# avocado Documentation

***Release 50.0***

**Avocado Development Team**

**May 29, 2017**

# Contents

Contents:

CHAPTER 1

# About Avocado

Avocado is a set of tools and libraries to help with automated testing.

One can call it a test framework with benefits. Native tests are written in Python and they follow the `unittest` pattern, but any executable can serve as a test.

Avocado is composed of:

- A test runner that lets you execute tests. Those tests can be either written in your language of choice, or be written in Python and use the available libraries. In both cases, you get facilities such as automated log and system information collection.
- Libraries that help you write tests in a concise, yet expressive and powerful way. You can find more information about what libraries are intended for test writers at *Libraries and APIs*.
- `Plugins` that can extend and add new functionality to the Avocado Framework.

Avocado is built on the experience accumulated with Autotest, while improving on its weaknesses and shortcomings.

Avocado tries as much as possible to comply with standard Python testing technology. Tests written using the Avocado API are derived from the unittest class, while other methods suited to functional and performance testing were added. The test runner is designed to help people to run their tests while providing an assortment of system and logging facilities, with no effort, and if you want more features, then you can start using the API features progressively.

CHAPTER 2

# Getting Started

The first step towards using Avocado is, quite obviously, installing it.

## Installing Avocado

### Installing from Packages

#### Fedora

Avocado is available in stock Fedora 24 and later. The main package name is `python-avocado`, and can be installed with:

```
dnf install python-avocado
```

Other available packages (depending on the Avocado version) may include:

- `python-avocado-examples`: contains example tests and other example files
- `python2-avocado-plugins-output-html`: HTML job report plugin
- `python2-avocado-plugins-runner-remote`: execution of jobs on a remote machine
- `python2-avocado-plugins-runner-vm`: execution of jobs on a libvirt based VM
- `python2-avocado-plugins-runner-docker`: execution of jobs on a Docker container

#### Fedora from Avocado's own Repo

The Avocado project also makes the latest release, and the LTS (Long Term Stability) releases available from its own package repository. To use it, first get the package repositories configuration file by running the following command:

```
sudo curl https://repos-avocadoproject.rhcloud.com/static/avocado-fedora.repo -o /etc/
↪yum.repos.d/avocado.repo
```

Now check if you have the `avocado` and `avocado-lts` repositories configured by running:

```
sudo dnf repolist avocado avocado-lts
...
repo id      repo name                           status
avocado      Avocado                             50
avocado-lts  Avocado LTS (Long Term Stability)   disabled
```

Regular users of Avocado will want to use the standard `avocado` repository, which tracks the latest Avocado releases. For more information about the LTS releases, please refer to the Avocado Long Term Stability thread and to your package management docs on how to switch to the `avocado-lts` repo.

Finally, after deciding between regular Avocado releases or LTS, you can install the RPM packages by running the following commands:

```
dnf install python-avocado
```

Additionally, other Avocado packages are available for Fedora:

- `python-avocado-examples`: contains example tests and other example files
- `python2-avocado-plugins-output-html`: HTML job report plugin
- `python2-avocado-plugins-runner-remote`: execution of jobs on a remote machine
- `python2-avocado-plugins-runner-vm`: execution of jobs on a libvirt based VM
- `python2-avocado-plugins-runner-docker`: execution of jobs on a Docker container

### Enterprise Linux

Avocado packages for Enterprise Linux are available from the Avocado project RPM repository. Additionally, some packages from the EPEL repo are necessary, so you need to enable it first. For EL7, running the following command should do it:

```
yum install https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm
```

Then you must use the Avocado project RHEL repo (https://repos-avocadoproject.rhcloud.com/static/avocado-el.repo). Running the following command should give you the basic Avocado installation ready:

```
curl https://repos-avocadoproject.rhcloud.com/static/avocado-el.repo -o /etc/yum.
↪repos.d/avocado.repo
yum install python-avocado
```

Other available packages (depending on the Avocado version) may include:

- `python-avocado-examples`: contains example tests and other example files
- `python2-avocado-plugins-output-html`: HTML job report plugin
- `python2-avocado-plugins-runner-remote`: execution of jobs on a remote machine
- `python2-avocado-plugins-runner-vm`: execution of jobs on a libvirt based VM
- `python2-avocado-plugins-runner-docker`: execution of jobs on a Docker container

The LTS (Long Term Stability) repositories are also available for Enterprise Linux. Please refer to the Avocado Long Term Stability thread and to your package management docs on how to switch to the `avocado-lts` repo.

### OpenSUSE

The OpenSUSE project packages LTS versions of Avocado. You can install packages by running the following commands:

```
sudo zypper install avocado
```

### Debian

DEB package support is available in the source tree (look at the `contrib/packages/debian` directory. No actual packages are provided by the Avocado project or the Debian repos.

## Generic installation from a GIT repository

First make sure you have a basic set of packages installed. The following applies to Fedora based distributions, please adapt to your platform:

```
sudo yum install -y git gcc python-devel python-pip libvirt-devel libyaml-devel␣
↪redhat-rpm-config xz-devel
```

Then to install Avocado from the git repository run:

```
git clone git://github.com/avocado-framework/avocado.git
cd avocado
sudo make requirements
sudo python setup.py install
```

Note that *python* and *pip* should point to the Python interpreter version 2.7.x. If you're having trouble to install, you can try again and use the command line utilities *python2.7* and *pip2.7*.

Please note that some Avocado functionality may be implemented by optional plugins. To install say, the HTML report plugin, run:

```
cd optional_plugins/html
sudo python setup.py install
```

If you intend to hack on Avocado, you may want to look at *Hacking and Using Avocado*.

## Installing from standard Python tools

Avocado can also be installed by the standard Python packaging tools, namely `pip`. On most POSIX systems with Python >= 2.7 and `pip` available, installation can be performed with the following commands:

```
pip install avocado-framework
```

**Note:** As a design decision, only the dependencies for the core Avocado test runner will be installed. You may notice, depending on your system, that some plugins will fail to load, due to those missing dependencies.

If you want to install all the requirements for all plugins, you may attempt to do so by running:

```
pip install -r https://raw.githubusercontent.com/avocado-framework/avocado/master/
↪requirements.txt
```

This way you only get the base avocado-framework without the optional plugins. Additionally the installation requires correctly configured system with the right compilers, header files and libraries available. The more predictable and complete Avocado experience can be achieved with the official RPM packages.

# Using Avocado

You should first experience Avocado by using the test runner, that is, the command line tool that will conveniently run your tests and collect their results.

## Running Tests

To do so, please run `avocado` with the `run` sub-command followed by a test reference, which could be either a path to the file, or a recognizable name:

```
$ avocado run /bin/true
JOB ID     : 381b849a62784228d2fd208d929cc49f310412dc
JOB LOG    : $HOME/avocado/job-results/job-2014-08-12T15.39-381b849a/job.log
 (1/1) /bin/true: PASS (0.01 s)
RESULTS    : PASS 1 | ERROR 0 | FAIL 0 | SKIP 0 | WARN 0 | INTERRUPT 0
JOB TIME   : 0.11 s
JOB HTML   : $HOME/avocado/job-results/job-2014-08-12T15.39-381b849a/html/results.html
```

You probably noticed that we used `/bin/true` as a test, and in accordance with our expectations, it passed! These are known as *simple tests*, but there is also another type of test, which we call *instrumented tests*. See more at *Test Types* or just keep reading.

---

**Note:** Although in most cases running `avocado run $test1 $test3 ...` is fine, it can lead to argument vs. test name clashes. The safest way to execute tests is `avocado run --$argument1 --$argument2 -- $test1 $test2`. Everything after – will be considered positional arguments, therefore test names (in case of `avocado run`)

---

## Listing tests

You have two ways of discovering the tests. You can simulate the execution by using the `--dry-run` argument:

```
avocado run /bin/true --dry-run
JOB ID     : 0000000000000000000000000000000000000000
JOB LOG    : /tmp/avocado-dry-runSeWniM/job-2015-10-16T15.46-0000000/job.log
 (1/1) /bin/true: SKIP
RESULTS    : PASS 0 | ERROR 0 | FAIL 0 | SKIP 1 | WARN 0 | INTERRUPT 0
JOB TIME   : 0.10 s
JOB HTML   : /tmp/avocado-dry-runSeWniM/job-2015-10-16T15.46-0000000/html/results.html
```

which supports all `run` arguments, simulates the run and even lists the test params.

The other way is to use `list` subcommand that lists the discovered tests If no arguments provided, Avocado lists "default" tests per each plugin. The output might look like this:

```
$ avocado list
INSTRUMENTED /usr/share/avocado/tests/abort.py
INSTRUMENTED /usr/share/avocado/tests/datadir.py
```

```
INSTRUMENTED /usr/share/avocado/tests/doublefail.py
INSTRUMENTED /usr/share/avocado/tests/doublefree.py
INSTRUMENTED /usr/share/avocado/tests/errortest.py
INSTRUMENTED /usr/share/avocado/tests/failtest.py
INSTRUMENTED /usr/share/avocado/tests/fiotest.py
INSTRUMENTED /usr/share/avocado/tests/gdbtest.py
INSTRUMENTED /usr/share/avocado/tests/gendata.py
INSTRUMENTED /usr/share/avocado/tests/linuxbuild.py
INSTRUMENTED /usr/share/avocado/tests/multiplextest.py
INSTRUMENTED /usr/share/avocado/tests/passtest.py
INSTRUMENTED /usr/share/avocado/tests/sleeptenmin.py
INSTRUMENTED /usr/share/avocado/tests/sleeptest.py
INSTRUMENTED /usr/share/avocado/tests/synctest.py
INSTRUMENTED /usr/share/avocado/tests/timeouttest.py
INSTRUMENTED /usr/share/avocado/tests/trinity.py
INSTRUMENTED /usr/share/avocado/tests/warntest.py
INSTRUMENTED /usr/share/avocado/tests/whiteboard.py
...
```

These Python files are considered by Avocado to contain `INSTRUMENTED` tests.

Let's now list only the executable shell scripts:

```
$ avocado list | grep ^SIMPLE
SIMPLE       /usr/share/avocado/tests/env_variables.sh
SIMPLE       /usr/share/avocado/tests/output_check.sh
SIMPLE       /usr/share/avocado/tests/simplewarning.sh
SIMPLE       /usr/share/avocado/tests/failtest.sh
SIMPLE       /usr/share/avocado/tests/passtest.sh
```

Here, as mentioned before, `SIMPLE` means that those files are executables treated as simple tests. You can also give the `--verbose` or `-V` flag to display files that were found by Avocado, but are not considered Avocado tests:

```
$ avocado list examples/gdb-prerun-scripts/ -V
Type       file
NOT_A_TEST examples/gdb-prerun-scripts/README
NOT_A_TEST examples/gdb-prerun-scripts/pass-sigusr1

SIMPLE: 0
INSTRUMENTED: 0
MISSING: 0
NOT_A_TEST: 2
```

Notice that the verbose flag also adds summary information.

## Writing a Simple Test

This very simple example of simple test written in shell script:

```
$ echo '#!/bin/bash' > /tmp/simple_test.sh
$ echo 'exit 0' >> /tmp/simple_test.sh
$ chmod +x /tmp/simple_test.sh
```

Notice that the file is given executable permissions, which is a requirement for Avocado to treat it as a simple test. Also notice that the script exits with status code 0, which signals a successful result to Avocado.

## Running A More Complex Test Job

You can run any number of test in an arbitrary order, as well as mix and match instrumented and simple tests:

```
$ avocado run failtest.py sleeptest.py synctest.py failtest.py synctest.py /tmp/
↪simple_test.sh
JOB ID     : 86911e49b5f2c36caeea41307cee4fecdcdfa121
JOB LOG    : $HOME/avocado/job-results/job-2014-08-12T15.42-86911e49/job.log
 (1/6) failtest.py:FailTest.test: FAIL (0.00 s)
 (2/6) sleeptest.py:SleepTest.test: PASS (1.00 s)
 (3/6) synctest.py:SyncTest.test: PASS (2.43 s)
 (4/6) failtest.py:FailTest.test: FAIL (0.00 s)
 (5/6) synctest.py:SyncTest.test: PASS (2.44 s)
 (6/6) /tmp/simple_test.sh.1: PASS (0.02 s)
RESULTS    : PASS 4 | ERROR 0 | FAIL 2 | SKIP 0 | WARN 0 | INTERRUPT 0
JOB TIME   : 5.98 s
JOB HTML   : $HOME/avocado/job-results/job-2014-08-12T15.42-86911e49/html/results.html
```

## Interrupting The Job On First Failed Test (failfast)

The Avocado `run` command has the option `--failfast on` to exit the job on first failed test:

```
$ avocado run --failfast on /bin/true /bin/false /bin/true /bin/true
JOB ID     : eaf51b8c7d6be966bdf5562c9611b1ec2db3f68a
JOB LOG    : $HOME/avocado/job-results/job-2016-07-19T09.43-eaf51b8/job.log
 (1/4) /bin/true: PASS (0.01 s)
 (2/4) /bin/false: FAIL (0.01 s)
Interrupting job (failfast).
RESULTS    : PASS 1 | ERROR 0 | FAIL 1 | SKIP 2 | WARN 0 | INTERRUPT 0
JOB TIME   : 0.12 s
JOB HTML   : /home/apahim/avocado/job-results/job-2016-07-19T09.43-eaf51b8/html/
↪results.html
```

The `--failfast` option accepts the argument `off`. Since it's disabled by default, the `off` argument only makes sense in replay jobs, when the original job was executed with `--failfast on`.

## Ignoring Missing Test References

When you provide a list of test references, Avocado will try to resolve all of them to tests. If one or more test references can not be resolved to tests, the Job will not be created. Example:

```
$ avocado run passtest.py badtest.py
Unable to resolve reference(s) 'badtest.py' with plugins(s) 'file', 'robot', 'external
↪', try running 'avocado list -V badtest.py' to see the details.
```

But if you want to execute the Job anyway, with the tests that could be resolved, you can use `--ignore-missing-references on`. The same message will appear in the UI, but the Job will be executed:

```
$ avocado run passtest.py badtest.py --ignore-missing-references on
Unable to resolve reference(s) 'badtest.py' with plugins(s) 'file', 'robot', 'external
↪', try running 'avocado list -V badtest.py' to see the details.
JOB ID     : 85927c113074b9defd64ea595d6d1c3fdfc1f58f
JOB LOG    : $HOME/avocado/job-results/job-2017-05-17T10.54-85927c1/job.log
```

```
 (1/1) passtest.py:PassTest.test: PASS (0.02 s)
RESULTS    : PASS 1 | ERROR 0 | FAIL 0 | SKIP 0 | WARN 0 | INTERRUPT 0 | CANCEL 0
JOB TIME   : 0.11 s
JOB HTML   : $HOME/avocado/job-results/job-2017-05-17T10.54-85927c1/html/results.html
```

The `--ignore-missing-references` option accepts the argument `off`. Since it's disabled by default, the `off` argument only makes sense in replay jobs, when the original job was executed with `--ignore-missing-references on`.

## Running Tests With An External Runner

It's quite common to have organically grown test suites in most software projects. These usually include a custom built, very specific test runner that knows how to find and run their own tests.

Still, running those tests inside Avocado may be a good idea for various reasons, including being able to have results in different human and machine readable formats, collecting system information alongside those tests (the Avocado's *sysinfo* functionality), and more.

Avocado makes that possible by means of its "external runner" feature. The most basic way of using it is:

```
$ avocado run --external-runner=/path/to/external_runner foo bar baz
```

In this example, Avocado will report individual test results for tests *foo*, *bar* and *baz*. The actual results will be based on the return code of individual executions of */path/to/external_runner foo*, */path/to/external_runner bar* and finally */path/to/external_runner baz*.

As another way to explain an show how this feature works, think of the "external runner" as some kind of interpreter and the individual tests as anything that this interpreter recognizes and is able to execute. A UNIX shell, say */bin/sh* could be considered an external runner, and files with shell code could be considered tests:

```
$ echo "exit 0" > /tmp/pass
$ echo "exit 1" > /tmp/fail
$ avocado run --external-runner=/bin/sh /tmp/pass /tmp/fail
JOB ID     : 4a2a1d259690cc7b226e33facdde4f628ab30741
JOB LOG    : /home/<user>/avocado/job-results/job-<date>-<shortid>/job.log
(1/2) /tmp/pass: PASS (0.01 s)
(2/2) /tmp/fail: FAIL (0.01 s)
RESULTS    : PASS 1 | ERROR 0 | FAIL 1 | SKIP 0 | WARN 0 | INTERRUPT 0
JOB TIME   : 0.11 s
JOB HTML   : /home/<user>/avocado/job-results/job-<date>-<shortid>/html/results.html
```

This example is pretty obvious, and could be achieved by giving */tmp/pass* and */tmp/fail* shell "shebangs" (*#!/bin/sh*), making them executable (*chmod +x /tmp/pass /tmp/fail)*, and running them as "SIMPLE" tests.

But now consider the following example:

```
$ avocado run --external-runner=/bin/curl http://local-avocado-server:9405/jobs/ \
                                          http://remote-avocado-server:9405/jobs/
JOB ID     : 56016a1ffffaba02492fdbd5662ac0b958f51e11
JOB LOG    : /home/<user>/avocado/job-results/job-<date>-<shortid>/job.log
(1/2) http://local-avocado-server:9405/jobs/: PASS (0.02 s)
(2/2) http://remote-avocado-server:9405/jobs/: FAIL (3.02 s)
RESULTS    : PASS 1 | ERROR 0 | FAIL 1 | SKIP 0 | WARN 0 | INTERRUPT 0
JOB TIME   : 3.14 s
JOB HTML   : /home/<user>/avocado/job-results/job-<date>-<shortid>/html/results.html
```

This effectively makes */bin/curl* an "external test runner", responsible for trying to fetch those URLs, and reporting PASS or FAIL for each of them.

# Debugging tests

## Showing test output

When developing new tests, you frequently want to look straight at the job log, without switching screens or having to "tail" the job log.

In order to do that, you can use `avocado --show test run ...` or `avocado run --show-job-log ...` options:

```
$ avocado --show test run examples/tests/sleeptest.py
...
Job ID: f9ea1742134e5352dec82335af584d1f151d4b85

START 1-sleeptest.py:SleepTest.test

PARAMS (key=timeout, path=*, default=None) => None
PARAMS (key=sleep_length, path=*, default=1) => 1
Sleeping for 1.00 seconds
PASS 1-sleeptest.py:SleepTest.test

Test results available in $HOME/avocado/job-results/job-2015-06-02T10.45-f9ea174
```

As you can see, the UI output is suppressed and only the job log is shown, making this a useful feature for test development and debugging.

## Interrupting tests execution

To interrupt a job execution a user can press `ctrl+c` which after a single press sends SIGTERM to the main test's process and waits for it to finish. If this does not help user can press `ctrl+c` again (after 2s grace period) which destroys the test's process ungracefully and safely finishes the job execution always providing the test results.

To pause the test execution a user can use `ctrl+z` which sends `SIGSTOP` to all processes inherited from the test's PID. We do our best to stop all processes, but the operation is not atomic and some new processes might not be stopped. Another `ctrl+z` sends `SIGCONT` to all processes inherited by the test's PID resuming the execution. Note the test execution time (concerning the test timeout) are still running while the test's process is stopped.

The test can also be interrupted by an Avocado feature. One example would be the *Debugging with GDB* *Debugging with GDB* feature.

For custom interactions it is also possible to use other means like `pdb` or `pydevd` *Avocado development tips* breakpoints. Beware it's not possible to use `STDIN` from tests (unless dark magic is used).

CHAPTER 3

# Writing Avocado Tests

We are going to write an Avocado test in Python and we are going to inherit from *avocado.Test*. This makes this test a so-called instrumented test.

## Basic example

Let's re-create an old time favorite, `sleeptest`[1]. It is so simple, it does nothing besides sleeping for a while:

```
import time

from avocado import Test

class SleepTest(Test):

    def test(self):
        sleep_length = self.params.get('sleep_length', default=1)
        self.log.debug("Sleeping for %.2f seconds", sleep_length)
        time.sleep(sleep_length)
```

This is about the simplest test you can write for Avocado, while still leveraging its API power.

### What is an Avocado Test

As can be seen in the example above, an Avocado test is a method that starts with `test` in a class that inherits from *avocado.Test*.

### Multiple tests and naming conventions

You can have multiple tests in a single class.

[1] sleeptest is a functional test for Avocado. It's "old" because we also have had such a test for Autotest for a long time.

To do so, just give the methods names that start with `test`, say `test_foo`, `test_bar` and so on. We recommend you follow this naming style, as defined in the PEP8 Function Names section.

For the class name, you can pick any name you like, but we also recommend that it follows the CamelCase convention, also known as CapWords, defined in the PEP 8 document under Class Names.

### Convenience Attributes

Note that the test class provides you with a number of convenience attributes:

- A ready to use log mechanism for your test, that can be accessed by means of `self.log`. It lets you log debug, info, error and warning messages.
- A parameter passing system (and fetching system) that can be accessed by means of `self.params`. This is hooked to the Varianter, about which you can find that more information at *Test parameters*.
- And many more (see *`avocado.core.test.Test`*)

To minimize the accidental clashes we define the public ones as properties so if you see something like `AttributeError: can't set attribute` double you are not overriding these.

## Test statuses

Avocado supports the most common exit statuses:

- `PASS` - test passed, there were no untreated exceptions
- `WARN` - a variant of `PASS` that keeps track of noteworthy events that ultimately do not affect the test outcome. An example could be `soft lockup` present in the `dmesg` output. It's not related to the test results and unless there are failures in the test it means the feature probably works as expected, but there were certain condition which might be nice to review. (some result plugins does not support this and report `PASS` instead)
- `SKIP` - the test's pre-requisites were not satisfied and the test's body was not executed (nor its `setUp()` and `tearDown`).
- `CANCEL` - the test was canceled somewhere during the *setUp()*, the test method or the *tearDown()*. The `setUp()` and `tearDown` methods are executed.
- `FAIL` - test did not result in the expected outcome. A failure points at a (possible) bug in the tested subject, and not in the test itself. When the test (and its) execution breaks, an `ERROR` and not a `FAIL` is reported."
- `ERROR` - this points (probably) at a bug in the test itself, and not in the subject being tested.It is usually caused by uncaught exception and such failures needs to be thoroughly explored and should lead to test modification to avoid this failure or to use `self.fail` along with description how the subject under testing failed to perform it's task.
- `INTERRUPTED` - this result can't be set by the test writer, it is only possible when the timeout is reached or when the user hits `CTRL+C` while executing this test.
- other - there are some other internal test statuses, but you should not ever face them.

As you can see the `FAIL` is a neat status, if tests are developed correctly. When writing tests always think about what its `setUp` should be, what the `test body` and is expected to go wrong in the test. To support you Avocado supports several methods:

### Test methods

The simplest way to set the status is to use `self.fail` or `self.error` directly from test. One can also use `self.skip` but only from the `setUp` method.

To remember a warning, one simply writes to `self.log.warning` logger. This won't interrupt the test execution, but it will remember the condition and, if there are no failures, will report the test as `WARN`.

### Turning errors into failures

Errors on Python code are commonly signaled in the form of exceptions being thrown. When Avocado runs a test, any unhandled exception will be seen as a test `ERROR`, and not as a `FAIL`.

Still, it's common to rely on libraries, which usually raise custom (or builtin) exceptions. Those exceptions would normally result in `ERROR` but if you are certain this is an odd behavior of the object under testing, you should catch the exception and explain the failure in `self.fail` method:

```
try:
    process.run("stress_my_feature")
except process.CmdError as details:
    self.fail("The stress comamnd failed: %s" % details)
```

If your test compounds of many executions and you can't get this exception in other case then expected failure, you can simplify the code by using `fail_on` decorator:

```
avocado.fail_on(process.CmdError)
def test(self):
    process.run("first cmd")
    process.run("second cmd")
    process.run("third cmd")
```

Once again, keeping your tests up-to-date and distinguishing between `FAIL` and `ERROR` will save you a lot of time while reviewing the test results.

## Saving test generated (custom) data

Each test instance provides a so called `whiteboard`. It can be accessed through `self.whiteboard`. This whiteboard is simply a string that will be automatically saved to test results after the test finishes (it's not synced during the execution so when the machine or python crashes badly it might not be present and one should use direct io to the `outputdir` for critical data). If you choose to save binary data to the whiteboard, it's your responsibility to encode it first (base64 is the obvious choice).

Building on the previously demonstrated `sleeptest`, suppose that you want to save the sleep length to be used by some other script or data analysis tool:

```
def test(self):
    sleep_length = self.params.get('sleep_length', default=1)
    self.log.debug("Sleeping for %.2f seconds", sleep_length)
    time.sleep(sleep_length)
    self.whiteboard = "%.2f" % sleep_length
```

The whiteboard can and should be exposed by files generated by the available test result plugins. The `results.json` file already includes the whiteboard for each test. Additionally, we'll save a raw copy of the whiteboard contents on a file named `whiteboard`, in the same level as the `results.json` file, for your convenience (maybe you want to use the result of a benchmark directly with your custom made scripts to analyze that particular benchmark result).

If you need to attach several output files, you can also use `self.outputdir`, which points to the `$RESULTS/test-results/$TEST_ID/data` location and is reserved for arbitrary test result data.

## Accessing test parameters

Each test has a set of parameters that can be accessed through `self.params.get($name, $path=None, $default=None)` where:

- name - name of the parameter (key)
- path - where to look for this parameter (when not specified uses mux-path)
- default - what to return when param not found

The path is a bit tricky. Avocado uses tree to represent parameters. In simple scenarios you don't need to worry and you'll find all your values in default path, but eventually you might want to check-out *Test parameters* to understand the details.

Let's say your test receives following params (you'll learn how to execute them in the following section):

```
$ avocado variants -m examples/tests/sleeptenmin.py.data/sleeptenmin.yaml --variants 2
...
Variant 1:    /run/sleeptenmin/builtin, /run/variants/one_cycle
    /run/sleeptenmin/builtin:sleep_method => builtin
    /run/variants/one_cycle:sleep_cycles  => 1
    /run/variants/one_cycle:sleep_length  => 600
...
```

In test you can access those params by:

```
self.params.get("sleep_method")     # returns "builtin"
self.params.get("sleep_cycles", '*', 10)     # returns 1
self.params.get("sleep_length", "/*/variants/*"  # returns 600
```

**Note:** The path is important in complex scenarios where clashes might occur, because when there are multiple values with the same key matching the query avocado raises an exception. As mentioned you can avoid those by using specific paths or by defining custom mux-path which allows specifying resolving hierarchy. More details can be found in *Test parameters*.

## Running multiple variants of tests

In previous section we describe the params handling so let's have a look on how to produce them and execute your tests with different params.

The variants system is pluggable so you might use custom plugins to produce and feed avocado with your params, but let's start with the plugin called "yaml_to_mux", which is shipped with avocado by default. It accepts `yaml` or even `json` files where using ordered dicts to create a tree-like structure and storing the non-dict variables as parameters and using custom tags to mark locations as multiplex domains. Let's use `examples/tests/sleeptenmin.py.data/sleeptenmin.yaml` file as an example:

```
sleeptenmin: !mux
    builtin:
        sleep_method: builtin
```

```
    shell:
        sleep_method: shell
variants: !mux
    one_cycle:
        sleep_cycles: 1
        sleep_length: 600
    six_cycles:
        sleep_cycles: 6
        sleep_length: 100
    one_hundred_cycles:
        sleep_cycles: 100
        sleep_length: 6
    six_hundred_cycles:
        sleep_cycles: 600
        sleep_length: 1
```

Which produces following structure and parameters:

```
$ avocado variants -m examples/tests/sleeptenmin.py.data/sleeptenmin.yaml --summary 2␣
↪--variants 2
Multiplex tree representation:
  run
       sleeptenmin
           builtin
               → sleep_method: builtin
           shell
                → sleep_method: shell
       variants
            one_cycle
                → sleep_length: 600
                → sleep_cycles: 1
            six_cycles
                → sleep_length: 100
                → sleep_cycles: 6
            one_hundred_cycles
                → sleep_length: 6
                → sleep_cycles: 100
            six_hundred_cycles
                 → sleep_length: 1
                 → sleep_cycles: 600

Multiplex variants:

Variant 1:    /run/sleeptenmin/builtin, /run/variants/one_cycle
    /run/sleeptenmin/builtin:sleep_method => builtin
    /run/variants/one_cycle:sleep_cycles  => 1
    /run/variants/one_cycle:sleep_length  => 600

Variant 2:    /run/sleeptenmin/builtin, /run/variants/six_cycles
    /run/sleeptenmin/builtin:sleep_method => builtin
    /run/variants/six_cycles:sleep_cycles => 6
    /run/variants/six_cycles:sleep_length => 100

Variant 3:    /run/sleeptenmin/builtin, /run/variants/one_hundred_cycles
    /run/sleeptenmin/builtin:sleep_method         => builtin
    /run/variants/one_hundred_cycles:sleep_cycles => 100
    /run/variants/one_hundred_cycles:sleep_length => 6
```

```
Variant 4:    /run/sleeptenmin/builtin, /run/variants/six_hundred_cycles
    /run/sleeptenmin/builtin:sleep_method         => builtin
    /run/variants/six_hundred_cycles:sleep_cycles => 600
    /run/variants/six_hundred_cycles:sleep_length => 1

Variant 5:    /run/sleeptenmin/shell, /run/variants/one_cycle
    /run/sleeptenmin/shell:sleep_method  => shell
    /run/variants/one_cycle:sleep_cycles => 1
    /run/variants/one_cycle:sleep_length => 600

Variant 6:    /run/sleeptenmin/shell, /run/variants/six_cycles
    /run/sleeptenmin/shell:sleep_method   => shell
    /run/variants/six_cycles:sleep_cycles => 6
    /run/variants/six_cycles:sleep_length => 100

Variant 7:    /run/sleeptenmin/shell, /run/variants/one_hundred_cycles
    /run/sleeptenmin/shell:sleep_method           => shell
    /run/variants/one_hundred_cycles:sleep_cycles => 100
    /run/variants/one_hundred_cycles:sleep_length => 6

Variant 8:    /run/sleeptenmin/shell, /run/variants/six_hundred_cycles
    /run/sleeptenmin/shell:sleep_method           => shell
    /run/variants/six_hundred_cycles:sleep_cycles => 600
    /run/variants/six_hundred_cycles:sleep_length => 1
```

You can see that it creates all possible variants of each `multiplex domain`, which are defined by `!mux` tag in the yaml file and displayed as single lines in tree view (compare to double lines which are individual nodes with values). In total it'll produce 8 variants of each test:

```
$ avocado run --mux-yaml examples/tests/sleeptenmin.py.data/sleeptenmin.yaml --␣
↪passtest.py
JOB ID     : cc7ef22654c683b73174af6f97bc385da5a0f02f
JOB LOG    : /home/medic/avocado/job-results/job-2017-01-22T11.26-cc7ef22/job.log
 (1/8) passtest.py:PassTest.test;1: PASS (0.01 s)
 (2/8) passtest.py:PassTest.test;2: PASS (0.01 s)
 (3/8) passtest.py:PassTest.test;3: PASS (0.01 s)
 (4/8) passtest.py:PassTest.test;4: PASS (0.01 s)
 (5/8) passtest.py:PassTest.test;5: PASS (0.01 s)
 (6/8) passtest.py:PassTest.test;6: PASS (0.01 s)
 (7/8) passtest.py:PassTest.test;7: PASS (0.01 s)
 (8/8) passtest.py:PassTest.test;8: PASS (0.01 s)
RESULTS    : PASS 8 | ERROR 0 | FAIL 0 | SKIP 0 | WARN 0 | INTERRUPT 0
JOB TIME   : 0.16 s
```

There are other options to influence the params so please check out `avocado run -h` and for details use *Test parameters*.

## Advanced logging capabilities

Avocado provides advanced logging capabilities at test run time. These can be combined with the standard Python library APIs on tests.

One common example is the need to follow specific progress on longer or more complex tests. Let's look at a very simple test example, but one multiple clear stages on a single test:

```
import logging
import time

from avocado import Test

progress_log = logging.getLogger("progress")

class Plant(Test):

    def test_plant_organic(self):
        rows = self.params.get("rows", default=3)

        # Preparing soil
        for row in range(rows):
            progress_log.info("%s: preparing soil on row %s",
                              self.name, row)

        # Letting soil rest
        progress_log.info("%s: letting soil rest before throwing seeds",
                          self.name)
        time.sleep(2)

        # Throwing seeds
        for row in range(rows):
            progress_log.info("%s: throwing seeds on row %s",
                              self.name, row)

        # Let them grow
        progress_log.info("%s: waiting for Avocados to grow",
                          self.name)
        time.sleep(5)

        # Harvest them
        for row in range(rows):
            progress_log.info("%s: harvesting organic avocados on row %s",
                              self.name, row)
```

From this point on, you can ask Avocado to show your logging stream, either exclusively or in addition to other builtin streams:

```
$ avocado --show app,progress run plant.py
```

The outcome should be similar to:

```
JOB ID     : af786f86db530bff26cd6a92c36e99bedcdca95b
JOB LOG    : /home/cleber/avocado/job-results/job-2016-03-18T10.29-af786f8/job.log
 (1/1) plant.py:Plant.test_plant_organic: progress: 1-plant.py:Plant.test_plant_
↪organic: preparing soil on row 0
progress: 1-plant.py:Plant.test_plant_organic: preparing soil on row 1
progress: 1-plant.py:Plant.test_plant_organic: preparing soil on row 2
progress: 1-plant.py:Plant.test_plant_organic: letting soil rest before throwing seeds
-progress: 1-plant.py:Plant.test_plant_organic: throwing seeds on row 0
progress: 1-plant.py:Plant.test_plant_organic: throwing seeds on row 1
progress: 1-plant.py:Plant.test_plant_organic: throwing seeds on row 2
progress: 1-plant.py:Plant.test_plant_organic: waiting for Avocados to grow
\progress: 1-plant.py:Plant.test_plant_organic: harvesting organic avocados on row 0
progress: 1-plant.py:Plant.test_plant_organic: harvesting organic avocados on row 1
progress: 1-plant.py:Plant.test_plant_organic: harvesting organic avocados on row 2
```

```
PASS (7.01 s)
RESULTS    : PASS 1 | ERROR 0 | FAIL 0 | SKIP 0 | WARN 0 | INTERRUPT 0
JOB TIME   : 7.11 s
JOB HTML   : /home/cleber/avocado/job-results/job-2016-03-18T10.29-af786f8/html/
↪results.html
```

The custom `progress` stream is combined with the application output, which may or may not suit your needs or preferences. If you want the `progress` stream to be sent to a separate file, both for clarity and for persistence, you can run Avocado like this:

```
$ avocado run plant.py --store-logging-stream progress
```

The result is that, besides all the other log files commonly generated, there will be another log file named `progress.INFO` at the job results dir. During the test run, one could watch the progress with:

```
$ tail -f ~/avocado/job-results/latest/progress.INFO
10:36:59 INFO | 1-plant.py:Plant.test_plant_organic: preparing soil on row 0
10:36:59 INFO | 1-plant.py:Plant.test_plant_organic: preparing soil on row 1
10:36:59 INFO | 1-plant.py:Plant.test_plant_organic: preparing soil on row 2
10:36:59 INFO | 1-plant.py:Plant.test_plant_organic: letting soil rest before␣
↪throwing seeds
10:37:01 INFO | 1-plant.py:Plant.test_plant_organic: throwing seeds on row 0
10:37:01 INFO | 1-plant.py:Plant.test_plant_organic: throwing seeds on row 1
10:37:01 INFO | 1-plant.py:Plant.test_plant_organic: throwing seeds on row 2
10:37:01 INFO | 1-plant.py:Plant.test_plant_organic: waiting for Avocados to grow
10:37:06 INFO | 1-plant.py:Plant.test_plant_organic: harvesting organic avocados on␣
↪row 0
10:37:06 INFO | 1-plant.py:Plant.test_plant_organic: harvesting organic avocados on␣
↪row 1
10:37:06 INFO | 1-plant.py:Plant.test_plant_organic: harvesting organic avocados on␣
↪row 2
```

The very same `progress` logger, could be used across multiple test methods and across multiple test modules. In the example given, the test name is used to give extra context.

## `unittest.TestCase` heritage

Since an Avocado test inherits from `unittest.TestCase`, you can use all the assertion methods that its parent.

The code example bellow uses `assertEqual`, `assertTrue` and `assertIsInstace`:

```
from avocado import Test

class RandomExamples(Test):
    def test(self):
        self.log.debug("Verifying some random math...")
        four = 2 * 2
        four_ = 2 + 2
        self.assertEqual(four, four_, "something is very wrong here!")

        self.log.debug("Verifying if a variable is set to True...")
        variable = True
        self.assertTrue(variable)

        self.log.debug("Verifying if this test is an instance of test.Test")
        self.assertIsInstance(self, test.Test)
```

### Running tests under other `unittest` runners

nose is another Python testing framework that is also compatible with `unittest`.

Because of that, you can run avocado tests with the `nosetests` application:

```
$ nosetests examples/tests/sleeptest.py
.
----------------------------------------------------------------------
Ran 1 test in 1.004s

OK
```

Conversely, you can also use the standard `unittest.main()` entry point to run an Avocado test. Check out the following code, to be saved as `dummy.py`:

```
from avocado import Test
from unittest import main

class Dummy(Test):
    def test(self):
        self.assertTrue(True)

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

It can be run by:

```
$ python dummy.py
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

But we'd still recommend using `avocado.main` instead which is our main entry point.

## Setup and cleanup methods

If you need to perform setup actions before/after your test, you may do so in the `setUp` and `tearDown` methods, respectively. We'll give examples in the following section.

## Running third party test suites

It is very common in test automation workloads to use test suites developed by third parties. By wrapping the execution code inside an Avocado test module, you gain access to the facilities and API provided by the framework. Let's say you want to pick up a test suite written in C that it is in a tarball, uncompress it, compile the suite code, and then executing the test. Here's an example that does that:

```
#!/usr/bin/env python

import os

from avocado import Test
```

```
from avocado import main
from avocado.utils import archive
from avocado.utils import build
from avocado.utils import process


class SyncTest(Test):

    """
    Execute the synctest test suite.
    """
    def setUp(self):
        """
        Set default params and build the synctest suite.
        """
        sync_tarball = self.params.get('sync_tarball',
                                       default='synctest.tar.bz2')
        self.sync_length = self.params.get('sync_length', default=100)
        self.sync_loop = self.params.get('sync_loop', default=10)
        # Build the synctest suite
        self.cwd = os.getcwd()
        tarball_path = os.path.join(self.datadir, sync_tarball)
        archive.extract(tarball_path, self.srcdir)
        self.srcdir = os.path.join(self.srcdir, 'synctest')
        build.make(self.srcdir)

    def test(self):
        """
        Execute synctest with the appropriate params.
        """
        os.chdir(self.srcdir)
        cmd = ('./synctest %s %s' %
               (self.sync_length, self.sync_loop))
        process.system(cmd)
        os.chdir(self.cwd)


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

Here we have an example of the `setUp` method in action: Here we get the location of the test suite code (tarball) through *avocado.Test.datadir()*, then uncompress the tarball through *avocado.utils.archive.extract()*, an API that will decompress the suite tarball, followed by *avocado.utils.build.make()*, that will build the suite.

In this example, the `test` method just gets into the base directory of the compiled suite and executes the `./synctest` command, with appropriate parameters, using *avocado.utils.process.system()*.

## Fetching asset files

To run third party test suites as mentioned above, or for any other purpose, we offer an asset fetcher as a method of Avocado Test class. The asset method looks for a list of directories in the `cache_dirs` key, inside the `[datadir.paths]` section from the configuration files. Read-only directories are also supported. When the asset file is not present in any of the provided directories, we will try to download the file from the provided locations, copying it to the first writable cache directory. Example:

```
cache_dirs = ['/usr/local/src/', '~/avocado/cache']
```

In the example above, `/usr/local/src/` is a read-only directory. In that case, when we need to fetch the asset from the locations, it will be copied to the `~/avocado/cache` directory.

If you don't provide a `cache_dirs`, we will create a `cache` directory inside the avocado `data_dir` location to put the fetched files in.

- Use case 1: no `cache_dirs` key in config files, only the asset name provided in the full url format:

```
...
    def setUp(self):
        stress = 'http://people.seas.harvard.edu/~apw/stress/stress-1.0.4.tar.gz'
        tarball = self.fetch_asset(stress)
        archive.extract(tarball, self.srcdir)
...
```

  In this case, `fetch_asset()` will download the file from the url provided, copying it to the `$data_dir/cache` directory. `tarball` variable will contains, for example, `/home/user/avocado/data/cache/stress-1.0.4.tar.gz`.

- Use case 2: Read-only cache directory provided. `cache_dirs = ['/mnt/files']`:

```
...
    def setUp(self):
        stress = 'http://people.seas.harvard.edu/~apw/stress/stress-1.0.4.tar.gz'
        tarball = self.fetch_asset(stress)
        archive.extract(tarball, self.srcdir)
...
```

  In this case, we try to find `stress-1.0.4.tar.gz` file in `/mnt/files` directory. If it's not there, since `/mnt/files` is read-only, we will try to download the asset file to the `$data_dir/cache` directory.

- Use case 3: Writable cache directory provided, along with a list of locations. `cache_dirs = ['~/avocado/cache']`:

```
...
    def setUp(self):
        st_name = 'stress-1.0.4.tar.gz'
        st_hash = 'e1533bc704928ba6e26a362452e6db8fd58b1f0b'
        st_loc = ['http://people.seas.harvard.edu/~apw/stress/stress-1.0.4.tar.gz
↪',
                  'ftp://foo.bar/stress-1.0.4.tar.gz']
        tarball = self.fetch_asset(st_name, asset_hash=st_hash,
                                   locations=st_loc)
        archive.extract(tarball, self.srcdir)
...
```

  In this case, we try to download `stress-1.0.4.tar.gz` from the provided locations list (if it's not already in `~/avocado/cache`). The hash was also provided, so we will verify the hash. To do so, we first look for a hashfile named `stress-1.0.4.tar.gz.sha1` in the same directory. If the hashfile is not present we compute the hash and create the hashfile for further usage.

  The resulting `tarball` variable content will be `~/avocado/cache/stress-1.0.4.tar.gz`. An exception will take place if we fail to download or to verify the file.

Detailing the `fetch_asset()` attributes:

- `name`: The name used to name the fetched file. It can also contains a full URL, that will be used as the first location to try (after serching into the cache directories).

- `asset_hash:` (optional) The expected file hash. If missing, we skip the check. If provided, before computing the hash, we look for a hashfile to verify the asset. If the hashfile is nor present, we compute the hash and create the hashfile in the same cache directory for further usage.
- `algorithm:` (optional) Provided hash algorithm format. Defaults to sha1.
- `locations:` (optional) List of locations that will be used to try to fetch the file from. The supported schemes are `http://`, `https://`, `ftp://` and `file://`. You're required to inform the full url to the file, including the file name. The first success will skip the next locations. Notice that for `file://` we just create a symbolic link in the cache directory, pointing to the file original location.
- `expire:` (optional) time period that the cached file will be considered valid. After that period, the file will be dowloaded again. The value can be an integer or a string containing the time and the unit. Example: '10d' (ten days). Valid units are `s` (second), `m` (minute), `h` (hour) and `d` (day).

The expected `return` is the asset file path or an exception.

## Test Output Check and Output Record Mode

In a lot of occasions, you want to go simpler: just check if the output of a given application matches an expected output. In order to help with this common use case, we offer the option `--output-check-record [mode]` to the test runner:

```
--output-check-record OUTPUT_CHECK_RECORD
                      Record output streams of your tests to reference files
                      (valid options: none (do not record output streams),
                      all (record both stdout and stderr), stdout (record
                      only stderr), stderr (record only stderr). Default:
                      none
```

If this option is used, it will store the stdout or stderr of the process (or both, if you specified `all`) being executed to reference files: `stdout.expected` and `stderr.expected`. Those files will be recorded in the test data dir. The data dir is in the same directory as the test source file, named `[source_file_name.data]`. Let's take as an example the test `synctest.py`. In a fresh checkout of Avocado, you can see:

```
examples/tests/synctest.py.data/stderr.expected
examples/tests/synctest.py.data/stdout.expected
```

From those 2 files, only stdout.expected is non empty:

```
$ cat examples/tests/synctest.py.data/stdout.expected
PAR : waiting
PASS : sync interrupted
```

The output files were originally obtained using the test runner and passing the option –output-check-record all to the test runner:

```
$ scripts/avocado run --output-check-record all synctest.py
JOB ID    : bcd05e4fd33e068b159045652da9eb7448802be5
JOB LOG   : $HOME/avocado/job-results/job-2014-09-25T20.20-bcd05e4/job.log
 (1/1) synctest.py:SyncTest.test: PASS (2.20 s)
RESULTS    : PASS 1 | ERROR 0 | FAIL 0 | SKIP 0 | WARN 0 | INTERRUPT 0
JOB TIME   : 2.30 s
```

After the reference files are added, the check process is transparent, in the sense that you do not need to provide special flags to the test runner. Now, every time the test is executed, after it is done running, it will check if the outputs are

exactly right before considering the test as PASSed. If you want to override the default behavior and skip output check entirely, you may provide the flag `--output-check=off` to the test runner.

The *avocado.utils.process* APIs have a parameter `allow_output_check` (defaults to `all`), so that you can select which process outputs will go to the reference files, should you chose to record them. You may choose `all`, for both stdout and stderr, `stdout`, for the stdout only, `stderr`, for only the stderr only, or `none`, to allow neither of them to be recorded and checked.

This process works fine also with simple tests, which are programs or shell scripts that returns 0 (PASSed) or != 0 (FAILed). Let's consider our bogus example:

```
$ cat output_record.sh
#!/bin/bash
echo "Hello, world!"
```

Let's record the output for this one:

```
$ scripts/avocado run output_record.sh --output-check-record all
JOB ID     : 25c4244dda71d0570b7f849319cd71fe1722be8b
JOB LOG    : $HOME/avocado/job-results/job-2014-09-25T20.49-25c4244/job.log
 (1/1) output_record.sh: PASS (0.01 s)
RESULTS    : PASS 1 | ERROR 0 | FAIL 0 | SKIP 0 | WARN 0 | INTERRUPT 0
JOB TIME   : 0.11 s
```

After this is done, you'll notice that a the test data directory appeared in the same level of our shell script, containing 2 files:

```
$ ls output_record.sh.data/
stderr.expected  stdout.expected
```

Let's look what's in each of them:

```
$ cat output_record.sh.data/stdout.expected
Hello, world!
$ cat output_record.sh.data/stderr.expected
$
```

Now, every time this test runs, it'll take into account the expected files that were recorded, no need to do anything else but run the test. Let's see what happens if we change the `stdout.expected` file contents to `Hello, Avocado!`:

```
$ scripts/avocado run output_record.sh
JOB ID     : f0521e524face93019d7cb99c5765aedd933cb2e
JOB LOG    : $HOME/avocado/job-results/job-2014-09-25T20.52-f0521e5/job.log
 (1/1) output_record.sh: FAIL (0.02 s)
RESULTS    : PASS 0 | ERROR 0 | FAIL 1 | SKIP 0 | WARN 0 | INTERRUPT 0
JOB TIME   : 0.12 s
```

Verifying the failure reason:

```
$ cat $HOME/avocado/job-results/job-2014-09-25T20.52-f0521e5/job.log
20:52:38 test       L0163 INFO | START 1-output_record.sh
20:52:38 test       L0164 DEBUG|
20:52:38 test       L0165 DEBUG| Test instance parameters:
20:52:38 test       L0173 DEBUG|
20:52:38 test       L0176 DEBUG| Default parameters:
20:52:38 test       L0180 DEBUG|
20:52:38 test       L0181 DEBUG| Test instance params override defaults whenever␣
↪available
20:52:38 test       L0182 DEBUG|
```

```
20:52:38 process    L0242 INFO | Running '$HOME/Code/avocado/output_record.sh'
20:52:38 process    L0310 DEBUG| [stdout] Hello, world!
20:52:38 test       L0565 INFO | Command: $HOME/Code/avocado/output_record.sh
20:52:38 test       L0565 INFO | Exit status: 0
20:52:38 test       L0565 INFO | Duration: 0.00313782691956
20:52:38 test       L0565 INFO | Stdout:
20:52:38 test       L0565 INFO | Hello, world!
20:52:38 test       L0565 INFO |
20:52:38 test       L0565 INFO | Stderr:
20:52:38 test       L0565 INFO |
20:52:38 test       L0060 ERROR|
20:52:38 test       L0063 ERROR| Traceback (most recent call last):
20:52:38 test       L0063 ERROR|   File "$HOME/Code/avocado/avocado/test.py", line␣
↪397, in check_reference_stdout
20:52:38 test       L0063 ERROR|     self.assertEqual(expected, actual, msg)
20:52:38 test       L0063 ERROR|   File "/usr/lib64/python2.7/unittest/case.py", line␣
↪551, in assertEqual
20:52:38 test       L0063 ERROR|     assertion_func(first, second, msg=msg)
20:52:38 test       L0063 ERROR|   File "/usr/lib64/python2.7/unittest/case.py", line␣
↪544, in _baseAssertEqual
20:52:38 test       L0063 ERROR|     raise self.failureException(msg)
20:52:38 test       L0063 ERROR| AssertionError: Actual test sdtout differs from␣
↪expected one:
20:52:38 test       L0063 ERROR| Actual:
20:52:38 test       L0063 ERROR| Hello, world!
20:52:38 test       L0063 ERROR|
20:52:38 test       L0063 ERROR| Expected:
20:52:38 test       L0063 ERROR| Hello, Avocado!
20:52:38 test       L0063 ERROR|
20:52:38 test       L0064 ERROR|
20:52:38 test       L0529 ERROR| FAIL 1-output_record.sh -> AssertionError: Actual␣
↪test sdtout differs from expected one:
Actual:
Hello, world!

Expected:
Hello, Avocado!

20:52:38 test       L0516 INFO |
```

As expected, the test failed because we changed its expectations.

## Test log, stdout and stderr in native Avocado modules

If needed, you can write directly to the expected stdout and stderr files from the native test scope. It is important to make the distinction between the following entities:

- The test logs
- The test expected stdout
- The test expected stderr

The first one is used for debugging and informational purposes. Additionally writing to *self.log.warning* causes test to be marked as dirty and when everything else goes well the test ends with WARN. This means that the test passed but there were non-related unexpected situations described in warning log.

You may log something into the test logs using the methods in *avocado.Test.log* class attributes. Consider the example:

```
class output_test(Test):

    def test(self):
        self.log.info('This goes to the log and it is only informational')
        self.log.warn('Oh, something unexpected, non-critical happened, '
                      'but we can continue.')
        self.log.error('Describe the error here and don't forget to raise '
                       'an exception yourself. Writing to self.log.error '
                       'won't do that for you.')
        self.log.debug('Everybody look, I had a good lunch today...')
```

If you need to write directly to the test stdout and stderr streams, Avocado makes two preconfigured loggers available for that purpose, named `avocado.test.stdout` and `avocado.test.stderr`. You can use Python's standard logging API to write to them. Example:

```
import logging

class output_test(Test):

    def test(self):
        stdout = logging.getLogger('avocado.test.stdout')
        stdout.info('Informational line that will go to stdout')
        ...
        stderr = logging.getLogger('avocado.test.stderr')
        stderr.info('Informational line that will go to stderr')
```

Avocado will automatically save anything a test generates on STDOUT into a `stdout` file, to be found at the test results directory. The same applies to anything a test generates on STDERR, that is, it will be saved into a `stderr` file at the same location.

Additionally, when using the runner's output recording features, namely the `--output-check-record` argument with values `stdout`, `stderr` or `all`, everything given to those loggers will be saved to the files `stdout.expected` and `stderr.expected` at the test's data directory (which is different from the job/test results directory).

## Setting a Test Timeout

Sometimes your test suite/test might get stuck forever, and this might impact your test grid. You can account for that possibility and set up a `timeout` parameter for your test. The test timeout can be set through the test parameters, as shown below.

