avocado Documentation

Release 92.0

Avocado Development Team

Oct 21, 2021

Quick Start

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.

CHAPTER 1

How does it work?

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.

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     : 3a5c4c51ceb5369f23702efb10b4209b111141b2
JOB LOG    : $HOME/avocado/job-results/job-2019-10-31T10.34-3a5c4c5/job.log
 (1/1) /bin/true: PASS (0.04 s)
RESULTS    : PASS 1 | ERROR 0 | FAIL 0 | SKIP 0 | WARN 0 | INTERRUPT 0 | CANCEL 0
JOB TIME   : 0.15 s
```

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**.

Tip: See more at the Test types section on the Avocado User's Guide.

CHAPTER 2

Why should I use it?

2.1 Multiple result formats

A regular run of Avocado will present the test results on standard output, a nice and colored report useful for human beings. But results for machines can also be generated.

Check the job-results folder (`$HOME/avocado/job-results/latest/`) to see the outputs.

Currently we support, out of box, the following output formats:

- **xUnit**: an XML format that contains test results in a structured form, and are used by other test automation projects, such as jenkins.
- **JSON**: a widely used data exchange format. The JSON Avocado plugin outputs job information, similarly to the xunit output plugin.
- **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).

Note: You can see the results of the latest job inside the folder `$HOME/avocado/job-results/latest/`. You can also specify at the command line the options `--xunit`, `--json` or `--tap` followed by a filename. Avocado will write the output on the specified filename.

When it comes to outputs, Avocado is very flexible. You can check the various **output plugins**. If you need something more sophisticated, visit our plugins section.

2.2 Sysinfo data collector

Avocado comes with a sysinfo plugin, which automatically gathers some system information per each job or even between tests. This is very helpful when trying to identify the cause of a test failure.

Check out the files stored at `$HOME/avocado/job-results/latest/sysinfo/`:

```
$ ls $HOME/avocado/job-results/latest/sysinfo/pre/
'brctl show'           hostname             modules
 cmdline              'ifconfig -a'         mounts
 cpuinfo               installed_packages  'numactl --hardware show'
 current_clocksource   interrupts           partitions
'df -mP'              'ip link'             scaling_governor
 dmesg                'ld --version'       'uname -a'
 dmidecode             lscpu                uptime
'fdisk -l'            'lspci -vvnn'         version
'gcc --version'        meminfo
```

For more information about sysinfo collector, please consult the Avocado User's Guide.

2.3 Job Replay and Job Diff

In order to reproduce a given job using the same data, one can use the `replay` subcommand, 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.

Example:

```
$ avocado 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
```

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

 ...
```

2.4 Extensible by plugins

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

For more information about plugins, please visit the Plugin System section on the Avocado User's Guide.

2.5 Utility libraries

When writing tests, developers often need to perform basic tasks on OS and end up having to implement these routines just to run they tests.

Avocado has **more than 40** *utility modules* that helps you to perform basic operations.

Below a small subset of our utility modules:

- **utils.vmimage**: This utility provides a API to download/cache VM images (QCOW) from the official distributions repositories.
- **utils.memory**: Provides information about memory usage.
- **utils.cpu**: Get information from the current's machine CPU.
- **utils.software_manager**: Software package management library.
- **utils.download**: Methods to download URLs and regular files.
- **utils.archive**: Module to help extract and create compressed archives.

CHAPTER 3

Avocado Python API

If the command-line is limiting you, then you can use our new API and create custom jobs and test suites:

```
import sys

from avocado.core.job import Job

with Job.from_config({'run.references': ['/bin/true']}) as job:
    sys.exit(job.run())
```

CHAPTER 4

How to install

It is super easy, just run the follow command:

```
$ pip3 install --user avocado-framework
```

This will install the avocado command in your home directory.

Note: For more details and alternative methods, please visit the Installing section on Avocado User's Guide

CHAPTER 5

Documentation

Please use the following links for full documentation, including installation methods, tutorials and API or browse this site for more content.

- latest release
- development version

CHAPTER 6

Bugs/Requests

Please use the GitHub issue tracker to submit bugs or request features.

CHAPTER 7

Changelog

Please consult the Avocado Releases for fixes and enhancements of each version.

CHAPTER 8

License

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.

CHAPTER 9

Build and Quality Status

Contents:

9.1 Welcome to 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.

9.1.1 How does it work?

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.

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     : 3a5c4c51ceb5369f23702efb10b4209b111141b2
JOB LOG    : $HOME/avocado/job-results/job-2019-10-31T10.34-3a5c4c5/job.log
 (1/1) /bin/true: PASS (0.04 s)
RESULTS    : PASS 1 | ERROR 0 | FAIL 0 | SKIP 0 | WARN 0 | INTERRUPT 0 | CANCEL 0
JOB TIME   : 0.15 s
```

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**.

Tip: See more at the Test types section on the Avocado User's Guide.

9.1.2 Why should I use it?

Multiple result formats

A regular run of Avocado will present the test results on standard output, a nice and colored report useful for human beings. But results for machines can also be generated.

Check the job-results folder (`$HOME/avocado/job-results/latest/`) to see the outputs.

Currently we support, out of box, the following output formats:

- **xUnit**: an XML format that contains test results in a structured form, and are used by other test automation projects, such as jenkins.
- **JSON**: a widely used data exchange format. The JSON Avocado plugin outputs job information, similarly to the xunit output plugin.
- **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).

Note: You can see the results of the latest job inside the folder `$HOME/avocado/job-results/latest/`. You can also specify at the command line the options `--xunit`, `--json` or `--tap` followed by a filename. Avocado will write the output on the specified filename.

When it comes to outputs, Avocado is very flexible. You can check the various **output plugins**. If you need something more sophisticated, visit our plugins section.

Sysinfo data collector

Avocado comes with a sysinfo plugin, which automatically gathers some system information per each job or even between tests. This is very helpful when trying to identify the cause of a test failure.

Check out the files stored at `$HOME/avocado/job-results/latest/sysinfo/`:

```
$ ls $HOME/avocado/job-results/latest/sysinfo/pre/
'brctl show'           hostname             modules
 cmdline              'ifconfig -a'         mounts
 cpuinfo               installed_packages  'numactl --hardware show'
 current_clocksource   interrupts           partitions
'df -mP'              'ip link'             scaling_governor
 dmesg                'ld --version'       'uname -a'
```

(continues on next page)

(continued from previous page)

```
 dmidecode             lscpu                uptime
'fdisk -l'            'lspci -vvnn'         version
'gcc --version'        meminfo
```

For more information about sysinfo collector, please consult the Avocado User's Guide.

Job Replay and Job Diff

In order to reproduce a given job using the same data, one can use the `replay` subcommand, 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.

Example:

```
$ avocado 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
```

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

 ...
```

Extensible by plugins

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

(continues on next page)

(continued from previous page)

```
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
...
```

For more information about plugins, please visit the Plugin System section on the Avocado User's Guide.

Utility libraries

When writing tests, developers often need to perform basic tasks on OS and end up having to implement these routines just to run they tests.

Avocado has **more than 40** *utility modules* that helps you to perform basic operations.

Below a small subset of our utility modules:

- **utils.vmimage**: This utility provides a API to download/cache VM images (QCOW) from the official distributions repositories.
- **utils.memory**: Provides information about memory usage.
- **utils.cpu**: Get information from the current's machine CPU.
- **utils.software_manager**: Software package management library.
- **utils.download**: Methods to download URLs and regular files.
- **utils.archive**: Module to help extract and create compressed archives.

9.1.3 Avocado Python API

If the command-line is limiting you, then you can use our new API and create custom jobs and test suites:

```
import sys

from avocado.core.job import Job

with Job.from_config({'run.references': ['/bin/true']}) as job:
    sys.exit(job.run())
```

9.1.4 How to install

It is super easy, just run the follow command:

```
$ pip3 install --user avocado-framework
```

This will install the avocado command in your home directory.

Note: For more details and alternative methods, please visit the Installing section on Avocado User's Guide

9.1.5 Documentation

Please use the following links for full documentation, including installation methods, tutorials and API or browse this site for more content.

- latest release
- development version

9.1.6 Bugs/Requests

Please use the GitHub issue tracker to submit bugs or request features.

9.1.7 Changelog

Please consult the Avocado Releases for fixes and enhancements of each version.

9.1.8 License

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.

9.1.9 Build and Quality Status

9.2 Avocado User's Guide

9.2.1 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 *Utility Libraries*.
- *Plugins* that can extend and add new functionality to the Avocado Framework.
- A Python API for creating custom jobs and test suites for more advanced users.

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.

9.2.2 Installing

Avocado is primarily written in Python, so a standard Python installation is possible and often preferable. You can also install from your Linux distribution repository, if available.

Note: Please note that this installs the Avocado core functionality. Many Avocado features are distributed as non-core plugins. Visit the Avocado Plugin section on the left menu.

Tip: If you are looking for Virtualization specific testing, also consider looking at Avocado-VT installation instructions after finishing the Avocado installation.

Installing from PyPI

The simplest installation method is through `pip`. On most POSIX systems with Python 3.4 (or later) and `pip` available, installation can be performed with a single command:

```
$ pip3 install --user avocado-framework
```

This will fetch the Avocado package (and possibly some of its dependencies) from the PyPI repository, and will attempt to install it in the user's home directory (usually under `~/.local`), which you might want to add to your `PATH` variable if not done already.

Tip: If you want to perform a system-wide installation, drop the `--user` switch.

If you want even more isolation, Avocado can also be installed in a Python virtual environment. with no additional steps besides creating and activating the "venv" itself:

```
$ python3 -m venv /path/to/new/virtual_environment
$ source /path/to/new/virtual_environment/bin/activate
$ pip3 install avocado-framework
```

Installing from packages

Fedora

Avocado modules are available on standard Fedora repos starting with version 29. To subscribe to the latest version stream, run:

```
$ dnf module enable avocado:latest
```

Or, to use the LTS (Long Term Stability) version stream, run:

```
$ dnf module enable avocado:82lts
```

Then proceed to install a module profile or individual packages. If you're unsure about what to do, simply run:

```
$ dnf module install avocado
```

Enterprise Linux

Avocado modules are also available on EPEL (Extra Packages for Enterprise Linux) repos, starting with version 8. To enable the EPEL repository, run:

```
$ dnf install https://dl.fedoraproject.org/pub/epel/epel-release-latest-8.noarch.rpm
```

Then to enable the module, run:

```
$ dnf module enable avocado:latest
```

And finally, install any number of packages, such as:

```
$ dnf install python3-avocado python3-avocado-plugins-output-html python3-avocado-
↪plugins-varianter-yaml-to-mux
```

Latest Development RPM Packages from COPR

Avocado provides a repository of continuously built packages from the GitHub repository's master branch. These packages are currently available for some of the latest Enterprise Linux and Fedora versions, for a few different architectures.

If you're interested in using the very latest development version of Avocado from RPM packages, you can do so by running:

```
$ dnf copr enable @avocado/avocado-latest
$ dnf install python3-avocado*
```

The following image shows the status of the Avocado packages building on COPR:

OpenSUSE

The OpenSUSE project provides packages for Avocado. Check the Virtualization:Tests project in OpenSUSE build service to get the packages from there.

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.

Installing from source code

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

```
$ sudo dnf install -y python3 git gcc python3-pip
```

Then to install Avocado from the git repository run:

```
$ git clone git://github.com/avocado-framework/avocado.git
$ cd avocado
$ python3 setup.py install --user
```

Optionally, to install the plugins run:

> $ python3 setup.py plugin –install=golang –user $ python3 setup.py plugin –install=html –user $ python3 setup.py plugin –install=result_upload –user $ python3 setup.py plugin –install=resultsdb –user $ python3 setup.py plugin –install=robot –user $ python3 setup.py plugin –install=varianter_cit –user $ python3 setup.py plugin –install=varianter_pict –user $ python3 setup.py plugin –install=varianter_yaml_to_mux –user

9.2.3 Introduction

Avocado Hello World

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.

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     : 3a5c4c51ceb5369f23702efb10b4209b111141b2
JOB LOG    : $HOME/avocado/job-results/job-2019-10-31T10.34-3a5c4c5/job.log
 (1/1) /bin/true: PASS (0.04 s)
RESULTS    : PASS 1 | ERROR 0 | FAIL 0 | SKIP 0 | WARN 0 | INTERRUPT 0 | CANCEL 0
JOB TIME   : 0.15 s
```

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.

Running a job with multiple tests

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

```
$ avocado run examples/tests/sleeptest.py examples/tests/failtest.py examples/tests/
↪synctest.py /tmp/simple_test.sh
JOB ID     : 2391dddf53b950631589bd1d44a5a6fdd023b400
JOB LOG    : $HOME/avocado/job-results/job-2021-09-27T16.35-2391ddd/job.log
 (1/4) examples/tests/sleeptest.py:SleepTest.test: STARTED
 (2/4) examples/tests/failtest.py:FailTest.test: STARTED
 (3/4) examples/tests/synctest.py:SyncTest.test: STARTED
 (4/4) /tmp/simple_test.sh: STARTED
 (4/4) /tmp/simple_test.sh: PASS (0.01 s)
 (2/4) examples/tests/failtest.py:FailTest.test: FAIL: This test is supposed to fail␣
↪(0.05 s)
 (1/4) examples/tests/sleeptest.py:SleepTest.test: PASS (1.02 s)
 (3/4) examples/tests/synctest.py:SyncTest.test: PASS (1.39 s)
RESULTS    : PASS 3 | ERROR 0 | FAIL 1 | SKIP 0 | WARN 0 | INTERRUPT 0 | CANCEL 0
JOB TIME   : 3.25 s
```

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`)

Using a different runner

Currently Avocado has two test runners: `nrunner` (the new runner) and `runner` (legacy). You can find a list of current runners installed with the `avocado plugins` command:

```
$ avocado plugins
Plugins that run test suites on a job (runners):
nrunner nrunner based implementation of job compliant runner
runner  The conventional test runner
```

During the test execution, you can select the runner using the option `--test-runner`, where the default is the nrunner one:

```
$ avocado run --test-runner='runner' /bin/true
```

Interrupting tests

Sending Signals

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.

Interrupting the job on first fail (failfast)

The Avocado `run` command has the option `--failfast` to exit the job on first failed test. The legacy runner runs tests sequentially, with the behavior of `--failfast` as follows:

```
$ avocado run --failfast /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 | CANCEL 0
JOB TIME   : 0.12 s
```

With the nrunner runner, tests are executed in parallel. The `--failfast` option work on the best effort to cancel tests that have not started yet. To replicate the same behavior as the legacy runner, use `--nrunner-max-parallel-tasks=1` to limit the number of tasks executed in parallel:

```
$ avocado run --failfast --nrunner-max-parallel-tasks=1 /bin/true /bin/false /bin/
↪true /bin/true
JOB ID     : 76bfe0e5cfa5efac3ab6881ee501cc5d4b69f913
JOB LOG    : $HOME/avocado/job-results/job-2021-09-27T16.41-76bfe0e/job.log
 (1/4) /bin/true: STARTED
 (1/4) /bin/true: PASS (0.01 s)
 (2/4) /bin/false: STARTED
 (2/4) /bin/false: FAIL (0.01 s)
Interrupting job (failfast).
RESULTS    : PASS 1 | ERROR 0 | FAIL 1 | SKIP 2 | WARN 0 | INTERRUPT 0 | CANCEL 0
JOB TIME   : 1.57 s
```

The default behavior, that is, when `--failfast` is **not** set, is to try to execute all tests in a job, regardless individual of test failures.

Note: Avocado versions 80.0 and earlier allowed replayed jobs to override the failfast configuration by setting `--failfast=off` in a `avocado replay ..` command line. This is no longer possible.

The hint files

Avocado team has added support to the "hint files". This feature is present since Avocado #78 and is a configuration file that you can add to your project root folder to help Avocado on the "test resolution" phase.

The idea is that, you know more about your tests than anybody else. And you can specify where your tests are, and what type (kind) they are. You just have to add a `.avocado.hint` in your root folder with the section `[kinds]` and one section for each kind that you are using.

On the specific test type section, you can specify 3 options: `uri`, `args` and `kwargs`.

Note: Some test types will convert `kwargs` into variable environments. Please check the documentation of the test type that you are using.

You can also use the keyword `$testpath` in any of the options inside the test type section. Avocado will replace `$testpath` with your test path, after the expansion.

For instance, below you will find a hint file example where we have only one test type `TAP`:

```
[kinds]
tap = ./tests/unit/*.sh

[tap]
uri = $testpath
args = --tap
kwargs = DEBUG=1
```

Let's suppose that you have 2 tests that matches `./tests/unit/*.sh`:

- `./tests/unit/foo.sh`
- `./tests/unit/bar.sh`

Avocado will run each one as a `TAP` test, as you desired.

Note: Please, keep in mind that hint files needs absolute paths when defining tests inside the `[kinds]` section.

Note: Also, note that hint files are only supported when using the next runner (`--test-runner='nrunner'`).

Since Avocado's next runner is capable of running tests not only in a subprocess but also in more isolated environments such as Podman containers, sending custom environment variables to the task executor can be achieved by using the `kwargs` parameter. Use a comma-separated list of variables here and Avocado will make sure your tests will receive those variables (regardless of the spawner type).

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 examples/tests/passtest.py badtest.py
Unable to resolve reference(s) 'badtest.py' with plugins(s) 'file', 'robot', 'external
↪', try running 'avocado -V list 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`, a boolean command-line option. The same message will appear in the UI, but the Job will be executed:

```
$ avocado run examples/tests/passtest.py badtest.py --ignore-missing-references
JOB ID     : e6d1f4d21d6a5e2e039f1acd1670a6882144c189
JOB LOG    : $HOME/avocado/job-results/job-2021-09-27T16.50-e6d1f4d/job.log
 (1/1) examples/tests/passtest.py:PassTest.test: STARTED
 (1/1) examples/tests/passtest.py:PassTest.test: PASS (0.01 s)
RESULTS    : PASS 1 | ERROR 0 | FAIL 0 | SKIP 0 | WARN 0 | INTERRUPT 0 | CANCEL 0
JOB TIME   : 1.49 s
```

Running tests with an external runner

Note: This feature is supported with the legacy runner only.

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 --test-runner=runner --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 and 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 --test-runner=runner --external-runner=/bin/sh /tmp/pass /tmp/fail
JOB ID     : 4a2a1d259690cc7b226e33facdde4f628ab30741
JOB LOG    : $HOME/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 | CANCEL 0
JOB TIME   : 0.11 s
```

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 --test-runner=runner --external-runner=/bin/curl https://google.com/
JOB ID     : 56016a1ffffaba02492fdbd5662ac0b958f51e11
JOB LOG    : $HOME/avocado/job-results/job-<date>-<shortid>/job.log
(1/1) https://google.com/: PASS (0.02 s)
RESULTS    : PASS 1 | ERROR 0 | FAIL 0 | SKIP 0 | WARN 0 | INTERRUPT 0 | CANCEL 0
JOB TIME   : 3.14 s
```

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.

Warning: The external runner is incompatible with loaders from *Understanding the test discovery (Avocado Loaders)*. If you use external runner and loader together the job will use the external runner and ignore the loader.

Runner outputs

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.

Note: The HTML report needs the `html` plugin enabled that is an optional plugin.

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 examples/tests/sleeptest.py examples/tests/failtest.py examples/tests/
↪synctest.py
JOB ID     : 2e83086e5d3f82dd68bdc8885e7cce1cebec5f27
JOB LOG    : $HOME/wrampazz/avocado/job-results/job-2021-09-27T17.00-2e83086/job.log
 (3/3) examples/tests/synctest.py:SyncTest.test: STARTED
 (1/3) examples/tests/sleeptest.py:SleepTest.test: STARTED
 (2/3) examples/tests/failtest.py:FailTest.test: STARTED
 (2/3) examples/tests/failtest.py:FailTest.test: FAIL: This test is supposed to fail␣
↪(0.02 s)
 (1/3) examples/tests/sleeptest.py:SleepTest.test: PASS (1.01 s)
 (3/3) examples/tests/synctest.py:SyncTest.test: PASS (1.24 s)
RESULTS    : PASS 2 | ERROR 0 | FAIL 1 | SKIP 0 | WARN 0 | INTERRUPT 0 | CANCEL 0
JOB HTML   : $HOME/avocado/job-results/job-2021-09-27T17.00-2e83086/results.html
JOB TIME   : 2.80 s
```

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.

As you can see, Avocado will print a nice UI with the job summary on the console. If you would like to inspect a detailed output of your tests, you can visit the folder: `$HOME/avocado/job-results/latest/` or a specific job folder.

Results for machine

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.

Currently, you can find three different formats available on this folder: **xUnit (XML)**, **JSON** and **TAP**.

1. 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 examples/tests/sleeptest.py examples/tests/failtest.py examples/tests/
↪synctest.py --xunit -
<?xml version="1.0" encoding="UTF-8"?>
<testsuite name="job-2021-09-27T17.01-2dd7837" tests="3" errors="0" failures="1"␣
↪skipped="0" time="2.340" timestamp="2021-09-27T17:01:36.455763">
    <testcase classname="&lt;unknown&gt;" name="2-examples/tests/failtest.py:FailTest.
↪test" time="0.026">
            <failure type="&lt;unknown&gt;" message="This test is supposed to fail"><!
↪[CDATA[<unknown>]]></failure>
            <system-out><![CDATA[[stdlog] 2021-09-27 17:01:34,722 test             ␣
↪L0312 INFO | INIT 1-FailTest.test
[stdlog] 2021-09-27 17:01:34,723 parameters       L0142 DEBUG| PARAMS (key=timeout,␣
↪path=*, default=None) => None
[stdlog] 2021-09-27 17:01:34,723 test             L0340 DEBUG| Test metadata:
[stdlog] 2021-09-27 17:01:34,723 test             L0342 DEBUG|   filename: $HOME/src/
↪avocado/avocado.dev/examples/tests/failtest.py
[stdlog] 2021-09-27 17:01:34,723 test             L0348 DEBUG|   teststmpdir: /var/
↪tmp/avocado_vi1xpequ
[stdlog] 2021-09-27 17:01:34,723 test             L0538 INFO | START 1-FailTest.test
[stdlog] 2021-09-27 17:01:34,724 test             L0207 DEBUG| DATA (filename=output.
↪expected) => NOT FOUND (data sources: variant, test, file)
[stdlog] 2021-09-27 17:01:34,724 stacktrace       L0039 ERROR|
[stdlog] 2021-09-27 17:01:34,724 stacktrace       L0041 ERROR| Reproduced traceback␣
↪from: $HOME/src/avocado/avocado.dev/avocado/core/test.py:794
[stdlog] 2021-09-27 17:01:34,725 stacktrace       L0045 ERROR| Traceback (most recent␣
↪call last):
[stdlog] 2021-09-27 17:01:34,725 stacktrace       L0045 ERROR|   File "$HOME/src/
↪avocado/avocado.dev/examples/tests/failtest.py", line 16, in test
[stdlog] 2021-09-27 17:01:34,725 stacktrace       L0045 ERROR|     self.fail('This␣
↪test is supposed to fail')
[stdlog] 2021-09-27 17:01:34,725 stacktrace       L0045 ERROR|   File "$HOME/src/
↪avocado/avocado.dev/avocado/core/test.py", line 980, in fail
[stdlog] 2021-09-27 17:01:34,725 stacktrace       L0045 ERROR|     raise exceptions.
↪TestFail(message)
[stdlog] 2021-09-27 17:01:34,725 stacktrace       L0045 ERROR| avocado.core.
↪exceptions.TestFail: This test is supposed to fail
[stdlog] 2021-09-27 17:01:34,725 stacktrace       L0046 ERROR|
[stdlog] 2021-09-27 17:01:34,725 test             L0799 DEBUG| Local variables:
[stdlog] 2021-09-27 17:01:34,740 test             L0802 DEBUG|  -> self <class
↪'failtest.FailTest'>: 1-FailTest.test
[stdlog] 2021-09-27 17:01:34,741 test             L0207 DEBUG| DATA (filename=output.
↪expected) => NOT FOUND (data sources: variant, test, file)
[stdlog] 2021-09-27 17:01:34,741 test             L0207 DEBUG| DATA (filename=stdout.
↪expected) => NOT FOUND (data sources: variant, test, file)
[stdlog] 2021-09-27 17:01:34,741 test             L0207 DEBUG| DATA (filename=stderr.
↪expected) => NOT FOUND (data sources: variant, test, file)
[stdlog] 2021-09-27 17:01:34,741 test             L0957 ERROR| FAIL 1-FailTest.test ->
↪ TestFail: This test is supposed to fail
[stdlog] 2021-09-27 17:01:34,741 test             L0949 INFO |
]]></system-out>
    </testcase>
    <testcase classname="&lt;unknown&gt;" name="1-examples/tests/sleeptest.
↪py:SleepTest.test" time="1.010"/>
    <testcase classname="&lt;unknown&gt;" name="3-examples/tests/synctest.py:SyncTest.
↪test" time="1.304"/>
</testsuite>
```

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

Note: In case your tests produce very long outputs, you can limit the number of embedded characters by `--xunit-max-test-log-chars`. If the output in the log file is longer it only attaches up-to max-test-log-chars characters one half starting from the beginning of the content, the other half from the end of the content.

2. JSON:

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

```
$ avocado run examples/tests/sleeptest.py examples/tests/failtest.py examples/tests/
↪synctest.py --json -
{
    "cancel": 0,
    "debuglog": "$HOME/avocado/job-results/job-2021-09-27T17.05-fd073c2/job.log",
    "errors": 0,
    "failures": 1,
    "interrupt": 0,
    "job_id": "fd073c26a1e1aacee59bc9e1914b7110e7ac3f8b",
    "pass": 2,
    "skip": 0,
    "tests": [
        {
            "end": 30759.486869323,
            "fail_reason": "This test is supposed to fail",
            "id": "2-examples/tests/failtest.py:FailTest.test",
            "logdir": "$HOME/avocado/job-results/job-2021-09-27T17.05-fd073c2/test-
↪results/2-examples_tests_failtest.py_FailTest.test",
            "logfile": "$HOME/avocado/job-results/job-2021-09-27T17.05-fd073c2/test-
↪results/2-examples_tests_failtest.py_FailTest.test/debug.log",
            "start": 30759.456017671,
            "status": "FAIL",
            "tags": {},
            "time": 0.030851651998091256,
            "whiteboard": ""
        },
        {
            "end": 30760.472274292,
            "fail_reason": "<unknown>",
            "id": "1-examples/tests/sleeptest.py:SleepTest.test",
            "logdir": "$HOME/avocado/job-results/job-2021-09-27T17.05-fd073c2/test-
↪results/1-examples_tests_sleeptest.py_SleepTest.test",
            "logfile": "$HOME/avocado/job-results/job-2021-09-27T17.05-fd073c2/test-
↪results/1-examples_tests_sleeptest.py_SleepTest.test/debug.log",
            "start": 30759.455787493,
            "status": "PASS",
            "tags": {},
            "time": 1.0164867989988124,
            "whiteboard": ""
        },
        {
            "end": 30760.690585313,
            "fail_reason": "<unknown>",
```

(continues on next page)

(continued from previous page)

```
            "id": "3-examples/tests/synctest.py:SyncTest.test",
            "logdir": "$HOME/avocado/job-results/job-2021-09-27T17.05-fd073c2/test-
↪results/3-examples_tests_synctest.py_SyncTest.test",
            "logfile": "$HOME/avocado/job-results/job-2021-09-27T17.05-fd073c2/test-
↪results/3-examples_tests_synctest.py_SyncTest.test/debug.log",
            "start": 30759.459244923,
            "status": "PASS",
            "tags": {},
            "time": 1.231340390000696,
            "whiteboard": ""
        }
    ],
    "time": 2.2786788409975998,
    "total": 3,
    "warn": 0
}
```

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.

3. 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 examples/tests/sleeptest.py --tap -
1..1
ok 1 examples/tests/sleeptest.py:SleepTest.test
```

Using the option –show

Probably, 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 ...`:

```
$ 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.

It's possible to silence all output to stdout (while keeping the error messages being printed to stderr). One can then use the return code to learn about the result:

```
$ avocado --show=none run examples/tests/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 --show=none 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 examples/tests/sleeptest.py examples/tests/synctest.py --xunit - --json␣
↪/tmp/result.json
<?xml version="1.0" encoding="UTF-8"?>
<testsuite name="job-2021-09-27T17.10-b37e5fe" tests="2" errors="0" failures="0"␣
↪skipped="0" time="2.220" timestamp="2021-09-27T17:10:28.757207">
    <testcase classname="&lt;unknown&gt;" name="1-examples/tests/sleeptest.
↪py:SleepTest.test" time="1.011"/>
    <testcase classname="&lt;unknown&gt;" name="2-examples/tests/synctest.py:SyncTest.
↪test" time="1.209"/>
</testsuite>

$ cat /tmp/result.json
{
    "cancel": 0,
    "debuglog": "$HOME/avocado/job-results/job-2021-09-27T17.10-b37e5fe/job.log",
    "errors": 0,
    "failures": 0,
    "interrupt": 0,
    "job_id": "b37e5fee226e3806c4d73fef180d7d2cee56464e",
    "pass": 2,
    "skip": 0,
}
```

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

```
avocado run examples/tests/sleeptest.py examples/tests/synctest.py --xunit - --json  -
avocado run: error: argument --json: Options --xunit --json 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.

Note: Avocado support "paginator" option, which, on compatible terminals, basically pipes the colored output to `less` to simplify browsing of the produced output. You an enable it with `--enable-paginator`.

Running simple tests with arguments

Note: This feature is supported with the legacy runner only.

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 existing-file␣
↪nonexisting-file
```

This will run 3 tests, the first one is a normal test defined by `existing_file.py` (most probably an instrumented test) and will be executed by the "file" loader. Then we have two script files which are going to be executed with `/bin/sh`.

Sysinfo collection

Note: This feature is not fully supported on nrunner runner yet.

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. You can also use the environment variable `AVOCADO_SYSINFODIR` which points to the sysinfo directory in results.
2. fail_commands - similar to commands, but gets executed only when the test fails.
3. files - file with new-line separated list of files to be copied.
4. fail_files - similar to files, but copied only when the test fails.
5. 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.

By default these are collected per-job but you can also run them per-test by setting `per_test = True` in the `sysinfo.collect` section.

The sysinfo is enabled by default and can also be disabled on the cmdline if needed by `--disable-sysinfo`.

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.

It is also possible to save only the files / commands which were changed during the course of the test, in the `post` directory, using the setting `optimize = True` in the `sysinfo.collect` section. This collects all sysinfo on `pre`, but saves only changed ones on `post`. It is set to False by default.

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

9.2.4 Basic Concepts

Attention: TODO: This section needs attention! Please, help us contributing to this document.

It is important to understand some basic concepts before start using Avocado.

Test Resolution

Note: Some definitions here may be out of date. The current runner can still be using some of these definitions in its design, however, we are working on an improved version of the runner, the NextRunner that will use an alternative strategy.

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 *WritingTests* documentation on how to use the tags functionality to mark derived classes as Avocado test classes.

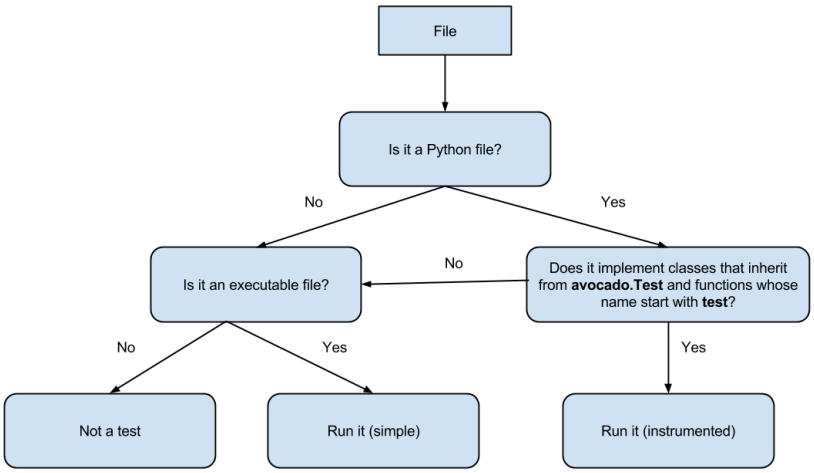
Identifiers and references

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 examples/tests/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

Warning: TODO: We are talking here about Test Resolver, but the reader was not introduced to this concept yet.

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.

When the test references are about Instrumented Tests, Avocado will find any Instrumented test that **starts** with the reference, like a "wildcard". For instance:

```
$ avocado run ./test.py:MyTest:test_foo
```

This command will resolve all tests (methods) that starts with *test_foo*. For more information about this type of tests, please visit the *Instrumented* section of this document.

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

Conventions

Even though each resolver implementation is free to interpret a reference string as it sees fit, it's a good idea to set common user expectations.

It's common for a single file to contain multiple tests. In that case, information about the specific test to reference can be added after the filesystem location and a colon, that is, for the reference:

```
passtest.py:PassTest.test
```

Unless a file with that exact name exists, most resolvers will split it into `passtest.py` as the filesystem path, and `PassTest.test` as an additional specification for the individual test. It's also possible that some resolvers will support regular expressions and globs for the additional information component.

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.

Examples of Test Names:

```
'/bin/true'
'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'
```

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>]
```

Example of Test IDs:

```
'1-/bin/true'
'2-passtest.py:Passtest.test;quiet-'
'3-file:///tmp/passtest.py:Passtest.test'
'4-multiple_tests.py:MultipleTests.test_hello;maximum_debug-df2f'
'5-type_specific.io-github-autotest-qemu.systemtap_tracing.qemu.qemu_free'
```

Test types

Avocado at its simplest configuration can run three different types of tests:

- simple
- python unittest
- instrumented

You can mix and match those in a single job.

Avocado plugins can also introduce additional test types.

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.

Python unittest

The discovery of classical Python unittest is also supported, although unlike Python unittest we still use static analysis to get individual tests so dynamically created cases are not recognized. Apart from that there should be no surprises when running unittests via Avocado.

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` and `SKIP` are considered successful. The `ERROR`, `FAIL` and `INTERRUPTED` signal failures.

TAP

TAP tests are pretty much like Simple tests in the sense that they are programs (either binaries or scripts) that will executed. The difference is that the test result will be decided based on the produced output, that should be in Test Anything Protocol format.

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.
- `WARN`: The test ran and something might have gone wrong but didn't explicitly failed.
- `CANCEL`: The test was canceled and didn't run.
- `INTERRUPTED`: The test was explicitly interrupted. Usually this means that a user hit CTRL+C while the job was still running or did not finish before the timeout specified.

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*.

9.2.5 Basic Operations

Job Replay

The process of replaying an Avocado Job is simply about loading the source Job's configuration and running a new Job based on that configuration.

For users, this is available as the `avocado replay` command. Its usage is straightforward. Suppose you've just run a simple job, also from the command line, such as:

```
$ avocado run /bin/true /bin/false
JOB ID     : 42c60bea72e6d55756bfc784eb2b354f788541cf
JOB LOG    : $HOME/avocado/job-results/job-2020-08-13T11.23-42c60be/job.log
 (1/2) /bin/true: PASS (0.01 s)
 (2/2) /bin/false: FAIL: Exited with status: '1', stdout: '' stderr: '' (0.08 s)
RESULTS    : PASS 1 | ERROR 0 | FAIL 1 | SKIP 0 | WARN 0 | INTERRUPT 0 | CANCEL 0
JOB HTML   : $HOME/avocado/job-results/job-2020-08-13T11.23-42c60be/results.html
JOB TIME   : 0.41 s
```

To run a new job with the configuration used by the previously executed job, it's possible to simply execute:

```
$ avocado replay latest
```

Resulting in:

```
JOB ID     : f3139826f1b169a0b456e0e880ffb83ed26d9858
SRC JOB ID : latest
JOB LOG    : $HOME/avocado/job-results/job-2020-08-13T11.24-f313982/job.log
 (1/2) /bin/true: PASS (0.01 s)
 (2/2) /bin/false: FAIL: Exited with status: '1', stdout: '' stderr: '' (0.07 s)
RESULTS    : PASS 1 | ERROR 0 | FAIL 1 | SKIP 0 | WARN 0 | INTERRUPT 0 | CANCEL 0
JOB HTML   : $HOME/avocado/job-results/job-2020-08-13T11.24-f313982/results.html
JOB TIME   : 0.39 s
```

It's also possible to use the other types of references to jobs, like the full directory path of the job results, or the Job IDs. That is, you can use the same references used in other commands such as `avocado jobs show`.

Legacy Job Replay

Note: This legacy version is expected to be removed in future versions.

Avocado's first, and now legacy, job replay version is based on the `run` command. It supports more command line options and use cases than the newer implementation discussed earlier, but it has some cons:

- It's not clear if options given to `avocado run --replay` are about the replayed job or if overriding aspects of the source job
- The implementation has to account for each of the options capable of being overridden

It's expected that more complex use cases for Jobs, including replays, should instead use the Job API directly. Regardless, the remainder of this section documents its behavior.

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 a 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
 (1/4) /bin/true;first-c49a: PASS (0.01 s)
 (2/4) /bin/true;second-f05f: PASS (0.01 s)
 (3/4) /bin/false;first-c49a: FAIL (0.04 s)
 (4/4) /bin/false;second-f05f: FAIL (0.04 s)
RESULTS    : PASS 2 | ERROR 0 | FAIL 2 | 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
 (1/4) /bin/true;first-c49a: SKIP
 (2/4) /bin/true;second-f05f: SKIP
 (3/4) /bin/false;first-c49a: FAIL (0.03 s)
 (4/4) /bin/false;second-f05f: FAIL (0.04 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.

Note: Avocado versions 80.0 and earlier allowed replayed jobs to override the failfast configuration by setting `--failfast` in a `avocado run --replay ..` command line. This is no longer possible.

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
```

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
```

(continues on next page)

(continued from previous page)

```
@@ -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.

Note: Avocado Diff will ignore files containing non UTF-8 characters, like binaries, as an example.

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`, a temporary html file will be created.

For those wiling to use a custom diff tool instead of the Avocado Diff tool, there is an option `--create-reports` that will, 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

...
```

Listing tests

Avocado can list your tests without run it. This can be handy sometimes.

