boottool version, that does not attempt to filter the result of the command / system call that returns the archicture.

Returns string with system archicteture, such as x86_64, ppc64, etc

get_bootloader()
Get the bootloader name that is detected on this machine

This module performs the same action as client side boottool.py get_type() method, but with a better name IMHO.

Returns name of detected bootloader

get_default()
Get the default entry index.

This module performs the same action as client side boottool.py get_default() method, but with a better name IMHO.

Returns an integer with the the default entry.

get_default_index()
Get the default entry index.

This module performs the same action as client side boottool.py get_default() method, but with a better name IMHO.

Returns an integer with the the default entry.

get_default_title()
Get the default entry title.

Conforms to the client side boottool.py API, but rely directly on grubby functionality.

Returns a string of the default entry title.

get_entries()
Get all entries information.

Returns a dictionary of index -> entry where entry is a dictionary of entry information as described for get_entry().

get_entry(*search_info*)
Get a single bootloader entry information.

NOTE: if entry is "fallback" and bootloader is grub use index instead of kernel title ("fallback") as fallback is a special option in grub

Parameters **search_info** – can be 'default', position number or title

Returns a dictionary of key->value where key is the type of entry information (ex. 'title', 'args', 'kernel', etc) and value is the value for that piece of information.

get_grubby_version()
Get the version of grubby that is installed on this machine

Returns tuple with (major, minor) grubby version

get_grubby_version_raw()
Get the version of grubby that is installed on this machine as is

Returns string with raw output from grubby –version

get_info(*entry='ALL'*)
Returns information on a given entry, or all of them if not specified

The information is returned as a set of lines, that match the output of 'grubby –info=<entry>'

Parameters **entry** (*string*) – entry description, usually an index starting from 0

Returns set of lines

get_info_lines(*entry='ALL'*)
Returns information on a given entry, or all of them if not specified

The information is returned as a set of lines, that match the output of 'grubby –info=<entry>'

Parameters **entry** (*string*) – entry description, usually an index starting from 0

Returns set of lines

get_title_for_kernel(*path*)
Returns a title for a particular kernel.

Parameters **path** – path of the kernel image configured in the boot config

Returns if the given kernel path is found it will return a string with the title for the found entry, otherwise returns None

get_titles()
Get the title of all boot entries.

Returns list with titles of boot entries

get_type()
Get the bootloader name that is detected on this machine

This module performs the same action as client side boottool.py get_type() method, but with a better name IMHO.

Returns name of detected bootloader

grubby_build(*topdir*, *tarball*)
Attempts to build grubby from the source tarball

grubby_install(*path=None*)
Attempts to install a recent enough version of grubby

So far tested on:

- Fedora 16 x86_64
- Debian 6 x86_64
- SuSE 12.1 x86_64
- RHEL 4 on ia64 (with updated python 2.4)
- RHEL 5 on ia64
- RHEL 6 on ppc64

grubby_install_backup(*path*)
Backs up the current grubby binary to make room the one we'll build

Parameters **path** (*string*) – path to the binary that should be backed up

grubby_install_fetch_tarball(*topdir*)
Fetches and verifies the grubby source tarball

grubby_install_patch_makefile()
Patch makefile, making CFLAGS more forgivable to older toolchains

remove_args(*kernel*, *args*)
Removes specified cmdline arguments.

Parameters

- **kernel** – can be a position number (index) or title
- **args** – argument to be removed of the current list of args

remove_kernel(*kernel*)
Removes a specific entry from the bootloader configuration.

Parameters **kernel** – entry position or entry title.

FIXME: param kernel should also take 'start' or 'end'.

set_default(*index*)
Sets the given entry number to be the default on every next boot

To set a default only for the next boot, use boot_once() instead.

This module performs the same action as client side boottool.py set_default() method, but with a better name IMHO.

Note: both –set-default=<kernel> and –set-default-index=<index> on grubby returns no error when it doesn't find the kernel or index. So this method will, until grubby gets fixed, always return success.

Parameters **index** – entry index number to set as the default.

set_default_by_index(*index*)
Sets the given entry number to be the default on every next boot

To set a default only for the next boot, use boot_once() instead.

This module performs the same action as client side boottool.py set_default() method, but with a better name IMHO.

Note: both –set-default=<kernel> and –set-default-index=<index> on grubby returns no error when it doesn't find the kernel or index. So this method will, until grubby gets fixed, always return success.