```
sleep_length: 5
timeout: 3
```

```
$ avocado run sleeptest.py --mux-yaml /tmp/sleeptest-example.yaml
JOB ID     : c78464bde9072a0b5601157989a99f0ba32a288e
JOB LOG    : $HOME/avocado/job-results/job-2016-11-02T11.13-c78464b/job.log
 (1/1) sleeptest.py:SleepTest.test: INTERRUPTED (3.04 s)
RESULTS    : PASS 0 | ERROR 0 | FAIL 0 | SKIP 0 | WARN 0 | INTERRUPT 1
JOB TIME   : 3.14 s
JOB HTML   : $HOME/avocado/job-results/job-2016-11-02T11.13-c78464b/html/results.html
```

```
$ cat $HOME/avocado/job-results/job-2016-11-02T11.13-c78464b/job.log
2016-11-02 11:13:01,133 job              L0384 INFO | Multiplex tree representation:
2016-11-02 11:13:01,133 job              L0386 INFO |  \-- run
2016-11-02 11:13:01,133 job              L0386 INFO |         -> sleep_length: 5
2016-11-02 11:13:01,133 job              L0386 INFO |         -> timeout: 3
2016-11-02 11:13:01,133 job              L0387 INFO |
2016-11-02 11:13:01,134 job              L0391 INFO | Temporary dir: /var/tmp/avocado_
↪PqDEyC
2016-11-02 11:13:01,134 job              L0392 INFO |
2016-11-02 11:13:01,134 job              L0399 INFO | Variant 1:    /run
2016-11-02 11:13:01,134 job              L0402 INFO |
2016-11-02 11:13:01,134 job              L0311 INFO | Job ID:␣
↪c78464bde9072a0b5601157989a99f0ba32a288e
2016-11-02 11:13:01,134 job              L0314 INFO |
2016-11-02 11:13:01,345 sysinfo          L0107 DEBUG| Not logging /proc/pci (file␣
↪does not exist)
2016-11-02 11:13:01,351 sysinfo          L0105 DEBUG| Not logging /proc/slabinfo␣
↪(lack of permissions)
2016-11-02 11:13:01,355 sysinfo          L0107 DEBUG| Not logging /sys/kernel/debug/
↪sched_features (file does not exist)
2016-11-02 11:13:01,388 sysinfo          L0388 INFO | Commands configured by file: /
↪etc/avocado/sysinfo/commands
2016-11-02 11:13:01,388 sysinfo          L0399 INFO | Files configured by file: /etc/
↪avocado/sysinfo/files
2016-11-02 11:13:01,388 sysinfo          L0419 INFO | Profilers configured by file: /
↪etc/avocado/sysinfo/profilers
2016-11-02 11:13:01,388 sysinfo          L0427 INFO | Profiler disabled
2016-11-02 11:13:01,394 multiplexer      L0166 DEBUG| PARAMS (key=timeout, path=*,␣
↪default=None) => 3
2016-11-02 11:13:01,395 test             L0216 INFO | START 1-sleeptest.py:SleepTest.
↪test
2016-11-02 11:13:01,396 multiplexer      L0166 DEBUG| PARAMS (key=sleep_length,␣
↪path=*, default=1) => 5
2016-11-02 11:13:01,396 sleeptest        L0022 DEBUG| Sleeping for 5.00 seconds
2016-11-02 11:13:04,411 stacktrace       L0038 ERROR|
2016-11-02 11:13:04,412 stacktrace       L0041 ERROR| Reproduced traceback from:
↪$HOME/src/avocado/avocado/core/test.py:454
2016-11-02 11:13:04,412 stacktrace       L0044 ERROR| Traceback (most recent call␣
↪last):
2016-11-02 11:13:04,413 stacktrace       L0044 ERROR|   File "/usr/share/avocado/
↪tests/sleeptest.py", line 23, in test
2016-11-02 11:13:04,413 stacktrace       L0044 ERROR|     time.sleep(sleep_length)
2016-11-02 11:13:04,413 stacktrace       L0044 ERROR|   File "$HOME/src/avocado/
↪avocado/core/runner.py", line 293, in sigterm_handler
2016-11-02 11:13:04,413 stacktrace       L0044 ERROR|     raise SystemExit("Test␣
↪interrupted by SIGTERM")
2016-11-02 11:13:04,414 stacktrace       L0044 ERROR| SystemExit: Test interrupted by␣
↪SIGTERM
2016-11-02 11:13:04,414 stacktrace       L0045 ERROR|
2016-11-02 11:13:04,414 test             L0459 DEBUG| Local variables:
2016-11-02 11:13:04,440 test             L0462 DEBUG|  -> self <class 'sleeptest.
↪SleepTest'>: 1-sleeptest.py:SleepTest.test
2016-11-02 11:13:04,440 test             L0462 DEBUG|  -> sleep_length <type 'int'>: 5
2016-11-02 11:13:04,440 test             L0592 ERROR| ERROR 1-sleeptest.py:SleepTest.
↪test -> TestError: SystemExit('Test interrupted by SIGTERM',): Test interrupted by␣
↪SIGTERM
```

The `yaml` file defines a test parameter `timeout` which overrides the default test timeout before the runner ends the

test forcefully by sending a class:*signal.SIGTERM* to the test, making it raise a `avocado.core.exceptions.TestTimeoutError`.

# Skipping Tests

Avocado offers some options for the test writers to skip a test:

## Test `skip()` Method

**Warning:** *self.skip()* will be deprecated at the end of 2017. Please adjust your tests to use the *self.cancel()* or the skip decorators instead.

Using the `skip()` method available in the Test API is only allowed inside the `setUp()` method. Calling `skip()` from inside the test is not allowed as, by concept, you cannot skip a test after it's already initiated.

The test below:

```
import avocado

class MyTestClass(avocado.Test):

    def setUp(self):
        if self.check_condition():
            self.skip('Test skipped due to the condition.')

    def test(self):
        pass

    def check_condition(self):
        return True
```

Will produce the following result:

```
$ avocado run test_skip_method.py
JOB ID     : 1bd8642400e3b6c584979504cafc4318f7a5fb65
JOB LOG    : $HOME/avocado/job-results/job-2017-02-03T17.16-1bd8642/job.log
 (1/1) test_skip_method.py:MyTestClass.test: SKIP
RESULTS    : PASS 0 | ERROR 0 | FAIL 0 | SKIP 1 | WARN 0 | INTERRUPT 0
JOB TIME   : 0.10 s
JOB HTML   : $HOME/avocado/job-results/job-2017-02-03T17.16-1bd8642/html/results.html
```

Notice that the *tearDown()* will not be executed when *skip()* is used. Any cleanup treatment has to be handled by the *setUp()*, before the call to *skip()*.

## Avocado Skip Decorators

Another way to skip tests is by using the Avocado skip decorators:

- `@avocado.skip(reason)`: Skips a test.
- `@avocado.skipIf(condition, reason)`: Skips a test if the condition is `True`.
- `@avocado.skipUnless(condition, reason)`: Skips a test if the condition is `False`

Those decorators can be used with both `setUp()` method and/or and in the `test*()` methods. The test below:

```
import avocado

class MyTest(avocado.Test):

    @avocado.skipIf(1 == 1, 'Skipping on True condition.')
    def test1(self):
        pass

    @avocado.skip("Don't want this test now.")
    def test2(self):
        pass

    @avocado.skipUnless(1 == 1, 'Skipping on False condition.')
    def test3(self):
        pass
```

Will produce the following result:

```
$ avocado run  test_skip_decorators.py
JOB ID     : 59c815f6a42269daeaf1e5b93e52269fb8a78119
JOB LOG    : $HOME/avocado/job-results/job-2017-02-03T17.41-59c815f/job.log
 (1/3) test_skip_decorators.py:MyTest.test1: SKIP
 (2/3) test_skip_decorators.py:MyTest.test2: SKIP
 (3/3) test_skip_decorators.py:MyTest.test3: PASS (0.02 s)
RESULTS    : PASS 1 | ERROR 0 | FAIL 0 | SKIP 2 | WARN 0 | INTERRUPT 0
JOB TIME   : 0.13 s
JOB HTML   : $HOME/avocado/job-results/job-2017-02-03T17.41-59c815f/html/results.html
```

Notice the `test3` was not skipped because the provided condition was not `False`.

Using the skip decorators, nothing is actually executed. We will skip the *setUp()* method, the test method and the *tearDown()* method.

## Cancelling Tests

You can cancel a test calling *self.cancel()* at any phase of the test (*setUp()*, test method or *tearDown()*). Test will finish with *CANCEL* status and will not make the Job to exit with a non-0 status. Example:

```
#!/usr/bin/env python

from avocado import Test
from avocado import main

from avocado.utils.process import run
from avocado.utils.software_manager import SoftwareManager


class CancelTest(Test):

    """
    Example tests that cancel the current test from inside the test.
    """

    def setUp(self):
        sm = SoftwareManager()
```

```
        self.pkgs = sm.list_all(software_components=False)

    def test_iperf(self):
        if 'iperf-2.0.8-6.fc25.x86_64' not in self.pkgs:
            self.cancel('iperf is not installed or wrong version')
        self.assertIn('pthreads',
                      run('iperf -v', ignore_status=True).stderr)

    def test_gcc(self):
        if 'gcc-6.3.1-1.fc25.x86_64' not in self.pkgs:
            self.cancel('gcc is not installed or wrong version')
        self.assertIn('enable-gnu-indirect-function',
                      run('gcc -v', ignore_status=True).stderr)

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

In a system missing the *iperf* package but with *gcc* installed in the correct version, the result will be:

```
JOB ID     : 39c1f120830b9769b42f5f70b6b7bad0b1b1f09f
JOB LOG    : $HOME/avocado/job-results/job-2017-03-10T16.22-39c1f12/job.log
 (1/2) /home/apahim/avocado/tests/test_cancel.py:CancelTest.test_iperf: CANCEL (1.15␣
↪s)
 (2/2) /home/apahim/avocado/tests/test_cancel.py:CancelTest.test_gcc: PASS (1.13 s)
RESULTS    : PASS 1 | ERROR 0 | FAIL 0 | SKIP 0 | WARN 0 | INTERRUPT 0 | CANCEL 1
JOB TIME   : 2.38 s
JOB HTML   : $HOME/avocado/job-results/job-2017-03-10T16.22-39c1f12/html/results.html
```

Notice that using the *self.cancel()* will cancel the rest of the test from that point on, but the *tearDown()* will still be executed.

Depending on the result format you're referring to, the *CANCEL* status is mapped to a corresponding valid status in that format. See the table below:

| Format | Corresponding Status |
|---|---|
| json | cancel |
| xunit | skipped |
| tap | ok |
| html | CANCEL (warning) |

## Docstring Directives

Some Avocado features, usually only available to instrumented tests, depend on setting directives on the test's class docstring. A docstring directive is composed of a marker (a literal `:avocado:` string), followed by the custom content itself, such as `:avocado: directive`.

This is similar to docstring directives such as `:param my_param: description` and shouldn't be a surprise to most Python developers.

The reason Avocado uses those docstring directives (instead of real Python code) is that the inspection done while looking for tests does not involve any execution of code.

For a detailed explanation about what makes a docstring format valid or not, please refer to our section on *Docstring Directives Rules*.

Now let's follow with some docstring directives examples.

## Explicitly enabling or disabling tests

If your test is a method in a class that directly inherits from *avocado.Test*, then Avocado will find it as one would expect.

Now, the need may arise for more complex tests, to use more advanced Python features such as inheritance. For those tests that are written in a class not directly inherting from *avocado.Test*, Avocado may need your help, because Avocado uses only static analysis to examine the files.

For example, suppose that you define a new test class that inherits from the Avocado base test class, that is, *avocado.Test*, and put it in `mylibrary.py`:

```
from avocado import Test


class MyOwnDerivedTest(Test):
    def __init__(self, methodName='test', name=None, params=None,
                 base_logdir=None, job=None, runner_queue=None):
        super(MyOwnDerivedTest, self).__init__(methodName, name, params,
                                               base_logdir, job,
                                               runner_queue)
        self.log('Derived class example')
```

Then you implement your actual test using that derived class, in `mytest.py`:

```
import mylibrary


class MyTest(mylibrary.MyOwnDerivedTest):

    def test1(self):
        self.log('Testing something important')

    def test2(self):
        self.log('Testing something even more important')
```

If you try to list the tests in that file, this is what you'll get:

```
scripts/avocado list mytest.py -V
Type       Test
NOT_A_TEST mytest.py

ACCESS_DENIED: 0
BROKEN_SYMLINK: 0
EXTERNAL: 0
FILTERED: 0
INSTRUMENTED: 0
MISSING: 0
NOT_A_TEST: 1
SIMPLE: 0
VT: 0
```

You need to give avocado a little help by adding a docstring directive. That docstring directive is `:avocado: enable`. It tells the Avocado safe test detection code to consider it as an avocado test, regardless of what the (admittedly simple) detection code thinks of it. Let's see how that works out. Add the docstring, as you can see the example below:

```
import mylibrary


class MyTest(mylibrary.MyOwnDerivedTest):
    """
    :avocado: enable
    """
    def test1(self):
        self.log('Testing something important')

    def test2(self):
        self.log('Testing something even more important')
```

Now, trying to list the tests on the `mytest.py` file again:

```
scripts/avocado list mytest.py -V
Type         Test
INSTRUMENTED mytest.py:MyTest.test1
INSTRUMENTED mytest.py:MyTest.test2

ACCESS_DENIED: 0
BROKEN_SYMLINK: 0
EXTERNAL: 0
FILTERED: 0
INSTRUMENTED: 2
MISSING: 0
NOT_A_TEST: 0
SIMPLE: 0
VT: 0
```

You can also use the `:avocado: disable` docstring directive, that works the opposite way: something that would be considered an Avocado test, but we force it to not be listed as one.

The docstring `:avocado: disable` is evaluated first by Avocado, meaning that if both `:avocado: disable` and `:avocado: enable` are present in the same docstring, the test will not be listed.

## Categorizing tests

Avocado allows tests to be given tags, which can be used to create test categories. With tags set, users can select a subset of the tests found by the test resolver (also known as test loader).

To make this feature easier to grasp, let's work with an example: a single Python source code file, named `perf.py`, that contains both disk and network performance tests:

```
from avocado import Test

class Disk(Test):

    """
    Disk performance tests

    :avocado: tags=disk,slow,superuser,unsafe
    """

    def test_device(self):
        device = self.params.get('device', default='/dev/vdb')
        self.whiteboard = measure_write_to_disk(device)
```

```
class Network(Test):

    """
    Network performance tests

    :avocado: tags=net,fast,safe
    """

    def test_latency(self):
        self.whiteboard = measure_latency()

    def test_throughput(self):
        self.whiteboard = measure_throughput()


class Idle(Test):

    """
    Idle tests
    """

    def test_idle(self):
        self.whiteboard = "test achieved nothing"
```

**Warning:** All docstring directives in Avocado require a strict format, that is, `:avocado:` followed by one or more spaces, and then followed by a single value **with no white spaces in between**. This means that an attempt to write a docstring directive like `:avocado: tags=foo, bar` will be interpreted as `:avocado: tags=foo,`.

Usually, listing and executing tests with the Avocado test runner would reveal all three tests:

```
$ avocado list perf.py
INSTRUMENTED perf.py:Disk.test_device
INSTRUMENTED perf.py:Network.test_latency
INSTRUMENTED perf.py:Network.test_throughput
INSTRUMENTED perf.py:Idle.test_idle
```

If you want to list or run only the network based tests, you can do so by requesting only tests that are tagged with `net`:

```
$ avocado list perf.py --filter-by-tags=net
INSTRUMENTED perf.py:Network.test_latency
INSTRUMENTED perf.py:Network.test_throughput
```

Now, suppose you're not in an environment where you're confortable running a test that will write to your raw disk devices (such as your development workstation). You know that some tests are tagged with `safe` while others are tagged with `unsafe`. To only select the "safe" tests you can run:

```
$ avocado list perf.py --filter-by-tags=safe
INSTRUMENTED perf.py:Network.test_latency
INSTRUMENTED perf.py:Network.test_throughput
```

But you could also say that you do **not** want the "unsafe" tests (note the *minus* sign before the tag):

```
$ avocado list perf.py --filter-by-tags=-unsafe
INSTRUMENTED perf.py:Network.test_latency
INSTRUMENTED perf.py:Network.test_throughput
```

**Tip:** The - sign may cause issues with some shells. One know error condition is to use spaces between `--filter-by-tags` and the negated tag, that is, `--filter-by-tags -unsafe` will most likely not work. To be on the safe side, use `--filter-by-tags=-tag`.

If you require tests to be tagged with **multiple** tags, just add them separate by commas. Example:

```
$ avocado list perf.py --filter-by-tags=disk,slow,superuser,unsafe
INSTRUMENTED perf.py:Disk.test_device
```

If no test contains all tags given on a single *–filter-by-tags* parameter, no test will be included:

```
$ avocado list perf.py --filter-by-tags=disk,slow,superuser,safe | wc -l
0
```

Test tags can be applied to test classes and to test methods. Tags are evaluated per method, meaning that the class tags will be inherited by all methods, being merged with method local tags. Example:

```
from avocado import Test

class MyClass(Test):
    """
    :avocado: tags=furious
    """

    def test1(self):
        """
        :avocado: tags=fast
        """
        pass

    def test2(self):
        """
        :avocado: tags=slow
        """
        pass
```

If you use the tag `furious`, all tests will be included:

```
$ avocado list furious_tests.py --filter-by-tags=furious
INSTRUMENTED test_tags.py:MyClass.test1
INSTRUMENTED test_tags.py:MyClass.test2
```

But using `fast` and `furious` will include only `test1`:

```
$ avocado list furious_tests.py --filter-by-tags=fast,furious
INSTRUMENTED test_tags.py:MyClass.test1
```

#### Multiple *–filter-by-tags*

While multiple tags in a single option will require tests with all the given tags (effectively a logical AND operation), it's also possible to use multiple `--filter-by-tags` (effectively a logical OR operation).

For instance To include all tests that have the *disk* tag and all tests that have the *net* tag, you can run:

```
$ avocado list perf.py --filter-by-tags=disk --filter-by-tags=net
INSTRUMENTED perf.py:Disk.test_device
INSTRUMENTED perf.py:Network.test_latency
INSTRUMENTED perf.py:Network.test_throughput
```

### Including tests without tags

The normal behavior when using *–filter-by-tags* is to require the given tags on all tests. In some situations, though, it may be desirable to include tests that have no tags set.

For instance, you may want to include tests of certain types that do not have support for tags (such as SIMPLE tests) or tests that have not (yet) received tags. Consider this command:

```
$ avocado list perf.py /bin/true --filter-by-tags=disk
INSTRUMENTED perf.py:Disk.test_device
```

Since it requires the *disk* tag, only one test was returned. By using the *–filter-by-tags-include-empty* option, you can force the inclusion of tests without tags:

```
$ avocado list perf.py /bin/true --filter-by-tags=disk --filter-by-tags-include-empty
SIMPLE       /bin/true
INSTRUMENTED perf.py:Idle.test_idle
INSTRUMENTED perf.py:Disk.test_device
```

## Python `unittest` Compatibility Limitations And Caveats

When executing tests, Avocado uses different techniques than most other Python unittest runners. This brings some compatibility limitations that Avocado users should be aware.

### Execution Model

One of the main differences is a consequence of the Avocado design decision that tests should be self contained and isolated from other tests. Additionally, the Avocado test runner runs each test in a separate process.

If you have a unittest class with many test methods and run them using most test runners, you'll find that all test methods run under the same process. To check that behavior you could add to your `setUp` method:

```
def setUp(self):
    print("PID: %s", os.getpid())
```

If you run the same test under Avocado, you'll find that each test is run on a separate process.

### Class Level `setUp` and `tearDown`

Because of Avocado's test execution model (each test is run on a separate process), it doesn't make sense to support unittest's `unittest.TestCase.setUpClass()` and `unittest.TestCase.tearDownClass()`. Test classes are freshly instantiated for each test, so it's pointless to run code in those methods, since they're supposed to keep class state between tests.

The `setUp` method is the only place in avocado where you are allowed to call the `skip` method, given that, if a test started to be executed, by definition it can't be skipped anymore. Avocado will do its best to enforce this boundary, so that if you use `skip` outside `setUp`, the test upon execution will be marked with the `ERROR` status, and the error message will instruct you to fix your test's code.

If you require a common setup to a number of tests, the current recommended approach is to to write regular `setUp` and `tearDown` code that checks if a given state was already set. One example for such a test that requires a binary installed by a package:

```
from avocado import Test

from avocado.utils import software_manager
from avocado.utils import path as utils_path
from avocado.utils import process


class BinSleep(Test):

    """
    Sleeps using the /bin/sleep binary
    """
    def setUp(self):
        self.sleep = None
        try:
            self.sleep = utils_path.find_command('sleep')
        except utils_path.CmdNotFoundError:
            software_manager.install_distro_packages({'fedora': ['coreutils']})
            self.sleep = utils_path.find_command('sleep')

    def test(self):
        process.run("%s 1" % self.sleep)
```

If your test setup is some kind of action that will last accross processes, like the installation of a software package given in the previous example, you're pretty much covered here.

If you need to keep other type of data a class across test executions, you'll have to resort to saving and restoring the data from an outside source (say a "pickle" file). Finding and using a reliable and safe location for saving such data is currently not in the Avocado supported use cases.

## Environment Variables for Simple Tests

Avocado exports Avocado variables and test parameters as BASH environment to the running test. Those variables are interesting to simple tests, because they can not make use of Avocado API directly with Python, like the native tests can do and also they can modify the test parameters.

Here are the current variables that Avocado exports to the tests:

| Environemnt Variable | Meaning | Example |
|---|---|---|
| AVOCADO_VERSION | Version of Avocado test runner | 0.12.0 |
| AVOCADO_TEST_BASEDIR | Base directory of Avocado tests | $HOME/Downloads/avocado-source/avocado |
| AVOCADO_TEST_DATADIR | Data directory for the test | $AVOCADO_TEST_BASEDIR/my_test.sh.data |
| AVOCADO_TEST_WORKDIR | Work directory for the test | /var/tmp/avocado_Bjr_rd/my_test.sh |
| AVOCADO_TEST_SRCDIR | Source directory for the test | /var/tmp/avocado_Bjr_rd/my-test.sh/src |
| AVOCADO_TESTS_COMMON_TMPDIR | Temporary directory created by the *teststmpdir* plugin. The directory is persistent throughout the tests in the same Job | /var/tmp/avocado_XhEdo/ |
| AVOCADO_TEST_LOGDIR | Log directory for the test | $HOME/logs/job-results/job-2014-09-16T14.38-ac332e6/test-results/$HOME/my_test.sh.1 |
| AVOCADO_TEST_LOGFILE | Log file for the test | $HOME/logs/job-results/job-2014-09-16T14.38-ac332e6/test-results/$HOME/my_test.sh.1/debug.log |
| AVOCADO_TEST_OUTPUTDIR | Output directory for the test | $HOME/logs/job-results/job-2014-09-16T14.38-ac332e6/test-results/$HOME/my_test.sh.1/data |
| AVOCADO_TEST_SYSINFODIR | The system information directory | $HOME/logs/job-results/job-2014-09-16T14.38-ac332e6/test-results/$HOME/my_test.sh.1/sysinfo |
| *** | All variables from –mux-yaml | TIMEOUT=60; IO_WORKERS=10; VM_BYTES=512M; ... |

## Simple Tests BASH extensions

To enhance simple tests one can use supported set of libraries we created. The only requirement is to use:

```
PATH=$(avocado "exec-path"):$PATH
```

which injects path to Avocado utils into shell PATH. Take a look into `avocado exec-path` to see list of available functions and take a look at `examples/tests/simplewarning.sh` for inspiration.

## Wrap Up

We recommend you take a look at the example tests present in the `examples/tests` directory, that contains a few samples to take some inspiration from. That directory, besides containing examples, is also used by the Avocado self test suite to do functional testing of Avocado itself.

It is also recommended that you take a look at the *API Reference*. for more possibilities.

CHAPTER 4

# Result Formats

A test runner must provide an assortment of ways to clearly communicate results to interested parties, be them humans or machines.

**Note:** There are several optional result plugins, you can find them in *Result plugins*.

## Results for human beings

Avocado has two different result formats that are intended for human beings:

- Its default UI, which shows the live test execution results on a command line, text based, UI.
- The HTML report, which is generated after the test job finishes running.

### Avocado command line UI

A regular run of Avocado will present the test results in a live fashion, that is, the job and its test(s) results are constantly updated:

```
$ avocado run sleeptest.py failtest.py synctest.py
JOB ID    : 5ffe479262ea9025f2e4e84c4e92055b5c79bdc9
JOB LOG   : $HOME/avocado/job-results/job-2014-08-12T15.57-5ffe4792/job.log
 (1/3) sleeptest.py:SleepTest.test: PASS (1.01 s)
 (2/3) failtest.py:FailTest.test: FAIL (0.00 s)
 (3/3) synctest.py:SyncTest.test: PASS (1.98 s)
RESULTS    : PASS 1 | ERROR 1 | FAIL 1 | SKIP 0 | WARN 0 | INTERRUPT 0
JOB TIME   : 3.27 s
JOB HTML   : $HOME/avocado/job-results/job-2014-08-12T15.57-5ffe4792/html/results.html
```

The most important thing is to remember that programs should never need to parse human output to figure out what happened to a test job run.

# Machine readable results

Another type of results are those intended to be parsed by other applications. Several standards exist in the test community, and Avocado can in theory support pretty much every result standard out there.

Out of the box, Avocado supports a couple of machine readable results. They are always generated and stored in the results directory in *results.$type* files, but you can ask for a different location too.

## xunit

The default machine readable output in Avocado is xunit.

xunit is an XML format that contains test results in a structured form, and are used by other test automation projects, such as jenkins. If you want to make Avocado to generate xunit output in the standard output of the runner, simply use:

```
$ avocado run sleeptest.py failtest.py synctest.py --xunit -
<?xml version="1.0" encoding="UTF-8"?>
<testsuite name="avocado" tests="3" errors="0" failures="1" skipped="0" time="3.
↪5769162178" timestamp="2016-05-04 14:46:52.803365">
        <testcase classname="SleepTest" name="1-sleeptest.py:SleepTest.test" time="1.
↪00204920769"/>
        <testcase classname="FailTest" name="2-failtest.py:FailTest.test" time="0.
↪00120401382446">
                <failure type="TestFail" message="This test is supposed to fail"><!
↪[CDATA[Traceback (most recent call last):
  File "/home/medic/Work/Projekty/avocado/avocado/avocado/core/test.py", line 490, in␣
↪_run_avocado
    raise test_exception
TestFail: This test is supposed to fail
]]></failure>
                <system-out><![CDATA[14:46:53 ERROR|
14:46:53 ERROR| Reproduced traceback from: /home/medic/Work/Projekty/avocado/avocado/
↪avocado/core/test.py:435
14:46:53 ERROR| Traceback (most recent call last):
14:46:53 ERROR|   File "/home/medic/Work/Projekty/avocado/avocado/examples/tests/
↪failtest.py", line 17, in test
14:46:53 ERROR|     self.fail('This test is supposed to fail')
14:46:53 ERROR|   File "/home/medic/Work/Projekty/avocado/avocado/avocado/core/test.py
↪", line 585, in fail
14:46:53 ERROR|     raise exceptions.TestFail(message)
14:46:53 ERROR| TestFail: This test is supposed to fail
14:46:53 ERROR|
14:46:53 ERROR| FAIL 2-failtest.py:FailTest.test -> TestFail: This test is supposed␣
↪to fail
14:46:53 INFO |
]]></system-out>
        </testcase>
        <testcase classname="SyncTest" name="3-synctest.py:SyncTest.test" time="2.
↪57366299629"/>
</testsuite>
```

**Note:** The dash - in the option *–xunit*, it means that the xunit result should go to the standard output.

## json

JSON is a widely used data exchange format. The json Avocado plugin outputs job information, similarly to the xunit output plugin:

```
$ avocado run sleeptest.py failtest.py synctest.py --json -
{
    "cancel": 0,
    "debuglog": "/home/cleber/avocado/job-results/job-2016-08-09T13.53-10715c4/job.log
↪",
    "errors": 0,
    "failures": 1,
    "job_id": "10715c4645d2d2b57889d7a4317fcd01451b600e",
    "pass": 2,
    "skip": 0,
    "tests": [
        {
            "end": 1470761623.176954,
            "fail_reason": "None",
            "logdir": "/home/cleber/avocado/job-results/job-2016-08-09T13.53-10715c4/
↪test-results/1-sleeptest.py:SleepTest.test",
            "logfile": "/home/cleber/avocado/job-results/job-2016-08-09T13.53-10715c4/
↪test-results/1-sleeptest.py:SleepTest.test/debug.log",
            "start": 1470761622.174918,
            "status": "PASS",
            "test": "1-sleeptest.py:SleepTest.test",
            "time": 1.0020360946655273,
            "url": "1-sleeptest.py:SleepTest.test",
            "whiteboard": ""
        },
        {
            "end": 1470761623.193472,
            "fail_reason": "This test is supposed to fail",
            "logdir": "/home/cleber/avocado/job-results/job-2016-08-09T13.53-10715c4/
↪test-results/2-failtest.py:FailTest.test",
            "logfile": "/home/cleber/avocado/job-results/job-2016-08-09T13.53-10715c4/
↪test-results/2-failtest.py:FailTest.test/debug.log",
            "start": 1470761623.192334,
            "status": "FAIL",
            "test": "2-failtest.py:FailTest.test",
            "time": 0.0011379718780517578,
            "url": "2-failtest.py:FailTest.test",
            "whiteboard": ""
        },
        {
            "end": 1470761625.656061,
            "fail_reason": "None",
            "logdir": "/home/cleber/avocado/job-results/job-2016-08-09T13.53-10715c4/
↪test-results/3-synctest.py:SyncTest.test",
            "logfile": "/home/cleber/avocado/job-results/job-2016-08-09T13.53-10715c4/
↪test-results/3-synctest.py:SyncTest.test/debug.log",
            "start": 1470761623.208165,
            "status": "PASS",
            "test": "3-synctest.py:SyncTest.test",
            "time": 2.4478960037231445,
            "url": "3-synctest.py:SyncTest.test",
            "whiteboard": ""
        }
```

```
    ],
    "time": 3.4510700702667236,
    "total": 3
}
```

**Note:** The dash - in the option *–json*, it means that the xunit result should go to the standard output.

Bear in mind that there's no documented standard for the Avocado JSON result format. This means that it will probably grow organically to accommodate newer Avocado features. A reasonable effort will be made to not break backwards compatibility with applications that parse the current form of its JSON result.

### TAP

Provides the basic TAP (Test Anything Protocol) results, currently in v12. Unlike most existing avocado machine readable outputs this one is streamlined (per test results):

```
$ avocado run sleeptest.py --tap -
1..1
# debug.log of sleeptest.py:SleepTest.test:
#   12:04:38 DEBUG| PARAMS (key=sleep_length, path=*, default=1) => 1
#   12:04:38 DEBUG| Sleeping for 1.00 seconds
#   12:04:39 INFO | PASS 1-sleeptest.py:SleepTest.test
#   12:04:39 INFO |
ok 1 sleeptest.py:SleepTest.test
```

### Silent result

This result disables all stdout logging (while keeping the error messages being printed to stderr). One can then use the return code to learn about the result:

```
$ avocado --silent run failtest.py
$ echo $?
1
```

In practice, this would usually be used by scripts that will in turn run Avocado and check its results:

```
#!/bin/bash
...
$ avocado --silent run /path/to/my/test.py
if [ $? == 0 ]; then
   echo "great success!"
elif
   ...
```

more details regarding exit codes in *Exit Codes* section.

## Multiple results at once

You can have multiple results formats at once, as long as only one of them uses the standard output. For example, it is fine to use the xunit result on stdout and the JSON result to output to a file:

```
$ avocado run sleeptest.py synctest.py --xunit - --json /tmp/result.json
<?xml version="1.0" encoding="UTF-8"?>
<testsuite name="avocado" tests="2" errors="0" failures="0" skipped="0" time="3.
↪64848303795" timestamp="2016-05-04 17:26:05.645665">
        <testcase classname="SleepTest" name="1-sleeptest.py:SleepTest.test" time="1.
↪00270605087"/>
        <testcase classname="SyncTest" name="2-synctest.py:SyncTest.test" time="2.
↪64577698708"/>
</testsuite>

$ cat /tmp/result.json
{
     "debuglog": "/home/cleber/avocado/job-results/job-2016-08-09T13.55-1a94ad6/job.
↪log",
     "errors": 0,
     ...
}
```

But you won't be able to do the same without the –json flag passed to the program:

```
$ avocado run sleeptest.py synctest.py --xunit - --json -
Options --json --xunit are trying to use stdout simultaneously
Please set at least one of them to a file to avoid conflicts
```

That's basically the only rule, and a sane one, that you need to follow.

## Exit Codes

Avocado exit code tries to represent different things that can happen during an execution. That means exit codes can be a combination of codes that were ORed together as a single exit code. The final exit code can be de-bundled so users can have a good idea on what happened to the job.

The single individual exit codes are:

- AVOCADO_ALL_OK (0)
- AVOCADO_TESTS_FAIL (1)
- AVOCADO_JOB_FAIL (2)
- AVOCADO_FAIL (4)
- AVOCADO_JOB_INTERRUPTED (8)

If a job finishes with exit code *9*, for example, it means we had at least one test that failed and also we had at some point a job interruption, probably due to the job timeout or a *CTRL+C*.

## Implementing other result formats

If you are looking to implement a new machine or human readable output format, you can refer to *avocado.plugins.xunit* and use it as a starting point.

If your result is something that is produced at once, based on the complete job outcome, you should create a new class that inherits from *avocado.core.plugin_interfaces.Result* and implements the *avocado.core.plugin_interfaces.Result.render()* method.

But, if your result implementation is something that outputs information live before/during/after tests, then the *avocado.core.plugin_interfaces.ResultEvents* interface is to one to look at. It will require you to implement the methods that will perform actions (write to a file/stream) for each of the defined events on a Job and test execution.

You can take a look at *Plugin System* for more information on how to write a plugin that will activate and execute the new result format.

CHAPTER 5

# Configuration

Avocado utilities have a certain default behavior based on educated, reasonable (we hope) guesses about how users like to use their systems. Of course, different people will have different needs and/or dislike our defaults, and that's why a configuration system is in place to help with those cases

The Avocado config file format is based on the (informal) INI file 'specification', that is implemented by Python's `ConfigParser`. The format is simple and straightforward, composed by *sections*, that contain a number of *keys* and *values*. Take for example a basic Avocado config file:

```
[datadir.paths]
base_dir = /var/lib/avocado
test_dir = /usr/share/avocado/tests
data_dir = /var/lib/avocado/data
logs_dir = ~/avocado/job-results
```

The `datadir.paths` section contains a number of keys, all of them related to directories used by the test runner. The `base_dir` is the base directory to other important Avocado directories, such as log, data and test directories. You can also choose to set those other important directories by means of the variables `test_dir`, `data_dir` and `logs_dir`. You can do this by simply editing the config files available.

## Config file parsing order

Avocado starts by parsing what it calls system wide config file, that is shipped to all Avocado users on a system wide directory, `/etc/avocado/avocado.conf`. Then it'll verify if there's a local user config file, that is located usually in `~/.config/avocado/avocado.conf`. The order of the parsing matters, so the system wide file is parsed, then the user config file is parsed last, so that the user can override values at will. There is another directory that will be scanned by extra config files, `/etc/avocado/conf.d`. This directory may contain plugin config files, and extra additional config files that the system administrator/avocado developers might judge necessary to put there.

Please note that for base directories, if you chose a directory that can't be properly used by Avocado (some directories require read access, others, read and write access), Avocado will fall back to some defaults. So if your regular user wants to write logs to `/root/avocado/logs`, Avocado will not use that directory, since it can't write files to that place. A new location, by default `~/avocado/job-results` will be selected instead.

The order of files described in this section is only valid if avocado was installed in the system. For people using avocado from git repos (usually avocado developers), that did not install it in the system, keep in mind that avocado will read the config files present in the git repos, and will ignore the system wide config files. Running `avocado config` will let you know which files are actually being used.

## Plugin config files

Plugins can also be configured by config files. In order to not disturb the main Avocado config file, those plugins, if they wish so, may install additional config files to `/etc/avocado/conf.d/[pluginname].conf`, that will be parsed after the system wide config file. Users can override those values as well at the local config file level. Considering the config for the hypothethical plugin `salad`:

```
[salad.core]
base = ceasar
dressing = ceasar
```

If you want, you may change `dressing` in your config file by simply adding a `[salad.core]` new section in your local config file, and set a different value for `dressing` there.

## Parsing order recap

So the file parsing order is:

- `/etc/avocado/avocado.conf`
- `/etc/avocado/conf.d/*.conf`
- `~/.config/avocado/avocado.conf`

In this order, meaning that what you set on your local config file may override what's defined in the system wide files.

**Note:** Please note that if avocado is running from git repos, those files will be ignored in favor of in tree configuration files. This is something that would normally only affect people developing avocado, and if you are in doubt, `avocado config` will tell you exactly which files are being used in any given situation.

**Note:** When avocado runs inside virtualenv than path for global config files is also changed. For example, *avocado.conf* comes from the virual-env path *venv/etc/avocado/avocado.conf*.

## Order of precedence for values used in tests

Since you can use the config system to alter behavior and values used in tests (think paths to test programs, for example), we established the following order of precedence for variables (from least precedence to most):

- default value (from library or test code)
- global config file
- local (user) config file
- command line switch

- test parameters

So the least important value comes from the library or test code default, going all the way up to the test parameters system.

## Config plugin

A configuration plugin is provided for users that wish to quickly see what's defined in all sections of their Avocado configuration, after all the files are parsed in their correct resolution order. Example:

```
$ avocado config
Config files read (in order):
    /etc/avocado/avocado.conf
    $HOME/.config/avocado/avocado.conf

    Section.Key     Value
    runner.base_dir /var/lib/avocado
    runner.test_dir /usr/share/avocado/tests
    runner.data_dir /var/lib/avocado/data
    runner.logs_dir ~/avocado/job-results
```

The command also shows the order in which your config files were parsed, giving you a better understanding of what's going on. The Section.Key nomenclature was inspired in `git config --list` output.

## Avocado Data Directories

When running tests, we are frequently looking to:

- Locate tests
- Write logs to a given location
- Grab files that will be useful for tests, such as ISO files or VM disk images

Avocado has a module dedicated to find those paths, to avoid cumbersome path manipulation magic that people had to do in previous test frameworks[1].

If you want to list all relevant directories for your test, you can use *avocado config –datadir* command to list those directories. Executing it will give you an output similar to the one seen below:

```
$ avocado config --datadir
Config files read (in order):
    /etc/avocado/avocado.conf
    $HOME/.config/avocado/avocado.conf

Avocado replaces config dirs that can't be accessed
with sensible defaults. Please edit your local config
file to customize values

Avocado Data Directories:
    base  $HOME/avocado
    tests $HOME/Code/avocado/examples/tests
    data  $HOME/avocado/data
    logs  $HOME/avocado/job-results
```

[1] For example, autotest.

Note that, while Avocado will do its best to use the config values you provide in the config file, if it can't write values to the locations provided, it will fall back to (we hope) reasonable defaults, and we notify the user about that in the output of the command.

The relevant API documentation and meaning of each of those data directories is in `avocado.core.data_dir`, so it's highly recommended you take a look.

You may set your preferred data dirs by setting them in the Avocado config files. The only exception for important data dirs here is the Avocado tmp dir, used to place temporary files used by tests. That directory will be in normal circumstances */var/tmp/avocado_XXXXX*, (where *XXXXX* is in actuality a random string) securely created on */var/tmp/*, unless the user has the *$TMPDIR* environment variable set, since that is customary among unix programs.

The next section of the documentation explains how you can see and set config values that modify the behavior for the Avocado utilities and plugins.

CHAPTER 6

# Test discovery

In this section you can learn how tests are being discovered and how to affect this process.

## The order of test loaders

Avocado supports different types of test starting with *SIMPLE* tests, which are simply executable files, then unittest-like tests called *INSTRUMENTED* up to some tests like the *avocado-vt* ones, which uses complex matrix of tests from config files that don't directly map to existing files. Given the number of loaders, the mapping from test names on the command line to executed tests might not always be unique. Additionally some people might always (or for given run) want to execute only tests of a single type.

To adjust this behavior you can either tweak `plugins.loaders` in avocado settings (`/etc/avocado/`), or temporarily using `--loaders` (option of `avocado run`) option.

This option allows you to specify order and some params of the available test loaders. You can specify either loader_name (`file`), loader_name + TEST_TYPE (`file.SIMPLE`) and for some loaders even additional params passed after `:` (`external:/bin/echo -e`. You can also supply `@DEFAULT`, which injects into that position all the remaining unused loaders.

To get help about `--loaders`:

```
$ avocado run --loaders ?
$ avocado run --loaders external:?
```

Example of how `--loaders` affects the produced tests (manually gathered as some of them result in error):

```
$ avocado run passtest.py boot this_does_not_exist /bin/echo
    > INSTRUMENTED passtest.py:PassTest.test
    > VT           io-github-autotest-qemu.boot
    > MISSING      this_does_not_exist
    > SIMPLE       /bin/echo
$ avocado run passtest.py boot this_does_not_exist /bin/echo --loaders @DEFAULT
↪"external:/bin/echo -e"
    > INSTRUMENTED passtest.py:PassTest.test
```

```
    > VT           io-github-autotest-qemu.boot
    > EXTERNAL     this_does_not_exist
    > SIMPLE       /bin/echo
$ avocado run passtest.py boot this_does_not_exist /bin/echo --loaders file.SIMPLE␣
↪file.INSTRUMENTED @DEFAULT external.EXTERNAL:/bin/echo
    > INSTRUMENTED passtest.py:PassTest.test
    > VT           io-github-autotest-qemu.boot
    > EXTERNAL     this_does_not_exist
    > SIMPLE       /bin/echo
```

## Running simple tests with arguments

This used to be supported out of the box by running `avocado run "test arg1 arg2"` but it was quite confusing and removed. It is still possible to achieve that by using shell and one can even combine normal tests and the parametrized ones:

```
$ avocado run --loaders file external:/bin/sh -- existing_file.py "'/bin/echo␣
↪something'" nonexisting-file
```

This will run 3 tests, the first one is a normal test defined by `existing_file.py` (most probably an instrumented test). Then we have `/bin/echo` which is going to be executed via `/bin/sh -c '/bin/echo something'`. The last one would be `nonexisting-file` which would execute `/bin/sh -c nonexisting-file` which most probably fails.

Note that you are responsible for quotating the test-id (see the `"'/bin/echo something'"` example).

## Filtering tests by tags

Avocado allows tests to be given tags, which can be used to create test categories. With tags set, users can select a subset of the tests found by the test resolver (also known as test loader). For more information about the test tags, visit WritingTests.html#categorizing-tests

CHAPTER 7

# Logging system

This section describes the logging system used in avocado and avocado tests.

## Tweaking the UI

Avocado uses python's logging system to produce UI and to store test's output. The system is quite flexible and allows you to tweak the output to your needs either by built-in stream sets, or directly by using the stream name. To tweak them you can use *avocado –show STREAM[:LEVEL][,STREAM[:LEVEL],...]*. Built-in streams with description (followed by list of associated python streams):

**app** The text based UI (avocado.app)

**test** Output of the executed tests (avocado.test, "")

**debug** Additional messages useful to debug avocado (avocado.app.debug)

**remote** Fabric/paramiko debug messages, useful to analyze remote execution (avocado.fabric, paramiko)

**early** Early logging before the logging system is set. It includes the test output and lots of output produced by used libraries. ("", avocado.test)

Additionally you can specify "all" or "none" to enable/disable all of pre-defined streams and you can also supply custom python logging streams and they will be passed to the standard output.

**Warning:** Messages with importance greater or equal WARN in logging stream "avocado.app" are always enabled and they go to the standard error.

## Storing custom logs

When you run a test, you can also store custom logging streams into the results directory by *avocado run –store-logging-stream [STREAM[:LEVEL] [STREAM[:LEVEL] ...]]*, which will produce *$STREAM.$LEVEL* files per each

(unique) entry in the test results directory.

**Note:** You have to specify separated logging streams. You can't use the built-in streams in this function.

**Note:** Currently the custom streams are stored only per job, not per each individual test.

## Paginator

Some subcommands (list, plugins, ...) support "paginator", which, on compatible terminals, basically pipes the colored output to *less* to simplify browsing of the produced output. One can disable it by *–paginator {on|off}*.

CHAPTER 8

# Sysinfo collection

Avocado comes with a `sysinfo` plugin, which automatically gathers some system information per each job or even between tests. This is very useful when later we want to know what caused the test's failure. This system is configurable but we provide a sane set of defaults for you.

In the default Avocado configuration (`/etc/avocado/avocado.conf`) there is a section `sysinfo.collect` where you can enable/disable the sysinfo collection as well as configure the basic environment. In `sysinfo.collectibles` section you can define basic paths of where to look for what commands/tasks should be performed before/during the sysinfo collection. Avocado supports three types of tasks:

1. commands - file with new-line separated list of commands to be executed before and after the job/test (single execution commands). It is possible to set a timeout which is enforced per each executed command in [sysinfo.collect] by setting "commands_timeout" to a positive number.
2. files - file with new-line separated list of files to be copied
3. profilers - file with new-line separated list of commands to be executed before the job/test and killed at the end of the job/test (follow-like commands)

Additionally this plugin tries to follow the system log via `journalctl` if available.

The sysinfo can also be enabled/disabled on the cmdline if needed by `--sysinfo on|off`.

After the job execution you can find the collected information in `$RESULTS/sysinfo` of `$RESULTS/test-results/$TEST/sysinfo`. They are categorized into `pre`, `post` and `profile` folders and the file-names are safely-escaped executed commands or file-names. You can also see the sysinfo in html results when you have html results plugin enabled.

**Warning:** If you are using avocado from sources, you need to manually place the `commands`/`files`/`profilers` into the `/etc/avocado/sysinfo` directories or adjust the paths in `$AVOCADO_SRC/etc/avocado/avocado.conf`.

# CHAPTER 9

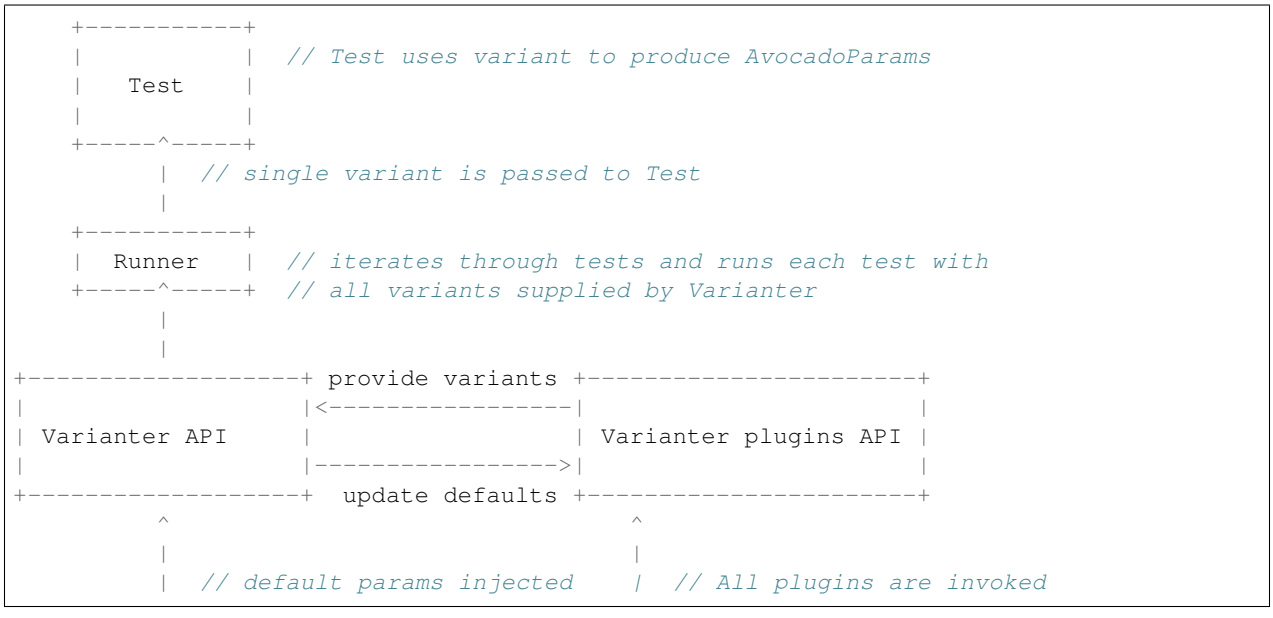
# Test parameters

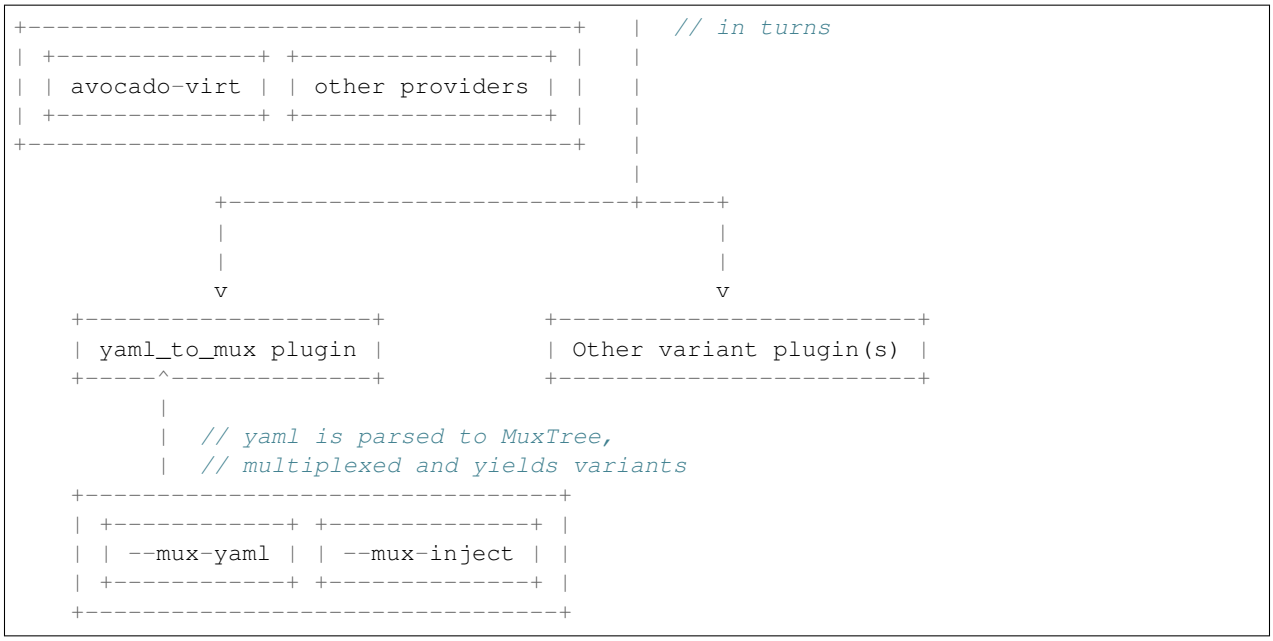
**Note:** This section describes in detail what test parameters are and how the whole variants mechanism works in Avocado. If you're interested in the basics, see *Accessing test parameters* or practical view by examples in *Yaml_to_mux plugin*.

Avocado allows passing parameters to tests, which effectively results in several different variants of each test. These parameters are available in (test's) `self.params` and are of *`avocado.core.varianter.AvocadoParams`* type.

The data for `self.params` are supplied by *`avocado.core.varianter.Varianter`* which asks all registered plugins for variants or uses default when no variants are defined.

Overall picture of how the params handling works is:






Let's introduce the basic keywords.

## Test's default params

*avocado.core.test.Test.default_params*

Every (instrumented) test can hardcode default params by storing a dict in `self.default_params`. This attribute is checked during *avocado.core.test.Test*'s `__init__` phase and if present it's used by *AvocadoParams*.

**Warning:** Don't confuse *Test's default params* with *Default params*

## TreeNode

*avocado.core.tree.TreeNode*

Is a node object allowing to create tree-like structures with parent->multiple_children relations and storing params. It can also report it's environment, which is set of params gathered from root to this node. This is used in tests where instead of passing the full tree only the leaf nodes are passed and their environment represents all the values of the tree.

## AvocadoParams

*avocado.core.varianter.AvocadoParams*

Is a "database" of params present in every (instrumented) avocado test. It's produced during *avocado.core.test.Test*'s `__init__` from a *variant*. It accepts a list of *TreeNode* objects; test name *avocado.core.test.TestName* (for logging purposes); list of default paths (*Mux path*) and the *Test's default params*.

In test it allows querying for data by using:

```
self.params.get($name, $path=None, $default=None)
```

Where:

- name - name of the parameter (key)
- path - where to look for this parameter (when not specified uses mux-path)
- default - what to return when param not found

Each *variant* defines a hierarchy, which is preserved so *AvocadoParams* follows it to return the most appropriate value or raise Exception on error.

## Mux path

As test params are organized in trees, it's possible to have the same variant in several locations. When they are produced from the same *TreeNode*, it's not a problem, but when they are a different values there is no way to distinguish which should be reported. One way is to use specific paths, when asking for params, but sometimes, usually when combining upstream and downstream variants, we want to get our values first and fall-back to the upstream ones when they are not found.

For example let's say we have upstream values in `/upstream/sleeptest` and our values in `/downstream/sleeptest`. If we asked for a value using path `"*"`, it'd raise an exception being unable to distinguish whether we want the value from `/downstream` or `/upstream`. We can set the mux path to `["/downstream/*", "/upstream/*"]` to make all relative calls (path starting with `*`) to first look in nodes in `/downstream` and if not found look into `/upstream`.

More practical overview of mux path is in *yaml_to_mux plugin* in *Resolution order* section.

## Variant

Variant is a set of params produced by *Varianter'_s and passed to the test by the test runner as ''params'* argument. The simplest variant is `None`, which still produces *AvocadoParams* with only the *Test's default params*. If dict is used as a *Variant*, it (safely) updates the default params. Last but not least the *Variant* can also be a `tuple(list, mux_path)` or just the `list` of *avocado.core.tree.TreeNode* with the params.

## Varianter

*avocado.core.varianter.Varianter*

Is an internal object which is used to interact with the variants mechanism in Avocado. It's lifecycle is compound of two stages. First it allows the core/plugins to inject default values, then it is parsed and only allows querying for values, number of variants and such.

Example workflow of *avocado run passtest.py -m example.yaml* is:

```
avocado run passtest.py -m example.yaml
  |
  + parser.finish -> Varianter.__init__  // dispatcher initializes all plugins
  |
  + $PLUGIN -> args.default_avocado_params.add_default_param  // could be used to␣
↪insert default values
  |
```

```
  + job.run_tests -> Varianter.is_parsed
  |
  + job.run_tests -> Varianter.parse
  |                     // processes default params
  |                     // initializes the plugins
  |                     // updates the default values
  |
  + job._log_variants -> Varianter.to_str  // prints the human readable␣
↪representation to log
  |
  + runner.run_suite -> Varianter.get_number_of_tests
  |
  + runner._iter_variants -> Varianter.itertests  // Yields variants
```

In order to allow force-updating the *Varianter* it supports `ignore_new_data`, which can be used to ignore new data. This is used by *Job Replay* to replace the current run *Varianter* with the one loaded from the replayed job. The workflow with `ignore_new_data` could look like this:

```
avocado run --replay latest -m example.yaml
  |
  + $PLUGIN -> args.default_avocado_params.add_default_param  // could be used to␣
↪insert default values
  |
  + replay.run -> Varianter.is_parsed
  |
  + replay.run  // Varianter object is replaced with the replay job's one
  |             // Varianter.ignore_new_data is set
  |
  + $PLUGIN -> args.default_avocado_params.add_default_param  // is ignored as new␣
↪data are not accepted
  |
  + job.run_tests -> Varianter.is_parsed
  |
  + job._log_variants -> Varianter.to_str
  |
  + runner.run_suite -> Varianter.get_number_of_tests
  |
  + runner._iter_variants -> Varianter.itertests
```

The *Varianter* itself can only produce an empty variant with the *Default params*, but it invokes all *Varianter plugins* and if any of them reports variants it yields them instead of the default variant.

## Default params

Unlike *Test's default params* the *Default params* is a mechanism to specify default values in *Varianter* or *Varianter plugins*. Their purpose is usually to define values dependent on the system which should not affect the test's results. One example is a qemu binary location which might differ from one host to another host, but in the end they should result in qemu being executable in test. For this reason the *Default params* do not affects the test's variant-id (at least not in the official *Varianter plugins*).

These params can be set from plugin/core by getting `default_avocado_params` from `args` and using:

```
default_avocado_params.add_default_parma(self, name, key, value, path=None)
```

Where:

- name - name of the plugin which injects data (not yet used for anything, but we plan to allow white/black listing)
- key - the parameter's name
- value - the parameter's value
- path - the location of this parameter. When the path does not exists yet, it's created out of *TreeNode*.

# Varianter plugins

*avocado.core.plugin_interfaces.Varianter*

A plugin interface that can be used to build custom plugins which are used by *Varianter* to get test variants. For inspiration see *avocado.plugins.yaml_to_mux.YamlToMux* which is in-core implementation of a multiplex varianter plugin and which is described in *Yaml_to_mux plugin*.