There are two ways of discovering the tests. One way is to simulate the execution by using the `--dry-run` argument:

```
$ avocado run /bin/true --dry-run
JOB ID     : 0000000000000000000000000000000000000000
JOB LOG    : /var/tmp/avocado-dry-run-k2i_uiqx/job-2020-09-02T09.09-0000000/job.log
 (1/1) /bin/true: CANCEL: Test cancelled due to --dry-run (0.01 s)
RESULTS    : PASS 0 | ERROR 0 | FAIL 0 | SKIP 0 | WARN 0 | INTERRUPT 0 | CANCEL 1
JOB HTML   : /var/tmp/avocado-dry-run-k2i_uiqx/job-2020-09-02T09.09-0000000/results.
↪html
JOB TIME   : 0.29 s
```

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 --loader
INSTRUMENTED /usr/share/doc/avocado/tests/abort.py
INSTRUMENTED /usr/share/doc/avocado/tests/datadir.py
INSTRUMENTED /usr/share/doc/avocado/tests/doublefail.py
INSTRUMENTED /usr/share/doc/avocado/tests/doublefree.py
INSTRUMENTED /usr/share/doc/avocado/tests/errortest.py
INSTRUMENTED /usr/share/doc/avocado/tests/failtest.py
```

(continues on next page)

(continued from previous page)

```
INSTRUMENTED /usr/share/doc/avocado/tests/fiotest.py
INSTRUMENTED /usr/share/doc/avocado/tests/gdbtest.py
INSTRUMENTED /usr/share/doc/avocado/tests/gendata.py
INSTRUMENTED /usr/share/doc/avocado/tests/linuxbuild.py
INSTRUMENTED /usr/share/doc/avocado/tests/multiplextest.py
INSTRUMENTED /usr/share/doc/avocado/tests/passtest.py
INSTRUMENTED /usr/share/doc/avocado/tests/sleeptenmin.py
INSTRUMENTED /usr/share/doc/avocado/tests/sleeptest.py
INSTRUMENTED /usr/share/doc/avocado/tests/synctest.py
INSTRUMENTED /usr/share/doc/avocado/tests/timeouttest.py
INSTRUMENTED /usr/share/doc/avocado/tests/warntest.py
INSTRUMENTED /usr/share/doc/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 --loader | grep ^SIMPLE
SIMPLE       /usr/share/doc/avocado/tests/env_variables.sh
SIMPLE       /usr/share/doc/avocado/tests/output_check.sh
SIMPLE       /usr/share/doc/avocado/tests/simplewarning.sh
SIMPLE       /usr/share/doc/avocado/tests/failtest.sh
SIMPLE       /usr/share/doc/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 --verbose list examples/gdb-prerun-scripts/
Type       Test                                          Tag(s)
NOT_A_TEST examples/gdb-prerun-scripts/README: Not an INSTRUMENTED (avocado.Test␣
↪based), PyUNITTEST (unittest.TestCase based) or SIMPLE (executable) test
NOT_A_TEST examples/gdb-prerun-scripts/pass-sigusr1: Not an INSTRUMENTED (avocado.
↪Test based), PyUNITTEST (unittest.TestCase based) or SIMPLE (executable) test
!GLIB      examples/gdb-prerun-scripts/: No GLib-like tests found
!GOLANG    examples/gdb-prerun-scripts/: No test matching this reference.
!ROBOT     examples/gdb-prerun-scripts/: No robot-like tests found
NOT_A_TEST examples/gdb-prerun-scripts/README: Not a supported test
NOT_A_TEST examples/gdb-prerun-scripts/pass-sigusr1: Not a supported test

TEST TYPES SUMMARY
==================
!glib: 1
!golang: 1
!robot: 1
not_a_test: 4
```

Notice that the verbose flag also adds summary information.

See also:

To read more about test discovery, visit the section "Understanding the test discovery (Avocado Loaders)".

9.2.6 Results Specification

On a machine that executed Avocado, 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-2021-09-28T14.21-e0775d9`. A typical results directory structure can be seen below

```
$HOME/avocado/job-results/job-2021-09-28T14.21-e0775d9/
├── avocado.core.DEBUG
├── id
├── jobdata
│   ├── args.json
│   ├── cmdline
│   ├── config
│   ├── pwd
│   ├── test_references
│   └── variants.json
├── job.log
├── results.html
├── results.json
├── results.tap
├── results.xml
├── sysinfo
│   ├── post
│   │   ├── brctl show
│   │   ├── cmdline
│   │   ├── cpuinfo
│   │   ├── current_clocksource
│   │   ├── df -mP
│   │   ├── dmesg
│   │   ├── dmidecode
│   │   ├── fdisk -l
│   │   ├── gcc --version
│   │   ├── hostname
│   │   ├── ifconfig -a
│   │   ├── interrupts
│   │   ├── ip link
│   │   ├── journalctl.gz
│   │   ├── ld --version
│   │   ├── lscpu
│   │   ├── lspci -vvnn
│   │   ├── meminfo
│   │   ├── modules
│   │   ├── mounts
│   │   ├── numactl --hardware show
│   │   ├── partitions
│   │   ├── pci
│   │   ├── scaling_governor
│   │   ├── sched_features
│   │   ├── slabinfo
│   │   ├── uname -a
│   │   ├── uptime
│   │   └── version
│   └── pre
│       ├── brctl show
│       ├── cmdline
│       ├── cpuinfo
│       ├── current_clocksource
│       ├── df -mP
│       ├── dmesg
│       ├── dmidecode
│       ├── fdisk -l
```

(continues on next page)

(continued from previous page)

```
│   │   ├── gcc --version
│   │   ├── hostname
│   │   ├── ifconfig -a
│   │   ├── interrupts
│   │   ├── ip link
│   │   ├── ld --version
│   │   ├── lscpu
│   │   ├── lspci -vvnn
│   │   ├── meminfo
│   │   ├── modules
│   │   ├── mounts
│   │   ├── numactl --hardware show
│   │   ├── partitions
│   │   ├── pci
│   │   ├── scaling_governor
│   │   ├── sched_features
│   │   ├── slabinfo
│   │   ├── uname -a
│   │   ├── uptime
│   │   └── version
│   └── profile
└── test-results
    ├── 1-examples_tests_sleeptest.py_SleepTest.test
    │   ├── debug.log
    │   └── whiteboard
    ├── 2-examples_tests_sleeptest.py_SleepTest.test
    │   ├── debug.log
    │   └── whiteboard
    └── 3-examples_tests_sleeptest.py_SleepTest.test
        ├── debug.log
        └── whiteboard
```

From what you can see, the results directory 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` subdirectory, with sub directories `pre`, `post` and `profile` that store sysinfo files pre test, post test and profiling info while the test was running, respectively.
3) A `data` subdirectory, where the test can output a number of files if necessary.

Test execution environment using the legacy runner

Each test is executed in a separate process. Due to how the underlying operating system works, a lot of the attributes of the parent process (the Avocado test **runner**) are passed down to the test process.

On GNU/Linux systems, a child process should be *"an exact duplicate of the parent process, except"* some items that are documented in the `fork(2)` man page.

Note: The next Runner (`--test-runner='nrunner'`) has support to different spawners types (podman, process, etc..). For more information, visit the `nrunner.spawner` configuration option.

Besides those operating system exceptions, the Avocado test runner changes the test process in the following ways:

1) The standard input (`STDIN`) is set to a `null device`. This is truth both for `sys.stdin` and for file descriptor `0`. Both will point to the same open null device file.

2) The standard output (`STDOUT`), as in `sys.stdout`, is redirected so that it doesn't interfere with the test runner's own output. All content written to the test's `sys.stdout` will be available in the logs under the `output` prefix.

   Warning: The file descriptor `1` (AKA `/dev/stdout`, AKA `/proc/self/fd/1`, etc) is **not** currently redirected for INSTRUMENTED tests. Any attempt to write directly to the file descriptor will interfere with the runner's own output stream. This behavior will be addressed in a future version.

3) The standard error (`STDERR`), as in `sys.stderr`, is redirected so that it doesn't interfere with the test runner's own errors. All content written to the test's `sys.stderr` will be available in the logs under the `output` prefix.

   Warning: The file descriptor `2` (AKA `/dev/stderr`, AKA `/proc/self/fd/2`, etc) is **not** currently redirected for INSTRUMENTED tests. Any attempt to write directly to the file descriptor will interfere with the runner's own error stream. This behavior will be addressed in a future version.

4) A custom handler for signal `SIGTERM` which will simply raise an exception (with the appropriate message) to be handled by the Avocado test runner, stating the fact that the test was interrupted by such a signal.

   Tip: By following the backtrace that is given alongside the in the test log (look for `RuntimeError: Test interrupted by SIGTERM`) a user can quickly grasp at which point the test was interrupted.

   Note: If the test handles `SIGTERM` differently and doesn't finish the test process quickly enough, it will receive then a `SIGKILL` which is supposed to definitely end the test process.

5) A number of *environment variables* that are set by Avocado, all prefixed with `AVOCADO_`.

If you want to see for yourself what is described here, you may want to run the example test `test_env.py` and examine its log messages.

9.2.7 Filtering tests by tags

Warning: The example perf.py is not distributed with avocado anymore. This is an old example that needs to be updated.

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).

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

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

Multiple tags (AND vs OR)

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 --loader 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 --loader 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 --loader 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
```

Using further categorization with keys and values

All the examples given so far are limited to "flat" tags. Sometimes, it's helpful to categorize tests with extra context. For instance, if you have tests that are sensitive to the platform endianness, you may way to categorize them by endianness, while at the same time, specifying the exact type of endianness that is required.

For instance, your tags can now have a key and value pair, like: `endianess:little` or `endianess:big`.

To list tests without any type of filtering would give you:

```
$ avocado list --loader byteorder.py
INSTRUMENTED byteorder.py:ByteOrder.test_le
INSTRUMENTED byteorder.py:ByteOrder.test_be
INSTRUMENTED byteorder.py:Generic.test
```

To list tests that are somehow related to endianness, you can use:

```
$ avocado list --loader byteorder.py --filter-by-tags endianness
INSTRUMENTED byteorder.py:ByteOrder.test_le
INSTRUMENTED byteorder.py:ByteOrder.test_be
```

And to be even more specific, you can use:

```
$ avocado list --loader byteorder.py --filter-by-tags endianness:big
INSTRUMENTED byteorder.py:ByteOrder.test_be
```

Now, suppose you intend to run tests on a little endian platform, but you'd still want to include tests that are generic enough to run on either little or big endian (but not tests that are specific to other types of endianness), you could use:

```
$ avocado list --loader byteorder.py --filter-by-tags endianness:big --filter-by-tags-
↪include-empty-key
INSTRUMENTED byteorder.py:ByteOrder.test_be
INSTRUMENTED byteorder.py:Generic.test
```

See also:

If you would like to understand how write plugins and how describe tags inside a plugin, please visit the section: *Writing Tests* on Avocado Test Writer's Guide.

9.2.8 Configuring

Warning: Please, keep in mind that we are doing a significant refactoring on settings to have consistency when using Avocado. Some options are changing soon.

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/doc/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` (when installed by your Linux distribution's package manager).

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.

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.

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.

Configuring via command-line

Besides the configuration files, the most used features can also be configured by command-line arguments. For instance, regardless what you have on your configuration files, you can disable sysinfo logging by running:

```
$ avocado run --disable-sysinfo /bin/true
```

So, command-line options always will have the highest precedence during the configuration parsing. Use this if you would like to change some behavior on just one or a few specific executions.

Parsing order recap

So the file parsing order is:

- `/etc/avocado/avocado.conf`
- `/etc/avocado/conf.d/*.conf`
- `avocado.plugins.settings` plugins (but they can insert to any location)
  - For more information about this, visit the "Contributor's Guide" section named "Writing an Avocado plugin"
- `~/.config/avocado/avocado.conf`

You can see the actual set of files/location by using `avocado config` which uses `*` to mark existing and used files:

```
$ avocado config
Config files read (in order, '*' means the file exists and had been read):
 * /etc/avocado/avocado.conf
 * /etc/avocado/conf.d/resultsdb.conf
 * /etc/avocado/conf.d/result_upload.conf
 * /etc/avocado/conf.d/jobscripts.conf
 * /etc/avocado/conf.d/gdb.conf
 * /etc/avocado_vt/conf.d/vt.conf
 * /etc/avocado_vt/conf.d/vt_joblock.conf
   $HOME/.config/avocado/avocado.conf

 Section.Key                              Value
 datadir.paths.base_dir                   /var/lib/avocado
 datadir.paths.test_dir                   /usr/share/doc/avocado/tests
 ...
```

Where the lower config files override values of the upper files and the `$HOME/.config/avocado/avocado.conf` file missing.

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.

Supported data types when configuring Avocado

As already said before, Avocado allows users to use both: configuration files and command-line options to configure its behavior. It is important to have a very well defined system type for the configuration file and argument options.

Although config files options and command-line arguments are always considered `strings`, you should give a proper format representation so those values can be parsed into a proper type internally on Avocado.

Currently Avocado supports the following data types for the configuration options: `string`, `integer`, `float`, `bool` and `list`. Besides those primitive data types Avocado also supports custom data types that can be used by a particular plugin.

Below, you will find information on how to set options based on those basic data types using both: configuration files and command-line arguments.

Strings

Strings are the basic ones and the syntax is the same in both configuration files and command-line arguments: Just the string that can be inside `""` or `''`.

Example using the configuration file:

```
[foo]
bar = 'hello world'
```

String and all following types could be used with or without quotes but using quotes for strings is important on the command line to safely handle empty spaces and distinguish it from a list type. Therefore, the following example will also be well handled:

```
[foo]
bar = hello world
```

Example using the command-line:

```
$ avocado run --foo bar /bin/true
```

Integers

Integer numbers are as simple as strings.

Example using the configuration file:

```
[run]
job_timeout = 60
```

Example using the command-line:

```
$ avocado run --job-timeout 50 /bin/true
```

Floats

Float numbers has the same representation as integers, but you should use . (dot) to separate the decimals. i.e: *80.3*.

Booleans

When talking about configuration files, accepted values for a boolean option are '1', 'yes', 'true', and 'on', which cause this method to return True, and '0', 'no', 'false', and 'off', which cause it to return False. But, when talking about command-line, booleans options don't need any argument, the option itself will enable or disable the settings, depending on the context.

Example using the configuration file:

```
[core]
verbose = true
```

Example using the command-line:

```
$ avocado run --verbose /bin/true
```

Note: Currently we still have some "old style boolean" options where you should pass "on" or "off" on the command-line. i.e: `--json-job-result=off`. Those options are going to be replaced soon.

Lists

Lists are peculiar when configuring. On configuration files you can use the default "python" syntax for lists: `["foo", "bar"]`, but when using the command-line arguments lists are strings separated by spaces:

Example using the configuration file:

```
[assets.fetch]
references = ["foo.py", "bar.py"]
```

Example using the command-line:

```
$ avocado assets fetch foo.py bar.py
```

Complete Configuration Reference

For a complete configuration reference, please visit *Avocado's Configuration Reference*.

Or you can see in your terminal, typing:

```
$ avocado config reference
```

9.2.9 Managing Requirements

Note: Test requirements are supported only on the nrunner runner.

A test's requirement can be fulfilled by the Requirements Resolver feature.

Test's requirements are specified in the test definition and are fulfilled based on the supported requirement *type*.

Test workflow with requirements

When a requirement is defined for a test, it is marked as a dependency for that test. The test will wait for all the requirements to complete successfully before it is started.

When any of the requirements defined on a test fails, the test is skipped.

Defining a test requirement

A test requirement is described in the JSON format. Following is an example of a requirement of *type package*:

```
{"type": "package", "name": "hello"}
```

To define a requirement for the test, use the test's docstring with the format of keywords `:avocado: requirement=`. The following example shows the same package requirement showed above inside a test docstring:

```
from avocado import Test


class PassTest(Test):
    """
    :avocado: requirement={"type": "package", "name": "hello"}
    """
    def test(self):
        """
        A success test
        """
```

It is possible to define multiple requirements for a test. Following is an example using more than one requirement definition:

```
from avocado import Test


class PassTest(Test):
    """
    :avocado: requirement={"type": "package", "name": "hello"}
    :avocado: requirement={"type": "package", "name": "bash"}
    """
    def test(self):
        """
        A success test
        """
```

Defining a requirement in the class docstring will fulfill the requirement for every test within a test class. Defining a requirement in the test docstring will fulfill the requirement for that single test only.

Supported types of requirements

The following *types* of requirements are supported:

Package

Support managing of packages using the Avocado Software Manager utility. The parameters available to use the package *type* of requirements are:

- *type*: *package*
- *name*: the package name (required)
- *action*: one of *install*, *check*, or *remove* (optional, defaults to *install*)

Following is an example of a test using the Package requirement:

```
from avocado import Test


class PassTest(Test):

    """
    Example test that passes.

    :avocado: requirement={"type": "package", "name": "hello"}
    """

    def test(self):
        """
        A test simply doesn't have to fail in order to pass
        """
```

Asset

Support fetching assets using the Avocado Assets utility. The parameters available to use the asset *type* of requirements are:

- *type*: *asset*

- *name*: the file name or uri (required)
- *asset_has*: hash of the file (optional)
- *algorithm*: hash algorithm (optional)
- *locations*: location(s) where the file can be fetched from (optional)
- *expire*: time in seconds for the asset to expire (optional)

9.2.10 Managing Assets

Note: Please note that we are constantly improving on how we handle assets inside Avocado. Probably some changes will be delivered during the next releases.

Assets are test artifacts that Avocado can download automatically either during the test execution, or before the test even starts (by parsing the test code or on-demand, manually registering them at the command-line).

Sometimes, depending on the use case, those assets can be a bottleneck for disk space. If the tests constantly use large assets, it is important to know how Avocado stores and handles those artifacts.

Listing assets

To list cached assets in the system, use the following command:

```
$ avocado assets list
```

This command supports `--by-size-filter` and `--by-days` options. When using the former, use a comparison filter and a size in bytes. For instance:

```
$ avocado assets list --by-size-filter=">=2048"
```

The command above will list only assets bigger than 2Kb. Avocado supports the following operators: =, >=, <=, < and >.

Now, to look for old assets (based on the access time), for example, 10 days older, use the `--by-days` option:

```
$ avocado assets list --by-days=10
```

Registering assets

To manually register a local asset in the cache, use the `register` command:

```
$ avocado assets register *NAME* *URL*
```

Where `NAME` is the unique name to associate with this asset and `URL` is the path to the local asset to be manually registered.

The `register` command also supports the `--hash` option, which allows the addition of the file's hash.

Fetching assets from instrumented tests

The `fetch` command allows the download of a limited definition of assets inside an Avocado Instrumented test. It uses a parser on instrumented test source to find `fetch_asset` calls composed of simple strings as parameters, or at least one level of variable in the same context with a string assignment, and fetch those assets without running the test. The only exception to strings as arguments is the `locations` parameter, which allows the user of a list.

Following are some examples of supported definitions of assets by the `fetch` command:

```
tarball_locations = [
    'https://mirrors.peers.community/mirrors/gnu/hello/hello-2.9.tar.gz',
    'https://mirrors.kernel.org/gnu/hello/hello-2.9.tar.gz',
    'http://gnu.c3sl.ufpr.br/ftp/hello-2.9.tar.gz',
    'ftp://ftp.funet.fi/pub/gnu/prep/hello/hello-2.9.tar.gz'
    ]
self.hello = self.fetch_asset(
    name='hello-2.9.tar.gz',
    asset_hash='cb0470b0e8f4f7768338f5c5cfe1688c90fbbc74',
    locations=tarball_locations)
```

```
kernel_url = ('https://archives.fedoraproject.org/pub/archive/fedora'
              '/linux/releases/29/Everything/x86_64/os/images/pxeboot'
              '/vmlinuz')
kernel_hash = '23bebd2680757891cf7adedb033532163a792495'
kernel_path = self.fetch_asset(kernel_url, asset_hash=kernel_hash)
```

To fetch the assets defined inside an instrumented test, use:

```
$ avocado assets fetch *AVOCADO_INSTRUMENTED*
```

Where `AVOCADO_INSTRUMENTED` is the path to the Avocado Instrumented file.

Removing assets

It is possible to remove files from the cache directories manually. The `purge` utility helps with that:

$ avocado assets purge –help

Assets can be removed applying the same filters as described when listing them. It is possible to remove assets by a size filter (`--by-size-filter`) or assets older than N days (`--by-days`).

Removing by overall cache limit

Besides the existing features, Avocado is able to set an overall limit, so that it matches the storage limitations locally or on CI systems.

For instance it may be the case that a GitLab cache limit is 4 GiB, in that case Avocado can sort assets by last access, and remove all that exceeds 4 GiB (that is, keep the last accessed 4 GiB worth of cached files). Use the `--by-overall-limit` option specifying the size limit:

```
$ avocado assets purge --by-overall-limit=4g
```

This ensures that the files which are not used for some time in the cache are automatically removed.

Please, note that at the moment, you can only use ‘b’, ‘k’, ‘m’, ‘g’, and ‘t’as suffixes.

Changing the default cache dirs

Assets are stored inside the `datadir.paths.cache_dirs` option. It is possible to change this in the configuration file. The current value is shown with the following command:

```
$ avocado config | grep datadir.paths.cache_dirs
```

9.2.11 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 finding those paths, to avoid cumbersome path manipulation magic.

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
    * /etc/avocado/conf.d/resultsdb.conf
    * /etc/avocado/conf.d/result_upload.conf
    * /etc/avocado/conf.d/jobscripts.conf
    * /etc/avocado/conf.d/gdb.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
    cache $HOME/avocado/data/cache
```

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.

9.2.12 Avocado logging system

This section describes the logging system used in Avocado.

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) are listed below:

app The text based UI (avocado.app)

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

debug Messages useful to debug the Avocado Framework (avocado.app.debug)

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 output.

Storing custom logs

When you run a test, you can also store custom logging streams into the results directory by running:

```
$ avocado run --store-logging-stream STREAM[:LEVEL][,STREAM[:LEVEL][,...]
```

This 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.

9.2.13 Understanding the plugin system

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

Note: A large number of out-of-the-box Avocado features are implemented as using the same plugin architecture available to third-party extensions.

This guide considers "core features", even though they're still 'plugable', those available with an installation of Avocado by itself (`pip install avocado-framework`). If a feature is part of an optional or third-party plugin package, this guide will reference it."

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):
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
...
```

Fully qualified named for a plugin

The Avocado plugin system uses namespaces to recognize and categorize plugins. The namespace separator here is the dot and every plugin that starts with `avocado.plugins.` will be recognized by the framework.

An example of a plugin's full qualified name:

`avocado.plugins.result.json`

This plugin will generate the job result in JSON format.

Note: Inside Avocado we will omit the prefix `avocado.plugins` to make the things clean.

Note: When listing plugins with `avocado plugins` pay attention to the namespace inside the parenthesis on each category description. You will realize that there are, for instance, two plugins with the name 'JSON'. But when you concatenate the fully qualified name it will become clear that they are actually two different plugins: `result.json` and `cli.json`.

Disabling a plugin

If you, as Avocado user, would like to disable a plugin, you can disable on config files.

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.

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`.

So, 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.

Changing 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.

Note: If you are interested on how plugins works and how to create your own plugin, visit the Plugin section on Contributor's Guide.

Pre and post plugins

Avocado provides interfaces (hooks) with which custom plugins can register to be called at various times. For instance, it's possible to trigger custom actions before and after the execution of a job, or before and after the execution of the tests from a job.

Let's discuss each interface briefly.

Before and after jobs

Avocado supports plug-ins which are (guaranteed to be) executed before the first test and after all tests finished.

The *pre* method of each installed plugin of type `job.prepost` will be called by the `run` command, that is, anytime an `avocado run <valid_test_reference>` command is executed.

Note: Conditions such as the `SystemExit` or `KeyboardInterrupt` exceptions being raised can interrupt the execution of those plugins.

Then, immediately after that, the job's *run* method is called, which attempts to run all job phases, from test suite creation to test execution.

Unless a `SystemExit` or `KeyboardInterrupt` is raised, or yet another major external event (like a system condition that Avocado can not control) it will attempt to run the `post` methods of all the installed plugins of type `job.prepost`. This even includes job executions where the *pre* plugin executions were interrupted.

Before and after tests

If you followed the previous section, you noticed that the job's *run* method was said to run all the test phases. Here's a sequence of the job phases:

1) *Creation of the test suite*

2) *Pre tests hook*

3) *Tests execution*

4) *Post tests hook*

Plugin writers can have their own code called at Avocado during a job by writing a that will be called at phase number 2 (`pre_tests`) by writing a method according to the *avocado.core.plugin_interfaces.JobPreTests()* interface. Accordingly, plugin writers can have their own called at phase number 4 (`post_tests`) by writing a method according to the *avocado.core.plugin_interfaces.JobPostTests()* interface.

Note that there's no guarantee that all of the first 3 job phases will be executed, so a failure in phase 1 (`create_test_suite`), may prevent the phase 2 (`pre_tests`) and/or 3 (`run_tests`) from from being executed.

Now, no matter what happens in the *attempted execution* of job phases 1 through 3, job phase 4 (`post_tests`) will be *attempted to be executed*. To make it extra clear, as long as the Avocado test runner is still in execution (that is, has not been terminated by a system condition that it can not control), it will execute plugin's `post_tests` methods.

As a concrete example, a plugin' `post_tests` method would not be executed after a `SIGKILL` is sent to the Avocado test runner on phases 1 through 3, because the Avocado test runner would be promptly interrupted. But, a `SIGTERM` and `KeyboardInterrupt` sent to the Avocado test runner under phases 1 though 3 would still cause the test runner to run `post_tests` (phase 4). Now, if during phase 4 a `KeyboardInterrupt` or `SystemExit` is received, the remaining plugins' `post_tests` methods 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.

Tests' logs plugin

It's natural that Avocado will be used in environments where access to the integral job results won't be easily accessible.

For instance, on Continuous Integration (CI) services, one usually gets access to the output produced on the console, while access to other files produced (generally called artifacts) may or may not be accessible.

For this reason, it may be helpful to simply output the logs for tests that have "interesting" outcomes, which usually means that fail and need to be investigated.

To show the content for test that are canceled, skipped and fail, you can set on your configuration file:

```
[job.output.testlogs]
statuses = ["CANCEL", "SKIP", "FAIL"]
```

At the end of the job, a header will be printed for each test that ended with any of the statuses given, followed by the raw content of its respective log file.

9.2.14 Understanding the test discovery (Avocado Loaders)

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

Note: The definitions here apply to the legacy runner.

Test Loaders

A Test Loader is an Avocado component that is responsible for discovering tests that Avocado can run. In the process, Avocado gathers enough information to allow the test to be run. Additionally, Avocado collects extra information available within the test, such as tags that can be used to filter out tests from actual execution.

This whole process is, unless otherwise stated or manually configured, safe, in the sense that no test code will be executed.

How Loaders discover tests

Avocado will apply ordering to the discovery process, so loaders that run earlier, will have higher precedence in discovering tests.

A loader implementation is free to implement whatever logic it needs to discover tests. The important fact about how a loader discover tests is that it should return one or more "test factory", an internal data structure that, as stated before, contains enough information to allow the test to be executed.

The order of test loaders

As described in previous sections, Avocado supports different types of test starting with `SIMPLE` tests, which are simply executable files, the basic Python unittest and tests called `INSTRUMENTED`.

With additional plugins new test types can be supported, 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.

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

```
$ avocado run --test-runner=runner 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 --test-runner=runner 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
```

Test References

A Test Reference is a string that can be resolved into (interpreted as) one or more tests by the Avocado Test Resolver.

Each resolver (a.k.a. loader) can handle the Test References differently. For example, External Loader will use the Test Reference as an argument for the external command, while the File Loader will expect a file path.

If you don't specify the loader that you want to use, all of the available loaders will be used to resolve the provided Test References. One by one, the Test References will be resolved by the first loader able to create a test list out of that reference.

Basic Avocado Loaders

Below you can find some extra details about the specific builtin Avocado loaders. For Loaders introduced to Avocado via plugins (VT, Robot, . . .), please refer to the corresponding loader/plugin documentation.

File Loader

For the File Loader, the loader responsible for discovering INSTRUMENTED, PyUNITTEST (classic python unittests) and SIMPLE tests.

If the file corresponds to an INSTRUMENTED or PyUNITTEST test, you can filter the Test IDs by adding to the Test Reference a `:` followed by a regular expression.

For instance, if you want to list all tests that are present in the `gdbtest.py` file, you can use the list command below:

```
$ avocado list --loader examples/tests/gdbtest.py
INSTRUMENTED examples/tests/gdbtest.py:GdbTest.test_start_exit
INSTRUMENTED examples/tests/gdbtest.py:GdbTest.test_existing_commands_raw
INSTRUMENTED examples/tests/gdbtest.py:GdbTest.test_existing_commands
INSTRUMENTED examples/tests/gdbtest.py:GdbTest.test_load_set_breakpoint_run_exit_raw
INSTRUMENTED examples/tests/gdbtest.py:GdbTest.test_load_set_breakpoint_run_exit
INSTRUMENTED examples/tests/gdbtest.py:GdbTest.test_generate_core
INSTRUMENTED examples/tests/gdbtest.py:GdbTest.test_set_multiple_break
INSTRUMENTED examples/tests/gdbtest.py:GdbTest.test_disconnect_raw
INSTRUMENTED examples/tests/gdbtest.py:GdbTest.test_disconnect
```

(continues on next page)

(continued from previous page)

```
INSTRUMENTED examples/tests/gdbtest.py:GdbTest.test_remote_exec
INSTRUMENTED examples/tests/gdbtest.py:GdbTest.test_stream_messages
INSTRUMENTED examples/tests/gdbtest.py:GdbTest.test_connect_multiple_clients
INSTRUMENTED examples/tests/gdbtest.py:GdbTest.test_server_exit
INSTRUMENTED examples/tests/gdbtest.py:GdbTest.test_multiple_servers
INSTRUMENTED examples/tests/gdbtest.py:GdbTest.test_server_stderr
INSTRUMENTED examples/tests/gdbtest.py:GdbTest.test_server_stdout
INSTRUMENTED examples/tests/gdbtest.py:GdbTest.test_remote
```

To filter the results, listing only the tests that have `test_disconnect` in their test method names, you can execute:

```
$ avocado list --loader examples/tests/gdbtest.py:test_disconnect
INSTRUMENTED examples/tests/gdbtest.py:GdbTest.test_disconnect_raw
INSTRUMENTED examples/tests/gdbtest.py:GdbTest.test_disconnect
```

As the string after the `:` is a regular expression, two tests were filtered in. You can manipulate the regular expression to have only the test with that exact name:

```
$ avocado list --loader examples/tests/gdbtest.py:test_disconnect$
INSTRUMENTED examples/tests/gdbtest.py:GdbTest.test_disconnect
```

The regular expression enables you to have more complex filters. Example:

```
$ avocado list --loader examples/tests/gdbtest.py:GdbTest.test_[le].*raw
INSTRUMENTED examples/tests/gdbtest.py:GdbTest.test_existing_commands_raw
INSTRUMENTED examples/tests/gdbtest.py:GdbTest.test_load_set_breakpoint_run_exit_raw
```

Once the test reference is providing you the expected outcome, you can replace the `list` subcommand with the `run` subcommand to execute your tests:

```
$ avocado run --test-runner=runner examples/tests/gdbtest.py:GdbTest.test_[le].*raw
JOB ID     : 333912fb02698ed5339a400b832795a80757b8af
JOB LOG    : $HOME/avocado/job-results/job-2017-06-14T14.54-333912f/job.log
 (1/2) examples/tests/gdbtest.py:GdbTest.test_existing_commands_raw: PASS (0.59 s)
 (2/2) examples/tests/gdbtest.py:GdbTest.test_load_set_breakpoint_run_exit_raw: PASS␣
↪(0.42 s)
RESULTS    : PASS 2 | ERROR 0 | FAIL 0 | SKIP 0 | WARN 0 | INTERRUPT 0 | CANCEL 0
JOB TIME   : 1.15 s
JOB HTML   : $HOME/avocado/job-results/job-2017-06-14T14.54-333912f/html/results.html
```

Warning: Specially when using regular expressions, it's recommended to individually enclose your Test References in quotes to avoid bash of corrupting them. In that case, the command from the example above would be: `avocado run "examples/tests/gdbtest.py:GdbTest.test_[le].*raw"`

External Loader

Using the External Loader, Avocado will consider that and External Runner will be in place and so Avocado doesn't really need to resolve the references. Instead, Avocado will pass the references as parameters to the External Runner. Example:

```
$ avocado run --test-runner=runner 20
Unable to resolve reference(s) '20' with plugins(s) 'file', 'robot',
'vt', 'external', try running 'avocado -V list 20' to see the details.
```

In the command above, no loaders can resolve `20` as a test. But running the command above with the External Runner `/bin/sleep` will make Avocado to actually execute `/bin/sleep 20` and check for its return code:

```
$ avocado run --test-runner=runner 20 --loaders external:/bin/sleep
JOB ID     : 42215ece2894134fb9379ee564aa00f1d1d6cb91
JOB LOG    : $HOME/avocado/job-results/job-2017-06-19T11.17-42215ec/job.log
 (1/1) 20: PASS (20.03 s)
RESULTS    : PASS 1 | ERROR 0 | FAIL 0 | SKIP 0 | WARN 0 | INTERRUPT 0 | CANCEL 0
JOB TIME   : 20.13 s
JOB HTML   : $HOME/avocado/job-results/job-2017-06-19T11.17-42215ec/html/results.html
```

Warning: It's safer to put your Test References at the end of the command line, after a –. That will avoid argument vs. Test References clashes. In that case, everything after the – will be considered positional arguments, therefore Test References. Considering that syntax, the command for the example above would be: `avocado run --loaders external:/bin/sleep -- 20`

TAP Loader

This loader enables Avocado to execute binaries or scripts and parse their Test Anything Protocol output.

The tests can be run as usual:

```
$ avocado run --test-runner=runner --loaders tap -- ./mytaptest
```

Notice that you have to be explicit about the test loader you're using, otherwise, since the test files are executable binaries, the `FileLoader` will detect the file as a `SIMPLE` test, making the whole test suite to be executed as one test only from the Avocado perspective. Because TAP test programs should exit with a zero exit status, this will cause the test to pass even if there are failures.

9.2.15 Advanced usage

Test Runner Selection

To effectively run a job with tests, Avocado makes use of a well described and pluggable interface. This means that users can choose (and developers can write) their own runners.

Runner choices can be seen by running `avocado plugins`:

```
...
Plugins that run test suites on a job (runners):
nrunner nrunner based implementation of job compliant runner
runner  The conventional test runner
```

And to select a different test runner, say, the legacy `runner`:

```
avocado run --test-runner=runner ...
```

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 -- $@
```

This example file is available at `examples/wrappers/strace.sh`.

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

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

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

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

The following is a working example:

```
$ avocado run --wrapper examples/wrappers/strace.sh /bin/true
```

The strace file will be located at Avocado log directory, on `test-results/1-_bin_true/` subdirectory.

Caveats

- 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.

9.2.16 What's next?

Now that you are familiar with the basic concepts and Avocado usage, you can write your tests.

As said before, you can write test on your favorite language. But if you would like to use the Avocado libraries and facilities, you can use Python or Bash.

If you would like to move forward on Avocado, we prepared the *"Avocado Test Writer's Guide"* for you. Have fun!

9.3 Avocado Test Writer's Guide

9.3.1 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.

9.3.2 Writing Avocado Tests with Python

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.

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*.

Note: Avocado also supports coroutines as tests. Simply declare your test method using the `async def` syntax, and Avocado will run it inside an asyncio loop.

Multiple tests and naming conventions

You can have multiple tests in a single class.

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.

[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.

- 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`, `self.error` or `self.cancel` directly from test.

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 command 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.

Turning errors into cancels

It is also possible to assume unhandled exception to be as a test `CANCEL` instead of a test `ERROR` simply by using `cancel_on` decorator:

```
def test(self):
    @avocado.cancel_on(TypeError)
    def foo():
        raise TypeError
    foo()
```

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 `$RESULTS/test-results/$TEST_ID/whiteboard`, 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 data files

Some tests can depend on data files, external to the test file itself. Avocado provides a test API that makes it really easy to access such files: *get_data()*.

For Avocado tests (that is, INSTRUMENTED tests) *get_data()* allows test data files to be accessed from up to three sources:

- **file** level data directory: a directory named after the test file, but ending with .data. For a test file /home/user/test.py, the file level data directory is /home/user/test.py.data/.
- **test** level data directory: a directory named after the test file and the specific test name. These are useful when different tests part of the same file need different data files (with the same name or not). Considering the previous example of /home/user/test.py, and supposing it contains two tests, MyTest.test_foo and MyTest.test_bar, the test level data directories will be, /home/user/test.py.data/MyTest.test_foo/ and home/user/test.py.data/MyTest.test_bar/ respectively.
- **variant** level data directory: if variants are being used during the test execution, a directory named after the variant will also be considered when looking for test data files. For test file /home/user/test.py, and test MyTest.test_foo, with variant debug-ffff, the data directory path will be /home/user/test.py.data/MyTest.test_foo/debug-ffff/.

Note: Unlike INSTRUMENTED tests, SIMPLE tests only define file and variant data_dirs, therefore the most-specific data-dir might look like /bin/echo.data/debug-ffff/.

Avocado looks for data files in the order defined at *DATA_SOURCES*, which are from most specific one, to most generic one. That means that, if a variant is being used, the **variant** directory is used first. Then the **test** level directory is attempted, and finally the **file** level directory. Additionally you can use get_data(filename, must_exist=False) to get expected location of a possibly non-existing file, which is useful when you intend to create it.

Tip: When running tests you can use the --log-test-data-directories command line option log the test data directories that will be used for that specific test and execution conditions (such as with or without variants). Look for "Test data directories" in the test logs.

Note: The previously existing API avocado.core.test.Test.datadir, used to allow access to the data directory based on the test file location only. This API has been removed. If, for whatever reason you still need to access the data directory based on the test file location only, you can use get_data(filename='', source='file', must_exist=False) instead.

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 the previous section we described how parameters are handled. Now, let's have a look at how to produce them and execute your tests with different parameters.

The variants subsystem is what allows the creation of multiple variations of parameters, and the execution of tests with those parameter variations. This subsystem is pluggable, so you might use custom plugins to produce variants. To keep things simple, let's use Avocado's primary implementation, called "yaml_to_mux".

The "yaml_to_mux" plugin accepts YAML files. Those will create a tree-like structure, store the variables as parameters and use 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:
```

(continues on next page)

(continued from previous page)

```
        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 (8):

Variant builtin-one_cycle-f659:    /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 builtin-six_cycles-723b:    /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 builtin-one_hundred_cycles-633a:    /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 builtin-six_hundred_cycles-a570:    /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 shell-one_cycle-55f5:    /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
```

(continues on next page)

(continued from previous page)

```
Variant shell-six_cycles-9e23:    /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 shell-one_hundred_cycles-586f:    /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 shell-six_hundred_cycles-1e84:    /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 --␣
↪examples/tests/passtest.py
JOB ID     : cc7ef22654c683b73174af6f97bc385da5a0f02f
JOB LOG    : $HOME/avocado/job-results/job-2017-01-22T11.26-cc7ef22/job.log
  (1/8) examples/tests/passtest.py:PassTest.test;run-sleeptenmin-builtin-variants-one_
↪cycle-0aae: STARTED
  (1/8) examples/tests/passtest.py:PassTest.test;run-sleeptenmin-builtin-variants-one_
↪cycle-0aae: PASS (0.01 s)
  (2/8) examples/tests/passtest.py:PassTest.test;run-sleeptenmin-builtin-variants-six_
↪cycles-ca95: STARTED
  (2/8) examples/tests/passtest.py:PassTest.test;run-sleeptenmin-builtin-variants-six_
↪cycles-ca95: PASS (0.01 s)
  (3/8) examples/tests/passtest.py:PassTest.test;run-sleeptenmin-builtin-variants-one_
↪hundred_cycles-e897: STARTED
  (3/8) examples/tests/passtest.py:PassTest.test;run-sleeptenmin-builtin-variants-one_
↪hundred_cycles-e897: PASS (0.01 s)
  (4/8) examples/tests/passtest.py:PassTest.test;run-sleeptenmin-builtin-variants-six_
↪hundred_cycles-b0b0: STARTED
  (4/8) examples/tests/passtest.py:PassTest.test;run-sleeptenmin-builtin-variants-six_
↪hundred_cycles-b0b0: PASS (0.01 s)
  (5/8) examples/tests/passtest.py:PassTest.test;run-sleeptenmin-shell-variants-one_
↪cycle-f35d: STARTED
  (5/8) examples/tests/passtest.py:PassTest.test;run-sleeptenmin-shell-variants-one_
↪cycle-f35d: PASS (0.01 s)
  (6/8) examples/tests/passtest.py:PassTest.test;run-sleeptenmin-shell-variants-six_
↪cycles-56b6: STARTED
  (6/8) examples/tests/passtest.py:PassTest.test;run-sleeptenmin-shell-variants-six_
↪cycles-56b6: PASS (0.01 s)
  (7/8) examples/tests/passtest.py:PassTest.test;run-sleeptenmin-shell-variants-one_
↪hundred_cycles-ec04: STARTED
  (7/8) examples/tests/passtest.py:PassTest.test;run-sleeptenmin-shell-variants-one_
↪hundred_cycles-ec04: PASS (0.01 s)
  (8/8) examples/tests/passtest.py:PassTest.test;run-sleeptenmin-shell-variants-six_
↪hundred_cycles-8fff: STARTED
  (8/8) examples/tests/passtest.py:PassTest.test;run-sleeptenmin-shell-variants-six_
↪hundred_cycles-8fff: PASS (0.01 s)
```

(continues on next page)

(continued from previous page)

```
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*.

`unittest.TestCase` heritage

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

The code example below 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

To perform setup actions before/after your test, you may use `setUp` and `tearDown` methods. The `tearDown` method is always executed even on `setUp` failure so don't forget to initialize your variables early in the `setUp`. Example of usage is in the next section *Running third party test suites*.

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 python3

import os

from avocado import Test
from avocado.utils import archive, build, process


class SyncTest(Test):

    """
    Execute the synctest test suite.

    :param sync_tarball: path to the tarball relative to a data directory
    :param default_symbols: whether to build with debug symbols (bool)
    :param sync_length: how many data should by used in sync test
    :param sync_loop: how many writes should be executed in sync test
    """

    def setUp(self):
        """
        Build the synctest suite.
        """
        self.cwd = os.getcwd()
        sync_tarball = self.params.get('sync_tarball', '*', 'synctest.tar.bz2')
        tarball_path = self.get_data(sync_tarball)
        if tarball_path is None:
            self.cancel('Test is missing data file %s' % tarball_path)
        archive.extract(tarball_path, self.workdir)
        srcdir = os.path.join(self.workdir, 'synctest')
        os.chdir(srcdir)
        if self.params.get('debug_symbols', default=True):
            build.make(srcdir,
```

(continues on next page)

(continued from previous page)

```
                    env={'CFLAGS': '-g -O0'},
                    extra_args='synctest',
                    allow_output_check='none')
        else:
            build.make(srcdir,
                    allow_output_check='none')

    def test(self):
        """
        Execute synctest with the appropriate params.
        """
        path = os.path.join(os.getcwd(), 'synctest')
        cmd = ('%s %s %s' %
               (path, self.params.get('sync_length', default=100),
                self.params.get('sync_loop', default=10)))
        process.system(cmd)
        os.chdir(self.cwd)
```

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.get_data()`, 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 fetch 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, Avocado 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/data/cache']
```

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

If the tester does not provide a `cache_dirs` for the test execution, Avocado creates 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 = 'https://fossies.org/linux/privat/stress-1.0.4.tar.gz'
        tarball = self.fetch_asset(stress)
        archive.extract(tarball, self.workdir)
...
```

  In this case, `fetch_asset()` will download the file from the URL provided, copying it to the `$data_dir/cache` directory. The `fetch_asset()` method returns the target location of the fetched asset. In this example, the `tarball` variable holds `/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 = 'https://fossies.org/linux/privat/stress-1.0.4.tar.gz'
        tarball = self.fetch_asset(stress)
        archive.extract(tarball, self.workdir)
...
```

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

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

```
...
    def setUp(self):
        st_name = 'stress-1.0.4.tar.gz'
        st_hash = 'e1533bc704928ba6e26a362452e6db8fd58b1f0b'
        st_loc = ['https://fossies.org/linux/privat/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.workdir)
...
```

In this case, Avocado tries to download `stress-1.0.4.tar.gz` from the provided locations list (if it's not already in the default cache, `~/avocado/data/cache`). As the hash was also provided, Avocado verifies the hash. To do so, Avocado first looks for a hash file named `stress-1.0.4.tar.gz.CHECKSUM` in the same directory. If the hash file is not available, Avocado computes the hash and creates the hash file for later use.

The resulting `tarball` variable content will be `~/avocado/cache/stress-1.0.4.tar.gz`. An exception is raised if Avocado fails to download or to verify the file.

- Use case 4: Low bandwidth available for download of a large file which takes a lot of time to download and causes a CI, like Travis, for example, to timeout the test execution. Do not cancel the test if the file is not available:

```
...
    def setUp(self):
        st_name = 'stress-1.0.4.tar.gz'
        st_hash = 'e1533bc704928ba6e26a362452e6db8fd58b1f0b'
        st_loc = ['https://fossies.org/linux/privat/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, find_only=True)
        archive.extract(tarball, self.workdir)
...
```

Setting the `find_only` parameter to `True` will make Avocado look for the asset in the cache, but will not attempt to download it if the asset is not available. The asset download can be done prior to the test execution using the command-line `avocado assets fetch INSTRUMENTED`.

In this example, if the asset is not available in the cache, the test will continue to run and when the test tries to use the asset, it will fail. A solution for that is presented in the next use case.

- Use case 5: Low bandwidth available for download or a large file which takes a lot of time to download and causes a CI, like Travis, for example, to timeout the test execution. Cancel the test if the file is not available:

```
...
    def setUp(self):
        st_name = 'stress-1.0.4.tar.gz'
        st_hash = 'e1533bc704928ba6e26a362452e6db8fd58b1f0b'
        st_loc = ['https://fossies.org/linux/privat/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, find_only=True,
                                   cancel_on_missing=True)
        archive.extract(tarball, self.workdir)
...
```

With `cancel_on_missing` set to `True` and `find_only` set to `True`, if the file is not available in the cache, the test is canceled.

Detailing the `fetch_asset()` parameters:

- `name`: The destination name used to the fetched file. It can also contains a full URI. The URI will be used as the location (after searching into the cache directories).
- `asset_hash`: (optional) The expected hash for the file. If missing, Avocado skips the hash check. If provided, before computing the hash, Avocado looks for a hash file to verify the asset. If the hash file is not available, Avocado computes the hash and creates the hash file in the same cache directory for later use.
- `algorithm`: (optional) Provided hash algorithm format. Defaults to sha1.
- `locations`: (optional) List of locations used to try to fetch the file. The supported schemes are `http://`, `https://`, `ftp://` and `file://`. The tester should inform the full url to the file, including the file name. The first fetch success skips the next locations. Notice that for `file://` Avocado creates a symbolic link in the cache directory, pointing to the original location of the file.
- `expire`: (optional) period while a cached file is considered valid. After that period, the file will be downloaded 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).
- `find_only`: (optional) tries to find the asset in the cache. If the asset file is not available in the cache, Avocado will not attempt to download it.
- `cancel_on_missing` (optional) if set to `True`, cancel the current running test if there is a problem while downloading the asset or if `find_only=True` and the asset is not available in the cache.

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

Test Output Check and Output Record Mode

Warning: The *–output-check-record* feature is legacy feature and it's works only with the legacy runner. For using legacy runner you have to use *–test-runner=runner*

In a lot of occasions, you want to go simpler: just check if the output of a given test matches an expected output. In order to help with this common use case, Avocado provides the `--output-check-record` option:

```
--output-check-record {none,stdout,stderr,both,combined,all}
                      Record the output produced by each test (from stdout
                      and stderr) into both the current executing result and
                      into reference files. Reference files are used on
                      subsequent runs to determine if the test produced the
```

(continues on next page)

(continued from previous page)

```
                        expected output or not, and the current executing
                        result is used to check against a previously recorded
                        reference file. Valid values: 'none' (to explicitly
                        disable all recording) 'stdout' (to record standard
                        output *only*), 'stderr' (to record standard error
                        *only*), 'both' (to record standard output and error
                        in separate files), 'combined' (for standard output
                        and error in a single file). 'all' is also a valid but
                        deprecated option that is a synonym of 'both'.
```

If this option is used, Avocado will store the content generated by the test in the standard (POSIX) streams, that is, `STDOUT` and `STDERR`. Depending on the option chosen, you may end up with different files recorded (into what we call "reference files"):

- `stdout` will produce a file named `stdout.expected` with the contents from the test process standard output stream (file descriptor 1)
- `stderr` will produce a file named `stderr.expected` with the contents from the test process standard error stream (file descriptor 2)
- `both` will produce both a file named `stdout.expected` and a file named `stderr.expected`
- `combined`: will produce a single file named `output.expected`, with the content from both test process standard output and error streams (file descriptors 1 and 2)
- `none` will explicitly disable all recording of test generated output and the generation reference files with that content

The reference files will be recorded in the first (most specific) test's data dir (*Accessing test data files*). Let's take as an example the test `synctest.py`. In a fresh checkout of the Avocado source code you can find the following reference files:

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

From those 2 files, only stdout.expected has some content:

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

This means that during a previous test execution, output was recorded with option `--output-check-record both` and content was generated on the `STDOUT` stream only:

```
$ avocado run --test-runner=runner --output-check-record both synctest.py
JOB ID     : b6306504351b037fa304885c0baa923710f34f4a
JOB LOG    : $JOB_RESULTS_DIR/job-2017-11-26T16.42-b630650/job.log
 (1/1) examples/tests/synctest.py:SyncTest.test: PASS (2.03 s)
RESULTS    : PASS 1 | ERROR 0 | FAIL 0 | SKIP 0 | WARN 0 | INTERRUPT 0 | CANCEL 0
JOB TIME   : 2.26 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. From this point on, after such as test (one with a reference file recorded) has finished running, Avocado will check if the output generated match the reference(s) file(s) content. If they don't match, the test will finish with a `FAIL` status.