Parameters **index** – entry index number to set as the default.

class autotest.client.tools.boottool.**OptionParser**(***kwargs*)
Bases: optparse.OptionParser

Command line option parser

Aims to maintain compatibility at the command line level with boottool

check_values(*opts*, *args*)
Validate the option the user has supplied

option_parser_usage = '%prog [options]'

opts_get_action(*opts*)
Gets the selected action from the parsed opts

opts_has_action(*opts*)
Checks if (parsed) opts has a first class action

class autotest.client.tools.boottool.**EfiVar**(*name*, *data*, *guid=None*, *attributes=None*)
Bases: object

Helper class to manipulate EFI firmware variables

This class has no notion of the EFI firmware variables interface, that is, where it should read from or write to in order to create or delete EFI variables.

On systems with kernel >= 2.6, that interface is a directory structure under /sys/firmware/efi/vars.

On systems with kernel <= 2.4, that interface is going to be a directory structure under /proc/efi/vars. But be advised: this has not been tested yet on kernels <= 2.4.

ATTR_BOOTSERVICE_ACCESS = 2

ATTR_NON_VOLATILE = 1

ATTR_RUNTIME_ACCESS = 4

DEFAULT_ATTRIBUTES = 7

FMT = '512H16B1L512H1L1I'

GUID_CONTENT = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)

GUID_FMT = '16B'

get_data()
Returns the variable data in a list ready for struct.pack()

get_name()
Returns the variable name in a list ready for struct.pack()

get_packed()
Returns the EFI variable raw data packed by struct.pack()

This data should be written to the appropriate interface to create an EFI variable

class autotest.client.tools.boottool.**EfiToolSys**
Bases: object

Interfaces with /sys/firmware/efi/vars provided by the kernel

This interface is present on kernels >= 2.6 with CONFIG_EFI and CONFIG_EFI_VARS options set.

BASE_PATH = '/sys/firmware/efi/vars'

DEL_VAR = '/sys/firmware/efi/vars/del_var'

NEW_VAR = '/sys/firmware/efi/vars/new_var'

check_basic_structure()
Checks the basic directory structure for the /sys/.../vars interface

create_variable(*name*, *data*, *guid=None*, *attributes=None*)
Creates a new EFI variable

Parameters

- **name** (*string*) – the name of the variable that will be created
- **data** (*string*) – user data that will populate the variable
- **guid** (*tuple*) – content for the guid value that composes the full variable name
- **attributes** – integer
- **attributes** – bitwise AND of the EFI attributes this variable will have set

delete_variable(*name*, *data*, *guid=None*, *attributes=None*)
Delete an existing EFI variable

Parameters

- **name** (*string*) – the name of the variable that will be deleted
- **data** (*string*) – user data that will populate the variable
- **guid** (*tuple*) – content for the guid value that composes the full variable name
- **attributes** – integer
- **attributes** – bitwise AND of the EFI attributes this variable will have set

class autotest.client.tools.boottool.**EliloConf**(*path='/etc/elilo.conf'*)
Bases: object

A simple parser for elilo configuration file

Has simple features to add and remove global options only, as this is all we need. grubby takes care of manipulating the boot entries themselves.

add_global_option(*key*, *val=None*)
Adds a global option to the updated elilo configuration file

Parameters

- **key** (*string*) – option name
- **key** – option value or None for options with no values

Returns None

get_updated_content()
Returns the config file content with options to add and remove applied

keyval_to_line(*keyval*)
Transforms a tuple into a text line suitable for the config file

Parameters **keyval** (*tuple*) – a tuple containing key and value

Returns a text line suitable for the config file

line_to_keyval(*line*)
Transforms a text line from the configuration file into a tuple

Parameters **line** (*string*) – line of text from the configuration file

Returns a tuple with key and value

matches_global_option_to_add(*line*)
Utility method to check if option is to be added

Parameters **line** (*string*) – line of text from the configuration file

Returns True or False

matches_global_option_to_remove(*line*)
Utility method to check if option is to be removed

Parameters **line** (*string*) – line of text from the configuration file

Returns True or False

remove_global_option(*key*, *val=None*)
Removes a global option to the updated elilo configuration file

Parameters

- **key** (*string*) – option name
- **key** – option value or None for options with no values

Returns None

update()
Writes the updated content to the configuration file

autotest.client.tools.boottool.**find_executable**(*executable*, *favorite_path=None*)
Returns whether the system has a given executable

Parameters **executable** (*string*) – the name of a file that can be read and executed

autotest.client.tools.boottool.**parse_entry**(*entry_str*, *separator='='*)
Parse entry as returned by boottool.