# Multiplexer

*avocado.core.mux*

`Multiplexer` or simply `Mux` is an abstract concept, which was the basic idea behind the tree-like params structure with the support to produce all possible variants. There is a core implementation of basic building blocks that can be used when creating a custom plugin. There is a demonstration version of plugin using this concept in *avocado.plugins.yaml_to_mux* which adds a parser and then uses this multiplexer concept to define an avocado plugin to produce variants from `yaml` (or `json`) files.

## Multiplexer concept

As mentioned earlier, this is an in-core implementation of building blocks intended for writing *Varianter plugins* based on a tree with *Multiplex domains* defined. The available blocks are:

- *MuxTree* - Object which represents a part of the tree and handles the multiplexation, which means producing all possible variants from a tree-like object.
- *MuxPlugin* - Base class to build *Varianter plugins*
- `MuxTreeNode` - Inherits from *TreeNode* and adds the support for control flags (`MuxTreeNode.ctrl`) and multiplex domains (`MuxTreeNode.multiplex`).

And some support classes and methods eg. for filtering and so on.

# Multiplex domains

A default *AvocadoParams* tree with variables could look like this:

```
Multiplex tree representation:
  paths
      → tmp: /var/tmp
      → qemu: /usr/libexec/qemu-kvm
  environ
     → debug: False
```

The multiplexer wants to produce similar structure, but also to be able to define not just one variant, but to define all possible combinations and then report the slices as variants. We use the term *Multiplex domains* to define that children of this node are not just different paths, but they are different values and we only want one at a time. In the representation we use double-line to visibily distinguish between normal relation and multiplexed relation. Let's modify our example a bit:

```
Multiplex tree representation:
  paths
      → tmp: /var/tmp
      → qemu: /usr/libexec/qemu-kvm
  environ
       production
           → debug: False
       debug
            → debug: True
```

The difference is that `environ` is now a `multiplex` node and it's children will be yielded one at a time producing two variants:

```
Variant 1:
  paths
      → tmp: /var/tmp
      → qemu: /usr/libexec/qemu-kvm
  environ
       production
            → debug: False
Variant 2:
  paths
      → tmp: /var/tmp
      → qemu: /usr/libexec/qemu-kvm
  environ
       debug
            → debug: False
```

Note that the `multiplex` is only about direct children, therefore the number of leaves in variants might differ:

```
Multiplex tree representation:
  paths
      → tmp: /var/tmp
      → qemu: /usr/libexec/qemu-kvm
  environ
       production
           → debug: False
       debug
            system
                → debug: False
            program
                 → debug: True
```

Produces one variant with `/paths` and `/environ/production` and other variant with `/paths`, `/environ/debug/system` and `/environ/debug/program`.

As mentioned earlier the power is not in producing one variant, but in defining huge scenarios with all possible variants. By using tree-structure with multiplex domains you can avoid most of the ugly filters you might know from Jenkin's sparse matrix jobs. For comparison let's have a look at the same example in avocado:

```
Multiplex tree representation:
  os
```

```
       distro
           redhat
                fedora
                    version
                        20
                        21
                    flavor
                         workstation
                         cloud
                rhel
                     5
                     6
       arch
            i386
            x86_64
```

Which produces:

```
Variant 1:    /os/distro/redhat/fedora/version/20, /os/distro/redhat/fedora/flavor/
↪workstation, /os/arch/i386
Variant 2:    /os/distro/redhat/fedora/version/20, /os/distro/redhat/fedora/flavor/
↪workstation, /os/arch/x86_64
Variant 3:    /os/distro/redhat/fedora/version/20, /os/distro/redhat/fedora/flavor/
↪cloud, /os/arch/i386
Variant 4:    /os/distro/redhat/fedora/version/20, /os/distro/redhat/fedora/flavor/
↪cloud, /os/arch/x86_64
Variant 5:    /os/distro/redhat/fedora/version/21, /os/distro/redhat/fedora/flavor/
↪workstation, /os/arch/i386
Variant 6:    /os/distro/redhat/fedora/version/21, /os/distro/redhat/fedora/flavor/
↪workstation, /os/arch/x86_64
Variant 7:    /os/distro/redhat/fedora/version/21, /os/distro/redhat/fedora/flavor/
↪cloud, /os/arch/i386
Variant 8:    /os/distro/redhat/fedora/version/21, /os/distro/redhat/fedora/flavor/
↪cloud, /os/arch/x86_64
Variant 9:    /os/distro/redhat/rhel/5, /os/arch/i386
Variant 10:    /os/distro/redhat/rhel/5, /os/arch/x86_64
Variant 11:    /os/distro/redhat/rhel/6, /os/arch/i386
Variant 12:    /os/distro/redhat/rhel/6, /os/arch/x86_64
```

Versus Jenkin's sparse matrix:

```
os_version = fedora20 fedora21 rhel5 rhel6
os_flavor = none workstation cloud
arch = i386 x86_64

filter = ((os_version == "rhel5").implies(os_flavor == "none") &&
          (os_version == "rhel6").implies(os_flavor == "none")) &&
         !(os_version == "fedora20" && os_flavor == "none") &&
         !(os_version == "fedora21" && os_flavor == "none")
```

Which is still relatively simple example, but it grows dramatically with inner-dependencies.

## MuxPlugin

*avocado.core.mux.MuxPlugin*

Defines the full interface required by *avocado.core.plugin_interfaces.Varianter*. The plugin writer should inherit from this `MuxPlugin`, then from the `Varianter` and call the:

```
self.initialize_mux(root, mux_path, debug)
```

Where:

- root - is the root of your params tree (compound of *TreeNode* -like nodes)
- mux_path - is the *Mux path* to be used in test with all variants
- debug - whether to use debug mode (requires the passed tree to be compound of `TreeNodeDebug`-like nodes which stores the origin of the variant/value/environment as the value for listing purposes and is __NOT__ intended for test execution.

This method must be called before the *Varianter*'s second stage (the latest opportunity is during `self.update_defaults`). The *MuxPlugin*'s code will take care of the rest.

# MuxTree

This is the core feature where the hard work happens. It walks the tree and remembers all leaf nodes or uses list of *MuxTrees* when another multiplex domain is reached while searching for a leaf.

When it's asked to report variants, it combines one variant of each remembered item (leaf node always stays the same, but *MuxTree* circles through it's values) which recursively produces all possible variants of different *multiplex domains*.

## Yaml_to_mux plugin

*avocado.plugins.yaml_to_mux*

So far everything was a bit theoretical, let's use examples to describe how the multiplexation works on a *avocado.plugins.yaml_to_mux* plugin. This plugin inherits from the *avocado.core.mux.MuxPlugin* and the only thing it implements is the argument parsing to get some input and a custom `yaml` parser (which is also capable of parsing `json`).

The `yaml` file is perfect for this task as it's easily read by both, humans and machines. Let's start with an example (line numbers at the first columns are for documentation purposes only, they are not part of the multiplex file format):

```
 1  hw:
 2      cpu: !mux
 3          intel:
 4              cpu_CFLAGS: '-march=core2'
 5          amd:
 6              cpu_CFLAGS: '-march=athlon64'
 7          arm:
 8              cpu_CFLAGS: '-mabi=apcs-gnu -march=armv8-a -mtune=arm8'
 9      disk: !mux
10          scsi:
11              disk_type: 'scsi'
12          virtio:
13              disk_type: 'virtio'
14  distro: !mux
15      fedora:
16          init: 'systemd'
17      mint:
18          init: 'systemv'
```

```
19  env: !mux
20      debug:
21          opt_CFLAGS: '-O0 -g'
22      prod:
23          opt_CFLAGS: '-O2'
```

**Warning:** On some architectures misbehaving versions of CYaml Python library were reported and Avocado always fails with `unacceptable character #x0000: control characters are not allowed`. To workaround this issue you need to either update the PyYaml to the version which works properly, or you need to remove the `python2.7/site-packages/yaml/cyaml.py` or disable CYaml import in Avocado sources. For details check out the Github issue

There are couple of key=>value pairs (lines 4,6,8,11,13,...) and there are named nodes which define scope (lines 1,2,3,5,7,9,...). There are also additional flags (lines 2, 9, 14, 19) which modifies the behavior.

### Nodes

They define context of the key=>value pairs allowing us to easily identify for what this values might be used for and also it makes possible to define multiple values of the same keys with different scope.

Due to their purpose the YAML automatic type conversion for nodes names is disabled, so the value of node name is always as written in the yaml file (unlike values, where *yes* converts to *True* and such).

Nodes are organized in parent-child relationship and together they create a tree. To view this structure use `avocado variants --tree -m <file>`:

```
run
     hw
         cpu
             intel
             amd
             arm
         disk
              scsi
              virtio
     distro
         fedora
         mint
     env
          debug
          prod
```

You can see that `hw` has 2 children `cpu` and `disk`. All parameters defined in parent node are inherited to children and extended/overwritten by their values up to the leaf nodes. The leaf nodes (`intel`, `amd`, `arm`, `scsi`, ...) are the most important as after multiplexation they form the parameters available in tests.

### Keys and Values

Every value other than dict (4,6,8,11) is used as value of the antecedent node.

Each node can define key/value pairs (lines 4,6,8,11,...). Additionally each children node inherits values of it's parent and the result is called node `environment`.

Given the node structure bellow:

```
devtools:
    compiler: 'cc'
    flags:
        - '-O2'
    debug: '-g'
    fedora:
        compiler: 'gcc'
        flags:
            - '-Wall'
    osx:
        compiler: 'clang'
        flags:
            - '-arch i386'
            - '-arch x86_64'
```

And the rules defined as:

- Scalar values (Booleans, Numbers and Strings) are overwritten by walking from the root until the final node.
- Lists are appended (to the tail) whenever we walk from the root to the final node.

The environment created for the nodes `fedora` and `osx` are:

- Node `//devtools/fedora` environment `compiler: 'gcc', flags: ['-O2', '-Wall']`
- Node `//devtools/osx` environment `compiler: 'clang', flags: ['-O2', '-arch i386', '-arch x86_64']`

Note that due to different usage of key and values in environment we disabled the automatic value conversion for keys while keeping it enabled for values. This means that the value can be of any YAML supported value, eg. bool, None, list or custom type, while the key is always string.

### Variants

In the end all leaves are gathered and turned into parameters, more specifically into `AvocadoParams`:

```
setup:
    graphic:
        user: "guest"
        password: "pass"
    text:
        user: "root"
        password: "123456"
```

produces `[graphic, text]`. In the test code you'll be able to query only those leaves. Intermediary or root nodes are available.

The example above generates a single test execution with parameters separated by path. But the most powerful multiplexer feature is that it can generate multiple variants. To do that you need to tag a node whose children are ment to be multiplexed. Effectively it returns only leaves of one child at the time.In order to generate all possible variants multiplexer creates cartesian product of all of these variants:

```
cpu: !mux
    intel:
    amd:
    arm:
fmt: !mux
    qcow2:
    raw:
```

Produces 6 variants:

```
/cpu/intel, /fmt/qcow2
/cpu/intel, /fmt/raw
...
/cpu/arm, /fmt/raw
```

The !mux evaluation is recursive so one variant can expand to multiple ones:

```
fmt: !mux
    qcow: !mux
        2:
        2v3:
    raw:
```

Results in:

```
/fmt/qcow2/2
/fmt/qcow2/2v3
/raw
```

### Resolution order

You can see that only leaves are part of the test parameters. It might happen that some of these leaves contain different values of the same key. Then you need to make sure your queries separate them by different paths. When the path matches multiple results with different origin, an exception is raised as it's impossible to guess which key was originally intended.

To avoid these problems it's recommended to use unique names in test parameters if possible, to avoid the mentioned clashes. It also makes it easier to extend or mix multiple YAML files for a test.

For multiplex YAML files that are part of a framework, contain default configurations, or serve as plugin configurations and other advanced setups it is possible and commonly desirable to use non-unique names. But always keep those points in mind and provide sensible paths.

Multiplexer also supports default paths. By default it's `/run/*` but it can be overridden by `--mux-path`, which accepts multiple arguments. What it does it splits leaves by the provided paths. Each query goes one by one through those sub-trees and first one to hit the match returns the result. It might not solve all problems, but it can help to combine existing YAML files with your ones:

```
qa:         # large and complex read-only file, content injected into /qa
    tests:
        timeout: 10
    ...
my_variants: !mux        # your YAML file injected into /my_variants
    short:
        timeout: 1
    long:
        timeout: 1000
```

You want to use an existing test which uses `params.get('timeout', '*')`. Then you can use `--mux-path '/my_variants/*' '/qa/*'` and it'll first look in your variants. If no matches are found, then it would proceed to `/qa/*`

Keep in mind that only slices defined in mux-path are taken into account for relative paths (the ones starting with `*`)

### Injecting files

You can run any test with any YAML file by:

```
avocado run sleeptest.py --mux-yaml file.yaml
```

This puts the content of `file.yaml` into `/run` location, which as mentioned in previous section, is the default `mux-path` path. For most simple cases this is the expected behavior as your files are available in the default path and you can safely use `params.get(key)`.

When you need to put a file into a different location, for example when you have two files and you don't want the content to be merged into a single place becoming effectively a single blob, you can do that by giving a name to your yaml file:

```
avocado run sleeptest.py --mux-yaml duration:duration.yaml
```

The content of `duration.yaml` is injected into `/run/duration`. Still when keys from other files don't clash, you can use `params.get(key)` and retrieve from this location as it's in the default path, only extended by the `duration` intermediary node. Another benefit is you can merge or separate multiple files by using the same or different name, or even a complex (relative) path.

Last but not least, advanced users can inject the file into whatever location they prefer by:

```
avocado run sleeptest.py --mux-yaml /my/variants/duration:duration.yaml
```

Simple `params.get(key)` won't look in this location, which might be the intention of the test writer. There are several ways to access the values:

- absolute location `params.get(key, '/my/variants/duration')`
- absolute location with wildcards `params.get(key, '/my/*)` (or `/*/duration/*...`)
- set the mux-path `avocado run ... --mux-path /my/*` and use relative path

It's recommended to use the simple injection for single YAML files, relative injection for multiple simple YAML files and the last option is for very advanced setups when you either can't modify the YAML files and you need to specify custom resolution order or you are specifying non-test parameters, for example parameters for your plugin, which you need to separate from the test parameters.

### Multiple files

You can provide multiple files. In such scenario final tree is a combination of the provided files where later nodes with the same name override values of the preceding corresponding node. New nodes are appended as new children:

```
file-1.yaml:
    debug:
        CFLAGS: '-O0 -g'
    prod:
        CFLAGS: '-O2'

file-2.yaml:
    prod:
        CFLAGS: '-Os'
    fast:
        CFLAGS: '-Ofast'
```

results in:

```
debug:
    CFLAGS: '-O0 -g'
prod:
    CFLAGS: '-Os'       # overriden
fast:
    CFLAGS: '-Ofast'    # appended
```

It's also possible to include existing file into another a given node in another file. This is done by the *!include : $path* directive:

```
os:
    fedora:
        !include : fedora.yaml
    gentoo:
        !include : gentoo.yaml
```

**Warning:** Due to YAML nature, it's **mandatory** to put space between *!include* and the colon (*:*) that must follow it.

The file location can be either absolute path or relative path to the YAML file where the *!include* is called (even when it's nested).

Whole file is **merged** into the node where it's defined.

#### Advanced YAML tags

There are additional features related to YAML files. Most of them require values separated by `":"`. Again, in all such cases it's mandatory to add a white space (`" "`) between the tag and the `":"`, otherwise `":"` is part of the tag name and the parsing fails.

#### !include

Includes other file and injects it into the node it's specified in:

```
my_other_file:
    !include : other.yaml
```

The content of `/my_other_file` would be parsed from the `other.yaml`. It's the hardcoded equivalent of the `-m $using:$path`.

Relative paths start from the original file's directory.

#### !using

Prepends path to the node it's defined in:

```
!using : /foo
bar:
    !using : baz
```

`bar` is put into `baz` becoming `/baz/bar` and everything is put into `/foo`. So the final path of `bar` is `/foo/baz/bar`.

### !remove_node

Removes node if it existed during the merge. It can be used to extend incompatible YAML files:

```
os:
    fedora:
    windows:
        3.11:
        95:
os:
    !remove_node : windows
    windows:
        win3.11:
        win95:
```

Removes the *windows* node from structure. It's different from *filter-out* as it really removes the node (and all children) from the tree and it can be replaced by you new structure as shown in the example. It removes *windows* with all children and then replaces this structure with slightly modified version.

As *!remove_node* is processed during merge, when you reverse the order, windows is not removed and you end-up with */windows/{win3.11,win95,3.11,95}* nodes.

### !remove_value

It's similar to *!remove_node* only with values.

### !mux

Children of this node will be multiplexed. This means that in first variant it'll return leaves of the first child, in second the leaves of the second child, etc. Example is in section *Variants*

### !filter-only

Defines internal filters. They are inherited by children and evaluated during multiplexation. It allows one to specify the only compatible branch of the tree with the current variant, for example:

```
cpu:
    arm:
        !filter-only : /disk/virtio
disk:
    virtio:
    scsi:
```

will skip the `[arm, scsi]` variant and result only in `[arm, virtio]`

_Note: It's possible to use `!filter-only` multiple times with the same parent and all allowed variants will be included (unless they are filtered-out by `!filter-out`)_

_Note2: The evaluation order is 1. filter-out, 2. filter-only. This means when you booth filter-out and filter-only a branch it won't take part in the multiplexed variants._

### !filter-out

Similarly to *!filter-only* only it skips the specified branches and leaves the remaining ones. (in the same example the use of `!filter-out : /disk/scsi` results in the same behavior). The difference is when a new disk type is introduced, `!filter-only` still allows just the specified variants, while `!filter-out` only removes the specified ones.

As for the speed optimization, currently Avocado is strongly optimized towards fast `!filter-out` so it's highly recommended using them rather than `!filter-only`, which takes significantly longer to process.

### Complete example

Let's take a second look at the first example:

```
 1    hw:
 2        cpu: !mux
 3            intel:
 4                cpu_CFLAGS: '-march=core2'
 5            amd:
 6                cpu_CFLAGS: '-march=athlon64'
 7            arm:
 8                cpu_CFLAGS: '-mabi=apcs-gnu -march=armv8-a -mtune=arm8'
 9        disk: !mux
10            scsi:
11                disk_type: 'scsi'
12            virtio:
13                disk_type: 'virtio'
14    distro: !mux
15        fedora:
16            init: 'systemd'
17        mint:
18            init: 'systemv'
19    env: !mux
20        debug:
21            opt_CFLAGS: '-O0 -g'
22        prod:
23            opt_CFLAGS: '-O2'
```

After filters are applied (simply removes non-matching variants), leaves are gathered and all variants are generated:

```
$ avocado variants -m examples/mux-environment.yaml
Variants generated:
Variant 1:    /hw/cpu/intel, /hw/disk/scsi, /distro/fedora, /env/debug
Variant 2:    /hw/cpu/intel, /hw/disk/scsi, /distro/fedora, /env/prod
Variant 3:    /hw/cpu/intel, /hw/disk/scsi, /distro/mint, /env/debug
Variant 4:    /hw/cpu/intel, /hw/disk/scsi, /distro/mint, /env/prod
Variant 5:    /hw/cpu/intel, /hw/disk/virtio, /distro/fedora, /env/debug
Variant 6:    /hw/cpu/intel, /hw/disk/virtio, /distro/fedora, /env/prod
Variant 7:    /hw/cpu/intel, /hw/disk/virtio, /distro/mint, /env/debug
Variant 8:    /hw/cpu/intel, /hw/disk/virtio, /distro/mint, /env/prod
Variant 9:    /hw/cpu/amd, /hw/disk/scsi, /distro/fedora, /env/debug
Variant 10:    /hw/cpu/amd, /hw/disk/scsi, /distro/fedora, /env/prod
Variant 11:    /hw/cpu/amd, /hw/disk/scsi, /distro/mint, /env/debug
Variant 12:    /hw/cpu/amd, /hw/disk/scsi, /distro/mint, /env/prod
Variant 13:    /hw/cpu/amd, /hw/disk/virtio, /distro/fedora, /env/debug
Variant 14:    /hw/cpu/amd, /hw/disk/virtio, /distro/fedora, /env/prod
Variant 15:    /hw/cpu/amd, /hw/disk/virtio, /distro/mint, /env/debug
```

```
Variant 16:    /hw/cpu/amd, /hw/disk/virtio, /distro/mint, /env/prod
Variant 17:    /hw/cpu/arm, /hw/disk/scsi, /distro/fedora, /env/debug
Variant 18:    /hw/cpu/arm, /hw/disk/scsi, /distro/fedora, /env/prod
Variant 19:    /hw/cpu/arm, /hw/disk/scsi, /distro/mint, /env/debug
Variant 20:    /hw/cpu/arm, /hw/disk/scsi, /distro/mint, /env/prod
Variant 21:    /hw/cpu/arm, /hw/disk/virtio, /distro/fedora, /env/debug
Variant 22:    /hw/cpu/arm, /hw/disk/virtio, /distro/fedora, /env/prod
Variant 23:    /hw/cpu/arm, /hw/disk/virtio, /distro/mint, /env/debug
Variant 24:    /hw/cpu/arm, /hw/disk/virtio, /distro/mint, /env/prod
```

Where the first variant contains:

```
/hw/cpu/intel/  => cpu_CFLAGS: -march=core2
/hw/disk/       => disk_type: scsi
/distro/fedora/ => init: systemd
/env/debug/     => opt_CFLAGS: -O0 -g
```

The second one:

```
/hw/cpu/intel/  => cpu_CFLAGS: -march=core2
/hw/disk/       => disk_type: scsi
/distro/fedora/ => init: systemd
/env/prod/      => opt_CFLAGS: -O2
```

From this example you can see that querying for `/env/debug` works only in the first variant, but returns nothing in the second variant. Keep this in mind and when you use the `!mux` flag always query for the pre-mux path, `/env/*` in this example.

CHAPTER 10

# Job Replay

In order to reproduce a given job using the same data, one can use the `--replay` option for the `run` command, informing the hash id from the original job to be replayed. The hash id can be partial, as long as the provided part corresponds to the initial characters of the original job id and it is also unique enough. Or, instead of the job id, you can use the string `latest` and avocado will replay the latest job executed.

Let's see an example. First, running a simple job with two test references:

```
$ avocado run /bin/true /bin/false
JOB ID     : 825b860b0c2f6ec48953c638432e3e323f8d7cad
JOB LOG    : $HOME/avocado/job-results/job-2016-01-11T16.14-825b860/job.log
 (1/2) /bin/true: PASS (0.01 s)
 (2/2) /bin/false: FAIL (0.01 s)
RESULTS    : PASS 1 | ERROR 0 | FAIL 1 | SKIP 0 | WARN 0 | INTERRUPT 0
JOB TIME   : 0.12 s
JOB HTML   : $HOME/avocado/job-results/job-2016-01-11T16.14-825b860/html/results.html
```

Now we can replay the job by running:

```
$ avocado run --replay 825b86
JOB ID     : 55a0d10132c02b8cc87deb2b480bfd8abbd956c3
SRC JOB ID : 825b860b0c2f6ec48953c638432e3e323f8d7cad
JOB LOG    : $HOME/avocado/job-results/job-2016-01-11T16.18-55a0d10/job.log
 (1/2) /bin/true: PASS (0.01 s)
 (2/2) /bin/false: FAIL (0.01 s)
RESULTS    : PASS 1 | ERROR 0 | FAIL 1 | SKIP 0 | WARN 0 | INTERRUPT 0
JOB TIME   : 0.11 s
JOB HTML   : $HOME/avocado/job-results/job-2016-01-11T16.18-55a0d10/html/results.html
```

The replay feature will retrieve the original test references, the variants and the configuration. Let's see another example, now using mux yaml file:

```
$ avocado run /bin/true /bin/false --mux-yaml mux-environment.yaml
JOB ID     : bd6aa3b852d4290637b5e771b371537541043d1d
JOB LOG    : $HOME/avocado/job-results/job-2016-01-11T21.56-bd6aa3b/job.log
 (1/48) /bin/true;1: PASS (0.01 s)
```

```
 (2/48) /bin/true;2: PASS (0.01 s)
 (3/48) /bin/true;3: PASS (0.01 s)
 (4/48) /bin/true;4: PASS (0.01 s)
 (5/48) /bin/true;5: PASS (0.01 s)
 (6/48) /bin/true;6: PASS (0.01 s)
 (7/48) /bin/true;7: PASS (0.01 s)
 (8/48) /bin/true;8: PASS (0.01 s)
 (9/48) /bin/true;9: PASS (0.01 s)
 (10/48) /bin/true;10: PASS (0.01 s)
 (11/48) /bin/true;11: PASS (0.01 s)
 (12/48) /bin/true;12: PASS (0.01 s)
 (13/48) /bin/true;13: PASS (0.01 s)
 (14/48) /bin/true;14: PASS (0.01 s)
 (15/48) /bin/true;15: PASS (0.01 s)
 (16/48) /bin/true;16: PASS (0.01 s)
 (17/48) /bin/true;17: PASS (0.01 s)
 (18/48) /bin/true;18: PASS (0.01 s)
 (19/48) /bin/true;19: PASS (0.01 s)
 (20/48) /bin/true;20: PASS (0.01 s)
 (21/48) /bin/true;21: PASS (0.01 s)
 (22/48) /bin/true;22: PASS (0.01 s)
 (23/48) /bin/true;23: PASS (0.01 s)
 (24/48) /bin/true;24: PASS (0.01 s)
 (25/48) /bin/false;1: FAIL (0.01 s)
 (26/48) /bin/false;2: FAIL (0.01 s)
 (27/48) /bin/false;3: FAIL (0.01 s)
 (28/48) /bin/false;4: FAIL (0.01 s)
 (29/48) /bin/false;5: FAIL (0.01 s)
 (30/48) /bin/false;6: FAIL (0.01 s)
 (31/48) /bin/false;7: FAIL (0.01 s)
 (32/48) /bin/false;8: FAIL (0.01 s)
 (33/48) /bin/false;9: FAIL (0.01 s)
 (34/48) /bin/false;10: FAIL (0.01 s)
 (35/48) /bin/false;11: FAIL (0.01 s)
 (36/48) /bin/false;12: FAIL (0.01 s)
 (37/48) /bin/false;13: FAIL (0.01 s)
 (38/48) /bin/false;14: FAIL (0.01 s)
 (39/48) /bin/false;15: FAIL (0.01 s)
 (40/48) /bin/false;16: FAIL (0.01 s)
 (41/48) /bin/false;17: FAIL (0.01 s)
 (42/48) /bin/false;18: FAIL (0.01 s)
 (43/48) /bin/false;19: FAIL (0.01 s)
 (44/48) /bin/false;20: FAIL (0.01 s)
 (45/48) /bin/false;21: FAIL (0.01 s)
 (46/48) /bin/false;22: FAIL (0.01 s)
 (47/48) /bin/false;23: FAIL (0.01 s)
 (48/48) /bin/false;24: FAIL (0.01 s)
RESULTS    : PASS 24 | ERROR 0 | FAIL 24 | SKIP 0 | WARN 0 | INTERRUPT 0
JOB TIME   : 0.19 s
JOB HTML   : $HOME/avocado/job-results/job-2016-01-11T21.56-bd6aa3b/html/results.html
```

We can replay the job as is, using `$ avocado run --replay latest`, or replay the job ignoring the variants, as below:

```
$ avocado run --replay bd6aa3b --replay-ignore variants
Ignoring variants from source job with --replay-ignore.
JOB ID     : d5a46186ee0fb4645e3f7758814003d76c980bf9
SRC JOB ID : bd6aa3b852d4290637b5e771b371537541043d1d
```

```
JOB LOG    : $HOME/avocado/job-results/job-2016-01-11T22.01-d5a4618/job.log
 (1/2) /bin/true: PASS (0.01 s)
 (2/2) /bin/false: FAIL (0.01 s)
RESULTS    : PASS 1 | ERROR 0 | FAIL 1 | SKIP 0 | WARN 0 | INTERRUPT 0
JOB TIME   : 0.12 s
JOB HTML   : $HOME/avocado/job-results/job-2016-01-11T22.01-d5a4618/html/results.html
```

Also, it is possible to replay only the variants that faced a given result, using the option `--replay-test-status`. See the example below:

```
$ avocado run --replay bd6aa3b --replay-test-status FAIL
JOB ID     : 2e1dc41af6ed64895f3bb45e3820c5cc62a9b6eb
SRC JOB ID : bd6aa3b852d4290637b5e771b371537541043d1d
JOB LOG    : $HOME/avocado/job-results/job-2016-01-12T00.38-2e1dc41/job.log
 (1/48) /bin/true;1: SKIP
 (2/48) /bin/true;2: SKIP
 (3/48) /bin/true;3: SKIP
 (4/48) /bin/true;4: SKIP
 (5/48) /bin/true;5: SKIP
 (6/48) /bin/true;6: SKIP
 (7/48) /bin/true;7: SKIP
 (8/48) /bin/true;8: SKIP
 (9/48) /bin/true;9: SKIP
 (10/48) /bin/true;10: SKIP
 (11/48) /bin/true;11: SKIP
 (12/48) /bin/true;12: SKIP
 (13/48) /bin/true;13: SKIP
 (14/48) /bin/true;14: SKIP
 (15/48) /bin/true;15: SKIP
 (16/48) /bin/true;16: SKIP
 (17/48) /bin/true;17: SKIP
 (18/48) /bin/true;18: SKIP
 (19/48) /bin/true;19: SKIP
 (20/48) /bin/true;20: SKIP
 (21/48) /bin/true;21: SKIP
 (22/48) /bin/true;22: SKIP
 (23/48) /bin/true;23: SKIP
 (24/48) /bin/true;24: SKIP
 (25/48) /bin/false;1: FAIL (0.01 s)
 (26/48) /bin/false;2: FAIL (0.01 s)
 (27/48) /bin/false;3: FAIL (0.01 s)
 (28/48) /bin/false;4: FAIL (0.01 s)
 (29/48) /bin/false;5: FAIL (0.01 s)
 (30/48) /bin/false;6: FAIL (0.01 s)
 (31/48) /bin/false;7: FAIL (0.01 s)
 (32/48) /bin/false;8: FAIL (0.01 s)
 (33/48) /bin/false;9: FAIL (0.01 s)
 (34/48) /bin/false;10: FAIL (0.01 s)
 (35/48) /bin/false;11: FAIL (0.01 s)
 (36/48) /bin/false;12: FAIL (0.01 s)
 (37/48) /bin/false;13: FAIL (0.01 s)
 (38/48) /bin/false;14: FAIL (0.01 s)
 (39/48) /bin/false;15: FAIL (0.01 s)
 (40/48) /bin/false;16: FAIL (0.01 s)
 (41/48) /bin/false;17: FAIL (0.01 s)
 (42/48) /bin/false;18: FAIL (0.01 s)
 (43/48) /bin/false;19: FAIL (0.01 s)
 (44/48) /bin/false;20: FAIL (0.01 s)
```

```
 (45/48) /bin/false;21: FAIL (0.01 s)
 (46/48) /bin/false;22: FAIL (0.01 s)
 (47/48) /bin/false;23: FAIL (0.01 s)
 (48/48) /bin/false;24: FAIL (0.01 s)
RESULTS    : PASS 0 | ERROR 0 | FAIL 24 | SKIP 24 | WARN 0 | INTERRUPT 0
JOB TIME   : 0.29 s
JOB HTML   : $HOME/avocado/job-results/job-2016-01-12T00.38-2e1dc41/html/results.html
```

Of which one special example is `--replay-test-status INTERRUPTED` or simply `--replay-resume`, which SKIPs the executed tests and only executes the ones which were CANCELED or not executed after a CANCELED test. This feature should work even on hard interruptions like system crash.

When replaying jobs that were executed with the `--failfast on` option, you can disable the `failfast` option using `--failfast off` in the replay job.

To be able to replay a job, avocado records the job data in the same job results directory, inside a subdirectory named `replay`. If a given job has a non-default path to record the logs, when the replay time comes, we need to inform where the logs are. See the example below:

```
$ avocado run /bin/true --job-results-dir /tmp/avocado_results/
JOB ID     : f1b1c870ad892eac6064a5332f1bbe38cda0aaf3
JOB LOG    : /tmp/avocado_results/job-2016-01-11T22.10-f1b1c87/job.log
 (1/1) /bin/true: PASS (0.01 s)
RESULTS    : PASS 1 | ERROR 0 | FAIL 0 | SKIP 0 | WARN 0 | INTERRUPT 0
JOB TIME   : 0.11 s
JOB HTML   : /tmp/avocado_results/job-2016-01-11T22.10-f1b1c87/html/results.html
```

Trying to replay the job, it fails:

```
$ avocado run --replay f1b1
can't find job results directory in '$HOME/avocado/job-results'
```

In this case, we have to inform where the job results directory is located:

```
$ avocado run --replay f1b1 --replay-data-dir /tmp/avocado_results
JOB ID     : 19c76abb29f29fe410a9a3f4f4b66387570edffa
SRC JOB ID : f1b1c870ad892eac6064a5332f1bbe38cda0aaf3
JOB LOG    : $HOME/avocado/job-results/job-2016-01-11T22.15-19c76ab/job.log
 (1/1) /bin/true: PASS (0.01 s)
RESULTS    : PASS 1 | ERROR 0 | FAIL 0 | SKIP 0 | WARN 0 | INTERRUPT 0
JOB TIME   : 0.11 s
JOB HTML   : $HOME/avocado/job-results/job-2016-01-11T22.15-19c76ab/html/results.html
```

CHAPTER 11

# Job Diff

Avocado Diff plugin allows users to easily compare several aspects of two given jobs. The basic usage is:

```
$ avocado diff 7025aaba 384b949c
--- 7025aaba9c2ab8b4bba2e33b64db3824810bb5df
+++ 384b949c991b8ab324ce67c9d9ba761fd07672ff
@@ -1,15 +1,15 @@

 COMMAND LINE
-/usr/bin/avocado run sleeptest.py
+/usr/bin/avocado run passtest.py

 TOTAL TIME
-1.00 s
+0.00 s

 TEST RESULTS
-1-sleeptest.py:SleepTest.test: PASS
+1-passtest.py:PassTest.test: PASS

 ...
```

Avocado Diff can compare and create an unified diff of:

- Command line.
- Job time.
- Variants and parameters.
- Tests results.
- Configuration.
- Sysinfo pre and post.

Only sections with different content will be included in the results. You can also enable/disable those sections with `--diff-filter`. Please see `avocado diff --help` for more information.

Jobs can be identified by the Job ID, by the results directory or by the key `latest`. Example:

```
$ avocado diff ~/avocado/job-results/job-2016-08-03T15.56-4b3cb5b/ latest
--- 4b3cb5bbbb2435c91c7b557eebc09997d4a0f544
+++ 57e5bbb3991718b216d787848171b446f60b3262
@@ -1,9 +1,9 @@

 COMMAND LINE
-/usr/bin/avocado run perfmon.py
+/usr/bin/avocado run passtest.py

 TOTAL TIME
-11.91 s
+0.00 s

 TEST RESULTS
-1-test.py:Perfmon.test: FAIL
+1-examples/tests/passtest.py:PassTest.test: PASS
```

Along with the unified diff, you can also generate the html (option `--html`) diff file and, optionally, open it on your preferred browser (option `--open-browser`):

```
$ avocado diff 7025aaba 384b949c --html /tmp/myjobdiff.html
/tmp/myjobdiff.html
```

If the option `--open-browser` is used without the `--html`, we will create a temporary html file.

For those wiling to use a custom diff tool instead of the Avocado Diff tool, we offer the option `--create-reports`, so we create two temporary files with the relevant content. The file names are printed and user can copy/paste to the custom diff tool command line:

```
$ avocado diff 7025aaba 384b949c --create-reports
/var/tmp/avocado_diff_7025aab_zQJjJh.txt /var/tmp/avocado_diff_384b949_AcWq02.txt

$ diff -u /var/tmp/avocado_diff_7025aab_zQJjJh.txt /var/tmp/avocado_diff_384b949_
↪AcWq02.txt
--- /var/tmp/avocado_diff_7025aab_zQJjJh.txt    2016-08-10 21:48:43.547776715 +0200
+++ /var/tmp/avocado_diff_384b949_AcWq02.txt    2016-08-10 21:48:43.547776715 +0200
@@ -1,250 +1,19 @@

 COMMAND LINE
 ============
-/usr/bin/avocado run sleeptest.py
+/usr/bin/avocado run passtest.py

 TOTAL TIME
 ==========
-1.00 s
+0.00 s

...
```

CHAPTER 12

# Running Tests Remotely

## Running Tests on a Remote Host

Avocado lets you run tests directly in a remote machine with SSH connection, provided that you properly set it up by installing Avocado in it.

You can check if this feature (a plugin) is enabled by running:

```
$ avocado plugins
...
remote  Remote machine options for 'run' subcommand
...
```

Assuming this feature is enabled, you should be able to pass the following options when using the `run` command in the Avocado command line tool:

```
--remote-hostname REMOTE_HOSTNAME
                      Specify the hostname to login on remote machine
--remote-port REMOTE_PORT
                      Specify the port number to login on remote machine.
                      Default: 22
--remote-username REMOTE_USERNAME
                      Specify the username to login on remote machine
--remote-password REMOTE_PASSWORD
                      Specify the password to login on remote machine
```

From these options, you are normally going to use *–remote-hostname* and *–remote-username* in case you did set up your VM with password-less SSH connection (through SSH keys).

### Remote Setup

Make sure you have:

1. Avocado packages installed. You can see more info on how to do that in the *Getting Started* section.

2. The remote machine IP address or fully qualified hostname and the SSH port number.
3. All pre-requisites for your test to run installed inside the remote machine (gcc, make and others if you want to compile a 3rd party test suite written in C, for example).

Optionally, you may have password less SSH login on your remote machine enabled.

### Running your test

Once the remote machine is properly set, you may run your test. Example:

```
$ scripts/avocado run --remote-hostname 192.168.122.30 --remote-username fedora␣
↪examples/tests/sleeptest.py examples/tests/failtest.py
REMOTE LOGIN  : fedora@192.168.122.30:22
JOB ID    : 60ddd718e7d7bb679f258920ce3c39ce73cb9779
JOB LOG   : $HOME/avocado/job-results/job-2014-10-23T11.45-a329461/job.log
 (1/2) examples/tests/sleeptest.py: PASS (1.00 s)
 (2/2) examples/tests/failtest.py: FAIL (0.00 s)
RESULTS    : PASS 1 | ERROR 0 | FAIL 1 | SKIP 0 | WARN 0 | INTERRUPT 0
JOB TIME   : 1.11 s
```

A bit of extra logging information is added to your job summary, mainly to distinguish the regular execution from the remote one. Note here that we did not need *–remote-password* because an SSH key was already set.

## Running Tests on a Virtual Machine

Sometimes you don't want to run a given test directly in your own machine (maybe the test is dangerous, maybe you need to run it in another Linux distribution, so on and so forth).

For those scenarios, Avocado lets you run tests directly in VMs defined as libvirt domains in your system, provided that you properly set them up.

You can check if this feature (a plugin) is enabled by running:

```
$ avocado plugins
...
vm      Virtual Machine options for 'run' subcommand
...
```

Assuming this feature is enabled, you should be able to pass the following options when using the `run` command in the Avocado command line tool:

```
--vm                  Run tests on Virtual Machine
--vm-hypervisor-uri VM_HYPERVISOR_URI
                      Specify hypervisor URI driver connection
--vm-domain VM_DOMAIN
                      Specify domain name (Virtual Machine name)
--vm-hostname VM_HOSTNAME
                      Specify VM hostname to login. By default Avocado
                      attempts to automatically find the VM IP address.
--vm-username VM_USERNAME
                      Specify the username to login on VM
--vm-password VM_PASSWORD
                      Specify the password to login on VM
--vm-cleanup          Restore VM to a previous state, before running the
                      tests
```

From these options, you are normally going to use *–vm-domain*, *–vm-hostname* and *–vm-username* in case you did set up your VM with password-less SSH connection (through SSH keys).

If your VM has the `qemu-guest-agent` installed, you can skip the `--vm-hostname` option. Avocado will then probe the VM IP from the agent.

### Virtual Machine Setup

Make sure you have:

1. A libvirt domain with the Avocado packages installed. You can see more info on how to do that in the *Getting Started* section.
2. The domain IP address or fully qualified hostname.
3. All pre-requesites for your test to run installed inside the VM (gcc, make and others if you want to compile a 3rd party test suite written in C, for example).

Optionally, you may have password less SSH login on your VM enabled.

### Running your test

Once the virtual machine is properly set, you may run your test. Example:

```
$ scripts/avocado run --vm-domain fedora20 --vm-username autotest --vm examples/tests/
↪sleeptest.py examples/tests/failtest.py
VM DOMAIN : fedora20
VM LOGIN  : autotest@192.168.122.30
JOB ID    : 60ddd718e7d7bb679f258920ce3c39ce73cb9779
JOB LOG   : $HOME/avocado/job-results/job-2014-09-16T18.41-60ddd71/job.log
 (1/2) examples/tests/sleeptest.py:SleepTest.test: PASS (1.00 s)
 (2/2) examples/tests/failtest.py:FailTest.test: FAIL (0.01 s)
RESULTS    : PASS 1 | ERROR 0 | FAIL 1 | SKIP 0 | WARN 0 | INTERRUPT 0
JOB TIME   : 1.11 s
```

A bit of extra logging information is added to your job summary, mainly to distinguish the regular execution from the remote one. Note here that we did not need *–vm-password* because the SSH key is already set.

## Running Tests on a Docker container

Avocado also lets you run tests on a Docker container, starting and cleaning it up automatically with every execution.

You can check if this feature (a plugin) is enabled by running:

```
$ avocado plugins
...
docker  Run tests inside docker container
...
```

### Docker container images

Avocado needs to be present inside the container image in order for the test execution to be properly performed. There's one ready to use image (`ldoktor/fedora-avocado`) in the default image repository (`docker.io`):

```
$ sudo docker pull ldoktor/fedora-avocado
Using default tag: latest
Trying to pull repository docker.io/ldoktor/fedora-avocado ...
latest: Pulling from docker.io/ldoktor/fedora-avocado
...
Status: Downloaded newer image for docker.io/ldoktor/fedora-avocado:latest
```

### Use custom docker images

One of the possible ways to use (and develop) Avocado is to create a docker image with your development tree. This is a good way to test your development branch without breaking your system.

To do so, you can following a few simple steps. Begin by fetching the source code as usual:

```
$ git clone github.com/avocado-framework/avocado.git avocado.git
```

You may want to make some changes to Avocado:

```
$ cd avocado.git
$ patch -p1 < MY_PATCH
```

Finally build a docker image:

```
$ docker build -t fedora-avocado-custom -f contrib/docker/Dockerfile.fedora .
```

And now you can run tests with your modified Avocado inside your container:

```
$ avocado run --docker fedora-avocado-custom examples/tests/passtest.py
```

### Running your test

Assuming your system is properly set to run Docker, including having an image with Avocado, you can run a test inside the container with a command similar to:

```
$ avocado run passtest.py warntest.py failtest.py --docker ldoktor/fedora-avocado --
↪docker-cmd "sudo docker"
JOB ID     : db309f5daba562235834f97cad5f4458e3fe6e32
JOB LOG    : $HOME/avocado/job-results/job-2016-07-25T08.01-db309f5/job.log
DOCKER     : Container id
↪'4bcbcd69801211501a0dde5926c0282a9630adbe29ecb17a21ef04f024366943'
DOCKER     : Container name 'job-2016-07-25T08.01-db309f5.avocado'
 (1/3) /avocado_remote_test_dir/$HOME/passtest.py:PassTest.test: PASS (0.00 s)
 (2/3) /avocado_remote_test_dir/$HOME/warntest.py:WarnTest.test: WARN (0.00 s)
 (3/3) /avocado_remote_test_dir/$HOME/failtest.py:FailTest.test: FAIL (0.00 s)
RESULTS    : PASS 1 | ERROR 0 | FAIL 1 | SKIP 0 | WARN 1 | INTERRUPT 0
JOB TIME   : 0.10 s
JOB HTML   : $HOME/avocado/job-results/job-2016-07-25T08.01-db309f5/html/results.html
```

## Environment Variables

Running remote instances os Avocado, for example using *remote* or *vm* plugins, the remote environment has a different set of environment variables. If you want to make available remotely variables that are available in the local

environment, you can use the *run* option *–env-keep*. See the example below:

```
$ export MYVAR1=foobar
$ env MYVAR2=foobar2 avocado run passtest.py --env-keep MYVAR1,MYVAR2 --remote-
↪hostname 192.168.122.30 --remote-username fedora
```

By doing that, both *MYVAR1* and *MYVAR2* will be available in remote environment.

## Known Issues

Given the modular architecture of Avocado, the fact that the `remote` feature is a plugin and also the fact that the plugins are engaged in no particular order, other plugins will not have the information that we are in a remote execution. As consequence, plugins that look for local resources that are available only remotely can fail. That's the case of the so called `multiplex` plugin. If you're using the multiplex plugin (`-m` or `--mux-yaml`) options in addition to the remote plugin (or any derived plugin, like `vm` or `docker`), the multiplex files must exist locally in the provided path. Notice the multiplex files must be also available remotely in the provided path, since we don't copy files for remote executions.

CHAPTER 13

# Debugging with GDB

Avocado has two different types of GDB support that complement each other:

- Transparent execution of executables inside the GNU Debugger. This takes standard and possibly unmodified tests that uses the *avocado.utils.process* APIs for running processes. By using a command line option, the executable is run on GDB. This allows the user to interact with GDB, but to the test itself, things are pretty much transparent.
- The *avocado.utils.gdb* APIs that allows a test to interact with GDB, including setting a executable to be run, setting breakpoints or any other types of commands. This requires a test written with that approach and API in mind.

**Tip:** Even though this section describes the use of the Avocado GDB features, which allow live debugging of binaries inside Avocado tests, it's also possible to debug some application offline by using tools such as rr. Avocado ships with an example wrapper script (to be used with `--wrapper`) for that purpose.

## Transparent Execution of Executables

This feature adds a few command line options to the Avocado `run` command:

```
$ avocado run --help
...
GNU Debugger support:

  --gdb-run-bin EXECUTABLE[:BREAKPOINT]
                        Run a given executable inside the GNU debugger,
                        pausing at a given breakpoint (defaults to "main")
  --gdb-prerun-commands EXECUTABLE:COMMANDS
                        After loading an executable in GDB, but before
                        actually running it, execute the GDB commands in the
                        given file. EXECUTABLE is optional, if omitted
                        COMMANDS will apply to all executables
```

```
  --gdb-coredump {on,off}
                        Automatically generate a core dump when the inferior
                        process received a fatal signal such as SIGSEGV or
                        SIGABRT
...
```

To get started you want to use `--gdb-run-bin`, as shown in the example bellow.

## Example

The simplest way is to just run `avocado run --gdb-run-bin=doublefree examples/tests/doublefree.py`, which wraps each executed executable with name `doublefree` inside GDB server and stops at the executable entry point.

Optionally you can specify single breakpoint using `--gdb-run-bin=doublefree:$breakpoint` (eg: `doublefree:1`) or just `doublefree:` to stop only when an interruption happens (eg: SIGABRT).

It's worth mentioning that when breakpoint is not reached, the test finishes without any interruption. This is helpful when you identify regions where you should never get in your code, or places which interests you and you can run your code in production and GDB variants. If after a long time you get to this place, the test notifies you and you can investigate the problem. This is demonstrated in `examples/tests/doublefree_nasty.py` test. To unveil the power of Avocado, run this test using:

```
avocado run --gdb-run-bin=doublefree: examples/tests/doublefree_nasty.py --gdb-prerun-
↪commands examples/tests/doublefree_nasty.py.data/gdb_pre --mux-yaml examples/tests/
↪doublefree_nasty.py.data/iterations.yaml
```

which executes 100 iterations of this test while setting all breakpoints from the `examples/tests/doublefree_nasty.py.data/gdb_pre` file (you can specify whatever GDB supports, not only breakpoints).

As you can see this test usually passes, but once in a while it gets into the problematic area. Imagine this is very hard to spot (dependent on HW registers, ...) and this is one way to combine regular testing and the possibility of debugging hard-to-get parts of your code.

## Caveats

Currently, when using the Avocado GDB plugin, that is, when using the *–gdb-run-bin* option, there are some caveats you should be aware of:

- It is not currently compatible with Avocado's *–output-check-record* feature
- There's no way to perform proper input to the process, that is, manipulate its *STDIN*
- The process *STDERR* content is mixed with the content generated by *gdbserver* on its own *STDERR* (because they are in fact, the same thing)

But, you can still depend on the process *STDOUT*, as exemplified by this fictional test:

```
from avocado import Test
from avocado.utils import process

class HelloOutputTest(Test):

    def test(self):
        result = process.run("/path/to/hello", ignore_status=True)
        self.assertIn("hello\n", result.stdout)
```

If run under GDB or not, *result.stdout* behavior and content is expected to be the same.

### Reasons for the caveats

There are a two basic reasons for the mentioned caveats:

- The architecture of Avocado's GDB feature
- GDB's own behavior and limitations

When using the Avocado GDB plugin, that is, *–gdb-run-bin*, Avocado runs a *gdbserver* instance transparently and controls it by means of a *gdb* process. When a given event happens, say a breakpoint is reached, it disconnects its own *gdb* from the server, and allows the user to use a standard *gdb* to connect to the *gdbserver*. This provides a natural and seamless user experience.

But, *gdbserver* has some limitations at this point, including:

- Not being able to set a controlling *tty*
- Not separating its own *STDERR* content from the application being run

These limitations are being addressed both on Avocado and GDB, and will be resolved in future Avocado versions.

### Workaround

If the application you're running as part of your test can read input from alternative sources (including devices, files or the network) and generate output likewise, then you should not be further limited.

### GDB support and avocado-virt

Another current limitation is the use of *avocado-virt* and *avocado* GDB support.

The supported API for transparent debugging is currently limited to *avocado.utils.process.run()*, and does not cover advanced uses of the *avocado.utils.process.SubProcess* class. The *avocado-virt* extension, though, uses *avocado.utils.process.SubProcess* class to execute *qemu* in the background.

This limitation will be addressed in future versions of *avocado* and *avocado-virt*.

## avocado.utils.gdb APIs

Avocado's GDB module, provides three main classes that lets a test writer interact with a *gdb* process, a *gdbserver* process and also use the GDB remote protocol for interaction with a remote target.

Please refer to *avocado.utils.gdb* for more information.

### Example

Take a look at `examples/tests/modify_variable.py` test:

```
def test(self):
    """
    Execute 'print_variable'.
    """
    path = os.path.join(self.srcdir, 'print_variable')
```

```
    app = gdb.GDB()
    app.set_file(path)
    app.set_break(6)
    app.run()
    self.log.info("\n".join(app.read_until_break()))
    app.cmd("set variable a = 0xff")
    app.cmd("c")
    out = "\n".join(app.read_until_break())
    self.log.info(out)
    app.exit()
    self.assertIn("MY VARIABLE 'A' IS: ff", out)
```

You can see that instead of running the executable using `process.run` we invoke *avocado.utils.gdb.GDB*. This allows us to automate the interaction with the GDB in means of setting breakpoints, executing commands and querying for output.

When you check the output (`--show-job-log`) you can see that despite declaring the variable as 0, ff is injected and printed instead.

CHAPTER 14

# Wrap executables run by tests

Avocado allows the instrumentation of executables being run by a test in a transparent way. The user specifies a script ("the wrapper") to be used to run the actual program called by the test.

If the instrumentation script is implemented correctly, it should not interfere with the test behavior. That is, the wrapper should avoid changing the return status, standard output and standard error messages of the original executable.

The user can be specific about which program to wrap (with a shell-like glob), or if that is omitted, a global wrapper that will apply to all programs called by the test.

## Usage

This feature is implemented as a plugin, that adds the *–wrapper* option to the Avocado *run* command. For a detailed explanation, please consult the Avocado man page.

Example of a transparent way of running strace as a wrapper:

```
#!/bin/sh
exec strace -ff -o $AVOCADO_TEST_LOGDIR/strace.log -- $@
```

To have all programs started by `test.py` wrapped with `~/bin/my-wrapper.sh`:

```
$ scripts/avocado run --wrapper ~/bin/my-wrapper.sh tests/test.py
```

To have only `my-binary` wrapped with `~/bin/my-wrapper.sh`:

```
$ scripts/avocado run --wrapper ~/bin/my-wrapper.sh:*my-binary tests/test.py
```

## Caveats

- It is not possible to debug with GDB (*–gdb-run-bin*) and use wrappers (*–wrapper*) at the same time. These two options are mutually exclusive.

- You can only set one (global) wrapper. If you need functionality present in two wrappers, you have to combine those into a single wrapper script.
- Only executables that are run with the `avocado.utils.process` APIs (and other API modules that make use of it, like mod:*avocado.utils.build*) are affected by this feature.

CHAPTER 15

# Plugin System

Avocado has a plugin system that can be used to extended it in a clean way.

## Listing plugins

The `avocado` command line tool has a builtin `plugins` command that lets you list available plugins. The usage is pretty simple:

```
$ avocado plugins
Plugins that add new commands (avocado.plugins.cli.cmd):
exec-path Returns path to avocado bash libraries and exits.
run       Run one or more tests (native test, test alias, binary or script)
sysinfo   Collect system information
...
Plugins that add new options to commands (avocado.plugins.cli):
remote  Remote machine options for 'run' subcommand
journal Journal options for the 'run' subcommand
...
```

Since plugins are (usually small) bundles of Python code, they may fail to load if the Python code is broken for any reason. Example:

```
$ avocado plugins
Failed to load plugin from module "avocado.plugins.exec_path": ImportError('No module␣
↪named foo',)
Plugins that add new commands (avocado.plugins.cli.cmd):
run       Run one or more tests (native test, test alias, binary or script)
sysinfo   Collect system information
...
```

# Writing a plugin

What better way to understand how an Avocado plugin works than creating one? Let's use another old time favorite for that, the "Print hello world" theme.

## Code example

Let's say you want to write a plugin that adds a new subcommand to the test runner, `hello`. This is how you'd do it:

```
from avocado.core.output import LOG_JOB
from avocado.core.plugin_interfaces import CLICmd


class HelloWorld(CLICmd):

    name = 'hello'
    description = 'The classical Hello World! plugin example.'

    def run(self, args):
        LOG_JOB.info(self.description)
```

As you can see, this plugins inherits from *avocado.core.plugin_interfaces.CLICmd*. This specific base class allows for the creation of new commands for the Avocado CLI tool. The only mandatory method to be implemented is *run* and it's the plugin main entry point.