You can disable this automatic check when a reference file exists by passing `--disable-output-check` to the test runner.

Tip: The *avocado.utils.process* APIs have a parameter called `allow_output_check` that let you individually select the output that will be part of the test output and recorded reference files. Some other APIs built on top of *avocado.utils.process*, such as the ones in *avocado.utils.build* also provide the same parameter.

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 examples/tests/output_check.sh
#!/bin/sh

# if you execute this script with avocado, avocado will check its stdout
# against the file stdout.expected and its stderr with the file stderr.expected,
# both located in output_check.sh.data, in the same directory as this source
# file.

# The expected files were generated using the option --output-check-record all
# of the avocado runner:
# avocado run output_check.sh --output-check-record all

echo "Hello, world!"
```

Let's record the output for this one:

```
$ avocado run --test-runner=runner output_check.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_check.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 the test data directory appeared in the same level of our shell script, containing 2 files:

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

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

```
$ cat output_check.sh.data/stdout.expected
Hello, world!
$ cat output_check.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!`:

```
$ avocado run --test-runner=runner output_check.sh
JOB ID    : f0521e524face93019d7cb99c5765aedd933cb2e
JOB LOG   : $HOME/avocado/job-results/job-2014-09-25T20.52-f0521e5/job.log
 (1/1) output_check.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/latest/job.log
    2017-10-16 14:23:02,567 test             L0381 INFO | START 1-output_check.sh
    2017-10-16 14:23:02,568 test             L0402 DEBUG| Test metadata:
    2017-10-16 14:23:02,568 test             L0403 DEBUG|   filename: $HOME/output_
↪check.sh
    2017-10-16 14:23:02,596 process          L0389 INFO | Running '$HOME/output_check.
↪sh'
    2017-10-16 14:23:02,603 process          L0499 INFO | Command '$HOME/output_check.
↪sh' finished with 0 after 0.00131011009216s
    2017-10-16 14:23:02,602 process          L0479 DEBUG| [stdout] Hello, world!
    2017-10-16 14:23:02,603 test             L1084 INFO | Exit status: 0
    2017-10-16 14:23:02,604 test             L1085 INFO | Duration: 0.00131011009216
    2017-10-16 14:23:02,604 test             L0274 DEBUG| DATA (filename=stdout.
↪expected) => $HOME/output_check.sh.data/stdout.expected (found at file source dir)
    2017-10-16 14:23:02,605 test             L0740 DEBUG| Stdout Diff:
    2017-10-16 14:23:02,605 test             L0742 DEBUG| --- $HOME/output_check.sh.
↪data/stdout.expected
    2017-10-16 14:23:02,605 test             L0742 DEBUG| +++ $HOME/avocado/job-
↪results/job-2017-10-16T14.23-8cba866/test-results/1-output_check.sh/stdout
    2017-10-16 14:23:02,605 test             L0742 DEBUG| @@ -1 +1 @@
    2017-10-16 14:23:02,605 test             L0742 DEBUG| -Hello, Avocado!
    2017-10-16 14:23:02,605 test             L0742 DEBUG| +Hello, world!
    2017-10-16 14:23:02,606 stacktrace       L0041 ERROR|
    2017-10-16 14:23:02,606 stacktrace       L0044 ERROR| Reproduced traceback from:
↪$HOME/git/avocado/avocado/core/test.py:872
    2017-10-16 14:23:02,606 stacktrace       L0047 ERROR| Traceback (most recent call␣
↪last):
    2017-10-16 14:23:02,606 stacktrace       L0047 ERROR|   File "$HOME/git/avocado/
↪avocado/core/test.py", line 743, in _check_reference_stdout
    2017-10-16 14:23:02,606 stacktrace       L0047 ERROR|     self.fail('Actual test␣
↪sdtout differs from expected one')
    2017-10-16 14:23:02,606 stacktrace       L0047 ERROR|   File "$HOME//git/avocado/
↪avocado/core/test.py", line 983, in fail
    2017-10-16 14:23:02,607 stacktrace       L0047 ERROR|     raise exceptions.
↪TestFail(message)
    2017-10-16 14:23:02,607 stacktrace       L0047 ERROR| TestFail: Actual test␣
↪sdtout differs from expected one
    2017-10-16 14:23:02,607 stacktrace       L0048 ERROR|
    2017-10-16 14:23:02,607 test             L0274 DEBUG| DATA (filename=stderr.
↪expected) => $HOME//output_check.sh.data/stderr.expected (found at file source dir)
    2017-10-16 14:23:02,608 test             L0965 ERROR| FAIL 1-output_check.sh ->␣
↪TestFail: Actual test sdtout differs from expected one
```

As expected, the test failed because we changed its expectations, so an unified diff was logged. The unified diffs are also present in the files *stdout.diff* and *stderr.diff*, present in the test results directory:

```
$ cat $HOME/avocado/job-results/latest/test-results/1-output_check.sh/stdout.diff
--- $HOME/output_check.sh.data/stdout.expected
+++ $HOME/avocado/job-results/job-2017-10-16T14.23-8cba866/test-results/1-output_
 ↪check.sh/stdout
@@ -1 +1 @@
-Hello, Avocado!
+Hello, world!
```

Note: Currently the *stdout*, *stderr* and *output* files are stored in text mode. Data that can not be decoded according to current locale settings, will be replaced according to https://docs.python.org/3/library/codecs.html#codecs.replace_

errors.

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 examples/tests/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) examples/tests/sleeptest.py:SleepTest.test;run-0fc1: STARTED
    (1/1) examples/tests/sleeptest.py:SleepTest.test;run-0fc1: INTERRUPTED: timeout␣
↪(3.01 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
    2021-10-01 15:44:53,622 job              L0319 INFO | Multiplex tree␣
↪representation:
2021-10-01 15:44:53,622 job              L0319 INFO |  \-- run
2021-10-01 15:44:53,622 job              L0319 INFO |
2021-10-01 15:44:53,622 job              L0319 INFO | Multiplex variants (1):
2021-10-01 15:44:53,622 job              L0319 INFO | Variant run-0fc1:    /run
2021-10-01 15:44:53,622 job              L0312 INFO | Temporary dir: /tmp/avocado_tmp_
↪hp4cswyn/avocado_job_pmn___6i
2021-10-01 15:44:53,622 job              L0313 INFO |
2021-10-01 15:44:53,622 job              L0306 INFO | Job ID:␣
↪927fdc4143e9e093a485319820825faacc0f36a3
2021-10-01 15:44:53,622 job              L0309 INFO |
2021-10-01 15:44:54,165 selector_events  L0059 DEBUG| Using selector: EpollSelector
2021-10-01 15:44:54,622 testlogs         L0094 INFO | examples/tests/sleeptest.
↪py:SleepTest.test;run-0fc1: STARTED
2021-10-01 15:44:57,653 testlogs         L0101 INFO | examples/tests/sleeptest.
↪py:SleepTest.test;run-0fc1: INTERRUPTED
2021-10-01 15:44:57,654 testlogs         L0103 INFO | More information in /home/
↪jarichte/avocado/job-results/job-2021-10-01T15.44-927fdc4/test-results/1-examples_
↪tests_sleeptest.py_SleepTest.test_run-0fc1
2021-10-01 15:44:57,762 job              L0643 INFO | Test results available in /home/
↪jarichte/avocado/job-results/job-2021-10-01T15.44-927fdc4
```

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

To skip tests is in Avocado, you must use one of the Avocado skip decorators:

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

Those decorators can be used with classes and 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) /tmp/test_skip_decorators.py:MyTest.test1: STARTED
    (1/3) /tmp/test_skip_decorators.py:MyTest.test1: SKIP: Skipping on True condition.
    (2/3) /tmp/test_skip_decorators.py:MyTest.test2: STARTED
    (2/3) /tmp/test_skip_decorators.py:MyTest.test2: SKIP: Don't want this test now.
    (3/3) /tmp/test_skip_decorators.py:MyTest.test3: STARTED
    (3/3) /tmp/test_skip_decorators.py:MyTest.test3: PASS (0.01 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.

Note: It's an erroneous condition, reported with test status `ERROR`, to use any of the skip decorators on the `tearDown()` method.

Advanced Conditionals

More advanced use cases may require to evaluate the condition for skipping tests later, and may also need to introspect into the class that contains the test method in question.

It's possible to achieve both by supplying a callable to the condition parameters instead. The following example does just that:

```
from avocado import Test, skipIf, skipUnless


class BaseTest(Test):
    """Base class for tests
```

(continues on next page)

(continued from previous page)

```
    :avocado: disable
    """

    SUPPORTED_ENVS = []

    @skipUnless(lambda x: 'BARE_METAL' in x.SUPPORTED_ENVS,
                'Bare metal environment is required')
    def test_bare_metal(self):
        pass

    @skipIf(lambda x: getattr(x, 'MEMORY', 0) < 4096,
            'Not enough memory for test')
    def test_large_memory(self):
        pass

    @skipUnless(lambda x: 'VIRTUAL_MACHINE' in x.SUPPORTED_ENVS,
                'Virtual Machine environment is required')
    def test_nested_virtualization(self):
        pass

    @skipUnless(lambda x: 'CONTAINER' in x.SUPPORTED_ENVS,
                'Container environment is required')
    def test_container(self):
        pass


class BareMetal(BaseTest):

    SUPPORTED_ENVS = ['BARE_METAL']
    MEMORY = 2048

    def test_specific(self):
        pass


class NonBareMetal(BaseTest):

    SUPPORTED_ENVS = ['VIRTUAL_MACHINE', 'CONTAINER']

    def test_specific(self):
        pass
```

Even though the conditions for skipping tests are defined in the `BaseTest` class, the conditions will be evaluated when the tests are actually checked for execution, in the `BareMetal` and `NonBareMetal` classes. The result of running that test is:

```
JOB ID     : 77d636c93ed3b5e6fef9c7b6c8d9fe0c84af1518
JOB LOG    : $HOME/avocado/job-results/job-2021-03-17T20.10-77d636c/job.log
 (01/10) examples/tests/skip_conditional.py:BareMetal.test_specific: STARTED
 (01/10) examples/tests/skip_conditional.py:BareMetal.test_specific: PASS (0.01 s)
 (02/10) examples/tests/skip_conditional.py:BareMetal.test_bare_metal: STARTED
 (02/10) examples/tests/skip_conditional.py:BareMetal.test_bare_metal: PASS (0.01 s)
 (03/10) examples/tests/skip_conditional.py:BareMetal.test_large_memory: STARTED
 (03/10) examples/tests/skip_conditional.py:BareMetal.test_large_memory: SKIP: Not␣
↪enough memory for test
```

(continues on next page)

(continued from previous page)

```
 (04/10) examples/tests/skip_conditional.py:BareMetal.test_nested_virtualization:␣
↪STARTED
 (04/10) examples/tests/skip_conditional.py:BareMetal.test_nested_virtualization:␣
↪SKIP: Virtual Machine environment is required
 (05/10) examples/tests/skip_conditional.py:BareMetal.test_container: STARTED
 (05/10) examples/tests/skip_conditional.py:BareMetal.test_container: SKIP: Container␣
↪environment is required
 (06/10) examples/tests/skip_conditional.py:NonBareMetal.test_specific: STARTED
 (06/10) examples/tests/skip_conditional.py:NonBareMetal.test_specific: PASS (0.01 s)
 (07/10) examples/tests/skip_conditional.py:NonBareMetal.test_bare_metal: STARTED
 (07/10) examples/tests/skip_conditional.py:NonBareMetal.test_bare_metal: SKIP: Bare␣
↪metal environment is required
 (08/10) examples/tests/skip_conditional.py:NonBareMetal.test_large_memory: STARTED
 (08/10) examples/tests/skip_conditional.py:NonBareMetal.test_large_memory: SKIP: Not␣
↪enough memory for test
 (09/10) examples/tests/skip_conditional.py:NonBareMetal.test_nested_virtualization:␣
↪STARTED
 (09/10) examples/tests/skip_conditional.py:NonBareMetal.test_nested_virtualization:␣
↪PASS (0.01 s)
 (10/10) examples/tests/skip_conditional.py:NonBareMetal.test_container: STARTED
 (10/10) examples/tests/skip_conditional.py:NonBareMetal.test_container: PASS (0.01 s)
RESULTS    : PASS 5 | ERROR 0 | FAIL 0 | SKIP 5 | WARN 0 | INTERRUPT 0 | CANCEL 0
JOB HTML   : $HOME/avocado/job-results/job-2021-03-17T20.10-77d636c/results.html
JOB TIME   : 0.82 s
```

Canceling 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:

```
from avocado import Test
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_text)

    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',
```

(continues on next page)

(continued from previous page)

```
                    run('gcc -v', ignore_status=True).stderr_text)
```

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

```
$ avocado run examples/tests/cancel_test.py
JOB ID     : 39c1f120830b9769b42f5f70b6b7bad0b1b1f09f
JOB LOG    : $HOME/avocado/job-results/job-2017-03-10T16.22-39c1f12/job.log
    (1/2) /tmp/cancel_test.py:CancelTest.test_iperf: STARTED
    (1/2) /tmp/cancel_test.py:CancelTest.test_iperf: CANCEL: iperf is not installed␣
↪or wrong version (2.76 s)
    (2/2) /tmp/cancel_test.py:CancelTest.test_gcc: STARTED
    (2/2) /tmp/cancel_test.py:CancelTest.test_gcc: PASS (1.59 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.

Declaring test as NOT-INSTRUMENTED

In order to say *this class is not an Avocado instrumented* test, one can use `:avocado: disable` directive. The result is that this class itself is not discovered as an instrumented test, but children classes might inherit it's `test*` methods (useful for base-classes):

```
from avocado import Test

class BaseClass(Test):
```

(continues on next page)

(continued from previous page)

```
    """
    :avocado: disable
    """
    def test_shared(self):
        pass

class SpecificTests(BaseClass):
    def test_specific(self):
        pass
```

Results in:

```
$ avocado list --loader test.py
INSTRUMENTED test.py:SpecificTests.test_specific
INSTRUMENTED test.py:SpecificTests.test_shared
```

The `test.py:BaseBase.test` is not discovered due the tag while the `test.py:SpecificTests.test_shared` is inherited from the base-class.

Declaring test as INSTRUMENTED

The `:avocado: enable` tag might be useful when you want to override that this is an *INSTRUMENTED* test, even though it is not inherited from `avocado.Test` class and/or when you want to only limit the `test*` methods discovery to the current class:

```
from avocado import Test

class NotInheritedFromTest:
    """
    :avocado: enable
    """
    def test(self):
        pass

class BaseClass(Test):
    """
    :avocado: disable
    """
    def test_shared(self):
        pass

class SpecificTests(BaseClass):
    """
    :avocado: enable
    """
    def test_specific(self):
        pass
```

Results in:

```
$ avocado list --loader test.py
INSTRUMENTED test.py:NotInheritedFromTest.test
INSTRUMENTED test.py:SpecificTests.test_specific
```

The `test.py:NotInheritedFromTest.test` will not really work as it lacks several required methods, but still is discovered as an *INSTRUMENTED* test due to `enable` tag and the `SpecificTests` only looks at it's

`test*` methods, ignoring the inheritance, therefore the `test.py:SpecificTests.test_shared` will not be discovered.

(Deprecated) enabling recursive discovery

The `:avocado: recursive` tag was used to enable recursive discovery, but nowadays this is the default. By using this tag one explicitly sets the class as *INSTRUMENTED*, therefore inheritance from *avocado.Test* is not required.

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,`.

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

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 across 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 Tests

Avocado exports some information, including test parameters, as environment variables to the running test.

While these variables are available to all tests, they are usually more interesting to SIMPLE tests. The reason is that SIMPLE tests can not make direct use of Avocado API. INSTRUMENTED tests will usually have more powerful ways, to access the same information.

Here is a list of the variables that Avocado currently exports to tests:

Environment 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_WORKDIR	Work directory for the test	/var/tmp/avocado_Bjr_rd/my_test.sh
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

SIMPLE tests written in shell can use a few Avocado utilities. In your shell code, check if the libraries are available with something like:

```
AVOCADO_SHELL_EXTENSIONS_DIR=$(avocado exec-path 2>/dev/null)
```

And if available, injects that directory containing those utilities into the PATH used by the shell, making those utilities readily accessible:

```
if [ $? == 0 ]; then
  PATH=$AVOCADO_SHELL_EXTENSIONS_DIR:$PATH
fi
```

For a full list of utilities, take a look into at the directory return by `avocado exec-path` (if any). Also, the example test `examples/tests/simplewarning.sh` can serve as further inspiration.

Tip: These extensions may be available as a separate package. For RPM packages, look for the `bash` sub-package.

SIMPLE Tests Status

With SIMPLE tests, Avocado checks the exit code of the test to determine whether the test PASSed or FAILed.

If your test exits with exit code 0 but you still want to set a different test status in some conditions, Avocado can search a given regular expression in the test outputs and, based on that, set the status to WARN or SKIP.

To use that feature, you have to set the proper keys in the configuration file. For instance, to set the test status to SKIP when the test outputs a line like this: '11:08:24 Test Skipped':

```
[simpletests.output]
skip_regex = ^\d\d:\d\d:\d\d Test Skipped$
```

That configuration will make Avocado to search the Python Regular Expression on both stdout and stderr. If you want to limit the search for only one of them, there's another key for that configuration, resulting in:

```
[simpletests.output]
skip_regex = ^\d\d:\d\d:\d\d Test Skipped$
skip_location = stderr
```

The equivalent settings can be present for the WARN status. For instance, if you want to set the test status to WARN when the test outputs a line starting with string `WARNING:`, the configuration file will look like this:

```
[simpletests.output]
skip_regex = ^\d\d:\d\d:\d\d Test Skipped$
skip_location = stderr
warn_regex = ^WARNING:
warn_location = all
```

Job Cleanup

Warning: The Job Cleanup feature is legacy feature and it's works only with the legacy runner. For using legacy runner you have to use *–test-runner=runner*

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.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 precede 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.

Signal Handlers

Avocado normal operation is related to run code written by users/test-writers. It means the test code can carry its own handlers for different signals or even ignore then. Still, as the code is being executed by Avocado, we have to make sure we will finish all the subprocesses we create before ending our execution.

Signals sent to the Avocado main process will be handled as follows:

- SIGSTP/Ctrl+Z: On SIGSTP, Avocado will pause the execution of the subprocesses, while the main process will still be running, respecting the timeout timer and waiting for the subprocesses to finish. A new SIGSTP will make the subprocesses to resume the execution.
- SIGINT/Ctrl+C: This signal will be forwarded to the test process and Avocado will wait until it's finished. If the test process does not finish after receiving a SIGINT, user can send a second SIGINT (after the 2 seconds ignore period). The second SIGINT will make Avocado to send a SIGKILL to the whole subprocess tree and then complete the main process execution.
- SIGTERM: This signal will make Avocado to terminate immediately. A SIGKILL will be sent to the whole subprocess tree and the main process will exit without completing the execution. Notice that it's a best-effort attempt, meaning that in case of fork-bomb, newly created processes might still be left behind.

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. Although one can inspire in https://github.com/avocado-framework-tests where people are allowed to share their basic system tests.

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

9.3.3 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


class Plant(Test):
    """Logs parts of the test progress in an specific logging stream."""

    def test_plant_organic(self):
        progress_log = logging.getLogger("avocado.test.progress")
        rows = int(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(1)

        # 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(2)

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

Note: TODO: Improve how we show the logs on the console.

Currently Avocado will store any log information that is part of the 'avocado.*' namespaces. You just need to choose a namespace when setting up your logger.

The result is that, besides all the other log files commonly generated, as part of the *debug.log* file at the job results dir, you can get your logging information. During the test run, one could watch the progress with:

```
$ tail -f ~/avocado/job-results/latest/test-results/1-_tmp_plant.py_Plant.test_plant_
↪organic/debug.log
[stdlog] 2021-10-06 09:18:57,989 avocado.test.progress L0018 INFO | 1-Plant.test_
↪plant_organic: preparing soil on row 1
[stdlog] 2021-10-06 09:18:57,989 avocado.test.progress L0018 INFO | 1-Plant.test_
↪plant_organic: preparing soil on row 2
[stdlog] 2021-10-06 09:18:57,989 avocado.test.progress L0022 INFO | 1-Plant.test_
↪plant_organic: letting soil rest before throwing seeds
[stdlog] 2021-10-06 09:18:58,990 avocado.test.progress L0028 INFO | 1-Plant.test_
↪plant_organic: throwing seeds on row 0
[stdlog] 2021-10-06 09:18:58,991 avocado.test.progress L0028 INFO | 1-Plant.test_
↪plant_organic: throwing seeds on row 1
[stdlog] 2021-10-06 09:18:58,991 avocado.test.progress L0028 INFO | 1-Plant.test_
↪plant_organic: throwing seeds on row 2
[stdlog] 2021-10-06 09:18:58,992 avocado.test.progress L0032 INFO | 1-Plant.test_
↪plant_organic: waiting for Avocados to grow
[stdlog] 2021-10-06 09:19:00,995 avocado.test.progress L0038 INFO | 1-Plant.test_
↪plant_organic: harvesting organic avocados on row 0
[stdlog] 2021-10-06 09:19:00,995 avocado.test.progress L0038 INFO | 1-Plant.test_
↪plant_organic: harvesting organic avocados on row 1
[stdlog] 2021-10-06 09:19:00,996 avocado.test.progress L0038 INFO | 1-Plant.test_
↪plant_organic: harvesting organic avocados on row 2
```

The very same namespace for the logger (`avocado.test.progress`), 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.

Showing custom log streams

Using –show

Alternatively, you can ask Avocado to show your logging stream, either exclusively or in addition to other builtin streams:

```
$ avocado --show app,avocado.test.progress run --test-runner='runner' -- logging_
↪streams.py
```

The outcome should be similar to:

```
JOB ID     : af786f86db530bff26cd6a92c36e99bedcdca95b
JOB LOG    : /home/user/avocado/job-results/job-2016-03-18T10.29-af786f8/job.log
 (1/1) logging_streams.py:Plant.test_plant_organic: avocado.test.progress: 1-logging_
↪streams.py:Plant.test_plant_organic: preparing soil on row 0
avocado.test.progress: 1-logging_streams.py:Plant.test_plant_organic: preparing soil␣
↪on row 1
avocado.test.progress: 1-logging_streams.py:Plant.test_plant_organic: preparing soil␣
↪on row 2
avocado.test.progress: 1-logging_streams.py:Plant.test_plant_organic: letting soil␣
↪rest before throwing seeds
avocado.test.progress: 1-logging_streams.py:Plant.test_plant_organic: throwing seeds␣
↪on row 0
avocado.test.progress: 1-logging_streams.py:Plant.test_plant_organic: throwing seeds␣
↪on row 1
avocado.test.progress: 1-logging_streams.py:Plant.test_plant_organic: throwing seeds␣
↪on row 2
avocado.test.progress: 1-logging_streams.py:Plant.test_plant_organic: waiting for␣
↪Avocados to grow
```

(continues on next page)

(continued from previous page)

```
avocado.test.progress: 1-logging_streams.py:Plant.test_plant_organic: harvesting␣
↪organic avocados on row 0
avocado.test.progress: 1-logging_streams.py:Plant.test_plant_organic: harvesting␣
↪organic avocados on row 1
avocado.test.progress: 1-logging_streams.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/user/avocado/job-results/job-2016-03-18T10.29-af786f8/html/results.
↪html
```

Using –store-logging-stream

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 --store-logging-stream=progress --test-runner=runner -- logging_streams.
↪py
```

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-logging_streams.py:Plant.test_plant_organic: preparing soil on row 0
10:36:59 INFO | 1-logging_streams.py:Plant.test_plant_organic: preparing soil on row 1
10:36:59 INFO | 1-logging_streams.py:Plant.test_plant_organic: preparing soil on row 2
10:36:59 INFO | 1-logging_streams.py:Plant.test_plant_organic: letting soil rest␣
↪before throwing seeds
10:37:01 INFO | 1-logging_streams.py:Plant.test_plant_organic: throwing seeds on row 0
10:37:01 INFO | 1-logging_streams.py:Plant.test_plant_organic: throwing seeds on row 1
10:37:01 INFO | 1-logging_streams.py:Plant.test_plant_organic: throwing seeds on row 2
10:37:01 INFO | 1-logging_streams.py:Plant.test_plant_organic: waiting for Avocados␣
↪to grow
10:37:06 INFO | 1-logging_streams.py:Plant.test_plant_organic: harvesting organic␣
↪avocados on row 0
10:37:06 INFO | 1-logging_streams.py:Plant.test_plant_organic: harvesting organic␣
↪avocados on row 1
10:37:06 INFO | 1-logging_streams.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.

Note: Keep in mind that the above example it is trying to bring to the job level, a test level message. This is a legacy feature, and you need to run with –test-runner='runner'. Since the new NRunner architecture is running the tests in a much more isolated process, Avocado keeps test's log messages inside the individual test folder at *test-results/test-id/debug.log* as mentioned before.

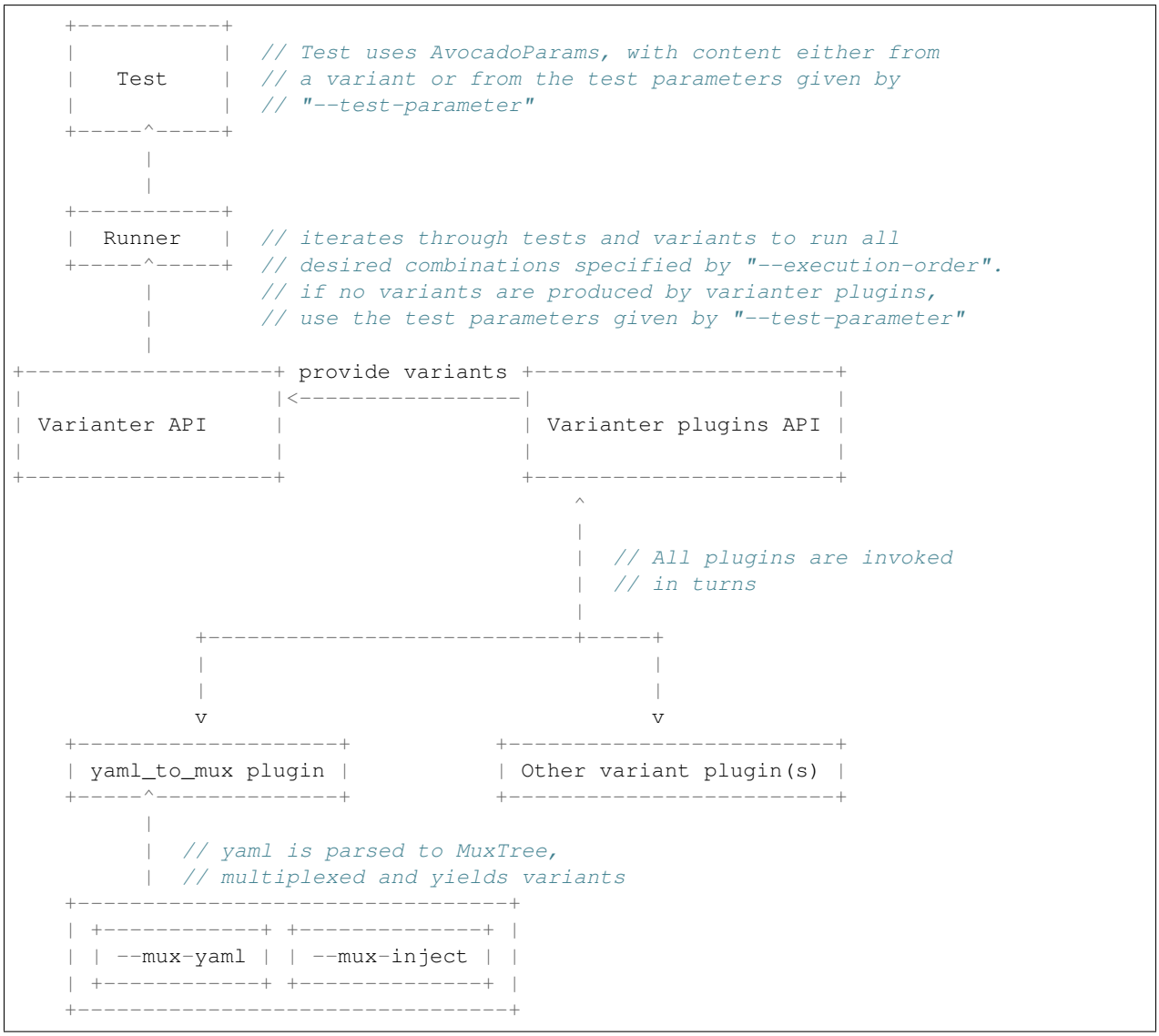
9.3.4 Test parameters

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. You can also access these parameters via the configuration dict at *run.test_parameters* namespace.

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.

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.TestID (for logging purposes) and a list of default paths (*Parameter Paths*).

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.

Parameter Paths

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 parameter paths 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 parameter paths 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 an empty *AvocadoParams*. Also, the *Variant* can also be a tuple(list, paths) or just the list of *avocado.core.tree.TreeNode* with the params.

Dumping/Loading Variants

Depending on the number of parameters, generating the Variants can be very compute intensive. As the Variants are generated as part of the Job execution, that compute intensive task will be executed by the systems under test, causing a possibly unwanted cpu load on those systems.

To avoid such situation, you can acquire the resulting JSON serialized variants file, generated out of the variants computation, and load that file on the system where the Job will be executed.

There are two ways to acquire the JSON serialized variants file:

- Using the `--json-variants-dump` option of the `avocado variants` command:

```
$ avocado variants --mux-yaml examples/yaml_to_mux/hw/hw.yaml --json-variants-
↪dump variants.json
...

$ file variants.json
variants.json: ASCII text, with very long lines, with no line terminators
```

- Getting the auto-generated JSON serialized variants file after a Avocado Job execution:

```
$ avocado run examples/tests/passtest.py --mux-yaml examples/yaml_to_mux/hw/hw.
↪yaml
...

$ file $HOME/avocado/job-results/latest/jobdata/variants.json
$HOME/avocado/job-results/latest/jobdata/variants.json: ASCII text, with very␣
↪long lines, with no line terminators
```

Once you have the `variants.json` file, you can load it on the system where the Job will take place:

```
$ avocado run examples/tests/passtest.py --json-variants-load variants.json
JOB ID     : f2022736b5b89d7f4cf62353d3fb4d7e3a06f075
JOB LOG    : $HOME/avocado/job-results/job-2018-02-09T14.39-f202273/job.log
   (1/6) examples/tests/passtest.py:PassTest.test;run-cpu-intel-disk-scsi-d340:␣
↪STARTED
   (1/6) examples/tests/passtest.py:PassTest.test;run-cpu-intel-disk-scsi-d340: PASS␣
↪(0.01 s)
   (2/6) examples/tests/passtest.py:PassTest.test;run-cpu-intel-disk-virtio-40ba:␣
↪STARTED
   (2/6) examples/tests/passtest.py:PassTest.test;run-cpu-intel-disk-virtio-40ba:␣
↪PASS (0.01 s)
   (3/6) examples/tests/passtest.py:PassTest.test;run-cpu-amd-disk-scsi-b3e2: STARTED
   (3/6) examples/tests/passtest.py:PassTest.test;run-cpu-amd-disk-scsi-b3e2: PASS (0.
↪01 s)
   (4/6) examples/tests/passtest.py:PassTest.test;run-cpu-amd-disk-virtio-9d9f:␣
↪STARTED
   (4/6) examples/tests/passtest.py:PassTest.test;run-cpu-amd-disk-virtio-9d9f: PASS␣
↪(0.01 s)
   (5/6) examples/tests/passtest.py:PassTest.test;run-cpu-arm-disk-scsi-0ceb: STARTED
   (5/6) examples/tests/passtest.py:PassTest.test;run-cpu-arm-disk-scsi-0ceb: PASS (0.
↪01 s)
   (6/6) examples/tests/passtest.py:PassTest.test;run-cpu-arm-disk-virtio-0254:␣
↪STARTED
   (6/6) examples/tests/passtest.py:PassTest.test;run-cpu-arm-disk-virtio-0254: PASS␣
↪(0.01 s)
RESULTS    : PASS 6 | ERROR 0 | FAIL 0 | SKIP 0 | WARN 0 | INTERRUPT 0 | CANCEL 0
```

(continues on next page)

(continued from previous page)

```
JOB TIME   : 0.51 s
JOB HTML   : $HOME/avocado/job-results/job-2018-02-09T14.39-f202273/results.html
```

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 examples/tests/passtest.py -m example.yaml
  |
  + parser.finish -> Varianter.__init__  // dispatcher initializes all plugins
  |
  + 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 *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
  |
  + replay.run -> Varianter.is_parsed
  |
  + replay.run  // Varianter object is replaced with the replay job's one
  |             // Varianter.ignore_new_data is set
  |
  + 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, but it invokes all *Varianter plugins* and if any of them reports variants it yields them instead of the default variant.

Test parameters

This is an Avocado core feature, that is, it's not dependent on any varianter plugin. In fact, it's only active when no Varianter plugin is used and produces a valid variant.

Avocado will use those simple parameters, and will pass them to all tests in a job execution. This is done on the command line via `--test-parameter`, or simply, `-p`. It can be given multiple times for multiple parameters.

Because Avocado parameters do not have a mechanism to define their types, test code should always consider that a parameter value is a string, and convert it to the appropriate type.

Note: Some varianter plugins would implicitly set parameters with different data types, but given that the same test can be used with different, or none, varianter plugins, it's safer if the test does an explicit check or type conversion.

Because the `avocado.core.varianter.AvocadoParams` mandates the concept of a parameter path (a legacy of the tree based Multiplexer) and these test parameters are flat, those test parameters are placed in the `/` path. This is to ensure maximum compatibility with tests that do not choose an specific parameter location.

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_varianter_yaml_to_mux.YamlToMux* which is an optional varianter plugin. Details about this plugin can be found here *Yaml_to_mux plugin*.

9.3.5 Utility Libraries

Avocado gives to you more than 40 Python utility libraries (so far), that can be found under the *avocado.utils*. You can use these libraries to avoid having to write necessary routines for your tests. 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.

See also:

If you would like a detailed API reference of this libraries, please visit the "Reference API" section on the left menu.

The following pages are the documentation for some of the Avocado utilities:

Warning: TODO: Looks like the utils libraries documentation will be mainly on docstrings, right? If so, maybe makes sense to have only documented on API reference? And any general instruction would be on module docstring. What you guys think?

avocado.utils.gdb

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, 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.