Parameters **entry_str** – one entry information as returned by boottool

Returns dictionary of key -> value where key is the string before the first ":" in an entry line and value is the string after it

cd_hash Module

common Module

crash_handler Module

Simple crash handling application for autotest

copyright Red Hat Inc 2009

author Lucas Meneghel Rodrigues <lmr@redhat.com>

autotest.client.tools.crash_handler.**gdb_report**(*path*)
Use GDB to produce a report with information about a given core.

Parameters **path** – Path to core file.

autotest.client.tools.crash_handler.**generate_random_string**(*length*)
Return a random string using alphanumeric characters.

@length: length of the string that will be generated.

autotest.client.tools.crash_handler.**get_info_from_core**(*path*)
Reads a core file and extracts a dictionary with useful core information.

Right now, the only information extracted is the full executable name.

Parameters **path** – Path to core file.

autotest.client.tools.crash_handler.**get_parent_pid**(*pid*)
Returns the parent PID for a given PID, converted to an integer.

Parameters **pid** – Process ID.

autotest.client.tools.crash_handler.**get_results_dir_list**(*pid*, *core_dir_basename*)
Get all valid output directories for the core file and the report. It works by inspecting files created by each test on /tmp and verifying if the PID of the process that crashed is a child or grandchild of the autotest test process. If it can't find any relationship (maybe a daemon that died during a test execution), it will write the core file to the debug dirs of all tests currently being executed. If there are no active autotest tests at a particular moment, it will return a list with ['/tmp'].

Parameters

- **pid** – PID for the process that generated the core
- **core_dir_basename** – Basename for the directory that will hold both the core dump and the crash report.

autotest.client.tools.crash_handler.**write_cores**(*core_data*, *dir_list*)
Write core files to all directories, optionally providing reports.

Parameters

- **core_data** – Contents of the core file.
- **dir_list** – List of directories the cores have to be written.
- **report** – Whether reports are to be generated for those core files.

autotest.client.tools.crash_handler.**write_to_file**(*filename*, *data*, *report=False*)
Write contents to a given file path specified. If not specified, the file will be created.

Parameters

- **file_path** – Path to a given file.
- **data** – File contents.
- **report** – Whether we'll use GDB to get a backtrace report of the file.

process_metrics Module

Program that parses autotest metrics results and prints them to stdout, so that the jenkins measurement-plots plugin can parse them.

Authors: Steve Conklin <sconklin@canonical.com> Brad Figg <brad.figg@canonical.com>

Copyright (C) 2012 Canonical Ltd.

This script is distributed under the terms and conditions of the GNU General Public License, Version 2 or later. See http://www.gnu.org/copyleft/gpl.html for details.

autotest.client.tools.process_metrics.**main**(*path*)

autotest.client.tools.process_metrics.**usage**()

regression Module

results2junit Module

Program that parses the autotest results and generates JUnit test results in XML format.

autotest.client.tools.results2junit.**dbg**(*ostr*)

autotest.client.tools.results2junit.**dump**(*obj*)

autotest.client.tools.results2junit.**file_load**(*file_name*)
Load the indicated file into a string and return the string.

autotest.client.tools.results2junit.**main**(*basedir*, *resfiles*)

autotest.client.tools.results2junit.**parse_results**(*text*)
Parse text containing Autotest results.

Returns A list of result 4-tuples.

autotest.client.tools.results2junit.**text_clean**(*text*)
This always seems like such a hack, however, there are some characters that we can't deal with properly so this function just removes them from the text passed in.

scan_results Module

Program that parses the autotest results and return a nicely printed final test result.

copyright Red Hat 2008-2009

autotest.client.tools.scan_results.**main**(*resfiles*)

autotest.client.tools.scan_results.**parse_results**(*text*)
Parse text containing Autotest results.

Returns A list of result 4-tuples.

autotest.client.tools.scan_results.**print_result**(*result*, *name_width*)
Nicely print a single Autotest result.

Parameters

- **result** – a 4-tuple
- **name_width** – test name maximum width

virt_disk Module

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

a

A

B

C

D

E

F

G

H

I

J

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z