This plugin uses *avocado.core.output.LOG_JOB* to produce the hello world output in the Job log. One can also use *avocado.core.output.LOG_UI* to produce output in the human readable output.

## Registering Plugins

Avocado makes use of the Stevedore library to load and activate plugins. Stevedore itself uses setuptools and its entry points to register and find Python objects. So, to make your new plugin visible to Avocado, you need to add to your setuptools based *setup.py* file something like:

```
setup(name='mypluginpack',
...
entry_points={
   'avocado.plugins.cli': [
      'hello = mypluginpack.hello:HelloWorld',
   ]
}
...
```

Then, by running either `$ python setup.py install` or `$ python setup.py develop` your plugin should be visible to Avocado.

## Fully qualified named for a plugin

The plugin registry mentioned earlier, (setuptools and its entry points) is global to a given Python installation. Avocado uses the namespace prefix `avocado.plugins.` to avoid name clashes with other software. Now, inside Avocado itself, there's no need keep using the `avocado.plugins.` prefix.

Take for instance, the Job Pre/Post plugins are defined on `setup.py`:

```
'avocado.plugins.job.prepost': [
   'jobscripts = avocado.plugins.jobscripts:JobScripts'
]
```

The setuptools entry point namespace is composed of the mentioned prefix `avocado.plugins.`, which is is then followed by the Avocado plugin type, in this case, `job.prepost`.

Inside avocado itself, the fully qualified name for a plugin is the plugin type, such as `job.prepost` concatenated to the name used in the entry point definition itself, in this case, `jobscripts`.

To summarize, still using the same example, the fully qualified Avocado plugin name is going to be `job.prepost.jobscripts`.

## Disabling a plugin

Even though a plugin can be installed and registered under setuptools entry points, it can be explicitly disabled in Avocado.

The mechanism available to do so is to add entries to the `disable` key under the `plugins` section of the Avocado configuration file. Example:

```
[plugins]
disable = ['cli.hello', 'job.prepost.jobscripts']
```

The exact effect on Avocado when a plugin is disabled depends on the plugin type. For instance, by disabling plugins of type `cli.cmd`, the command implemented by the plugin should no longer be available on the Avocado command line application. Now, by disabling a `job.prepost` plugin, those won't be executed before/after the execution of the jobs.

## Default plugin execution order

In many situations, such as result generation, not one, but all of the enabled plugin types will be executed. The order in which the plugins are executed follows the lexical order of the entry point name.

For example, for the JSON result plugin, whose fully qualified name is `result.json`, has an entry point name of `json`, as can be seen on its registration code in `setup.py`:

```
...
entry_points={
   'avocado.plugins.result': [
      'json = avocado.plugins.jsonresult:JSONResult',
...
```

If it sounds too complicated, it isn't. It just means that for plugins of the same type, a plugin named `automated` will be executed before the plugin named `uploader`.

In the default Avocado set of result plugins, it means that the JSON plugin (`json`) will be executed before the XUnit plugin (`xunit`). If the HTML result plugin is installed and enabled (`html`) it will be executed before both JSON and XUnit.

## Configuring the plugin execution order

On some circumstances it may be necessary to change the order in which plugins are executed. To do so, add a `order` entry a configuration file section named after the plugin type. For `job.prepost` plugin types, the section name has to be named `plugins.job.prepost`, and it would look like this:

```
[plugins.job.prepost]
order = ['myplugin', 'jobscripts']
```

That configuration sets the `job.prepost.myplugin` plugin to execute before the standard Avocado `job.prepost.jobscripts` does.

## Wrap Up

We have briefly discussed the making of Avocado plugins. We recommend the Stevedore documentation and also a look at the *avocado.core.plugin_interfaces* module for the various plugin interface definitions.

Some plugins examples are available in the Avocado source tree, under `examples/plugins`.

Finally, exploring the real plugins shipped with Avocado in *avocado.plugins* is the final "documentation" source.

CHAPTER 16

# Optional Plugins

## Optional Plugins

The following pages are the documentation for some of the Avocado optional plugins:

### Result plugins

Optional plugins providing various types of job results.

#### HTML results Plugin

This optional plugin creates beautiful human readable results.

To install the HTML plugin from pip, use:

```
pip install avocado-framework-plugin-result-html
```

Once installed it produces the results in job results dir:

```
$ avocado run sleeptest.py failtest.py synctest.py
...
JOB HTML  : /home/medic/avocado/job-results/job-2014-08-12T15.57-5ffe4792/html/
↪results.html
...
```

This can be disabled via –html-job-result on|off. One can also specify a custom location via –html . Last but not least –open-browser can be used to start browser automatically once the job finishes.

#### ResultsDB Plugin

This optional plugin is intended to propagate the Avocado Job results to a given ResultsDB API URL.

To install the ResultsDB plugin from pip, use:

```
pip install avocado-framework-plugin-resultsdb
```

Usage:

```
avocado run passtest.py --resultsdb-api http://resultsdb.example.com/api/v2.0/
```

Optionally, you can provide the URL where the Avocado logs are published:

```
avocado run passtest.py --resultsdb-api http://resultsdb.example.com/api/v2.0/ --
↪resultsdb-logs http://avocadologs.example.com/
```

The –resultsdb-logs is a convenience option that will create links to the logs in the ResultsDB records. The links will then have the following formats:

- ResultDB group (Avocado Job):

  ```
  http://avocadologs.example.com/job-2017-04-21T12.54-1cefe11/
  ```

- ResultDB result (Avocado Test):

  ```
  http://avocadologs.example.com/job-2017-04-21T12.54-1cefe11/test-results/1-
  ↪passtest.py:PassTest.test/
  ```

You can also set the ResultsDB API URL and logs URL using a config file:

```
[plugins.resultsdb]
api_url = http://resultsdb.example.com/api/v2.0/
logs_url = http://avocadologs.example.com/
```

And then run the Avocado command without the –resultsdb-api and –resultsdb-logs options. Notice that the command line options will have precedence over the configuration file.

## Robot Plugin

This optional plugin enables Avocado to work with tests originally written using the Robot Framework API.

To install the Robot plugin from pip, use:

```
$ sudo pip install avocado-framework-plugin-robot
```

After installed, you can list/run Robot tests the same way you do with other types of tests.

To list the tests, execute:

```
$ avocado list ~/path/to/robot/tests/test.robot
```

Directories are also accepted. To run the tests, execute:

```
$ avocado run ~/path/to/robot/tests/test.robot
```

CHAPTER 17

# Advanced Topics and Maintenance

## Reference Guide

This guide presents information on the Avocado basic design and its internals.

### Job, test and identifiers

#### Job ID

The Job ID is a random SHA1 string that uniquely identifies a given job.

The full form of the SHA1 string is used is most references to a job:

```
$ avocado run sleeptest.py
JOB ID     : 49ec339a6cca73397be21866453985f88713ac34
...
```

But a shorter version is also used at some places, such as in the job results location:

```
JOB LOG    : $HOME/avocado/job-results/job-2015-06-10T10.44-49ec339/job.log
```

#### Test References

A Test Reference is a string that can be resolved into (interpreted as) one or more tests by the Avocado Test Resolver. A given resolver plugin is free to interpret a test reference, it is completely abstract to the other components of Avocado.

---

**Note:** Mapping the Test References to tests can be affected by command-line switches like *–external-runner*, which completelly changes the meaning of the given strings.

---

### Test Name

A test name is an arbitrarily long string that unambiguously points to the source of a single test. In other words the Avocado Test Resolver, as configured for a particular job, should return one and only one test as the interpretation of this name.

This name can be as specific as necessary to make it unique. Therefore it can contain an arbitrary number of variables, prefixes, suffixes, tags, etc. It all depends on user preferences, what is supported by Avocado via its Test Resolvers and the context of the job.

The output of the Test Resolver when resolving Test References should always be a list of unambiguous Test Names (for that particular job).

Notice that although the Test Name has to be unique, one test can be run more than once inside a job.

By definition, a Test Name is a Test Reference, but the reciprocal is not necessarily true, as the latter can represent more than one test.

### Variant IDs

The varianter component creates different sets of variables (known as "variants"), to allow tests to be run individually in each of them.

A Variant ID is an arbitrary and abstract string created by the varianter plugin to identify each variant. It should be unique per variant inside a set. In other words, the varianter plugin generates a set of variants, identified by unique IDs.

A simpler implementation of the varianter uses serial integers as Variant IDs. A more sophisticated implementation could generate Variant IDs with more semantic, potentially representing their contents.

### Test ID

A test ID is a string that uniquely identifies a test in the context of a job. When considering a single job, there are no two tests with the same ID.

A test ID should encapsulate the Test Name and the Variant ID, to allow direct identification of a test. In other words, by looking at the test ID it should be possible to identify:

- What's the test name
- What's the variant used to run this test (if any)

Test IDs don't necessarily keep their uniqueness properties when considered outside of a particular job, but two identical jobs run in the exact same environment should generate a identical sets of Test IDs.

Syntax:

```
<unique-id>-<test-name>[;<variant-id>]
```

Examples of test-names:

```
'/bin/true'
'/bin/grep foobar /etc/passwd'
'passtest.py:Passtest.test'
'file:///tmp/passtest.py:Passtest.test'
'multiple_tests.py:MultipleTests.test_hello'
'type_specific.io-github-autotest-qemu.systemtap_tracing.qemu.qemu_free'
```

## Test Types

Avocado at its simplest configuration can run two different types of tests[1]. You can mix and match those in a single job.

### Instrumented

These are tests written in Python or BASH with the Avocado helpers that use the Avocado test API.

To be more precise, the Python file must contain a class derived from `avocado.test.Test`. This means that an executable written in Python is not always an instrumented test, but may work as a simple test.

The instrumented tests allows the writer finer control over the process including logging, test result status and other more sophisticated test APIs.

Test statuses `PASS`, `WARN`, `START` and `SKIP` are considered as successful builds. The `ABORT`, `ERROR`, `FAIL`, `ALERT`, `RUNNING`, `NOSTATUS` and `INTERRUPTED` are considered as failed ones.

### Simple

Any executable in your box. The criteria for PASS/FAIL is the return code of the executable. If it returns 0, the test PASSes, if it returns anything else, it FAILs.

## Test Statuses

Avocado sticks to the following definitions of test statuses:

- `` `PASS` ``: The test passed, which means all conditions being tested have passed.
- `` `FAIL` ``: The test failed, which means at least one condition being tested has failed. Ideally, it should mean a problem in the software being tested has been found.
- `` `ERROR` ``: An error happened during the test execution. This can happen, for example, if there's a bug in the test runner, in its libraries or if a resource breaks unexpectedly. Uncaught exceptions in the test code will also result in this status.
- `` `SKIP` ``: The test runner decided a requested test should not be run. This can happen, for example, due to missing requirements in the test environment or when there's a job timeout.

## Libraries and APIs

The Avocado libraries and its APIs are a big part of what Avocado is.

But, to avoid having any issues you should understand what parts of the Avocado libraries are intended for test writers and their respective API stability promises.

### Test APIs

At the most basic level there's the Test APIs which you should use when writing tests in Python and planning to make use of any other utility library.

[1] Avocado plugins can introduce additional test types.

The Test APIs can be found in the `avocado` main module, and its most important member is the `avocado.Test` class. By conforming to the `avocado.Test` API, that is, by inheriting from it, you can use the full set of utility libraries.

The Test APIs are guaranteed to be stable across a single major version of Avocado. That means that a test written for a given version of Avocado should not break on later minor versions because of Test API changes.

### Utility Libraries

There are also a large number of utility libraries that can be found under the `avocado.utils` namespace. These are very general in nature and can help you speed up your test development.

The utility libraries may receive incompatible changes across minor versions, but these will be done in a staged fashion. If a given change to an utility library can cause test breakage, it will first be documented and/or deprecated, and only on the next subsequent minor version it will actually be changed.

What this means is that upon updating to later minor versions of Avocado, you should look at the Avocado Release Notes for changes that may impact your tests.

### Core (Application) Libraries

Finally, everything under `avocado.core` is part of the application's infrastructure and should not be used by tests.

Extensions and Plugins can use the core libraries, but API stability is not guaranteed at any level.

## Test Resolution

When you use the Avocado runner, frequently you'll provide paths to files, that will be inspected, and acted upon depending on their contents. The diagram below shows how Avocado analyzes a file and decides what to do with it:

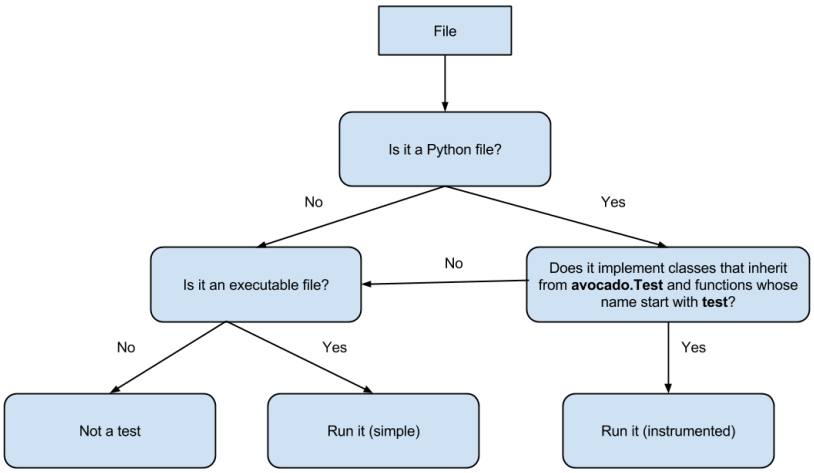
It's important to note that the inspection mechanism is safe (that is, python classes and files are not actually loaded and executed on discovery and inspection stage). Due to the fact Avocado doesn't actually load the code and classes, the introspection is simple and will *not* catch things like buggy test modules, missing imports and miscellaneous bugs in the code you want to list or run. We recommend only running tests from sources you trust, use of static checking and reviews in your test development process.

Due to the simple test inspection mechanism, avocado will not recognize test classes that inherit from a class derived from `avocado.Test`. Please refer to the *Writing Avocado Tests* documentation on how to use the tags functionality to mark derived classes as avocado test classes.

## Results Specification

On a machine that executed tests, job results are available under `[job-results]/job-[timestamp]-[short job ID]`, where `logdir` is the configured Avocado logs directory (see the data dir plugin), and the directory name includes a timestamp, such as `job-2014-08-12T15.44-565e8de`. A typical results directory structure can be seen below

```
$HOME/avocado/job-results/job-2014-08-13T00.45-4a92bc0/
- id
- jobdata
|   - args
|   - cmdline
|   - config
|   - multiplex
```




```
|   - pwd
|   - test_references
- job.log
- results.json
- results.xml
- sysinfo
|   - post
|   |   - brctl_show
|   |   - cmdline
|   |   - cpuinfo
|   |   - current_clocksource
|   |   - df_-mP
|   |   - dmesg_-c
|   |   - dmidecode
|   |   - fdisk_-l
|   |   - gcc_--version
|   |   - hostname
|   |   - ifconfig_-a
|   |   - interrupts
|   |   - ip_link
|   |   - ld_--version
|   |   - lscpu
|   |   - lspci_-vvnn
|   |   - meminfo
|   |   - modules
|   |   - mount
|   |   - mounts
```

```
|   |   - numactl_--hardware_show
|   |   - partitions
|   |   - scaling_governor
|   |   - uname_-a
|   |   - uptime
|   |   - version
|   - pre
|   |   - brctl_show
|   |   - cmdline
|   |   - cpuinfo
|   |   - current_clocksource
|   |   - df_-mP
|   |   - dmesg_-c
|   |   - dmidecode
|   |   - fdisk_-l
|   |   - gcc_--version
|   |   - hostname
|   |   - ifconfig_-a
|   |   - interrupts
|   |   - ip_link
|   |   - ld_--version
|   |   - lscpu
|   |   - lspci_-vvnn
|   |   - meminfo
|   |   - modules
|   |   - mount
|   |   - mounts
|   |   - numactl_--hardware_show
|   |   - partitions
|   |   - scaling_governor
|   |   - uname_-a
|   |   - uptime
|   |   - version
|   - profile
- test-results
    - tests
        - sleeptest.py.1
        |   - data
        |   - debug.log
        |   - sysinfo
        |       - post
        |       - pre
        - sleeptest.py.2
        |   - data
        |   - debug.log
        |   - sysinfo
        |       - post
        |       - pre
        - sleeptest.py.3
            - data
            - debug.log
            - sysinfo
                - post
                - pre

22 directories, 65 files
```

From what you can see, the results dir has:

1. A human readable `id` in the top level, with the job SHA1.
2. A human readable `job.log` in the top level, with human readable logs of the task
3. Subdirectory `jobdata`, that contains machine readable data about the job.
4. A machine readable `results.xml` and `results.json` in the top level, with a summary of the job information in xUnit/json format.
5. A top level `sysinfo` dir, with sub directories `pre`, `post` and `profile`, that store sysinfo files pre/post/during job, respectively.
6. Subdirectory `test-results`, that contains a number of subdirectories (filesystem-friendly test ids). Those test ids represent instances of test execution results.

#### Test execution instances specification

The instances should have:

1. A top level human readable `job.log`, with job debug information
2. A `sysinfo` subdir, with sub directories `pre` and `post`, that store sysinfo files pre test and post test, respectively.
3. A `data` subdir, where the test can output a number of files if necessary.

## Pre and post tests plugins

Avocado supports plug-ins which are (guaranteed to be) executed before the first test and after all tests finished. The interfaces are *avocado.core.plugin_interfaces.JobPre*, resp. *avocado.core.plugin_interfaces.JobPost*.

Note the `pre_tests` might not be executed due to earlier failure which prevents the tests from being executed.

The same applies for `post_tests`, but it is possible to have `post_tests` executed even when `pre_tests` were not. Additionally the `post_tests` are (obviously) not executed on `SIGKILL`. On the other hand they are executed on `SIGTERM` and `KeyboardInterrupt` while running the tests, but once the `post_tests` are executed the `KeyboardInterrupt` or `SystemExit` interrupts their processing (to avoid hangs) and remaining plug-ins will **NOT** be executed.

#### Jobscripts plugin

Avocado ships with a plugin (installed by default) that allows running scripts before and after the actual execution of Jobs. A user can be sure that, when a given "pre" script is run, no test in that job has been run, and when the "post" scripts are run, all the tests in a given job have already finished running.

#### Configuration

By default, the script directory location is:

```
/etc/avocado/scripts/job
```

Inside that directory, that is a directory for pre-job scripts:

```
/etc/avocado/scripts/job/pre.d
```

And for post-job scripts:

```
/etc/avocado/scripts/job/post.d
```

All the configuration about the Pre/Post Job Scripts are placed under the `avocado.plugins.jobscripts` config section. To change the location for the pre-job scripts, your configuration should look something like this:

```
[plugins.jobscripts]
pre = /my/custom/directory/for/pre/job/scripts/
```

Accordingly, to change the location for the post-job scripts, your configuration should look something like this:

```
[plugins.jobscripts]
post = /my/custom/directory/for/post/scripts/
```

A couple of other configuration options are available under the same section:

- `warn_non_existing_dir`: gives warnings if the configured (or default) directory set for either pre or post scripts do not exist
- `warn_non_zero_status`: gives warnings if a given script (either pre or post) exits with non-zero status

### Script Execution Environment

All scripts are run in separate process with some environment variables set. These can be used in your scripts in any way you wish:

- `AVOCADO_JOB_UNIQUE_ID`: the unique *job-id*.
- `AVOCADO_JOB_STATUS`: the current status of the job.
- `AVOCADO_JOB_LOGDIR`: the filesystem location that holds the logs and various other files for a given job run.

Note: Even though these variables should all be set, it's a good practice for scripts to check if they're set before using their values. This may prevent unintended actions such as writing to the current working directory instead of to the `AVOCADO_JOB_LOGDIR` if this is not set.

Finally, any failures in the Pre/Post scripts will not alter the status of the corresponding jobs.

## Job Cleanup

It's possible to register a callback function that will be called when all the tests have finished running. This effectively allows for a test job to clean some state it may have left behind.

At the moment, this feature is not intended to be used by test writers, but it's seen as a feature for Avocado extensions to make use.

To register a callback function, your code should put a message in a very specific format in the "runner queue". The Avocado test runner code will understand that this message contains a (serialized) function that will be called once all tests finish running.

Example:

```
from avocado import Test

def my_cleanup(path_to_file):
   if os.path.exists(path_to_file):
      os.unlink(path_to_file)
```

```
class MyCustomTest(Test):
...
   cleanup_file = '/tmp/my-custom-state'
   self.runner_queue.put({"func_at_exit": self.my_cleanup,
                          "args": (cleanup_file),
                          "once": True})
...
```

This results in the `my_cleanup` function being called with positional argument `cleanup_file`.

Because `once` was set to `True`, only one unique combination of function, positional arguments and keyword arguments will be registered, not matter how many times they're attempted to be registered. For more information check *avocado.utils.data_structures.CallbackRegister.register()*.

### Docstring Directives Rules

Avocado INSTRUMENTED tests, those written in Python and using the *avocado.Test* API, can make use of special directives specified as docstrings.

To be considered valid, the docstring must match this pattern: *avocado.core.safeloader.DOCSTRING_DIRECTIVE_RE_RAW*.

An Avocado docstring directive has two parts:

1. The marker, which is the literal string `:avocado:`.
2. The content, a string that follows the marker, separated by at least one white space or tab.

The following is a list of rules that makes a docstring directive be a valid one:

- It should start with `:avocado:`, which is the docstring directive "marker"
- At least one whitespace or tab must follow the marker and preceed the docstring directive "content"
- The "content", which follows the marker and the space, must begin with an alphanumeric character, that is, characters within "a-z", "A-Z" or "0-9".
- After at least one alphanumeric character, the content may contain the following special symbols too: `_`, `,`, `=` and `:`.
- An end of string (or end of line) must immediately follow the content.

## Contribution and Community Guide

Useful pointers on how to participate of the Avocado community and contribute.

### Hacking and Using Avocado

Since version 0.31.0, our plugin system requires Setuptools entry points to be registered. If you're hacking on Avocado and want to use the same, possibly modified, source for running your tests and experiments, you may do so with one additional step:

```
$ make develop
```

On POSIX systems this will create an "egg link" to your original source tree under "$HOME/.local/lib/pythonX.Y/site-packages". Then, on your original source tree, an "egg info" directory will be created, containing, among other things, the Setuptools entry points mentioned before. This works like a symlink, so you only need to run this once (unless you add a new entry-point, then you need to re-run it to make it available).

Avocado supports various plugins, which are distributed as separate projects, for example "avocado-vt" and "avocado-virt". These also need to be deployed and linked in order to work properly with the avocado from sources (installed version works out of the box). To simplify this you can use *make requirements-plugins* from the main avocado project to install requirements of the plugins and *make link* to link and develop the plugins. The workflow could be:

```
$ cd $AVOCADO_PROJECTS_DIR
$ git clone $AVOCADO_GIT
$ git clone $AVOCADO_PROJECT2
$ # Add more projects
$ cd avocado    # go into the main avocado project dir
$ make requirements-plugins
$ make link
```

You should see the process and status for each directory.

## Contact information

- Avocado-devel mailing list: https://www.redhat.com/mailman/listinfo/avocado-devel
- Avocado IRC channel: irc.oftc.net #avocado

## Contributing to Avocado

Avocado uses github and the github pull request development model. You can find a primer on how to use github pull requests here. Every Pull Request you send will be automatically tested by Travis CI and review will take place in the Pull Request as well.

For people who don't like the github development model, there is the option of sending the patches to the Mailing List, following a workflow more traditional in Open Source development communities. The patches will be reviewed in the Mailing List, should you opt for that. Then a maintainer will collect the patches, integrate them on a branch, and then those patches will be submitted as a github Pull Request. This process tries to ensure that every contributed patch goes through the CI jobs before it is considered good for inclusion.

### Git workflow

- Fork the repository in github.
- Clone from your fork:

```
$ git clone git@github.com:<username>/avocado.git
```

- Enter the directory:

```
$ cd avocado
```

- Create a `remote`, pointing to the upstream:

```
$ git remote add upstream git@github.com:avocado-framework/avocado.git
```

- Configure your name and e-mail in git:

```
$ git config --global user.name "Your Name"
$ git config --global user.email email@foo.bar
```

- Golden tip: never work on local branch master. Instead, create a new local branch and checkout to it:

```
$ git checkout -b my_new_local_branch
```

- Code and then commit your changes:

```
$ git add new-file.py
$ git commit -s
# or "git commit -as" to commit all changes
```

- Write a good commit message, pointing motivation, issues that you're addressing. Usually you should try to explain 3 points in the commit message: motivation, approach and effects:

```
header          <- Limited to 72 characters. No period.
                <- Blank line
message         <- Any number of lines, limited to 72 characters per line.
                <- Blank line
Reference:      <- External references, one per line (issue, trello, ...)
Signed-off-by:  <- Signature and acknowledgment of licensing terms when
                   contributing to the project (created by git commit -s)
```

- Make sure your code is working (install your version of avocado, test your change, run `make check` to make sure you didn't introduce any regressions).
- Paste the `job.log` file content from the previous step in a pastebin service, like fpaste.org. If you have `fpaste` installed, you can simply run:

```
$ fpaste ~/avocado/job-results/latest/job.log
```

- Rebase your local branch on top of upstream master:

```
$ git fetch
$ git rebase upstream/master
(resolve merge conflicts, if any)
```

- Push your commit(s) to your fork:

```
$ git push origin my_new_local_branch
```

- Create the Pull Request on github. Add the relevant information to the Pull Request description.
- In the Pull Request discussion page, comment with the link to the job.log output/file.
- Check if your Pull Request passes the CI (travis). Your Pull Request will probably be ignored until it's all green.

Now you're waiting for feedback on github Pull Request page. Once you get some, join the discussion, answer the questions, make clear if you're going to change the code based on some review and, if not, why. Feel free to disagree with the reviewer, they probably have different use cases and opinions, which is expected. Try describing yours and suggest other solutions, if necessary.

New versions of your code should not be force-updated (unless explicitly requested by the code reviewer). Instead, you should:

- Create a new branch out of your previous branch:

```
$ git checkout my_new_local_branch
$ git checkout -b my_new_local_branch_v2
```

- Code, and amend the commit(s) and/or create new commits. If you have more than one commit in the PR, you will probably need to rebase interactively to amend the right commits. `git cola` or `git citool` can be handy here.
- Rebase your local branch on top of upstream master:

```
$ git fetch
$ git rebase upstream/master
(resolve merge conflicts, if any)
```

- Push your changes:

```
$ git push origin my_new_local_branch_v2
```

- Create a new Pull Request for this new branch. In the Pull Request description, point the previous Pull Request and the changes the current Pull Request introduced when compared to the previous Pull Request(s).
- Close the previous Pull Request on github.

After your PR gets merged, you can sync the master branch on your local repository propagate the sync to the master branch in your fork repository on github:

```
$ git checkout master
$ git pull upstream master
$ git push
```

From time to time, you can remove old branches to avoid pollution:

```
# To list branches along with time reference:
$ git for-each-ref --sort='-authordate:iso8601' --format=' %(authordate:iso8601)%09
↪%(refname)' refs/heads
# To remove branches from your fork repository:
$ git push origin :my_old_branch
```

### Signing commits

Optionally you can sign the commits using GPG signatures. Doing it is simple and it helps from unauthorized code being merged without notice.

All you need is a valid GPG signature, git configuration, slightly modified workflow to use the signature and eventually even setup in github so one benefits from the "nice" UI.

Get a GPG signature:

```
# Google for howto, but generally it works like this
$ gpg --gen-key  # defaults are usually fine (using expiration is recommended)
$ gpg --send-keys $YOUR_KEY    # to propagate the key to outer world
```

Enable it in git:

```
$ git config --global user.signingkey $YOUR_KEY
```

(optional) Link the key with your GH account:

```
1. Login to github
2. Go to settings->SSH and GPG keys
3. Add New GPG key
4. run $(gpg -a --export $YOUR_EMAIL) in shell to see your key
5. paste the key there
```

Use it:

```
# You can sign commits by using '-S'
$ git commit -S
# You can sign merges by using '-S'
$ git merge -S
```

**Warning:** You can not use the merge button on github to do signed merges as github does not have your private key.

#### Licensing

Except where otherwise indicated in a given source file, all original contributions to Avocado are licensed under the GNU General Public License version 2 (GPLv2) or any later version.

By contributing you agree that these contributions are your own (or approved by your employer) and you grant a full, complete, irrevocable copyright license to all users and developers of the Avocado project, present and future, pursuant to the license of the project.

### Tests Repositories

We encourage you or your company to create public Avocado tests repositories so the community can also benefit of your tests. We will be pleased to advertise your repository here in our documentation.

List of known community and third party maintained repositories:

- https://github.com/avocado-framework-tests/avocado-misc-tests: Community maintained Avocado miscellaneous tests repository. There you will find, among others, performance tests like `lmbench`, `stress`, cpu tests like `ebizzy` and generic tests like `ltp`. Some of them were ported from Autotest Client Tests repository.
- https://github.com/scylladb/scylla-cluster-tests: Avocado tests for Scylla Clusters. Those tests can automatically create a scylla cluster, some loader machines and then run operations defined by the test writers, such as database workloads.

## Avocado development tips

### Interrupting test

In case you want to "pause" the running test, you can use SIGTSTP (ctrl+z) signal sent to the main avocado process. This signal is forwarded to test and it's children processes. To resume testing you repeat the same signal.

Note: that the job/test timeouts are still enabled on stopped processes.

## In tree utils

You can find handy utils in *avocado.utils.debug*:

### measure_duration

Decorator can be used to print current duration of the executed function and accumulated duration of this decorated function. It's very handy when optimizing.

Usage:

```
from avocado.utils import debug
...
@debug.measure_duration
def your_function(...):
```

During the execution look for:

```
PERF: <function your_function at 0x29b17d0>: (0.1s, 11.3s)
PERF: <function your_function at 0x29b17d0>: (0.2s, 11.5s)
```

## Line-profiler

You can measure line-by-line performance by using line_profiler. You can install it using pip:

```
pip install line_profiler
```

and then simply mark the desired function with *@profile* (no need to import it from anywhere). Then you execute:

```
kernprof -l -v ./scripts/avocado run ...
```

and when the process finishes you'll see the profiling information. (sometimes the binary is called *kernprof.py*)

## Remote debug with Eclipse

Eclipse is a nice debugging frontend which allows remote debugging. It's very simple. The only thing you need is Eclipse with pydev plugin. The simplest way is to use `pip install pydevd` and then you set the breakpoint by:

```
import pydevd
pydevd.settrace(host="$IP_ADDR_OF_ECLIPSE_MACHINE", stdoutToServer=False,␣
↪stderrToServer=False, port=5678, suspend=True, trace_only_current_thread=False,␣
↪overwrite_prev_trace=False, patch_multiprocessing=False)
```

Before you run the code, you need to start the Eclipse's debug server. Switch to *Debug* perspective (you might need to open it first *Window->Perspective->Open Perspective*). Then start the server from *Pydev->Start Debug Server*.

Now whenever the pydev.settrace() code is executed, it contacts Eclipse debug server (port *8000* by default, don't forget to open it) and you can easily continue in execution. This works on every remote machine which has access to your Eclipse's port *8000* (you can override it).

## Using Trello cards in Eclipse

Eclipse allows us to create tasks. They are pretty cool as you see the status (not started, started, current, done) and by switching tasks it automatically resumes where you previously finished (opened files, ...)

Avocado is planned using Trello, which is not yet supported by Eclipse. Anyway there is a way to at least get read-only list of your commits. This guide is based on https://docs.google.com/document/d/1jvmJcCStE6QkJ0z5ASddc3fNmJwhJPOFN7X9-GLyabM/ which didn't work well with lables and descriptions. The only difference is you need to use *Query Pattern*:

```
\"url\":\"https://trello.com/[^/]*/[^/]*/({Id}[^\"]+)({Description})\"
```

Setup Trello key:

1. Create a Trello account
2. Get (developer_key) here: https://trello.com/1/appKey/generate
3. Get user_token from following address (replace key with your key): https://trello.com/1/authorize?key=\protect\T1\textdollardeveloper_key&name=Mylyn%20Tasks&expiration=never&response_type=token
4. Address with your assigned tasks (task_addr) is: https://trello.com/1/members/my/cards?key=developer_key&token=\protect\T1\textdollaruser_token Open it in web browser and you should see *[]* or *[$list_of_cards]* without any passwords.

Configure Eclipse:

1. We're going to need Web Templates, which are not yet upstream. We need to use incubator version.
2. *Help->Install New Software...*
3. *-> Add*
4. Name: *Incubator*
5. Location: http://download.eclipse.org/mylyn/incubator/3.10
6. *-> OK*
7. Select *Mylyn Tasks Connector: Web Templates (Advanced) (Incubation)* (use filter text to find it)
8. Install it (*Next->Agree->Next...*)
9. Restart Eclipse
10. Open the Mylyn Team Repositories *Window->Show View->Other...->Mylyn->Team Repositories*
11. Right click the *Team Repositories* and select *New->Repository*
12. Use *Task Repository -> Next*
13. Use *Web Template (Advanced) -> Next*
14. In the Properties for Task Repository dialog box, enter https://trello.com
15. In the Server field and give the repository a label (eg. *Trello API*).
16. In the Additional Settings section set *applicationkey = $developer_key* and *userkey = $user_token*.
17. In the Advanced Configuration set the Task URL to https://trello.com/c/
18. Set New Task URL to https://trello.com
19. Set the Query Request URL (no changes required): https://trello.com/1/members/my/cards?key=\protect\T1\textdollar\protect\T1\textbraceleftapplicationkey\protect\T1\textbraceright&token=\protect\T1\textdollar\protect\T1\textbraceleftuserkey\protect\T1\textbraceright

20. For the Query Pattern enter *"url":"https://trello.com/[^/]*/[^/]*/({Id}[^"]+)({Description})"*
21. -> *Finish*

Create task query:

1. Create a query by opening the *Mylyn Task List*.
2. Right click the pane and select *New Query*.
3. Select Trello API as the repository.
4. -> *Next*
5. Enter the name of your query.
6. Expand the Advanced Configuration and make sure the Query Pattern is filled in
7. Press *Preview* to confirm that there are no errors.
8. Press *Finish*.
9. Trello tasks assigned to you will now appear in the Mylyn Task List.

Note you can start using tasks by clicking the small bubble in front of the name. This closes all editors. Try opening some and then click the bubble again. They should get closed. When you click the bubble third time, it should resume all the open editors from before.

My usual workflow is:

1. git checkout $branch
2. Eclipse: select task
3. git commit ...
4. Eclipse: unselect task
5. git checkout $other_branch
6. Eclipse: select another_task

This way you always have all the files present and you can easily resume your work.

# Releasing avocado

So you have all PRs approved, the Sprint meeting is done and now Avocado is ready to be released. Great, let's go over (most of) the details you need to pay attention to.

## Bump the version number

For the Avocado versioning, two files need to receive a manual version update:

- `VERSION`
- `python-avocado.spec`

followed by `make propagate-version` to propagate this change to all optional and "linkabe" plugins sharing the parent dir (eg. `avocado-vt`). Don't forget to commit the changes of "linked" plugins as they live in different repositories.

An example diff (after the `make propagate-version`) looks like this:

```
diff --git a/VERSION b/VERSION
index dd0353d..aafccd8 100644
--- a/VERSION
+++ b/VERSION
@@ -1 +1 @@
-48.0
+49.0
diff --git a/optional_plugins/html/VERSION b/optional_plugins/html/VERSION
index dd0353d..aafccd8 100644
--- a/optional_plugins/html/VERSION
+++ b/optional_plugins/html/VERSION
@@ -1 +1 @@
-48.0
+49.0
diff --git a/optional_plugins/robot/VERSION b/optional_plugins/robot/VERSION
index dd0353d..aafccd8 100644
--- a/optional_plugins/robot/VERSION
+++ b/optional_plugins/robot/VERSION
@@ -1 +1 @@
-48.0
+49.0
diff --git a/optional_plugins/runner_docker/VERSION b/optional_plugins/runner_docker/
↪VERSION
index dd0353d..aafccd8 100644
--- a/optional_plugins/runner_docker/VERSION
+++ b/optional_plugins/runner_docker/VERSION
@@ -1 +1 @@
-48.0
+49.0
diff --git a/optional_plugins/runner_remote/VERSION b/optional_plugins/runner_remote/
↪VERSION
index dd0353d..aafccd8 100644
--- a/optional_plugins/runner_remote/VERSION
+++ b/optional_plugins/runner_remote/VERSION
@@ -1 +1 @@
-48.0
+49.0
diff --git a/optional_plugins/runner_vm/VERSION b/optional_plugins/runner_vm/VERSION
index dd0353d..aafccd8 100644
--- a/optional_plugins/runner_vm/VERSION
+++ b/optional_plugins/runner_vm/VERSION
@@ -1 +1 @@
-48.0
+49.0
diff --git a/python-avocado.spec b/python-avocado.spec
index 6a4b067..4b9dba8 100644
--- a/python-avocado.spec
+++ b/python-avocado.spec
@@ -12,7 +12,7 @@

 Summary: Framework with tools and libraries for Automated Testing
 Name: python-%{srcname}
-Version: 48.0
+Version: 49.0
 Release: 1%{?dist}
 License: GPLv2
 Group: Development/Tools
@@ -259,6 +259,9 @@ examples of how to write tests on your own.
```

```
 %{_datadir}/avocado/wrappers

 %changelog
+* Wed Apr 12 2017 Lukas Doktor <ldoktor@redhat.com> - 49.0-0
+- Testing release
+
 * Mon Apr  3 2017 Cleber Rosa <cleber@localhost.localdomain> - 48.0-1
 - Updated exclude directives and files for optional plugins
```

You can find on git such commits that will help you get oriented for other repos.

## Which repositories you should pay attention to

In general, a release of avocado includes taking a look and eventually release content in the following repositories:

- `avocado`
- `avocado-vt`

## Tag all repositories

When everything is in good shape, commit the version changes and tag that commit in master with:

```
$ git tag -u $(GPG_ID) -s $(RELEASE) -m 'Avocado Release $(RELEASE)'
```

Then the tag should be pushed to the GIT repository with:

```
$ git push --tags
```

## Build RPMs

Go to the source directory and do:

```
$ make rpm
...
+ exit 0
```

This should be all. It will build packages using `mock`, targeting your default configuration. That usually means the same platform you're currently on.

## Sign Packages

All the packages should be signed for safer public consumption. The process is, of course, dependent on the private keys, put is based on running:

```
$ rpm --resign
```

For more information look at the `rpmsign(8)` man page.

### Upload packages to repository

The current distribution method is based on serving content over HTTP. That means that repository metadata is created locally and synchronized to the well know public Web server. A process similar to:

```
$ cd $REPO_ROOT && for DIR in epel-?-noarch fedora-??-noarch; \
do cd $DIR && createrepo -v . && cd ..; done;
```

Creates the repo metadata locally. Then a command similar to:

```
$ rsync -va $REPO_ROOT user@repo_web_server:/path
```

Is used to copy the content over.

### Write release notes

Release notes give an idea of what has changed on a given development cycle. Good places to go for release notes are:

1. Git logs
2. Trello Cards (Look for the Done lists)
3. Github compare views: https://github.com/avocado-framework/avocado/compare/0.28.0...0.29.0

Go there and try to write a text that represents the changes that the release encompasses.

### Upload package to PyPI

Users may also want to get Avocado from the PyPI repository, so please upload there as well. To help with the process, please run:

```
$ make pypi
```

And follow the URL and brief instructions given.

### Configure Read The Docs

On https://readthedocs.org/dashboard/avocado-framework/edit/:

- Click in **Versions**. In **Choose Active Versions**, find the version you're releasing and check the **Active** option. **Submit**.
- Click in **Versions** again. In **Default Version**, select the new version you're releasing. **Submit**.

### Send e-mails to avocado-devel and other places

Send the e-mail with the release notes to avocado-devel and virt-test-devel.

## Other Resources

This is a collection of some other varied Avocado related sources on the web:

- Mindmap from 2015 workshop demonstrating features and examples are available here.

CHAPTER 18

# API Reference

## Test APIs

This is the bare mininum set of APIs that users should use, and can rely on, while writing tests.

### Module contents

avocado.**main**
alias of TestProgram

**class** avocado.**Test**(*methodName='test'*, *name=None*, *params=None*, *base_logdir=None*, *job=None*, *runner_queue=None*)
Bases: unittest.case.TestCase

Base implementation for the test class.

You'll inherit from this to write your own tests. Typically you'll want to implement setUp(), test*() and tearDown() methods on your own tests.

Initializes the test.

**Parameters**

- **methodName** – Name of the main method to run. For the sake of compatibility with the original unittest class, you should not set this.
- **name** (*avocado.core.test.TestName*) – Pretty name of the test name. For normal tests, written with the avocado API, this should not be set. This is reserved for internal Avocado use, such as when running random executables as tests.
- **base_logdir** – Directory where test logs should go. If None provided, it'll use avocado.data_dir.create_job_logs_dir().
- **job** – The job that this test is part of.

**Raises** *avocado.core.test.NameNotTestNameError*

**basedir**
The directory where this test (when backed by a file) is located at

**cache_dirs = None**

**cancel**(*message=None*)
Cancels the test.

This method is expected to be called from the test method, not anywhere else, since by definition, we can only cancel a test that is currently under execution. If you call this method outside the test method, avocado will mark your test status as ERROR, and instruct you to fix your test in the error message.

**Parameters** **message** (*str*) – an optional message that will be recorded in the logs

**datadir**
Returns the path to the directory that contains test data files

**default_params = {}**

**error**(*message=None*)
Errors the currently running test.

After calling this method a test will be terminated and have its status as ERROR.

**Parameters** **message** (*str*) – an optional message that will be recorded in the logs

**fail**(*message=None*)
Fails the currently running test.

After calling this method a test will be terminated and have its status as FAIL.

**Parameters** **message** (*str*) – an optional message that will be recorded in the logs

**fail_class**

**fail_reason**

**fetch_asset**(*name*, *asset_hash=None*, *algorithm='sha1'*, *locations=None*, *expire=None*)
Method o call the utils.asset in order to fetch and asset file supporting hash check, caching and multiple locations.

**Parameters**

- **name** – the asset filename or URL
- **asset_hash** – asset hash (optional)
- **algorithm** – hash algorithm (optional, defaults to sha1)
- **locations** – list of URLs from where the asset can be fetched (optional)
- **expire** – time for the asset to expire

**Raises** **EnvironmentError** – When it fails to fetch the asset

**Returns** asset file local path

**filename**
Returns the name of the file (path) that holds the current test

**get_state**()
Serialize selected attributes representing the test state

**Returns** a dictionary containing relevant test state data

**Return type** dict

**job**
The job this test is associated with

**log**
The enhanced test log

**logdir**
Path to this test's logging dir

**logfile**
Path to this test's main *debug.log* file

**name**
The test name (TestName instance)

**outputdir**
Directory available to test writers to attach files to the results

**params**
Parameters of this test (AvocadoParam instance)

**report_state**()
Send the current test state to the test runner process

**run_avocado**()
Wraps the run method, for execution inside the avocado runner.

**Result** Unused param, compatibility with `unittest.TestCase`.

**runner_queue**
The communication channel between test and test runner

**running**
Whether this test is currently being executed

**set_runner_queue**(*runner_queue*)
Override the runner_queue

**skip**(*message=None*)
Skips the currently running test.

This method should only be called from a test's setUp() method, not anywhere else, since by definition, if a test gets to be executed, it can't be skipped anymore. If you call this method outside setUp(), avocado will mark your test status as ERROR, and instruct you to fix your test in the error message.

**Parameters** **message** (*str*) – an optional message that will be recorded in the logs

**srcdir = None**

**status**
The result status of this test

**teststmpdir**
Returns the path of the temporary directory that will stay the same for all tests in a given Job.

**time_elapsed = -1**

**time_end = -1**

**time_start = -1**

**timeout = None**

**traceback**

**whiteboard = ''**

**workdir = None**

avocado.**fail_on**(*exceptions=None*)
Fail the test when decorated function produces exception of the specified type.

(For example, our method may raise IndexError on tested software failure. We can either try/catch it or use this decorator instead)

**Parameters** **exceptions** – Tuple or single exception to be assumed as test fail [Exception]

**Note** self.error and self.skip behavior remains intact

**Note** To allow simple usage param "exceptions" must not be callable

avocado.**skip**(*message=None*)
Decorator to skip a test.

avocado.**skipIf**(*condition*, *message=None*)
Decorator to skip a test if a condition is True.

avocado.**skipUnless**(*condition*, *message=None*)
Decorator to skip a test if a condition is False.

**exception** avocado.**TestError**
Bases: *avocado.core.exceptions.TestBaseException*

Indicates that the test was not fully executed and an error happened.

This is the sort of exception you raise if the test was partially executed and could not complete due to a setup, configuration, or another fatal condition.

**status = 'ERROR'**

**exception** avocado.**TestFail**
Bases: *avocado.core.exceptions.TestBaseException*, exceptions.AssertionError

Indicates that the test failed.

TestFail inherits from AssertionError in order to keep compatibility with vanilla python unittests (they only consider failures the ones deriving from AssertionError).

**status = 'FAIL'**

**exception** avocado.**TestCancel**
Bases: *avocado.core.exceptions.TestBaseException*

Indicates that a test was canceled.

Should be thrown when the cancel() test method is used.

**status = 'CANCEL'**

## Utilities APIs

This is a set of utility APIs that Avocado provides as added value to test writers.

It's suppose to be generic, without any knowledge of Avocado and reusable in different projects.

**Note:** In the current version there is a hidden knowledge of avocado logging streams. More about this issue can be found here https://trello.com/c/4QyUgWsW/720-get-rid-of-avocado-test-loggers-from-avocado-utils

## Subpackages

### avocado.utils.external package

#### Submodules

#### avocado.utils.external.gdbmi_parser module

avocado.utils.external.gdbmi_parser.**parse**(*tokens*)

avocado.utils.external.gdbmi_parser.**process**(*input*)

avocado.utils.external.gdbmi_parser.**scan**(*input*)

#### avocado.utils.external.spark module

**class** avocado.utils.external.spark.**GenericASTBuilder**(*AST*, *start*)
Bases: *avocado.utils.external.spark.GenericParser*

**buildASTNode**(*args*, *lhs*)

**nonterminal**(*type*, *args*)

**preprocess**(*rule*, *func*)

**terminal**(*token*)

**class** avocado.utils.external.spark.**GenericASTMatcher**(*start*, *ast*)
Bases: *avocado.utils.external.spark.GenericParser*

**foundMatch**(*args*, *func*)

**match**(*ast=None*)

**match_r**(*node*)

**preprocess**(*rule*, *func*)

**resolve**(*list*)

**class** avocado.utils.external.spark.**GenericASTTraversal**(*ast*)

**default**(*node*)

**postorder**(*node=None*)

**preorder**(*node=None*)

**prune**()

**typestring**(*node*)

**exception** avocado.utils.external.spark.**GenericASTTraversalPruningException**
Bases: exceptions.Exception

**class** avocado.utils.external.spark.**GenericParser**(*start*)

**add**(*set*, *item*, *i=None*, *predecessor=None*, *causal=None*)

**addRule**(*doc*, *func*, *_preprocess=1*)

**ambiguity**(*rules*)

**augment** (*start*)

**buildTree** (*nt*, *item*, *tokens*, *k*)

**causal** (*key*)

**collectRules** ()

**computeNull** ()

**deriveEpsilon** (*nt*)

**error** (*token*)

**finalState** (*tokens*)

**goto** (*state*, *sym*)

**gotoST** (*state*, *st*)

**gotoT** (*state*, *t*)

**isnullable** (*sym*)

**makeNewRules** ()

**makeSet** (*token*, *sets*, *i*)

**makeSet_fast** (*token*, *sets*, *i*)

**makeState** (*state*, *sym*)

**makeState0** ()

**parse** (*tokens*)

**predecessor** (*key*, *causal*)

**preprocess** (*rule*, *func*)

**resolve** (*list*)

**skip** (*lhs_rhs*, *pos=0*)

**typestring** (*token*)

**class** avocado.utils.external.spark.**GenericScanner** (*flags=0*)

**error** (*s*, *pos*)

**makeRE** (*name*)

**position** (*newpos=None*)

**reflect** ()

**t_default** (*s*)
( . | n )+

**tokenize** (*s*)

### Module contents

## Submodules

## avocado.utils.archive module

Module to help extract and create compressed archives.

**exception** avocado.utils.archive.**ArchiveException**
Bases: exceptions.Exception

Base exception for all archive errors.

**class** avocado.utils.archive.**ArchiveFile**(*filename*, *mode='r'*)
Bases: object

Class that represents an Archive file.

Archives are ZIP files or Tarballs.

Creates an instance of *ArchiveFile*.

**Parameters**

- **filename** – the archive file name.
- **mode** – file mode, *r* read, *w* write.

**add**(*filename*, *arcname=None*)
Add file to the archive.

**Parameters**

- **filename** – file to archive.
- **arcname** – alternative name for the file in the archive.

**close**()
Close archive.

**extract**(*path='.'*)
Extract all files from the archive.

**Parameters** **path** – destination path.

**list**()
List files to the standard output.

**classmethod open**(*filename*, *mode='r'*)
Creates an instance of *ArchiveFile*.

**Parameters**

- **filename** – the archive file name.
- **mode** – file mode, *r* read, *w* write.

avocado.utils.archive.**compress**(*filename*, *path*)
Compress files in an archive.

**Parameters**

- **filename** – archive file name.
- **path** – origin directory path to files to compress. No individual files allowed.

avocado.utils.archive.**create**(*filename*, *path*)
Compress files in an archive.

**Parameters**

- **filename** – archive file name.
- **path** – origin directory path to files to compress. No individual files allowed.

avocado.utils.archive.**extract**(*filename*, *path*)
Extract files from an archive.

**Parameters**

- **filename** – archive file name.
- **path** – destination path to extract to.

avocado.utils.archive.**is_archive**(*filename*)
Test if a given file is an archive.

**Parameters** **filename** – file to test.

**Returns** *True* if it is an archive.

avocado.utils.archive.**uncompress**(*filename*, *path*)
Extract files from an archive.

**Parameters**

- **filename** – archive file name.
- **path** – destination path to extract to.

## avocado.utils.asset module

Asset fetcher from multiple locations

**class** avocado.utils.asset.**Asset**(*name*, *asset_hash*, *algorithm*, *locations*, *cache_dirs*, *expire=None*)
Bases: object

Try to fetch/verify an asset file from multiple locations.

Initialize the Asset() class.

**Parameters**

- **name** – the asset filename. url is also supported
- **asset_hash** – asset hash
- **algorithm** – hash algorithm
- **locations** – list of locations fetch asset from
- **cache_dirs** – list of cache directories
- **expire** – time in seconds for the asset to expire

**fetch**()
Fetches the asset. First tries to find the asset on the provided cache_dirs list. Then tries to download the asset from the locations list provided.

**Raises** **EnvironmentError** – When it fails to fetch the asset

**Returns** The path for the file on the cache directory.

## avocado.utils.astring module

Operations with strings (conversion and sanitation).

The unusual name aims to avoid causing name clashes with the stdlib module string. Even with the dot notation, people may try to do things like

> import string ... from avocado.utils import string

And not notice until their code starts failing.

avocado.utils.astring.**bitlist_to_string**(*data*)
Transform from bit list to ASCII string.

**Parameters** **data** – Bit list to be transformed

avocado.utils.astring.**iter_tabular_output**(*matrix*, *header=None*)
Generator for 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.

avocado.utils.astring.**shell_escape**(*command*)
Escape special characters from a command so that it can be passed as a double quoted (" ") string in a (ba)sh command.

**Parameters** **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

avocado.utils.astring.**string_safe_encode**(*string*)
People tend to mix unicode streams with encoded strings. This function tries to replace any input with a valid utf-8 encoded ascii stream.

avocado.utils.astring.**string_to_bitlist**(*data*)
Transform from ASCII string to bit list.

**Parameters** **data** – String to be transformed

avocado.utils.astring.**string_to_safe_path**(*string*)
Convert string to a valid file/dir name. :param string: String to be converted :return: String which is safe to pass as a file/dir name (on recent fs)

avocado.utils.astring.**strip_console_codes**(*output*, *custom_codes=None*)
Remove the Linux console escape and control sequences from the console output. Make the output readable and can be used for result check. Now only remove some basic console codes using during boot up.

**Parameters**

- **output** (*string*) – The output from Linux console
- **custom_codes** – The codes added to the console codes which is not covered in the default codes

**Returns** the string without any special codes

**Return type** string

avocado.utils.astring.**tabular_output**(*matrix*, *header=None*)
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.

**Returns** String with the tabular output, lines separated by unix line feeds.

**Return type** str

## avocado.utils.aurl module

URL related functions.

The strange name is to avoid accidental naming collisions in code.

avocado.utils.aurl.**is_url**(*path*)
Return *True* if path looks like an URL.

**Parameters** **path** – path to check.

**Return type** Boolean.

## avocado.utils.build module

avocado.utils.build.**make**(*path*, *make='make'*, *env=None*, *extra_args=''*, *ignore_status=None*, *allow_output_check=None*, *process_kwargs=None*)
Run make, adding MAKEOPTS to the list of options.

**Parameters**

- **make** – what make command name to use.
- **env** – dictionary with environment variables to be set before calling make (e.g.: CFLAGS).
- **extra** – extra command line arguments to pass to make.
- **allow_output_check** (*str*) – Whether to log the command stream outputs (stdout and stderr) of the make process in the test stream files. Valid values: 'stdout', for allowing only standard output, 'stderr', to allow only standard error, 'all', to allow both standard output and error, and 'none', to allow none to be recorded (default). The default here is 'none', because usually we don't want to use the compilation output as a reference in tests.

**Returns** exit status of the make process

avocado.utils.build.**run_make**(*path*, *make='make'*, *env=None*, *extra_args=''*, *ignore_status=None*, *allow_output_check=None*, *process_kwargs=None*)
Run make, adding MAKEOPTS to the list of options.

**Parameters**

- **make** – what make command name to use.
- **env** – dictionary with environment variables to be set before calling make (e.g.: CFLAGS).
- **extra** – extra command line arguments to pass to make.

- **allow_output_check** (*str*) – Whether to log the command stream outputs (stdout and stderr) of the make process in the test stream files. Valid values: 'stdout', for allowing only standard output, 'stderr', to allow only standard error, 'all', to allow both standard output and error, and 'none', to allow none to be recorded (default). The default here is 'none', because usually we don't want to use the compilation output as a reference in tests.
- **process_kwargs** – Additional key word arguments to the underlying process running the make.