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.workdir, '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)
```

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=test`) you can see that despite declaring the variable as 0, ff is injected and printed instead.

avocado.utils.vmimage

This utility provides a API to download/cache VM images (QCOW) from the official distributions repositories.

Basic Usage

Import `vmimage` module:

```
>>> from avocado.utils import vmimage
```

Get an image, which consists in an object with the path of the downloaded/cached base image and the path of the external snapshot created out of that base image:

```
>>> image = vmimage.get()
>>> image
<Image name=Fedora version=26 arch=x86_64>
>>> image.name
'Fedora'
>>> image.path
'/tmp/Fedora-Cloud-Base-26-1.5.x86_64-d369c285.qcow2'
>>> image.get()
'/tmp/Fedora-Cloud-Base-26-1.5.x86_64-e887c743.qcow2'
>>> image.path
'/tmp/Fedora-Cloud-Base-26-1.5.x86_64-e887c743.qcow2'
>>> image.version
26
>>> image.base_image
'/tmp/Fedora-Cloud-Base-26-1.5.x86_64.qcow2'
```

If you provide more details about the image, the object is expected to reflect those details:

```
>>> image = vmimage.get(arch='aarch64')
>>> image
<Image name=FedoraSecondary version=26 arch=aarch64>
>>> image.name
'FedoraSecondary'
>>> image.path
'/tmp/Fedora-Cloud-Base-26-1.5.aarch64-07b8fbda.qcow2'

>>> image = vmimage.get(version=7)
>>> image
<Image name=CentOS version=7 arch=x86_64>
>>> image.path
'/tmp/CentOS-7-x86_64-GenericCloud-1708-dd8139c5.qcow2'
```

Notice that, unlike the `base_image` attribute, the `path` attribute will be always different in each instance, as it actually points to an external snapshot created out of the base image:

```
>>> i1 = vmimage.get()
>>> i2 = vmimage.get()
>>> i1.path == i2.path
False
```

Custom Image Provider

If you need your own Image Provider, you can extend the `vmimage.IMAGE_PROVIDERS` list, including your provider class. For instance, using the `vmimage` utility in an Avocado test, we could add our own provider with:

```
from avocado import Test

from avocado.utils import vmimage

class MyProvider(vmimage.ImageProviderBase):

    name = 'MyDistro'

    def __init__(self, version='[0-9]+', build='[0-9]+.[0-9]+',
                 arch=os.uname()[4]):
```

(continues on next page)

(continued from previous page)

```
        """
        :params version: The regular expression that represents
                         your distro version numbering.
        :params build: The regular expression that represents
                       your build version numbering.
        :params arch: The default architecture to look images for.
        """
        super(MyProvider, self).__init__(version, build, arch)

        # The URL which contains a list of the distro versions
        self.url_versions = 'https://dl.fedoraproject.org/pub/fedora/linux/releases/'

        # The URL which contains a list of distro images
        self.url_images = self.url_versions + '{version}/CloudImages/{arch}/images/'

        # The images naming pattern
        self.image_pattern = 'Fedora-Cloud-Base-{version}-{build}.{arch}.qcow2$'

class MyTest(Test):

    def setUp(self):
        vmimage.IMAGE_PROVIDERS.add(MyProvider)
        image = vmimage.get('MyDistro')
        ...

    def test(self):
        ...
```

Supported images

The vmimage library has no hardcoded limitations of versions or architectures that can be supported. You can use it as you wish. This is the list of images that we tested and they work with vmimage:

Provider	Version	Architecture
centos	8	aarch64
centos	8	ppc64le
centos	8	x86_64
centos	7	x86_64
cirros	0.5.2	arm
cirros	0.5.2	aarch64
cirros	0.5.2	i386
cirros	0.5.2	ppc64
cirros	0.5.2	ppc64le
cirros	0.5.2	powerpc
cirros	0.5.2	x86_64
cirros	0.4.0	arm
cirros	0.4.0	aarch64
cirros	0.4.0	i386
cirros	0.4.0	ppc64
cirros	0.4.0	ppc64le
cirros	0.4.0	powerpc
cirros	0.4.0	x86_64

Continued on next page

Table 1 – continued from previous page

Provider	Version	Architecture
debian	9.13.28-20211002	arm64
debian	9.13.28-20211002	amd64
debian	10.11.0	arm64
debian	10.11.0	amd64
fedora	33	aarch64
fedora	33	ppc64le
fedora	33	s390x
fedora	33	x86_64
fedora	34	aarch64
fedora	34	ppc64le
fedora	34	s390x
fedora	34	x86_64
ubuntu	18.04	aarch64
ubuntu	18.04	ppc64el
ubuntu	18.04	s390x
ubuntu	18.04	x86_64
ubuntu	20.10	aarch64
ubuntu	20.10	ppc64el
ubuntu	20.10	s390x
ubuntu	20.10	x86_64
ubuntu	21.04	aarch64
ubuntu	21.04	ppc64el
ubuntu	21.04	s390x
ubuntu	21.04	x86_64
opensuse	15.1	aarch64
opensuse	15.1	x86_64
opensuse	15.2	x86_64

9.3.6 Subclassing Avocado

Subclassing Avocado Test class to extend its features is quite straight forward and it might constitute a very useful resource to have some shared/recurrent code hosted in your project repository.

In this section we propose an project organization that will allow you to create and install your so called sub-framework.

Let's use, as an example, a project called Apricot Framework. Here's the proposed filesystem structure:

```
~/git/apricot (master)$ tree
.
├── apricot
│   ├── __init__.py
│   └── test.py
├── README.rst
├── setup.py
├── tests
│   └── test_example.py
└── VERSION
```

- `setup.py`: In the `setup.py` it is important to specify the `avocado-framework` package as a dependency:

```
from setuptools import setup, find_packages

setup(name='apricot',
      description='Apricot - Avocado SubFramework',
      version=open("VERSION", "r").read().strip(),
      author='Apricot Developers',
      author_email='apricot-devel@example.com',
      packages=['apricot'],
      include_package_data=True,
      install_requires=['avocado-framework']
      )
```

- `VERSION`: Version your project as you wish:

```
1.0
```

- `apricot/__init__.py`: Make your new test class available in your module root:

```
__all__ = ['ApricotTest']

from apricot.test import ApricotTest
```

- `apricot/test.py`: Here you will be basically extending the Avocado Test class with your own methods and routines:

```
from avocado import Test

class ApricotTest(Test):
    def setUp(self):
        self.log.info("setUp() executed from Apricot")

    def some_useful_method(self):
        return True
```

- `tests/test_example.py`: And this is how your test will look like:

```
from apricot import ApricotTest

class MyTest(ApricotTest):
    def test(self):
        self.assertTrue(self.some_useful_method())
```

To (non-intrusively) install your module, use:

```
~/git/apricot (master)$ python setup.py develop --user
running develop
running egg_info
writing requirements to apricot.egg-info/requires.txt
writing apricot.egg-info/PKG-INFO
writing top-level names to apricot.egg-info/top_level.txt
writing dependency_links to apricot.egg-info/dependency_links.txt
reading manifest file 'apricot.egg-info/SOURCES.txt'
writing manifest file 'apricot.egg-info/SOURCES.txt'
running build_ext
Creating /home/user/.local/lib/python2.7/site-packages/apricot.egg-link (link to .)
apricot 1.0 is already the active version in easy-install.pth
```

(continues on next page)

(continued from previous page)

```
Installed /home/user/git/apricot
Processing dependencies for apricot==1.0
Searching for avocado-framework==55.0
Best match: avocado-framework 55.0
avocado-framework 55.0 is already the active version in easy-install.pth

Using /home/user/git/avocado
Using /usr/lib/python2.7/site-packages
Searching for six==1.10.0
Best match: six 1.10.0
Adding six 1.10.0 to easy-install.pth file

Using /usr/lib/python2.7/site-packages
Searching for pbr==3.1.1
Best match: pbr 3.1.1
Adding pbr 3.1.1 to easy-install.pth file
Installing pbr script to /home/user/.local/bin

Using /usr/lib/python2.7/site-packages
Finished processing dependencies for apricot==1.0
```

And to run your test:

```
~/git/apricot$ avocado run tests/test_example.py
JOB ID     : 02c663eb77e0ae6ce67462a398da6972791793bf
JOB LOG    : $HOME/avocado/job-results/job-2017-11-16T12.44-02c663e/job.log
    (1/1) tests/test_example.py:MyTest.test: STARTED
    (1/1) tests/test_example.py:MyTest.test: PASS (0.03 s)
RESULTS    : PASS 1 | ERROR 0 | FAIL 0 | SKIP 0 | WARN 0 | INTERRUPT 0 | CANCEL 0
JOB TIME   : 0.95 s
JOB HTML   : $HOME/avocado/job-results/job-2017-11-16T12.44-02c663e/results.html
```

9.3.7 Integrating Avocado

Coverage.py

Testing software is important, but knowing the effectiveness of the tests, like which parts are being exercised by the tests, may help develop new tests.

Coverage.py is a tool designed for measuring code coverage of Python programs. It runs monitoring the program's source, taking notes of which parts of the code have been executed.

It is possible to use Coverage.py while running Avocado Instrumented tests. As Avocado spawn sub-processes to run the tests, the *concurrency* parameter should be set to *multiprocessing*.

To make the Coverage.py parameters visible to other processes spawned by Avocado, create the `.coveragerc` file in the project's root folder. Following is an example:

```
[run]
concurrency = multiprocessing
source = foo/bar
parallel = true
```

According to the documentation of Coverage.py, when measuring coverage in a multi-process program, setting the *parallel* parameter will keep the data separate during the measurement.

With the `.coveragerc` file set, one possible workflow to use Coverage.py to measure Avocado tests is:

```
coverage run -m avocado run tests/foo
coverage combine
coverage report
```

The first command uses Coverage.py to measure the code coverage of the Avocado tests. Then, *coverage combine* combines all measurement files to a single `.coverage` data file. The *coverage report* shows the report of the coverage measurement.

For other options related to Coverage.py, visit the software documentation.

9.4 Avocado Contributor's Guide

Useful pointers on how to participate of the Avocado community and contribute.

9.4.1 Brief introduction

First of all, we would like to thank you for taking the time to contribute! We collected here useful pointers on how to participate in the Avocado community and how to contribute.

And keep in mind that our procedures and guides are far from perfection, and need constant improvements. Feel free to propose changes to this, or any other, guide in a pull request.

Happy Hacking!

9.4.2 How can I contribute?

Note: 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 with: a) our code of conduct; b) that these contributions are your own (or approved by your employer), and c) 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.

Report a bug

If a test fails, congratulations, you have just found a bug. And If you have precise steps to reproduce, awesome! You're on your way to reporting a useful bug report.

Don't be afraid to report bugs, even if you're not sure if they're valid. The most that can happen is that we find out together that this is a feature instead!

Avocado is using Github's issue tracking system for collecting and discussing issues. If you have a possible candidate, do not hesitate, share with us by creating a new issue.

Suggest enhancements

The same is valid when suggesting a new feature or enhancements: Don't think twice, just submit the feature request using the same link. Our community will evaluate if the feature request is valid and when it might become a part of the Avocado Framework.

Contribute with code

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 the CI system and review will take place in the Pull Request as well.

For people who don't like the Github development model, there is an option to send the patches to the Mailing List, following a more traditional workflow in Open Source development communities. The patches are reviewed in the Mailing List, should you opt for that. Then a maintainer will collect the patches, integrate them on a branch, and submit it as a GitHub Pull Request. This process ensures that every contributed patch goes through the CI jobs before being considered suitable for inclusion.

Remember that we do have a small "Feature Freeze" period right before the release day (usually no longer than one week). It means that during this time, no new feature can be merged into the master branch.

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
```

See also:

Please, read our Commit Style Guide on Style Guides section manual.

- 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 system. 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
```

Code Review

Every single Pull Request in Avocado has to be reviewed by at least one other developer. All members of the core team have permission to merge a Pull Request, but some conditions have to be fulfilled before merging the code:

- Pull Request has to pass the CI tests.
- One 'Approved' code review should be given.
- No explicit disapproval should be present.

Pull Requests failing in CI will not be merged, and reviews won't be given to them until all the problems are sorted out. In case of a weird failure, or false-negative (eg. due to too many commits in a single PR), please reach the developers by @name/email/irc or other means.

While reviewing the code, one should:

- Verify that the code is sound and clean.
- Run the highest level of selftests per each new commit in the merge. The `contrib/scripts/avocado-check-pr.sh` contrib script should simplify this step.
- Verify that code works to its purpose.
- Make sure the commits organization is proper (i.e. code is well organized in atomic commits, there's no extra/unwanted commits, ...).
- Provide an in-line feedback with explicit questions and/or requests of improvements.
- Provide a general feedback in the review message, being explicit about what's expected for the next Pull Request version, if that's the case.

When the Pull Request is approved, the reviewer will merge the code or wait for someone with merge permission to merge it.

Using `avocado-check-pr.sh`

The `contrib/scripts/avocado-check-pr.sh` script is here to simplify the per-commit-check. You can simply prepare the merge and initiate `AVOCADO_CHECK_LEVEL=99 contrib/scripts/avocado-check-pr.sh` to run all checks per each commit between your branch and the same branch on the `origin/master` (you can specify different remote origin).

Use `./contrib/scripts/avocado-check-pr.sh -h` to learn more about the options. We can recommend the following command:

```
$ AVOCADO_PARALLEL_CHECK=yes AVOCADO_CHECK_LEVEL=99
$ ./contrib/scripts/avocado-check-pr.sh -i -v
```

And due to PARALLEL false-negatives running in a second terminal to re-check potential failures:

```
$$ while :; do read AAA; python -m unittest $AAA; done
```

Note: Before first use you might need to create `~/.config/github_checker.ini` and fill github user/token entries (while on it you can also specify some defaults)

Share your tests

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.

Documentation

Warning: TODO: Create how to contribute with documentation.

9.4.3 Development environment

Attention: TODO: This section needs attention! Please, help us contributing to this document.

Warning: TODO: Needs improvement here. i.e: virtualenvs, GPG, etc.

Installing dependencies

You need to install few dependencies before start coding:

```
$ sudo dnf install gcc python-devel enchant
```

Then install all the python dependencies:

```
$ make requirements-dev
```

Or if you already have pip installed, you can run directly:

```
$ pip install -r requirements-dev.txt
```

Installing in develop mode

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:

```
$ python3 setup.py develop [--user]
```

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". 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).

You can install external plugins as you wish, and/or according to the specific plugin's maintainer recommendations.

Plugins that are developed by the Avocado team, will try to follow the same Setuptools standard for distributing the packages. Because of that, as a facility, you can use `make requirements-plugins` from the main Avocado project to install requirements of the plugins and `make develop-external` to install plugins in develop mode to. You just need to set where your plugins are installed, by using the environment variable `$AVOCADO_EXTERNAL_PLUGINS_PATH`. 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
$ export AVOCADO_EXTERNAL_PLUGINS_PATH=$AVOCADO_PROJECTS_DIR
$ make develop-external
```

You should see the process and status of each directory.

9.4.4 Style guides

Commit style guide

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)
```

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, and slightly modified workflow to use the signature. Eventually, set it up in GitHub; hence, benefiting 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.

Code style guide

Warning: TODO: Add the Code Style Guide.

9.4.5 Writing an Avocado 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_UI
from avocado.core.plugin_interfaces import CLICmd


class HelloWorld(CLICmd):

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

    def run(self, config):
        LOG_UI.info(self.description)
```

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_UI* to produce the hello world output in the console.

Note: Different loggers can be used in other contexts and for different purposes. One such example is *avocado.core.output.LOG_JOB*, which can be used to output to job log files when running a job.

Registering configuration options (settings)

It is usual for a plugin to allow users to do some degree of configuration based on command-line options and/or configuration options. A plugin might change its behavior depending on a specific configuration option.

Frequently, those settings come from configuration files and, sometimes, from the command-line arguments. Like in most UNIX-like tools, command-line options will override values defined inside the configuration files.

You, as a plugin writer, don't need to handle this configuration by yourself. Avocado provides a common API that can be used by plugins in order to register options and get values.

If your plugin has options available to the users, register it using the *Settings.register_option()* method during your plugin configuration stage. The options are parsed and provided to the plugin as a config dictionary.

Let's take our Hello World example and change the message based on a "message" option:

```
from avocado.core.output import LOG_UI
from avocado.core.plugin_interfaces import CLICmd
from avocado.core.settings import settings


class HelloWorld(CLICmd):

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

    def configure(self, parser):
        settings.register_option(section='hello',
                                 key='message',
                                 key_type=str,
                                 default=self.description,
                                 help_msg="Configure the message to display")

    def run(self, config):
        msg = config.get('hello.message')
        LOG_UI.info(msg)
```

The code in the example above registers a **configuration namespace** (*hello.message*) inside the configuration file only. A namespace is a **section** (*hello*) followed by a **key** (*message*). In other words, the following entry in your configuration file is also valid and will be parsed:

```
[hello]
message = My custom message
```

As you can see in the example above, you need to set a **default** value and this value will be used if the option is not present in the configuration file. This means that you can have a very small configuration file or even an empty one.

This is a very basic example of how to configure options inside your plugin.

Adding command-line options

Now, let's say you would like to also allow this change via the command-line option of your plugin (if your plugin is a command-line plugin). You need to register in any case and use the same method to connect your **option namespace** with your command-line option.

```
from avocado.core.output import LOG_UI
from avocado.core.plugin_interfaces import CLICmd
from avocado.core.settings import settings


class HelloWorld(CLICmd):

    name = 'hello_parser'
    description = "The classical Hello World plugin example!"

    def configure(self, parser):
        parser = super(HelloWorld, self).configure(parser)

        settings.register_option(section='hello',
                                 key='message',
                                 key_type=str,
                                 default=self.description,
                                 help_msg="Configure the message to display",
                                 parser=parser,
                                 long_arg='--hello-message')

    def run(self, config):
        msg = config.get('hello.message')
        LOG_UI.info(msg)
```

Note: Keep in mind that not all options should have a "command-line" option. Try to keep the command-line as clean as possible. We use command-line only for options that constantly need to change and when editing the configuration file is not handy.

For more information about how this registration process works, visit the *Settings.register_option()* method documentation.

Registering plugins

Avocado makes use of the *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:

```
from setuptools import setup

if __name__ == '__main__':
    setup(name='avocado-hello-world-option',
          version='1.0',
          description='Avocado Hello World CLI command with config option',
          py_modules=['hello_option'],
          entry_points={
              'avocado.plugins.cli.cmd': ['hello_option = hello_option:HelloWorld'],
              }
          )
```

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

Namespace

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 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`.

Plugin config files

Plugins can extend the list of config files parsed by `Settings` objects by dropping the individual config files into `/etc/avocado/conf.d` (linux/posix-way) or they can take advantages of the Python entry point using `avocado.plugins.settings`.

1. */etc/avocado/conf.d*:

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 = caesar
dressing = caesar
```

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.

2. *avocado.plugins.settings*:

This entry-point uses `avocado.core.plugin_interfaces.Settings`-like object to extend the list of parsed files. It only accepts individual files, but you can use something like `glob.glob("*.conf")` to add all config files inside a directory.

You need to create the plugin (eg. `my_plugin/settings.py`):

```
from avocado.core.plugin_interfaces import Settings

class MyPluginSettings(Settings):
    def adjust_settings_paths(self, paths):
        paths.extend(glob.glob("/etc/my_plugin/conf.d/*.conf"))
```

And register it in your `setup.py` entry-points:

```
from setuptools import setup
...
setup(name="my-plugin",
      entry_points={
          'avocado.plugins.settings': [
              "my-plugin-settings = my_plugin.settings.MyPluginSettings",
              ],
          ...
```

Which extends the list of files to be parsed by settings object. Note this has to be executed early in the code so try to keep the required deps minimal (for example the *avocado.core.settings.settings* is not yet available).

New test type plugin example

For a new test type to be recognized and executed by Avocado's NRunner architecture, there needs to be two types of plugins and one optional:

- resolvers: they resolve references into proper test descriptions that Avocado can run.
- discoverers (optional): They are doing the same job as resolvers but without a reference. They are used when the tests can be created from different data e.g. *config files*.
- runners: these make use of the resolutions made by resolvers and actually execute the tests, reporting the results back to Avocado

The following example shows real code for a resolver and a runner for a *magic* test type. This *magic* test simply passes or fails depending on the test reference.

Resolver and Discoverer example

The resolver implementation will simply set the test type (*magic*) and transform the reference given into its **url**:

```
# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 2 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
#
# See LICENSE for more details.
#
# Copyright: Red Hat Inc. 2020
# Authors: Cleber Rosa <crosa@redhat.com>

"""
Test resolver for magic test words
"""

from avocado.core.nrunner import Runnable
from avocado.core.plugin_interfaces import Discoverer, Resolver
from avocado.core.resolver import (ReferenceResolution,
                                   ReferenceResolutionResult)
```

(continues on next page)

(continued from previous page)

```
VALID_MAGIC_WORDS = ['pass', 'fail']


class MagicResolver(Resolver):

    name = 'magic'
    description = 'Test resolver for magic words'

    @staticmethod
    def resolve(reference):
        if reference not in VALID_MAGIC_WORDS:
            return ReferenceResolution(
                reference,
                ReferenceResolutionResult.NOTFOUND,
                info='Word "%s" is not a valid magic word' % (reference))

        return ReferenceResolution(reference,
                                   ReferenceResolutionResult.SUCCESS,
                                   [Runnable('magic', reference)])


class MagicDiscoverer(Discoverer):

    name = 'magic-discoverer'
    description = 'Test discoverer for magic words'

    @staticmethod
    def discover():
        resolutions = []
        for reference in VALID_MAGIC_WORDS:
            resolutions.append(MagicResolver.resolve(reference))
        return resolutions
```

Runner example

The runner will receive the `Runnable` information created by the resolver plugin. Runners can be written in any language, but this implementation reuses some base Python classes.

First, *avocado.core.nrunner.BaseRunner* is used to write the runner **class**. And second, the *avocado.core.nrunner.BaseRunner* is used to create the command line application, which uses the previously implemented runner class for `magic` test types.

```
from avocado.core import nrunner
from avocado.core.runners.utils.messages import FinishedMessage, StartedMessage


class MagicRunner(nrunner.BaseRunner):
    """Runner for magic words

    When creating the Runnable, use the following attributes:

     * kind: should be 'magic';

     * uri: the magic word, either "pass" or "fail";
```

(continues on next page)

(continued from previous page)

```
     * args: not used;

     * kwargs: not used;

    Example:

       runnable = Runnable(kind='magic',
                           uri='pass')
    """

    def run(self):
        yield StartedMessage.get()
        if self.runnable.uri in ['pass', 'fail']:
            result = self.runnable.uri
        else:
            result = 'error'
        yield FinishedMessage.get(result)


class RunnerApp(nrunner.BaseRunnerApp):
    PROG_NAME = 'avocado-runner-magic'
    PROG_DESCRIPTION = 'nrunner application for magic tests'
    RUNNABLE_KINDS_CAPABLE = {'magic': MagicRunner}


def main():
    nrunner.main(RunnerApp)


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

Activating the new test type plugins

The plugins need to be registered so that Avocado knows about it. See *Registering plugins* for more information. This is the code that can be used to register these plugins:

```
from setuptools import setup

name = 'magic'
module = 'avocado_magic'
resolver_ep = '%s = %s.resolver:%s' % (name, module, 'MagicResolver')
discoverer_ep = '%s = %s.resolver:%s' % (name, module, 'MagicDiscoverer')
runner_ep = '%s = %s.runner:%s' % (name, module, 'MagicRunner')
runner_script = 'avocado-runner-%s = %s.runner:main' % (name, module)


if __name__ == '__main__':
    setup(name=name,
          version='1.0',
          description='Avocado "magic" test type',
          py_modules=[module],
          entry_points={
              'avocado.plugins.resolver': [resolver_ep],
```

(continues on next page)

(continued from previous page)

```
            'avocado.plugins.discoverer': [discoverer_ep],
            'avocado.plugins.runnable.runner': [runner_ep],
            'console_scripts': [runner_script],
            }
      )
```

With that, you need to either run `python setup.py install` or `python setup.py develop`.

Note: The last entry, registering a `console_script`, is recommended because it allows one to experiment with the runner as a command line application (`avocado-runner-magic` in this case). Also, depending on the spawner implementation used to run the tests, having a runner that can be executed as an application (and not a Python class) is a requirement.

Listing the new test type plugins

With the plugins activated, you should be able to run `avocado plugins` and find (among other output):

```
Plugins that resolve test references (resolver):
...
magic                Test resolver for magic words
...
```

Resolving magic tests

Resolving the "pass" and "fail" references that the magic plugin knows about can be seen by running `avocado list pass fail`:

```
magic pass
magic fail
```

And you may get more insight into the resolution results, by adding a verbose parameter and another reference. Try running `avocado -V list pass fail something-else`:

```
Type  Test Tag(s)
magic pass
magic fail

Resolver             Reference      Info
avocado-instrumented pass           File "pass" does not end with ".py"
exec-test            pass           File "pass" does not exist or is not a executable␣
↪file
golang               pass
avocado-instrumented fail           File "fail" does not end with ".py"
exec-test            fail           File "fail" does not exist or is not a executable␣
↪file
golang               fail
avocado-instrumented something-else File "something-else" does not end with ".py"
exec-test            something-else File "something-else" does not exist or is not a␣
↪executable file
golang               something-else
magic                something-else Word "something-else" is not a valid magic word
```

(continues on next page)

(continued from previous page)

```
python-unittest       something-else File "something-else" does not end with ".py"
robot                 something-else File "something-else" does not end with ".robot"
tap                   something-else File "something-else" does not exist or is not a␣
↪executable file

TEST TYPES SUMMARY
==================
magic: 2
```

It's worth realizing that magic (and other plugins) were asked to resolve the `something-else` reference, but couldn't:

```
Resolver              Reference      Info
...
magic                 something-else Word "something-else" is not a valid magic word
...
```

Running magic tests

The common way of running Avocado tests is to run them through `avocado run`. In this case, we're discussing tests for the "nrunner" architecture, so the common way of running these "magic" tests is through a command starting with `avocado run --test-runner=nrunner`.

To run both the `pass` and `fail` magic tests, you'd run `avocado run -- pass fail`:

```
$ avocado run -- pass fail
JOB ID     : 86fd45f8c1f2fe766c252eefbcac2704c2106db9
JOB LOG    : $HOME/avocado/job-results/job-2021-02-05T12.43-86fd45f/job.log
 (1/2) pass: STARTED
 (1/2) pass: PASS (0.00 s)
 (2/2) fail: STARTED
 (2/2) fail: FAIL (0.00 s)
RESULTS    : PASS 1 | ERROR 0 | FAIL 1 | SKIP 0 | WARN 0 | INTERRUPT 0 | CANCEL 0
JOB HTML   : $HOME/avocado/job-results/job-2021-02-05T12.43-86fd45f/results.html
JOB TIME   : 1.83 s
```

9.4.6 The "nrunner" and "legacy runner" test runner

This section details a test runner called "nrunner", also known as N(ext) Runner, and the architecture around. It compares it with the older, legacy (no longer default) test runner, simply called "runner".

At its essence, this new architecture is about making Avocado more capable and flexible, and even though it starts with a major internal paradigm change within the test runner, it will also affect users and test writers.

The *avocado.core.nrunner* module was initially responsible for most of the N(ext)Runner code, but as development continues, it's spreading around to other places in the Avocado source tree. Other components with different and seemingly unrelated names, say the "resolvers" or the "spawners", are also pretty much about the nrunner and are not used in the legacy architecture.

Motivation

There are a number of reasons for introducing a different architecture and implementation. Some of them are related to limitations found in the legacy implementation, that were found to be too hard to remove without major breakage.

Also, missing features that are deemed important would be a better fit wihin a different architecture.

For instance, these are the limitations of the Avocado legacy test runner:

- Test execution limited to the same machine, given that the communication between runner and test is a Python queue
- Test execution is limited to a single test at a time (serial execution)
- Test processes are not properly isolated and can affect the test runner (including the "UI")

And these are some features which it's believed to be more easily implemented under a different architecture and implementation:

- Remote test execution
- Different test execution isolation models provided by the test runner (process, container, virtual machine)
- Distributed execution of tests across a pool of any combination of processes, containers, virtual machines, etc.
- Parallel execution of tests
- Optimized runners for a given environment and or test type (for instance, a runner written in RUST to run tests written in RUST in an environment that already has RUST installed but not much else)
- Notification of execution results to many simultaneous "status servers"
- Disconnected test execution, so that results can be saved to a device and collected by the runner
- Simplified and automated deployment of the runner component into execution environments such as containers and virtual machines

NRunner and Legacy Runner components of Avocado

Whenever we mention the **current** architecture or implementation, we are talking about the nrunner. It includes:

- `avocado list` command
- `avocado run` command
- *`avocado.core.resolver`* module to resolve tests
- *`avocado.core.spawners`* modules to spawn tasks

Whenever we talk about legacy runner, we are talking about:

- `avocado list --loader` command
- `avocado run --test-runner=runner` command
- *`avocado.core.loader`* module to find tests

Basic Avocado usage and workflow

Avocado is described as "a set of tools and libraries to help with automated testing". The most visible aspect of Avocado is its ability to run tests, and display the results. We're talking about someone doing:

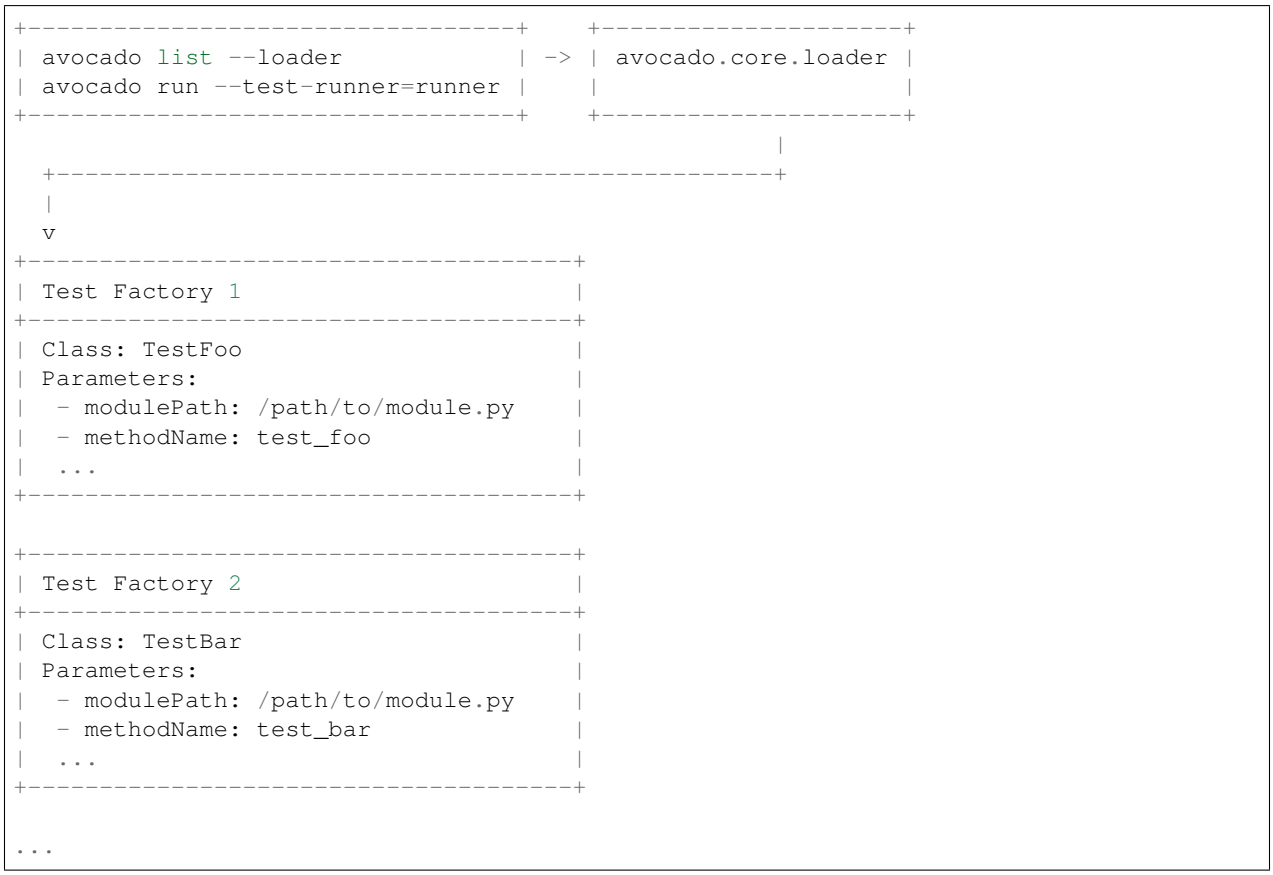
```
$ avocado run mytests.py othertests.sh
```

To be able to complete such a command, Avocado needs to find the tests, and then to execute them. Those two major steps are described next.

Finding tests

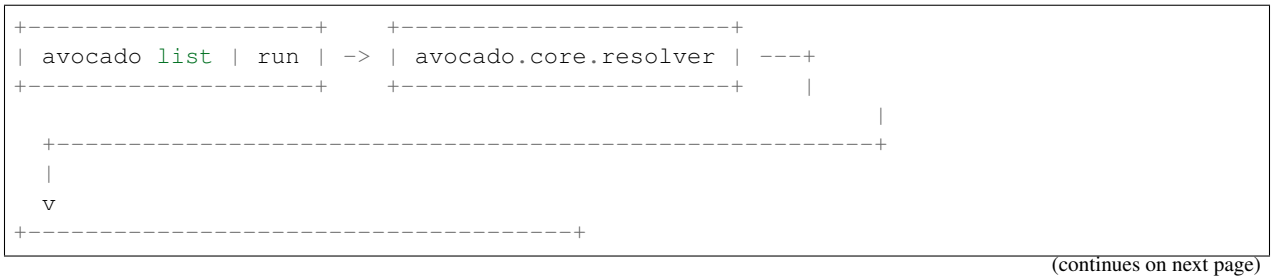
The first thing Avocado needs to do, before actually running any tests, is translating the "names" given as arguments to `avocado run` into actual tests. Even though those names will usually be file names, this is not a requirement. Avocado calls those "names" given as arguments to `avocado run` "test references", because they are references that hopefully "point to" tests.

Here we need to make a distincion between the legacy architecture, and the nrunner architecture. In the legacy Avocado test runner, this process happens by means of the *avocado.core.loader* module. The very same mechanism, is used when listing tests. This produces an internal representation of the tests, which we simply call a "factory":



Because the nrunner is now the default implementation, to distinguish between implementations and select the legacy implementation you must use: `avocado list --loader` and `avocado run--test-runner=runner`.

On the nrunner architecture, a different terminology and foundation is used. Each one of the test references given to `list` or `run` will be "resolved" into zero or more tests. Being more precise and verbose, resolver plugins will produce *avocado.core.resolver.ReferenceResolution*, which contain zero or more *avocado.core.nrunner.Runnable*, which are described in the following section. Overall, the process looks like:



(continues on next page)

(continued from previous page)

```
| ReferenceResolution #1                |
+---------------------------------------+
| Reference: /bin/true                  |
| Result: SUCCESS                       |
| +-----------------------------------+ |
| | Resolution #1 (Runnable):         | |
| |  - kind: exec-test                | |
| |  - uri: /bin/true                 | |
| +-----------------------------------+ |
+---------------------------------------+

+---------------------------------------+
| ReferenceResolution #2                |
+---------------------------------------+
| Reference: test.py                    |
| Result: SUCCESS                       |
| +-----------------------------------+ |
| | Resolution #1 (Runnable):         | |
| |  - kind: python-unittest          | |
| |  - uri: test.py:Test.test_1       | |
| +-----------------------------------+ |
| +-----------------------------------+ |
| | Resolution #2 (Runnable):         | |
| |  - kind: python-unittest          | |
| |  - uri: test.py:Test.test_2       | |
| +-----------------------------------+ |
+---------------------------------------+

...
```

Running Tests

The idea of **testing** has to do with checking the expected output of a given action. This action, within the realm of software development with automated testing, has to do with the output or outcome of a "code payload" when executed under a given controlled environment.

The legacy Avocado architecture uses the "Test Factories" described earlier to load and execute such a "code payload". Each of those test factories contain the name of a Python class to be instantiated, and a number of arguments that will be given to that class initialization.

So the primary "code payload" for every Avocado test in the legacy architecture will always be Python code that inherits from *avocado.core.test.Test*. Even when the user wants to run a standalone executable (a `SIMPLE` test in the legacy architecture terminology), that still means loading and instantiating (effectively executing) the Python class' *avocado.core.test.SimpleTest* code.

Once all the test factories are found by *avocado.core.loader*, as described in the previous section, the legacy architecture runs tests roughly following these steps:

1. Create one (and only one) queue to communicate with the test **processes**
2. For each test factory found by the loader:

a. Unpack the test factory into a test class and its parameters, that is, `test_class, parameters = test_factory`

b. Instantiate a new **process** for the test

c. Within the new process, instantiate the Python class, that is, `test = test_class(**parameters)`

d. Give the test access to queue, that is `test.set_runner_queue(queue)`

e. Monitor the queue and the test process until it finishes or needs to be terminated.

Having to describe the "Test factory" as Python classes and its parameters, besides increasing the complexity for new types of tests, severely limits or prevents some of goals for the N(ext)Runner architecture listed earlier. It should be clear that:

1. one unique queue makes communicating with multiple tests at the same time hard
2. test factories contain a Python class (**code**) that will be instantiated in the new process
3. to instantiate Python classes in other systems would require serializing them, which is error prone (AKA pickling nightmares)
4. the execution of tests depends on the previous point, so running tests in a local process is tightly coupled and hard coded into the test execution code

Now let's shift our attention to the nrunner architecture. In the nrunner architecture, a *avocado.core.nrunner.Runnable* describe a "code payload" that will be executed, but they are not executable code themselves. Because they are **data** and not **code**, they are easily serialized and transported to different environments. Running the payload described by a `Runnable` is delegated to another component.

Most often, this component is a standalone executable (see *avocado.core.spawners.common.SpawnMethod.STANDALONE_EXECUTABLE*) compatible with a specific command line interface. The most important interfaces such scripts must implement are the `runnable-run` and `task-run` interfaces.

Once all the `Runnable(s)` (within the `ReferenceResolution(s)`) are created by *avocado.core.resolver*, the `avocado run --test-runner=nrunner` implementation follows roughly the following steps:

1. Creates a status server that binds to a TCP port and waits for status messages from any number of clients
2. Creates the chosen `Spawner`, with `ProcessSpawner` being the default
3. For each *avocado.core.nrunner.Runnable* found by the resolver, turns it into a *avocado.core.nrunner.Task*, which means giving it the following extra information:

a. The status server(s) that it should report to

b. An unique identification, so that its messages to the status server can be uniquely identified

4. For each resulting *avocado.core.nrunner.Task* in the previous step:

a. Asks the spawner to spawn it

b. Asks the spawner to check if the task seems to be alive right after spawning it, to give the user early indication of possible crashes

5. Waits until all tasks have provided a `result` to the status server

If any of the concepts mentioned here were not clear, please check their full descriptions in the next section.

Concepts

Runnable

A runnable is a description of an entity that can be executed and produce some kind of result. It's a passive entity that can not execute itself and can not produce results itself.

This description of a runnable is abstract on purpose. While the most common use case for a Runnable is to describe how to execute a test, there seems to be no reason to bind that concept to a test. Other Avocado subsystems, such as `sysinfo`, could very well leverage the same concept to describe say, commands to be executed.

A Runnable's `kind`

The most important information about a runnable is the declaration of its kind. A kind should be a globally unique name across the entire Avocado community and users.

When choosing a Runnable `kind` name, it's advisable that it should be:

- Informative
- Succinct
- Unique

If a kind is thought to be generally useful to more than one user (where a user may mean a project using Avocado), it's a good idea to also have a generic name. For instance, if a Runnable is going to describe how to run native tests for the Go programming language, its `kind` should probably be `go`.

On the other hand, if a Runnable is going to be used to describe tests that behave in a very peculiar way for a specific project, it's probably a good idea to map its `kind` name to the project name. For instance, if one is describing how to run an `iotest` that is part of the `QEMU` project, it may be a good idea to name this kind `qemu-iotest`.

A Runnable's `uri`

Besides a `kind`, each runnable kind may require a different amount of information to be provided so that it can be instantiated.

Based on the accumulated experience so far, it's expected that a Runnable's `uri` is always going to be required. Think of the URI as the one piece of information that can uniquely distinguish the entity (of a given `kind`) that will be executed.

If, for instance, a given runnable describes the execution of a executable file already present in the system, it may use its path, say `/bin/true`, as its `uri` value. If a runnable describes a web service endpoint, its `uri` value may just as well be its network URI, such as `https://example.org:8080`.

Runnable examples

Possibly the simplest example for the use of a Runnable is to describe how to run a standalone executable, such as the ones available on your `/bin` directory.

As stated earlier, a runnable must declare its kind. For standalone executables, a name such as `exec` fulfills the naming suggestions given earlier.

A Runnable can be created in a number of ways. The first one is through *avocado.core.nrunner.Runnable*, a very low level (and internal) API. Still, it serves as an example:

```
>>> from avocado.core import nrunner
>>> runnable = nrunner.Runnable('exec', '/bin/true')
>>> runnable
<Runnable kind="exec" uri="/bin/true" args="()" kwargs="{}" tags="None" requirements=
↪"None">
```

The second way is through a JSON based file, which, for the lack of a better term, we're calling a (Runnable) "recipe". The recipe file itself will look like:

```
{"kind": "exec", "uri": "/bin/true"}
```

And example the code to create it:

```
>>> from avocado.core import nrunner
>>> runnable = nrunner.Runnable.from_recipe("/path/to/recipe.json")
>>> runnable
<Runnable kind="exec" uri="/bin/true" args="()" kwargs="{}" tags="None" requirements=
↪"None">>
```

The third way to create a Runnable, is even more internal. Its usage is **discouraged**, unless you are creating a tool that needs to create Runnables based on the user's input from the command line:

```
>>> from avocado.core import nrunner
>>> runnable = nrunner.Runnable.from_args({'kind': 'exec', 'uri': '/bin/true'})
>>> runnable
<Runnable kind="exec" uri="/bin/true" args="()" kwargs="{}" tags="None" requirements=
↪"None">>
```

Runner

A Runner, within the context of the nrunner architecture, is an active entity. It acts on the information that a runnable contains, and quite simply, should be able to run what the Runnable describes.

A Runner will usually be tied to a specific `kind` of Runnable. That type of relationship (Runner is capable of running kind "foo" and Runnable is of the same kind "foo") is the expected mechanism that will be employed when selecting a Runner.

A Runner can take different forms, depending on which layer one is interacting with. At the lowest layer, a Runner may be a Python class that inherits from *`avocado.core.nrunner.BaseRunner`*, and implements at least a matching constructor method, and a `run()` method that should yield dictionary(ies) as result(s).

At a different level, a runner can take the form of an executable that follows the `avocado-runner-$KIND` naming pattern and conforms to a given interface/behavior, including accepting standardized command line arguments and producing standardized output.

Tip: for a very basic example of the interface expected, refer to `selftests/functional/test_nrunner_interface.py` on the Avocado source code tree.

Runner output

A Runner should, if possible, produce status information on the progress of the execution of a Runnable. While the Runner is executing what a Runnable describes, should it produce interesting information, the Runner should attempt to forward that along its generated status.

For instance, using the `exec` Runner example, it's helpful to start producing status that the process has been created and it's running as soon as possible, even if no other output has been produced by the executable itself. These can be as simple as a sequence of:

```
{"status": "started"}
{"status": "running"}
{"status": "running"}
```

When the process is finished, the Runner may return:

```
{"status": "finished", "returncode": 0, 'stdout': b'', 'stderr': b''}
```

Tip: Besides the status of `finished`, and a return code which can be used to determine a success or failure status, a Runner may not be obliged to determine the overall PASS/FAIL outcome. Whoever called the runner may be responsible to determine its overall result, including a PASS/FAIL judgement.

Even though this level of information is expected to be generated by the Runner, whoever is calling a Runner, should be prepared to receive as little information as possible, and act accordingly. That includes receiving no information at all.

For instance, if a Runner fails to produce any information within a given amount of time, it may be considered faulty and be completely discarded. This would probably end up being represented as a `TIMED_OUT` kind of status on a higher layer (say at the "Job" layer).

Task

A task is one specific instance/occurrence of the execution of a runnable with its respective runner. They should have a unique identifier, although a task by itself won't enforce its uniqueness in a process or any other type of collection.

A task is responsible for producing and reporting status updates. This status updates are in a format similar to those received from a runner, but will add more information to them, such as its unique identifier.

A different agreggate structure should be used to keep track of the execution of tasks.

Recipe

A recipe is the serialization of the runnable information in a file. The format chosen is JSON, and that should allow both quick and easy machine handling and also manual creation of recipes when necessary.

Runners

A runner can be capable of running one or many different kinds of runnables. A runner should implement a `capabilities` command that returns, among other info, a list of runnable kinds that it can (to the best of its knowledge) run. Example:

```
python3 -m avocado.core.nrunner capabilities
{"runnables": ["noop", "exec", "exec-test", "python-unittest"],
 "commands": ["capabilities", "runnable-run", "runnable-run-recipe",
 "task-run", "task-run-recipe"]}
```

Runner scripts

The primary runner implementation is a Python module that can be run, as shown before, with the `avocado.core.nrunner` module name. Additionally it's also available as the `avocado-runner` script.

Runner Execution

While the `exec` runner given as example before will need to create an extra process to actually run the standalone executable given, that is an implementation detail of that specific runner. Other types of runners may be able to run the code the users expects it to run, while still providing feedback about it in the same process.

The runner's main method (`run()`) operates like a generator, and yields results which are dictionaries with relevant information about it.

Trying it out - standalone

It's possible to interact with the runner features by using the command line. This interface is not stable at all, and may be changed or removed in the future.

Runnables from parameters

You can run a "noop" runner with:

```
avocado-runner runnable-run -k noop
```

You can run an "exec" runner with:

```
avocado-runner runnable-run -k exec -u /bin/sleep -a 3.0
```

You can run an "exec-test" runner with:

```
avocado-runner runnable-run -k exec-test -u /bin/true
```

You can run a "python-unittest" runner with:

```
avocado-runner runnable-run -k python-unittest -u unittest.TestCase
```

Runnables from recipes

You can run a "noop" recipe with:

```
avocado-runner runnable-run-recipe examples/nrunner/recipes/runnables/noop.json
```

You can run an "exec-test" runner with:

```
avocado-runner runnable-run-recipe examples/nrunner/recipes/runnables/exec_test_sleep_
↪3.json
```

You can run a "python-unittest" runner with:

```
avocado-runner runnable-run-recipe examples/nrunner/recipes/runnables/python_unittest.
↪json
```

Writing new runner scripts

Even though you can write runner scripts in any language, if you're writing a new runner script in Python, you can benefit from the *avocado.core.nrunner.BaseRunnerApp* class and from the *avocado.core.nrunner.BaseRunner* class.

The following is a complete example of a script that could be named `avocado-runner-foo` that could act as a nrunner compatible runner for runnables with kind `foo`.

```
1 #!/usr/bin/env python3
2
3 from avocado.core import nrunner
4 from avocado.core.runners.utils.messages import FinishedMessage, StartedMessage
5
```

(continues on next page)

(continued from previous page)

```
6
7  class FooRunner(nrunner.BaseRunner):
8      def run(self):
9          yield StartedMessage.get()
10         yield FinishedMessage.get('pass')
11
12
13 class RunnerApp(nrunner.BaseRunnerApp):
14     PROG_NAME = 'avocado-runner-foo'
15     PROG_DESCRIPTION = '*EXPERIMENTAL* N(ext) Runner for tests foo'
16     RUNNABLE_KINDS_CAPABLE = {'foo': FooRunner}
17
18
19 def main():
20     nrunner.main(RunnerApp)
21
22
23 if __name__ == '__main__':
24     main()
```

Runners messages

When run as part of a job, every runner has to send information about its execution status to the Avocado job. That information is sent by messages which have different types based on the information which they are transmitting.

Avocado understands three main types of messages:

- started (required)
- running
- finished (required)

The started and finished messages are obligatory and every runner has to send those. The running messages can contain different information during runner run-time like logs, warnings, errors .etc and that information will be processed by the avocado core.

The messages are standard Python dictionaries with a specific structure. You can create it by yourself based on the table Supported message types, or you can use helper methods in *avocado.core.runners.utils.messages* which will generate them for you.

Supported message types

class avocado.core.messages.**StartMessageHandler**

Handler for started message.

It will create the test base directories and triggers the 'start_test' event.

This have to be triggered when the runner starts the test.

Parameters

- **status** – 'started'
- **time** (*float*) – start time of the test

example: {'status': 'started', 'time': 16444.819830573}

class avocado.core.messages.**FinishMessageHandler**
Handler for finished message.

It will report the test status and triggers the 'end_test' event.

This is triggered when the runner ends the test.

Parameters

- **status** – 'finished'
- **result** (*avocado.core.teststatus.STATUSES*) – test result
- **time** (*float*) – end time of the test
- **fail_reason** (*string*) – Optional parameter for brief specification, of the failed result.

example: {'status': 'finished', 'result': 'pass', 'time': 16444.819830573}

Running messages

This message can be used during the run-time and has different properties based on the information which is being transmitted.

class avocado.core.messages.**LogMessageHandler**
Handler for log message.

It will save the log to the debug.log file in the task directory.

Parameters

- **status** – 'running'
- **type** – 'log'
- **log** (*string*) – log message
- **time** (*float*) – Time stamp of the message

example: {'status': 'running', 'type': 'log', 'log': 'log message', 'time': 18405.55351474}

class avocado.core.messages.**StdoutMessageHandler**
Handler for stdout message.

It will save the stdout to the stdout and debug file in the task directory.

Parameters

- **status** – 'running'
- **type** – 'stdout'
- **log** (*bytes*) – stdout message
- **encoding** (*str*) – optional value for decoding messages
- **time** (*float*) – Time stamp of the message

example: {'status': 'running', 'type': 'stdout', 'log': 'stdout message', 'time': 18405.55351474}

class avocado.core.messages.**StderrMessageHandler**
Handler for stderr message.

It will save the stderr to the stderr and debug file in the task directory.

Parameters

- **status** – 'running'
- **type** – 'stderr'
- **log** (*bytes*) – stderr message
- **encoding** (*str*) – optional value for decoding messages
- **time** (*float*) – Time stamp of the message

example: {'status': 'running', 'type': 'stderr', 'log': 'stderr message', 'time': 18405.55351474}

class avocado.core.messages.**WhiteboardMessageHandler**

Handler for whiteboard message.

It will save the stderr to the whiteboard file in the task directory.

Parameters

- **status** – 'running'
- **type** – 'whiteboard'
- **log** (*bytes*) – whiteboard message
- **encoding** (*str*) – optional value for decoding messages
- **time** (*float*) – Time stamp of the message

example: {'status': 'running', 'type': 'whiteboard', 'log': 'whiteboard message', 'time': 18405.55351474}

class avocado.core.messages.**FileMessageHandler**

Handler for file message.

In task directory will save log into the runner specific file. When the file doesn't exist, the file will be created. If the file exist, the message data will be appended at the end.

Parameters

- **status** – 'running'
- **type** – 'file'
- **path** (*string*) – relative path to the file. The file will be created under the Task directory and the absolute path will be created as *absolute_task_directory_path/relative_file_path*.
- **log** (*bytes*) – data to be saved inside file
- **time** (*float*) – Time stamp of the message

example: {'status': 'running', 'type': 'file', 'path':'foo/runner.log', 'log': 'this will be saved inside file', 'time': 18405.55351474}

9.4.7 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 the 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 *Plugins* for more information on how to write a plugin that will activate and execute the new result format.

9.4.8 Request for Comments (RFCs)

What is a RFC?

Warning: TODO: Better describe our RFC model here.

Submiting a RFC

Warning: TODO: Better describe our RFC model here.

Previous 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]

9.4.9 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.

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`

How to release?

All the necessary steps are in JSON "testplans" to be executed with the following commands:

```
$ scripts/avocado-run-testplan -t examples/testplans/release/pre.json
$ scripts/avocado-run-testplan -t examples/testplans/release/release.json
```

Just follow the steps and have a nice release!

How to refresh Fedora/EPEL modules

This is an outline of the steps to update the Fedora/EPEL `avocado:latest` module stream when there is a new upstream release of `avocado`. This example is based on updating from 82.0 to 83.0.

Update downstream python-avocado package

1. Use pagure to create a personal fork of the downstream Fedora dist-git `python-avocado` package source repository https://src.fedoraproject.org/rpms/python-avocado if you don't already have one.
2. Clone your personal fork repository to your local workspace.
3. Checkout the `latest` branch–which is the stream branch used by the `avocado:latest` module definition. Make sure your `latest` branch is in sync with the most recent commits from the official dist-git repo you forked from.
4. Locate the official upstream commit hash and date corresponding to the upstream GitHub release tag. (eg., https://github.com/avocado-framework/avocado/releases/tag/75.1) Use those values to update the `%global commit` and `%global commit_date` lines in the downstream `python-avocado.spec` file.
5. Update the `Version:` line with the new release tag.
6. Reset the `Release:` line to `1%{?gitrel}%{?dist}`.
7. Add a new entry at the beginning of the `%changelog` section with a message similar to `Sync with upstream release 83.0.`.

8. See what changed in the upstream SPEC file since the last release. You can do this by comparing branches on GitHub (eg., https://github.com/avocado-framework/avocado/compare/82.0..83.0) and searching for `python-avocado.spec`. If there are changes beyond just the `%global commit`, `%global commit_date`, and `Version:` lines, and the `%changelog` section, make any necessary corresponding changes to the downstream SPEC file. Note: the commit hash in the upstream SPEC file *will* be different that what gets put in the downstream SPEC file since the upstream hash was added to the file before the released commit was made. Add an additional note to your `%changelog` message if there were any noteworthy changes.
9. Download the new upstream source tarball based on the updated SPEC by running:

```
spectool -g python-avocado.spec
```

10. Add the new source tarball to the dist-git lookaside cache and update your local repo by running:

```
fedpkg new-sources avocado-83.0.tar.gz
```

11. Create a Fedora source RPM from the updated SPEC file and tarball by running:

```
fedpkg --release f33 srpm
```

    It should write an SRPM file (eg., `python-avocado-83.0-1.fc33.src.rpm`) to the current directory.
12. Test build the revised package locally using `mock`. Run the build using the same Fedora release for which the SRPM was created:

```
mock -r fedora-33-x86_64 python-avocado-83.0-1.fc33.src.rpm
```

13. If the package build fails, go back and fix the SPEC file, re-create the SRPM, and retry the mock build. It is occasionally necessary to create a patch to disable specific tests or pull in some patches from upstream to get the package to build correctly. See https://src.fedoraproject.org/rpms/python-avocado/tree/69lts as an example.
14. Repeat the SRPM generation and mock build for all other supported Fedora releases, Fedora Rawhide, and the applicable EPEL (currently EPEL8).
15. When you have successful builds for all releases, `git add`, `git commit`, and `git push` your updates.

Update downstream avocado module

1. Use pagure to create a personal fork of the downstream Fedora dist-git `avocado` module source repository https://src.fedoraproject.org/modules/avocado if you don't already have one.
2. Clone your personal fork repository to your local workspace.
3. Checkout the `latest` branch–which the stream branch used for the `avocado:latest` module definition. Make sure your `latest` branch is in sync with the latest commits to the official dist-git repo you forked from.
4. If there are any new or removed `python-avocado` sub-packages, adjust the `avocado.yaml` modulemd file accordingly.
5. Test with a scratch module build for the latest supported Fedora release (f33), including the SRPM created earlier:

```
fedpkg module-scratch-build --requires platform:f33 --buildrequires platform:f33 -
↪-file avocado.yaml --srpm .../python-avocado/python-avocado-83.0-1.fc33.src.rpm
```

    You can use https://release-engineering.github.io/mbs-ui/ to monitor the build progress.

6. If the module build fails, go back and fix the modulemd file and try again. Depending on the error, it may necessary to go back and revise the package SPEC file.
7. Repeat the scratch module build for all other supported Fedora releases, Fedora Rawhide, and EPEL8 (`platform:el8`). If you're feeling confident, you can skip this step.
8. When you have successful scratch module builds for all releases, `git add`, `git commit`, `git push` your update. Note: if `avocado.yaml` didn't need modifying, it is still necessary to make a new commit since official module builds are tracked internally by their git commit hash. Recall that `git commit` has an `--allow-empty` option.

Release revised module

1. Create PRs to merge the `python-avocado` rpm and `avocado` module changes into the `latest` branches of the master dist-git repositories. If you have commit privileges to the master repositories, you could also opt to push directly.
2. After the `python-avocado` rpm and `avocado` module changes have been merged...
3. From the `latest` branch of your module repository in your local workspace, submit the module build using `fedpkg module-build`. The MBS (Module Build Service) will use stream expansion to automatically build the module for all current Fedora/EPEL releases. Again, you can use https://release-engineering.github.io/mbs-ui/ to monitor the progress of the builds.
4. If you want to test the built modules at this point, use `odcs` (On Demand Compose Service) to create a temporary compose for your Fedora release:

   ```
   odcs create module avocado:latest:3120200121201503:f636be4b
   ```

   You can then use `wget` to download the repofile from the URL referenced in the output to `/etc/yum.repos.d/` and then you'll be able to install your newly built `avocado:latest` module. Don't forget to remove the odcs repofile when you are done testing.
5. Use https://bodhi.fedoraproject.org/ to create new updates for `avocado:latest` (using options type=enhancement, severity=low, default for everything else) for each Fedora release and EPEL8 – except Rawhide which happens automatically.
6. Bodhi will push the updates to the testing repositories in a day or two. Following the push and after the Fedora mirrors have had a chance to sync, you'll be able to install the new module by including the `dnf` option `--enablerepo=updates-testing-modular` (`epel-testing-modular` for EPEL).
7. After receiving enough bodhi karma votes (three by default) or after enough days have elapsed (seven for Fedora, twelve for EPEL), bodhi will push the updated modules to the stable repositories. At that point, the updated modules will be available by default without any extra arguments to `dnf`.

9.4.10 Avocado development tips

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)
```

Note: If you are running a test with Avocado, and want to measure the duration of a method/function, make sure to enable the *debug* logging stream. Example:

```
avocado --show avocado.app.debug run examples/tests/assets.py
```

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 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).

9.4.11 Contact information

- Avocado-devel mailing list: https://www.redhat.com/mailman/listinfo/avocado-devel
- Avocado IRC channel: irc.oftc.net #avocado
- Avocado GitHub repository: https://github.com/avocado-framework/avocado/

9.5 Optional plugins

9.5.1 Golang Plugin

This optional plugin enables Avocado to list and run tests written using the Go programming language.

To install the Golang plugin from pip, use:

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

If you're running Fedora, you can install the package `golang-tests` and run any of the tests included there. You can try running the `math` or `bufio` tests like this:

```
$ GOPATH=/usr/lib/golang avocado list math
golang math:TestNaN
golang math:TestAcos
golang math:TestAcosh
golang math:TestAsin
... skip ...
```

And:

```
$ GOPATH=/usr/lib/golang avocado run math
JOB ID     : 9453e09dc5a035e465de6abd570947909d6be228
JOB LOG    : $HOME/avocado/job-results/job-2021-10-01T13.11-9453e09/job.log
 (001/417) math:TestNaN: STARTED
 (002/417) math:TestAcos: STARTED
 (001/417) math:TestNaN: PASS (0.50 s)
 (002/417) math:TestAcos: PASS (0.51 s)
 (003/417) math:TestAcosh: STARTED
 (004/417) math:TestAsin: STARTED
 (003/417) math:TestAcosh: PASS (0.50 s)
 (004/417) math:TestAsin: PASS (0.51 s)
 (005/417) math:TestAsinh: STARTED
 (006/417) math:TestAtan: STARTED
^C
RESULTS    : PASS 4 | ERROR 0 | FAIL 0 | SKIP 413 | WARN 0 | INTERRUPT 0 | CANCEL 0
JOB HTML   : $HOME/avocado/job-results/job-2021-10-01T13.11-9453e09/results.html
JOB TIME   : 2.76 s
```

Another option is to try the countavocados examples provided with avocado. Please fetch the avocado code where this example is included.

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

Also, disable the Module-aware mode, this can be done with the GO111MODULE environment variable:

```
$ go env -w GO111MODULE=off
```

Then you can `list` and `run` the countavocados tests provided with the plugin:

```
$ GOPATH=$PWD/avocado/optional_plugins/golang/tests  avocado -V list countavocados
Type   Test                                Tag(s)
golang countavocados:TestEmptyContainers
golang countavocados:TestNoContainers
golang countavocados:ExampleContainers
```

(continues on next page)

(continued from previous page)

```
Resolver             Reference     Info
avocado-instrumented countavocados File "countavocados" does not end with ".py"
exec-test            countavocados File "countavocados" does not exist or is not a␣
↪executable file

TEST TYPES SUMMARY
==================
golang: 3
```

And

```
 $ GOPATH=$PWD/avocado/optional_plugins/golang/tests  avocado run countavocados
 JOB ID     : c4284429a1ff97cd737b6e6fe1c5a83f91007317
 JOB LOG    : $HOME/avocado/job-results/job-2021-10-01T13.35-c428442/job.log
 (1/3) countavocados:TestEmptyContainers: STARTED
 (2/3) countavocados:TestNoContainers: STARTED
 (1/3) countavocados:TestEmptyContainers: PASS (0.50 s)
 (2/3) countavocados:TestNoContainers: PASS (0.50 s)
 (3/3) countavocados:ExampleContainers: STARTED
 (3/3) countavocados:ExampleContainers: PASS (0.50 s)
RESULTS    : PASS 3 | ERROR 0 | FAIL 0 | SKIP 0 | WARN 0 | INTERRUPT 0 | CANCEL 0
JOB HTML   : $HOME/avocado/job-results/job-2021-10-01T13.35-c428442/results.html
JOB TIME   : 2.12 s
```

9.5.2 Result plugins

Optional plugins providing various types of job results.

If you want to run the examples provided below, please fetch the avocado code where these examples are included.

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

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 avocado/examples/tests/sleeptest.py avocado/examples/tests/failtest.py ␣
↪avocado/examples/tests/synctest.py
JOB ID     : 480461f676fcf2a8c1c449ca1252be9521ffcceb
JOB LOG    : $HOME/avocado/job-results/job-2021-09-30T16.02-480461f/job.log
(2/3) avocado/examples/tests/failtest.py:FailTest.test: STARTED
(1/3) avocado/examples/tests/sleeptest.py:SleepTest.test: STARTED
(2/3) avocado/examples/tests/failtest.py:FailTest.test: FAIL: This test is supposed␣
↪to fail (0.04 s)
(3/3) avocado/examples/tests/synctest.py:SyncTest.test: STARTED
(1/3) avocado/examples/tests/sleeptest.py:SleepTest.test: PASS (1.01 s)
(3/3) avocado/examples/tests/synctest.py:SyncTest.test: PASS (1.17 s)
RESULTS    : PASS 2 | ERROR 0 | FAIL 1 | SKIP 0 | WARN 0 | INTERRUPT 0 | CANCEL 0
```

(continues on next page)

(continued from previous page)

```
JOB HTML    : $HOME/avocado/job-results/job-2021-09-30T16.02-480461f/results.html
JOB TIME    : 2.76 s
```

This can be disabled via `--disable-html-job-result`. 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.

Results Upload Plugin

This optional plugin is intended to upload the Avocado Job results to a dedicated sever.

To install the Result Upload plugin from pip, use:

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

Usage:

```
$ avocado run avocado/examples/tests/passtest.py --result-upload-url www@avocadologs.
↪example.com:/var/www/html
JOB ID     : f40403c7409ef998f293a7c83ee456c32cb6547a
JOB LOG    : $HOME/avocado/job-results/job-2021-09-30T22.16-f40403c/job.log
 (1/1) avocado/examples/tests/passtest.py:PassTest.test: STARTED
 (1/1) avocado/examples/tests/passtest.py:PassTest.test: PASS (0.01 s)
RESULTS    : PASS 1 | ERROR 0 | FAIL 0 | SKIP 0 | WARN 0 | INTERRUPT 0 | CANCEL 0
JOB HTML   : $HOME/avocado/job-results/job-2021-09-30T22.16-f40403c/results.html
```

Avocado logs will be available at following URL:

- ssh

  www@avocadologs.example.com:/var/www/html/job-2021-09-30T22.16-f40403c

- html (If web server is enabled)

  http://avocadologs.example.com/job-2021-09-30T22.16-f40403c/

Such links may be referred by other plugins, such as the ResultsDB plugin.

By default upload will be handled by following command

```
rsync -arz -e 'ssh -o LogLevel=error -o stricthostkeychecking=no -o␣
↪userknownhostsfile=/dev/null -o batchmode=yes -o passwordauthentication=no'
```

Optionally, you can customize uploader command, for example following command upload logs to Google storage:

```
$ avocado run avocado/examples/tests/passtest.py --result-upload-url='gs://avocadolog
↪' --result-upload-cmd='gsutil -m cp -r'
```

You can also set the ResultUpload URL and command using a config file:

```
[plugins.result_upload]
url = www@avocadologs.example.com:/var/www/htmlavocado/job-results
command='rsync -arzq'
```

And then run the Avocado command without the explicit command options. Notice that the command line options will have precedence over the configuration file.

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 avocado/examples/tests/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 avocado/examples/tests/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-2021-09-30T22.16-f40403c/
  ```

- ResultDB result (Avocado Test):

  ```
  http://avocadologs.example.com/job-2021-09-30T22.16-f40403c/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.

9.5.3 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.

For example, use the test included in the avocado code

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

To list the tests, execute:

```
$ avocado list avocado/optional_plugins/robot/tests/avocado.robot
robot $HOME/avocado/optional_plugins/robot/tests/avocado.robot:Avocado.NoSleep
robot $HOME/avocado/optional_plugins/robot/tests/avocado.robot:Avocado.Sleep
```

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

```
$ avocado run avocado/optional_plugins/robot/tests/avocado.robot
JOB ID     : 1501f546890024f2af8e26ab49ba511154bebab9
JOB LOG    : $HOME/avocado/job-results/job-2021-09-30T21.48-1501f54/job.log
 (2/2) $HOME/avocado/optional_plugins/robot/tests/avocado.robot:Avocado.Sleep: STARTED
 (1/2) $HOME/avocado/optional_plugins/robot/tests/avocado.robot:Avocado.NoSleep:␣
↪STARTED
 (1/2) $HOME/avocado/optional_plugins/robot/tests/avocado.robot:Avocado.NoSleep: PASS␣
↪(0.06 s)
 (2/2) $HOME/avocado/optional_plugins/robot/tests/avocado.robot:Avocado.Sleep: PASS␣
↪(0.07 s)
RESULTS    : PASS 2 | ERROR 0 | FAIL 0 | SKIP 0 | WARN 0 | INTERRUPT 0 | CANCEL 0
JOB HTML   : $HOME/avocado/job-results/job-2021-09-30T21.48-1501f54/results.html
JOB TIME   : 0.99 s
```

9.5.4 CIT Varianter Plugin

This plugin is an implementation of a "Combinatorial Interaction Testing with Constraints" algorithm for the Avocado varianter functionality. It generates an optimal number of variants, which in turn become different test scenarios. To learn more about this algorithm, please take a look to the papers listed below.

To run the example below, use the test included in the avocado code

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

Please refer to `avocado/examples/varianter_cit/params.cit` for an example of a input file.

Input file format

The following is the general structure of a input file:

```
PARAMETERS
Parameter_1 [Value_1,  Value_2, Value_3, Value_4]
Parameter_2 [Value_1,  Value_2, Value_3, Value_4]
Parameter_3 [Value_1,  Value_2, Value_3, Value_4]

CONSTRAINTS
Parameter_1 != Value_1 || Parameter_2 != Value_3
Parameter_3 != Value_2 || Parameter_2 != Value_4 || Parameter_1 != Value_4
```

The input file has two parts, parameters and constraints.

Parameters

- Each line represent one parameter.
- Each parameter has a name, and a list of values inside brackets.

Constraints:

- Constraints have to be in Conjunctive normal form.
- Constraints use these tree operands: `!=, OR, AND`

- `||` represents operand `OR` and new line represents operand `AND`.
- In the example, this is the logic formula

```
((P_1 != V1 OR P_2 != V_3) AND (P_3 != V_2 OR P_2 != V_4 OR P_1 != Value_4))
```

Usage

Note: the algorithm employed here can be CPU intensive. If you want more information on the progress of the combinatorial calculation, add `--debug` to a command line, such as `avocado variants --debug --cit-parameter-file $PATH`

Cit varianter plugin runs with two parameters:

- `--cit-parameter-file` with path to the input file
- `--cit-order-of-combinations` with strength of combination (default is 2)

To see the variants generated by this demo implementation, execute:

```
$ avocado variants --cit-parameter-file avocado/examples/varianter_cit/params.cit
CIT Variants (28):
Variant red-square-solid-plastic-anodic-6-4-4-2:    /
Variant green-circle-gas-leather-cathodic-7-5-4-1:    /
Variant green-triangle-liquid-leather-anodic-5-4-1-3:    /
Variant green-square-liquid-plastic-anodic-3-1-4-5:    /
Variant red-triangle-solid-leather-anodic-5-2-4-1:    /
Variant black-triangle-gas-leather-anodic-7-1-1-2:    /
Variant green-circle-solid-aluminum-cathodic-7-1-5-4:    /
Variant red-square-gas-plastic-cathodic-6-3-5-3:    /
Variant gold-triangle-solid-leather-anodic-6-5-1-4:    /
Variant gold-triangle-gas-leather-anodic-3-2-5-2:    /
Variant gold-square-gas-plastic-cathodic-5-1-1-1:    /
Variant red-circle-gas-plastic-anodic-1-1-3-3:    /
Variant red-circle-gas-aluminum-cathodic-3-3-1-5:    /
Variant black-triangle-solid-plastic-cathodic-5-5-5-5:    /
Variant gold-triangle-gas-leather-anodic-7-4-2-5:    /
Variant black-triangle-gas-aluminum-cathodic-6-1-2-1:    /
Variant gold-square-liquid-leather-cathodic-3-5-2-3:    /
Variant black-square-solid-aluminum-cathodic-7-2-4-3:    /
Variant black-circle-liquid-aluminum-anodic-1-4-5-1:    /
Variant black-triangle-gas-leather-cathodic-7-3-3-1:    /
Variant green-square-solid-aluminum-cathodic-1-3-2-2:    /
Variant gold-triangle-gas-aluminum-anodic-1-3-4-4:    /
Variant red-square-liquid-plastic-anodic-7-2-2-4:    /
Variant gold-circle-liquid-aluminum-anodic-5-5-3-2:    /
Variant red-triangle-gas-leather-anodic-1-5-1-5:    /
Variant gold-circle-liquid-aluminum-cathodic-5-3-2-4:    /
Variant black-square-solid-plastic-cathodic-3-4-3-4:    /
Variant green-circle-liquid-plastic-cathodic-6-2-3-5:    /
```

Note: The exact variants generated are not guaranteed to be the same across executions.

You can enable more verbosity, making each variant to show its content:

```
$ avocado variants --cit-parameter-file avocado/examples/varianter_cit/params.cit -c
CIT Variants (28):

Variant red-circle-solid-plastic-cathodic-6-3-3-1:    /
    /:coating  => cathodic
    /:color    => red
    /:material => plastic
    /:p10      => 1
    /:p7       => 6
    /:p8       => 3
    /:p9       => 3
    /:shape    => circle
    /:state    => solid

Variant black-circle-liquid-aluminum-anodic-6-5-1-2:    /
    /:coating  => anodic
    /:color    => black
    /:material => aluminum
    /:p10      => 2
    /:p7       => 6
    /:p8       => 5
    /:p9       => 1
    /:shape    => circle
    /:state    => liquid

... Skip 26 more variants ...
```

To execute tests with those combinations use:

```
$ avocado run avocado/examples/tests/passtest.py --cit-parameter-file avocado/
↪examples/varianter_cit/params.cit
JOB ID     : 6abd9e9f1ff9ed33a353ca8f3ef845cd4cc404a5
JOB LOG    : $HOME/avocado/job-results/job-2018-07-23T08.46-6abd9e9/job.log
 (01/25) passtest.py:PassTest.test;black-circle-gas-plastic-anodic-3-3-5-5: PASS (0.
↪04 s)
 (02/25) passtest.py:PassTest.test;gold-square-liquid-leather-anodic-3-2-1-4: PASS (0.
↪03 s)
 (03/25) passtest.py:PassTest.test;green-square-gas-plastic-cathodic-3-5-4-1: PASS (0.
↪04 s)
 (04/25) passtest.py:PassTest.test;gold-circle-solid-leather-anodic-6-4-4-2: PASS (0.
↪04 s)
 (05/25) passtest.py:PassTest.test;green-triangle-liquid-aluminum-cathodic-7-4-5-1:␣
↪PASS (0.04 s)
 (06/25) passtest.py:PassTest.test;black-circle-gas-plastic-cathodic-1-4-3-4: PASS (0.
↪04 s)
 (07/25) passtest.py:PassTest.test;red-square-gas-leather-anodic-3-4-2-3: PASS (0.04␣
↪s)
 (08/25) passtest.py:PassTest.test;gold-triangle-solid-leather-anodic-1-3-2-1: PASS␣
↪(0.04 s)
 (09/25) passtest.py:PassTest.test;green-circle-gas-plastic-cathodic-7-1-2-4: PASS (0.
↪04 s)
 (10/25) passtest.py:PassTest.test;green-triangle-gas-aluminum-cathodic-6-2-2-5: PASS␣
↪(0.04 s)
 (11/25) passtest.py:PassTest.test;black-circle-liquid-plastic-cathodic-5-5-2-2: PASS␣
↪(0.03 s)
 (12/25) passtest.py:PassTest.test;red-square-solid-aluminum-anodic-5-2-3-1: PASS (0.
↪04 s)
```

(continues on next page)

(continued from previous page)

```
 (13/25) passtest.py:PassTest.test;gold-square-solid-leather-anodic-7-5-3-5: PASS (0.
↪04 s)
 (14/25) passtest.py:PassTest.test;green-triangle-solid-leather-anodic-1-5-1-3: PASS␣
↪(0.04 s)
 (15/25) passtest.py:PassTest.test;black-circle-liquid-leather-cathodic-6-1-1-1: PASS␣
↪(0.04 s)
 (16/25) passtest.py:PassTest.test;red-triangle-liquid-plastic-anodic-6-3-3-3: PASS␣
↪(0.04 s)
 (17/25) passtest.py:PassTest.test;green-triangle-solid-plastic-cathodic-5-3-4-4:␣
↪PASS (0.04 s)
 (18/25) passtest.py:PassTest.test;red-square-liquid-aluminum-anodic-6-5-5-4: PASS (0.
↪04 s)
 (19/25) passtest.py:PassTest.test;red-square-gas-aluminum-cathodic-7-3-1-2: PASS (0.
↪04 s)
 (20/25) passtest.py:PassTest.test;red-square-liquid-aluminum-anodic-1-1-4-5: PASS (0.
↪04 s)
 (21/25) passtest.py:PassTest.test;gold-circle-gas-plastic-anodic-5-4-1-5: PASS (0.04␣
↪s)
 (22/25) passtest.py:PassTest.test;gold-circle-solid-leather-anodic-5-1-5-3: PASS (0.
↪04 s)
 (23/25) passtest.py:PassTest.test;red-circle-liquid-plastic-cathodic-1-2-5-2: PASS␣
↪(0.04 s)
 (24/25) passtest.py:PassTest.test;green-triangle-solid-aluminum-anodic-3-1-3-2: PASS␣
↪(0.04 s)
 (25/25) passtest.py:PassTest.test;black-circle-solid-aluminum-cathodic-7-2-4-3: PASS␣
↪(0.03 s)
RESULTS    : PASS 25 | ERROR 0 | FAIL 0 | SKIP 0 | WARN 0 | INTERRUPT 0 | CANCEL 0
JOB TIME   : 1.21 s
JOB HTML   : $HOME/avocado/job-results/job-2018-07-23T08.46-6abd9e9/results.html
```

Publications

The publication by Ahmed, Bestoun S., Kamal Z. Zamli, and Chee Peng Lim, entitled "Application of particle swarm optimization to uniform and variable strength covering array construction", Applied Soft Computing, 12(4), 2012, pp. 1330-1347, contains the basis for the algorithm and implementation of this feature.

Additionally, the publication by Bestoun S. Ahmed, Amador Pahim, Cleber R. Rosa Junior, D. Richard Kuhn and Miroslav Bures, entitled "Towards an Automated Unified Framework to Run Applications for Combinatorial Interaction Testing", contain a practical use case of this software.

9.5.5 PICT Varianter plugin

avocado_varianter_pict

This plugin uses a third-party tool to provide variants created by "Pair-Wise" algorithms, also known as Combinatorial Independent Testing.

Installing PICT

PICT is a free software (MIT licensed) tool that implements combinatorial testing. More information about it can be found at https://github.com/Microsoft/pict/ .

If you're building from sources, make sure you have a C++ compiler such as GCC or clang, and make. The included `Makefile` should work out of the box and give you a `pict` binary.

Then copy the `pict` binary to a location in your `$PATH`. Alternatively, you may use the plugin `--pict-binary` command line option to provide a specific location of the pict binary, but that is not as convenient as having it on your `$PATH`.

Using the PICT Varianter Plugin

To run the example below, use the test included in the avocado code

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

The following listing is a sample (simple) PICT file included at `avocado/examples/varianter_pict/params.pict`

```
arch: intel, amd
block_driver: scsi, ide, virtio
net_driver: rtl8139, e1000, virtio
guest: windows, linux
host: rhel6, rhel7, rhel8
```

To list the variants generated with the default combination order (2, that is, do a pairwise idenpendent combinatorial testing):

```
$ avocado variants --pict-parameter-file=avocado/examples/varianter_pict/params.pict
Pict Variants (11):
Variant amd-scsi-rtl8139-windows-rhel6-acff:    /run
Variant intel-scsi-virtio-linux-rhel8-26df:    /run
Variant amd-ide-virtio-windows-rhel7-3fe7:    /run
Variant amd-virtio-e1000-linux-rhel7-bf2d:    /run
Variant intel-scsi-e1000-windows-rhel8-4808:    /run
Variant intel-scsi-rtl8139-linux-rhel7-2975:    /run
Variant intel-virtio-rtl8139-windows-rhel8-6632:    /run
Variant intel-ide-rtl8139-linux-rhel6-edd2:    /run
Variant intel-virtio-virtio-windows-rhel6-e95a:    /run
Variant amd-ide-e1000-linux-rhel8-5fcc:    /run
Variant amd-ide-e1000-linux-rhel6-eb43:    /run
```

To list the variants generated with a 3-way combination:

```
$ avocado variants --pict-parameter-file=avocado/examples/varianter_pict/params.pict \
  --pict-order-of-combinations=3

Pict Variants (28):
Variant intel-ide-virtio-windows-rhel7-aea5:    /run
...skip...
Variant intel-scsi-e1000-linux-rhel7-9f61:    /run
```

To run tests, just replace the `variants` avocado command for `run`:

```
$ avocado run --pict-parameter-file=avocado/examples/varianter_pict/params.pict /bin/
↪true
```

The tests given in the command line should then be executed with all variants produced by the combinatorial algorithm implemented by PICT.

9.5.6 Multiplexer

avocado_varianter_yaml_to_mux.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_varianter_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.

9.5.7 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 `avocado-params` 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 visibly 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 Jenkins 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 Jenkins 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_varianter_yaml_to_mux.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, paths, debug)
```

Where:

- root - is the root of your params tree (compound of *TreeNode* -like nodes)
- paths - is the *Parameter Paths* 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 *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*.

9.5.8 Yaml_to_mux plugin

avocado_varianter_yaml_to_mux

This plugin utilizes the `multiplexation` mechanism to produce variants out of a yaml file. This section is example-based, if you are interested in test parameters and/or `multiplexation` overview, please take a look at *Test parameters*.

As mentioned earlier, it inherits from the *avocado_varianter_yaml_to_mux.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 below:

```
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 key is always a string and the value can be YAML value, eg. bool, list, custom type, or string. Please be aware that due to limitation None type can be provided in yaml specifically as string 'null'.

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 meant 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.

Special values

As you might have noticed, we are using mapping/dicts to define the structure of the params. To avoid surprises we disallowed the smart typing of mapping keys so:

```
on: on
```

Won't become `True: True`, but the key will be preserved as string `on: True`.

You might also want to use dict as values in your params. This is also supported but as we can't easily distinguish whether that value is a value or a node (structure), you have to either embed it into another object (list, ..) or you have to clearly state the type (yaml tag `!!python/dict`). Even then the value won't be a standard dictionary, but it'll be `collections.OrderedDict` and similarly to nodes structure all keys are preserved as strings and no smart type detection is used. Apart from that it should behave similarly as dict, only you get the values ordered by the order they appear in the file.

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'       # overridden
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 selftests/.data/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
```

(continues on next page)

(continued from previous page)

```
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.

Injecting values

Beyond the values injected by YAML files specified it's also possible inject values directly from command line to the final multiplex tree. It's done by the argument `--mux-inject`. The format of expected value is `[path:]key:node_value`.

Warning: When no path is specified to `--mux-inject` the parameter is added under tree root `/`. For example: running avocado passing `--mux-inject my_key:my_value` the parameter can be accessed calling `self.params.get('my_key')`. If the test writer wants to put the injected value in any other path location, like extending the `/run` path, it needs to be informed on avocado run call. For example: `--mux-inject /run/:my_key:my_value` makes possible to access the parameters calling `self.params.get('my_key', '/run')`

A test that gets parameters without a defined path, such as `examples/tests/multiplextest.py`:

```
os_type = self.params.get('os_type', default='linux')
```

Running it:

```
$ avocado --show=test run -- examples/tests/multiplextest.py  | grep os_type
PARAMS (key=os_type, path=*, default=linux) => 'linux'
```

Now, injecting a value, by default will put it in /, which is not in the default list of paths searched for:

```
$ avocado --show=test run --mux-inject os_type:myos -- examples/tests/multiplextest.
↪py  | grep os_type
PARAMS (key=os_type, path=*, default=linux) => 'linux'
```

A path that is searched for by default is /run. To set the value to that path use:

```
$ avocado --show=test run --mux-inject /run:os_type:myos -- examples/tests/
↪multiplextest.py  | grep os_type
PARAMS (key=os_type, path=*, default=linux) => 'myos'
```

Or, add the / to the list of paths searched for by default:

```
$ avocado --show=test run --mux-inject os_type:myos --mux-path / -- examples/tests/
↪multiplextest.py  | grep os_type
PARAMS (key=os_type, path=*, default=linux) => 'myos'
```

Warning: By default, the values are parsed for the respective data types. When not possible, it falls back to string. If you want to maintain some value as string, enclose within quotes, properly escaped, and eclose that again in quotes. For example: a value of `1` is treated as integer, a value of `1,2` is treated as list, a value of `abc` is treated as string, a value of `1,2,5-10` is treated as list of integers as `1,2,-5`. If you want to maintain this as string, provide the value as `"\"1,2,5-10\""`

9.6 Avocado Releases

9.6.1 How we release Avocado

The regular releases are released after each sprint, which usually takes 3 weeks. Regular releases are supported only until the next version is released.

We also understand that there are multiple projects currently depending on the stability of Avocado and we don't want their work to be disrupted by incompatibilities nor instabilities in new releases.

Because of that, we have **LTS releases**, that are regular releases considering the release cycle, but a new branch is created and bugfixes are backported on demand for a period of about 18 months after the release. Every year (or so) a new LTS version is released. Two subsequent LTS versions are guaranteed to have 6 months of supportability overlap.

9.6.2 Long Term Stability Releases

92.0 LTS

The Avocado team is proud to present another LTS (Long Term Stability) release: Avocado 92.0, AKA "Monsters, Inc.", is now available!

LTS Release

For more information on what a LTS release means, please read *RFC: Long Term Stability*.

Upgrading from 82.x to 92.0

Upgrading Installations

Avocado is available on a number of different repositories and installation methods. You can find the complete details *here*.

After looking at your installation options, please consider the following when planning an in-place upgrade or a deployment version bump:

- When using Python's own package management, that is, `pip`, simply choose a version lower to the *next* Avocado release to benefit from minor releases (bugfixes) in this LTS series. In short, `avocado-framework<93.0` will get you the latest release of this LTS series.
- When using RPM packages on Linux distributions that provide modular packages, select the `92lts` stream. If no modules are available, use your package manager tools (such as DNF's versionlock plugin) to pin the installation to the 92.x versions.

Porting Tests

Test API compatibility

There is only one known incompatible changes on the Test API:

- The *`avocado.Test`* used to contain a reference to its *`avocado.core.job.Job`* in its `job` attribute. Even though it was intended for internal use, the fact that the attribute was available to tests, means that tests could be relying on it. If your test uses the `job` attribute, please consider looking at the suite's configuration instead, and passing that information as a test parameter.

Utility API compatibility

The changes in the utility APIs (those that live under the `avocado.utils` namespace are too many to present porting suggestion. Please refer to the *Utility APIs* section for a comprehensive list of changes, including new features your test may be able to leverage.

General points

- With regards to logging, from now on, Avocado limits itself to `avocado.*` loggers. If your test is logging using the root logger (`logging.info(...)`), make sure you are using a proper namespace. i.e:

```
import logging
LOG = logging.getLogger('avocado.test.foo')
LOG.info('your message')
```

Important Announcement

Since the previous LTS version (82.0), Avocado has switched the default runner, from the implementation most people currently use (internally simply called `runner`), to the newer architecture and implementation called `nrunner`).

Users migrating from Avocado 82.x will be impacted by this change and should act accordingly.

To keep using the current (soon to be legacy) runner, you **must** set the `--test-runner=runner` command line option (or the equivalent `test_runner` configuration option, under section `[run]`).

Known issues are being tracked on our GitHub project page, with the `nrunner` tag, and new issue reports are appreciated.

Changes from previous LTS

Note: This is not a collection of all changes encompassing all releases from 82.0 to 92.0. This list contains changes that are relevant to users of 82.0, when evaluating an upgrade to 92.0.

When compared to the last LTS (version 82.0), the main changes introduced by this versions are:

Users / Test Writers

- The `nrunner` test runner implementation is now the default on every `avocado run` command (or equivalent Job API scripts). Since the previous release, `nrunner` supports:

1. the "fail fast" (`run --failfast`) feature.
2. The `--dry-run` feature.
3. *the varianter* feature.
4. UNIX domain sockets as the communication channel with runners (the new default).
5. a `sysinfo` runner, which will allow for sysinfo collection on any supported spawner.
6. early notification of missing runners for tasks in the requested suite.

- Yaml To Mux plugin now properly supports `None` values.
- Command line options related to results, such as `--json-job-result`, `--tap-job-result`, `--xunit-job-result` and `--html-job-result` are now "proper boolean" options (such as `--disable-json-job-result`, `--disable-xunit-job-result`, etc).
- The JSON results (`results.json`) now contain a field with the path of the test log file.
- Pre and Post (job) plugins are now respected when used with the Job API.
- Support for `avocado list` "extra information" has been restored. This is used in Avocado-VT loaders. They will be removed (again) for good after its usage is deprecated and removed in Avocado-VT.
- It's now possible to set a timeout (via the `task.timeout.running` configuration option) for nrunner tasks. Effectively this works as an execution timeout for tests run with `--test-runner=nrunner`.
- The `avocado assets` command introduces three different subcommands:

    1. `register` allows users to register their own assets with the avocado assets register command. Then, the registered asset can be used transparently with the *`avocado.core.test.Test.fetch_asset()`* by its name. This feature helps with tests that need to use assets that can not be downloaded by Avocado itself.
    2. `list` allows listing of assets based on their sizes or the number of days since they have been last accessed.
    3. `purge` allows purging of assets based on their sizes or the number of days since they have been last accessed. For more information please refer to *Managing Assets*.

- The assets plugin `fetch` command (`avocado assets fetch`) now supports:

1. fetching assets defined in a Python list in `INSTRUMENTED` tests.
2. setting a timeout for the download of assets.