**Returns** the make command result object

## avocado.utils.cpu module

Get information from the current's machine CPU.

avocado.utils.cpu.**cpu_has_flags**(*flags*)
Check if a list of flags are available on current CPU info

**Parameters** **flags** (*list*) – A *list* of cpu flags that must exists on the current CPU.

**Returns** *bool* True if all the flags were found or False if not

**Return type** *list*

avocado.utils.cpu.**cpu_online_list**()
Reports a list of indexes of the online cpus

avocado.utils.cpu.**get_cpu_arch**()
Work out which CPU architecture we're running on

avocado.utils.cpu.**get_cpu_vendor_name**()
Get the current cpu vendor name

**Returns** string 'intel' or 'amd' or 'power7' depending on the current CPU architecture.

**Return type** *string*

avocado.utils.cpu.**online_cpus_count**()
Return Number of Online cpus in the system

avocado.utils.cpu.**total_cpus_count**()
Return Number of Total cpus in the system including offline cpus

## avocado.utils.crypto module

avocado.utils.crypto.**hash_file**(*filename*, *size=None*, *algorithm='md5'*)
Calculate the hash value 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
- **size** – If provided, hash only the first size bytes of the file.

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

avocado.utils.crypto.**hash_wrapper**(*algorithm='md5'*, *data=None*)
Returns an hash object of data using either md5 or sha1 only.

**Parameters** **input** – Optional input string that will be used to update the hash.

**Returns** Hash object.

## avocado.utils.data_factory module

Generate data useful for the avocado framework and tests themselves.

avocado.utils.data_factory.**generate_random_string**(*length*, *ignore='!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'*, *convert=''*)
Generate a random string using alphanumeric characters.

**Parameters**

- **length** (*int*) – 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.

avocado.utils.data_factory.**make_dir_and_populate**(*basedir='/tmp'*)
Create a directory in basedir and populate with a number of files.

The files just have random text contents.

**Parameters** **basedir** (*str*) – Base directory where directory should be generated.

**Returns** Path of the dir created and populated.

**Return type** str

## avocado.utils.data_structures module

This module contains handy classes that can be used inside avocado core code or plugins.

**class** avocado.utils.data_structures.**Borg**
Multiple instances of this class will share the same state.

This is considered a better design pattern in Python than more popular patterns, such as the Singleton. Inspired by Alex Martelli's article mentioned below:

**See** http://www.aleax.it/5ep.html

**class** avocado.utils.data_structures.**CallbackRegister**(*name*, *log*)
Bases: object

Registers pickable functions to be executed later.

**Parameters** **name** – Human readable identifier of this register

**register**(*func*, *args*, *kwargs*, *once=False*)
Register function/args to be called on self.destroy() :param func: Pickable function :param args: Pickable positional arguments :param kwargs: Pickable keyword arguments :param once: Add unique (func,args,kwargs) combination only once

**run**()
Call all registered function

**unregister**(*func*, *args*, *kwargs*)
Unregister (func,args,kwargs) combination :param func: Pickable function :param args: Pickable positional arguments :param kwargs: Pickable keyword arguments

**class** avocado.utils.data_structures.**LazyProperty**(*f_get*)
Bases: object

Lazily instantiated property.

Use this decorator when you want to set a property that will only be evaluated the first time it's accessed. Inspired by the discussion in the Stack Overflow thread below:

**See** http://stackoverflow.com/questions/15226721/

avocado.utils.data_structures.**compare_matrices**(*matrix1*, *matrix2*, *threshold=0.05*)
Compare 2 matrices nxm and return a matrix nxm with comparison data and stats. When the first columns match, they are considered as header and included in the results intact.

**Parameters**

- **matrix1** – Reference Matrix of floats; first column could be header.
- **matrix2** – Matrix that will be compared; first column could be header
- **threshold** – Any difference greater than this percent threshold will be reported.

**Returns** Matrix with the difference in comparison, number of improvements, number of regressions, total number of comparisons.

avocado.utils.data_structures.**geometric_mean**(*values*)
Evaluates the geometric mean for a list of numeric values. This implementation is slower but allows unlimited number of values. :param values: List with values. :return: Single value representing the geometric mean for the list values. :see: http://en.wikipedia.org/wiki/Geometric_mean

avocado.utils.data_structures.**ordered_list_unique**(*object_list*)
Returns an unique list of objects, with their original order preserved

avocado.utils.data_structures.**time_to_seconds**(*time*)
Convert time in minutes, hours and days to seconds. :param time: Time, optionally including the unit (i.e. '10d')

## avocado.utils.debug module

This file contains tools for (not only) Avocado developers.

avocado.utils.debug.**log_calls**(*length=None*, *cls_name=None*)
Use this as decorator to log the function call altogether with arguments. :param length: Max message length :param cls_name: Optional class name prefix

avocado.utils.debug.**log_calls_class**(*length=None*)
Use this as decorator to log the function methods' calls. :param length: Max message length

avocado.utils.debug.**measure_duration**(*func*)
Use this as decorator to measure duration of the function execution. The output is "Function $name: ($current_duration, $accumulated_duration)"

## avocado.utils.disk module

Disk utilities

avocado.utils.disk.**freespace**(*path*)

## avocado.utils.distro module

This module provides the client facilities to detect the Linux Distribution it's running under.

**class** avocado.utils.distro.**LinuxDistro**(*name*, *version*, *release*, *arch*)
Bases: object

Simple collection of information for a Linux Distribution

Initializes a new Linux Distro

**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 often 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.

**class** avocado.utils.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 expression 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

avocado.utils.distro.**register_probe**(*probe_class*)
Register a probe to be run during autodetection

avocado.utils.distro.**detect**()
Attempts to detect the Linux Distribution running on this machine

**Returns** the detected *LinuxDistro* or UNKNOWN_DISTRO

**Return type** *LinuxDistro*

## avocado.utils.download module

Methods to download URLs and regular files.

avocado.utils.download.**get_file**(*src*, *dst*, *permissions=None*, *hash_expected=None*, *hash_algorithm='md5'*, *download_retries=1*)
Gets a file from a source location, optionally using caching.

If no hash_expected is provided, simply download the file. Else, keep trying to download the file until download_failures exceeds download_retries or the hashes match.

If the hashes match, return dst. If download_failures exceeds download_retries, raise an EnvironmentError.

**Parameters**

- **src** – source path or URL. May be local or a remote file.
- **dst** – destination path.
- **permissions** – (optional) set access permissions.
- **hash_expected** – Hash string that we expect the file downloaded to have.
- **hash_algorithm** – Algorithm used to calculate the hash string (md5, sha1).
- **download_retries** – Number of times we are going to retry a failed download.

**Raise** EnvironmentError.

**Returns** destination path.

avocado.utils.download.**url_download**(*url*, *filename*, *data=None*, *timeout=300*)
Retrieve a file from given url.

**Parameters**

- **url** – source URL.
- **filename** – destination path.
- **data** – (optional) data to post.
- **timeout** – (optional) default timeout in seconds.

**Returns** *None*.

avocado.utils.download.**url_download_interactive**(*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

avocado.utils.download.**url_open**(*url*, *data=None*, *timeout=5*)
Wrapper to `urllib2.urlopen()` with timeout addition.

**Parameters**

- **url** – URL to open.
- **data** – (optional) data to post.
- **timeout** – (optional) default timeout in seconds.

**Returns** file-like object.

**Raises** *URLError*.

## avocado.utils.filelock module

Utility for individual file access control implemented via PID lock files.

**exception** avocado.utils.filelock.**AlreadyLocked**
Bases: exceptions.Exception

**class** avocado.utils.filelock.**FileLock**(*filename*, *timeout=0*)
Bases: object

Creates an exclusive advisory lock for a file. All processes should use and honor the advisory locking scheme, but uncooperative processes are free to ignore the lock and access the file in any way they choose.

**exception** avocado.utils.filelock.**LockFailed**
Bases: exceptions.Exception

## avocado.utils.gdb module

Module that provides communication with GDB via its GDB/MI interpreter

**class** avocado.utils.gdb.**GDB**(*path='/usr/bin/gdb'*, **extra_args*)
Bases: object

Wraps a GDB subprocess for easier manipulation

**DEFAULT_BREAK = 'main'**

**REQUIRED_ARGS = ['–interpreter=mi', '–quiet']**

**cli_cmd**(*command*)
Sends a cli command encoded as an MI command

**Parameters** **command** (*str*) – a regular GDB cli command

**Returns** a CommandResult instance

**Return type** CommandResult

**cmd**(*command*)
Sends a command and parses all lines until prompt is received

**Parameters** **command** (*str*) – the GDB command, hopefully in MI language

**Returns** a CommandResult instance

**Return type** CommandResult

**cmd_exists**(*command*)
Checks if a given command exists

**Parameters** **command** (*str*) – a GDB MI command, including the dash (-) prefix

**Returns** either True or False

**Return type** bool

**connect**(*port*)
Connects to a remote debugger (a gdbserver) at the given TCP port

This uses the "extended-remote" target type only

**Parameters** **port** (*int*) – the TCP port number

**Returns** a CommandResult instance

**Return type** CommandResult

**del_break**(*number*)
Deletes a breakpoint by its number

**Parameters** **number** (*int*) – the breakpoint number

**Returns** a CommandResult instance

**Return type** CommandResult

**disconnect**()
Disconnects from a remote debugger

**Returns** a CommandResult instance

**Return type** CommandResult

**exit**()
Exits the GDB application gracefully

**Returns** the result of `subprocess.POpen.wait()`, that is, a `subprocess.POpen.returncode`

**Return type** int or None

**read_gdb_response**(*timeout=0.01*, *max_tries=100*)
Read raw responses from GDB

**Parameters**

- **timeout** (*float*) – the amount of time to way between read attempts
- **max_tries** (*int*) – the maximum number of cycles to try to read until a response is obtained

**Returns** a string containing a raw response from GDB

**Return type** str

**read_until_break**(*max_lines=100*)
Read lines from GDB until a break condition is reached

**Parameters** **max_lines** (*int*) – the maximum number of lines to read

**Returns** a list of messages read

**Return type** list of str

**run**(*args=[]*)
Runs the application inside the debugger

**Parameters** **args** (*builtin.list*) – the arguments to be passed to the binary as command line arguments

**Returns** a `CommandResult` instance

**Return type** `CommandResult`

**send_gdb_command**(*command*)
Send a raw command to the GNU debugger input

**Parameters** **command** (*str*) – the GDB command, hopefully in MI language

**Returns** None

**set_break**(*location*, *ignore_error=False*)
Sets a new breakpoint on the binary currently being debugged

**Parameters** **location** (*str*) – a breakpoint location expression as accepted by GDB

**Returns** a `CommandResult` instance

**Return type** `CommandResult`

**set_file**(*path*)
Sets the file that will be executed

**Parameters** **path** (*str*) – the path of the binary that will be executed

**Returns** a `CommandResult` instance

**Return type** `CommandResult`

**class** avocado.utils.gdb.**GDBServer**(*path='/usr/bin/gdbserver'*, *port=None*, *wait_until_running=True*, **extra_args*)

Bases: object

Wraps a gdbserver instance

Initializes a new gdbserver instance

**Parameters**

- **path** (*str*) – location of the gdbserver binary
- **port** (*int*) – tcp port number to listen on for incoming connections
- **wait_until_running** (*bool*) – wait until the gdbserver is running and accepting connections. It may take a little after the process is started and it is actually bound to the allocated port
- **extra_args** – optional extra arguments to be passed to gdbserver

**INIT_TIMEOUT = 5.0**

The time to optionally wait for the server to initialize itself and be ready to accept new connections

**PORT_RANGE = (20000, 20999)**

The range from which a port to GDB server will try to be allocated from

**REQUIRED_ARGS = ['–multi']**

The default arguments used when starting the GDB server process

**exit**(*force=True*)

Quits the gdb_server process

Most correct way of quitting the GDB server is by sending it a command. If no GDB client is connected, then we can try to connect to it and send a quit command. If this is not possible, we send it a signal and wait for it to finish.

**Parameters** **force** (*bool*) – if a forced exit (sending SIGTERM) should be attempted

**Returns** None

**class** avocado.utils.gdb.**GDBRemote**(*host*, *port*, *no_ack_mode=True*, *extended_mode=True*)

Bases: object

Initializes a new GDBRemote object.

A GDBRemote acts like a client that speaks the GDB remote protocol, documented at:

https://sourceware.org/gdb/current/onlinedocs/gdb/Remote-Protocol.html

Caveat: we currently do not support communicating with devices, only with TCP sockets. This limitation is basically due to the lack of use cases that justify an implementation, but not due to any technical shortcoming.

**Parameters**

- **host** (*str*) – the IP address or host name
- **port** (*int*) – the port number where the the remote GDB is listening on
- **no_ack_mode** (*bool*) – if the packet transmission confirmation mode should be disabled
- **extended_mode** – if the remote extended mode should be enabled

**cmd**(*command_data*, *expected_response=None*)

Sends a command data to a remote gdb server

Limitations: the current version does not deal with retransmissions.

**Parameters**

- **command_data** (*str*) – the remote command to send the the remote stub
- **expected_response** (*str*) – the (optional) response that is expected as a response for the command sent

**Raises** RetransmissionRequestedError, UnexpectedResponseError

**Returns** raw data read from from the remote server

**Return type** str

**connect**()
Connects to the remote target and initializes the chosen modes

**set_extended_mode**()
Enable extended mode. In extended mode, the remote server is made persistent. The 'R' packet is used to restart the program being debugged. Original documentation at:

https://sourceware.org/gdb/current/onlinedocs/gdb/Packets.html#extended-mode

**start_no_ack_mode**()
Request that the remote stub disable the normal +/- protocol acknowledgments. Original documentation at:

https://sourceware.org/gdb/current/onlinedocs/gdb/General-Query-Packets.html#QStartNoAckMode

## avocado.utils.genio module

Avocado generic IO related functions.

**exception** avocado.utils.genio.**GenIOError**
Bases: exceptions.Exception

Base Exception Class for all IO exceptions

avocado.utils.genio.**ask**(*question*, *auto=False*)
Prompt the user with a (y/n) question.

**Parameters**

- **question** (*str*) – Question to be asked
- **auto** (*bool*) – Whether to return "y" instead of asking the question

**Returns** User answer

**Return type** str

avocado.utils.genio.**close_log_file**(*filename*)

avocado.utils.genio.**log_line**(*filename*, *line*)
Write a line to a file.

**Parameters**

- **filename** – Path of file to write to, either absolute or relative to the dir set by set_log_file_dir().
- **line** – Line to write.

avocado.utils.genio.**read_all_lines**(*filename*)
Return all lines of a given file

This utility method returns an empty list in any error scenario, that is, it doesn't attempt to identify error paths and raise appropriate exceptions. It does exactly the opposite to that.

This should be used when it's fine or desirable to have an empty set of lines if a file is missing or is unreadable.

**Parameters** **filename** (*str*) – Path to the file.

**Returns** all lines of the file as list

**Return type** builtin.list

avocado.utils.genio.**read_file**(*filename*)
Read the entire contents of file.

**Parameters** **filename** (*str*) – Path to the file.

**Returns** File contents

**Return type** str

avocado.utils.genio.**read_one_line**(*filename*)
Read the first line of filename.

**Parameters** **filename** (*str*) – Path to the file.

**Returns** First line contents

**Return type** str

avocado.utils.genio.**set_log_file_dir**(*directory*)
Set the base directory for log files created by log_line().

**Parameters** **dir** – Directory for log files.

avocado.utils.genio.**write_file**(*filename*, *data*)
Write data to a file.

**Parameters**

- **filename** (*str*) – Path to the file.
- **line** (*str*) – Line to be written.

avocado.utils.genio.**write_file_or_fail**(*filename*, *data*)
Write to a file and raise exception on write failure

**Parameters**

- **filename** (*str*) – Path to file
- **data** (*str*) – Data to be written to file

**Raises** ***GenIOError*** – On write Failure

avocado.utils.genio.**write_one_line**(*filename*, *line*)
Write one line of text to filename.

**Parameters**

- **filename** (*str*) – Path to the file.
- **line** (*str*) – Line to be written.

## avocado.utils.git module

APIs to download/update git repositories from inside python scripts.

**class** avocado.utils.git.**GitRepoHelper**(*uri*, *branch='master'*, *lbranch=None*, *commit=None*, *destination_dir=None*, *base_uri=None*)

Bases: object

Helps to deal with git repos, mostly fetching content from a repo

Instantiates a new GitRepoHelper

**Parameters**

- **uri** (*string*) – git repository url
- **branch** (*string*) – git remote branch
- **lbranch** (*string*) – git local branch name, if different from remote
- **commit** (*string*) – specific commit to download
- **destination_dir** (*string*) – path of a dir where to save downloaded code
- **base_uri** (*string*) – a closer, usually local, git repository url from where to fetch content first from

**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

avocado.utils.git.**get_repo**(*uri*, *branch='master'*, *lbranch=None*, *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
- **lbranch** (*string*) – git local branch name, if different from remote
- **commit** (*string*) – specific commit to download
- **destination_dir** (*string*) – path of a dir where to save downloaded code
- **base_uri** (*string*) – a closer, usually local, git repository url from where to fetch content first from

## avocado.utils.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).

avocado.utils.iso9660.**iso9660**(*path*)

Checks the available tools on a system and chooses class accordingly

This is a convenience function, that will pick the first available 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** avocado.utils.iso9660.**Iso9660IsoInfo**(*path*)

Bases: avocado.utils.iso9660.MixInMntDirMount, avocado.utils.iso9660.BaseIso9660

Represents a ISO9660 filesystem

This implementation is based on the cdrkit's isoinfo tool

**read**(*path*)

**class** avocado.utils.iso9660.**Iso9660IsoRead**(*path*)

Bases: avocado.utils.iso9660.MixInMntDirMount, avocado.utils.iso9660.BaseIso9660

Represents a ISO9660 filesystem

This implementation is based on the libcdio's iso-read tool

**close**()

**copy**(*src*, *dst*)

**read**(*path*)

**class** avocado.utils.iso9660.**Iso9660Mount**(*path*)

Bases: avocado.utils.iso9660.BaseIso9660

Represents a mounted ISO9660 filesystem.

initializes a mounted ISO9660 filesystem

**Parameters** **path** (*str*) – path to the ISO9660 file

**close**()
Perform umount operation on the temporary dir

**Return type** None

**copy**(*src*, *dst*)

**Parameters**

- **src** (*str*) – source
- **dst** (*str*) – destination

**Return type** None

**mnt_dir**

**read**(*path*)
Read data from path

**Parameters** **path** (*str*) – path to read data

**Returns** data content

**Return type** str

## avocado.utils.kernel module

**class** avocado.utils.kernel.**KernelBuild**(*version*, *config_path=None*, *work_dir=None*, *data_dirs=None*)
Bases: object

Build the Linux Kernel from official tarballs.

Creates an instance of *KernelBuild*.

**Parameters**

- **version** – kernel version ("3.19.8").
- **config_path** – path to config file.
- **work_dir** – work directory.
- **data_dirs** – list of directories to keep the downloaded kernel

**Returns** None.

**SOURCE = 'linux-{version}.tar.gz'**

**URL = 'https://www.kernel.org/pub/linux/kernel/v3.x/'**

**build**()
Build kernel from source.

**configure**()
Configure/prepare kernel source to build.

**download**()
Download kernel source.

**uncompress**()
Uncompress kernel source.

avocado.utils.kernel.**check_version**(*version*)
This utility function compares the current kernel version with the version parameter and gives assertion error if the version parameter is greater.

**Parameters** **version** (*string*) – version to be compared with current kernel version

## avocado.utils.linux_modules module

Linux kernel modules APIs

avocado.utils.linux_modules.**BUILTIN = 2**
Config built-in to kernel (=*y*)

avocado.utils.linux_modules.**MODULE = 1**
Config compiled as loadable module (=*m*)

avocado.utils.linux_modules.**NOT_SET = 0**
Config commented out or not set

avocado.utils.linux_modules.**check_kernel_config**(*config_name*)
Reports the configuration of $config_name of the current kernel

**Parameters** **config_name** (*str*) – Name of kernel config to search

**Returns** Config status in running kernel (NOT_SET, BUILTIN, MODULE)

**Return type** int

avocado.utils.linux_modules.**get_loaded_modules**()
Gets list of loaded modules. :return: List of loaded modules.

avocado.utils.linux_modules.**get_submodules**(*module_name*)
Get all submodules of the module.

**Parameters** **module_name** (*str*) – Name of module to search for

**Returns** List of the submodules

**Return type** builtin.list

avocado.utils.linux_modules.**load_module**(*module_name*)
Checks if a module has already been loaded. :param module_name: Name of module to check :return: True if module is loaded, False otherwise :rtype: Bool

avocado.utils.linux_modules.**loaded_module_info**(*module_name*)
Get loaded module details: Size and Submodules.

**Parameters** **module_name** (*str*) – Name of module to search for

**Returns** Dictionary of module name, size, submodules if present, filename, version, number of modules using it, list of modules it is dependent on, list of dictionary of param name and type

**Return type** dict

avocado.utils.linux_modules.**module_is_loaded**(*module_name*)
Is module loaded

**Parameters** **module_name** (*str*) – Name of module to search for

**Returns** True if module is loaded

**Return type** bool

avocado.utils.linux_modules.**parse_lsmod_for_module**(*l_raw*, *module_name*, *escape=True*)
Use a regexp to parse raw lsmod output and get module information :param l_raw: raw output of lsmod :type l_raw: str :param module_name: Name of module to search for :type module_name: str :param escape: Escape regexp tokens in module_name, default True :type escape: bool :return: Dictionary of module info, name, size, submodules if present :rtype: dict

avocado.utils.linux_modules.**unload_module**(*module_name*)
Removes a module. Handles dependencies. If even then it's not possible to remove one of the modules, it will throw an error.CmdError exception.

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

## avocado.utils.lv_utils module

**exception** avocado.utils.lv_utils.**LVException**
Bases: exceptions.Exception

Base Exception Class for all exceptions

avocado.utils.lv_utils.**get_diskspace**(*disk*)
Get the entire disk space of a given disk

**Parameters** **disk** – Name of the disk to find free space

**Returns** size in bytes

avocado.utils.lv_utils.**lv_check**(*vg_name*, *lv_name*)
Check whether provided Logical volume exists.

**Parameters**

- **vg_name** – Name of the volume group
- **lv_name** – Name of the logical volume

avocado.utils.lv_utils.**lv_create**(*vg_name*, *lv_name*, *lv_size*, *force_flag=True*)
Create a Logical volume in a volume group. The volume group must already exist.

**Parameters**

- **vg_name** – Name of the volume group
- **lv_name** – Name of the logical volume
- **lv_size** – Size for the logical volume to be created

avocado.utils.lv_utils.**lv_list**()
List available group volumes.

:return list available logical volumes

avocado.utils.lv_utils.**lv_mount**(*vg_name*, *lv_name*, *mount_loc*, *create_filesystem=''*)
Mount a Logical volume to a mount location.

**Parameters**

- **vg_name** – Name of volume group
- **lv_name** – Name of the logical volume
- **create_filesystem** – Can be one of ext2, ext3, ext4, vfat or empty if the filesystem was already created and the mkfs process is skipped

**Mount_loc** Location to mount the logical volume

avocado.utils.lv_utils.**lv_reactivate**(*vg_name*, *lv_name*, *timeout=10*)
In case of unclean shutdowns some of the lvs is still active and merging is postponed. Use this function to attempt to deactivate and reactivate all of them to cause the merge to happen.

**Parameters**

- **vg_name** – Name of volume group
- **lv_name** – Name of the logical volume
- **timeout** – Timeout between operations

avocado.utils.lv_utils.**lv_remove**(*vg_name*, *lv_name*)
Remove a logical volume.

**Parameters**

- **vg_name** – Name of the volume group
- **lv_name** – Name of the logical volume

avocado.utils.lv_utils.**lv_revert**(*vg_name*, *lv_name*, *lv_snapshot_name*)
Revert the origin to a snapshot.

**Parameters**

- **vg_name** – An existing volume group
- **lv_name** – An existing logical volume
- **lv_snapshot_name** – Name of the snapshot be to reverted

avocado.utils.lv_utils.**lv_revert_with_snapshot**(*vg_name*, *lv_name*, *lv_snapshot_name*, *lv_snapshot_size*)
Perform Logical volume merge with snapshot and take a new snapshot.

**Parameters**

- **vg_name** – Name of volume group in which lv has to be reverted
- **lv_name** – Name of the logical volume to be reverted
- **lv_snapshot_name** – Name of the snapshot be to reverted
- **lv_snapshot_size** – Size of the snapshot

avocado.utils.lv_utils.**lv_take_snapshot**(*vg_name*, *lv_name*, *lv_snapshot_name*, *lv_snapshot_size*)
Take a snapshot of the original Logical volume.

**Parameters**

- **vg_name** – An existing volume group
- **lv_name** – An existing logical volume
- **lv_snapshot_name** – Name of the snapshot be to created
- **lv_snapshot_size** – Size of the snapshot

avocado.utils.lv_utils.**lv_umount**(*vg_name*, *lv_name*)
Unmount a Logical volume from a mount location.

**Parameters**

- **vg_name** – Name of volume group
- **lv_name** – Name of the logical volume

avocado.utils.lv_utils.**thin_lv_create**(*vg_name*, *thinpool_name='lvthinpool'*, *thinpool_size='1.5G'*, *thinlv_name='lvthin'*, *thinlv_size='1G'*)

Create a thin volume from given volume group.

**Parameters**

- **vg_name** – An exist volume group
- **thinpool_name** – The name of thin pool
- **thinpool_size** – The size of thin pool to be created
- **thinlv_name** – The name of thin volume
- **thinlv_size** – The size of thin volume

avocado.utils.lv_utils.**vg_check**(*vg_name*)

Check whether provided volume group exists.

**Parameters** **vg_name** – Name of the volume group.

avocado.utils.lv_utils.**vg_create**(*vg_name*, *pv_list*, *force=False*)

Create a volume group by using the block special devices

**Parameters**

- **vg_name** – Name of the volume group
- **pv_list** – List of physical volumes
- **force** – Create volume group forcefully

avocado.utils.lv_utils.**vg_list**()

List available volume groups.

:return List of volume groups.

avocado.utils.lv_utils.**vg_ramdisk**(*disk*, *vg_name*, *ramdisk_vg_size*, *ramdisk_basedir*, *ramdisk_sparse_filename*)

Create vg on top of ram memory to speed up lv performance. When disk is specified size of the physical volume is taken from existing disk space.

**Parameters**

- **disk** – Name of the disk in which volume groups are created.
- **vg_name** – Name of the volume group.
- **ramdisk_vg_size** – Size of the ramdisk virtual group (MB).
- **ramdisk_basedir** – Base directory for the ramdisk sparse file.
- **ramdisk_sparse_filename** – Name of the ramdisk sparse file.

**Returns** ramdisk_filename, vg_ramdisk_dir, vg_name, loop_device

**Raises** ***LVException*** – On failure

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.

avocado.utils.lv_utils.**vg_ramdisk_cleanup**(*ramdisk_filename=None*, *vg_ramdisk_dir=None*, *vg_name=None*, *loop_device=None*)

Inline cleanup function in case of test error.

It detects whether the components were initialized and if so it tries to remove them. In case of failure it raises summary exception.

**Parameters**

- **ramdisk_filename** – Name of the ramdisk sparse file.
- **vg_ramdisk_dir** – Location of the ramdisk file

**Vg_name** Name of the volume group

**Loop_device** Name of the disk or loop device

**Raises** ***LVException*** – In case it fail to clean things detected in system

avocado.utils.lv_utils.**vg_remove**(*vg_name*)

Remove a volume group.

**Parameters** **vg_name** – Name of the volume group

## avocado.utils.memory module

avocado.utils.memory.**drop_caches**()

Writes back all dirty pages to disk and clears all the caches.

avocado.utils.memory.**freememtotal**()

Read `MemFree` from meminfo.

avocado.utils.memory.**get_buddy_info**(*chunk_sizes*, *nodes='all'*, *zones='all'*)

Get the fragement status of the host.

It uses the same method to get the page size in buddyinfo. The expression to evaluate it is:

```
2^chunk_size * page_size
```

The chunk_sizes can be string make up by all orders that you want to check split 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

avocado.utils.memory.**get_huge_page_size**()
Get size of the huge pages for this system.

**Returns** Huge pages size (KB).

avocado.utils.memory.**get_num_huge_pages**()
Get number of huge pages for this system.

**Returns** Number of huge pages.

avocado.utils.memory.**get_thp_value**(*feature*)
Gets the value of the thp feature arg passed

**Param feature** Thp feature to get value

avocado.utils.memory.**memtotal**()
Read `Memtotal` from meminfo.

avocado.utils.memory.**node_size**()
Return node size.

**Returns** Node size.

avocado.utils.memory.**numa_nodes**()
Get a list of NUMA nodes present on the system.

**Returns** List with nodes.

avocado.utils.memory.**read_from_meminfo**(*key*)
Retrieve key from meminfo.

**Parameters** **key** – Key name, such as `MemTotal`.

avocado.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

avocado.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

avocado.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

avocado.utils.memory.**rounded_memtotal**()
Get memtotal, properly rounded.

**Returns** Total memory, KB.

avocado.utils.memory.**set_num_huge_pages**(*num*)
Set number of huge pages.

**Parameters** **num** – Target number of huge pages.

avocado.utils.memory.**set_thp_value**(*feature*, *value*)
Sets THP feature to a given value

**Parameters**

- **feature** (*str*) – Thp feature to set
- **value** (*str*) – Value to be set to feature

## avocado.utils.multipath module

Module with multipath related utility functions. It needs root access.

avocado.utils.multipath.**device_exists**(*path*)
Checks if a given path exists.

**Returns** True if path exists, False if does not exist.

avocado.utils.multipath.**flush_path**(*path_name*)
Flushes the given multipath.

**Returns** Returns False if command fails, True otherwise.

avocado.utils.multipath.**form_conf_mpath_file**(*blacklist=''*, *defaults_extra=''*)
Form a multipath configuration file, and restart multipath service.

**Parameters**

- **blacklist** – Entry in conf file to indicate blacklist section.
- **defaults_extra** – Extra entry in conf file in defaults section.

avocado.utils.multipath.**get_mpath_name**(*wwid*)
Get multipath name for a given wwid.

**Parameters** **wwid** – wwid of multipath device.

**Returns** Name of multipath device.

avocado.utils.multipath.**get_multipath_wwids**()
Get list of multipath wwids.

**Returns** List of multipath wwids.

avocado.utils.multipath.**get_paths**(*wwid*)
Get list of paths, given a multipath wwid.

**Returns** List of paths.

avocado.utils.multipath.**get_policy**(*wwid*)
Gets path_checker policy, given a multipath wwid.

**Returns** path checker policy.

avocado.utils.multipath.**get_size**(*wwid*)
Gets size of device, given a multipath wwid.

**Returns** size of multipath device.

avocado.utils.multipath.**get_svc_name**()
Gets the multipath service name based on distro.

## avocado.utils.network module

Module with network related utility functions

avocado.utils.network.**find_free_port**(*start_port*, *end_port*, *address='localhost'*)
Return a host free port in the range [start_port, end_port].

**Parameters**

- **start_port** – First port that will be checked.
- **end_port** – Port immediately after the last one that will be checked.

avocado.utils.network.**find_free_ports**(*start_port*, *end_port*, *count*, *address='localhost'*)
Return count of host free ports in the range [start_port, end_port].

**Parameters**

- **count** – Initial number of ports known to be free in the range.
- **start_port** – First port that will be checked.
- **end_port** – Port immediately after the last one that will be checked.

avocado.utils.network.**is_port_free**(*port*, *address*)
Return True if the given port is available for use.

**Parameters** **port** – Port number

## avocado.utils.output module

Utility functions for user friendly display of information.

**class** avocado.utils.output.**ProgressBar**(*minimum=0*, *maximum=100*, *width=75*, *title=''*)
Bases: object

Displays interactively the progress of a given task

Inspired/adapted from https://gist.github.com/t0xicCode/3306295

Initializes a new progress bar

**Parameters**

- **minimum** (*integer*) – minimum (initial) value on the progress bar
- **maximum** (*integer*) – maximum (final) value on the progress bar
- **with** – number of columns, that is screen width

**append_amount**(*amount*)
Increments the current amount value.

**draw**()
Prints the updated text to the screen.

**update_amount**(*amount*)
Performs sanity checks and update the current amount.

**update_percentage**(*percentage*)
Updates the progress bar to the new percentage.

avocado.utils.output.**display_data_size**(*size*)
Display data size in human readable units (SI).

**Parameters** **size** (*int*) – Data size, in Bytes.

**Returns** Human readable string with data size, using SI prefixes.

## avocado.utils.partition module

Utility for handling partitions.

**class** avocado.utils.partition.**MtabLock**
Bases: object

**mtab = None**

**class** avocado.utils.partition.**Partition**(*device*, *loop_size=0*, *mountpoint=None*)
Bases: object

Class for handling partitions and filesystems

**Parameters**

- **device** – The device in question (e.g."/dev/hda2"). If device is a file it will be mounted as loopback.
- **loop_size** – Size of loopback device (in MB). Defaults to 0.
- **mountpoint** – Where the partition to be mounted to.

**get_mountpoint**(*filename=None*)
Find the mount point of this partition object.

**Parameters** **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

**static list_mount_devices**()
Lists mounted file systems and swap on devices.

**static list_mount_points**()
Lists the mount points.

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

**Parameters**

- **fstype** – the filesystem type, such as "ext3", "ext2". Defaults to previously set type or "ext2" if none has set.
- **args** – arguments to be passed to mkfs command.

**mount**(*mountpoint=None*, *fstype=None*, *args=''*)
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.

**unmount** (*force=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 parallel.

When the unmount fails and force==True we unmount the partition ungracefully.

**Returns** 1 on success, 2 on force umount success

**Raises** *PartitionError* – On failure

**exception** avocado.utils.partition.**PartitionError** (*partition*, *reason*, *details=None*)
Bases: exceptions.Exception

Generic PartitionError

## avocado.utils.path module

Avocado path related functions.

**exception** avocado.utils.path.**CmdNotFoundError** (*cmd*, *paths*)
Bases: exceptions.Exception

Indicates that the command was not found in the system after a search.

**Parameters**

- **cmd** – String with the command.
- **paths** – List of paths where we looked after.

**class** avocado.utils.path.**PathInspector** (*path*)
Bases: object

**get_first_line**()

**has_exec_permission**()

**is_empty**()

**is_python**()

**is_script** (*language=None*)

avocado.utils.path.**find_command** (*cmd*, *default=None*)
Try to find a command in the PATH, paranoid version.

**Parameters**

- **cmd** – Command to be found.
- **default** – Command path to use as a fallback if not found in the standard directories.

**Raise** *avocado.utils.path.CmdNotFoundError* in case the command was not found and no default was given.

avocado.utils.path.**get_path** (*base_path*, *user_path*)
Translate a user specified path to a real path. If user_path is relative, append it to base_path. If user_path is absolute, return it as is.

**Parameters**

- **base_path** – The base path of relative user specified paths.
- **user_path** – The user specified path.

avocado.utils.path.**init_dir**(**args*)
Wrapper around os.path.join that creates dirs based on the final path.

**Parameters** **args** – List of dir arguments that will be os.path.joined.

**Returns** directory.

**Return type** str

avocado.utils.path.**usable_ro_dir**(*directory*)
Verify whether dir exists and we can access its contents.

If a usable RO is there, use it no questions asked. If not, let's at least try to create one.

**Parameters** **directory** – Directory

avocado.utils.path.**usable_rw_dir**(*directory*)
Verify whether we can use this dir (read/write).

Checks for appropriate permissions, and creates missing dirs as needed.

**Parameters** **directory** – Directory

## avocado.utils.pci module

Module for all PCI devices related functions.

avocado.utils.pci.**get_cfg**(*dom_pci_address*)
Gets the hardware configuration data of the given PCI address.

**Note** Specific for ppc64 processor.

**Parameters** **dom_pci_address** – Partial PCI address including domain addr and at least bus addr (0003:00, 0003:00:1f.2, ...)

**Returns** dictionary of configuration data of a PCI address.

avocado.utils.pci.**get_disks_in_pci_address**(*pci_address*)
Gets disks in a PCI address.

**Parameters** **pci_address** – Any segment of a PCI address (1f, 0000:00:1f, ...)

**Returns** list of disks in a PCI address.

avocado.utils.pci.**get_domains**()
Gets all PCI domains. Example, it returns ['0000', '0001', ...]

**Returns** List of PCI domains.

avocado.utils.pci.**get_driver**(*pci_address*)
Gets the kernel driver in use of given PCI address. (first match only)

**Parameters** **pci_address** – Any segment of a PCI address (1f, 0000:00:1f, ...)

**Returns** driver of a PCI address.

avocado.utils.pci.**get_interfaces_in_pci_address**(*pci_address*, *pci_class*)
Gets interface in a PCI address.

**e.g: host = pci.get_interfaces_in_pci_address("0001:01:00.0", "net")** ['enP1p1s0f0'] host = pci.get_interfaces_in_pci_address("0004:01:00.0", "fc_host") ['host6']

**Parameters**

- **pci_address** – Any segment of a PCI address (1f, 0000:00:1f, ...)
- **class** – Adapter type (FC(fc_host), FCoE(net), NIC(net), SCSI(scsi)..)

**Returns** list of generic interfaces in a PCI address.

avocado.utils.pci.**get_mask**(*pci_address*)
Gets the mask of PCI address. (first match only)

**Note** There may be multiple memory entries for a PCI address.

**Note** This mask is calculated only with the first such entry.

**Parameters** **pci_address** – Any segment of a PCI address (1f, 0000:00:1f, ...)

**Returns** mask of a PCI address.

avocado.utils.pci.**get_memory_address**(*pci_address*)
Gets the memory address of a PCI address. (first match only)

**Note** There may be multiple memory address for a PCI address.

**Note** This function returns only the first such address.

**Parameters** **pci_address** – Any segment of a PCI address (1f, 0000:00:1f, ...)

**Returns** memory address of a pci_address.

avocado.utils.pci.**get_nics_in_pci_address**(*pci_address*)
Gets network interface(nic) in a PCI address.

**Parameters** **pci_address** – Any segment of a PCI address (1f, 0000:00:1f, ...)

**Returns** list of network interfaces in a PCI address.

avocado.utils.pci.**get_num_interfaces_in_pci**(*dom_pci_address*)
Gets number of interfaces of a given partial PCI address starting with full domain address.

**Parameters** **dom_pci_address** – Partial PCI address including domain address (0000, 0000:00:1f, 0000:00:1f.2, etc)

**Returns** number of devices in a PCI domain.

avocado.utils.pci.**get_pci_addresses**()
Gets list of PCI addresses in the system. Does not return the PCI Bridges/Switches.

**Returns** list of full PCI addresses including domain (0000:00:14.0)

avocado.utils.pci.**get_pci_class_name**(*pci_address*)
Gets pci class name for given pci bus address

**e.g: >>> pci.get_pci_class_name("0000:01:00.0")** 'scsi_host'

**Parameters** **pci_address** – Any segment of a PCI address(1f, 0000:00:if, ...)

**Returns** class name for corresponding pci bus address

avocado.utils.pci.**get_pci_fun_list**(*pci_address*)
Gets list of functions in the given PCI address. Example: in address 0000:03:00, functions are 0000:03:00.0 and 0000:03:00.1

**Parameters** **pci_address** – Any segment of a PCI address (1f, 0000:00:1f, ...)

**Returns** list of functions in a PCI address.

avocado.utils.pci.**get_pci_id**(*pci_address*)
Gets PCI id of given address. (first match only)

**Parameters** **pci_address** – Any segment of a PCI address (1f, 0000:00:1f, ...)

**Returns** PCI ID of a PCI address.

avocado.utils.pci.**get_pci_id_from_sysfs**(*full_pci_address*)
Gets the PCI ID from sysfs of given PCI address.

**Parameters** **full_pci_address** – Full PCI address including domain (0000:03:00.0)

**Returns** PCI ID of a PCI address from sysfs.

avocado.utils.pci.**get_pci_prop**(*pci_address*, *prop*)
Gets specific PCI ID of given PCI address. (first match only)

**Parameters**

- **pci_address** – Any segment of a PCI address (1f, 0000:00:1f, ...)
- **part** – prop of PCI ID.

**Returns** specific PCI ID of a PCI address.

avocado.utils.pci.**get_slot_from_sysfs**(*full_pci_address*)
Gets the PCI slot of given address.

**Note** Specific for ppc64 processor.

**Parameters** **full_pci_address** – Full PCI address including domain (0000:03:00.0)

**Returns** slot of PCI address from sysfs.

avocado.utils.pci.**get_slot_list**()
Gets list of PCI slots in the system.

**Note** Specific for ppc64 processor.

**Returns** list of slots in the system.

avocado.utils.pci.**get_vpd**(*dom_pci_address*)
Gets the VPD (Virtual Product Data) of the given PCI address.

**Note** Specific for ppc64 processor.

**Parameters** **dom_pci_address** – Partial PCI address including domain addr and at least bus addr (0003:00, 0003:00:1f.2, ...)

**Returns** dictionary of VPD of a PCI address.

## avocado.utils.process module

Functions dedicated to find and run external commands.

avocado.utils.process.**CURRENT_WRAPPER = None**
The active wrapper utility script.

**exception** avocado.utils.process.**CmdError**(*command=None*, *result=None*, *additional_text=None*)
Bases: exceptions.Exception

**class** avocado.utils.process.**CmdResult**(*command=''*, *stdout=''*, *stderr=''*, *exit_status=None*, *duration=0*, *pid=None*)
Bases: object

Command execution result.

**Parameters**

- **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
- **pid** – ID of the process

**class** avocado.utils.process.**GDBSubProcess**(*cmd*, *verbose=True*, *allow_output_check='all'*, *shell=False*, *env=None*, *sudo=False*, *ignore_bg_processes=False*)

Bases: object

Runs a subprocess inside the GNU Debugger

Creates the subprocess object, stdout/err, reader threads and locks.

**Parameters**

- **cmd** (*str*) – Command line to run.
- **verbose** (*bool*) – Whether to log the command run and stdout/stderr. Currently unused and provided for compatibility only.
- **allow_output_check** (*str*) – Whether to log the command stream outputs (stdout and stderr) in the test stream files. Valid values: 'stdout', for allowing only standard output, 'stderr', to allow only standard error, 'all', to allow both standard output and error (default), and 'none', to allow none to be recorded. Currently unused and provided for compatibility only.
- **sudo** – This param will be ignored in this implementation, since the GDB wrapping code does not have support to run commands under sudo just yet.
- **ignore_bg_processes** – This param will be ignored in this implementation, since the GDB wrapping code does not have support to run commands in that way.

**create_and_wait_on_resume_fifo**(*path*)

Creates a FIFO file and waits until it's written to

**Parameters** **path** (*str*) – the path that the file will be created

**Returns** first character that was written to the fifo

**Return type** str

**generate_core**()

**generate_gdb_connect_cmds**()

**generate_gdb_connect_sh**()

**handle_break_hit**(*response*)

**handle_fatal_signal**(*response*)

**run**(*timeout=None*)

**wait_for_exit**()

Waits until debugger receives a message about the binary exit

**class** avocado.utils.process.**SubProcess**(*cmd*, *verbose=True*, *allow_output_check='all'*, *shell=False*, *env=None*, *sudo=False*, *ignore_bg_processes=False*)

Bases: object

Run a subprocess in the background, collecting stdout/stderr streams.

Creates the subprocess object, stdout/err, reader threads and locks.

**Parameters**

- **cmd** (*str*) – Command line to run.
- **verbose** (*bool*) – Whether to log the command run and stdout/stderr.
- **allow_output_check** (*str*) – Whether to log the command stream outputs (stdout and stderr) in the test stream files. Valid values: 'stdout', for allowing only standard output, 'stderr', to allow only standard error, 'all', to allow both standard output and error (default), and 'none', to allow none to be recorded.
- **shell** (*bool*) – Whether to run the subprocess in a subshell.
- **env** (*dict*) – Use extra environment variables.
- **sudo** – Whether the command requires admin privileges to run, so that sudo will be prepended to the command. The assumption here is that the user running the command has a sudo configuration such that a password won't be prompted. If that's not the case, the command will straight out fail.
- **ignore_bg_processes** – When True the process does not wait for child processes which keep opened stdout/stderr streams after the main process finishes (eg. forked daemon which did not closed the stdout/stderr). Note this might result in missing output produced by those daemons after the main thread finishes and also it allows those daemons to be running after the process finishes.

**get_pid**()
Reports PID of this process

**get_stderr**()
Get the full stderr of the subprocess so far.

**Returns** Standard error of the process.

**Return type** str

**get_stdout**()
Get the full stdout of the subprocess so far.

**Returns** Standard output of the process.

**Return type** str

**kill**()
Send a signal.SIGKILL to the process.

**poll**()
Call the subprocess poll() method, fill results if rc is not None.

**run**(*timeout=None*, *sig=15*)
Start a process and wait for it to end, returning the result attr.

If the process was already started using .start(), this will simply wait for it to end.

**Parameters**

- **timeout** (*float*) – Time (seconds) we'll wait until the process is finished. If it's not, we'll try to terminate it and get a status.
- **sig** (*int*) – Signal to send to the process in case it did not end after the specified timeout.

**Returns** The command result object.

**Return type** A *CmdResult* instance.

**send_signal**(*sig*)
Send the specified signal to the process.

**Parameters** **sig** – Signal to send.

**start**()
Start running the subprocess.

This method is particularly useful for background processes, since you can start the subprocess and not block your test flow.

**Returns** Subprocess PID.

**Return type** int

**stop**()
Stop background subprocess.

Call this method to terminate the background subprocess and wait for it results.

**terminate**()
Send a `signal.SIGTERM` to the process.

**wait**()
Call the subprocess poll() method, fill results if rc is not None.

avocado.utils.process.**UNDEFINED_BEHAVIOR_EXCEPTION = None**
Exception to be raised when users of this API need to know that the execution of a given process resulted in undefined behavior. One concrete example when a user, in an interactive session, let the inferior process exit before before avocado resumed the debugger session. Since the information is unknown, and the behavior is undefined, this situation will be flagged by an exception.

avocado.utils.process.**WRAP_PROCESS = None**
The global wrapper. If set, run every process under this wrapper.

avocado.utils.process.**WRAP_PROCESS_NAMES_EXPR = []**
Set wrapper per program names. A list of wrappers and program names. Format: [ ('/path/to/wrapper.sh', 'progname'), ... ]

**class** avocado.utils.process.**WrapSubProcess**(*cmd*, *verbose=True*, *allow_output_check='all'*, *shell=False*, *env=None*, *wrapper=None*, *sudo=False*, *ignore_bg_processes=False*)
Bases: *avocado.utils.process.SubProcess*

Wrap subprocess inside an utility program.

avocado.utils.process.**binary_from_shell_cmd**(*cmd*)
Tries to find the first binary path from a simple shell-like command.

**Note** It's a naive implementation, but for commands like: *VAR=VAL binary -args || true* gives the right result (binary)

**Parameters** **cmd** – simple shell-like binary

**Returns** first found binary from the cmd

avocado.utils.process.**can_sudo**(*cmd=None*)
Check whether sudo is available (or running as root)

avocado.utils.process.**get_children_pids**(*ppid*, *recursive=False*)
Get all PIDs of children/threads of parent ppid param ppid: parent PID param recursive: True to return all levels of sub-processes return: list of PIDs of all children/threads of ppid

avocado.utils.process.**get_sub_process_klass**(*cmd*)
Which sub process implementation should be used

Either the regular one, or the GNU Debugger version

**Parameters** **cmd** – the command arguments, from where we extract the binary name

avocado.utils.process.**kill_process_by_pattern**(*pattern*)
Send SIGTERM signal to a process with matched pattern.

**Parameters** **pattern** – normally only matched against the process name

avocado.utils.process.**kill_process_tree**(*pid*, *sig=9*, *send_sigcont=True*)
Signal a process and all of its children.

If the process does not exist – return.

**Parameters**

- **pid** – The pid of the process to signal.
- **sig** – The signal to send to the processes.

avocado.utils.process.**pid_exists**(*pid*)
Return True if a given PID exists.

**Parameters** **pid** – Process ID number.

avocado.utils.process.**process_in_ptree_is_defunct**(*ppid*)
Verify if any processes deriving from PPID are in the defunct state.

Attempt to verify if parent process and any children from PPID is defunct (zombie) or not.

**Parameters** **ppid** – The parent PID of the process to verify.

avocado.utils.process.**run**(*cmd*, *timeout=None*, *verbose=True*, *ignore_status=False*, *allow_output_check='all'*, *shell=False*, *env=None*, *sudo=False*, *ignore_bg_processes=False*)
Run a subprocess, returning a CmdResult object.

**Parameters**

- **cmd** (*str*) – Command line to run.
- **timeout** (*float*) – Time limit in seconds before attempting to kill the running process. This function will take a few seconds longer than 'timeout' to complete if it has to kill the process.
- **verbose** (*bool*) – Whether to log the command run and stdout/stderr.
- **ignore_status** (*bool*) – Whether to raise an exception when command returns =! 0 (False), or not (True).
- **allow_output_check** (*str*) – Whether to log the command stream outputs (stdout and stderr) in the test stream files. Valid values: 'stdout', for allowing only standard output, 'stderr', to allow only standard error, 'all', to allow both standard output and error (default), and 'none', to allow none to be recorded.
- **shell** (*bool*) – Whether to run the command on a subshell

- **env** (*dict*) – Use extra environment variables
- **sudo** – Whether the command requires admin privileges to run, so that sudo will be prepended to the command. The assumption here is that the user running the command has a sudo configuration such that a password won't be prompted. If that's not the case, the command will straight out fail.

**Returns** An *CmdResult* object.

**Raise** *CmdError*, if `ignore_status=False`.

`avocado.utils.process.`**`safe_kill`**(*pid*, *signal*)
Attempt to send a signal to a given process that may or may not exist.

**Parameters** **signal** – Signal number.

`avocado.utils.process.`**`should_run_inside_gdb`**(*cmd*)
Wether the given command should be run inside the GNU debugger

**Parameters** **cmd** – the command arguments, from where we extract the binary name

`avocado.utils.process.`**`should_run_inside_wrapper`**(*cmd*)
Wether the given command should be run inside the wrapper utility.

**Parameters** **cmd** – the command arguments, from where we extract the binary name

`avocado.utils.process.`**`split_gdb_expr`**(*expr*)
Splits a GDB expr into (binary_name, breakpoint_location)

Returns `avocado.gdb.GDB.DEFAULT_BREAK` as the default breakpoint if one is not given.

**Parameters** **expr** (*str*) – an expression of the form <binary_name>[:<breakpoint>]

**Returns** a (binary_name, breakpoint_location) tuple

**Return type** tuple

`avocado.utils.process.`**`system`**(*cmd*, *timeout=None*, *verbose=True*, *ignore_status=False*, *allow_output_check='all'*, *shell=False*, *env=None*, *sudo=False*, *ignore_bg_processes=False*)
Run a subprocess, returning its exit code.

**Parameters**

- **cmd** (*str*) – Command line to run.
- **timeout** (*float*) – Time limit in seconds before attempting to kill the running process. This function will take a few seconds longer than 'timeout' to complete if it has to kill the process.
- **verbose** (*bool*) – Whether to log the command run and stdout/stderr.
- **ignore_status** (*bool*) – Whether to raise an exception when command returns =! 0 (False), or not (True).
- **allow_output_check** (*str*) – Whether to log the command stream outputs (stdout and stderr) in the test stream files. Valid values: 'stdout', for allowing only standard output, 'stderr', to allow only standard error, 'all', to allow both standard output and error (default), and 'none', to allow none to be recorded.
- **shell** (*bool*) – Whether to run the command on a subshell
- **env** (*dict*) – Use extra environment variables.

- **sudo** – Whether the command requires admin privileges to run, so that sudo will be prepended to the command. The assumption here is that the user running the command has a sudo configuration such that a password won't be prompted. If that's not the case, the command will straight out fail.

**Returns** Exit code.

**Return type** int

**Raise** *CmdError*, if `ignore_status=False`.

`avocado.utils.process.`**`system_output`**(*cmd*, *timeout=None*, *verbose=True*, *ignore_status=False*, *allow_output_check='all'*, *shell=False*, *env=None*, *sudo=False*, *ignore_bg_processes=False*, *strip_trail_nl=True*)

Run a subprocess, returning its output.

**Parameters**

- **cmd** (*str*) – Command line to run.
- **timeout** (*float*) – Time limit in seconds before attempting to kill the running process. This function will take a few seconds longer than 'timeout' to complete if it has to kill the process.
- **verbose** (*bool*) – Whether to log the command run and stdout/stderr.
- **ignore_status** – Whether to raise an exception when command returns =! 0 (False), or not (True).
- **allow_output_check** (*str*) – Whether to log the command stream outputs (stdout and stderr) in the test stream files. Valid values: 'stdout', for allowing only standard output, 'stderr', to allow only standard error, 'all', to allow both standard output and error (default), and 'none', to allow none to be recorded.
- **shell** (*bool*) – Whether to run the command on a subshell
- **env** (*dict*) – Use extra environment variables
- **sudo** (*bool*) – Whether the command requires admin privileges to run, so that sudo will be prepended to the command. The assumption here is that the user running the command has a sudo configuration such that a password won't be prompted. If that's not the case, the command will straight out fail.
- **ignore_bg_processes** (*bool*) – Whether to ignore background processes
- **strip_trail_nl** (*bool*) – Whether to strip the trailing newline

**Returns** Command output.

**Return type** str

**Raise** *CmdError*, if `ignore_status=False`.

## avocado.utils.runtime module

Module that contains runtime configuration

`avocado.utils.runtime.`**`CURRENT_JOB = None`**

Sometimes it's useful for the framework and API to know about the job that is currently running, if one exists

`avocado.utils.runtime.`**`CURRENT_TEST = None`**

Sometimes it's useful for the framework and API to know about the test that is currently running, if one exists

## avocado.utils.script module

Module to handle scripts creation.

avocado.utils.script.**DEFAULT_MODE = 509**
What is commonly known as "0775" or "u=rwx,g=rwx,o=rx"

**class** avocado.utils.script.**Script**(*path*, *content*, *mode=509*)
Bases: object

Class that represents a script.

Creates an instance of *Script*.

Note that when the instance inside a with statement, it will automatically call save() and then remove() for you.

**Parameters**

- **path** – the script file name.
- **content** – the script content.
- **mode** – set file mode, defaults what is commonly known as 0775.

**remove**()
Remove script from the file system.

**Returns** *True* if script has been removed, otherwise *False*.

**save**()
Store script to file system.

**Returns** *True* if script has been stored, otherwise *False*.

**class** avocado.utils.script.**TemporaryScript**(*name*, *content*, *prefix='avocado_script'*, *mode=509*)
Bases: *avocado.utils.script.Script*

Class that represents a temporary script.