- *`avocado.skipIf()`* and *`avocado.skipUnless()`* now allow the condition to be a callable, to be evaluate much later, and also gives them access to the test class. For more information, please refer to the documentation: *Advanced Conditionals*.
- The presentation of SIMPLE tests has been improved in the sense that they are now much more configurable. One can now set the `simpletests.status.failure_fields` to configure how the status line showed just after a failed test will look like, and `job.output.testlogs.logfiles` to determine the files that will be shown at the end of the job for failed tests.
- Avocado's safeloader (the system used to find Python based tests without executing them) received a major overhaul and now supports:

1. Multi-level module imports, such as `from my.base.test import Test` where a project may contain a `my/base` directory structure containing `test.py` that defines a custom `Test` class.
2. Support for following the import/inheritance hierarchy when a module contains an import for a given symbol, instead of the actual `class` definition of a symbol.
3. Considers coroutines (AKA `async def`) as valid tests, reducing the number of boiler plate code necessary for tests of `asyncio` based code.
4. Supports class definitions (containing tests or not) that use a typing hint with subscription, commonly used in generics.

- Test parameters given with `-p` are now supported when using the nrunner.
- Improved checks when users attempt to use the varianter and simple parameters (`-p`) at the same time.
- All status server URIs in the configuration are now respected for `nrunner` executions.
- The resolver plugins now have access to the job/suite configuration.
- The data directories now have fewer heuristics and are now more predictable and consistent with the configuration set.
- The root logger for Python's `logging` should no longer be impacted by Avocado's own logging initialization and clean-up (which now limits itself to `avocado.*` loggers).
- The Podman spawner (`--nrunner-spawner=podman`) will now attempt to use a Container image (`--spawner-podman-image=`) that matches the host Linux distribution. If it's not possible to detect the host distribution, the latest Fedora image will be used.
- The `exec-test` runner now accepts a configuration (`runner.exectest.exitcodes.skip`) that will determine valid exit codes to be treated as `SKIP` test results.
- The Loader based on the YAML Multiplexer has been removed. Users are advised to use Job API and multiple test suites to fulfill similar use cases.
- The GLib plugin has been removed. Users are advised to use TAP test types instead, given that GLib's GTest framework now defaults to producing TAP output.
- The paginator feature is now a boolean style option. To enable it, use `--enable-paginator`.
- The nrunner status server now has two different options regarding its URI. The first one, `--nrunner-status-server-listen` determines the URI to which a status server will listen. The second one, `--nrunner-status-server-uri` determines where the results will be sent to. This allows the status server to be on a different network location than the tasks reporting to it.
- The `avocado-software-manager` command line application now properly returns exit status for failures.

- The Podman spawner now exposes command-line options to set the container image (`--spawner-podman-image`) and the Podman binary (`--spawner-podman-bin`) used on an avocado invocation.
- The Requirements Resolver feature has been introduced, and it's available for general use. It allows users to describe requirements tests may have, and will attempt to fulfill those before the test is executed. It has support for "package" requirements, meaning operating system level packages such as RPM, DEB, etc, and "asset" requirements, allowing users to declare any asset obtainable with *`avocado.utils.asset`* to be downloaded, cached and thus be available to tests.

  This can greatly simplify the setup of the environments the tests will run on, and at the same time, not cause test errors because of the missing requirements (which will cause the test to be skipped).

  For more information please refer to the *Managing Requirements* section.
- `avocado list` got a `--json` option, which will output the list of tests in a machine-readable format.
- The minimal Python version requirement now is 3.6. Python 3.5 and earlier are not tested nor supported starting with this release.
- Because of the characteristics of the nrunner architecture, it has been decided that log content generated by tests will not be copied to the `job.log` file, but will only be available on the respective test logs on the `test-results` directory. Still, users will often need to know if tests have been started or have finished while looking at the `job.log` file. This feature has been implemented by means of the `testlogs` plugin.
- Avocado will log a warning, making it clear that it can not check the integrity of a requested asset when no hash is given. This is related to users of the *`avocado.utils.asset`* module or *`avocado.Test.fetch_asset()`* utility method.
- Avocado's cache directory defined in the configuration will now have the ultimate saying, instead of the dynamic probe for "sensible" cache directories that could end up not respecting the user's configurations.
- The man page has been thoroughly updated and put in sync with the current avocado command features and options.
- Avocado can now run from Python eggs. It's expected that official egg builds will be made available in the future. Avocado is planning to use eggs as an automatic and transparent deployment mechanism for environments such as containers and VMs.
- The `datadir.paths.logs_dir` and `datadir.paths.data_dir` are set to more consistent and predictable values, and won't rely anymore on dynamic probes for "suitable" directories.
- The Human UI plugin can now be configured to omit certain statuses from being show in a new line. This can be used, for instance, to prevent the `STARTED` lines to be shown, showing only the final test result.
- The nrunner `exec` runnable kind does not exist anymore, and its functionality was consolidated into the `exec-test`.
- Executing Python's unittest that are skipped are now always shown as having status `SKIP`, instead of the previous `CANCEL`.
- Avocado will no longer incorporate log messages coming from any logger (including the "root logger") into the test's and job's log files. Only loggers that under the `avocado.` namespace will be included. Users are encouraged to continue to follow the pattern:

```
self.log.info("message goes here")
```

  When logging from a test. When logging from somewhere else, the following pattern is advised (replace `my.namespace` accordingly):

```
import logging
LOG = logging.getLogger('avocado.my.namespace')
LOG.info('your message')
```

- Python 3.10 is now fully supported.
- The reason for fail/error/skip tests in Python unittest are now given on the various test result formats (including on the UI).

Bug Fixes

- The `run.dict_variants` setting is now properly registered in an Init plugin.
- The avocado replay command was calling pre/post plugins twice after a change delegated that responsibility to *avocado.core.job.Job.run()*.
- The *avocado.core.safeloader* now supports relative imports with names, meaning that syntax such as `from ..upper import foo` was not properly parsed.
- The TAP parser (*avocado.core.tapparser*) will not choke on unexpected content, ignoring it according to the standard.
- The assets plugin (`avocado assets` command) now returns meaningful exit code on some failures and success situations.
- The extraction of DEB packages by means of

avocado.utils.software_manager.SoftwareManager.extract_from_package() was fixed and does not depend on the `ar` utility anymore (as it now uses the *avocado.utils.ar* module).

- The `--store-logging-stream` parameter value was being incorrectly parsed as a list of characters. If a `bar` value is given, it would generate the `b.INFO`, `a.INFO`, and `r.INFO` file. The fix parses the command line arguments by treating the value as a comma-separated list (that becomes a set).
- If a job contains multiple test suites with the same name, and tests within those suites also have the same name, test results would be overwritten. Now job name uniqueness is enforced and no test results from a suite should be able to overwrite others.
- *avocado.utils.network.interfaces.NetworkInterface.is_admin_link_up()* and *avocado.utils.network.interfaces.NetworkInterface.is_operational_link_up()* now behave properly on interfaces based on bonding.
- *avocado.utils.process* utilities that use `sudo` would check for executable permissions on the binary. Many systems will have sudo with the executable bit set, but not the readable bit. This is now accounted for.
- The "external runner" feature now works properly when used outside of an avocado command-line invocation, that is, when used in a script based on the Job APIs.
- Avocado will now give an error message and exit cleanly, instead of crashing, when the resulting test suite to be executed contains no tests. That can happen, for instance, when invalid references are given along with the `--ignore-missing-references` command-line option.
- A crash when running `avocado distro --distro-def-create` has been fixed.
- Properties, that is, methods decorated with `@property` are no longer seen as tests.
- If a path to a Python unittest file contained dots, the conversion to a unittest "dotted name" would fail.
- Tests on classes that inherit from one marked with `:avocado: disable` were not being detected.

nrunner stabilization

- `avocado.core.nrunner.Runnables` created by suites will now contain the full suite configuration.
- The nrunner implementation for `exec-test` suffered from a limitation to the amount of output it could collect. It was related to the size of the `PIPE` used internally by the Python subprocess module. This limitation has now been lifted.
- nrunner will now properly translate reference names with absolute paths into Python unittest "dotted names".
- The nrunner status server can be configured with the maximum buffer size that it uses.
- The `avocado-instrumented` nrunner runner now covers all valid test statuses.
- The correct failure reason for tests executed with the nrunner is now being captured, instead of a possible exception caused by an error within the runner itself.
- The nrunner status server socket is now properly closed, which allows multiple test suites in a job to not conflict.
- The nrunner status server now properly handles the asyncio API with Python 3.6.
- The `testlog` plugin wasn't able to show the log location for tests executed via the `avocado-runner-avocado-instrumented` runner and this is now fixed.
- The `avocado-runner-avocado-instrumented` was producing duplicate log entries because of Avocado's log handler for the *avocado.core.test.Test* was previously configured to propagate the logged messages.
- The nrunner TAP runner now supports/parses large amounts of data, where it would previously crash when buffers were overrun.
- The Podman spawner will now respect the Podman binary set in the job configuration.
- The `whiteboard` file and data are now properly saved when using the nrunner.
- Some occurrences of the incorrect `AVOCADO_TEST_OUTPUT_DIR` environment variable name were renamed to the proper name (`AVOCADO_TEST_OUTPUTDIR`).
- The selection of an nrunner based runner, from its Python module name/path, has been fixed.
- The nrunner now properly sets all test status to the suite summary, making sure that errors are communicated to the end-user through, among other means, the avocado execution exit code.
- When running tests in parallel, multiple downloads of the same image (when using *avocado.utils.vmimage*) is now prevented by a better (early) locking.
- A condition in which tests running in parallel could collide over the existence of the asset's cache directory (created by other running tests) is now fixed.

Utility APIs

- A new *avocado.utils.ar* module was introduced that allows extraction of UNIX ar archive contents.
- A new *avocado.utils.sysinfo* module that powers the sysinfo feature, but is now also accessible to job/test writers.
- Times related to the duration of tasks are now limited to nanosecond precision to improve readability.
- A new module *avocado.utils.dmesg* with utilities for interacting with the kernel ring buffer messages.
- A new utility *avocado.utils.linux.is_selinux_enforcing()* allows quick check of SELinux enforcing status.

- The *avocado.utils.pmem* library introduced a number of new utility methods, adding support for daxctl operations such as `offline-memory`, `online-memory`, and `reconfigure-device`.
- *avocado.utils.software_manager.SoftwareManager.extract_from_package()* is a new method that lets users extract the content of supported package types (currently RPM and deb).
- *avocado.utils.pci* now accommodates newer slot names.
- *avocado.utils.memory* now properly handles the 16GB hugepages with both the HASH and Radix MMU (by removing the check in case Radix is used).
- The *avocado.utils.cloudinit* module will give a better error message when the system is not capable of creating ISO images, with a solution for resolution.
- Various documentation improvements for the `avocado.core.multipath` module.
- The *avocado.utils.partition* utility module now properly keeps track of loop devices and multiple mounts per device.
- A specific exception, and thus a clearer error message, is now used when a command with an empty string is given to *avocado.utils.process.run()*.

avocado.utils.ssh

- *avocado.utils.ssh.Session* now contains a *avocado.utils.ssh.Session.cleanup_master()* method and a *avocado.utils.ssh.Session.control_master* property.
- *avocado.utils.ssh* now respects the username set when copying files via `scp`.

avocado.utils.vmimage

- The *avocado.utils.vmimage* can now access both current and non-current Fedora versions (which are hosted at different locations).
- *avocado.utils.vmimage.get()* is now deprecated in favor of *avocado.utils.vmimage.Image.from_parameters()*

avocado.utils.network

- The *avocado.utils.network.interfaces* now supports setting configuration for SuSE-based systems.
- *avocado.utils.network.interfaces.NetworkInterface* can now access and present information on interfaces that do not have an IP address assigned to them.
- The *avocado.utils.network.hosts* won't consider bonding_masters anymore, a file that may exist at `/sys/class/net`, as the name of an interface.
- The *avocado.utils.network.interfaces* now support configuration files compatible with SuSE distros.
- *avocado.utils.network.interfaces.NetworkInterface.remove_link()* is a new utility method that allows one to delete a virtual interface link.
- *avocado.utils.network.hosts.Host.get_default_route_interface()* is a new utility method that allows one to get a list of default routes interfaces.

avocado.utils.cpu

- The *avocado.utils.cpu* now makes available mapping of vendor names to the data that matches in `/proc/cpuinfo` on that vendor's CPUs (*avocado.utils.cpu.VENDORS_MAP*). This allows users to have visibility about the logic used to determine the vendor's name, and overwrite it if needed.
- The *avocado.utils.cpu* library now properly handles the s390x z13 family of CPUs.

avocado.utils.distro

- avocado.utils.distro can now detect the distribution on remote machines.
- The avocado.utils.distro will now correctly return a avocado.utils.distro.UNKNOWN_DISTRO on non UNIX systems, instead of crashing.

Complete list of changes

For a complete list of changes between the last LTS release (82.1) and this release, please check out the Avocado commit changelog.

82.0 LTS

The Avocado team is proud to present another LTS (Long Term Stability) release: Avocado 82.0, AKA "Avengers: Endgame", is now available!

LTS Release

For more information on what a LTS release means, please read *RFC: Long Term Stability*.

Upgrading from 69.x to 82.0

Upgrading Installations

Avocado is available on a number of different repositories and installation methods. You can find the complete details in Installing Avocado. After looking at your installation options, please consider the following when planning an in-place upgrade or a deployment version bump:

- Avocado previously also supported Python 2, but it now supports Python 3 only. If your previous installation was based on Python 2, please consider that the upgrade path includes moving to Python 3. Dependency libraries, syntax changes, and maybe even the availability of a Python 3 interpreter are examples of things to consider.
- No issues were observed when doing an in-place upgrade from Python 2 based Avocado 69.x LTS to Python 3 based Avocado 82.0 LTS.
- When using Python's own package management, that is, `pip`, simply switch to a Python 3 based pip (`python3 -m pip` is an option) and install the `avocado-framework<83.0` package to get the latest release of the current LTS version.
- When using RPM packages, please notice that there's no package `python-avocado` anymore. Please use `python3-avocado` instead. The same is true for plugins packages, they all have the `python3-avocado-plugins` prefix.

Porting Tests (Test API compatibility)

If you're migrating from the previous LTS version, these are the changes on the Test API that most likely will affect your test:

- The `avocado.main` function isn't available anymore. If you were importing it but not really executing the test script, simply remove it. If you need to execute Avocado tests as scripts, you need to resort to the Job API instead. See `examples/jobs/passjob_with_test.py` for an example.

Porting Tests (Utility API compatibility)

The changes in the utility APIs (those that live under the `avocado.utils` namespace are too many to present porting suggestion. Please refer to the *Utility APIs* section for a comprehensive list of changes, including new features your test may be able to leverage.

Changes from previous LTS

Note: This is not a collection of all changes encompassing all releases from 69.0 to 82.0. This list contains changes that are relevant to users of 69.0, when evaluating an upgrade to 82.0.

When compared to the last LTS (version 69.3), the main changes introduced by this versions are:

Users

- Avocado can now run on systems with nothing but Python 3 (and "quasi-standard-library" module `setuptools`). This means that it won't require extra packages, and should be easier to deploy on containers, embedded systems, etc. Optional plugins may have additional requirements.
- Improved safeloader support for Python unit tests, including support for finding test classes that use multiple inheritances. As an example, Avocado's `safeloader` is now able to properly find all of its own tests (almost 1000 of them).
- Removal of old and redundant command-line options, such as `--silent` and `--show-job-log` in favor of `--show=none` and `--show=test`, respectively.
- Job result categorization support, by means of the `--job-category` option to the run command, allows a user to create an easy to find directory, within the job results directory, for a given type of executed jobs.
- The glib plugin got a configuration option for safe/unsafe operation, that is, whether it will execute binaries in an attempt to find the whole list of tests. Look for the `glib.conf` shipped with the plugin to enable the unsafe mode.
- The HTML report got upgrades as pop-up whiteboard, filtering support and resizable columns.
- When using the output check record features, duplicate files created by different tests/variants will be consolidated into unique files.
- The new `vmimage` command allows a user to list the virtual machine images downloaded by means of *`avocado.utils.vmimage`* or download new images via the avocado `vmimage get` command.
- The avocado `assets fetch` command now accepts a `--ignore-errors` option that returns exit code 0 even when some of the assets could not be fetched.
- The avocado `sysinfo` feature file will now work out of the box on pip based installations.

- The sysinfo collection now logs a much clearer message when a command is not found and thus can not have its output collected.
- Users can now select which runner plugin will be used to run tests. To select a runner on the command line, use the `--test-runner` option. Please refer to `avocado plugins` to see the runner plugins available.
- A new runner, called `nrunner`, has been introduced and has distinguishing features such as parallel test execution support either in processes or in Podman based containers.
- A massive documentation overhaul, now designed around guides to different target audiences. The "User's Guide", "Test Writer's Guide" and "Contributor's Guide" can be easily found as first level sections containing curated content for those audiences.
- It's now possible to enforce colored or non-colored output, no matter if the output is a terminal or not. The configuration item color was introduced in the `runner.output` section, and recognizes the values `auto`, `always`, or `never`.
- The jsonresult plugin added `warn` and `interrupt` fields containing counters for the tests that ended with WARN and INTERRUPTED status, respectively.
- Avocado's *`avocado.utils.software_manager`* functionality is now also made available as the `avocado-software-manager` command-line tool.
- Avocado now supports "hint files" that can tweak how the Avocado resolver will recognize tests. This is useful for projects making use of Avocado as a test runner, and it can allow complete integration with a simple configuration file in a project repository. For more information check out the *documentation*.
- The `--ignore-missing-references` option now takes no parameter. The feature it controls is not enabled unless you supply the command line option (but no on or off is required).
- A brand new command, `jobs`, enables users to, among other things, list information about previously executed jobs. A command such as `avocado jobs show` will show the latest job information.
- The remote, vm, and docker runner plugins were removed.
- The `multiplex` command, an alias to variants, has been removed.
- A new settings API that is tightly linked to the Job API. You can see all the existing configurations at runtime by running avocado config reference. To integrate Avocado to an existing project or a CI environment, a custom job with a few configurations will give you a lot of flexibility with very little need to write Python code. Some examples are available at `examples/jobs`.

Test Writers

- Python 2 support has been removed. Support for Python versions include 3.5, 3.6, 3.7 and 3.8. If you require Python 2 support, the 69.X LTS version should be used.
- A fully usable Job API, making most of Avocado's functionalities programmable and highly customizable.
- Support for multiple test suites in a Job, so that each test suite can be configured differently and independently from each other. Fulfill your use case easily (or let your imagination go wild) and define different runners, different parameters to different test suites, or run some test suites locally, while others run isolated on containers. Anything that is configurable with the new settings API should be transparently configurable in the context of a test suite (provided the test suite deals with that feature).
- A completely new implementation of the CIT Varianter plugin, now with support for constraints. Refer to *CIT Varianter Plugin* for more information.
- The new *`avocado.cancel_on()`* decorator has been added to the Test APIs, allowing you to define the conditions for a test to be considered canceled. See one example here.

- Avocado can now use tags inside Python Unittests, and not only on its own Instrumented tests.
- The tags feature (see *Categorizing tests*) now supports an extended character set, adding . and - to the allowed characters. A tag such as `:avocado: tags=machine:s390-ccw-virtio` is now valid.
- INSTRUMENTED tests using the *avocado.Test.fetch_asset()* can take advantage of plugins that will attempt to download (and cache) assets before the test execution. This should make the overall test execution more reliable, and give better test execution times as the download time will be excluded. Users can also manually execute the avocado assets command to manually fetch assets from tests.
- The *avocado.Test.fetch_asset()* method now has two new parameters: `find_only` and `cancel_on_missing`. These can be combined to cancel tests if the asset is missing after a download attempt (`find_only=False`) or only if it's present in the local system without a download having been attempted during the test (`find_only=True`). This can bring better determinism for tests that would download sizable assets, and/or allow test jobs to be executable in offline environments.
- A new test type, `TAP` has been introduced along with a new loader and resolver. With a `TAP` test, it's possible to execute a binary or script, similar to a `SIMPLE` test, and parse its Test Anything Protocol output to determine the test status.
- The decorators *avocado.skip()*, *avocado.skipIf()*, and *avocado.skipUnless()* can now be used to decorate entire classes, resulting in all its tests getting skipped if/when the given condition is satisfied.
- The "log level" of Avocado is now defined using the standard Python level names. If you have a custom configuration for this setting, you may need to adjust it.
- The yaml_to_mux varianter plugin now attempts to respect the type of the value given to `--mux-inject`. For example, 1 is treated as an integer, a value of `1,2` is treated as a list, a value of `abc` is treated as a string, and a value of `1,2,5-10` is treated as a list of integers as `1,2,-5` (as it is evaluated by `ast.literal_eval()`).
- For users of the Job API, a "dictionary-based" varianter was introduced, that allows you to describe the variations of tests in a test suite directly via a Python dictionary.
- The `avocado.utils.runtime` module has been removed.
- The test runner feature that would allow binaries to be run transparently inside GDB was removed. The reason for dropping such a feature has to do with how it limits the test runner to run one test at a time, and the use of the `avocado.utils.runtime` mechanism, also removed.
- The "standalone job" feature was removed. The alternative is to use an Avocado Job (using the Job API), with a test defined on the same file, as can be seen on the example file `examples/jobs/passjob_with_test.py` in the source tree.

Utility APIs

- Two simple utility APIs, *avocado.utils.genio.append_file()* and *avocado.utils.genio.append_one_line()* have been added.
- The new *avocado.utils.datadrainer* provides an easy way to read from and write to various input/output sources without blocking a test (by spawning a thread for that).
- The new *avocado.utils.diff_validator* can help test writers to make sure that given changes have been applied to files.
- *avocado.utils.partition* now allows `mkfs` and `mount` flags to be set.
- Users of the `avocado.utils.partition.mount()` function can now skip checking if the devices/mountpoints are mounted, which is useful for bind mounts.
- *avocado.utils.cpu.get_cpu_vendor_name()* now returns the CPU vendor name for POWER9.

- The *avocado.utils.cpu* changed how it identifies CPU vendors, architectures, and families, making those more consistent across the board.
- The names of the *avocado.utils.cpu* functions changed, from what's listed on left hand side (now deprecated) the ones on the right hand side:
- *avocado.utils.cpu.total_cpus_count()* => *avocado.utils.cpu.total_count()*
- avocado.utils.cpu._get_cpu_info() => avocado.utils.cpu._get_info()
- avocado.utils.cpu._get_cpu_status() => avocado.utils.cpu._get_status()
- *avocado.utils.cpu.get_cpu_vendor_name()* => *avocado.utils.cpu.get_vendor()*
- *avocado.utils.cpu.get_cpu_arch()* => *avocado.utils.cpu.get_arch()*
- *avocado.utils.cpu.cpu_online_list()* => *avocado.utils.cpu.online_list()*
- *avocado.utils.cpu.online_cpus_count()* => *avocado.utils.cpu.online_count()*
- *avocado.utils.cpu.get_cpuidle_state()* => *avocado.utils.cpu.get_idle_state()*
- *avocado.utils.cpu.set_cpuidle_state()* => *avocado.utils.cpu.set_idle_state()*
- *avocado.utils.cpu.set_cpufreq_governor()* => *avocado.utils.cpu.set_freq_governor()*
- *avocado.utils.cpu.get_cpufreq_governor()* => *avocado.utils.cpu.get_freq_governor()*
- Additionally, *avocado.utils.cpu.get_arch()* implementation for powerpc has been corrected to return powerpc instead of cpu family values like power8, power9.
- New *avocado.utils.cpu.get_family()* is added to get the cpu family values like power8, power9.
- The *avocado.utils.cpu.online()* and *avocado.utils.cpu.offline()* will now check the status of the CPU before attempting to apply a possibly (unnecessary) action.
- The *avocado.utils.asset* module now allows a given location, as well as a list, to be given, simplifying the most common use case.
- *avocado.utils.process.SubProcess.stop()* now supports setting a timeout.
- *avocado.utils.memory* now properly handles huge pages for the POWER platform.
- *avocado.utils.ssh* now allows password-based authentication, in addition to public key-based authentication.
- The new *avocado.utils.ssh.Session.get_raw_ssh_command()* method allows access to the generated (local) commands, which could be used for advanced use cases, such as running multiple (remote) commands in a test. See the examples/apis/utils/ssh.py for an example.
- The *avocado.utils.ssh.Session.cmd()* method now allows users to ignore the exit status of the command with the *ignore_status parameter*.
- *avocado.utils.path.usable_ro_dir()* will no longer create a directory, but will just check for its existence and the right level of access.
- *avocado.utils.archive.compress()* and *avocado.utils.archive.uncompress()* and now supports LZMA compressed files transparently.
- The *avocado.utils.vmimage* module now has providers for the CirrOS cloud images.
- The *avocado.utils.vmimage* library now allows a user to define the qemu-img binary that will be used for creating snapshot images via the *avocado.utils.vmimage.QEMU_IMG* variable.

- The *avocado.utils.vmimage* module will not try to create snapshot images when it's not needed, acting lazily in that regard. It now provides a different method for download-only operations, *avocado.utils.vmimage.Image.download()* that returns the base image location. The behavior of the *avocado.utils.vmimage.Image.get()* method is unchanged in the sense that it returns the path of a snapshot image.
- The *avocado.utils.configure_network* module introduced a number of utilities, including MTU configuration support, a method for validating network among peers, IPv6 support, etc.
- The *avocado.utils.configure_network.set_ip()* function now supports different interface types through a `interface_type` parameter, while still defaulting to Ethernet.
- *avocado.utils.configure_network.is_interface_link_up()* is a new utility function that returns, quite obviously, whether an interface link is up.
- The *avocado.utils.network* module received a complete overhaul and provides features for getting, checking, and setting network information from local and even remote hosts.
- The *avocado.utils.network.interfaces* module now supports different types of output produced by `iproute`.
- *avocado.utils.kernel* received a number of fixes and cleanups, and also new features. It's now possible to configure the kernel for multiple targets, and also set kernel configurations at configuration time without manually touching the kernel configuration files. It also introduced the *avocado.utils.kernel.KernelBuild.vmlinux* property, allowing users to access that image if it was built.
- New functions such as *avocado.utils.multipath.add_path()*, *avocado.utils.multipath.remove_path()*, *avocado.utils.multipath.get_mpath_status()* and *avocado.utils.multipath.suspend_mpath()* have been introduced to the *avocado.utils.multipath* module.
- The new *avocado.utils.pmem* module provides an interface to manage persistent memory. It allows for creating, deleting, enabling, disabling, and re-configuring both namespaces and regions depending on supported hardware. It wraps the features present on the ndctl and daxctl binaries.
- All of the *avocado.utils.gdb* APIs are now back to a working state, with many fixes related to bytes and strings, as well as buffered I/O caching fixes.

Contributors

- The Avocado configuration that is logged during a job execution is now the dictionary that is produced by the new *avocado.core.settings* module, instead of the configuration file(s) content. This is relevant because this configuration contains the result of everything that affects a job, such as defaults registered by plugins, command-line options, all in addition to the configuration file. The goal is to have more consistent behavior and increased job "replayability".

Complete list of changes

For a complete list of changes between the last LTS release (69.3) and this release, please check out the Avocado commit changelog.

69.0 LTS

The Avocado team is proud to present another LTS (Long Term Stability) release: Avocado 69.0, AKA "The King's Choice", is now available!

LTS Release

For more information on what a LTS release means, please read *RFC: Long Term Stability*.

Upgrading from 52.x to 69.0

Upgrading Installations

Avocado is available on a number of different repositories and installation methods. You can find the complete details in *Installing Avocado*. After looking at your installation options, please consider the following highlights about the changes in the Avocado installation:

- Avocado fully supports both Python 2 and 3, and both can even be installed simultaneously. When using RPM packages, if you ask to have `python-avocado` installed, it will be provided by the Python 2 based package. If you want a Python 3 based version you must use the `python3-avocado` package. The same is true for plugins, which have a `python2-avocado-plugins` or `python3-avocado-plugins` prefix.
- Avocado can now be properly installed without super user privileges. Previously one would see an error such as `could not create '/etc/avocado': Permission denied` when trying to do a source or PIP based installation.
- When installing Avocado on Python "venvs", the user's base data directory is now within the venv. If you had content outside the venv, such as results or tests directories, please make sure that you either configure your data directories on the `[datadir.paths]` section of your configuration file, or move the data over.

Porting Tests (Test API compatibility)

If you're migration from the previous LTS version, these are the changes on the Test API that most likely will affect your test.

Note: Between non-LTS releases, the Avocado Test APIs receive a lot of effort to be kept as stable as possible. When that's not possible, a deprecation strategy is applied and breakage can occur. For guaranteed stability across longer periods of time, LTS releases such as this one should be used.

- Support for default test parameters, given via the class level `default_params` dictionary has been removed. If your test contains a snippet similar to:

```
default_params = {'param1': 'value1',
                  'param2': 'value2'}
def test(self):
    value1 = self.params.get('param1')
    value2 = self.params.get('param2')
```

  It should be rewritten to look like this:

```
def test(self):
    value1 = self.params.get('param1', default='value1')
    value2 = self.params.get('param2', default='value2')
```

- Support for getting parameters using the `self.params.key` syntax has been removed. If your test contains a snippet similar to:

```
def test(self):
    value1 = self.params.key1
```

It should be rewritten to look like this:

```
def test(self):
    value1 = self.params.get('key1')
```

- Support for the `datadir` test class attribute has been removed in favor of the `get_data()` method. If your test contains a snippet similar to:

```
def test(self):
    data = os.path.join(self.datadir, 'data')
```

It should be rewritten to look like this:

```
def test(self):
    data = self.get_data('data')
```

- Support for for `srcdir` test class attribute has been removed in favor of the `workdir` attribute. If your test contains a snippet similar to:

```
def test(self):
    compiled = os.path.join(self.srcdir, 'binary')
```

It should be rewritten to look like this:

```
def test(self):
    compiled = os.path.join(self.workdir, 'binary')
```

- The `:avocado: enable` and `:avocado: recursive` tags may not be necessary anymore, given that "recursive" is now the default loader behavior. If you test contains:

```
def test(self):
    """
    :avocado: enable
    """
```

Or:

```
def test(self):
    """
    :avocado: recursive
    """
```

Consider removing the tags completely, and check if the default loader behavior is sufficient with:

```
$ avocado list your-test-file.py
```

- Support for the `skip` method has been removed from the `avocado.Test` class. If your test contains a snippet similar to:

```
def test(self):
    if not condition():
        self.skip("condition not suitable to keep test running")
```

It should be rewritten to look like this:

```
def test(self):
    if not condition():
        self.cancel("condition not suitable to keep test running")
```

Porting Tests (Utility API compatibility)

The changes in the utility APIs (those that live under the `avocado.utils` namespace are too many to present porting suggestion. Please refer to the *Utility APIs* section for a comprehensive list of changes, including new features your test may be able to leverage.

Changes from previous LTS

Note: This is not a collection of all changes encompassing all releases from 52.0 to 69.0. This list contains changes that are relevant to users of 52.0, when evaluating an upgrade to 69.0.

When compared to the last LTS (version 52.1), the main changes introduced by this versions are:

Test Writers

Test APIs

- Test writers will get better protection against mistakes when trying to overwrite *avocado.core.test.Test* "properties". Some of those were previously implemented using *avocado.utils.data_structures.LazyProperty()* which did not prevent test writers from overwriting them.
- The `avocado.Test.default_parameters` mechanism for setting default parameters on tests has been removed. This was introduced quite early in the Avocado development, and allowed users to set a dictionary at the class level with keys/values that would serve as default parameter values. The recommended approach now, is to just provide default values when calling the `self.params.get` within a test method, such as `self.params.get("key", default="default_value_for_key")`.
- The `__getattr__` interface for `self.params` has been removed. It used to allow users to use a syntax such as `self.params.key` when attempting to access the value for key `key`. The supported syntax is `self.params.get("key")` to achieve the same thing.
- The support for test data files has been improved to support more specific sources of data. For instance, when a test file used to contain more than one test, all of them shared the same `datadir` property value, thus the same directory which contained data files. Now, tests should use the newly introduced *get_data()* API, which will attempt to locate data files specific to the variant (if used), test name, and finally file name. For more information, please refer to the section *Accessing test data files*.
- The `avocado.Test.srcdir` attribute has been removed, and with it, the `AVOCADO_TEST_SRCDIR` environment variable set by Avocado. Tests should have been modified by now to make use of the *avocado.Test.workdir* instead.
- The `avocado.Test.datadir` attribute has been removed, and with it, the `AVOCADO_TEST_DATADIR` environment variable set by Avocado. Tests should now to make use of the *avocado.Test.get_data()* instead.
- Switched the *FileLoader* discovery to *:avocado: recursive* by default. All tags *enable*, *disable* and *recursive* are still available and might help fine-tuning the class visibility.

- The deprecated `skip` method, previously part of the *avocado.Test* API, has been removed. To skip a test, you can still use the *avocado.skip()*, *avocado.skipIf()* and *avocado.skipUnless()* decorators.
- The *Avocado Test class* now exposes the *tags* to the test. The test may use that information, for instance, to decide on default behavior.
- The Avocado test loader, which does not load or execute Python source code that may contain tests for security reasons, now operates in a way much more similar to the standard Python object inheritance model. Before, classes containing tests that would not directly inherit from *avocado.Test* would require a docstring statement (either `:avocado: enable` or `:avocado: recursive`). This is not necessary for most users anymore, as the recursive detection is now the default behavior.

Utility APIs

- The *avocado.utils.archive* module now supports the handling of gzip files that are not compressed tarballs.
- *avocado.utils.astring.ENCODING* is a new addition, and holds the encoding used on many other Avocado utilities. If your test needs to convert between binary data and text, we recommend you use it as the default encoding (unless your test knows better).
- *avocado.utils.astring.to_text()* now supports setting the error handler. This means that when a perfect decoding is not possible, users can choose how to handle it, like, for example, ignoring the offending characters.
- The *avocado.utils.astring.tabular_output()* will now properly strip trailing whitespace from lines that don't contain data for all "columns". This is also reflected in the (tabular) output of commands such as `avocado list -v`.
- Simple bytes and "unicode strings" utility functions have been added to *avocado.utils.astring*, and can be used by extension and test writers that need consistent results across Python major versions.
- The *avocado.utils.cpu.set_cpuidle_state()* function now takes a boolean value for its `disable` parameter (while still allowing the previous integer (0/1) values to be used). The goal is to have a more Pythonic interface, and to drop support legacy integer (0/1) use in the upcoming releases.
- The *avocado.utils.cpu* functions, such as `avocado.utils.cpu.cpu_oneline_list()` now support the S390X architecture.
- The *avocado.utils.distro* module has dropped the probe that depended on the Python standard library `platform.dist()`. The reason is the `platform.dist()` has been deprecated since Python 2.6, and has been removed on the upcoming Python 3.8.
- The *avocado.utils.distro* module introduced a probe for the Ubuntu distros.
- The `avocado.core.utils.vmimage` library now allows users to expand the builtin list of image providers. If you have a local cache of public images, or your own images, you can quickly and easily register your own providers and thus use your images on your tests.
- The *avocado.utils.vmimage* library now contains support for Avocado's own JeOS ("Just Enough Operating System") image. A nice addition given the fact that it's the default image used in Avocado-VT and the latest version is available in the following architectures: x86_64, aarch64, ppc64, ppc64le and s390x.
- The *avocado.utils.vmimage* library got a provider implementation for OpenSUSE. The limitation is that it tracks the general releases, and not the rolling releases (called Tumbleweed).
- The *avocado.utils.vmimage.get()* function now provides a directory in which to put the snapshot file, which is usually discarded. Previously, the snapshot file would always be kept in the cache directory, resulting in its pollution.

- The exception raised by the utility functions in *avocado.utils.memory* has been renamed from `MemoryError` and became *avocado.utils.memory.MemError*. The reason is that `MemoryError` is a Python standard exception, that is intended to be used on different situations.
- When running a process by means of the *avocado.utils.process* module utilities, the output of such a process is captured and can be logged in a `stdout`/`stderr` (or combined `output`) file. The logging is now more resilient to decode errors, and will use the `replace` error handler by default. Please note that the downside is that this *may* produce different content in those files, from what was actually output by the processes if decoding error conditions happen.
- The *avocado.utils.process* has seen a number of changes related to how it handles data from the executed processes. In a nutshell, process output (on both `stdout` and `stderr`) is now considered binary data. Users that need to deal with text instead, should use the newly added *avocado.utils.process.CmdResult.stdout_text* and *avocado.utils.process.CmdResult.stderr_text*, which are convenience properties that will attempt to decode the `stdout` or `stderr` data into a string-like type using the encoding set, and if none is set, falling back to the Python default encoding. This change of behavior was needed to accommodate Python's 2 and Python's 3 differences in bytes and string-like types and handling.
- The *avocado.utils.process* library now contains helper functions similar to the Python 2 `commands.getstatusoutput()` and `commands.getoutput()` which can be of help to people porting code from Python 2 to Python 3.
- New *avocado.utils.process.get_parent_pid()* and *avocado.utils.process.get_owner_id()* process related functions
- The *avocado.utils.kernel* library now supports setting the URL that will be used to fetch the Linux kernel from, and can also build installable packages on supported distributions (such as `.deb` packages on Ubuntu).
- The *avocado.utils.iso9660* module gained a pycdlib based backend, which is very capable, and pure Python ISO9660 library. This allows us to have a working *avocado.utils.iso9660* backend on environments in which other backends may not be easily installable.
- The *avocado.utils.iso9660.iso9660()* function gained a capabilities mechanism, in which users may request a backend that implement a given set of features.
- The *avocado.utils.iso9660* module, gained "create" and "write" capabilities, currently implemented on the pycdlib based backend. This allows users of the *avocado.utils.iso9660* module to create ISO images programmatically - a task that was previously done by running `mkisofs` and similar tools.
- The *avocado.utils.download* module, and the various utility functions that use it, will have extended logging, including the file size, time stamp information, etc.
- A brand new module, *avocado.utils.cloudinit*, that aides in the creation of ISO files containing configuration for the virtual machines compatible with cloudinit. Besides authentication credentials, it's also possible to define a "phone home" address, which is complemented by a simple phone home server implementation. On top of that, a very easy to use function to wait on the phone home is available as *avocado.utils.cloudinit.wait_for_phone_home()*.
- A new utility library, *avocado.utils.ssh*, has been introduced. It's a simple wrapper around the OpenSSH client utilities (your regular `/usr/bin/ssh`) and allows a connection/session to be easily established, and commands to be executed on the remote endpoint using that previously established connection.
- The *avocado.utils.cloudinit* module now adds support for instances to be configured to allow `root` logins and authentication configuration via SSH keys.
- New *avocado.utils.disk.get_disk_blocksize()* and *avocado.utils.disk.get_disks()* disk related utilities.
- A new network related utility function, `avocado.utils.network.PortTracker` was ported from Avocado-Virt, given the perceived general value in a variety of tests.

- A new memory utility utility, *avocado.utils.memory.MemInfo*, and its ready to use instance avocado.utils.memory.meminfo, allows easy access to most memory related information on Linux systems.
- A number of improvements to the *avocado.utils.lv_utils* module now allows users to choose if they want or not to use ramdisks, and allows for a more concise experience when creating Thin Provisioning LVs.
- New utility function in the *avocado.utils.genio* that allows for easy matching of patterns in files. See avocado.utils.is_pattern_in_file() for more information.
- New utility functions are available to deal with filesystems, such as *avocado.utils.disk.get_available_filesystems()* and *avocado.utils.disk.get_filesystem_type()*.
- The *avocado.utils.process.kill_process_tree()* now supports waiting a given timeout, and returns the PIDs of all process that had signals delivered to.
- The avocado.utils.network.is_port_free() utility function now supports IPv6 in addition to IPv4, as well as UDP in addition to TCP.
- A new *avocado.utils.cpu.get_pid_cpus()* utility function allows one to get all the CPUs being used by a given process and its threads.
- The *avocado.utils.process* module now exposes the timeout parameter to users of the *avocado.utils.process.SubProcess* class. It allows users to define a timeout, and the type of signal that will be used to attempt to kill the process after the timeout is reached.

Users

- Passing parameters to tests is now possible directly on the Avocado command line, without the use of any varianter plugin. In fact, when using variants, these parameters are (currently) ignored. To pass one parameter to a test, use -p NAME=VAL, and repeat it for other parameters.
- The test filtering mechanism using tags now support "key:val" assignments for further categorization. See *Python unittest Compatibility Limitations And Caveats* for more details.
- The output generated by tests on stdout and stderr are now properly prefixed with [stdout] and [stderr] in the job.log. The prefix is **not** applied in the case of $test_result/stdout and $test_result/stderr files, as one would expect.
- The installation of Avocado from sources has improved and moved towards a more "Pythonic" approach. Installation of files in "non-Pythonic locations" such as /etc are no longer attempted by the Python setup.py code. Configuration files, for instance, are now considered package data files of the avocado package. The end result is that installation from source works fine outside virtual environments (in addition to installations *inside* virtual environments). For instance, the locations of /etc (config) and /usr/libexec (libexec) files changed to live within the pkg_data (eg. /usr/lib/python2.7/site-packages/avocado/etc) by default in order to not to modify files outside the package dir, which allows user installation and also the distribution of wheel packages. GNU/Linux distributions might still modify this to better follow their conventions (eg. for RPM the original locations are used). Please refer to the output of the avocado config command to see the configuration files that are actively being used on your installation.
- SIMPLE tests were limited to returning PASS, FAIL and WARN statuses. Now SIMPLE tests can now also return SKIP status. At the same time, SIMPLE tests were previously limited in how they would flag a WARN or SKIP from the underlying executable. This is now configurable by means of regular expressions.
- Sysinfo collection can now be enabled on a test level basis.
- Avocado can record the output generated from a test, which can then be used to determine if the test passed or failed. This feature is commonly known as "output check". Traditionally, users would choose to record the output from STDOUT and/or STDERR into separate streams, which would be saved into different files.

Some tests suites actually put all content of `STDOUT` and `STDERR` together, and unless we record them together, it'd be impossible to record them in the right order. This version introduces the `combined` option to `--output-check-record` option, which does exactly that: it records both `STDOUT` and `STDERR` into a single stream and into a single file (named `output` in the test results, and `output.expected` in the test data directory).

- The complete output of tests, that is the combination of `STDOUT` and `STDERR` is now also recorded in the test result directory as a file named `output`.
- When the output check feature finds a mismatch between expected and actual output, will now produce a unified diff of those, instead of printing out their full content. This makes it a lot easier to read the logs and quickly spot the differences and possibly the failure cause(s).
- The output check feature will now use the to the most specific data source location available, which is a consequence of the switch to the use of the `get_data()` API discussed previously. This means that two tests in a single file can generate different output, generate different `stdout.expected` or `stderr.expected`.
- *SIMPLE <test_type_simple>* tests can also finish with `SKIP` OR `WARN` status, depending on the output produced, and the Avocado test runner configuration. It now supports patterns that span across multiple lines. For more information, refer to *SIMPLE Tests Status*.
- A better handling of interruption related signals, such as `SIGINT` and `SIGTERM`. Avocado will now try harder to not leave test processes that don't respond to those signals, and will itself behave better when it receives them. For a complete description refer to *signal_handlers*.
- Improvements in the serialization of TestIDs allow test result directories to be properly stored and accessed on Windows based filesystems.
- The deprecated `jobdata/urls` link to `jobdata/test_references` has been removed.
- The `avocado` command line argument parser is now invoked before plugins are initialized, which allows the use of `--config` with configuration file that influence plugin behavior.
- The test log now contains a number of metadata about the test, under the heading `Test metadata:`. You'll find information such as the test file name (if one exists), its `workdir` and its `teststmpdir` if one is set.
- The test runner will now log the test initialization (look for `INIT` in your test logs) in addition to the already existing start of test execution (logged as `START`).
- The test profilers, which are defined by default in `/etc/avocado/sysinfo/profilers`, are now executed without a backing shell. While Avocado doesn't ship with examples of shell commands as profilers, or suggests users to do so, it may be that some users could be using that functionality. If that's the case, it will now be necessary to write a script that wraps you previous shell command. The reason for doing so, was to fix a bug that could leave profiler processes after the test had already finished.
- The Human UI plugin, will now show the "reason" behind test failures, cancellations and others right along the test result status. This hopefully will give more information to users without requiring them to resort to logs every single time.
- When installing and using Avocado in a Python virtual environment, the ubiquitous "venvs", the base data directory now respects the virtual environment. If you have are using the default data directory outside of a venv, please be aware that the updated
- Avocado packages are now available in binary "wheel" format on PyPI. This brings faster, more convenient and reliable installs via `pip`. Previously, the source-only tarballs would require the source to be built on the target system, but the wheel package install is mostly an unpack of the already compiled files.
- The legacy options `--filter-only`, `--filter-out` and `--multiplex` have now been removed. Please adjust your usage, replacing those options with `--mux-filter-only`, `--mux-filter-out` and `--mux-yaml` respectively.
- The location of the Avocado configuration files can now be influenced by third parties by means of a new plugin.

- The configuration files that have been effectively parsed are now displayed as part of `avocado config` command output.

Output Plugins

- Including test logs in TAP plugin is disabled by default and can be enabled using `--tap-include-logs`.
- The TAP result format plugin received improvements, including support for reporting Avocado tests with CANCEL status as SKIP (which is the closest status available in the TAP specification), and providing more visible warning information in the form of comments when Avocado tests finish with WARN status (while maintaining the test as a PASS, since TAP doesn't define a WARN status).
- A new (optional) plugin is available, the "result uploader". It allows job results to be copied over to a centralized results server at the end of job execution. Please refer to *Results Upload Plugin* for more information.
- Added possibility to limit the amount of characters embedded as "system-out" in the xunit output plugin (`--xunit-max-test-log-chars XX`).
- The `xunit` result plugin can now limit the amount of output generated by individual tests that will make into the XML based output file. This is intended for situations where tests can generate prohibitive amounts of output that can render the file too large to be reused elsewhere (such as imported by Jenkins).
- The xunit output now names the job after the Avocado job results directory. This should make the correlation of results displayed in UIs such as Jenkins and the complete Avocado results much easier.
- The xUnit plugin now should produce output that is more compatible with other implementations, specifically newer Jenkin's as well as Ant and Maven. The specific change was to format the time field with 3 decimal places.
- Redundant (and deprecated) fields in the test sections of the JSON result output were removed. Now, instead of `url`, `test` and `id` carrying the same information, only `id` remains.

Test Loader Plugins

- A new loader implementation, that reuses (and resembles) the YAML input used for the varianter yaml_to_mux plugin. It allows the definition of test suite based on a YAML file, including different variants for different tests. For more information refer to yaml_loader.
- Users of the YAML test loader have now access to a few special keys that can tweak test attributes, including adding prefixes to test names. This allows users to easily differentiate among execution of the same test, but executed different configurations. For more information, look for "special keys" in the YAML Loader plugin documentation.
- A new plugin enables users to list and execute tests based on the GLib test framework. This plugin allows individual tests inside a single binary to be listed and executed.
- Avocado can now run list and run standard Python unittests, that is, tests written in Python that use the `unittest` library alone.
- Support for listing and running golang tests has been introduced. Avocado can now discover tests written in Go, and if Go is properly installed, Avocado can run them.

Varianter Plugins

- A new varianter plugin has been introduced, based on PICT. PICT is a "Pair Wise" combinatorial tool, that can generate optimal combination of parameters to tests, so that (by default) at least a unique pair of parameter values will be tested at once.

- A new varianter plugin, the *CIT Varianter Plugin*. This plugin implements a "Pair-Wise", also known as "Combinatorial Independent Testing" algorithm, in pure Python. This exciting new functionality is provided thanks to a collaboration with the Czech Technical University in Prague.
- Users can now dump variants to a (JSON) file, and also reuse a previously created file in their future jobs execution. This allows users to avoid recomputing the variants on every job, which might bring significant speed ups in job execution or simply better control of the variants used during a job. Also notice that even when users do not manually dump a variants file to a specific location, Avocado will automatically save a suitable file at `jobdata/variants.json` as part of a Job results directory structure. The feature has been isolated into a varianter implementation called `json_variants`, that you can see with `avocado plugins`.

Test Runner Plugins

- The command line options `--filter-by-tags` and `--filter-by-tags-include-empty` are now white listed for the remote runner plugin.
- The remote runner plugin will now respect `~/.ssh/config` configuration.

Complete list of changes

For a complete list of changes between the last LTS release (52.1) and this release, please check out the Avocado commit changelog.

52.0 LTS

The Avocado team is proud to present another release: Avocado version 52.0, the second Avocado LTS version.

What's new?

When compared to the last LTS (v36), the main changes introduced by this versions are:

- Support for TAP[2] version 12 results, which are generated by default in test results directory (`results.tap` file).
- The download of assets in tests now allow for an expiration time.
- Environment variables can be propagated into tests running on remote systems.
- The plugin interfaces have been moved into the `avocado.core.plugin_interfaces` module.
- Support for running tests in a Docker container.
- Introduction of the "Fail Fast" feature (`--failfast on` option) to the `run` command, which interrupts the Job on a first test failure.
- Special keyword `latest` for replaying previous jobs.
- Support to replay a Job by path (in addition to the Job ID method and the `latest` keyword).
- Diff-like categorized report of jobs (`avocado diff <JOB_1> <JOB_2>`).
- The introduction of a `rr` based wrapper.
- 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).

- Set `LC_ALL=C` by default on sysinfo collection to simplify avocado diff comparison between different machines.
- Result plugins system is now pluggable and the results plugins (JSON, XUnit, HTML) were turned into stevedore plugins. They are now listed in the `avocado plugins` command.
- Multiplexer was replaced with Varianter plugging system with defined API to register plugins that generate test variants.
- Old `--multiplex` argument, which used to turn yaml files into variants, is now handled by an optional plugin called `yaml_to_mux` and the `--multiplex` option is being deprecated in favour of the `--mux-yaml` option, which behaves the same way.
- It's now possible to disable plugins by using the configuration file.
- Better error handling of the virtual machine plugin (`--vm-domain` and related options).
- When discovering tests on a directory, the result now is a properly alphabetically ordered list of tests.
- Plugins can now be setup in Avocado configuration file to run at a specific order.
- Support for filtering tests by user supplied "tags".
- Users can now see the test tags when listing tests with the `-V` (verbose) option.
- 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 (e.g. to keep those files for `rr`).
- 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.
- Encoding support was improved and now Avocado should safely treat localized test-names.
- 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 to allow sharing state/exchanging data between tests. The path for that directory is available via Test API (`self.teststmpdir`) and via environment variable (`AVOCADO_TESTS_COMMON_TMPDIR`).
- Avocado is now available on Fedora standard repository. The package name is `python2-avocado`. The optional plugins and examples packages are also available. Run `dnf search avocado` to list them all.
- Optional plugins and examples packages are also available on PyPI under `avocado-framework` name.
- Avocado test writers can now use a family of decorators, namely `avocado.skip()`, `avocado.skipIf()` and `avocado.skipUnless()` to skip the execution of tests.
- Sysinfo collection based on command execution now allows a timeout to be set in the Avocado configuration file.
- The non-local runner plugins, the html plugin and the yaml_to_mux plugin are now distributed in separate packages.
- The Avocado main process will now try to kill all test processes before terminating itself when it receives a SIGTERM.
- Support for new type of test status, `CANCEL`, and of course the mechanisms to set a test with this status (e.g. via `self.cancel()`).
- `avocado.TestFail`, `avocado.TestError` and `avocado.TestCancel` are now public Avocado Test APIs, available in the main *avocado* namespace.
- Introduction of the robot plugin, which allows Robot Framework tests to be listed and executed natively within Avocado.
- A brand new ResultsDB optional plugin.

- Listing of supported loaders (`--loaders \?`) was refined.
- Variant-IDs generated by yaml_to_mux plugin now include leaf node names to make them more meaningful, making easier to skim through the results.
- *yaml_to_mux* now supports internal filters defined inside the YAML file expanding the filtering capabilities even further.
- Avocado now supports resuming jobs that were interrupted.
- 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.
- New `avocado variants` has been added which supersedes the `avocado multiplex`.
- Loaders were tweaked to provide more info on `avocado list -V` especially when they don't recognize the reference.
- Users can use `--ignore-missing-references on` to run a job with undiscovered test references
- Users can now choose in which order the job will execute tests (from its suite) and variants. The two available options are `--execution-order=variants-per-test` (default) or `--execution-order=tests-per-variant`.
- Test methods can be recursively discovered from parent classes by upon the `:avocado: recursive` docstring directive.

Besides the list above, we had several improvements in our `utils` libraries that are important for test writers, some of them are listed below:

- `time_to_seconds`, `geometric_mean` and `compare_matrices` were added in `avocado.utils.data_structures`.
- `avocado.utils.distro` was refined.
- Many `avocado.utils` new modules were introduced, like `filelock`, `lv_utils`, `multipath`, `partition` and `pci`.
- `avocado.utils.memory` contains several new methods.
- New `avocado.utils.process.SubProcess.get_pid()` method.
- `sudo` support in `avocado.utils.process` was improved
- The `avocado.utils.process` library makes it possible to ignore spawned background processes.
- New `avocado.utils.linux_modules.check_kernel_config()`.
- Users of the `avocado.utils.process` module will now be able to access the process ID in the `avocado.utils.process.CmdResult`.
- Improved `avocado.utils.iso9660` with a more complete standard API across all back-end implementations.
- Improved `avocado.utils.build.make()`, which will now return the make process exit status code.
- The `avocado.Test` class now better exports (and protects) the core class attributes members (such as `params` and `runner_queue`).
- `avocado.utils.linux_modules` functions now returns 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`.

It is also worth mentioning:

- Improved documentation, with new sections to Release Notes and Optional Plugins, very improved Contribution and Community Guide. New content and new examples everywhere.
- The avocado-framework-tests GitHub organization was founded to encourage companies to share Avocado tests.
- Bugs were always handled as high priority and every single version was delivered with all the reported bugs properly fixed.

When compared to the last LTS, we had:

- 1187 commits (and counting).
- 15 new versions.
- 4811 more lines of Python code (+27,42%).
- 1800 more lines of code comment (+24,67%).
- 31 more Python files (+16,48%).
- 69 closed GitHub issues.
- 34 contributors from at least 12 different companies, 26 of them contributing for the fist time to the project.

Switching from 36.4 to 52.0

You already know what new features you might expect, but let's emphasize the main changes required to your workflows/tests when switching from 36.4 to 52.0

Installation

All the previously supported ways to install Avocado are still valid and few new ones were added, but beware that Avocado was split into several optional plugins so you might want to adjust your scripts/workflows.

- Multiplexer (the YAML parser which used to generate variants) was turned into an optional plugin `yaml_to_mux` also known as `avocado_framework_plugin_varianter_yaml_to_mux`. Without it Avocado does not require PyYAML, but you need it to support the parsing of YAML files to variants (unless you use a different plugin with similar functionality, which is now also possible).
- The HTML result plugin is now also an optional plugin so one has to install it separately.
- The remote execution features (`--remote-hostname`, `--vm-domain`, `--docker`) were also turned into optional plugins so if you need those you need to install them separately.
- Support for virtual environment (`venv`) was greatly improved and we do encourage people who want to use `pip` to do that via this method.

As for the available ways:

- Fedora/RHEL can use our custom repositories, either LTS-only or all releases. Note that latest versions (non-lts) are also available directly in Fedora and also in EPEL.
- OpenSUSE - Ships the 36 LTS versions, hopefully they'll start shipping the 52 ones as well (but we are not in charge of that process)
- Debian - The `contrib/packages/debian` script is still available, although un-maintained for a long time
- PyPI/pip - Avocado as well as all optional plugins are available in PyPI and can be installed via `pip install avocado-framework\*`, or selectively one by one.

- From source - Makefile target `install` is still available but it does not install the optional plugins. You have to install them one by one by going to their directory (eg. `cd optional_plugins/html` and running `sudo python setup.py install`)

As before you can find the details in *Installing Avocado*.

Usage

Note: As mentioned in previous section some previously core features were turned into optional plugins. Do check your install script if some command described here are missing on your system.

Most workflows should work the same, although there are few little changes and a few obsoleted constructs which are still valid, but you should start using the new ones.

The hard changes which does not provide backward compatibility:

- **Human result was tweaked a bit:**
  - The `TESTS` entry (displaying number of tests) was removed as one can easily get this information from `RESULTS`.
  - Instead of tests time (sum of test times) you get job time (duration of the job execution) in the human result
- **Json results also contain some changes:**
  - They are pretty-printed
  - As `cancel` status was introduced, json result contain an entry of number of canceled tests (`cancel`)
  - url was renamed to id (url entry is to be removed in `53.0` so this is actually a soft change with a backward compatibility support)
- The `avocado multilex|variants` does not expect multiplex YAML files as positional arguments, one has to use `-m|--mux-yaml` followed by one or more paths.
- Test variants are not serialized numbers anymore in the default `yaml_to_mux` (`multiplexer`), but ordered list of leaf-node names of the variant followed by hash of the variant content (paths+environment). Therefore instead of `my_test:1` you can get something like `my_test:arm64-virtio_scsi-RHEL7-4a3c`.
- `results.tap` is now generated by default in job results along the `results.json` and `results.xml` (unless disabled)
- The `avocado run --replay` and `avocado diff` are unable to parse results generated by `36.4` to this date. We should be able to introduce such feature with not insignificant effort, but no one was interested yet.

And the still working but to be removed in `53.0` constructs:

- The long version of the `-m|--multiplex` argument available in `avocado run|multiplex|variants` was renamed to `-m|--mux-yaml` which corresponds better to the rest of `--mux-*` arguments.
- The `avocado multiplex` was renamed to `avocado variants`
- The `avocado multiplex|variants` arguments were reworked to better suite the possible multiple varianter plugins:
  - Instead of picking between `tree` representation of list of variants one can use `--summary`, resp `--variants` followed by verbosity, which supersedes `-c|contents`, `-t|--tree`, `-i|--inherit`

  - Instead of `--filter-only|--filter-out` the `--mux-filter-only|--mux-filter-out` are available
  - The `--mux-path` is now also available in `avocado multiplex|variants`

Test API

Main features stayed the same, there are few new ones so do check our documentation for details. Anyway while porting tests you should pay attention to following changes:

- If you were overriding `avocado.Test` attributes (eg. `name`, `params`, `runner_queue`, ...) you'll get an `AttributeError: can't set attribute` error as most of them were turned into properties to avoid accidental override of the important attributes.
- The `tearDown` method is now executed almost always (always when the `setUp` is entered), including when the test is interrupted while running `setUp`. This might require some changes to your `setUp` and `tearDown` methods but generally it should make them simpler. (See *Setup and cleanup methods* and following chapters for details)
- Test exceptions are publicly available directly in `avocado` (`TestError`, `TestFail`, `TestCancel`) and when raised inside test they behave the same way as `self.error`, `self.fail` or `self.cancel`. (See *avocado*)
- New status is available called `CANCEL`. It means the test (or even just `setUp`) started but the test does not match prerequisites. It's similar to `SKIP` in other frameworks, but the `SKIP` result is reserved for tests that were not executed (nor the `setUp` was entered). The `CANCEL` status can be signaled by `self.cancel` or by raising `avocado.TestCancel` exception and the `SKIP` should be set only by `avocado.skip`, `avocado.skipIf` or `avocado.skipUnless` decorators. The `self.skip` method is still supported but will be removed after in `53.0` so you should replace it by `self.cancel` which has similar meaning but it additionally executes the `tearDown`. (See *Test statuses*
- The `tag` argument of `avocado.Test` was removed as it is part of `name`, which can only be `avocado.core.test.TestName` instance. (See *avocado.core.test.Test()*)
- The `self.job.logdir` which used to be abused to share state/data between tests inside one job can now be dropped towards the `self.teststmpdir`, which is a shared temporary directory which sustains throughout job execution and even between job executions if set via `AVOCADO_TESTS_COMMON_TMPDIR` environmental value. (See *avocado.core.test.Test.teststmpdir()*)
- Those who write inherited test classes will be pleasantly surprised as it is now possible to mark a class as avocado test including all `test*` methods coming from all parent classes (similarly to how dynamic discovery works inside Python unittest, see *docstring-directive-recursive* for details)
- The `self.text_output` is not published after the test execution. If you were using it simply open the `self.logfile` and read the content yourself.

Utils API

Focusing only on the changes you might need to adjust the usage of:

- `avocado.utils.build.make` calls as it now reports only `exit_status`. To get the full result object you need to execute `avocado.utils.build.run_make`.
- `avocado.utils.distro` reports `Red Hat Enterprise Linux/rhel` instead of `Red Hat/redhat`.
- `avocado.process` where the check for availability of `sudo` was improved, which might actually start executing some code which used to fail in 36.4.

Also check out the `avocado.utils` for complete list of available utils as there were many additions between 36.4 and 52.0.

Complete list of changes

For a complete list of changes between the last LTS release (36.4) and this release, please check out the Avocado commit changelog.

The Next LTS

The Long Term Stability releases of Avocado are the result of the accumulated changes on regular (non-LTS) releases.

This section tracks the changes introduced on each regular (non-LTS) Avocado release, and gives a sneak preview of what will make into the next LTS release.

What's new?

When compared to the last LTS (82.x), the main changes to be introduced by the next LTS version are:

Test Writers

Test APIs

Utility APIs

Users

Output Plugins

Test Loader Plugins

Varianter Plugins

Test Runner Plugins

Complete list of changes

For a complete list of changes between the last LTS release (82.0) and this release, please check out the Avocado commit changelog.

9.6.3 Regular Releases

92.0 Monsters, Inc.

The Avocado team is proud to present another release: Avocado 92.0, AKA "Monsters, Inc.", is now available!

Release documentation: Avocado 92.0

Important Announcement

On the previous version (91.0), Avocado has switched the default runner, from the implementation most people currently use (internally simply called `runner`), to the newer architecture and implementation called `nrunner`).

Users migrating from Avocado 90.0 or earlier versions will be impacted by this change and should act accordingly.

To keep using the current (soon to be legacy) runner, you **must** set the `--test-runner=runner` command line option (or the equivalent `test_runner` configuration option, under section `[run]`).

Known issues are being tracked on our GitHub project page, with the `nrunner` tag, and new issue reports are appreciated.

Users/Test Writers

- The Human UI plugin can now be configured to omit certain statuses from being show in a new line. This can be used, for instance, to prevent the `STARTED` lines to be shown, showing only the final test result.
- The nrunner `exec` runnable kind does not exist anymore, and its functionality was consolidated into the `exec-test`.
- Executing Python's unittest that are skipped are now always shown as having status `SKIP`, instead of the previous `CANCEL`.
- Avocado will no longer incorporate log messages coming from any logger (including the "root logger") into the test's and job's log files. Only loggers that under the `avocado.` namespace will be included. Users are encouraged to continue to follow the pattern:

  ```
  self.log.info("message goes here")
  ```

  When logging from a test. When logging from somewhere else, the following pattern is advised (replace `my.namespace` accordingly):

  ```
  import logging
  LOG = logging.getLogger('avocado.my.namespace')
  LOG.info('your message')
  ```

- Python 3.10 is now fully supported.
- The reason for fail/error/skip tests in Python unittest are now given on the various test result formats (including on the UI).

Bug Fixes

- Properties, that is, methods decorated with `@property` are no longer seen as tests.
- If a path to a Python unittest file contained dots, the conversion to a unittest "dotted name" would fail.
- Tests on classes that inherit from one marked with `:avocado: disable` were not being detected.

Utility APIs

- A specific exception, and thus a clearer error message, is now used when a command with an empty string is given to *`avocado.utils.process.run()`*.

Misc Changes

- Added example jobs for `exec-test` runnables, and for YAML to Mux.
- The test logs on the nrunner generated `debug.log` file are now prefixed with `[stdlog]` instead of `[debug]` to avoid confusion with the log level of same name.
- Added setuptools entrypoints for all nrunner implementations under `avocado.plugins.runnable.runner`.
- The Podman spawner now checks for the existence of the `podman` binary earlier.
- Misc documentation improvements.

Internal Changes

- CI deployment checks with different installation methods (sources) were added.
- The `--disble-plugin-check` argument to `selftests/check.py` now properly accepts multiple plugin names.
- Templates for GitHub issues for bugs and feature requests were added.

—

For more information, please check out the complete Avocado changelog.

91.0 Thelma & Louise

The Avocado team is proud to present another release: Avocado 91.0, AKA "Thelma & Louise", is now available!

Release documentation: Avocado 91.0

Important Announcement

Avocado has switched the default runner, from the implementation most people currently use (internally simply called `runner`), to the newer architecture and implementation called `nrunner`.

Users installing and relying on the latest Avocado release will be impacted by this change and should act accordingly.

To keep using the current (soon to be legacy) runner, you **must** set the `--test-runner=runner` command line option (or the equivalent `test_runner` configuration option, under section `[run]`).

Known issues are being tracked on our GitHub project page, with the `nrunner` tag, and new issue reports are appreciated.

Users/Test Writers

- As per the previous section, the `nrunner` test runner implementation is now the default on every `avocado run` command (or equivalent Job API scripts). Since the previous release, `nrunner` supports:

1. the "fail fast" (`run --failfast`) feature.
2. *the varianter* feature.
3. UNIX domain sockets as the communication channel with runners (the new default).

4. a `sysinfo` runner, which will allow for sysinfo collection on any supported spawner.
5. early notification of missing runners for tasks in the requested suite.

- The assets plugin fetch command (`avocado assets fetch`) now supports:

1. fetching assets defined in a Python list in `INSTRUMENTED` tests.
2. setting a timeout for the download of assets.

- Improved checks when users attempt to use the varianter and simple parameters (`-p`) at the same time.
- The Podman spawner (`--nrunner-spawner=podman`) will now attempt to use a Container image (`--spawner-podman-image=`) that matches the host Linux distribution. If it's not possible to detect the host distribution, the latest Fedora image will be used.

Bug Fixes

- The extraction of DEB packages by means of *avocado.utils.software_manager.SoftwareManager.extract_from_package()* was fixed and does not depend on `ar` utility anymore (as it now uses the *avocado.utils.ar* module).
- The `--store-logging-stream` parameter value was being incorrectly parsed as a list of characters. If a `bar` value is given, it would generate the `b.INFO`, `a.INFO`, and `r.INFO` file. The fix parses the command line arguments by treating the value as a comma separated list (that becomes a set).
- `nrunner` will now properly translate reference names with absolute paths into Python unittest "dotted names".
- The `TAP` parser (*avocado.core.tapparser*) will not choke on unexpected content, ignoring it according to the standard.
- `avocado.core.nrunner.Runnables` created by suites will now contain the full suite configuration.
- If a job contains multiple test suites with the same name, and tests within those suites also have the same name, test results would be overwritten. Now job name uniqueness is enforced and no test results from a suite should be able to overwrite other's.
- Some occurrences of the incorrect `AVOCADO_TEST_OUTPUT_DIR` environment variable name were renamed to the proper name (`AVOCADO_TEST_OUTPUTDIR`).

Utility APIs

- *avocado.utils.network.interfaces.NetworkInterface* can now access and present information on interfaces that do not have an IP address assigned to them.
- *avocado.utils.distro* can now detect the distribution on remote machines.
- A new *avocado.utils.ar* module was introduced that allows extraction of UNIX `ar` archive contents.
- A new *avocado.utils.sysinfo* module that powers the `sysinfo` feature, but is now also accessible to job/test writers.
- Times related to the duration of tasks are now limited to nanosecond precision to improve readability.
- The *avocado.utils.distro* will now correctly return a `avocado.utils.distro.UNKNOWN_DISTRO` on non UNIX systems, instead of crashing.
- The *avocado.utils.network.hosts* won't consider anymore `bonding_masters`, a file that may exist at `/sys/class/net`, as the name of an interface.

Misc Changes

- Many documentation improvements.
- The `setup.py` script received many improvements, including:

1. A new `test` command implementation.
2. A new `plugin` command.

- Various logging related improvements, especially regarding the elimination of Python's root logger usage.

Internal Changes

- Major improvements to the CI, including:

1. additional tests for `setup.py`
2. most of the "pre-release" test plan was migrated to a CI job

- The *avocado.Test.workdir()* is now initialized lazily, which prevents never used work directories from being created and kept in the test results' directory.
- A circular import condition was fixed related to the Settings plugins initialized during avocado early initialization.
- A rename of the `requirements-selftests.txt` file to `requirements-dev.txt` to better capture the fact that the selftests requirements are a subset of the requirements needed for development.
- `selftests/safeloader.sh` now supports checking a single file.
- Debugging aids were added to the Task/StateMachine/Repo systems of the `nrunner`. These can be seen by default at the `avocado.core.DEBUG` file in the job's result directory.
- An `nrunner` based job its tasks are now better bound by the job ID, with messages being logged in case the status server receives messages destined for another job.
- The very long and detailed temporary directory prefix used on `TestCaseTmpDir` was shortened so that limitations on paths such as socket names are not easily hit.
- Clean up of duplicate methods in the *avocado.core.loader.FileLoader* class.

—

For more information, please check out the complete Avocado changelog.

90.0 Bladerunner

The Avocado team is proud to present another release: Avocado 90.0, AKA "Bladerunner", is now available!

Release documentation: Avocado 90.0

Important Announcement

The Avocado team is planning to switch the default runner, from the implementation most people currently use (internally simply called `runner`), to the newer architecture and implementation called `nrunner`. This is scheduled to happen on version 91.0 (the next release).

Users installing and relying on the latest Avocado release will be impacted by this change and should plan accordingly.

To keep using the current (soon to be legacy) runner, you **must** set the `--test-runner=runner` command line option (or the equivalent `test_runner` configuration option, under section `[run]`).

Known issues are being tracked on our GitHub project page, with the `nrunner` tag, and new issue reports are appreciated.

Tip: To select the `nrunner` on this release (90.0 and earlier), run `avocado run --test-runner=nrunner`.

Users/Test Writers

- Avocado's safeloader (the system used to find Python based tests without executing them) received a major overhaul and now supports:

1. Multi-level module imports, such as `from my.base.test import Test` where a project may contain a `my/base` directory structure containing `test.py` that defines a custom `Test` class.
2. Support for following the import/inheritance hierarchy when a module contains an import for a given symbol, instead of the actual `class` definition of a symbol.
3. Considers coroutines (AKA `async def`) as valid tests, reducing the number of boiler plate code necessary for tests of `asyncio` based code.
4. Supports class definitions (containing tests or not) that use a typing hint with subscription, commonly used in generics.

- Test parameters given with `-p` are now support when using the `nrunner`.
- All status server URIs in the configuration are now respected for `nrunner` executions.
- The resolver plugins now have access to the job/suite configuration.
- The data directories now have less heuristics and are now more predictable and consistent with the configuration set.
- The JSON results (`results.json`) now contain a field with the path of the test log file.
- The root logger for Python's `logging` should no longer be impacted by Avocado's own logging initialization and clean up (which now limits itself to `avocado.*` loggers).

Bug Fixes

- The `whiteboard` file and data are now properly saved when using the `nrunner`
- The Podman spawner will now respect the Podman binary set in the job configuration.
- The date and time fields shown on some result formats, such as in the HTML report, now are proper dates/times, and not Python's "monotonic" date/time.
- The correct failure reason for tests executed with the `nrunner` are now being captured, instead of a possible exception caused by a error within the runner itself.

Utility APIs

- *`avocado.utils.ssh`* now respects the username set when copying files via `scp`.

Misc Changes

- Update of all executable script's "shebangs" to `/usr/bin/env python3` from `/usr/bin/env python`
- Better handling of `KeyboardInterrupt` exceptions on early stages of the Avocado execution.
- The list of external resources was updated adding a number of projects that either are extensions of Avocado, or that use Avocado for their testing needs.

Internal Changes

- `selftests/check_tmp_dirs` will only check for directories, ignoring files.
- The examples in the documentation no longer contain user references to specific users, using generic names and paths instead.
- A duplicated step has been removed from pre-release test plan.
- A setuptools command to build the man page was added.
- Updates to the Travis CI jobs, testing only Python 3.9 on s390x, ppc64le, and arm64, following the move to GHA.
- A weekly GHA CI job was introduced.
- Better standardization of the messages that `nrunner` runners generate by means of new utility methods.
- Allows the exclusion of optional plugins when doing `python3 setup.py develop`.

For more information, please check out the complete Avocado changelog.

89.0 Shrek

The Avocado team is proud to present another release: Avocado 89.0, AKA "Shrek", is now available!

Release documentation: Avocado 89.0

Important Announcement

The Avocado team is planning to switch the default runner, from the implementation most people currently use (internally simply called `runner`), to the newer architecture and implementation called `nrunner`. This may happen as soon as version 90.0 (the next release).

Users installing and relying on the latest Avocado release will be impacted by this change and should plan accordingly.

To keep using the current (soon to be legacy) runner, you **must** set the `--test-runner=runner` command line option (or the equivalent `test_runner` configuration option, under section `[run]`).

Known issues are being tracked on our GitHub project page, with the `nrunner` tag, and new issue reports are appreciated.

Tip: To select the `nrunner` on this release (89.0 and earlier), run `avocado run --test-runner=nrunner`.

Users/Test Writers

- A new `asset` requirement type has been introduced, allowing users to declare any asset obtainable with *avocado.utils.asset* to be downloaded, cached and thus be available to tests.
- `--dry-run` is now supported for the `nrunner`.
- The man page has been thoroughly updated and put in sync with the current `avocado` command features and options.
- Avocado can now run from Python eggs. It's expected that official egg builds will be made available starting with Avocado 90.0. Avocado is planning to use eggs as an automatic and transparent deployment mechanism for environments such as containers and VMs.
- The `datadir.paths.logs_dir` and `datadir.paths.data_dir` are set to more consistent and predictable values, and won't rely anymore on dynamic probes for "suitable" directories.

Bug Fixes

- The `nrunner` now properly sets all test status status to the suite summary, making sure that errors are communicated to the end user through, among other means, the `avocado` execution exit code.
- When running tests in parallel, multiple downloads of the same image (when using *avocado.utils.vmimage*) is now prevented by a better (early) locking.
- A condition in which tests running in parallel could collide over the existence of the asset's cache directory (created by other running tests) is now fixed.

Utility APIs

- *avocado.utils.software_manager.SoftwareManager.extract_from_package()* is a new method that lets users extract the content of supported package types (currently `RPM` and `deb`).
- *avocado.utils.vmimage.get()* is now deprecated in favor of *avocado.utils.vmimage.Image.from_parameters()*

Internal Changes

- *avocado.core.plugin_interfaces.Discoverer* is a new type of plugin interface that has been introduced to allow tests to be discovered without the need of references.
- Avocado now uses `time.monotonic()` pretty much everywhere it's possible. This time function will survive clock updates, and will never go back.
- The safeloader, the Avocado component that looks for `avocado-instrumented` and `python-unittest` tests without executing possibly untrusted code, has seen a big refactor in this release, with an extended test coverage too.
- The `avocado-runner-requirement-package` will now check for a package before installing it. This is an optimization and reduces the chance of multiple instances attempting to install packages at the same time.
- Improvements to the handling and saving of messages generated by the `nrunner`.
- The `nrunner` received some prep work for supporting variants. Jobs using the `nrunner` can now see the variants being applied to test suites, but be aware that the parameters on variants are still not passed to the tests.
- The requirement runnables now have access to their "parent" configuration.

Misc Changes

- The documentation has been update and gives more precise instructions for the set up of development environments.
- Major changes to the CI, in a trend towards using more GH Actions based jobs.

For more information, please check out the complete Avocado changelog.

88.1 The Serpent

This is a hotfix release for 88.0, with only one change to accommodate a documentation build error on readthedocs.org caused by a new version of an external package requirement.

For the other (more relevant) changes in the 88.x release, please refer to the *88.0 Release Notes*.

88.0 The Serpent

The Avocado team is proud to present another release: Avocado 88.0, AKA "The Serpent", is now available!

Release documentation: Avocado 88.0

Users/Test Writers

- The Requirements Resolver feature has been introduced, and it's available for general use. It allows users to describe requirements tests may have, and will attempt to fulfill those before the test is executed. This initial version has support for "package" requirements, meaning operating system level packages such as RPM, DEB, etc.

  Long story short, if you're writing a functional test that manipulates Logical Volumes, you may want to declare that the `lvm2` is a package requirement of your test.

  This can greatly simplify the setup of the environments the tests will run on, and at the same time, not cause test errors because of the missing requirements (which will cause the test to be skipped).

  For more information please refer to the *Managing Requirements* section.
- `avocado list` got a `--json` option, which will output the list of tests in a machine readable format.
- The minimal Python version requirement now is 3.6. Python 3.5 and earlier are not tested nor supported starting with this release.
- Because of the characteristics of the nrunner architecture, it has been decided that log content generated by tests will **not** be copied to the `job.log` file, but will only be available on the respective test logs on the `test-results` directory. Still, will often need to know if tests have been started or have finished while looking at the `job.log` file. This feature has been implemented by means of the `testlogs` plugin.
- Avocado will log a warning, making it clear that it can not check the integrity of a requested asset when no hash is given. This is related to users of the *`avocado.utils.asset`* module or *`avocado.Test.fetch_asset()`* utility method.
- Avocado's cache directory defined in the configuration will now have the ultimate saying, instead of the dynamic probe for "sensible" cache directories that could end up not respecting user's configurations.

Bug Fixes

- Avocado will now give an error message and exit cleanly, instead of crashing, when the resulting test suite to be executed contains no tests. That can happen, for instance, when invalid references are given along with the `--ignore-missing-references` command line option.
- A crash when running `avocado distro --distro-def-create` has been fixed.

Internal Changes

- All Python files tracked by version control are now checked by linters.
- An *`nrunner Task`* class now has a *`category`*. Only if a task has its category set to `test` (the default) it will be accounted for in the test results.
- *`avocado.utils.process`* now uses `time.monotonic()` to handle timeouts, which is better suited for the task and will survive clock updates.
- The `core.show` configuration item (also available as the `--show` command line option) is now a set of logging streams.
- A *`Task's`* identifier now gets converted to a *`avocado.core.test_id.TestID`* before being handed over to result plugins.
- The `avocado-runner-avocado-instrumented` runner now better handles its own errors (in addition to the exceptions possibly raised by tests).

For more information, please check out the complete Avocado changelog.

87.0 Braveheart

The Avocado team is proud to present another release: Avocado 87.0, AKA "Braveheart", is now available!

Release documentation: Avocado 87.0

Users/Test Writers

- The `avocado assets` command has been expanded and now can purge the cache based on its overall size. To keep 4 GiB of the most recently accessed files, you can run `avocado assets purge --by-overall-limit=4g`. For more information, please refer to the documentation: *Removing by overall cache limit*.
- *`avocado.skipIf()`* and *`avocado.skipUnless()`* now allow the condition to be a callable, to be evaluate much later, and also gives them access to the test class. For more information, please refer to the documentation: *Advanced Conditionals*.
- The presentation of SIMPLE tests have been improved in the sense that they're are now much more configurable. One can now set the `simpletests.status.failure_fields` to configure how the status line shown just after a failed test will look like, and `job.output.testlogs.logfiles` to determine the files that will be shown at the end of the job for failed tests.

Bug Fixes

- The `avocaod.core.safeloader` now supports relative imports with names, meaning that syntax such as `from ..upper import foo` is not properly parsed.

- The nrunner TAP runner now supports/parses large amounts of data, where it would previously crash when buffers were overrun.
- The assets plugin (`avocado assets` command) now returns meaningful exit code on some failures and success situations.

Utility APIs

- The *avocado.utils.partition* utility module now properly keeps track of loop devices and multiple mounts per device.

Internal Changes

- The nrunner message handling code was mostly rewritten, with specific handlers for specific message types. Also, the expected (mandatory and optional) is now documented.
- The *avocado.core.nrunner.Task* identifier is now automatically assigned if one is not explicitly provided.
- The `selftests/check.py` Job API-based script now prints a list of the failed tests at the end of the job.
- The nrunner standalone runners are now on their own directory on the source code tree (`avocado/core/runners`).
- The nrunner base class runner is now an abstract base class.
- The Job's Test suite for the nrunner architecture now contains Runnables instead of Tasks, which are a better fit at that stage. Tasks will be created closer to the execution of the Job. This solves the dilemma of changing a Task identifier, which should be avoided if possible.
- The CI jobs on Cirrus have been expanded to run the selftests in a Fedora based container environment, and a simple smokecheck on Windows.
- A GitHub actions based job was added to the overall CI systems, initially doing the static style/lint checks.
- The selftests have been reorganized into directories for utility modules and plugins. This should, besides making it easier to find the test file for a particular featured based on its type, also facilitate the repo split.
- A number of test status which are not being used were removed, and the current definitions now better match the general style and are documented.
- COPR RPM package check not attempts to install a specific package NVR (name-version-release).
- Many Python code lint improvements, with new checks added.

Misc Changes

- Updated Debian packaging, now based on Pybuild build system

For more information, please check out the complete Avocado changelog.

86.0 The Dig

The Avocado team is proud to present another release: Avocado 86.0, AKA "The Dig", is now available!

Release documentation: Avocado 86.0

Users/Test Writers

- The `avocado assets` command now introduces two new different subcommands: `list` and `purge`. Both allow listing and purging of assets based on their sizes or the number of days since they have been last accessed. For more information please refer to *Managing Assets*.

Bug Fixes

- The `avocado replay` command was calling pre/post plugins twice after a change delegated that responsibility to *avocado.core.job.Job.run()*.
- The `testlog` plugin wasn't able to show the log location for tests executed via the `avocado-runner-avocado-instrumented` runner (for the nrunner architecture') and this is now fixed.
- The `avocado-runner-avocado-instrumented` was producing duplicate log entries because of Avocado's log handler for the *avocado.core.test.Test* was previously configured to propagate the logged messages.

Utility APIs

- The *avocado.utils.cpu* now makes available a mapping of vendor names to the data that matches in `/proc/cpuinfo` on that vendor's CPUs (*avocado.utils.cpu.VENDORS_MAP*). This allows users to have visibility about the logic used to determine the vendor's name, and overwrite it if needed.
- Various documentation improvements for the `avocado.core.multipath` module.

Internal Changes

- The *avocado.core.test.Test* class no longer require to be given an *avocado.core.job.Job* as an argument. This breaks (in a good way) the circular relationship between those, and opens up the possibility for deprecation of legacy code.
- A number of lint checks were added.
- Remove unnecessary compatibility code for Python 3.4 and earlier.

Misc Changes

For more information, please check out the complete Avocado changelog.

85.0 Bacurau

The Avocado team is proud to present another release: Avocado 85.0, AKA "Bacurau", is now available!

Release documentation: Avocado 85.0

Users/Test Writers

- It's now possible to set a timeout (via the `task.timeout.running` configuration option) for nrunner tasks. Effectively this works as an execution timeout for tests run with `--test-runner=nrunner`.
- Users of the asset feature can now register their own assets with a `avocado assets register` command. Then, the registered asset can be used transparently with the *avocado.core.test.Test.fetch_asset()* by its name. This feature helps with tests that need to use assets that can not be downloaded by Avocado itself.

Utility APIs

- The *avocado.utils.cloudinit* module will give a better error message when the system is not capable of creating ISO images, with a solution for resolution.
- The *avocado.utils.vmimage* can now access both current and non-current Fedora versions (which are hosted at different locations).
- The *avocado.utils.network.interfaces* now supports setting configuration for SuSE based systems.

Internal Changes

- The `make link`, useful for developing Avocado with external plugins (say Avocado-VT), became `make develop-external`, and it requires the `AVOCADO_EXTERNAL_PLUGINS_PATH` variable to now be set.
- Various cleanups to the `Makefile` and consolidation into the `setup.py` file.
- A large number additional lint and style checks and fixes were added.
- The "SoB" check (`selftests/signedoff-check.sh`) is now case insensitive.

Misc Changes

For more information, please check out the complete Avocado changelog.

84.0 The Intouchables

The Avocado team is proud to present another release: Avocado 84.0, AKA "The Intouchables", is now available!

Release documentation: Avocado 84.0

Users/Test Writers

- Yaml To Mux plugin now properly supports `None` values.
- Command line options related to results, such as `--json-job-result`, `--tap-job-result`, `--xunit-job-result` and `--html-job-result` are now "proper boolean" options (such as `--disable-json-job-result`, `--disable-xunit-job-result`, etc).
- Pre and Post (job) plugins are now respected in when used with the Job API.
- Support for `avocado list` "extra information" has been restored. This is used in Avocado-VT loaders. They will be removed (again) for good after its usage is deprecated and removed in Avocado-VT.