Creates an instance of *TemporaryScript*.

Note that when the instance inside a with statement, it will automatically call save() and then remove() for you.

When the instance object is garbage collected, it will automatically call remove() for you.

**Parameters**

- **name** – the script file name.
- **content** – the script content.
- **prefix** – prefix for the temporary directory name.
- **mode** – set file mode, default to 0775.

**remove**()

avocado.utils.script.**make_script**(*path*, *content*, *mode=509*)
Creates a new script stored in the file system.

**Parameters**

- **path** – the script file name.
- **content** – the script content.
- **mode** – set file mode, default to 0775.

**Returns** the script path.

avocado.utils.script.**make_temp_script**(*name*, *content*, *prefix='avocado_script'*, *mode=509*)
Creates a new temporary script stored in the file system.

**Parameters**

- **path** – the script file name.
- **content** – the script content.
- **prefix** – the directory prefix Default to 'avocado_script'.
- **mode** – set file mode, default to 0775.

**Returns** the script path.

## avocado.utils.service module

avocado.utils.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

avocado.utils.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

avocado.utils.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

avocado.utils.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

avocado.utils.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

avocado.utils.service.**service_manager**(*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

avocado.utils.service.**specific_service_manager**(*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

avocado.utils.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 process.run or similar function

**Return type** builtin.list

avocado.utils.service.**sys_v_init_result_parser**(*command*)

Parse results from sys_v style commands.

command status: return true if service is running. command is_enabled: return true if service is enabled. command list: return a dict from service name to status. command others: return true if operate success.

**Parameters** **command** (*str.*) – command.

**Returns** different from the command.

avocado.utils.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 process.run or similar functions

**Return type** builtin.list

avocado.utils.service.**systemd_result_parser**(*command*)
Parse results from systemd style commands.

command status: return true if service is running. command is_enabled: return true if service is enabled. command list: return a dict from service name to status. command others: return true if operate success.

**Parameters** **command** (*str.*) – command.

**Returns** different from the command.

## avocado.utils.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 2009-2014

**class** avocado.utils.software_manager.**AptBackend**
Bases: *avocado.utils.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.

Initializes the base command and the debian package repository.

**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** avocado.utils.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** avocado.utils.software_manager.**DnfBackend**
Bases: *avocado.utils.software_manager.YumBackend*

Implements the dnf backend for software manager.

DNF is the successor to yum in recent Fedora.

Initializes the base command and the DNF package repository.

**class** avocado.utils.software_manager.**DpkgBackend**
Bases: *avocado.utils.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** avocado.utils.software_manager.**RpmBackend**
Bases: *avocado.utils.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** avocado.utils.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.

Lazily instantiate the object

**class** avocado.utils.software_manager.**SystemInspector**
Bases: object

System inspector class.

This may grow up to include more complete reports of operating system and machine properties.

Probe system, and save information for future reference.

**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** avocado.utils.software_manager.**YumBackend**(*cmd='yum'*)
Bases: *avocado.utils.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.

Initializes the base command and the yum package repository.

**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** avocado.utils.software_manager.**ZypperBackend**
Bases: *avocado.utils.software_manager.RpmBackend*

Implements the zypper backend for software manager.

Set of operations for the zypper package manager, found on SUSE Linux.

Initializes the base command and the yum package repository.

**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

avocado.utils.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 currently 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

avocado.utils.software_manager.**main**()

## avocado.utils.stacktrace module

Traceback standard module plus some additional APIs.

avocado.utils.stacktrace.**analyze_unpickable_item**(*path_prefix*, *obj*)
Recursive method to obtain unpickable objects along with location

**Parameters**

- **path_prefix** – Path to this object
- **obj** – The sub-object under introspection

**Returns** [($path_to_the_object, $value), ...]

avocado.utils.stacktrace.**log_exc_info**(*exc_info*, *logger=''*)
Log exception info to logger_name.

**Parameters**

- **exc_info** – Exception info produced by sys.exc_info()
- **logger** – Name or logger instance (defaults to '')

avocado.utils.stacktrace.**log_message**(*message*, *logger=''*)
Log message to logger.

**Parameters**

- **message** – Message
- **logger** – Name or logger instance (defaults to '')

avocado.utils.stacktrace.**prepare_exc_info**(*exc_info*)
Prepare traceback info.

**Parameters** **exc_info** – Exception info produced by sys.exc_info()

avocado.utils.stacktrace.**str_unpickable_object**(*obj*)
Return human readable string identifying the unpickable objects

**Parameters** **obj** – The object for analysis

**Raises** **ValueError** – In case the object is pickable

avocado.utils.stacktrace.**tb_info**(*exc_info*)
Prepare traceback info.

**Parameters** **exc_info** – Exception info produced by sys.exc_info()

## avocado.utils.wait module

avocado.utils.wait.**wait_for**(*func*, *timeout*, *first=0.0*, *step=1.0*, *text=None*)
Wait until func() evaluates to True.

If func() evaluates to True before timeout expires, return the value of func(). Otherwise return None.

**Parameters**

- **timeout** – Timeout in seconds
- **first** – Time to sleep before first attempt
- **steps** – Time to sleep between attempts in seconds
- **text** – Text to print while waiting, for debug purposes

### Module contents

## Internal (Core) APIs

Internal APIs that may be of interest to Avocado hackers.

### Subpackages

#### avocado.core.restclient package

#### Subpackages

#### avocado.core.restclient.cli package

#### Subpackages

#### avocado.core.restclient.cli.actions package

#### Submodules

#### avocado.core.restclient.cli.actions.base module

avocado.core.restclient.cli.actions.base.**action**(*function*)
Simple function that marks functions as CLI actions

**Parameters** **function** – the function that will receive the CLI action mark

#### avocado.core.restclient.cli.actions.server module

Module that implements the actions for the CLI App when the job toplevel command is used

avocado.core.restclient.cli.actions.server.**list_brief**(*app*)
Shows the server API list

avocado.core.restclient.cli.actions.server.**status**(*app*)
Shows the server status

#### Module contents

#### avocado.core.restclient.cli.args package

#### Submodules

#### avocado.core.restclient.cli.args.base module

This module has base action arguments that are used on other top level commands

These top level commands import these definitions for uniformity and consistency sake

### avocado.core.restclient.cli.args.server module

This module has actions for the server command

### Module contents

### Submodules

### avocado.core.restclient.cli.app module

This is the main entry point for the rest client cli application

**class** avocado.core.restclient.cli.app.**App**
Bases: object

Base class for CLI application

Initializes a new app instance.

This class is intended both to be used by the stock client application and also to be reused by custom applications. If you want, say, to limit the amount of command line actions and its arguments, you can simply supply another argument parser class to this constructor. Of course another way to customize it is to inherit from this and modify its members at will.

**dispatch_action**()
Calls the actions that was specified via command line arguments.

This involves loading the relevant module file.

**initialize_connection**()
Initialize the connection instance

**run**()
Main entry point for application

### avocado.core.restclient.cli.parser module

REST client application command line parsing

**class** avocado.core.restclient.cli.parser.**Parser**(***kwargs*)
Bases: argparse.ArgumentParser

The main CLI Argument Parser.

Initializes a new parser

**add_arguments_on_all_modules**(*prefix='avocado.core.restclient.cli.args'*)
Add arguments that are present on all Python modules at a given prefix

**Parameters** **prefix** – a Python module namespace

**add_arguments_on_module**(*name*, *prefix*)
Add arguments that are present on a given Python module

**Parameters** **name** – the name of the Python module, without the namespace

### Module contents

### Submodules

### avocado.core.restclient.connection module

This module provides connection classes the avocado server.

A connection is a simple wrapper around a HTTP request instance. It is this basic object that allows methods to be called on the remote server.

avocado.core.restclient.connection.**get_default**()
Returns the global, default connection to avocado-server

**Returns** an avocado.core.restclient.connection.Connection instance

**class** avocado.core.restclient.connection.**Connection**(*hostname=None*, *port=None*, *username=None*, *password=None*)
Bases: object

Connection to the avocado server

Initializes a connection to an avocado-server instance

**Parameters**

- **hostname** (*str*) – the hostname or IP address to connect to
- **port** (*int*) – the port number where avocado-server is running
- **username** (*str*) – the name of the user to be authenticated as
- **password** (*str*) – the password to use for authentication

**check_min_version**(*data=None*)
Checks the minimum server version

**get_api_list**()
Gets the list of APIs the server makes available to the current user

**get_url**(*path=None*)
Returns a representation of the current connection as an HTTP URL

**ping**()
Tests connectivity to the currently set avocado-server

This is intentionally a simple method that will only return True if a request is made, and a response is received from the server.

**request**(*path*, *method=<function get>*, *check_status=True*, ***data*)
Performs a request to the server

This method is heavily used by upper level API methods, and more often than not, those upper level API methods should be used instead.

**Parameters**

- **path** (*str*) – the path on the server where the resource lives
- **method** – the method you want to call on the remote server, defaults to a HTTP GET
- **check_status** – whether to check the HTTP status code that comes with the response. If set to *True*, it will depend on the method chosen. If set to *False*, no check will be performed. If an integer is given then that specific status will be checked for.

- **data** – keyword arguments to be passed to the remote method

**Returns** JSON data

### avocado.core.restclient.response module

Module with base model functions to manipulate JSON data

**class** avocado.core.restclient.response.**BaseResponse**(*json_data*)
Bases: object

Base class that provides commonly used features for response handling

**REQUIRED_DATA = []**

**exception** avocado.core.restclient.response.**InvalidJSONError**
Bases: exceptions.Exception

Data given to a loader/decoder is not valid JSON

**exception** avocado.core.restclient.response.**InvalidResultResponseError**
Bases: exceptions.Exception

Returned result response does not conform to expectation

Even though the result may be a valid json, it may not have the required or expected information that would normally be sent by avocado-server.

**class** avocado.core.restclient.response.**ResultResponse**(*json_data*)
Bases: *avocado.core.restclient.response.BaseResponse*

Provides a wrapper around an ideal result response

This class should be instantiated with the JSON data received from an avocado-server, and will check if the required data members are present and thus the response is well formed.

**REQUIRED_DATA = ['count', 'next', 'previous', 'results']**

### Module contents

## Submodules

## avocado.core.app module

The core Avocado application.

**class** avocado.core.app.**AvocadoApp**
Bases: object

Avocado application.

**run**()

## avocado.core.data_dir module

Library used to let avocado tests find important paths in the system.

The general reasoning to find paths is:

- When running in tree, don't honor avocado.conf. Also, we get to run/display the example tests shipped in tree.

- When avocado.conf is in /etc/avocado, or ~/.config/avocado, then honor the values there as much as possible. If they point to a location where we can't write to, use the next best location available.
- The next best location is the default system wide one.
- The next best location is the default user specific one.

avocado.core.data_dir.**clean_tmp_files**()
Try to clean the tmp directory by removing it.

This is a useful function for avocado entry points looking to clean after tests/jobs are done. If OSError is raised, silently ignore the error.

avocado.core.data_dir.**create_job_logs_dir**(*logdir=None*, *unique_id=None*)
Create a log directory for a job, or a stand alone execution of a test.

**Parameters**

- **logdir** – Base log directory, if *None*, use value from configuration.
- **unique_id** – The unique identification. If *None*, create one.

**Return type** basestring

avocado.core.data_dir.**get_base_dir**()
Get the most appropriate base dir.

The base dir is the parent location for most of the avocado other important directories.

**Examples:**

- Log directory
- Data directory
- Tests directory

avocado.core.data_dir.**get_data_dir**()
Get the most appropriate data dir location.

The data dir is the location where any data necessary to job and test operations are located.

**Examples:**

- ISO files
- GPG files
- VM images
- Reference bitmaps

avocado.core.data_dir.**get_datafile_path**(**args*)
Get a path relative to the data dir.

**Parameters** **args** – Arguments passed to os.path.join. Ex ('images', 'jeos.qcow2')

avocado.core.data_dir.**get_logs_dir**()
Get the most appropriate log dir location.

The log dir is where we store job/test logs in general.

avocado.core.data_dir.**get_test_dir**()
Get the most appropriate test location.

The test location is where we store tests written with the avocado API.

The heuristics used to determine the test dir are: 1) If an explicit test dir is set in the configuration system, it is used. 2) If user is running Avocado out of the source tree, the example test dir is used 3) System wide test dir is used 4) User default test dir (~/avocado/tests) is used

avocado.core.data_dir.**get_tmp_dir**(*basedir=None*)
Get the most appropriate tmp dir location.

The tmp dir is where artifacts produced by the test are kept.

**Examples:**

- Copies of a test suite source code
- Compiled test suite source code

## avocado.core.decorators module

avocado.core.decorators.**fail_on**(*exceptions=None*)
Fail the test when decorated function produces exception of the specified type.

(For example, our method may raise IndexError on tested software failure. We can either try/catch it or use this decorator instead)

**Parameters** **exceptions** – Tuple or single exception to be assumed as test fail [Exception]

**Note** self.error and self.skip behavior remains intact

**Note** To allow simple usage param "exceptions" must not be callable

avocado.core.decorators.**skip**(*message=None*)
Decorator to skip a test.

avocado.core.decorators.**skipIf**(*condition*, *message=None*)
Decorator to skip a test if a condition is True.

avocado.core.decorators.**skipUnless**(*condition*, *message=None*)
Decorator to skip a test if a condition is False.

## avocado.core.dispatcher module

Extensions/plugins dispatchers.

**class** avocado.core.dispatcher.**CLICmdDispatcher**
Bases: *avocado.core.dispatcher.Dispatcher*

Calls extensions on configure/run

Automatically adds all the extension with entry points registered under 'avocado.plugins.cli.cmd'

**class** avocado.core.dispatcher.**CLIDispatcher**
Bases: *avocado.core.dispatcher.Dispatcher*

Calls extensions on configure/run

Automatically adds all the extension with entry points registered under 'avocado.plugins.cli'

**class** avocado.core.dispatcher.**Dispatcher**(*namespace*, *invoke_kwds={}*)
Bases: stevedore.enabled.EnabledExtensionManager

Base dispatcher for various extension types

**NAMESPACE_PREFIX = 'avocado.plugins.'**
Default namespace prefix for Avocado extensions

**enabled**(*extension*)
Checks configuration for explicit mention of plugin in a disable list

If configuration section or key doesn't exist, it means no plugin is disabled.

**fully_qualified_name**(*extension*)
Returns the Avocado fully qualified plugin name

**Parameters** **extension** (stevedore.extension.Extension) – an Stevedore Extension instance

**names**()
Returns the names of the discovered extensions

This differs from stevedore.extension.ExtensionManager.names() in that it returns names in a predictable order, by using standard sorted().

**plugin_type**()
Subset of entry points namespace for this dispatcher

Given an entry point *avocado.plugins.foo*, plugin type is *foo*. If entry point does not conform to the Avocado standard prefix, it's returned unchanged.

**settings_section**()
Returns the config section name for the plugin type handled by itself

**static store_load_failure**(*manager*, *entrypoint*, *exception*)

**class** avocado.core.dispatcher.**JobPrePostDispatcher**
Bases: *avocado.core.dispatcher.Dispatcher*

Calls extensions before Job execution

Automatically adds all the extension with entry points registered under 'avocado.plugins.job.prepost'

**map_method**(*method_name*, *job*)

**class** avocado.core.dispatcher.**ResultDispatcher**
Bases: *avocado.core.dispatcher.Dispatcher*

**map_method**(*method_name*, *result*, *job*)

**class** avocado.core.dispatcher.**ResultEventsDispatcher**(*args*)
Bases: *avocado.core.dispatcher.Dispatcher*

**map_method**(*method_name*, **args*)

**class** avocado.core.dispatcher.**VarianterDispatcher**
Bases: *avocado.core.dispatcher.Dispatcher*

**map_method**(*method_name*, **args*, ***kwargs*)

**map_method_copy**(*method_name*, **args*)
The same as map_method, but use copy.deepcopy on each passed arg

## avocado.core.exceptions module

Exception classes, useful for tests, and other parts of the framework code.

**exception** avocado.core.exceptions.**JobBaseException**
Bases: exceptions.Exception

The parent of all job exceptions.

You should be never raising this, but just in case, we'll set its status' as FAIL.

**status = 'FAIL'**

**exception** avocado.core.exceptions.**JobError**
Bases: *avocado.core.exceptions.JobBaseException*

A generic error happened during a job execution.

**status = 'ERROR'**

**exception** avocado.core.exceptions.**OptionValidationError**
Bases: exceptions.Exception

An invalid option was passed to the test runner

**status = 'ERROR'**

**exception** avocado.core.exceptions.**TestAbortError**
Bases: *avocado.core.exceptions.TestBaseException*

Indicates that the test was prematurely aborted.

**status = 'ERROR'**

**exception** avocado.core.exceptions.**TestBaseException**
Bases: exceptions.Exception

The parent of all test exceptions.

You should be never raising this, but just in case, we'll set its status' as FAIL.

**status = 'FAIL'**

**exception** avocado.core.exceptions.**TestCancel**
Bases: *avocado.core.exceptions.TestBaseException*

Indicates that a test was canceled.

Should be thrown when the cancel() test method is used.

**status = 'CANCEL'**

**exception** avocado.core.exceptions.**TestError**
Bases: *avocado.core.exceptions.TestBaseException*

Indicates that the test was not fully executed and an error happened.

This is the sort of exception you raise if the test was partially executed and could not complete due to a setup, configuration, or another fatal condition.

**status = 'ERROR'**

**exception** avocado.core.exceptions.**TestFail**
Bases: *avocado.core.exceptions.TestBaseException*, exceptions.AssertionError

Indicates that the test failed.

TestFail inherits from AssertionError in order to keep compatibility with vanilla python unittests (they only consider failures the ones deriving from AssertionError).

**status = 'FAIL'**

**exception** avocado.core.exceptions.**TestInterruptedError**
Bases: *avocado.core.exceptions.TestBaseException*

Indicates that the test was interrupted by the user (Ctrl+C)

**status = 'INTERRUPTED'**

**exception** avocado.core.exceptions.**TestNotFoundError**
Bases: *avocado.core.exceptions.TestBaseException*

Indicates that the test was not found in the test directory.

**status = 'ERROR'**

**exception** avocado.core.exceptions.**TestSetupFail**
Bases: *avocado.core.exceptions.TestBaseException*

Indicates an error during a setup or cleanup procedure.

**status = 'ERROR'**

**exception** avocado.core.exceptions.**TestSetupSkip**
Bases: *avocado.core.exceptions.TestBaseException*

Indictates that the test is skipped in setUp().

Should be thrown when skip() is used in setUp().

**status = 'SKIP'**

**exception** avocado.core.exceptions.**TestSkipError**
Bases: *avocado.core.exceptions.TestBaseException*

Indictates that the test is skipped.

Should be thrown when various conditions are such that the test is inappropriate. For example, inappropriate architecture, wrong OS version, program being tested does not have the expected capability (older version).

**status = 'SKIP'**

**exception** avocado.core.exceptions.**TestTimeoutInterrupted**
Bases: *avocado.core.exceptions.TestBaseException*

Indicates that the test did not finish before the timeout specified.

**status = 'INTERRUPTED'**

**exception** avocado.core.exceptions.**TestWarn**
Bases: *avocado.core.exceptions.TestBaseException*

Indicates that bad things (may) have happened, but not an explicit failure.

**status = 'WARN'**

## avocado.core.exit_codes module

Avocado exit codes.

These codes are returned on the command line and may be used by applications that interface (that is, run) the Avocado command line application.

Besides main status about the execution of the command line application, these exit status may also give extra, although limited, information about test statuses.

avocado.core.exit_codes.**AVOCADO_ALL_OK = 0**
Both job and tests PASSed

avocado.core.exit_codes.**AVOCADO_FAIL = 4**
Something else went wrong and avocado failed (or crashed). Commonly used on command line validation errors.

avocado.core.exit_codes.**AVOCADO_GENERIC_CRASH = -1**
Avocado generic crash

avocado.core.exit_codes.**AVOCADO_JOB_FAIL = 2**
Something went wrong with an Avocado Job execution, usually by an explicit *avocado.core.exceptions.JobError* exception.

avocado.core.exit_codes.**AVOCADO_JOB_INTERRUPTED = 8**
The job was explicitly interrupted. Usually this means that a user hit CTRL+C while the job was still running.

avocado.core.exit_codes.**AVOCADO_TESTS_FAIL = 1**
Job went fine, but some tests FAILed or ERRORed

## avocado.core.job module

Job module - describes a sequence of automated test operations.

**class** avocado.core.job.**Job**(*args=None*)
Bases: object

A Job is a set of operations performed on a test machine.

Most of the time, we are interested in simply running tests, along with setup operations and event recording.

Creates an instance of Job class.

**Parameters** **args** – an instance of argparse.Namespace.

**create_test_suite**()
Creates the test suite for this Job

This is a public Job API as part of the documented Job phases

**logdir = None**
The log directory for this job, also known as the job results directory. If it's set to None, it means that the job results directory has not yet been created.

**post_tests**()
Run the post tests execution hooks

By default this runs the plugins that implement the *avocado.core.plugin_interfaces.JobPostTests* interface.

**pre_tests**()
Run the pre tests execution hooks

By default this runs the plugins that implement the *avocado.core.plugin_interfaces.JobPreTests* interface.

**run**()
Runs all job phases, returning the test execution results.

This method is supposed to be the simplified interface for jobs, that is, they run all phases of a job.

**Returns** Integer with overall job status. See *avocado.core.exit_codes* for more information.

**run_tests**()

**test_suite = None**
The list of discovered/resolved tests that will be attempted to be run by this job. If set to None, it means that test resolution has not been attempted. If set to an empty list, it means that no test was found during resolution.

**time_elapsed = None**
The total amount of time the job took from start to finish, or *-1* if it has not been started by means of the *run()* method

**time_end = None**
The time at which the job has finished or *-1* if it has not been started by means of the *run()* method.

**time_start = None**
The time at which the job has started or *-1* if it has not been started by means of the *run()* method.

**class** avocado.core.job.**TestProgram**
Bases: object

Convenience class to make avocado test modules executable.

**parseArgs**(*argv*)

**runTests**()

avocado.core.job.**main**
alias of *TestProgram*

## avocado.core.job_id module

avocado.core.job_id.**create_unique_job_id**()
Create a 40 digit hex number to be used as a job ID string. (similar to SHA1)

**Returns** 40 digit hex number string

**Return type** str

## avocado.core.jobdata module

Record/retrieve job information

avocado.core.jobdata.**get_id**(*path*, *jobid*)
Gets the full Job ID using the results directory path and a partial Job ID or the string 'latest'.

avocado.core.jobdata.**get_resultsdir**(*logdir*, *jobid*)
Gets the job results directory using a Job ID.

avocado.core.jobdata.**record**(*args*, *logdir*, *mux*, *references=None*, *cmdline=None*)
Records all required job information.

avocado.core.jobdata.**retrieve_args**(*resultsdir*)
Retrieves the job args from the results directory.

avocado.core.jobdata.**retrieve_cmdline**(*resultsdir*)
Retrieves the job command line from the results directory.

avocado.core.jobdata.**retrieve_config**(*resultsdir*)
Retrieves the job settings from the results directory.

avocado.core.jobdata.**retrieve_pwd**(*resultsdir*)
Retrieves the job pwd from the results directory.

avocado.core.jobdata.**retrieve_references**(*resultsdir*)
Retrieves the job test references from the results directory.

avocado.core.jobdata.**retrieve_variants**(*resultsdir*)
Retrieves the job Mux object from the results directory.

## avocado.core.loader module

Test loader module.

avocado.core.loader.**ALL = True**
All tests (including broken ones)

avocado.core.loader.**AVAILABLE = None**
Available tests (for listing purposes)

**class** avocado.core.loader.**AccessDeniedPath**
Bases: object

Dummy object to represent reference pointing to a inaccessible path

**class** avocado.core.loader.**BrokenSymlink**
Bases: object

Dummy object to represent reference pointing to a BrokenSymlink path

avocado.core.loader.**DEFAULT = False**
Show default tests (for execution)

**class** avocado.core.loader.**DummyLoader**(*args*, *extra_params*)
Bases: *avocado.core.loader.TestLoader*

Dummy-runner loader class

**discover**(*url*, *which_tests=False*)

**static get_decorator_mapping**()

**static get_type_label_mapping**()

**name = 'dummy'**

**class** avocado.core.loader.**ExternalLoader**(*args*, *extra_params*)
Bases: *avocado.core.loader.TestLoader*

External-runner loader class

**discover**(*reference*, *which_tests=False*)

**Parameters**

- **reference** – arguments passed to the external_runner
- **which_tests** – Limit tests to be displayed (ALL, AVAILABLE or DEFAULT)

**Returns** list of matching tests

**static get_decorator_mapping**()

**static get_type_label_mapping**()

**name = 'external'**

**class** avocado.core.loader.**FileLoader**(*args*, *extra_params*)
Bases: *avocado.core.loader.TestLoader*

Test loader class.

**discover**(*reference*, *which_tests=False*)
Discover (possible) tests from a directory.

Recursively walk in a directory and find tests params. The tests are returned in alphabetic order.

Afterwards when "allowed_test_types" is supplied it verifies if all found tests are of the allowed type. If not return None (even on partial match).

**Parameters**

- **reference** – the directory path to inspect.
- **which_tests** – Limit tests to be displayed (ALL, AVAILABLE or DEFAULT)

**Returns** list of matching tests

**static get_decorator_mapping**()

**static get_type_label_mapping**()

**name = 'file'**

**exception** avocado.core.loader.**InvalidLoaderPlugin**
Bases: *avocado.core.loader.LoaderError*

Invalid loader plugin

**exception** avocado.core.loader.**LoaderError**
Bases: exceptions.Exception

Loader exception

**exception** avocado.core.loader.**LoaderUnhandledReferenceError**(*unhandled_references*, *plugins*)
Bases: *avocado.core.loader.LoaderError*

Test References not handled by any resolver

**class** avocado.core.loader.**MissingTest**
Bases: object

Class representing reference which failed to be discovered

**class** avocado.core.loader.**NotATest**
Bases: object

Class representing something that is not a test

**class** avocado.core.loader.**TestLoader**(*args*, *extra_params*)
Bases: object

Base for test loader classes

**discover**(*reference*, *which_tests=False*)
Discover (possible) tests from an reference.

**Parameters**

- **reference** (*str*) – the reference to be inspected.
- **which_tests** – Limit tests to be displayed (ALL, AVAILABLE or DEFAULT)

**Returns** a list of test matching the reference as params.

**static get_decorator_mapping**()
Get label mapping for display in test listing.

**Returns** Dict {TestClass: decorator function}

**get_extra_listing**()

**static get_type_label_mapping**()
Get label mapping for display in test listing.

**Returns** Dict {TestClass: 'TEST_LABEL_STRING'}

**name = None**

**class** avocado.core.loader.**TestLoaderProxy**
Bases: object

**clear_plugins**()

**discover**(*references*, *which_tests=False*, *force=None*)
Discover (possible) tests from test references.

**Parameters**

- **references** (*builtin.list*) – a list of tests references; if [] use plugin defaults
- **which_tests** – Limit tests to be displayed (ALL, AVAILABLE or DEFAULT)
- **force** – don't raise an exception when some test references are not resolved to tests.

**Returns** A list of test factories (tuples (TestClass, test_params))

**get_base_keywords**()

**get_decorator_mapping**()

**get_extra_listing**()

**get_type_label_mapping**()

**load_plugins**(*args*)

**load_test**(*test_factory*)
Load test from the test factory.

**Parameters** **test_factory** (*tuple*) – a pair of test class and parameters.

**Returns** an instance of *avocado.core.test.Test*.

**register_plugin**(*plugin*)

avocado.core.loader.**add_loader_options**(*parser*)

avocado.core.loader.**filter_test_tags**(*test_suite*, *filter_by_tags*, *include_empty=False*)
Filter the existing (unfiltered) test suite based on tags

The filtering mechanism is agnostic to test type. It means that if users request filtering by tag and the specific test type does not populate the test tags, it will be considered to have empty tags.

**Parameters**

- **test_suite** (*dict*) – the unfiltered test suite
- **filter_by_tags** (*list of comma separated tags (['foo,bar', 'fast'])*) – the list of tag sets to use as filters
- **include_empty** (*bool*) – if true tests without tags will not be filtered out

## avocado.core.mux module

This file contains mux-enabled implementations of parts useful for creating a custom Varianter plugin.

**class** avocado.core.mux.**Control**(*code*, *value=None*)
Bases: object

Container used to identify node vs. control sequence

**class** avocado.core.mux.**MuxPlugin**
Bases: object

Base implementation of Mux-like Varianter plugin. It should be used as a base class in conjunction with *avocado.core.plugin_interfaces.Varianter*.

**debug = None**

**default_params = None**

**initialize_mux**(*root*, *mux_path*, *debug*)
Initialize the basic values

**Note** We can't use __init__ as this object is intended to be used via dispatcher with no __init__ arguments.

**mux_path = None**

**root = None**

**to_str**(*summary*, *variants*, ***kwargs*)
See *avocado.core.plugin_interfaces.Varianter.to_str()*

**update_defaults**(*defaults*)
See *avocado.core.plugin_interfaces.Varianter.update_defaults()*

**variants = None**

**class** avocado.core.mux.**MuxTree**(*root*)
Bases: object

Object representing part of the tree from the root to leaves or another multiplex domain. Recursively it creates multiplexed variants of the full tree.

**Parameters** **root** – Root of this tree slice

**iter_variants**()
Iterates through variants without verifying the internal filters

:yield all existing variants

**class** avocado.core.mux.**MuxTreeNode**(*name=''*, *value=None*, *parent=None*, *children=None*)
Bases: *avocado.core.tree.TreeNode*

Class for bounding nodes into tree-structure with support for multiplexation

**fingerprint**()

**merge**(*other*)
Merges *other* node into this one without checking the name of the other node. New values are appended, existing values overwritten and unaffected ones are kept. Then all other node children are added as children (recursively they get either appended at the end or merged into existing node in the previous position.

**class** avocado.core.mux.**MuxTreeNodeDebug**(*name=''*, *value=None*, *parent=None*, *children=None*, *srcyaml=None*)
Bases: *avocado.core.mux.MuxTreeNode*, *avocado.core.tree.TreeNodeDebug*

Debug version of TreeNodeDebug :warning: Origin of the value is appended to all values thus it's not suitable for running tests.

**merge**(*other*)

avocado.core.mux.**apply_filters**(*root*, *filter_only=None*, *filter_out=None*)
Apply a set of filters to the tree.

The basic filtering is filter only, which includes nodes, and the filter out rules, that exclude nodes.

Note that filter_out is stronger than filter_only, so if you filter out something, you could not bypass some nodes by using a filter_only rule.

**Parameters**

- **root** – Root node of the multiplex tree.
- **filter_only** – the list of paths which will include nodes.
- **filter_out** – the list of paths which will exclude nodes.

**Returns** the original tree minus the nodes filtered by the rules.

avocado.core.mux.**path_parent**(*path*)
From a given path, return its parent path.

**Parameters** **path** – the node path as string.

**Returns** the parent path as string.

## avocado.core.output module

Manages output and logging in avocado applications.

avocado.core.output.**BUILTIN_STREAMS = {'test': 'test output', 'debug': 'tracebacks and other debugging info', 'app':**
Builtin special keywords to enable set of logging streams

avocado.core.output.**BUILTIN_STREAM_SETS = {'all': 'all builtin streams', 'none': 'disables regular output (leaving o**
Groups of builtin streams

**class** avocado.core.output.**FilterInfoAndLess**(*name=''*)
Bases: logging.Filter

Initialize a filter.

Initialize with the name of the logger which, together with its children, will have its events allowed through the filter. If no name is specified, allow every event.

**filter**(*record*)

**class** avocado.core.output.**FilterWarnAndMore**(*name=''*)
Bases: logging.Filter

Initialize a filter.

Initialize with the name of the logger which, together with its children, will have its events allowed through the filter. If no name is specified, allow every event.

**filter**(*record*)

avocado.core.output.**LOG_JOB = <logging.Logger object>**
Pre-defined Avocado job/test logger

avocado.core.output.**LOG_UI = <logging.Logger object>**
Pre-defined Avocado human UI logger

**class** avocado.core.output.**LoggingFile**(*prefix=''*, *level=10*, *logger=[<logging.RootLogger object>]*)
Bases: object

File-like object that will receive messages pass them to logging.

Constructor. Sets prefixes and which logger is going to be used.

:param prefix - The prefix for each line logged by this object.

**add_logger**(*logger*)

**flush**()

**isatty**()

**rm_logger**(*logger*)

**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** avocado.core.output.**MemStreamHandler**(*stream=None*)
Bases: logging.StreamHandler

Handler that stores all records in self.log (shared in all instances)

Initialize the handler.

If stream is not specified, sys.stderr is used.

**emit**(*record*)

**flush**()
This is in-mem object, it does not require flushing

**log = []**

**exception** avocado.core.output.**PagerNotFoundError**
Bases: exceptions.Exception

**class** avocado.core.output.**Paginator**
Bases: object

Paginator that uses less to display contents on the terminal.

Contains cleanup handling for when user presses 'q' (to quit less).

**close**()

**flush**()

**write**(*msg*)

**class** avocado.core.output.**ProgressStreamHandler**(*stream=None*)
Bases: logging.StreamHandler

Handler class that allows users to skip new lines on each emission.

Initialize the handler.

If stream is not specified, sys.stderr is used.

**emit**(*record*)

avocado.core.output.**STD_OUTPUT = <avocado.core.output.StdOutput object>**
Allows modifying the sys.stdout/sys.stderr

**class** avocado.core.output.**StdOutput**
Bases: object

Class to modify sys.stdout/sys.stderr

**close**()
Enable original sys.stdout/sys.stderr and cleanup

**enable_outputs**()
Enable sys.stdout/sys.stderr (either with 2 streams or with paginator)

**enable_paginator**()
Enable paginator

**enable_stderr**()
Enable sys.stderr and disable sys.stdout

**fake_outputs**()
Replace sys.stdout/sys.stderr with in-memory-objects

**print_records**()
Prints all stored messages as they occurred into streams they were produced for.

**records = []**
List of records of stored output when stdout/stderr is disabled

avocado.core.output.**TERM_SUPPORT = <avocado.core.output.TermSupport object>**
Transparently handles colored terminal, when one is used

**class** avocado.core.output.**TermSupport**
Bases: object

**COLOR_BLUE = '\x1b[94m'**

**COLOR_DARKGREY = '\x1b[90m'**

**COLOR_GREEN = '\x1b[92m'**

**COLOR_RED = '\x1b[91m'**

**COLOR_YELLOW = '\x1b[93m'**

**CONTROL_END = '\x1b[0m'**

**ESCAPE_CODES = ['\x1b[94m', '\x1b[92m', '\x1b[93m', '\x1b[91m', '\x1b[90m', '\x1b[0m', '\x1b[1D', '\x1b[1C']**
Class to help applications to colorize their outputs for terminals.

This will probe the current terminal and colorize output only if the stdout is in a tty or the terminal type is recognized.

**MOVE_BACK = '\x1b[1D'**

**MOVE_FORWARD = '\x1b[1C'**

**disable**()
Disable colors from the strings output by this class.

**error_str**()
Print a error string (red colored).

If the output does not support colors, just return the original string.

**fail_header_str**(*msg*)
Print a fail header string (red colored).

If the output does not support colors, just return the original string.

**fail_str**()
Print a fail string (red colored).

If the output does not support colors, just return the original string.

**header_str**(*msg*)
Print a header string (blue colored).

If the output does not support colors, just return the original string.

**healthy_str**(*msg*)
Print a healthy string (green colored).

If the output does not support colors, just return the original string.

**interrupt_str**()
Print an interrupt string (red colored).

If the output does not support colors, just return the original string.

**partial_str**(*msg*)
Print a string that denotes partial progress (yellow colored).

If the output does not support colors, just return the original string.

**pass_str**()
Print a pass string (green colored).

If the output does not support colors, just return the original string.

**skip_str**()
Print a skip string (yellow colored).

If the output does not support colors, just return the original string.

**warn_header_str**(*msg*)
Print a warning header string (yellow colored).

If the output does not support colors, just return the original string.

**warn_str**()
Print an warning string (yellow colored).

If the output does not support colors, just return the original string.

**class** avocado.core.output.**Throbber**
Bases: object

Produces a spinner used to notify progress in the application UI.

**MOVES = ['', '', '', '']**

**STEPS = ['-', '\\', '|', '/']**

**render**()

avocado.core.output.**add_log_handler**(*logger*, *klass=<class 'logging.StreamHandler'>*, *stream=<open file '<stdout>'*, *mode 'w'>*, *level=20*, *fmt='%(name)s: %(message)s'*)
Add handler to a logger.

**Parameters**

- **logger_name** – the name of a logging.Logger instance, that is, the parameter to logging.getLogger()
- **klass** – Handler class (defaults to logging.StreamHandler)
- **stream** – Logging stream, to be passed as an argument to klass (defaults to sys.stdout)
- **level** – Log level (defaults to *INFO*')

- **fmt** – Logging format (defaults to `%(name)s: %(message)s`)

avocado.core.output.**disable_log_handler**(*logger*)

avocado.core.output.**early_start**()
Replace all outputs with in-memory handlers

avocado.core.output.**log_plugin_failures**(*failures*)
Log in the application UI failures to load a set of plugins

**Parameters** **failures** – a list of load failures, usually coming from a *avocado.core.dispatcher.Dispatcher* attribute *load_failures*

avocado.core.output.**reconfigure**(*args*)
Adjust logging handlers accordingly to app args and re-log messages.

## avocado.core.parser module

Avocado application command line parsing.

**class** avocado.core.parser.**ArgumentParser**(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *version=None*, *parents=[]*, *formatter_class=<class 'argparse.HelpFormatter'>*, *prefix_chars='-'*, *fromfile_prefix_chars=None*, *argument_default=None*, *conflict_handler='error'*, *add_help=True*)
Bases: argparse.ArgumentParser

Class to override argparse functions

**error**(*message*)

**class** avocado.core.parser.**FileOrStdoutAction**(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)
Bases: argparse.Action

Controls claiming the right to write to the application standard output

**class** avocado.core.parser.**Parser**
Bases: object

Class to Parse the command line arguments.

**finish**()
Finish the process of parsing arguments.

Side effect: set the final value for attribute *args*.

**start**()
Start to parsing arguments.

At the end of this method, the support for subparsers is activated. Side effect: update attribute *args* (the namespace).

## avocado.core.plugin_interfaces module

**class** avocado.core.plugin_interfaces.**CLI**
Bases: *avocado.core.plugin_interfaces.Plugin*

Base plugin interface for adding options (non-commands) to the command line

Plugins that want to add extra options to the core command line application or to sub commands should use the 'avocado.plugins.cli' namespace.

**configure**(*parser*)
Configures the command line parser with options specific to this plugin

**run**(*args*)
Execute any action the plugin intends.

Example of action may include activating a special features upon finding that the requested command line options were set by the user.

Note: this plugin class is not intended for adding new commands, for that please use *CLICmd*.

**class** avocado.core.plugin_interfaces.**CLICmd**
Bases: *avocado.core.plugin_interfaces.Plugin*

Base plugin interface for adding new commands to the command line app

Plugins that want to add extensions to the run command should use the 'avocado.plugins.cli.cmd' namespace.

**configure**(*parser*)
Lets the extension add command line options and do early configuration

By default it will register its *name* as the command name and give its *description* as the help message.

**description = None**

**name = None**

**run**(*args*)
Entry point for actually running the command

**class** avocado.core.plugin_interfaces.**JobPost**
Bases: *avocado.core.plugin_interfaces.Plugin*

Base plugin interface for adding actions after a job runs

Plugins that want to add actions to be run after a job runs, should use the 'avocado.plugins.job.prepost' namespace and implement the defined interface.

**post**(*job*)
Entry point for actually running the post job action

**class** avocado.core.plugin_interfaces.**JobPostTests**
Bases: *avocado.core.plugin_interfaces.Plugin*

Base plugin interface for adding actions after a job runs tests

Plugins using this interface will run at the a time equivalent to plugins using the *JobPost* interface, that is, at *avocado.core.job.Job.post_tests()*. This is because *JobPost* based plugins will eventually be modified to really run after the job has finished, and not after it has run tests.

**post_tests**(*job*)
Entry point for job running actions after the tests execution

**class** avocado.core.plugin_interfaces.**JobPre**
Bases: *avocado.core.plugin_interfaces.Plugin*

Base plugin interface for adding actions before a job runs

Plugins that want to add actions to be run before a job runs, should use the 'avocado.plugins.job.prepost' namespace and implement the defined interface.

**pre** (*job*)
Entry point for actually running the pre job action

**class** avocado.core.plugin_interfaces.**JobPreTests**
Bases: *avocado.core.plugin_interfaces.Plugin*

Base plugin interface for adding actions before a job runs tests

This interface looks similar to *JobPre*, but it's intended to be called at a very specific place, that is, between *avocado.core.job.Job.create_test_suite()* and *avocado.core.job.Job.run_tests()*.

**pre_tests** (*job*)
Entry point for job running actions before tests execution

**class** avocado.core.plugin_interfaces.**Plugin**
Bases: object

**class** avocado.core.plugin_interfaces.**Result**
Bases: *avocado.core.plugin_interfaces.Plugin*

**render** (*result*, *job*)
Entry point with method that renders the result

This will usually be used to write the result to a file or directory.

**Parameters**

- **result** (*avocado.core.result.Result*) – the complete job result
- **job** (*avocado.core.job.Job*) – the finished job for which a result will be written

**class** avocado.core.plugin_interfaces.**ResultEvents**
Bases: *avocado.core.plugin_interfaces.JobPreTests*, *avocado.core.plugin_interfaces.JobPostTests*

Base plugin interface for event based (stream-able) results

Plugins that want to add actions to be run after a job runs, should use the 'avocado.plugins.result_events' namespace and implement the defined interface.

**end_test** (*result*, *state*)
Event triggered when a test finishes running

**start_test** (*result*, *state*)
Event triggered when a test starts running

**test_progress** (*progress=False*)
Interface to notify progress (or not) of the running test

**class** avocado.core.plugin_interfaces.**Varianter**
Bases: *avocado.core.plugin_interfaces.Plugin*

Base plugin interface for producing test variants usually from cmd line options

**to_str** (*summary*, *variants*, ***kwargs*)
Return human readable representation

The summary/variants accepts verbosity where 0 means silent and maximum is up to the plugin.

**Parameters**

- **summary** – How verbose summary to output (int)
- **variants** – How verbose list of variants to output (int)

- **kwargs** – Other free-form arguments

**Return type** str

**update_defaults**(*defaults*)
Add default values

**Note** Those values should not be part of the variant_id

## avocado.core.result module

Contains the Result class, used for result accounting.

**class** avocado.core.result.**Result**(*job*)
Bases: object

Result class, holder for job (and its tests) result information.

Creates an instance of Result.

**Parameters** **job** – an instance of *avocado.core.job.Job*.

**check_test**(*state*)
Called once for a test to check status and report.

**Parameters** **test** – A dict with test internal state

**end_test**(*state*)
Called when the given test has been run.

**Parameters** **state** (*dict*) – result of *avocado.core.test.Test.get_state*.

**end_tests**()
Called once after all tests are executed.

**start_test**(*state*)
Called when the given test is about to run.

**Parameters** **state** (*dict*) – result of *avocado.core.test.Test.get_state*.

## avocado.core.runner module

Test runner module.

avocado.core.runner.**TIMEOUT_PROCESS_ALIVE = 60**
when test reported status but the process did not finish

avocado.core.runner.**TIMEOUT_PROCESS_DIED = 10**
when the process died but the status was not yet delivered

avocado.core.runner.**TIMEOUT_TEST_INTERRUPTED = 1**
when test was interrupted (ctrl+c/timeout)

**class** avocado.core.runner.**TestRunner**(*job*, *result*)
Bases: object

A test runner class that displays tests results.

Creates an instance of TestRunner class.

**Parameters**

- **job** – an instance of *avocado.core.job.Job*.

- **result** – an instance of *avocado.core.result.Result*

**DEFAULT_TIMEOUT = 86400**

**run_suite**(*test_suite*, *variants*, *timeout=0*, *replay_map=None*)
Run one or more tests and report with test result.

**Parameters**

- **test_suite** – a list of tests to run.
- **variants** – A varianter iterator to produce test params.
- **timeout** – maximum amount of time (in seconds) to execute.

**Returns** a set with types of test failures.

**run_test**(*test_factory*, *queue*, *summary*, *job_deadline=0*)
Run a test instance inside a subprocess.

**Parameters**

- **test_factory** (tuple of *avocado.core.test.Test* and dict.) – Test factory (test class and parameters).
- **queue** (*:class`multiprocessing.Queue` instance.*) – Multiprocess queue.
- **summary** (*set.*) – Contains types of test failures.
- **job_deadline** (*int.*) – Maximum time to execute.

**class** avocado.core.runner.**TestStatus**(*job*, *queue*)
Bases: object

Test status handler

**Parameters**

- **job** – Associated job
- **queue** – test message queue

**early_status**
Get early status

**finish**(*proc*, *started*, *step*, *deadline*, *result_dispatcher*)
Wait for the test process to finish and report status or error status if unable to obtain the status till deadline.

**Parameters**

- **proc** – The test's process
- **started** – Time when the test started
- **first** – Delay before first check
- **step** – Step between checks for the status
- **deadline** – Test execution deadline
- **result_dispatcher** – Result dispatcher (for test_progress notifications)

**wait_for_early_status**(*proc*, *timeout*)
Wait until early_status is obtained :param proc: test process :param timeout: timeout for early_state :raise exceptions.TestError: On timeout/error

avocado.core.runner.**add_runner_failure**(*test_state*, *new_status*, *message*)
Append runner failure to the overall test status.

**Parameters**

- **test_state** – Original test state (dict)
- **new_status** – New test status (PASS/FAIL/ERROR/INTERRUPTED/...)
- **message** – The error message

## avocado.core.safeloader module

Safe (AST based) test loader module utilities

avocado.core.safeloader.**DOCSTRING_DIRECTIVE_RE_RAW = '\\s*:avocado:[ \\t]+([a-zA-Z0-9]+?[a-zA-Z0-9_:,\\=]*)**

Gets the docstring directive value from a string. Used to tweak test behavior in various ways

avocado.core.safeloader.**check_docstring_directive**(*docstring*, *directive*)
Checks if there's a given directive in a given docstring

**Return type** bool

avocado.core.safeloader.**find_class_and_methods**(*path*, *method_pattern=None*, *base_class=None*)
Attempts to find methods names from a given Python source file

**Parameters**

- **path** (*str*) – path to a Python source code file
- **method_pattern** – compiled regex to match against method name
- **base_class** (*str or None*) – only consider classes that inherit from a given base class (or classes that inherit from any class if None is given)

avocado.core.safeloader.**get_docstring_directives**(*docstring*)
Returns the values of the avocado docstring directives

**Parameters** **docstring** (*str*) – the complete text used as documentation

**Return type** builtin.list

avocado.core.safeloader.**get_docstring_directives_tags**(*docstring*)
Returns the test categories based on a *:avocado: tags=category* docstring

**Return type** set

avocado.core.safeloader.**modules_imported_as**(*module*)
Returns a mapping of imported module names whether using aliases or not

The goal of this utility function is to return the name of the import as used in the rest of the module, whether an aliased import was used or not.

For code such as:

```
>>> import foo as bar
```

This function should return {"foo": "bar"}

And for code such as:

```
>>> import foo
```

It should return {"foo": "foo"}

Please note that only global level imports are looked at. If there are imports defined, say, inside functions or class definitions, they will not be seen by this function.

**Parameters** **module** (_ast.Module) – module, as parsed by ast.parse()

**Returns** a mapping of names {<realname>: <alias>} of modules imported

**Return type** dict

## avocado.core.settings module

Reads the avocado settings from a .ini file (from python ConfigParser).

**exception** avocado.core.settings.**ConfigFileNotFound**(*path_list*)
Bases: *avocado.core.settings.SettingsError*

Error thrown when the main settings file could not be found.

**class** avocado.core.settings.**Settings**(*config_path=None*)
Bases: object

Simple wrapper around ConfigParser, with a key type conversion available.

Constructor. Tries to find the main settings file and load it.

**Parameters** **config_path** – Path to a config file. Useful for unittesting.

**get_value**(*section*, *key*, *key_type=<type 'str'>*, *default=<object object>*, *allow_blank=False*)
Get value from key in a given config file section.

**Parameters**

- **section** (*str*) – Config file section.
- **key** (*str*) – Config file key, relative to section.
- **key_type** (either string based names representing types, including *str*, *int*, *float*, *bool*, *list* and *path*, or the types themselves limited to str, int, float, bool and list.) – Type of key.
- **default** – Default value for the key, if none found.
- **allow_blank** – Whether an empty value for the key is allowed.

**Returns** value, if one available in the config. default value, if one provided.

**Raises** SettingsError, in case key is not set and no default was provided.

**no_default = <object object>**

**process_config_path**(*pth*)

**exception** avocado.core.settings.**SettingsError**
Bases: exceptions.Exception

Base settings error.

**exception** avocado.core.settings.**SettingsValueError**
Bases: *avocado.core.settings.SettingsError*

Error thrown when we could not convert successfully a key to a value.

avocado.core.settings.**convert_value_type**(*value*, *value_type*)
Convert a string value to a given value type.

**Parameters**

- **value** (*str.*) – Value we want to convert.
- **value_type** (*str or type.*) – Type of the value we want to convert.

**Returns** Converted value type.

**Return type** Dependent on value_type.

**Raise** TypeError, in case it was not possible to convert values.

## avocado.core.status module

Maps the different status strings in avocado to booleans.

This is used by methods and functions to return a cut and dry answer to whether a test or a job in avocado PASSed or FAILed.

## avocado.core.sysinfo module

**class** avocado.core.sysinfo.**Collectible**(*logf*)
Bases: object

Abstract class for representing collectibles by sysinfo.

**readline**(*logdir*)
Read one line of the collectible object.

**Parameters** **logdir** – Path to a log directory.

**class** avocado.core.sysinfo.**Command**(*cmd*, *logf=None*, *compress_log=False*)
Bases: *avocado.core.sysinfo.Collectible*

Collectible command.

**Parameters**

- **cmd** – String with the command.
- **logf** – Basename of the file where output is logged (optional).
- **compress_logf** – Wether to compress the output of the command.

**run**(*logdir*)
Execute the command as a subprocess and log its output in logdir.

**Parameters** **logdir** – Path to a log directory.

**class** avocado.core.sysinfo.**Daemon**(*cmd*, *logf=None*, *compress_log=False*)
Bases: *avocado.core.sysinfo.Command*

Collectible daemon.

**Parameters**

- **cmd** – String with the daemon command.
- **logf** – Basename of the file where output is logged (optional).
- **compress_logf** – Wether to compress the output of the command.

**run**(*logdir*)
Execute the daemon as a subprocess and log its output in logdir.

**Parameters** **logdir** – Path to a log directory.

**stop**()
Stop daemon execution.

**class** avocado.core.sysinfo.**JournalctlWatcher**(*logf=None*)
Bases: *avocado.core.sysinfo.Collectible*

Track the content of systemd journal into a compressed file.

**Parameters** **logf** – Basename of the file where output is logged (optional).

**run**(*logdir*)

**class** avocado.core.sysinfo.**LogWatcher**(*path*, *logf=None*)
Bases: *avocado.core.sysinfo.Collectible*

Keep track of the contents of a log file in another compressed file.

This object is normally used to track contents of the system log (/var/log/messages), and the outputs are gzipped since they can be potentially large, helping to save space.

**Parameters**

- **path** – Path to the log file.
- **logf** – Basename of the file where output is logged (optional).

**run**(*logdir*)
Log all of the new data present in the log file.

**class** avocado.core.sysinfo.**Logfile**(*path*, *logf=None*)
Bases: *avocado.core.sysinfo.Collectible*

Collectible system file.

**Parameters**

- **path** – Path to the log file.
- **logf** – Basename of the file where output is logged (optional).

**run**(*logdir*)
Copy the log file to the appropriate log dir.

**Parameters** **logdir** – Log directory which the file is going to be copied to.

**class** avocado.core.sysinfo.**SysInfo**(*basedir=None*, *log_packages=None*, *profiler=None*)
Bases: object

Log different system properties at some key control points:

•start_job

•start_test

•end_test

•end_job

Set sysinfo collectibles.

**Parameters**

- **basedir** – Base log dir where sysinfo files will be located.

- **log_packages** – Whether to log system packages (optional because logging packages is a costly operation). If not given explicitly, tries to look in the config files, and if not found, defaults to False.
- **profiler** – Wether to use the profiler. If not given explicitly, tries to look in the config files.

**add_cmd**(*cmd*, *hook*)
Add a command collectible.

**Parameters**

- **cmd** – Command to log.
- **hook** – In which hook this cmd should be logged (start job, end job).

**add_file**(*filename*, *hook*)
Add a system file collectible.

**Parameters**

- **filename** – Path to the file to be logged.
- **hook** – In which hook this file should be logged (start job, end job).

**add_watcher**(*filename*, *hook*)
Add a system file watcher collectible.

**Parameters**

- **filename** – Path to the file to be logged.
- **hook** – In which hook this watcher should be logged (start job, end job).

**end_job_hook**()
Logging hook called whenever a job finishes.

**end_test_hook**()
Logging hook called after a test finishes.

**start_job_hook**()
Logging hook called whenever a job starts.

**start_test_hook**()
Logging hook called before a test starts.

avocado.core.sysinfo.**collect_sysinfo**(*args*)
Collect sysinfo to a base directory.

**Parameters** **args** – argparse.Namespace object with command line params.

## avocado.core.test module

Contains the base test implementation, used as a base for the actual framework tests.

avocado.core.test.**COMMON_TMPDIR_NAME = 'AVOCADO_TESTS_COMMON_TMPDIR'**
Environment variable used to store the location of a temporary directory which is preserved across all tests execution (usually in one job)

**class** avocado.core.test.**DryRunTest**(**args*, ***kwargs*)
Bases: *avocado.core.test.MockingTest*

Fake test which logs itself and reports as CANCEL

This class substitutes other classes. Let's just ignore the remaining arguments and only set the ones supported by avocado.Test

**setUp**()

**class** avocado.core.test.**ExternalRunnerTest**(*name*, *params=None*, *base_logdir=None*, *job=None*, *external_runner=None*)
Bases: *avocado.core.test.SimpleTest*

**filename**

**test**()

**class** avocado.core.test.**MockingTest**(**args*, ***kwargs*)
Bases: *avocado.core.test.Test*

Class intended as generic substitute for avocado tests which will not be executed for some reason. This class is expected to be overridden by specific reason-oriented sub-classes.

This class substitutes other classes. Let's just ignore the remaining arguments and only set the ones supported by avocado.Test

**test**()

**exception** avocado.core.test.**NameNotTestNameError**
Bases: exceptions.Exception

The given test name is not a TestName instance

With the introduction of *avocado.core.test.TestName*, it's not allowed to use other types as the name parameter to a test instance. This exception is raised when this is attempted.

**class** avocado.core.test.**ReplaySkipTest**(**args*, ***kwargs*)
Bases: *avocado.core.test.MockingTest*

Skip test due to job replay filter.

This test is skipped due to a job replay filter. It will never have a chance to execute.

This class substitutes other classes. Let's just ignore the remaining arguments and only set the ones supported by avocado.Test

**test**(**args*, ***kwargs*)

**class** avocado.core.test.**SimpleTest**(*name*, *params=None*, *base_logdir=None*, *job=None*)
Bases: *avocado.core.test.Test*

Run an arbitrary command that returns either 0 (PASS) or !=0 (FAIL).

**execute_cmd**()
Run the executable, and log its detailed execution.

**filename**
Returns the name of the file (path) that holds the current test

**re_avocado_log = <_sre.SRE_Pattern object at 0x24cf510>**

**test**()
Run the test and postprocess the results

**class** avocado.core.test.**Test**(*methodName='test'*, *name=None*, *params=None*, *base_logdir=None*, *job=None*, *runner_queue=None*)
Bases: unittest.case.TestCase

Base implementation for the test class.

You'll inherit from this to write your own tests. Typically you'll want to implement setUp(), test*() and tearDown() methods on your own tests.

Initializes the test.

**Parameters**

- **methodName** – Name of the main method to run. For the sake of compatibility with the original unittest class, you should not set this.
- **name** (*avocado.core.test.TestName*) – Pretty name of the test name. For normal tests, written with the avocado API, this should not be set. This is reserved for internal Avocado use, such as when running random executables as tests.
- **base_logdir** – Directory where test logs should go. If None provided, it'll use `avocado.data_dir.create_job_logs_dir()`.
- **job** – The job that this test is part of.

**Raises** *avocado.core.test.NameNotTestNameError*

**basedir**
The directory where this test (when backed by a file) is located at

**cache_dirs = None**

**cancel**(*message=None*)
Cancels the test.

This method is expected to be called from the test method, not anywhere else, since by definition, we can only cancel a test that is currently under execution. If you call this method outside the test method, avocado will mark your test status as ERROR, and instruct you to fix your test in the error message.

**Parameters** **message** (*str*) – an optional message that will be recorded in the logs

**datadir**
Returns the path to the directory that contains test data files

**default_params = {}**
*default_params* will be deprecated by the end of 2017.

**error**(*message=None*)
Errors the currently running test.

After calling this method a test will be terminated and have its status as ERROR.

**Parameters** **message** (*str*) – an optional message that will be recorded in the logs

**fail**(*message=None*)
Fails the currently running test.

After calling this method a test will be terminated and have its status as FAIL.

**Parameters** **message** (*str*) – an optional message that will be recorded in the logs

**fail_class**

**fail_reason**

**fetch_asset**(*name*, *asset_hash=None*, *algorithm='sha1'*, *locations=None*, *expire=None*)
Method o call the utils.asset in order to fetch and asset file supporting hash check, caching and multiple locations.

**Parameters**

- **name** – the asset filename or URL

- **asset_hash** – asset hash (optional)
- **algorithm** – hash algorithm (optional, defaults to sha1)
- **locations** – list of URLs from where the asset can be fetched (optional)
- **expire** – time for the asset to expire

**Raises** **EnvironmentError** – When it fails to fetch the asset

**Returns** asset file local path

**filename**
Returns the name of the file (path) that holds the current test

**get_state**()
Serialize selected attributes representing the test state

**Returns** a dictionary containing relevant test state data

**Return type** dict

**job**
The job this test is associated with

**log**
The enhanced test log

**logdir**
Path to this test's logging dir

**logfile**
Path to this test's main *debug.log* file

**name**
The test name (TestName instance)

**outputdir**
Directory available to test writers to attach files to the results

**params**
Parameters of this test (AvocadoParam instance)

**report_state**()
Send the current test state to the test runner process

**run_avocado**()
Wraps the run method, for execution inside the avocado runner.

**Result** Unused param, compatibility with `unittest.TestCase`.

**runner_queue**
The communication channel between test and test runner

**running**
Whether this test is currently being executed

**set_runner_queue**(*runner_queue*)
Override the runner_queue

**skip**(*message=None*)
Skips the currently running test.

This method should only be called from a test's setUp() method, not anywhere else, since by definition, if a test gets to be executed, it can't be skipped anymore. If you call this method outside setUp(), avocado will mark your test status as ERROR, and instruct you to fix your test in the error message.

**Parameters** **message** (*str*) – an optional message that will be recorded in the logs

**srcdir = None**

**status**
The result status of this test

**teststmpdir**
Returns the path of the temporary directory that will stay the same for all tests in a given Job.

**time_elapsed = -1**
duration of the test execution (always recalculated from time_end - time_start

**time_end = -1**
(unix) time when the test finished (could be forced from test)

**time_start = -1**
(unix) time when the test started (could be forced from test)

**timeout = None**
Test timeout (the timeout from params takes precedence)

**traceback**

**whiteboard = ''**
Arbitrary string which will be stored in *$logdir/whiteboard* location when the test finishes.

**workdir = None**

**class** avocado.core.test.**TestError**(**args*, ***kwargs*)
Bases: *avocado.core.test.Test*

Generic test error.

**test**()

**class** avocado.core.test.**TestName**(*uid*, *name*, *variant=None*, *no_digits=None*)
Bases: object

Test name representation

Test name according to avocado specification

**Parameters**

- **uid** – unique test id (within the job)
- **name** – test name (identifies the executed test)
- **variant** – variant id
- **no_digits** – number of digits of the test uid

**str_filesystem**()
File-system friendly representation of the test name

**class** avocado.core.test.**TimeOutSkipTest**(**args*, ***kwargs*)
Bases: *avocado.core.test.MockingTest*

Skip test due job timeout.

This test is skipped due a job timeout. It will never have a chance to execute.

This class substitutes other classes. Let's just ignore the remaining arguments and only set the ones supported by avocado.Test

**test**(**args*, ***kwargs*)

## avocado.core.tree module

Tree data structure with nodes.

This tree structure (Tree drawing code) was inspired in the base tree data structure of the ETE 2 project:

http://pythonhosted.org/ete2/

A library for analysis of phylogenetics trees.

Explicit permission has been given by the copyright owner of ETE 2 Jaime Huerta-Cepas <jhcepas@gmail.com> to take ideas/use snippets from his original base tree code and re-license under GPLv2+, given that GPLv3 and GPLv2 (used in some avocado files) are incompatible.

**class** avocado.core.tree.**FilterSet**
Bases: set

Set of filters in standardized form

**add**(*item*)

**update**(*items*)

**class** avocado.core.tree.**OutputList**(*values*, *nodes*, *yamls*)
Bases: list

List with some debug info

**class** avocado.core.tree.**OutputValue**(*value*, *node*, *srcyaml*)
Bases: object

Ordinary value with some debug info

**class** avocado.core.tree.**TreeEnvironment**
Bases: dict

TreeNode environment with values, origins and filters

**copy**()

**class** avocado.core.tree.**TreeNode**(*name=''*, *value=None*, *parent=None*, *children=None*)
Bases: object

Class for bounding nodes into tree-structure.

**add_child**(*node*)
Append node as child. Nodes with the same name gets merged into the existing position.

**detach**()
Detach this node from parent

**environment**
Node environment (values + preceding envs)

**fingerprint**()
Reports string which represents the value of this node.

**get_environment**()
Get node environment (values + preceding envs)

**get_leaves**()
Get list of leaf nodes

**get_node**(*path*, *create=False*)

**Parameters**

- **path** – Path of the desired node (relative to this node)
- **create** – Create the node (and intermediary ones) when not present

**Returns** the node associated with this path

**Raises** **ValueError** – When path doesn't exist and create not set

**get_parents**()
Get list of parent nodes

**get_path**(*sep='/'*)
Get node path

**get_root**()
Get root of this tree

**is_leaf**
Is this a leaf node?

**iter_children_preorder**()
Iterate through children

**iter_leaves**()
Iterate through leaf nodes

**iter_parents**()
Iterate through parent nodes to root

**merge**(*other*)
Merges *other* node into this one without checking the name of the other node. New values are appended, existing values overwritten and unaffected ones are kept. Then all other node children are added as children (recursively they get either appended at the end or merged into existing node in the previous position.

**parents**
List of parent nodes

**path**
Node path

**root**
Root of this tree

**set_environment_dirty**()
Set the environment cache dirty. You should call this always when you query for the environment and then change the value or structure. Otherwise you'll get the old environment instead.

**class** avocado.core.tree.**TreeNodeDebug**(*name=''*, *value=None*, *parent=None*, *children=None*, *srcyaml=None*)
Bases: *avocado.core.tree.TreeNode*

Debug version of TreeNodeDebug :warning: Origin of the value is appended to all values thus it's not suitable for running tests.

**merge**(*other*)
Override origin with the one from other tree. Updated/Newly set values are going to use this location as origin.

**class** avocado.core.tree.**ValueDict**(*srcyaml*, *node*, *values*)
Bases: dict

Dict which stores the origin of the items

**iteritems**()
Slower implementation with the use of __getitem__

avocado.core.tree.**get_named_tree_cls**(*path*, *klass=<class 'avocado.core.tree.TreeNodeDebug'>*)
Return TreeNodeDebug class with hardcoded yaml path

avocado.core.tree.**tree_view**(*root*, *verbose=None*, *use_utf8=None*)
Generate tree-view of the given node :param root: root node :param verbose: verbosity (0, 1, 2, 3) :param use_utf8: Use utf-8 encoding (None=autodetect) :return: string representing this node's tree structure

## avocado.core.varianter module

Multiplex and create variants.

**class** avocado.core.varianter.**AvocadoParam**(*leaves*, *name*)
Bases: object

This is a single slice params. It can contain multiple leaves and tries to find matching results.

**Parameters**

- **leaves** – this slice's leaves
- **name** – this slice's name (identifier used in exceptions)

**get_or_die**(*path*, *key*)
Get a value or raise exception if not present :raise NoMatchError: When no matches :raise KeyError: When value is not certain (multiple matches)

**iteritems**()
Very basic implementation which iterates through __ALL__ params, which generates lots of duplicate entries due to inherited values.

**str_leaves_variant**
String with identifier and all params

**class** avocado.core.varianter.**AvocadoParams**(*leaves*, *test_id*, *mux_path*, *default_params*)
Bases: object

Params object used to retrieve params from given path. It supports absolute and relative paths. For relative paths one can define multiple paths to search for the value. It contains compatibility wrapper to act as the original avocado Params, but by special usage you can utilize the new API. See get() docstring for details.

You can also iterate through all keys, but this can generate quite a lot of duplicate entries inherited from ancestor nodes. It shouldn't produce false values, though.

In this version each new "get()" call is logged into avocado.LOG_JOB. This is subject of change (separate file, perhaps)

**Parameters**

- **leaves** – List of TreeNode leaves defining current variant
- **test_id** – test id
- **mux_path** – list of entry points
- **default_params** – dict of params used when no matches found

**Note:** *default_params* will be deprecated by the end of 2017.

**get**(*key*, *path=None*, *default=None*)
Retrieve value associated with key from params :param key: Key you're looking for :param path: namespace ['*'] :param default: default value when not found :raise KeyError: In case of multiple different values (params clash)

**iteritems**()
Iterate through all available params and yield origin, key and value of each unique value.

**log**(*key*, *path*, *default*, *value*)
Predefined format for displaying params query

**objects**(*key*, *path=None*)
Return the names of objects defined using a given key.

**Parameters** **key** – The name of the key whose value lists the objects (e.g. 'nics').

**exception** avocado.core.varianter.**NoMatchError**
Bases: exceptions.KeyError

**class** avocado.core.varianter.**Varianter**(*debug=False*)
Bases: object

This object takes care of producing test variants

**Parameters** **debug** – Store whether this instance should debug the mux

**Note** people need to check whether mux uses debug and reflect that in order to provide the right results.

**add_default_param**(*name*, *key*, *value*, *path=None*)
Stores the path/key/value into default params

This allow injecting default arguments which are mainly intended for machine/os-related params. It should not affect the test results and by definition it should not affect the variant id.

**Parameters**

- **name** – Name of the component which injects this param
- **key** – Key to which we'd like to assign the value
- **value** – The key's value
- **path** – Optional path to the node to which we assign the value, by default '/'.

**get_number_of_tests**(*test_suite*)

**Returns** overall number of tests * number of variants

**is_parsed**()
Reports whether the varianter was already parsed

**itertests**()
Yields all variants of all plugins

**The variant is defined as dictionary with at least:**

- variant_id - name of the current variant
- **variant - AvocadoParams-compatible variant (usually a list of** TreeNodes but dict or simply None are also possible values)
- mux_path - default path(s)

:yield variant

**parse** (*args*)
Apply options defined on the cmdline and initialize the plugins.

**Parameters** **args** – Parsed cmdline arguments

**to_str** (*summary=0*, *variants=0*, ***kwargs*)
Return human readable representation

The summary/variants accepts verbosity where 0 means do not display at all and maximum is up to the plugin.

**Parameters**

- **summary** – How verbose summary to output (int)
- **variants** – How verbose list of variants to output (int)
- **kwargs** – Other free-form arguments

**Return type** str

### avocado.core.version module

### Module contents

## Extension (plugin) APIs

Extension APIs that may be of interest to plugin writers.

### Submodules

### avocado.plugins.archive module

Result Archive Plugin

**class** avocado.plugins.archive.**Archive**
Bases: *avocado.core.plugin_interfaces.Result*

**description = 'Result archive (ZIP) support'**

**name = 'zip_archive'**

**render** (*result*, *job*)

**class** avocado.plugins.archive.**ArchiveCLI**
Bases: *avocado.core.plugin_interfaces.CLI*

**configure** (*parser*)

**description = 'Result archive (ZIP) support to run command'**

**name = 'zip_archive'**

**run** (*args*)

## avocado.plugins.config module

**class** avocado.plugins.config.**Config**
Bases: *avocado.core.plugin_interfaces.CLICmd*

Implements the avocado 'config' subcommand

**configure** (*parser*)

**description = 'Shows avocado config keys'**

**name = 'config'**

**run** (*args*)

## avocado.plugins.diff module

Job Diff

**class** avocado.plugins.diff.**Diff**
Bases: *avocado.core.plugin_interfaces.CLICmd*

Implements the avocado 'diff' subcommand

**configure** (*parser*)
Add the subparser for the diff action.

**Parameters** **parser** – Main test runner parser.

**description = 'Shows the difference between 2 jobs.'**

**name = 'diff'**

**run** (*args*)

## avocado.plugins.distro module

avocado.plugins.distro.**DISTRO_PKG_INFO_LOADERS = {'deb': <class 'avocado.plugins.distro.DistroPkgInfoLoader**
the type of distro that will determine what loader will be used

**class** avocado.plugins.distro.**Distro**
Bases: *avocado.core.plugin_interfaces.CLICmd*

Implements the avocado 'distro' subcommand

**configure** (*parser*)

**description = 'Shows detected Linux distribution'**

**get_output_file_name** (*args*)
Adapt the output file name based on given args

It's not uncommon for some distros to not have a release number, so adapt the output file name to that

**name = 'distro'**

**run** (*args*)

**class** avocado.plugins.distro.**DistroDef** (*name*, *version*, *release*, *arch*)
Bases: *avocado.utils.distro.LinuxDistro*

More complete information on a given Linux Distribution

Can and should include all the software packages that ship with the distro, so that an analysis can be made on whether a given package that may be responsible for a regression is part of the official set or an external package.

**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

**to_dict**()
Returns the representation as a dictionary

**to_json**()
Returns the representation of the distro as JSON

**class** avocado.plugins.distro.**DistroPkgInfoLoader**(*path*)
Bases: object

Loads information from the distro installation tree into a DistroDef

It will go through all package files and inspect them with specific package utilities, collecting the necessary information.

**get_package_info**(*path*)
Returns information about a given software package

Should be implemented by classes inheriting from DistroDefinitionLoader.

**Parameters** **path** (*str*) – path to the software package file

**Returns** tuple with name, version, release, checksum and arch

**Return type** tuple

**get_packages_info**()
This method will go through each file, checking if it's a valid software package file by calling *is_software_package()* and calling load_package_info() if it's so.

**is_software_package**(*path*)
Determines if the given file at *path* is a software package

This check will be used to determine if load_package_info() will be called for file at *path*. This method should be implemented by classes inheriting from *DistroPkgInfoLoader* and could be as simple as checking for a file suffix.

**Parameters** **path** (*str*) – path to the software package file

**Returns** either True if the file is a valid software package or False otherwise

**Return type** bool

**class** avocado.plugins.distro.**DistroPkgInfoLoaderDeb**(*path*)
Bases: *avocado.plugins.distro.DistroPkgInfoLoader*

Loads package information for DEB files

**get_package_info**(*path*)

**is_software_package**(*path*)

**class** avocado.plugins.distro.**DistroPkgInfoLoaderRpm**(*path*)
Bases: *avocado.plugins.distro.DistroPkgInfoLoader*

Loads package information for RPM files

**get_package_info**(*path*)

**is_software_package**(*path*)
Systems needs to be able to run the rpm binary in order to fetch information on package files. If the rpm binary is not available on this system, we simply ignore the rpm files found

**class** avocado.plugins.distro.**SoftwarePackage**(*name*, *version*, *release*, *checksum*, *arch*)
Bases: object

Definition of relevant information on a software package

**to_dict**()
Returns the representation as a dictionary

**to_json**()
Returns the representation of the distro as JSON

avocado.plugins.distro.**load_distro**(*path*)
Loads the distro from an external file

**Parameters** **path** (*str*) – the location for the input file

**Returns** a dict with the distro definition data

**Return type** dict

avocado.plugins.distro.**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 often 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

avocado.plugins.distro.**save_distro**(*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

## avocado.plugins.envkeep module

**class** avocado.plugins.envkeep.**EnvKeep**
Bases: *avocado.core.plugin_interfaces.CLI*

Keep environment variables on remote executions

**configure** (*parser*)

**description = 'Keep variables in remote environment'**

**name = 'envkeep'**

**run** (*args*)

## avocado.plugins.exec_path module

Libexec PATHs modifier

**class** avocado.plugins.exec_path.**ExecPath**
Bases: *avocado.core.plugin_interfaces.CLICmd*

Implements the avocado 'exec-path' subcommand

**description = 'Returns path to avocado bash libraries and exits.'**

**name = 'exec-path'**

**run** (*args*)
Print libexec path and finish

**Parameters** **args** – Command line args received from the run subparser.

## avocado.plugins.gdb module

Run tests with GDB goodies enabled.

**class** avocado.plugins.gdb.**GDB**
Bases: *avocado.core.plugin_interfaces.CLI*

Run tests with GDB goodies enabled

**configure** (*parser*)

**description = "GDB options for the 'run' subcommand"**

**name = 'gdb'**

**run** (*args*)

## avocado.plugins.human module

Human result UI

**class** avocado.plugins.human.**Human** (*args*)
Bases: *avocado.core.plugin_interfaces.ResultEvents*

Human result UI

**description = 'Human Interface UI'**

**end_test** (*result*, *state*)

**name = 'human'**

**output_mapping = {'SKIP': '', 'PASS': '', 'INTERRUPTED': '', 'WARN': '', 'CANCEL': '', 'ERROR': '', 'FAIL': ''}**

**post_tests** (*job*)

**pre_tests** (*job*)

**start_test** (*result*, *state*)

**test_progress** (*progress=False*)

**class** avocado.plugins.human.**HumanJob**
Bases: *avocado.core.plugin_interfaces.JobPre*, *avocado.core.plugin_interfaces.JobPost*

Human result UI

**description = 'Human Interface UI'**

**name = 'human'**

**post** (*job*)

**pre** (*job*)

## avocado.plugins.jobscripts module

**class** avocado.plugins.jobscripts.**JobScripts**
Bases: *avocado.core.plugin_interfaces.JobPre*, *avocado.core.plugin_interfaces.JobPost*

**description = 'Runs scripts before/after the job is run'**

**name = 'jobscripts'**

**post** (*job*)

**pre** (*job*)

## avocado.plugins.journal module

Journal Plugin

**class** avocado.plugins.journal.**Journal**
Bases: *avocado.core.plugin_interfaces.CLI*

Test journal

**configure** (*parser*)

**description = "Journal options for the 'run' subcommand"**

**name = 'journal'**

**run** (*args*)

**class** avocado.plugins.journal.**JournalResult** (*args*)
Bases: *avocado.core.plugin_interfaces.ResultEvents*

Test Result Journal class.

This class keeps a log of the test updates: started running, finished, etc. This information can be forwarded live to an avocado server and provide feedback to users from a central place.

Creates an instance of ResultJournal.

**Parameters** **job** – an instance of *avocado.core.job.Job*.

**description = 'Journal event based results implementation'**

**end_test** (*result*, *state*)

**lazy_init_journal** (*state*)

**name = 'journal'**

**post_tests** (*job*)

**pre_tests** (*job*)

**start_test** (*result*, *state*)

**test_progress** (*progress=False*)

## avocado.plugins.jsonresult module

JSON output module.

**class** avocado.plugins.jsonresult.**JSONCLI**
Bases: *avocado.core.plugin_interfaces.CLI*

JSON output

**configure** (*parser*)

**description = "JSON output options for 'run' command"**

**name = 'json'**

**run** (*args*)

**class** avocado.plugins.jsonresult.**JSONResult**
Bases: *avocado.core.plugin_interfaces.Result*

**description = 'JSON result support'**

**name = 'json'**

**render** (*result*, *job*)

## avocado.plugins.list module

**class** avocado.plugins.list.**List**
Bases: *avocado.core.plugin_interfaces.CLICmd*

Implements the avocado 'list' subcommand

**configure** (*parser*)
Add the subparser for the list action.

**Parameters** **parser** – Main test runner parser.

**description = 'List available tests'**

**name = 'list'**

**run** (*args*)

**class** avocado.plugins.list.**TestLister**(*args*)
Bases: object

Lists available test modules

**list**()

## avocado.plugins.multiplex module

**class** avocado.plugins.multiplex.**Multiplex**(**args*, ***kwargs*)
Bases: *avocado.plugins.variants.Variants*

DEPRECATED version of the "avocado multiplex" command which is replaced by "avocado variants" one.

**name = 'multiplex'**

**run**(*args*)

## avocado.plugins.plugins module

Plugins information plugin

**class** avocado.plugins.plugins.**Plugins**
Bases: *avocado.core.plugin_interfaces.CLICmd*

Plugins information

**configure**(*parser*)

**description = 'Displays plugin information'**

**name = 'plugins'**

**run**(*args*)

## avocado.plugins.replay module

**class** avocado.plugins.replay.**Replay**
Bases: *avocado.core.plugin_interfaces.CLI*

Replay a job

**configure**(*parser*)

**description = "Replay options for 'run' subcommand"**

**load_config**(*resultsdir*)

**name = 'replay'**

**run**(*args*)

## avocado.plugins.run module

Base Test Runner Plugins.

**class** avocado.plugins.run.**Run**
Bases: *avocado.core.plugin_interfaces.CLICmd*

Implements the avocado 'run' subcommand

**configure** (*parser*)
Add the subparser for the run action.

**Parameters** **parser** – Main test runner parser.

**description = 'Runs one or more tests (native test, test alias, binary or script)'**

**name = 'run'**

**run** (*args*)
Run test modules or simple tests.

**Parameters** **args** – Command line args received from the run subparser.

## avocado.plugins.sysinfo module

System information plugin

**class** avocado.plugins.sysinfo.**SysInfo**
Bases: *avocado.core.plugin_interfaces.CLICmd*

Collect system information

**configure** (*parser*)
Add the subparser for the run action.

**Parameters** **parser** – Main test runner parser.

**description = 'Collect system information'**

**name = 'sysinfo'**

**run** (*args*)

## avocado.plugins.tap module

TAP output module.

**class** avocado.plugins.tap.**TAP**
Bases: *avocado.core.plugin_interfaces.CLI*

TAP Test Anything Protocol output avocado plugin

**configure** (*parser*)

**description = 'TAP - Test Anything Protocol results'**

**name = 'TAP'**

**run** (*args*)

**class** avocado.plugins.tap.**TAPResult** (*args*)
Bases: *avocado.core.plugin_interfaces.ResultEvents*

TAP output class

**description = 'TAP - Test Anything Protocol results'**

**end_test** (*result*, *state*)
Log the test status and details

**name = 'tap'**

**post_tests** (*job*)

**pre_tests** (*job*)
Log the test plan

**start_test** (*result*, *state*)

**test_progress** (*progress=False*)

avocado.plugins.tap.**file_log_factory** (*log_file*)
Generates a function which simulates writes to logger and outputs to file

**Parameters** **log_file** – The output file

## avocado.plugins.teststmpdir module

Tests temporary directory plugin

**class** avocado.plugins.teststmpdir.**TestsTmpDir**
Bases: *avocado.core.plugin_interfaces.JobPre*, *avocado.core.plugin_interfaces.JobPost*

**description = 'Creates a temporary directory for tests consumption'**

**name = 'teststmpdir'**

**post** (*job*)

**pre** (*job*)

## avocado.plugins.variants module

**class** avocado.plugins.variants.**Variants** (**args*, ***kwargs*)
Bases: *avocado.core.plugin_interfaces.CLICmd*

Implements "variants" command to visualize/debug test variants and params

**configure** (*parser*)

**description = 'Tool to analyze and visualize test variants and params'**

**name = 'variants'**

**run** (*args*)

avocado.plugins.variants.**map_verbosity_level** (*level*)

## avocado.plugins.wrapper module

**class** avocado.plugins.wrapper.**Wrapper**
Bases: *avocado.core.plugin_interfaces.CLI*

Implements the '–wrapper' flag for the 'run' subcommand

**configure** (*parser*)

**description = "Implements the '–wrapper' flag for the 'run' subcommand"**

**name = 'wrapper'**

**run** (*args*)

## avocado.plugins.xunit module

xUnit module.

**class** avocado.plugins.xunit.**XUnitCLI**
Bases: *avocado.core.plugin_interfaces.CLI*

xUnit output

**configure**(*parser*)

**description = 'xUnit output options'**

**name = 'xunit'**

**run**(*args*)

**class** avocado.plugins.xunit.**XUnitResult**
Bases: *avocado.core.plugin_interfaces.Result*

**PRINTABLE = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789!"#$%&\'()*+,-./:;<=**

**UNKNOWN = '<unknown>'**

**description = 'XUnit result support'**

**name = 'xunit'**

**render**(*result*, *job*)

## avocado.plugins.yaml_to_mux module

Multiplexer plugin to parse yaml files to params

**class** avocado.plugins.yaml_to_mux.**ListOfNodeObjects**
Bases: list

Used to mark list as list of objects from whose node is going to be created

**class** avocado.plugins.yaml_to_mux.**Value**
Bases: tuple

Used to mark values to simplify checking for node vs. value

**class** avocado.plugins.yaml_to_mux.**YamlToMux**
Bases: *avocado.core.mux.MuxPlugin*, *avocado.core.plugin_interfaces.Varianter*

Processes the mux options into varianter plugin

**description = 'Multiplexer plugin to parse yaml files to params'**

**initialize**(*args*)

**name = 'yaml_to_mux'**

**class** avocado.plugins.yaml_to_mux.**YamlToMuxCLI**
Bases: *avocado.core.plugin_interfaces.CLI*

Defines arguments for YamlToMux plugin

**configure**(*parser*)
Configures "run" and "multiplex" subparsers

**description = "YamlToMux options for the 'run' subcommand"**

**name = 'yaml_to_mux'**

**run**(*args*)
The YamlToMux varianter plugin handles these

avocado.plugins.yaml_to_mux.**create_from_yaml**(*paths*, *debug=False*)
Create tree structure from yaml-like file :param fileobj: File object to be processed :raise SyntaxError: When yaml-file is corrupted :return: Root of the created tree structure

## Module contents

CHAPTER 19

# Avocado Release Notes

## Release Notes

The following pages summarize what is new in Avocado:

### 50.0 A Dog's Will

The Avocado team is proud to present another release: Avocado version 50.0, aka, "A Dog's Will" now available!

Release documentation: Avocado 50.0

The major changes introduced on this version are listed below, roughly categorized into major topics and intended audience:

#### Users/Test Writers

- Avocado now supports resuming jobs that were interrupted. This means that a system crash, or even an intentional interruption, won't prevent you from continuing the execution of a job. To use this feature, provide `--replay-resume` on the Avocado execution that proceeds the crash or interruption.
- The docstring directives that Avocado uses to allow for *test categorization* was previously limited to a class docstring. Now, individual test methods can also have their own tags, while also respecting the ones at the class level. The documentation has been updated with an *example*.
- The HTML report now presents the test ID and variant ID in separate columns, allowing users to also sort and filter results based on those specific fields.
- The HTML report will now show the test parameters used in a test when the user hovers the cursor over the test name.
- Avocado now reports the total job execution time on the UI, instead of just the tests execution time. This may affect users that are looking for the `TESTS TIME:` line, and reinforce that machine readable formats such as JSON and XUnit are more dependable than the UI intended for humans.

- The *avocado.core.plugin_interfaces.JobPre* is now properly called *before* *avocado.core.job.Job.run()*, and accordingly *avocado.core.plugin_interfaces.JobPost* is called *after* it. Some plugins which depended on the previous behavior can use the *avocado.core.plugin_interfaces.JobPreTests* and *avocado.core.plugin_interfaces.JobPostTests* for a similar behavior. As a example on how to write plugin code that works properly this Avocado version, as well as on previous versions, take a look at this accompanying Avocado-VT plugin commit.
- The Avocado `multiplex` command has been renamed to `variants`. Users of `avocado multiplex` will notice a deprecation message, and are urged to switch to the new command. The command line options and behavior of the `variants` command is identical to the `multiplex` one.
- The number of variants produced with the `multiplex` command (now `variants`) was missing in the previous version. It's now been restored.

### Internal Changes

- Avocado's own internal tests now can be given different level marks, and will run a different level on different environments. The idea is to increase coverage without having false positives on more restricted environments.
- The `test_tests_tmp_dir` selftests that was previously disable due to failure on our CI environment was put back to be executed.
- The amount of the test runner will wait for the test process exit status has received tweaks and is now better documented (see *avocado.core.runner.TIMEOUT_TEST_INTERRUPTED*, *avocado.core.runner.TIMEOUT_PROCESS_DIED* and *avocado.core.runner.TIMEOUT_PROCESS_ALIVE*).
- Some cleanups and refactors were made to how the `SKIP` and `CANCEL` test statuses are implemented.

For more information, please check out the complete Avocado changelog.

### Release Meeting

The Avocado release meetings are now open to the community via Hangouts on Air. The meetings are recorded and made available on the Avocado Test Framework YouTube channel.

For this release, you can watch the meeting on this link.

---

Sprint theme: https://trello.com/c/FleklxHi/1016-sprint-theme-a-dog-s-will-2000

## 49.0 The Physician

The Avocado team is proud to present another release: Avocado version 49.0, aka, "The Physician" now available!

Release documentation: Avocado 49.0

The major changes introduced on this version are listed below, roughly categorized into major topics and intended audience:

### Users/Test Writers

- A brand new ResultsDB plugin. This allows Avocado jobs to send results directly to any ResultsDB server.
- Avocado's *data_dir* is now set by default to `/var/lib/avocado/data` instead of `/usr/share/avocado/data`. This was a problem because `/usr` must support read only mounts, and is not intended for that purpose at all.
- When users run `avocado list --loaders ?` they used to receive a single list containing loader plugins **and** test types, all mixed together. Now users will get one loader listed per line, along with the test types that each loader supports.
- Variant-IDs created by the multiplexer are now much more meaningful. Previously, the Variant-ID would be a simple sequential integer, it now combines information about the leaf names in the multiplexer tree and a 4 digit fingerprint. As a quick example, users will now get `sleeptest.py:SleepTest.test;short-beaf` instead of `sleeptest.py:SleepTest.test;1` as test IDs when using the multiplexer.
- The multiplexer now supports the use filters defined inside the YAML files, and greatly expand its filtering capabilities.
- [BUGFIX] Instrumented tests support docstring directives, but only one of the supported directives (either enable/disable or tags) at once. It's now possible to use both in a single docstring.
- [BUGFIX] Some result plugins would generate some output even when the job did not contain a valid test suite.
- [BUGFIX] Avocado would crash when listing tests with the `file` loader disabled. `MissingTests` used to be initialized by the file loader, but are now registered as a part of the loader proxy (similar to a plugin manager) so this is not an issue anymore.

### Distribution

- The packages on Avocado's own RPM repository are now a lot more similar to the ones in the Fedora and EPEL repositories. This will make future maintenance easier, and also allows users to switch between versions with greater ease.

For more information, please check out the complete Avocado changelog.

### Release Meeting

The Avocado release meetings are now open to the community via Hangouts on Air. The meetings are recorded and made available on the Avocado Test Framework YouTube channel.

For this release, you can watch the meeting on this link.

---

Sprint theme: https://trello.com/c/CuQX9Mew/991-sprint-theme-the-physician-2013

## 48.0 Lost Boundaries

The Avocado team is proud to present another release: Avocado version 48.0, aka, "Lost Boundaries" now available!

Release documentation: Avocado 48.0

The major changes introduced on this version are listed below, roughly categorized into major topics and intended audience:

### Users/Test Writers

- Users of *avocado.utils.linux_modules* functions will find that a richer set of information is provided in their return values. It now includes module name, size, submodules if present, filename, version, number of modules using it, list of modules it is dependent on and finally a list of params.
- *avocado.TestFail*, *avocado.TestError* and *avocado.TestCancel* are now public Avocado Test APIs, available from the main `avocado` namespace. The reason is that test suites may want to define their own exceptions that, while have some custom meaning, also act as a way to fail (or error or cancel) a test.
- Support for new type of test status, CANCEL, and of course the mechanisms to set a test with this status. CANCEL is a lot like what many people think of SKIP, but, to keep solid definitions and predictable behavior, a SKIP(ped) test is one that was **never** executed, and a CANCEL(ed) test is one that was partially executed, and then canceled. Calling `self.skip()` from within a test is now deprecated to adhere even closer to these definitions. Using the `skip*` decorators (which are outside of the test execution) is still permitted and won't be deprecated.
- Introduction of the `robot` plugin, which allows Robot Framework tests to be listed and executed natively within Avocado. Just think of a super complete Avocado job that runs build tests, unit tests, functional and integration tests... and, on top of it, interactive UI tests for your application!
- Adjustments to the use of `AVOCADO_JOB_FAIL` and `AVOCADO_FAIL` exit status code by Avocado. This matters if you're checking the exact exit status code that Avocado may return on error condtitions.

### Documentation / Contrib

- Updates to the `README` and Getting Started documentation section, which now mention the updated package names and are pretty much aligned with each other.

### Distribution

- Avocado optional plugins are now also available on PyPI, that is, can be installed via `pip`. Here's a list of the current package pages:
- https://pypi.python.org/pypi/avocado-framework-plugin-result-html
- https://pypi.python.org/pypi/avocado-framework-plugin-runner-remote
- https://pypi.python.org/pypi/avocado-framework-plugin-runner-vm
- https://pypi.python.org/pypi/avocado-framework-plugin-runner-docker
- https://pypi.python.org/pypi/avocado-framework-plugin-robot

For more information, please check out the complete Avocado changelog.

### Release Meeting

The Avocado release meetings are now open to the community via Hangouts on Air. The meetings are recorded and made available on the Avocado Test Framework YouTube channel.

For this release, you can watch the meeting on this link.

---

Sprint theme: https://trello.com/c/Y02Koizf/952-sprint-theme-lost-boundaries

## 47.0 The Lost Wife

The Avocado team is proud to present another release: Avocado version 47.0, aka, "The Lost Wife" now available!

Release documentation: Avocado 47.0

The major changes introduced on this version are listed below, roughly categorized into major topics and intended audience:

### Users/Test Writers

- The *avocado.Test* class now better exports (and protects) the core class attributes members (such as `params` and `runner_queue`). These were turned into properties so that they're better highlighted in the docs and somehow protected when users would try to replace them.
- Users sending `SIGTERM` to Avocado can now expect it to be properly handled. The handling done by Avocado includes sending the same `SIGTERM` to all children processes.

### Internal improvements

- The multiplexer has just become a proper plugin, implementing the also new *avocado.core.plugin_interfaces.Varianter* interface.
- The selftests wouldn't check for the proper location of the avocado job results directory, and always assumed that `~/avocado/job-results` exists. This is now properly verified and fixed.

### Bug fixes

- The UI used to show the number of tests in a `TESTS: <no_of_tests>` line, but that would not take into account the number of variants. Since the following line also shows the current test and the total number of tests (including the variants) the `TESTS: <no_of_tests>` was removed.
- The Journal plugin would crash when used with the remote (and derivative) runners.
- The whiteboard would not be created when the current working directory would change inside the test. This was related to the `datadir` not being returned as an absolute path.

### Documentation / Contrib

- The `avocado` man page (`man 1 avocado`) is now update and lists all currently available commands and options. Since some command and options depend on installed plugins, the man page includes all "optional" plugins (remote runner, vm runner, docker runner and html).

For more information, please check out the complete Avocado changelog.

### Release Meeting

The Avocado release meetings are now open to the community via Hangouts on Air. The meetings are recorded and made available on the Avocado Test Framework YouTube channel.

For this release, you can watch the meeting on this link.

---

Sprint theme: https://trello.com/c/HaFLiXyD/928-sprint-theme-the-lost-wife

## 46.0 Burning Bush

The Avocado team is proud to present another release: Avocado version 46.0, aka, "Burning Bush" now available!

Release documentation: Avocado 46.0

The major changes introduced on this version are listed below, roughly categorized into major topics and intended audience:

### Users/Test Writers

- Avocado test writers can now use a family of decorators, namely *avocado.skip()*, *avocado.skipIf()* and *avocado.skipUnless()* to skip the execution of tests. These are similar to the well known `unittest` decorators.
- Sysinfo collection based on command execution now allows a timeout to be set. This makes test job executions with sysinfo enabled more reliable, because the job won't hang until it reaches the job timeout.
- Users will receive better error messages from the multiplexer (variant subsystem) when the given YAML files do not exist.
- Users of the *avocado.utils.process.system_output()* will now get the command output with the trailing newline stripped by default. If needed, a parameter can be used to preserve the newline. This is now consistent with most Python process execution utility APIs.

### Distribution

- The non-local runner plugins are now distributed in separate RPM packages. Users installing from RPM packages should also install packages such as `avocado-plugins-runner-remote`, `avocado-plugins-runner-vm` and `avocado-plugins-runner-docker`. Users upgrading from previous Avocado versions should also install these packages manually or they will lose the corresponding functionality.

### Internal improvements

- Python 2.6 support has been dropped. This now paves the way for our energy to be better spent on developing new features and also bring proper support for Python 3.x.

### Bug fixes

- The TAP result plugin was printing an incorrect test plan when using the multiplexer (variants) mechanism. The total number of tests to be executed (the first line in TAP output) did not account for the number of variants.
- The remote, vm and docker runners would print some UI related messages even when other types of result (such as TAP, json, etc) would be set to output to STDOUT.
- Under some scenarios, an Avocado test would create an undesirable and incomplete job result directory on demand.

### Documentation / Contrib

- The Avocado page on PythonHosted.org now redirects to our official documentation page.
- We now document how to pause and unpause tests.

- A script to simplify bisecting with Avocado has been added to the `contrib` directory.

For more information, please check out the complete Avocado changelog.

### Release Meeting

The Avocado release meetings are now open to the community via Hangouts on Air. The meetings are recorded and made available on the Avocado Test Framework YouTube channel.

For this release, you can watch the meeting on this link.

---

Sprint theme: https://trello.com/c/I6KG9bpq/893-sprint-theme-burning-bush

## 45.0 Anthropoid

The Avocado team is proud to present another release: Avocado version 45.0, aka, "Anthropoid", is now available!

Release documentation: Avocado 45.0

The major changes introduced on this version are listed below, roughly categorized into major topics and intended audience:

### Users/Test Writers

- Tests running with the external runner (`--external-runner`) feature will now have access to the extended behavior for SIMPLE tests, such as being able to exit a test with the WARNING status.
- Users will now be able to properly run tests based on any Unicode string (as a test reference). To achieve that, the support for arguments to SIMPLE tests was dropped, as it was impossible to have a consistent way to determine if special characters were word separators, arguments or part of the main test name. To overcome the removal of support for arguments on SIMPLE tests, one can use custom loader configurations and the external runner.
- Test writers now have access to a test temporary directory that will last not only for the duration of the test, but for the duration of the whole job execution. This is a feature that has been requested by many users and one practical example is a test reusing binaries built on by a previous test on the same job. Please note that Avocado still provides as much test isolation and independence as before, but now allows tests to share this one directory.
- When running jobs with the TAP plugin enabled (the default), users will now also get a `results.tap` file created by default in their job results directory. This is similar to how JSON, XUNIT and other supported result formats already operate. To disable the TAP creation, either disable the plugin or use `--tap-job-result=off`.

### Distribution

- Avocado is now available on Fedora. That's great news for test writers and test runners, who will now be able to rely on Avocado installed on test systems much more easily. Because of Fedora's rules that favor the stability of packages during a given release, users will find older Avocado versions (currently 43.0) on already released Fedora versions. For users interested in packages for the latest Avocado releases, we'll continue to provide updated packages on our own repo.
- After some interruption, we've addressed issues that were preventing the update of Avocado packages on PyPI, and thus, preventing users from getting the latest Avocado versions when running `$ pip install avocado-framework`.

### Internal improvements

- The HTML report plugin contained a font, included by the default bootstrap framework data files, that was not really used. It has now been removed.
- The selfcheck will now require commits to have a `Signed-off-by` line, in order to make sure contributors are aware of the terms of their contributions.

For more information, please check out the complete Avocado changelog.

### Release Meeting

The Avocado release meetings are now open to the community via Hangouts on Air. The meetings are recorded and made available on the Avocado Test Framework YouTube channel.

For this release, you can watch the meeting on this link.

---

Sprint theme: https://trello.com/c/fwEUquwd/881-sprint-theme-anthropoid

## 44.0 The Shadow Self

The Avocado team is proud to present another release: Avocado version 44.0, aka, "The Shadow Self", is now available!

Release documentation: Avocado 44.0

The major changes introduced on this version are listed below, roughly categorized into major topics and intended audience:

### Users/Test Writers

- Avocado now supports filtering tests by user supplied "tags". These tags are given in docstrings, similar to the already existing docstring directives that force Avocado to either enable or disable the detection of a class as an Avocado INSTRUMENTED test. With this feature, you can now write your tests more freely accross Python files and choose to run only a subset of them, based on the their tag values. For more information, please take a look at *Categorizing tests*.
- Users can now choose to keep the complete set of files, including temporary ones, created during an Avocado job run by using the `--keep-tmp` option.
- The `--job-results-dir` option was previously used to point to where the job results should be saved. Some features, such as job replay, also look for content (`jobdata`) into the job results dir, and it now respects the value given in `--job-results-dir`.

### Documentation

- A warning is now present to help avocado users on some architectures and older PyYAML versions to work around failures in the Multiplexer.

### Bugfixes

- A quite nasty, logging related, `RuntimeError` would happen every now and then. While it was quite hard to come up with a reproducer (and thus a precise fix), this should be now a thing of the past.
- The Journal plugin could not handle Unicode input, such as in test names.

### Internal improvements

- Selftests are now also executed under EL7. This means that Avocado on EL7, and EL7 packages, have an additional level of quality assurance.
- The old `check-long` Makefile target is now named `check-full` and includes both tests that take a long time to run, but also tests that are time sensitive, and that usually fail when not enough computing resources are present.

For more information, please check out the complete Avocado changelog.

### Release Meeting

The Avocado release meetings are now open to the community via Hangouts on Air. The meetings are recorded and made available on the Avocado Test Framework YouTube channel.

For this release, you can watch the meeting on this link.

---

Sprint theme: https://trello.com/c/CLTdFYLW/869-sprint-theme-the-shadow-self

## 43.0 The Emperor and the Golem

The Avocado team is proud to present another release: Avocado version 43.0, aka, "The Emperor and the Golem", is now available!

Release documentation: Avocado 43.0

The major changes introduced on this version are listed below, roughly categorized into major topics and intended audience:

### Users/Test Writers

- The `--remote-no-copy` option has been removed. The reason is that the copying of tests to the remote hosts (as set with `--remote-hostname`) was also removed. That feature, while useful to some, had a lot of corner cases. Instead of keeping a feature with a lot of known caveats, it was decided that users should setup the remote machines so that tests are available before Avocado attempts to run them.
- The *`avocado.utils.process`* library, one of the most complex pieces of utility code that Avocado ships, now makes it possible to ignore background processes that never finish (while Avocado is reading from their file descriptors to properly return their output to the caller). The reason for such a feature is that if a command spawn many processes, specially daemon-like ones that never finish, the *`avocado.utils.process.run()`* function would hang indefinitely. Since waiting for all the children processes to finish is the right thing to do, users need to set the `ignore_bg_processes` parameter to `True` to request this newly added behavior.

- When discovering tests on a directory, that is, when running `avocado list /path/to/tests/directory` or `avocado run /path/to/tests/directory`, Avocado would return tests in a non predictable way, based on `os.walk()`. Now, the result is a properly alphabetically ordered list of tests.
- The ZIP Archive feature (AKA as `--archive` or `-z`) feature, which allows to archive job results is now a proper plugin.
- Plugins can now be setup to run at a specific order. This is a response to a user issue/request, where the `--archive` feature would run before some other results would be generated. This feature is not limited to plugins of type *result*. It allows any ordering on the enabled set of plugins of a given plugin type.
- A contrib script that looks for a job result directory based on a partial (or complete) job ID is now available at `contrib/scripts/avocado-get-job-results-dir.py`. This should be useful inside automation scripts or even for interactive users.

### Documentation

- Users landing on http://avocado-framework.readthedocs.io would previously be redirect to the "latest" documentation, which tracks the development master branch. This could be confusing since the page titles would contain a version notice with the latest *released* version. Users will now be redirected by default to the latest *released* version, matching the page title, although the version tracking the master branch will still be available at the http://avocado-framework.readthedocs.io/en/latest URL.

### Bugfixes

- During the previous development cycle, a bug where `journalctl` would receive *KeyboardInterrupt* received an workaround by using the `subprocess` library instead of Avocado's own *avocado.utils.process*, which was missing a default handler for *SIGINT*. With the misbehavior of Avocado's library now properly addressed, and consequently, we've reverted the workaround applied previously.
- The TAP plugin would fail at the *end_test* event with certain inputs. This has now been fixed, and in the event of errors, a better error message will be presented.

### Internal improvements

- The `test_utils_partition.py` selftest module now makes use of the `avocado.core.utils.process.can_sudo()` function, and will only be run when the user is either running as root or has sudo correctly configured.
- Avocado itself preaches that tests should not attempt to skip themselves during their own execution. The idea is that, once a test started executing, you can't say it wasn't executed (skipped). This is actually enforced in *avocado.Test* based tests. But since Avocado's own selftests are based on `unittest.TestCase`, some of them were using skip at the "wrong" place. This is now fixed.
- The *avocado.core.job.Job* class received changes that make it more closer to be usable as a formally announced and supported API. This is another set of changes towards the so-called "Job API" support.
- There is now a new plugin type, named *result_events*. This replaces the previous implementation that used *avocado.core.result.Result* as a base class. There's now a single *avocado.core.result.Result* instance in a given job, which tracks the results, while the plugins that act on result events (such as test has started, test has finished, etc) are based on the `avocado.core.plugins_interfaces.ResultEvents`.
- A new `result_events` plugin called `human` now replaces the old *HumanResult* implementation.
- Ported versions of the TAP and journal plugins to the new `result_events` plugin type.

For more information, please check out the complete Avocado changelog.

### Release Meeting

The Avocado release meetings are now open to the community via Hangouts on Air. The meetings are recorded and made available on the Avocado Test Framework YouTube channel.

For this release, you can watch the meeting on this link.

---

Sprint theme: https://trello.com/c/r2fwf66S/853-sprint-theme-the-emperor-and-the-golem-1952

## 42.0 Stranger Things

The Avocado team is proud to present another release: Avocado version 42.0, aka, "Stranger Things", is now available!

Release documentation: Avocado 42.0

The major changes introduced on this version are listed below, roughly categorized into major topics and intended audience:

### Users/Test Writers

- Multiplexer: it now defines an API to inject and merge data into the multiplexer tree. With that, it's now possible to come up with various mechanisms to feed data into the Multiplexer. The standard way to do so continues to be by YAML files, which is now implemented in the `avocado.plugins.yaml_to_mux` plugin module. The *–multiplex* option, which used to load YAML files into the multiplexer is now deprecated in favor of *–mux-yaml*.
- Docker improvements: Avocado will now name the container accordingly to the job it's running. Also, it not allows generic Docker options to be passed by using *–docker-options* on the Avocado command line.
- It's now possible to disable plugins by using the configuration file. This is documented at *Disabling a plugin*.
- `avocado.utils.iso9660`: this utils module received a lot of TLC and it now provides a more complete standard API across all backend implementations. Previously, only the mount based backend implementation would support the *mnt_dir* API, which would point to a filesystem location where the contents of the ISO would be available. Now all other backends can support that API, given that requirements (such as having the right privileges) are met.
- Users of the `avocado.utils.process` module will now be able to access the process ID in the `avocado.utils.process.CmdResult`
- Users of the `avocado.utils.build` module will find an improved version of `avocado.utils.build.make()` which will now return the *make* process exit status code.
- Users of the virtual machine plugin (`--vm-domain` and related options) will now receive better messages when errors occur.

### Documentation

- Added section on how to use custom Docker images with user's own version of Avocado (or anything else for that matter).
- Added section on how to install Avocado using standard OpenSUSE packages.

- Added section on `unittest` compatibility limitations and caveats.
- A link to Scylla Clusters tests has been added to the list of Avocado test repos.
- Added section on how to install Avocado by using standard Python packages.

### Developers

- The *make develop* target will now activate in-tree optional plugins, such as the HTML report plugin.
- The *selftests/run* script, usually called as part of *make check*, will now fail at the first failure (by default). This is controlled by the *SELF_CHECK_CONTINUOUS* environment variable.
- The *make check* target can also run tests in parallel, which can be enabled by setting the environment variable *AVOCADO_PARALLEL_CHECK*.

### Bugfixes

- An issue where *KeyboardInterrupts* would be caught by the *journalctl* run as part of sysinfo was fixed with a workaround. The root cause appears to be located in the `avocado.utils.process` library, and a task is already on track to verify that possible bug.
- `avocado.util.git` module had an issue where git executions would generate content that would erroneously be considered as part of the output check mechanism.

### Internal improvements

- Selftests are now run while building Enterprise Linux 6 packages. Since most Avocado developers use newer platforms for development, this should make Avocado more reliable for users of those older platforms.

For more information, please check out the complete Avocado changelog.

### Release Meeting

The Avocado release meetings are now open to the community via Hangouts on Air. The meetings are recorded and made available on the Avocado Test Framework YouTube channel.

For this release, you can watch the meeting on this link.

---

Sprint theme: https://trello.com/c/icVc5Szx/851-sprint-theme-stranger-things

## 41.0 Outlander

The Avocado team is proud to present another release: Avocado version 41.0, aka, “Outlander”, is now available!

Release documentation: Avocado 41.0

The major changes introduced on this version are listed below, roughly categorized into major topics and intended audience:

### Users/Test Writers

- Multiplex: remove the *-s* (system-wide) shortcut to avoid confusion with *silent* from main apps.
- New `avocado.utils.linux_modules.check_kernel_config()` method, with which users can check if a kernel configuration is not set, a module or built-in.
- Show link to file which failed to be processed by sysinfo.
- New *path* key type for settings that auto-expand tilde notation, that is, when using `avocado.core.settings.Settings.get_value()` you can get this special value treatment.
- The automatic VM IP detection that kicks in when one uses *–vm-domain* without a matching *–vm-hostname*, now uses a more reliable method (libvirt/qemu-gust-agent query). On the other hand, the QEMU guest agent is now required if you intend to omit the VM IP/hostname.
- Warn users when sysinfo configuration files are not present, and consequently no sysinfo is going to be collected.
- Set *LC_ALL=C* by default on sysinfo collection to simplify *avocado diff* comparison between different machines. It can be tweaked in the config file (*locale* option under *sysinfo.collect*).
- Remove deprecated option *–multiplex-files*.
- List result plugins (JSON, XUnit, HTML) in *avocado plugins* command output.

### Documentation

- Mention to the community maintained repositories.
- Add GIT workflow to the contribution guide.

### Developers

- New *make check-long* target to run long tests. For example, the new *FileLockTest*.
- New *make variables* target to display Makefile variables.
- Plugins: add optional plugins directory *optional_plugins*. This also adds all directories to be found under *optional_plugins* to the list of candidate plugins when running *make clean* or *make link*.

### Bugfixes

- Fix *undefined name* error `avocado.core.remote.runner`.
- Ignore *r* when checking for avocado in remote executions.
- Skip file if *UnicodeDecodeError* is raised when collecting sysinfo.
- Sysinfo: respect package collection on/off configuration.
- Use *-y* in *lvcreate* to ignore warnings `avocado.utils.lv_utils`.
- Fix crash in `avocado.core.tree` when printing non-string values.
- *setup.py*: fix the virtualenv detection so readthedocs.org can properly probe Avocado's version.

### Internal improvements

- Cleanup runner->multiplexer API
- Replay re-factoring, renamed *avocado.core.replay* to `avocado.core.jobdata`.
- Partition utility class defaults to ext2. We documented that and reinforced in the accompanying unittests.
- Unittests for `avocado.utils.partition` has now more specific checks for the conditions necessary to run the Partition tests (sudo, mkfs.ext2 binary).
- Several Makefile improvements.

For more information, please check out the complete Avocado changelog.

### Release Meeting

The Avocado release meetings are now open to the community via Hangouts on Air. The meetings are recorded and made available on the Avocado Test Framework YouTube channel.

For this release, you can watch the meeting on this link.

---

Sprint theme: https://trello.com/c/5oShOR1D/812-sprint-theme-outlander

## 40.0 Dr Who

The Avocado team is proud to present another release: Avocado version 40.0, aka, "Dr Who", is now available!

Release documentation: Avocado 40.0

The major changes introduced on this version are listed below.

- The introduction of a tool that generated a diff-like report of two jobs. For more information on this feature, please check out its own *documentation*.
- The `avocado.utils.process` library has been enhanced by adding the `avocado.utils.process.SubProcess.get_pid()` method, and also by logging the command name, status and execution time when verbose mode is set.
- The introduction of a rr based wrapper. With such a wrapper, it's possible to transparently record the process state (when executed via the `avocado.utils.process` APIs), and deterministically replay them later.
- The coredump generation contrib scripts will check if the user running Avocado is privileged to actually generate those dumps. This means that it won't give errors in the UI about failures on pre/post scripts, but will record that in the appropriate job log.
- BUGFIX: The `--remote-no-copy` command line option, when added to the `--remote-*` options that actually trigger the remote execution of tests, will now skip the local test discovery altogether.
- BUGFIX: The use of the asset fetcher by multiple avocado executions could result in a race condition. This is now fixed, backed by a file based utility lock library: `avocado.utils.filelock`.
- BUGFIX: The asset fetcher will now properly check the hash on `file:` based URLs.
- BUGFIX: A busy loop in the `avocado.utils.process` library that was reported by our users was promptly fixed.

- BUGFIX: Attempts to install Avocado on bare bones environments, such as virtualenvs, won't fail anymore due to dependencies required at `setup.py` execution time. Of course Avocado still requires some external Python libraries, but these will only be required after installation. This should let users to `pip install avocado-framework` successfully.

For more information, please check out the complete Avocado changelog.

### Release Meeting

The Avocado release meetings are now open to the community via Hangouts on Air. The meetings are recorded and made available on the Avocado Test Framework YouTube channel.

For this release, you can watch the meeting on this link.

---

Sprint theme: https://trello.com/c/P1Ps7T0F/782-sprint-theme-dr-who

## 39.0 The Hateful Eight

The Avocado team is proud to present another incremental release: version 39.0, aka, "The Hateful Eight", is now available!

Release documentation: Avocado 39.0

The major changes introduced on this version are listed below.

- Support for running tests in Docker container. Now, in addition to running tests on a (libvirt based) Virtual Machine or on a remote host, you can now run tests in transient Docker containers. The usage is as simple as:

```
$ avocado run mytests.py --docker ldoktor/fedora-avocado
```

  The container will be started, using `ldoktor/fedora-avocado` as the image. This image contains a Fedora based system with Avocado already installed, and it's provided at the official Docker hub.

- Introduction of the "Fail Fast" feature.

  By running a job with the `--failfast` flag, the job will be interrupted after the very first test failure. If your job only makes sense if it's a complete PASS, this feature can save you a lot of time.

- Avocado supports replaying previous jobs, selected by using their Job IDs. Now, it's also possible to use the special keyword `latest`, which will cause Avocado to rerun the very last job.

- Python's standard signal handling is restored for SIGPIPE, and thus for all tests running on Avocado.

  In previous releases, Avocado introduced a change that set the default handler to SIGPIPE, which caused the application to be terminated. This seemed to be the right approach when testing how the Avocado app would behave on broken pipes on the command line, but it introduced side effects to a lot of Python code. Instead of exceptions, the affected Python code would receive the signal themselves.

  This is now reverted to the Python standard, and the signal behavior of Python based tests running on Avocado should not surprise anyone.

- The project release notes are now part of the official documentation. That means that users can quickly find when a given change was introduced.

Together with those changes listed, a total of 38 changes made into this release. For more information, please check out the complete Avocado changelog.

### Release Meeting

The Avocado release meetings are now open to the community via Hangouts on Air. The meetings are recorded and made available on the Avocado Test Framework YouTube channel.

For this release, you can watch the meeting on this link.

---

Sprint theme: https://trello.com/c/nEiT7IjJ/755-sprint-theme-the-hateful-eight

## 38.0 Love, Ken

You guessed it right: this is another Avocado release announcement: release 38.0, aka "Love, Ken", is now out!

Release documentation: Avocado 38.0

Another development cycle has just finished, and our community will receive this new release containing a nice assortment of bug fixes and new features.

- The download of assets in tests now allow for an expiration time. This means that tests that need to download any kind of external asset, say a tarball, can now automatically benefit from the download cache, but can also keep receiving new versions automatically.

  Suppose your asset uses an asset named *myproject-daily.tar.bz2*, and that your test runs 50 times a day. By setting the expire time to *1d* (1 day), your test will benefit from cache on most runs, but will still fetch the new version when the 24 hours from the first download have passed.

  For more information, please check out the documentation on the *expire* parameter to the *fetch_asset()* method.

- Environment variables can be propagated into tests running on remote systems. It's a known fact that one way to influence application behavior, including test, is to set environment variables. A command line such as:

```
$ MYAPP_DEBUG=1 avocado run myapp_test.py
```

  Will work as expected on a local system. But Avocado also allows running tests on remote machines, and up until now, it has been lacking a way to propagate environment variables to the remote system.

  Now, you can use:

```
$ MYAPP_DEBUG=1 avocado run --env-keep MYAPP_DEBUG \
  --remote-host test-machine myapp_test.py
```

- The plugin interfaces have been moved into the *avocado.core.plugin_interfaces* module. This means that plugin writers now have to import the interface definitions this namespace, example:

```
...
from avocado.core.plugin_interfaces import CLICmd

class MyCommand(CLICmd):
...
```

  This is a way to keep ourselves honest, and say that there's no difference from plugin interfaces to Avocado's core implementation, that is, they may change at will. For greater stability, one should be tracking the LTS releases.

  Also, it effectively makes all plugins the same, whether they're implemented and shipped as part of Avocado, or as part of external projects.

- A contrib script for running kvm-unit-tests. As some people are aware, Avocado has indeed a close relation to virtualization testing. Avocado-VT is one obvious example, but there are other virtualization related test suites can Avocado can run.

  This release adds a contrib script that will fetch, download, compile and run kvm-unit-tests using Avocado's external runner feature. This gives results in a better granularity than the support that exists in Avocado-VT, which gives only a single PASS/FAIL for the entire test suite execution.

For more information, please check out the Avocado changelog.

#### Avocado-VT

Also, while we focused on Avocado, let's also not forget that Avocado-VT maintains it's own fast pace of incoming niceties.

- s390 support: Avocado-VT is breaking into new grounds, and now has support for the s390 architecture. Fedora 23 for s390 has been added as a valid guest OS, and s390-virtio has been added as a new machine type.
- Avocado-VT is now more resilient against failures to persist its environment file, and will only give warnings instead of errors when it fails to save it.
- An improved implementation of the "job lock" plugin, which prevents multiple Avocado jobs with VT tests to run simultaneously. Since there's no finer grained resource locking in Avocado-VT, this is a global lock that will prevent issues such as image corruption when two jobs are run at the same time.

  This new implementation will now check if existing lock files are stale, that is, they are leftovers from previous run. If the processes associated with these files are not present, the stale lock files are deleted, removing the need to clean them up manually. It also outputs better debugging information when failures to acquire lock.

The complete list of changes to Avocado-VT are available on Avocado-VT changelog.

#### Miscellaneous

While not officially part of this release, this development cycle saw the introduction of new tests on our avocado-misc-tests. Go check it out!

Finally, since Avocado and Avocado-VT are not newly born anymore, we decided to update information mentioning KVM-Autotest, virt-test on so on around the web. This will hopefully redirect new users to the Avocado community and avoid confusion.

Happy hacking and testing!

---

Sprint Theme: https://trello.com/c/Y6IIFXBS/732-sprint-theme

### 37.0 Trabant vs. South America

This is another proud announcement: Avocado release 37.0, aka "Trabant vs. South America", is now out!

Release documentation: Avocado 37.0

This release is yet another collection of bug fixes and some new features. Along with the same changes that made the 36.0lts release[1], this brings the following additional changes:

- TAP[2] version 12 support, bringing better integration with other test tools that accept this streaming format as input.

- Added niceties on Avocado's utility libraries "build" and "kernel", such as automatic parallelism and resource caching. It makes tests such as "linuxbuild.py" (and your similar tests) run up to 10 times faster.
- Fixed an issue where Avocado could leave processes behind after the test was finished.
- Fixed a bug where the configuration for tests data directory would be ignored.
- Fixed a bug where SIMPLE tests would not properly exit with WARN status.

For a complete list of changes please check the Avocado changelog[3].

For Avocado-VT, please check the full Avocado-VT changelog[4].

Happy hacking and testing!

---

[1] https://www.redhat.com/archives/avocado-devel/2016-May/msg00025.html
[2] https://en.wikipedia.org/wiki/Test_Anything_Protocol
[3] https://github.com/avocado-framework/avocado/compare/35.0...37.0
[4] https://github.com/avocado-framework/avocado-vt/compare/35.0...37.0
[5] http://avocado-framework.readthedocs.io/en/37.0/GetStartedGuide.html#installing-avocado
Sprint Theme: https://trello.com/c/XbIUqU1Y/673-sprint-theme

## 36.0 LTS

This is a very proud announcement: Avocado release 36.0lts, our very first "Long Term Stability" release, is now out!

Release documentation: Avocado 36.0

### LTS in a nutshell

This release marks the beginning of a special cycle that will last for 18 months. Avocado usage in production environments should favor the use of this LTS release, instead of non-LTS releases.

Bug fixes will be provided on the "36lts"[1] branch until, at least, September 2017. Minor releases, such as "36.1lts", "36.2lts" an so on, will be announced from time to time, incorporating those stability related improvements.

Keep in mind that no new feature will be added. For more information, please read the "Avocado Long Term Stability" RFC[2].

### Changes from 35.0

As mentioned in the release notes for the previous release (35.0), only bug fixes and other stability related changes would be added to what is now 36.0lts. For the complete list of changes, please check the GIT repo change log[3].

### Install avocado

The Avocado LTS packages are available on a separate repository, named "avocado-lts". These repositories are available for Fedora 22, Fedora 23, EPEL 6 and EPEL 7.

Updated ".repo" files are available on the usual locations:

- https://repos-avocadoproject.rhcloud.com/static/avocado-fedora.repo

- https://repos-avocadoproject.rhcloud.com/static/avocado-el.repo

Those repo files now contain definitions for both the "LTS" and regular repositories. Users interested in the LTS packages, should disable the regular repositories and enable the "avocado-lts" repo.

Instructions are available in our documentation on how to install either with packages or from source[4].

Happy hacking and testing!

---

[1] https://github.com/avocado-framework/avocado/tree/36lts
[2] https://www.redhat.com/archives/avocado-devel/2016-April/msg00038.html
[3] https://github.com/avocado-framework/avocado/compare/35.0...36.0lts
[4] http://avocado-framework.readthedocs.io/en/36lts/GetStartedGuide.html#installing-avocado

## 35.0 Mr. Robot

This is another proud announcement: Avocado release 35.0, aka "Mr. Robot", is now out!

This release, while a "regular" release, will also serve as a beta for our first "long term stability" (aka "lts") release. That means that the next release, will be version "36.0lts" and will receive only bug fixes and minor improvements. So, expect release 35.0 to be pretty much like "36.0lts" feature-wise. New features will make into the "37.0" release, to be released after "36.0lts". Read more about the details on the specific RFC[9].

The main changes in Avocado for this release are:

- A big round of fixes and on machine readable output formats, such as xunit (aka JUnit) and JSON. The xunit output, for instance, now includes tests with schema checking. This should make sure interoperability is even better on this release.
- Much more robust handling of test references, aka test URLs. Avocado now properly handles very long test references, and also test references with non-ascii characters.
- The avocado command line application now provides richer exit status[1]. If your application or custom script depends on the avocado exit status code, you should be fine as avocado still returns zero for success and non-zero for errors. On error conditions, though, the exit status code are richer and made of combinable (ORable) codes. This way it's possible to detect that, say, both a test failure and a job timeout occurred in a single execution.
- [SECURITY RELATED] The remote execution of tests (including in Virtual Machines) now allows for proper checks of host keys[2]. Without these checks, avocado is susceptible to a man-in-the-middle attack, by connecting and sending credentials to the wrong machine. This check is *disabled* by default, because users depend on this behavior when using machines without any prior knowledge such as cloud based virtual machines. Also, a bug in the underlying SSH library may prevent existing keys to be used if these are in ECDSA format[3]. There's an automated check in place to check for the resolution of the third party library bug. Expect this feature to be *enabled* by default in the upcoming releases.
- Pre/Post Job hooks. Avocado now defines a proper interface for extension/plugin writers to execute actions while a Job is runnning. Both Pre and Post hooks have access to the Job state (actually, the complete Job instance). Pre job hooks are called before tests are run, and post job hooks are called at the very end of the job (after tests would have usually finished executing).
- Pre/Post job scripts[4]. As a feature built on top of the Pre/Post job hooks described earlier, it's now possible to put executable scripts in a configurable location, such as */etc/avocado/scripts/job/pre.d* and have them called by Avocado before the execution of tests. The executed scripts will receive some information about the job via environment variables[5].
- The implementation of proper Test-IDs[6] in the test result directory.

Also, while not everything is (yet) translated into code, this release saw various and major RFCs, which are definitely shaping the future of Avocado. Among those:

- Introduce proper test IDs[6]
- Pre/Post *test* hooks[7]
- Multi-stream tests[8]
- Avocado maintainability and integration with avocado-vt[9]
- Improvements to job status (completely implemented)[10]

For a complete list of changes please check the Avocado changelog[11]. For Avocado-VT, please check the full Avocado-VT changelog[12].

### Install avocado

Instructions are available in our documentation on how to install either with packages or from source[13].

Updated RPM packages are be available in the project repos for Fedora 22, Fedora 23, EPEL 6 and EPEL 7.

### Packages

As a heads up, we still package the latest version of the various Avocado sub projects, such as the very popular Avocado-VT and the pretty much experimental Avocado-Virt and Avocado-Server projects.

For the upcoming releases, there will be changes in our package offers, with a greater focus on long term stability packages for Avocado. Other packages may still be offered as a convenience, or may see a change of ownership. All in the best interest of our users. If you have any concerns or questions, please let us know.

Happy hacking and testing!

---

[1] http://avocado-framework.readthedocs.org/en/35.0/ResultFormats.html#exit-codes
[2] https://github.com/avocado-framework/avocado/blob/35.0/etc/avocado/avocado.conf#L41
[3] https://github.com/avocado-framework/avocado/blob/35.0/selftests/functional/test_thirdparty_bugs.py#L17
[4] http://avocado-framework.readthedocs.org/en/35.0/ReferenceGuide.html#job-pre-and-post-scripts
[5] http://avocado-framework.readthedocs.org/en/35.0/ReferenceGuide.html#script-execution-environment
[6] https://www.redhat.com/archives/avocado-devel/2016-March/msg00024.html
[7] https://www.redhat.com/archives/avocado-devel/2016-April/msg00000.html
[8] https://www.redhat.com/archives/avocado-devel/2016-April/msg00042.html
[9] https://www.redhat.com/archives/avocado-devel/2016-April/msg00038.html
[10] https://www.redhat.com/archives/avocado-devel/2016-April/msg00010.html
[11] https://github.com/avocado-framework/avocado/compare/0.34.0...35.0
[13] https://github.com/avocado-framework/avocado-vt/compare/0.34.0...35.0
[12] http://avocado-framework.readthedocs.org/en/35.0/GetStartedGuide.html#installing-avocado
Sprint Theme: https://trello.com/c/7dWknPDJ/637-sprint-theme

### 0.34.0 The Hour of the Star

Hello to all test enthusiasts out there, specially to those that cherish, care or are just keeping an eye on the greenest test framework there is: Avocado release 0.34.0, aka The Hour of the Star, is now out!

The main changes in Avocado for this release are:

- A complete overhaul of the logging and output implementation. This means that all Avocado output uses the standard Python logging library making it very consistent and easy to understand [1].
- Based on the logging and output overhaul, the command line test runner is now very flexible with its output. A user can choose exactly what should be output. Examples include application output only, test output only, both application and test output or any other combination of the builtin streams. The user visible command line option that controls this behavior is *–show*, which is an application level option, that is, it's available to all avocado commands. [2]
- Besides the builtin streams, test writers can use the standard Python logging API to create new streams. These streams can be shown on the command line as mentioned before, or persisted automatically in the job results by means of the *–store-logging-stream* command line option. [3][4]
- The new *avocado.core.safeloader* module, intends to make it easier to to write new test loaders for various types of Python code. [5][6]
- Based on the new *avocado.core.safeloader* module, a contrib script called *avocado-find-unittests*, returns the name of unittest.TestCase based tests found on a given number of Python source code files. [7]
- Avocado is now able to run its own selftest suite. By leveraging the *avocado-find-unittests* contrib script and the External Runner [8] feature. A Makefile target is available, allowing developers to run *make selfcheck* to have the selftest suite run by Avocado. [9]
- Partial Python 3 support. A number of changes were introduced that allow concurrent Python 2 and 3 support on the same code base. Even though the support for Python 3 is still *incomplete*, the *avocado* command line application can already run some limited commands at this point.
- Asset fetcher utility library. This new utility library, and INSTRUMENTED test feature, allows users to transparently request external assets to be used in tests, having them cached for later use. [10]
- Further cleanups in the public namespace of the avocado Test class.
- [BUG FIX] Input from the local system was being passed to remote systems when running tests with either in remote systems or VMs.
- [BUG FIX] HTML report stability improvements, including better Unicode handling and support for other versions of the Pystache library.
- [BUG FIX] Atomic updates of the "latest" job symlink, allows for more reliable user experiences when running multiple parallel jobs.
- [BUG FIX] The avocado.core.data_dir module now dynamically checks the configuration system when deciding where the data directory should be located. This allows for later updates, such as when giving one extra *–config* parameter in the command line, to be applied consistently throughout the framework and test code.
- [MAINTENANCE] The CI jobs now run full checks on each commit on any proposed PR, not only on its topmost commit. This gives higher confidence that a commit in a series is not causing breakage that a later commit then inadvertently fixes.

For a complete list of changes please check the Avocado changelog[11].

For Avocado-VT, please check the full Avocado-VT changelog[12].

### Avocado Videos

As yet another way to let users know about what's available in Avocado, we're introducing short videos with very targeted content on our very own YouTube channel: https://www.youtube.com/channel/UCP4xob52XwRad0bU_8V28rQ

The first video available demonstrates a couple of new features related to the advanced logging mechanisms, introduced on this release: https://www.youtube.com/watch?v=8Ur_p5p6YiQ

### Install avocado

Instructions are available in our documentation on how to install either with packages or from source[13].

Updated RPM packages are be available in the project repos for Fedora 22, Fedora 23, EPEL 6 and EPEL 7.

Happy hacking and testing!

---

[1] http://avocado-framework.readthedocs.org/en/0.34.0/LoggingSystem.html
[2] http://avocado-framework.readthedocs.org/en/0.34.0/LoggingSystem.html#tweaking-the-ui
[3] http://avocado-framework.readthedocs.org/en/0.34.0/LoggingSystem.html#storing-custom-logs
[4] http://avocado-framework.readthedocs.org/en/0.34.0/WritingTests.html#advanced-logging-capabilities
[5] https://github.com/avocado-framework/avocado/blob/0.34.0/avocado/core/safeloader.py
[6] http://avocado-framework.readthedocs.org/en/0.34.0/api/core/avocado.core.html#module-avocado.core.safeloader
[7] https://github.com/avocado-framework/avocado/blob/0.34.0/contrib/avocado-find-unittests
[8] http://avocado-framework.readthedocs.org/en/0.34.0/GetStartedGuide.html#running-tests-with-an-external-runner
[9] https://github.com/avocado-framework/avocado/blob/0.34.0/Makefile#L33
[10] http://avocado-framework.readthedocs.org/en/0.34.0/WritingTests.html#fetching-asset-files
[11] https://github.com/avocado-framework/avocado/compare/0.33.0...0.34.0
[12] https://github.com/avocado-framework/avocado-vt/compare/0.33.0...0.34.0
[13] http://avocado-framework.readthedocs.org/en/latest/GetStartedGuide.html#installing-avocado
Sprint Theme: https://trello.com/c/QIbM3NvY/590-sprint-theme

## 0.33.0 Lemonade Joe or Horse Opera

Hello big farmers, backyard gardeners and supermarket reapers! Here is a new announcement to all the appreciators of the most delicious green fruit out here. Avocado release 0.33.0, aka, Lemonade Joe or Horse Opera, is now out!

The main changes in Avocado are:

- Minor refinements to the Job Replay feature introduced in the last release.
- More consistency naming for the status of tests that were not executed. Namely, the TEST_NA has been renamed to SKIP all across the internal code and user visible places.
- The avocado Test class has received some cleanups and improvements. Some attributes that back the class implementation but are not intended for users to rely upon are now hidden or removed. Additionally some the internal attributes have been turned into proper documented properties that users should feel confident to rely upon. Expect more work on this area, resulting in a cleaner and leaner base Test class on the upcoming releases.

- The avocado command line application used to show the main app help message even when help for a specific command was asked for. This has now been fixed.
- It's now possible to use the avocado process utility API to run privileged commands transparently via SUDO. Just add the "sudo=True" parameter to the API calls and have your system configured to allow that command without asking interactively for a password.
- The software manager and service utility API now knows about commands that require elevated privileges to be run, such as installing new packages and starting and stopping services (as opposed to querying packages and services status). Those utility APIs have been integrated with the new SUDO features allowing unprivileged users to install packages, start and stop services more easily, given that the system is properly configured to allow that.
- A nasty "fork bomb" situation was fixed. It was caused when a SIMPLE test written in Python used the Avocado's "main()" function to run itself.
- A bug that prevented SIMPLE tests from being run if Avocado was not given the absolute path of the executable has been fixed.
- A cleaner internal API for registering test result classes has been put into place. If you have written your own test result class, please take a look at avocado.core.result.register_test_result_class.
- Our CI jobs now also do quick "smoke" checks on every new commit (not only the PR's branch HEAD) that are proposed on github.
- A new utility function, binary_from_shell_cmd, has been added to process API allows to extract the executable to be run from complex command lines, including ones that set shell variable names.
- There have been internal changes to how parameters, including the internally used timeout parameter, are handled by the test loader.
- Test execution can now be PAUSED and RESUMED interactively! By hitting CTRL+Z on the Avocado command line application, all processes of the currently running test are PAUSED. By hitting CTRL+Z again, they are RESUMED.
- The Remote/VM runners have received some refactors, and most of the code that used to live on the result test classes have been moved to the test runner classes. The original goal was to fix a bug, but turns out test runners were more suitable to house some parts of the needed functionality.

For a complete list of changes please check the Avocado changelog[1].

For Avocado-VT, there were also many changes, including:

- A new utility function, get_guest_service_status, to get service status in a VM.
- A fix for ssh login timeout error on remote servers.
- Fixes for usb ehci on PowerPC.
- Fixes for the screenshot path, when on a remote host
- Added libvirt function to create volumes with by XML files
- Added utility function to get QEMU threads (get_qemu_threads)

And many other changes. Again, for a complete list of changes please check the Avocado-VT changelog[2].

### Install avocado

Instructions are available in our documentation on how to install either with packages or from source[3].

Updated RPM packages are be available in the project repos for Fedora 22, Fedora 23, EPEL 6 and EPEL 7.

Happy hacking and testing!

[1] https://github.com/avocado-framework/avocado/compare/0.32.0...0.33.0
[2] https://github.com/avocado-framework/avocado-vt/compare/0.32.0...0.33.0
[3] http://avocado-framework.readthedocs.org/en/latest/GetStartedGuide.html#installing-avocado
Sprint Theme: https://www.youtube.com/watch?v=H5Lg_14m-sM

## 0.32.0 Road Runner

Hi everyone! A new year brings a new Avocado release as the result of Sprint #32: Avocado 0.32.0, aka, "Road Runner".

The major changes introduced in the previous releases were put to trial on this release cycle, and as a result, we have responded with documentation updates and also many fixes. This release also marks the introduction of a great feature by a new member of our team: Amador Pahim brought us the Job Replay feature! Kudos!!!

So, for Avocado the main changes are:

- Job Replay: users can now easily re-run previous jobs by using the –replay command line option. This will re-run the job with the same tests, configuration and multiplexer variants that were used on the origin one. By using –replay-test-status, users can, for example, only rerun the failed tests of the previous job. For more check our docs[1].
- Documentation changes in response to our users feedback, specially regarding the setup.py install/develop requirement.
- Fixed the static detection of test methods when using repeated names.
- Ported some Autotest tests to Avocado, now available on their own repository[2]. More contributions here are very welcome!

For a complete list of changes please check the Avocado changelog[3].

For Avocado-VT, there were also many changes, including:

- Major documentation updates, making them simpler and more in sync with the Avocado documentation style.
- Refactor of the code under the avocado_vt namespace. Previously most of the code lived under the plugin file itself, now it better resembles the structure in Avocado and the plugin files are hopefully easier to grasp.

Again, for a complete list of changes please check the Avocado-VT changelog[4].

### Install avocado

Instructions are available in our documentation on how to install either with packages or from source[5].

Updated RPM packages are be available in the project repos for Fedora 22, Fedora 23, EPEL 6 and EPEL 7.

Happy hacking and testing!

[1] http://avocado-framework.readthedocs.org/en/0.32.0/Replay.html
[2] http://github.com/avocado-framework/avocado-misc-tests
[3] https://github.com/avocado-framework/avocado/compare/0.31.0...0.32.0

[4] https://github.com/avocado-framework/avocado-vt/compare/0.31.0...0.32.0
[5] http://avocado-framework.readthedocs.org/en/0.32.0/GetStartedGuide.html

## 0.31.0 Lucky Luke

Hi everyone! Right on time for the holidays, Avocado reaches the end of Sprint 31, and together with it, we're very happy to announce a brand new release! This version brings stability fixes and improvements to both Avocado and Avocado-VT, some new features and a major redesign of our plugin architecture.

For Avocado the main changes are:

- It's now possible to register callback functions to be executed when all tests finish, that is, at the end of a particular job[1].
- The software manager utility library received a lot of love on the Debian side of things. If you're writing tests that install software packages on Debian systems, you may be in for some nice treats and much more reliable results.
- Passing malformed commands (such as ones that can not be properly split by the standard shlex library) to the process utility library is now better dealt with.
- The test runner code received some refactors and it's a lot easier to follow. If you want to understand how the Avocado test runner communicates with the processes that run the test themselves, you may have a much better code reading experience now.
- Updated inspektor to the latest and greatest, so that our code is kept is shiny and good looking (and performing) as possible.
- Fixes to the utility GIT library when using a specific local branch name.
- Changes that allow our selftest suite to run properly on virtualenvs.
- Proper installation requirements definition for Python 2.6 systems.
- A completely new plugin architecture[2]. Now we offload all plugin discovery and loading to the Stevedore library. Avocado now defines precise (and simpler) interfaces for plugin writers. Please be aware that the public and documented interfaces for plugins, at the moment, allows adding new commands to the avocado command line app, or adding new options to existing commands. Other functionality can be achived by "abusing" the core avocado API from within plugins. Our goal is to expand the interfaces so that other areas of the framework can be extended just as easily.

For a complete list of changes please check the Avocado changelog[3].

Avocado-VT received just too many fixes and improvements to list. Please refer to the changelog[4] for more information.

### Install avocado

Instructions are available in our documentation on how to install either with packages or from source[5].

Within a couple of hours, updated RPM packages will be available in the project repos for Fedora 22, Fedora 23, EPEL 6 and EPEL 7.

Happy hacking and testing!

---

[1] http://avocado-framework.readthedocs.org/en/0.31.0/ReferenceGuide.html#job-cleanup

[2] http://avocado-framework.readthedocs.org/en/0.31.0/Plugins.html
[3] https://github.com/avocado-framework/avocado/compare/0.30.0...0.31.0
[4] https://github.com/avocado-framework/avocado-vt/compare/0.30.0...0.31.0
[5] http://avocado-framework.readthedocs.org/en/0.31.0/GetStartedGuide.html

## 0.30.0 Jimmy's Hall

Hello! Avocado reaches the end of Sprint 30, and with it, we have a new release available! This version brings stability fixes and improvements to both Avocado and Avocado-vt.

As software doesn't spring out of life itself, we'd like to acknowledge the major contributions by Lucas (AKA lmr) since the dawn of time for Avocado (and earlier projects like Autotest and virt-test). Although the Avocado team at Red Hat was hit by some changes, we're already extremely happy to see that this major contributor (and good friend) has not gone too far.

Now back to the more informational part of the release notes. For Avocado the main changes are:

- New RPM repository location, check the docs[1] for instructions on how to install the latest releases
- Makefile rules for building RPMs are now based on mock, to ensure sound dependencies
- Packaged versions are now available for Fedora 22, newly released Fedora 23, EL6 and EL7
- The software manager utility library now supports DNF
- The avocado test runner now supports a dry run mode, which allows users to check how a job would be executed, including tests that would be found and parameters that would be passed to it. This is currently complementary to the avocado list command.
- The avocado test runner now supports running simple tests with parameters. This may come in handy for simple use cases when Avocado will wrap a test suite, but the test suite needs some command line arguments.

Avocado-vt also received many bugfixes[3]. Please refer to the changelog for more information.

### Install avocado

Instructions are available in our documentation on how to install either with packages or from source[1].

Happy hacking and testing!

---

[1] http://avocado-framework.readthedocs.org/en/0.30.0/GetStartedGuide.html
[2] https://github.com/avocado-framework/avocado/compare/0.29.0...0.30.0
[3] https://github.com/avocado-framework/avocado-vt/compare/0.29.0...0.30.0

## 0.29.0 Steven Universe

Hello! Avocado reaches the end of Sprint 29, and with it, we have a great release coming! This version of avocado once brings new features and plenty of bugfixes:

- The remote and VM plugins now work with –multiplex, so that you can use both features in conjunction. * The VM plugin can now auto detect the IP of a given libvirt domain you pass to it, reducing typing and providing an easier and more pleasant experience. * Temporary directories are now properly cleaned up and no re-creation of directories happens, making avocado more secure.

- Avocado docs are now also tagged by release. You can see the specific documentation of this one at our readthedocs page [1]
- Test introspection/listing is safer: Now avocado does not load python modules to introspect its contents, an alternative method, based on the Python AST parser is used, which means now avocado will not load possible badly written/malicious code at listing stage. You can find more about that in our test resolution documentation [2]
- You can now specify low level loaders to avocado to customize your test running experience. You can learn more about that in the Test Discovery documentation [3]
- The usual many bugfixes and polishing commits. You can see the full amount of 96 commits at [4]

For our Avocado VT plugin, the main changes are:

- The vt-bootstrap process is now more robust against users interrupting previous bootstrap attempts
- Some issues with RPM install in RHEL hosts were fixed
- Issues with unsafe temporary directories were fixed, making the VT tests more secure.
- Issues with unattended installed were fixed
- Now the address of the virbr0 bridge is properly auto detected, which means that our unattended installation content server will work out of the box as long as you have a working virbr0 bridge.

Install avocado

As usual, go to https://copr.fedoraproject.org/coprs/lmr/Autotest/ to install our YUM/DNF repo and get the latest goodies!

Happy hacking and testing!

---

[1] http://avocado-framework.readthedocs.org/en/0.29.0
[2] http://avocado-framework.readthedocs.org/en/0.29.0/ReferenceGuide.html#test-resolution
[3] http://avocado-framework.readthedocs.org/en/0.29.0/Loaders.html
[4] https://github.com/avocado-framework/avocado/compare/0.28.0...0.29.0

## 0.28.0 Jára Cimrman, The Investigation of the Missing Class Register

This release basically polishes avocado, fixing a number of small usability issues and bugs, and debuts avocado-vt as the official virt-test replacement!

Let's go with the changes from our last release, 0.27.0:

Changes in avocado:

- The avocado human output received another stream of tweaks and it's more compact, while still being informative. Here's an example:

```
JOB ID     : f2f5060440bd57cba646c1f223ec8c40d03f539b
JOB LOG : /home/user/avocado/job-results/job-2015-07-27T17.13-f2f5060/job.log
TESTS      : 1
 (1/1) passtest.py:PassTest.test: PASS (0.00 s)
RESULTS : PASS 1 | ERROR 0 | FAIL 0 | SKIP 0 | WARN 0 | INTERRUPT 0
JOB HTML : /home/user/avocado/job-results/job-2015-07-27T17.13-f2f5060/html/
↪results.html
TIME       : 0.00 s
```

- The unittest system was completely revamped, paving the way for making avocado self-testable! Stay tuned for what we have on store.
- Many bugfixes. Check [1] for more details.

Changes in avocado-vt:

- The Spice Test provider has been separated from tp-qemu, and changes reflected in avocado-vt [2].
- A number of bugfixes found by our contributors in the process of moving avocado-vt into the official virt-testing project. Check [3] for more details.

See you in a few weeks for our next release! Happy testing!

The avocado development team

---

[1] https://github.com/avocado-framework/avocado/compare/0.27.0...0.28.0
[2] https://github.com/avocado-framework/avocado-vt/commit/fd9b29bbf77d7f0f3041e66a66517f9ba6b8bf48
[3] https://github.com/avocado-framework/avocado-vt/compare/0.27.0...0.28.0

## 0.27.1

Hi guys, we're up to a new avocado release! It's basically a bugfix release, with a few usability tweaks.

- The avocado human output received some extra tweaks. Here's how it looks now:

```
$ avocado run passtest
JOB ID     : f186c729dd234c8fdf4a46f297ff0863684e2955
JOB LOG : /home/lmr/avocado/job-results/job-2015-08-15T08.09-f186c72/job.log
TESTS      : 1
(1/1) passtest.py:PassTest.test: PASS (0.00 s)
RESULTS : PASS 1 | ERROR 0 | FAIL 0 | SKIP 0 | WARN 0 | INTERRUPT 0
JOB HTML : /home/lmr/avocado/job-results/job-2015-08-15T08.09-f186c72/html/
↪results.html
TIME       : 0.00 s
```

- Bugfixes. You may refer to [1] for the full list of 58 commits.

Changes in avocado-vt:

- Bugfixes. In particular, a lot of issues related to –vt-type libvirt were fixed and now that backend is fully functional.

News:

We, the people that bring you avocado will be at LinuxCon North America 2015 (Aug 17-19). If you are attending, please don't forget to drop by and say hello to yours truly (lmr). And of course, consider attending my presentation on avocado [2].

---

[1] https://github.com/avocado-framework/avocado/compare/0.27.0...0.27.1
[2] http://sched.co/3Xh9

## 0.27.0 Terminator: Genisys

Hi guys, here I am, announcing yet another avocado release! The most exciting news for this release is that our avocado-vt plugin was merged with the virt-test project. The avocado-vt plugin will be very important for QEMU/KVM/Libvirt developers, so the main avocado received updates to better support the goal of having a good quality avocado-vt.

Changes in avocado:

- The avocado human output received some tweaks and it's more compact, while still being informative. Here's an example:

```
JOB ID     : f2f5060440bd57cba646c1f223ec8c40d03f539b
JOB LOG : /home/user/avocado/job-results/job-2015-07-27T17.13-f2f5060/job.log
JOB HTML : /home/user/avocado/job-results/job-2015-07-27T17.13-f2f5060/html/
↪results.html
TESTS      : 1
 (1/1) passtest.py:PassTest.test: PASS (0.00 s)
RESULTS : PASS 1 | ERROR 0 | FAIL 0 | SKIP 0 | WARN 0 | INTERRUPT 0
TIME       : 0.00 s
```

- The avocado test loader was refactored and behaves more consistently in different test loading scenarios.
- The *utils* API received new modules and functions:
- NEW avocado.utils.cpu: APIs related to CPU information on linux boxes [1]
- NEW avocado.utils.git: APIs to clone/update git repos [2]
- NEW avocado.utils.iso9660: Get information about ISO files [3]
- NEW avocado.utils.service: APIs to control services on linux boxes (systemv and systemd) [4]
- NEW avocado.utils.output: APIs that help avocado based CLI programs to display results to users [5]
- UPDATE avocado.utils.download: Add url_download_interactive
- UPDATE avocado.utils.download: Add new params to get_file
- Bugfixes. You may refer to [6] for the full list of 64 commits.

Changes in avocado-vt:

- Merged virt-test into avocado-vt. Basically, the virt-test core library (virttest) replaced most uses of autotest by equivalent avocado API calls, and its code was brought up to the virt-test repository [7]. This means, among other things, that you can simply install avocado-vt through RPM and enjoy all the virt tests without having to clone another repository manually to bootstrap your tests. More details about the process will be sent on an e-mail to the avocado and virt-test mailing lists. Please go to [7] for instructions on how to get started with all our new tools.

See you in a couple of weeks for our next release! Happy testing!

---

[1] http://avocado-framework.readthedocs.org/en/latest/api/utils/avocado.utils.html#module-avocado.utils.cpu
[2] http://avocado-framework.readthedocs.org/en/latest/api/utils/avocado.utils.html#module-avocado.utils.git
[3] http://avocado-framework.readthedocs.org/en/latest/api/utils/avocado.utils.html#module-avocado.utils.iso9660
[4] http://avocado-framework.readthedocs.org/en/latest/api/utils/avocado.utils.html#module-avocado.utils.service
[5] http://avocado-framework.readthedocs.org/en/latest/api/utils/avocado.utils.html#module-avocado.utils.output
[6] https://github.com/avocado-framework/avocado/compare/0.26.0...0.27.0

[7] https://github.com/avocado-framework/avocado-vt/commit/20dd39ef00db712f78419f07b10b8f8edbd19942
[8] http://avocado-vt.readthedocs.org/en/latest/GetStartedGuide.html

## 0.26.0 The Office

Hi guys, I'm here to announce avocado 0.26.0. This release was dedicated to polish aspects of the avocado user experience, such as documentation and behavior.

Changes

- Now avocado tests that raise exceptions that don't inherit from *avocado.core.exceptions.TestBaseException* now will be marked as ERRORs. This change was made to make avocado to have clearly defined test statuses. A new decorator, *avocado.fail_on_error* was added to let arbitrary exceptions to raise errors, if users need a more relaxed behavior.
- The avocado.Test() utility method skip() now can only be called from inside the setUp() method. This was made because by definition, if we get to the test execution step, by definition it can't be skipped anymore. It's important to keep the concepts clear and well separated if we want to give users a good experience.
- More documentation polish and updates. Make sure you check out our documentation website http://avocado-framework.readthedocs.org/en/latest/.
- A number of avocado command line options and help text was reviewed and updated.
- A new, leaner and mobile friendly version of the avocado website is live. Please check http://avocado-framework.github.io/ for more information.
- We have the first version of the avocado dashboard! avocado dashboard is the initial version of an avocado web interface, and will serve as the frontend to our testing database. You can check out a screenshot here: https://cloud.githubusercontent.com/assets/296807/8536678/dc5da720-242a-11e5-921c-6abd46e0f51e.png
- And the usual bugfixes. You can take a look at the full list of 68 commits here: https://github.com/avocado-framework/avocado/compare/0.25.0...0.26.0

## 0.25.0 Blade

Hi guys, I'm here to announce the newest avocado release, 0.25.0. This is an important milestone in avocado development, and we would like to invite you to be a part of the development process, by contributing PRs, testing and giving feedback on the test runner's usability and new plugins we came up with.

### What to expect

This is the first release aimed for general use. We did our best to deliver a coherent and enjoyable experience, but keep in mind that it's a young project, so please set your expectations accordingly. What is expected to work well:

- Running avocado 'instrumented' tests
- Running arbitrary executables as tests
- Automatic test discovery and run of tests on directories
- xUnit/JSON report

### Known Issues

- HTML report of test jobs with multiplexed tests has a minor naming display issue that is scheduled to be fixed by next release.
- avocado-vt might fail to load if virt-test was not properly bootstrapped. Make sure you always run bootstrap in the virt-test directory on any virt-test git updates to prevent the issue. Next release will have more mechanisms to give the user better error messages on tough to judge situations (virt-test repo with stale or invalid config files that need update).

### Changes

- The Avocado API has been greatly streamlined. After a long discussion and several rounds of reviews and planning, now we have a clear separation of what is intended as functions useful for test developers and plugin/core developers:
- avocado.core is intended for plugin/core developers. Things are more fluid on this space, so that we can move fast with development
- avocado.utils is a generic library, with functions we found out to be useful for a variety of tests and core code alike.
- avocado has some symbols exposed at its top level, with the test API:
- our Test() class, derived from the unittest.TestCase() class
- a main() entry point, similar to unittest.main()
- VERSION, that gives the user the avocado version (eg 0.25.0).

Those symbols and classes/APIs will be changed more carefully, and release notes will certainly contain API update notices. In other words, we'll be a lot more mindful of changes in this area, to reduce the maintenance cost of writing avocado tests.

We believe this more strict separation between the available APIs will help test developers to quickly identify what they need for test development, and reduce following a fast moving target, what usually happens when we have a new project that does not have clear policies behind its API design.

- There's a new plugin added to the avocado project: avocado-vt. This plugin acts as a wrapper for the virt-test test suite (https://github.com/autotest/virt-test), allowing people to use avocado to list and run the tests available for that test suite. This allows people to leverage a number of the new cool avocado features for the virt tests themselves:
- HTML reports, a commonly asked feature for the virt-test suite. You can see a screenshot of what the report looks like here: https://cloud.githubusercontent.com/assets/296807/7406339/7699689e-eed7-11e4-9214-38a678c105ec.png
- You can run virt-tests on arbitrary order, and multiple instances of a given test, something that is also currently not possible with the virt test runner (also a commonly asked feature for the suite.
- System info collection. It's a flexible feature, you get to configureeasily what gets logged/recorded between tests.
- The avocado multiplexer (test matrix representation/generation system) also received a lot of work and fixes during this release. One of the most visible (and cool) features of 0.25.0 is the new, improved –tree representation of the multiplexer file:

```
$ avocado multiplex examples/mux-environment.yaml -tc
   run
        hw
```

```
            cpu
                intel
                → cpu_CFLAGS: -march=core2
                amd
                → cpu_CFLAGS: -march=athlon64
                arm
                 → cpu_CFLAGS: -mabi=apcs-gnu -march=armv8-a -mtune=arm8
            disk
                 scsi
                 → disk_type: scsi
                 virtio
                  → disk_type: virtio
        distro
            fedora
            → init: systemd
            mint
             → init: systemv
        env
             debug
             → opt_CFLAGS: -O0 -g
             prod
              → opt_CFLAGS: -O2
```

We hope you find the multiplexer useful and enjoyable.

- If an avocado plugin fails to load, due to factors such as missing dependencies, environment problems and misconfiguration, in order to notify users and make them mindful of what it takes to fix the root causes for the loading errors, those errors are displayed in the avocado stderr stream.

However, often we can't fix the problem right now and don't need the constant stderr nagging. If that's the case, you can set in your local config file:

```
[plugins]
# Suppress notification about broken plugins in the app standard error.
# Add the name of each broken plugin you want to suppress the notification
# in the list. The names can be easily seen from the stderr messages. Example:
# avocado.core.plugins.htmlresult ImportError No module named pystache
# add 'avocado.core.plugins.htmlresult' as an element of the list below.
skip_broken_plugin_notification = []
```

- Our documentation has received a big review, that led to a number of improvements. Those can be seen online (http://avocado-framework.readthedocs.org/en/latest/), but if you feel so inclined, you can build the documentation for local viewing, provided that you have the sphinx python package installed by executing:

```
$ make -C docs html
```

Of course, if you find places where our documentation needs fixes/improvements, please send us a PR and we'll gladly review it.

- As one would expect, many bugs were fixed. You can take a look at the full list of 156 commits here: https://github.com/avocado-framework/avocado/compare/0.24.0...0.25.0

CHAPTER 20

# Request For Comments (RFCs)

## Request For Comments (RFCs)

The following list contains archivals of accepted, Request For Comments posted and discussed on the Avocado Devel Mailing List.

### RFC: Long Term Stability

This RFC contains proposals and clarifications regarding the maintenance and release processes of Avocado.

We understand there are multiple teams currently depending on the stability of Avocado and we don't want their work to be disrupted by incompatibilities nor instabilities in new releases.

This version is a minor update to previous versions of the same RFC (see *Changelog*) which drove the release of Avocado 36.0 LTS. The Avocado team has plans for a new LTS release in the near future, so please consider reading and providing feedback on the proposals here.

#### TL;DR

We plan to keep the current approach of sprint releases every 3-4 weeks, but we're introducing "Long Term Stability" releases which should be adopted in production environments where users can't keep up with frequent upgrades.

#### Introduction

We make new releases of Avocado every 3-4 weeks on average. In theory at least, we're very careful with backwards compatibility. We test Avocado for regressions and we try to document any issues, so upgrading to a new version should be (again, in theory) safe.

But in practice both intended and unintended changes are introduced during development, and both can be frustrating for conservative users. We also understand it's not feasible for users to upgrade Avocado very frequently in a production environment.

The objective of this RFC is to clarify our maintenance practices and introduce Long Term Stability (LTS) releases, which are intended to solve, or at least mitigate, these problems.

### Our definition of maintained, or stable

First of all, Avocado and its sub-projects are provided 'AS IS' and WITHOUT ANY WARRANTY, as described in the LICENSE file.

The process described here doesn't imply any commitments or promises. It's just a set of best practices and recommendations.

When something is identified as "stable" or "maintained", it means the development community makes a conscious effort to keep it working and consider reports of bugs and issues as high priorities. Fixes submitted for these issues will also be considered high priorities, although they will be accepted only if they pass the general acceptance criteria for new contributions (design, quality, documentation, testing, etc), at the development team discretion.

### Maintained projects and platforms

The only maintained project as of today is the Avocado Test Runner, including its APIs and core plugins (the contents of the main avocado git repository).

Other projects kept under the "Avocado Umbrella" in github may be maintained by different teams (e.g.: Avocado-VT) or be considered experimental (e.g.: avocado-server and avocado-virt).

More about Avocado-VT in its own section further down.

As a general rule, fixes and bug reports for Avocado when running in any modern Linux distribution are welcome.

But given the limited capacity of the development team, packaged versions of Avocado will be tested and maintained only for the following Linux distributions:

- RHEL 7.x (latest)
- Fedora (stable releases from the Fedora projects)

Currently all packages produced by the Avocado projects are "noarch". That means that they could be installable on any hardware platform. Still, the development team will currently attempt to provide versions that are stable for the following platforms:

- x86
- ppc64le

Contributions from the community to maintain other platforms and operating systems are very welcome.

The lists above may change without prior notice.

### Avocado Releases

The proposal is to have two different types of Avocado releases:

### Sprint Releases

(This is the model we currently adopt in Avocado)

They happen every 3-4 weeks (the schedule is not fixed) and their versions are numbered serially, with decimal digits in the format <major>.<minor>. Examples: 47.0, 48.0, 49.0. Minor releases are rare, but necessary to correct some major issue with the original release (47.1, 47.2, etc).

Only the latest Sprint Release is maintained.

In Sprint Releases we make a conscious effort to keep backwards compatibility with the previous version (APIs and behavior) and as a general rule and best practice, incompatible changes in Sprint Releases should be documented in the release notes and if possible deprecated slowly, to give users time to adapt their environments.

But we understand changes are inevitable as the software evolves and therefore there's no absolute promise for API and behavioral stability.

### Long Term Stability (LTS) Releases

LTS releases should happen whenever the team feels the code is stable enough to be maintained for a longer period of time, ideally once or twice per year (no fixed schedule).

They should be maintained for 18 months, receiving fixes for major bugs in the form of minor (sub-)releases. With the exception of these fixes, no API or behavior should change in a minor LTS release.

They will be versioned just like Sprint Releases, so looking at the version number alone will not reveal the differentiate release process and stability characteristics.

In practice each major LTS release will imply in the creation of a git branch where only important issues affecting users will be fixed, usually as a backport of a fix initially applied upstream. The code in a LTS branch is stable, frozen for new features.

Notice that although within a LTS release there's a expectation of stability because the code is frozen, different (major) LTS releases may include changes in behavior, API incompatibilities and new features. The development team will make a considerable effort to minimize and properly document these changes (changes when comparing it to the last major LTS release).

Sprint Releases are replaced by LTS releases. I.e., in the cycle when 52.0 (LTS) is released, that's also the version used as a Sprint Release (there's no 52.0 – non LTS – in this case).

New LTS releases should be done carefully, with ample time for announcements, testing and documentation. It's recommended that one or two sprints are dedicated as preparations for a LTS release, with a Sprint Release serving as a "LTS beta" release.

Similarly, there should be announcements about the end-of-life (EOL) of a LTS release once it approaches its 18 months of life.

### Deployment details

Sprint and LTS releases, when packaged, whenever possible, will be preferably distributed through different package channels (repositories).

This is possible for repository types such as *YUM/DNF repos*. In such cases, users can disable the regular channel, and enable the LTS version. A request for the installation of Avocado packages will fetch the latest version available in the enabled repository. If the LTS repository channel is enabled, the packages will receive minor updates (bugfixes only), until a new LTS version is released (roughly every 12 months).

If the non-LTS channel is enabled, users will receive updates every 3-4 weeks.

On other types of repos such as PyPI which have no concept of "sub-repos" or "channels", users can request a version smaller than the version that succeeds the current LTS to get the latest LTS (including minor releases). Suppose the current LTS major version is 52, but there have been minor releases 52.1 and 52.2. By running:

```
pip install 'avocado-framework<53.0'
```

pip provide LTS version 52.2. If 52.3 gets released, they will be automatically deployed instead. When a new LTS is released, users would still get the latest minor release from the 52.0 series, unless they update the version specification.

The existence of LTS releases should never be used as an excuse to break a Sprint Release or to introduce gratuitous incompatibilities there. In other words, Sprint Releases should still be taken seriously, just as they are today.

### Timeline example

Consider the release numbers as date markers. The bullet points beneath them are information about the release itself or events that can happen anytime between one release and the other. Assume each sprint is taking 3 weeks.

**36.0**

- **LTS** release (the only LTS release available at the time of writing)

**37.0 .. 49.0**

- sprint releases
- 36.1 LTS release
- 36.2 LTS release
- 36.3 LTS release
- 36.4 LTS release

**50.0**

- sprint release
- start preparing a LTS release, so 51.0 will be a **beta LTS**

**51.0**

- sprint release
- **beta LTS** release

**52.0**

- **LTS** release
- 52lts branch is created
- packages go into LTS repo
- both **36.x LTS** and **52.x LTS** maintained from this point on

**53.0**

- sprint release
- minor bug that affects 52.0 is found, fix gets added to master and 52lts branches
- bug does **not** affect 36.x LTS, so a backport is **not** added to the 36lts branch

**54.0**

- sprint release 54.0
- LTS release 52.1
- minor bug that also affects 52.x LTS and 36.x LTS is found, fix gets added to master, 52lts and 36lts branches

**55.0**

- sprint release
- LTS release 36.5
- LTS release 52.2
- critical bug that affects 52.2 *only* is found, fix gets added to 52lts and **52.3 LTS is immediately released**

**56.0**

- sprint release

**57.0**

- sprint release

**58.0**

- sprint release

**59.0**

- sprint release
- EOL for **36.x LTS** (18 months since the release of 36.0), 36lts branch is frozen permanently.

A few points are worth taking notice here:

- Multiple LTS releases can co-exist before EOL
- Bug discovery can happen at any time
- The bugfix occurs ASAP after its discovery
- The severity of the defect determines the timing of the release
  - moderate and minor bugfixes to lts branches are held until the next sprint release
  - critical bugs are released asynchronously, without waiting for the next sprint release

### Avocado-VT

Avocado-VT is an Avocado plugin that allows "VT tests" to be run inside Avocado. It's a third-party project maintained mostly by Engineers from Red Hat QE with assistance from the Avocado team and other community members.

It's a general consensus that QE teams use Avocado-VT directly from git, usually following the master branch, which they control.

There's no official maintenance or stability statement for Avocado-VT. Even though the upstream community is quite friendly and open to both contributions and bug reports, Avocado-VT is made available without any promises for compatibility or supportability.

When packaged and versioned, Avocado-VT rpms should be considered just snapshots, available in packaged form as a convenience to users outside of the Avocado-VT development community. Again, they are made available without any promises of compatibility or stability.

- Which Avocado version should be used by Avocado-VT?

  This is up to the Avocado-VT community to decide, but the current consensus is that to guarantee some stability in production environments, Avocado-VT should stick to a specific LTS release of Avocado. In other words, the Avocado team recommends production users of Avocado-VT not to install Avocado from its master branch or upgrade it from Sprint Releases.

Given each LTS release will be maintained for 18 months, it should be reasonable to expect Avocado-VT to upgrade to a new LTS release once a year or so. This process will be done with support from the Avocado team to avoid disruptions, with proper coordination via the avocado mailing lists.

In practice the Avocado development team will keep watching Avocado-VT to detect and document incompatibilities, so when the time comes to do an upgrade in production, it's expected that it should happen smoothly.

- Will it be possible to use the latest Avocado and Avocado-VT together?

  Users are welcome to *try* this combination. The Avocado development team itself will do it internally as a way to monitor incompatibilities and regressions.

  Whenever Avocado is released, a matching versioned snapshot of Avocado-VT will be made. Packages containing those Avocado-VT snapshots, for convenience only, will be made available in the regular Avocado repository.

### Changelog

Changes from Version 4:

- Moved changelog to the bottom of the document
- Changed wording on bug handling for LTS releases ("important issues")
- Removed ppc64 (big endian) from list of platforms
- If bugs also affect older LTS release during the transition period, a backport will also be added to the corresponding branch
- Further work on the *Timeline example*, adding summary of important points and more release examples, such as the whole list of 36.x releases and the (fictional) 36.5 and 52.3

Changes from Version 3:

- Converted formatting to REStructuredText
- Replaced "me" mentions on version 1 changelog with proper name (Ademar Reis)
- Renamed section "Misc Details" to *Deployment Details*
- Renamed "avocado-vt" to "Avocado-VT"
- Start the timeline example with version 36.0
- Be explicit on timeline example that a minor bug did not generate an immediate release

Changes from Version 2:

- Wording changes on second paragraph ("... nor instabilities...")
- Clarified on "Introduction" that change of behavior is introduced between regular releases
- Updated distro versions for which official packages are built
- Add more clear explanation on official packages on the various hardware platforms
- Used more recent version numbers as examples, and the planned new LTS version too
- Explain how users can get the LTS version when using tools such as pip
- Simplified the timeline example, with examples that will possibly match the future versions and releases
- Documented current status of Avocado-VT releases and packages

Changes from Version 1:

- Changed "Support" to "Stability" and "supported" to "maintained" [Jeff Nelson]

- Misc improvements and clarifications in the supportability/stability statements [Jeff Nelson, Ademar Reis]
- Fixed a few typos [Jeff Nelson, Ademar Reis]

## Indices and tables

- genindex
- modindex
- search

# Python Module Index

## a

# Index

## A

## B

## C

## D

## E

## F



## G

## H



## I

## J



## K



## L

## M

## N



## O



## P

## R

## S

## T

## U

## V



## W



## X



## Y



## Z