Bug Fixes

- The `run.dict_variants` setting is now properly registered in an `Init` plugin.
- The nrunner implementation for `exec` and `exec-test` suffered from a limitation to the amount of output it could collect. It was related the size of the PIPE used internally by the Python `subprocess` module. This limitation has been now lifted.
- The nrunner status server can be configured with the maximum buffer size that it uses.
- The `avocado-instrumented` nrunner runner now covers all valid test status.
- The nrunner status server socket is now properly closed, which allows multiple test suites in a job to not conflict.
- The nrunner status server now properly handles the `asyncio` API under Python 3.6.

Utility APIs

- *`avocado.utils.pci`* now accommodates newer slot names.
- *`avocado.utils.memory`* now properly handles the 16GB hugepages with both the HASH and Radix MMU (by removing the check in case Radix is used).
- *`avocado.utils.ssh.Session`* now contains a *`avocado.utils.ssh.Session.cleanup_master()`* method and a **:property:‘avocado.utils.ssh.Session.control_master‘** property.

Internal Changes

- Yaml To Mux documentation updates regarding the data types and null values.
- Release documentation now include the Fedora/EPEL refresh steps.
- BP000 is included and approved.
- The `Makefile` now works on systems such as Fedora 33 because a bad substitution was fixed.
- Only enough `nrunner` workers to deal with the number of tasks in a suite are created and started.
- All nrunner based runners are now checked with a basic interface test.
- The same check script (`selftests/check.py`) is now used run under RPM builds.

Misc Changes

- The contrib scripts to run the KVM unit tests was updated and supports the nrunner and skip exit codes.

For more information, please check out the complete Avocado changelog.

83.0 Crime and Punishment

The Avocado team is proud to present another release: Avocado 83.0, AKA “Crime and Punishment”, is now available!

Release documentation: Avocado 83.0

Users/Test Writers

- All configuration whose namespace start with the `runner.` prefix will be forwarded to runners. This allows centrally managed configuration to be sent to runners executed by different types of spawners.
- The `exec-test` runner now accepts a configuration (`runner.exectest.exitcodes.skip`) that will determine valid exit codes to be treated as `SKIP` test results.
- The Loader based on the YAML Multiplexer has been removed. Users are advised to use Job API and multiple test suites to fulfill similar use cases.
- The GLib plugin has been removed. Users are advised to use TAP test types instead, given that GLib's GTest framework now defaults to producing TAP output.
- A runner for GO, aka golang, tests, compatible with the nrunner, has been introduced.
- The paginator feature is now a boolean style option. To enable it, use `--enable-paginator`.
- The nrunner status server now has two different options regarding its URI. The first one, `--nrunner-status-server-listen` determines the URI in which a status server will listen to. The second one, `--nrunner-status-server-uri` determines where the results will be sent to. This allows status server to be on a different network location than the tasks reporting to it.
- The `avocado-software-manager` command line application now properly returns exit status for failures.
- The Podman spawner now exposes command line options to set the container image (`--spawner-podman-image`) and the Podman binary (`--spawner-podman-bin`) used on an `avocado` invocation.
- Command line options related to results, such as `--json-job-result`, `--tap-job-result`, `--xunit-job-result` and `--html-job-result` currently take a `on` or `off` parameter. That is now deprecated and a warning has been added. Those options will soon become "proper boolean" options (such as `--enable-$type-job-result` and/or `--disable-$type-job-result`).

Bug Fixes

- *avocado.utils.network.interfaces.NetworkInterface.is_admin_link_up()* and *avocado.utils.network.interfaces.NetworkInterface.is_operational_link_up()* now behave properly on interfaces based on bonding.
- The selection of an nrunner based runner, from its Python module name/path has been fixed.
- *avocado.utils.process* utilities that use `sudo` would check for executable permissions on the binary. Many systems will have `sudo` with the executable bit set, but not the readable bit. This is now accounted for.
- The "external runner" feature now works properly when used outside of a `avocado` command line invocation, that is, when used in a script based on the Job APIs.

Utility APIs

- A new module *avocado.utils.dmesg* with utilities for interacting with the kernel ring buffer messages.
- A new utility *avocado.utils.linux.is_selinux_enforcing()* allows quick check of SELinux enforcing status.
- The *avocado.utils.network.interfaces* now support configuration files compatible with SuSE distros.

- *avocado.utils.network.interfaces.NetworkInterface.remove_link()* is a new utility method that allows one to delete a virtual interface link.
- *avocado.utils.network.hosts.Host.get_default_route_interface()* is a new utility method that allows one to get a list of default routes interfaces.
- The *avocado.utils.cpu* library now properly handles s390x z13 family of CPUs.
- The *avocado.utils.pmem* library introduced a number of new utility methods, adding support for daxctl operations such as offline-memory, online-memory and reconfigure-device.

Internal Changes

- The safeloader has been migrated from using `imp` (deprecated) to the more modern `importlib`.
- Instead of using hardcoded `..` to refer to the parent directory, portability was improved by switching to `os.path.pardir()`.
- Runners based on the *avocado.core.nrunner* module, when called on the command line, can now omit the `--kind` parameter, if information can be gathered from the executable name.
- Avocado's `make check` is now based on a Job API script, found at `selftests/check.py`. It combines previously separate set of tests described by multiple command line executions.
- CI "smoke checks" for OS X and Windows have been introduced. This does not mean, however, that Avocado is supported on those platforms.

For more information, please check out the complete Avocado changelog.

82.0 Avengers: Endgame

The Avocado team is proud to present another release: Avocado 82.0, AKA "Avengers: Endgame", is now available!

This release is also an *LTS* Release, with a different *Release Notes* that covers the changes since *69.x LTS*.

Release documentation: Avocado 82.0

Bug Fixes

- Avocado can now find tests on classes that are imported using relative `import` statements with multiple classes. Previously only the first class imported in such a statement was properly processed.
- `avocado run` will now create test suites without an automatic (and usually very verbose) name, but instead without a name, given that there will be only one suite on such jobs. This restores the `avocado run` behavior users expected and are used to.
- Hint files are now being respected again, this time within the context of test suite creation.
- Filtering by tags is now working properly when using the resolver, that is, when using `avocado list --resolver -t $tag -- $reference`.
- Test suites now properly respect the configuration given to them, as opposed to using a configuration composed by the default registered option values.
- Fixed the "elapsed time" produced by the `avocado-instrumented` nrunner runner (that is, `avocado-runner-avocado-instrumented`).
- `avocado --verbose list --resolver -- $reference` has reinstated the presentation of failed resolution information, which is useful for understanding why a test reference was not resolved into a test.

- The "legacy replay plugin", that is, `avocado run --replay`, can now replay a subset of tests based on their status.
- The `avocado diff` command won't crash anymore if given `sysinfo` files with binary content. It will log the issue, and not attempt to present binary differences.
- The HTML report generated by `avocado diff` now runs properly and won't crash.
- The asset fetcher plugin won't crash anymore due to differences in the AST based node attributes.
- *`avocado.utils.process.FDDrainer`* now properly respects the presence and absence of newlines produced when running new processes via *`avocado.utils.process.run()`* and friends. This also fixes tests that relied on the "output check" feature because of missing newlines.
- The `nrunner` plugin will now always display test status in the most natural order, that is, `STARTED` before `PASS` or `FAIL`.
- The `nrunner` plugin will now properly set the job status in case of test failures, resulting in the job (and `avocado run`) exit status to properly signal failures.
- A vast documentation review was performed, with many fixes and improvements.

For more information, please check out the complete Avocado changelog.

81.0 Avengers: Infinity War

The Avocado team is proud to present another release: Avocado 81.0, AKA "Avengers: Infinity War", is now available!

This release introduces many exciting new features. We can't even wait to get to the more specific sections below to talk about some of the highlights:

- A new test runner architecture, previously known as the "N(ext) Runner", now available as the "nrunner" plugin. It currently allows tests to be run in parallel in either processes or into Podman based containers. In the near future, it should include LXC, Kata Containers, QEMU/KVM based virtual machines, etc. It also includes the foundation of a requirement resolution mechanism, in which tests can declare what they need to run (specific Operating System versions, architectures, packages, etc). Expect the Avocado feature set to evolve around this new architecture.
- A fully usable Job API, making most of Avocado's functionalities programmable and highly customizable. Expect the Job API to be declared public soon, that is, to be available as `avocado.Job` (instead of the current *`avocado.core.job.Job`*) just like the Avocado Test API is available at *`avocado.Test`*.
- A new settings API that is tightly linked to the Job API. You can see all the existing configurations at runtime by running `avocado config reference`. To integrate Avocado to an existing project or a CI environment, a custom job with a few configurations will give you a lot of flexibility with very little need to write Python code. Some examples are available at `examples/jobs`.
- Support for multiple test suites in a Job, so that each test suite can be configured differently and independently of each other. Fulfill your use case easily (or let your imagination go wild) and define different runners for different test suites, different parameters to different test suites, or run some test suites locally, while others isolated on containers. Anything that is configurable with the new settings API should be transparently configurable in the context of a test suite (provided the test suite deals with that feature).

This release is also a "pre-LTS release". Development sprint #82 will focus on stabilization, culminating in the 82.0 LTS release.

Release documentation: Avocado 81.0

Users/Test Writers

- The `remote`, `vm` and `docker` runners (which would run jobs on remote, vm and docker containers) were removed, after having being deprecated on version 78.0.
- The "standalone job" feature, in which a test could be run as a standalone job was removed after having being deprecated on version 80.0. The alternative is to use an Avocado Job (using the Job API), with a test defined on the same file, as can be seen on the example file `examples/jobs/passjob_with_test.py` in the source tree.
- The `yaml_to_mux` varianter plugin now attempts to respect the type of the value given to `--mux-inject`. For example, `1` is treated as integer, a value of `1,2` is treated as list a value of `abc` is treated as string, and a value of `1,2,5-10` is treated as list of integers as `1,2,-5` (as it is evaluated by `ast.literal_eval()`).
- Python unittests names are now similar to Avocado's own instrumented tests names, that is, they list the file name as a path, followed by the class and method name. The positive aspect of this change is that that they can be reused again as a test reference (which means you can copy and paste the name, and re-run it).
- The `avocado-runner-*` standalone runners can now look for a suitable Python class to handle a given test kind by using `setuptools` entrypoints.
- For users of the Job API, a "dictionary based" varianter was introduced, that allows you to describe the variations of tests in a test suite directly via a Python dictionary.
- The output produced on the human UI for failed `SIMPLE` tests is now much more straightforward and contains more relevant data.
- Users attempting to use both the `--loader` and the `--external-runner` features will be warned against it, because of its inherent incompatibility with each other.
- A new `avocado replay` command supersedes the `avocado run --replay` command/option.
- The previous experimental command `nlist` has been removed, and its functionality can now be activated by using `avocado list --resolver`. This is part of promotion of the N(ext) Runner architecture from experimental to being integrated into Avocado.

Bug Fixes

- Filtering using tags while listing the tests (but not while running them) was broken on the previous release, and has now been fixed.
- Result event plugins were misbehaving because they were instantiated too early. Now they're loaded later and lazily.
- Failure to load and run the Python unittest with the nrunner's *avocado.core.nrunner.PythonUnittestRunner* depending on the directory it was called from is now fixed.

Utility APIs

- The *avocado.utils.vmimage* now contains an auxiliary documentation (*Supported images*) that lists the exact Operating System names, versions and architectures that have been tested with an Avocado release.
- The *avocado.utils.pmem* library can now check if a given command is supported by the underlying `ndctl` binary.

Internal Changes

- Improvements to the selftests, including a collection of jobs that are run as tests, and a job that tests a good number of Job API features using variants.
- The *avocado.core.settings* is a completely redesigned module, and central to Avocado's future set and Job API. It was present as avocado.core.future.settings on previous versions. All module and plugins have been migrated to the new API.
- The *avocado.utils.software_manager* module has been split into a finer grained directory and module structure.
- Various documentation content improvements, and various build warnings were addressed.
- The avocado_variants attribute is no longer kept in the job configuration as an instance of a *avocado.core.varianter.Varianter*, instead, the configuration for the various variants are kept in the configuration and it's instantiated when needed.
- *avocado.utils.wait* now uses time.monotonic(), which makes it more reliable and less susceptible to errors when the system clock changes while this utility function is running.
- Refactors resulting in more code being shared among Avocado Instrumented and Python unittest handling on the *avocado.core.safeloader* module.
- The *avocado.core.safeloader* module now supports relative imports when attempting to follow imports to find valid classes with tests.
- A new *avocado.core.suite* was introduced, which is the basis of the multiple test suite support in a Job.
- Codeclimate.com is now being used for code coverage services.
- Codeclimate.com now has the bandit plugin enabled, which means that security related alerts are also caught and shown on the analysis.

For more information, please check out the complete Avocado changelog.

80.0 Parasite

The Avocado team is proud to present another release: Avocado 80.0, AKA "Parasite", is now available!

This release (and the previous one) contains mainly internal changes in preparation for the N(ext) Runner architecture to replace the current one, and for the Job API to become a fully supported feature.

It's expected that release 81.0 will be the last release containing major changes before a "pre-LTS release". This way, development sprint #82 will focus on stabilization, culminating in an 82.0 LTS release.

Release documentation: Avocado 80.0

Users/Test Writers

- The Avocado configuration that is logged during a job execution is now the dictionary that is produced by the avocado.core.future.settings module, instead of the configuration file(s) content. This is relevant because this configuration contains the result of everything that affects a job, such as defaults registered by plugins, command line options, all in addition to the configuration file. The goal is to have more consistent behavior and increased job "replayability".
- As explained in the previous point, an Avocado Job is now configured by the configuration set by the avocado.core.future.settings code. Because of the way this module works, options need to be registered, before the content on the config files can be considered valid values for a given option. This has been done for a large

number of Avocado features, but be advised that some configuration may not yet be seen by the job, because of the lack of option registration. We're working to identify and enable complete feature configuration on the next release.

- The "log level" of an Avocado is now defined using the standard Python level names. If you have a custom configuration for this setting, you may need to adjust it (usually only a matter of lowercase to uppercase).
- The runner that will be used in a job can now be defined in the command line (in addition to being previously supported by a configuration entry). If you want to try out the experimental N(ext) Runner, for instance, you should be able to use a command such as `avocado run --test-runner=nrunner /path/to/my/tests`.
- The N(ext) Runner received support for job timeouts, and won't run further tests if the timeout expires.
- The N(ext) Runner now users the same Test ID that the current test runner uses, both in the to-be-removed `avocado nrun` and in the `avocado run --test-runner=nrunner` scenario.
- A brand new command, `jobs` enables users to, among other things to list information about previously executed jobs. A command such as `avocado jobs show` will show the latest job information.
- The "standalone jobs" feature has been **deprecated**. This feature allows users to write a test, that contains a builtin job executor for such a test that allows the test file to be executable. This will be replaced by the Job API, which transparently supports the specification of the **same** file as a source of tests.
- The `avocado run --loaders ?` command to list available loaders has been removed. This command line usage pattern is not consistent across Avocado (or follows the POSIX guidelines), and with the N(ext) Runner architecture depending on the *`avocado.core.resolver`* feature set, one will be able to see the resolvers with the `avocado plugins` command.
- The lower level *`avocado.core.job.Job`*, instead of the `avocado run` command, is now responsible for generating result files, such as the JSON (`results.json`), xUnit (`results.xml`), etc. This allows users of the Job API, as well as users of the `avocado run` command to have results generated as intended.
- The lower level *`avocado.core.job.Job`*, instead of the `avocado run` command, is now also responsible for collecting the job-level system information (AKA `sysinfo`). This allows users of the Job API, as well as users of the `avocado run` command to have this feature available.

Bug Fixes

- The `avocado sysinfo` command reverts to the pre-regression behavior, and now creates a directory following the `sysinfo-$TIMESTAMP` pattern and uses that for the content of the sysinfo output, instead of using the current directory by default.
- An incorrect configuration key name of the `result_upload` command, as part of the "results_upload" plugin, was fixed.
- *`avocado.utils.disk.get_disks()`* now supports all block devices, like multipaths, LVs, etc. Previously it used to return only `/dev/sdX` devices.

Utility APIs

- All of the *`avocado.utils.gdb`* APIs are now back to a working state, with many fixes related to bytes and strings, as well as buffered I/O caching fixes.
- *`avocado.utils.pmem`* now supports the `all` namespace behavior for newer versions of the `ndctl` utility.
- *`avocado.utils.software_manager`* support for the Zypper package manager was improved to support the installation of package build dependencies.

Internal Changes

- Refactors for the *avocado.core.nrunner.BaseRunnerApp* that made the list of commands available as a class attribute avoiding multiple resolutions and string manipulation when a command needs to be resolved.
- The N(ext) Runner received some foundation work for the persistence and retrieval of test generated artifacts. The work takes into consideration that tests may be run disconnected of the the overall test job, and the job can retrieve those at a later time.
- The N(ext) Runner spawner selection is on the `avocado nrun` command is now done by means of the `--spawner=` option that takes a spawner name, instead of the previous `--podman-spawner` option. This logic should be kept on the `avocado run` implementation and allow for new spawners to be used transparently.
- Internal reliability improvements to the N(ext) Runner status server implementation.
- The `avocado nrun` command now respects the `--verbose` command line option, producing less output if it's not given.
- The core sysinfo implementation received cleanups and now makes now distinction between collection at job or test time, and works on both or at any other moment.
- The `avocado.core.future.settings` now allows command line parsers to be added to previously registered options. This allows features that don't require a command line to register options, and plugins that want to control such options with a command line to do so in a decoupled and extensive way.
- A new plugin interface, *avocado.core.plugin_interfaces.Init*, was introduced to allow plugins that need to initialize themselves very early (and automatically) on Avocado. Such plugins have no knowledge of the current configuration, but may use that interface to register new options (among other things).
- An Avocado Job is now run as part of the selftests suite, and more can be added. This is intended to avoid breakage of the Job API as it gets closer to become a supported feature.

For more information, please check out the complete Avocado changelog.

79.0 La vita è bella

The Avocado team is proud to present another release: Avocado 79.0, AKA "La vita è bella", is now available!

This releases contains mainly internal changes in preparation for the N(ext)Runner architecture to replace the current one. It's expected that an LTS release will be done within the next two or three releases, before the switch the current runner architecture is deprecated and removed.

Release documentation: Avocado 79.0

Users/Test Writers

- The Remote, VM and Docker runner plugins have been deprecated. The current implementation would require a major rewrite to be compatible with the new Fabric API (currently uses the Fabric3 API). Also, the N(ext)Runner architecture requires that individual tests be executed in isolated environments (be them local or remote) and the current implementation actually runs a complete Avocado Job so it's not suitable to be reused in the N(ext)Runner.
- The Avocado docstring directives (the ones that go into docstrings and are prefixed with `:avocado:`) now support `requirement` entries. Those will be used as part of the "Requirements Resolver" features, as per *BP002*.

- The `--ignore-missing-references` option, which used to take a `on` or `off` parameter, now takes no parameter. Now, the feature it controls is not enabled unless you supply the command line option (but no `on` or `off` is required).

Bug Fixes

- When using the Job API (with the conventional runner or the N(ext)Runner) the `job.log` ended up being empty empty, but now it produces just like when using the Avocado command line tool. This fix is part of the stabilization effort to declare the Job API as supported soon.
- Fixed an issue with the *avocado.core.safeloader* that would return duplicate tests when both a parent and child class implemented methods with the same name.
- Fixed an issue in the `avocado.core.utils.cpu.cpu_has_flags()` that could cause a crash because of a mixed used of bytes coming from reading `/proc/cpuinfo` and a string based regex.

Utility APIs

- The *avocado.utils.gdb.GDBRemote* implementation of the GDB Remote Protocol now deals with bytes (instead of possibly multibyte strings), more in line with the original protocol specification.
- Users of the `avocado.utils.partition.mount()` can now skip checking if the devices/mountpoints are mounted, which is useful for bind mounts.
- The *avocado.utils.cpu.online()* and *avocado.utils.cpu.offline()* will now check the status of the CPU before attempting to apply a possibly (unnecessary) action.
- The `avocado.utils.software_manager.DnfBackend` now properly implements a `build_dep` functionality, which differs from its parent `avocado.utils.software_manager.YUMBackend`.

Internal Changes

- Optional plugins (shipped by Avocado) will now require a matching Avocado version. This should prevent users from having installation and usage problems with versions mismatch.
- A number of selftests were ported from `unittest.TestCase` to *avocado.Test*, making use of Avocado's features and following a "eat your own dog food" approach.
- A new code style lint check is now enforced, W601, which drops the use of `has_key()` in favor for the `key in` idiom.
- The N(ext)Runner main module, *avocado.core.nrunner*, now has two explicit registries for the two different types of supported runners. The first one, *avocado.core.nrunner.RUNNERS_REGISTRY_STANDALONE_EXECUTABLE* is populated at run time with standalone executable runners available on the system (those named `avocado-runner-$kind`). The second one, *avocado.core.nrunner.RUNNERS_REGISTRY_PYTHON_CLASS* contains Python based runner implementations, which are currently set manually following a class implementation definition (but may be converted to dynamic lookups, such as setuptools' entrypoints in the future).
- The N(ext)Runner example job is one way of checking the progress of its integration into the overall Avocado framework. It's been broken, but it's now back to operation status and being used by the release process in the `jobs/timesensitive.py` job, which has replaced the `make check-full` rule.
- The N(ext)Runner standard runner implementations, say, `avocado-runner-exec-text`, will now create an "output directory" on behalf of the test, and communicate its location via the

`AVOCADO_TEST_OUTPUT_DIR` environment variable. Further work will implement the retrieval and storage of individual tests' output into an organized Avocado Job result structure.

- The `nrun` command, a temporary entrypoint into the N(ext)Runner, will now show a list of tasks that failed with `fail` or `error` results, which can be helpful while debugging Avocado's own selftests failures (or for those brave enough to be running nrun already).
- A number of optional plugins, including `resultsdb`, `results_upload`, `varianter_cit` and `varianter_pict` have been migrated to the "future" settings API, which delivers a consistent configuration between command line, configuration files and Job API usage.
- Documentation improvements on the *Fetching asset files* section, and on the explanation of the current and *The "nrunner" and "legacy runner" test runner* architecture.
- Because the minimum supported Python version was lifted from 3.4 to 3.5 back in Avocado version 74.0, it was possible, but not done before, to upgrade the `asyncio` syntax from the `asyncio.coroutine()` and `yield from` to the more modern `async def` and `await` syntax.
- Python 3.8 is now formally supported, being enabled in the Python package manifest, and being actively tested on our CI.

For more information, please check out the complete Avocado changelog.

78.0 Outbreak

The Avocado team is proud to present another release: Avocado 78.0, AKA "Outbreak", is now available!

Release documentation: Avocado 78.0

Users/Test Writers

- The HTML plugin now produces reports with resizeable columns and standardized tooltips (besides some internal cleanups).
- The `avocado assets fetch` command now accepts a `--ignore-errors` option that returns exit code 0 even when some of the assets could not be fetched. This is useful in some unattended executions such as CI environments, in which the `avocado assets fetch` is used in conjunction with the canceling of tests that depend on assets not found. Without this option, an entire CI job can be stopped at the initial failure.
- Avocado now supports "hint files" that can tweak how the Avocado resolver will recognize tests. This is useful for projects making use of Avocado as a test runner, and it can allow complete integration with a simple configuration file in a project repository. For more information check out the documentation about *The hint files*.
- The experimental N(ext) Runner now allows users to set the number of parallel tasks with the `--parallel-tasks` command line option (or by setting the `parallel_tasks` configuration under the `nrun` section). The default value is still the same (twice the number of CPUs, minus one).
- The experimental N(ext) Runner now checks the status of tasks right after spawning them. This can serve as an indication if a task crashes too soon. Users will now see a "<task> spawned and alive" on most cases.
- The experimental N(ext) Runner now provides a container based execution of tasks with command line option `--podman-spawner`. While this is not yet intended for general use, it serves as an early technology preview of the multiple test isolation strategies that will be fully supported by the N(ext) Runner.
- The `avocado vmimage get` command now returns a proper error exit code when it fails to retrieve the requested image.

Bug Fixes

- The *avocado.utils.asset* used to produce an empty string when the asset name parameter was not a full URL, resulting in a broken hash value.
- The *avocado.utils.asset* could fail trying to remove a temporary file that may not ever have been created.

Utility APIs

- The CentOS provider of the *avocado.utils.vmimage* module now supports the location and image file names for version 8.
- The OpenSUSE provider of the *avocado.utils.vmimage* module now returns the pure version numbers, instead of the ones containing the `Leap_` prefixes.
- The Debian provider of the the *avocado.utils.vmimage* module now properly matches the version numbers.
- The Ubuntu provider of the the *avocado.utils.vmimage* module now doesn't attempt to convert versions into numbers, which could result in lost digits (10.40 would become 10.4).
- The *avocado.utils.network.interfaces* module now supports different types output produced by `iproute`.
- The *avocado.utils.ssh.Session.cmd()* method now allows users to ignore the exit status of the command with the `ignore_status` parameter.
- The *avocado.utils.cpu* changed how it identifies CPU vendors, architectures and families, making those more consistent across the board.

Internal Changes

- The experimental N(ext) Runner now produces less ambiguous state messages, with a dedicated `result` field on the final state message, instead of reusing the `status` field.
- A "release job" was introduced to be run in addition to the other selftests before a release is cut. It currently includes a complete coverage of all the *:mod:'avocado.utils.vmimage* providers, amounting to almost 200 test variations.
- The `loader_yaml` and `html` plugins were migrated to the new (future) settings API.

For more information, please check out the complete Avocado changelog.

77.0 The Hangover

The Avocado team is proud to present another release: Avocado 77.0, AKA "The Hangover", is now available!

Release documentation: Avocado 77.0

Users/Test Writers

- The *avocado.Test.fetch_asset* method now has two new parameters: `find_only` and `cancel_on_missing`. These can be combined to cancel tests if the asset is missing after a download attempt (`find_only=False`) or only if it's present in the local system without a download having been attempted

during the test (`find_only=True`). This can bring better determinism for tests that would download sizable assets, and/or allow test jobs to be executable in offline environments.

- The `avocado-software-manager` script, a frontend to the *avocado.utils.software_manager* module, now produces output as expected from a script.
- The `multiplex` command, an alias to `variants`, has been deprecated for a long time, and has now finally been removed.

Bug Fixes

- When a dry-run is executed, by passing the `--dry-run` command line option, the proper file name of the test will be shown, instead of the file implementing the "fake" `avocado.core.test.DryRun` class.
- Users of *avocado.utils.ssh.Session* as a context manager, would have all the exceptions captured and suppressed because of a buggy `__exit__` implementation.

Utility APIs

- The new *avocado.utils.pmem* module provides an interface for manage persistent memory. It allows for creating, deleting, enabling, disabling and re-configuring both namespaces and regions depending on supported hardware. It wraps the features present on the `ndctl` and `daxctl` binaries.
- The new `avocado.utils.ssh.Session.get_raw_ssh_commands()` allows access to the generated (local) commands, which could be used for advanced use cases, such as running multiple (remote) commands in a test. See the `examples/apis/utils/ssh.py` for an example.
- The *avocado.utils.network* module received a complete overhaul, and provides features for getting, checking and setting network information from local and even remote hosts.
- Better documentation for the *avocado.utils.ssh*, *avocado.utils.cloudinit*, *avocado.utils.service* and other modules.

Internal Changes

- The foundation of the *BP001* has been implemented, in the form of the `avocado.core.future.settings` and by adjusting pretty much all of Avocado's code to make use of it. In the near future, this is going to replace *avocado.core.settings*.
- It's now easier to write a runner script that extends the types of runnables supported by the N(ext) Runner. For an example, please refer to `examples/nrunner/runners/avocado-runner-foo`.
- Many more refactors on the *avocado.utils.asset* module.

For more information, please check out the complete Avocado changelog.

76.0 Hotel Mumbai

The Avocado team is proud to present another release: Avocado 76.0, AKA "Hotel Mumbai", is now available!

Release documentation: Avocado 76.0

Users/Test Writers

- The decorators *avocado.skip()*, *avocado.skipIf()* and *avocado.skipUnless()* can now be used to decorate entire classes, resulting in all its tests getting skipped if/when the condition given is satisfied.
- A TAP capable test runner for the N(ext) Runner has been introduced and is available as `avocado-runner-tap`. Paired with the resolver implementation introduced in the previous release, this allows the `avocado nrun` command to find and execute tests that produce `TAP` compatible output.
- Avocado's *avocado.utils.software_manager* functionality is now also made available as the `avocado-software-manager` command line tool.
- The `sysinfo` collection now logs a much clearer message when a command is not found and thus can not have its output collected.
- Documentation improvements and fixes in guide sections and utility libraries.
- A second blueprint, *BP002*, was approved (and committed) to Avocado. It's about a proposal about a "Requirements resolver", that should give tests automatic resolution of various types of requirements they may need to run.

Bug Fixes

- The N(ext) Runner will now properly escape Runnable arguments that start with a dash when generating a command to execute a runner, avoiding the runner itself to try to parse it as an option to itself.
- The Journal plugin will now only perform its test status journaling tasks if the `--journal` option is given, as it was originally intended.
- The HTML plugin has been pinned to the jinja2 package version compatible with Python 3.5 and later.

Utility APIs

- The *avocado.utils.kernel.KernelBuild.build()* now allows the definition of the number of jobs, using semantics very similar to the one used by GNU make itself. That means one should be careful when using `None`, as it means no limit to the number of parallel jobs.

Internal Changes

- Workarounds on Travis CI for caching failures on s390x and aarch64.
- Many refactors on the *avocado.utils.asset* module
- Multiple refactors on the N(ext) Runner code

For more information, please check out the complete Avocado changelog.

Changes expected for the next release (77.0)

We are working hard to use a good name convention related to configuration options (either via command-line or via configuration file). Because of that, to keep consistency, some options are going to be changed.

Beginning with this release (76.0), users will notice a few warnings (i.e `FutureWarning`) messages on the STDERR. Those are early warnings of changes that will be introduced soon, because of the work mentioned before. On the next release (77.0), it's expected that compatibility will be affected.

In the end, we will have an improved configuration module, that will handle both command line and configuration options. This intends to deliver a better way to register and to retrieve configuration options. Also, soon we will provide better documentation and a complete template config file, covering all options supported.

For more information, please visit the *BP001*.

75.1 Voyage to the Prehistoric Planet (minor release)

The Avocado team is proud to present another release: Avocado 75.1, AKA "Voyage to the Prehistoric Planet", is now available!

Release documentation: Avocado 75.1

Changes from 75.0 to 75.1

- The file used as the project description, `README.rst` was slightly changed to only contain reStructuredText content, and be accepted into the PyPI repository.
- The missing 75.0 release notes document was added.
- A missing slash from the readthedocs.org badge URL was added.

75.0 Release Changes

The following are the original changes part of the 75.0 release.

Users/Test Writers

- The very first blueprint was approved (and committed) to Avocado. It's about a "Configuration by convention" proposal, which will positively impact users deploying and using Avocado, and will end up making the Job API have a much better usability.
- Warnings for the deprecation of some options, as determined by the design decisions on the "Configuration by convention" blueprint have been added to the command line tool. Users should pay attention to not rely on the content on `STDERR`, as it may contain those warnings.
- The `jsonresult` plugin, that generated a JSON representation of the job results, added `warn` and `interrupt` fields containing counters for the tests that ended with WARN and INTERRUPTED status, respectively.
- The still experimental "N(ext) Runner" has introduced an initial integration with the Avocado Job. Users running `avocado plugins` will see a new entry under "Plugins that run test suites on a job (runners)". The only way to activate this runner right now is to run a custom job such as the one in `examples/job/nrunner.py`.

Bug Fixes

- The YAML Loader did not behave correctly when a `None` reference was given to it. It would previously try to open a file named `None`.

Utility APIs

- A previously deprecated function called `thin_lv_created` was removed from the *avocado.utils.lv_utils* module.
- *avocado.utils.configure_network.is_interface_link_up()* is a new utility function that returns, quite obviously, whether an interface link is up.

Internal Changes

- Inspektor was replaced with a PyLint for the lint checks due to parallel execution errors that were plaguing CI, mostly on non-x86 architectures.
- The *avocado.utils.asset* received a number of refactors, in preparation for some major changes expected for the next releases.
- The *avocado.utils.cloudinit* selftest now queries the allocated port from the created socket itself, which removes a race condition that existed previously and caused intermittent test failures.
- A test for the sysinfo content on the HTML report was added, removing the need for the manual test on the release test plan.
- The deployment selftests have been reorganized, and now are based on Ansible roles (and other best practices).
- The handling of a "Job results directory" resolution, based either on its ID (partial or complete) or path has been improved, and has internally been moved from the *avocado.core.jobdata* to *avocado.core.data_dir*.

For more information, please check out the complete Avocado changelog.

75.0 Voyage to the Prehistoric Planet

The Avocado team is proud to present another release: Avocado 75.0, AKA "Voyage to the Prehistoric Planet", is now available!

Release documentation: Avocado 75.0

Users/Test Writers

- The very first blueprint was approved (and committed) to Avocado. It's about a "Configuration by convention" proposal, which will positively impact users deploying and using Avocado, and will end up making the Job API have a much better usability.
- Warnings for the deprecation of some options, as determined by the design decisions on the "Configuration by convention" blueprint have been added to the command line tool. Users should pay attention to not rely on the content on `STDERR`, as it may contain those warnings.
- The `jsonresult` plugin, that generated a JSON representation of the job results, added `warn` and `interrupt` fields containing counters for the tests that ended with WARN and INTERRUPTED status, respectively.
- The still experimental "N(ext) Runner" has introduced an initial integration with the Avocado Job. Users running `avocado plugins` will see a new entry under "Plugins that run test suites on a job (runners)". The only way to activate this runner right now is to run a custom job such as the one in `examples/job/nrunner.py`.

Bug Fixes

- The YAML Loader did not behave correctly when a `None` reference was given to it. It would previously try to open a file named `None`.

Utility APIs

- A previously deprecated function called `thin_lv_created` was removed from the *avocado.utils.lv_utils* module.
- *avocado.utils.configure_network.is_interface_link_up()* is a new utility function that returns, quite obviously, whether an interface link is up.

Internal Changes

- Inspektor was replaced with a PyLint for the lint checks due to parallel execution errors that were plaguing CI, mostly on non-x86 architectures.
- The *avocado.utils.asset* received a number of refactors, in preparation for some major changes expected for the next releases.
- The *avocado.utils.cloudinit* selftest now queries the allocated port from the created socket itself, which removes a race condition that existed previously and caused intermittent test failures.
- A test for the sysinfo content on the HTML report was added, removing the need for the manual test on the release test plan.
- The deployment selftests have been reorganized, and now are based on Ansible roles (and other best practices).
- The handling of a "Job results directory" resolution, based either on its ID (partial or complete) or path has been improved, and has internally been moved from the *avocado.core.jobdata* to *avocado.core.data_dir*.

For more information, please check out the complete Avocado changelog.

74.0 Home Alone

The Avocado team is proud to present another release: Avocado 74.0, AKA "Home Alone", is now available!

Release documentation: Avocado 74.0

Users/Test Writers

- A new test type, `TAP` has been introduced along with a new loader and resolver. With a `TAP` test, it's possible to execute a binary or script, similar to a `SIMPLE` test, and part its Test Anything Protocol output to determine the test status.
- It's now possible to enforce colored or non-colored output, no matter if the output is a terminal or not. The configuration item `color` was introduced in the `runner.output` section, and recognize the values `auto`, `always` or `never`.

Bug Fixes

- The `safeloader` mechanism that discovers both Avocado's Python based `INSTRUMENTED` tests, and Python's native unittests, would fail to find any tests if any of the classes on a given file contained references to a module that was not on a parent location. Now, the `safeloader` code will continue the discovery process, ignoring the modules that were not found at parent locations.

Utility APIs

- *avocado.utils.kernel* received a number of fixes and cleanups, and also new features. It's now possible to configure the kernel for multiple targets, and also set kernel configurations at configuration time without manually touching the kernel configuration files. It also introduced the *avocado.utils.kernel.KernelBuild.vmlinux()* property, allowing users to access that image if it was built.
- *avocado.utils.network* utilities `avocado.utils.network.ping_check()` and `avocado.utils.network.set_mtu_host()` now are plain functions, instead of methods of a class that shared nothing between them.
- New functions such as *avocado.utils.multipath.add_path()*, :func:*avocado.utils.multipath.remove_path()* *avocado.utils.multipath.get_mpath_status()* and *avocado.utils.multipath.suspend_mpath()* have been introduced :func:to the *avocado.utils.multipath* module.
- The *avocado.utils.vmimage* module will not try to create snapshot images when it's not needed, acting lazily in that regard. It now provides a different method for download-only operations, *avocado.utils.vmimage.Image.download()* that returns the base image location. The behavior of the *avocado.utils.vmimage.Image.get()* method is unchanged in the sense that it returns the path of a snapshot image.

Internal Changes

- A PyLint configuration file was added to the tree, facilitating the use of the standard Python linter when developing Avocado in IDEs that support this feature.

For more information, please check out the complete Avocado changelog.

73.0 Pulp Fiction

The Avocado team is proud to present another release: Avocado 73.0, AKA "Pulp Fiction", is now available!

Release documentation: Avocado 73.0

Users/Test Writers

- `INSTRUMENTED` tests using the *avocado.core.test.Test.fetch_asset()* can take advantage of plugins that will attempt to download (and cache) assets before the test execution. This should make the overall test execution more reliable, and give better test execution times as the download time will be excluded. Users can also manually execute the `avocado assets` command to manually fetch assets from tests.
- The still experimental "N(ext) Runner" support for Avocado Instrumented tests is more complete and supports tag filtering and passing tags to the tests.

- A new architecture for "finding" tests has been introduced as an alternative to the *avocado.core.loader* code. It's based around the *avocado.core.resolver*, and it's currently used in the still experimental "N(ext) Runner". It currently supports tests of the following types: `avocado-instrumented`, `exec-test`, `glib`, `golang`, `python-unittest` and `robot`. You can experiment it by running `avocado nlist`, similarly to how `avocado list` is used.
- Avocado `sysinfo` feature file will now work out of the box on `pip` based installations. Previously, it would require configuration files tweaks to adjust installation paths.
- A massive documentation overhaul, now designed around guides to different target audiences. The "User's Guide", "Test Writer's Guide" and "Contributor's Guide" can be easily found as first lever sections contain curated content for those audiences.

Bug Fixes

- Content supposed to be UI only could leak into TAP files, making them invalid.
- Avocado's `sysinfo` feature will now run commands without a shell, resulting in more proper captured output, without shell related content.
- *avocado.utils.process.SubProcess.send_signal()* will now send a signal to itself correctly even when using `sudo` mode.

Utility APIs

- The *avocado.utils.vmimage* library now allows a user to define the `qemu-img` binary that will be used for creating snapshot images via the *avocado.utils.vmimage.QEMU_IMG* variable.
- The *avocado.utils.configure_network* module introduced a number of utilities, including MTU configuration support, a method for validating network among peers, IPv6 support, etc.
- The *avocado.utils.configure_network.set_ip()* function now supports different interface types through a `interface_type` parameter, while still defaulting to `Ethernet`.

Internal Changes

- Package support for Enterprise Linux 8.
- Increased CI coverage, having tests now run on four different hardware architectures: amd64 (x86_64), arm64 (aarch64), ppc64le and s390x.
- Packit support adding extended CI coverage, with RPM packages being built for Pull Requests and results shown on GitHub.
- Pylint checks for w0703 were enabled.
- Runners, such as the remote runner, vm runner, docker runner, and the default local runner now conform to a "runner" interface and can be seen as proper plugins with `avocado plugins`.
- Avocado's configuration parser will now treat values with relative paths as a special value, and evaluate their content in relation to the Python's distribution directory where Avocado is installed.

For more information, please check out the complete Avocado changelog.

72.0 Once upon a time in Holywood

The Avocado team is proud to present another release: Avocado 70.0, AKA "Once upon a time in Holywood", is now available!

Release documentation: Avocado 72.0

Users/Test Writers

- The new `vmimage` command allows a user to list the virtual machine images downloaded by means of *avocado.utils.vmimage* or download new images via the `avocado vmimage get` command.
- The tags feature (see *Categorizing tests*) now supports an extended character set, adding `.` and `-` to the allowed characters. A tag such as `:avocado: tags=machine:s390-ccw-virtio` is now valid.
- The still experimental "N(ext) Runner", introduced on version 71.0, can now run most Avocado Instrumented tests, and possibly any test who implements a matching `avocado-runner-$(TEST_TYPE)` script that conforms to the expected interface.

Bug Fixes

- A bug introduced in version 71.0 rendered *avocado.utils.archive* incapable of handling LZMA (also known as `xz`) archives was fixed.
- A Python 3 (bytes versus text) related issue with *avocado.utils.cpu.get_cpu_vendor_name()* has been fixed.

Utility APIs

- *avocado.utils.ssh* now allows password based authentication, in addition to public key based authentication.
- *avocado.utils.path.usable_ro_dir()* will no longer create a directory, but will just check for its existence and the right level of access.
- *avocado.utils.archive.compress()* and *avocado.utils.archive.uncompress()* and now supports LZMA compressed files transparently.
- The *avocado.utils.vmimage* now has providers for the CirrOS cloud images.

Internal Changes

- Package build fixes for Fedora 31 and Fedora 32.
- Increased test coverage of mux-suite and the yaml-loader features.
- A number of pylint checks were added, including w0201, w1505, w1509, w0402 and w1113.

For more information, please check out the complete Avocado changelog.

71.0 Downton Abbey

The Avocado team is proud to present another release: Avocado 70.0, AKA "Downton Abbey", is now available!

Release documentation: Avocado 71.0

Users/Test Writers

- Avocado can now run on systems with nothing but Python 3 (and "quasi-standard-library" module `setuptools`). This means that it won't require extra packages, and should be easier to deploy on containers, embedded systems, etc. Optional plugins may have additional requirements.
- A new and still experimental test runner implementation, known as "N(ext) Runner" has been introduced. It brings a number of different concepts, increasing the decoupling between a test (and its runner) and the job. For more information, please refer to *the early documentation <nrunner>*.
- The new *`avocado.cancel_on()`* decorator has been added to the Test APIs, allowing you to define the conditions for a test to be considered canceled. See one example *here*.
- The `glib` plugin got a configuration option its safe/unsafe operation, that is, whether it will execute binaries in an attempt to find the whole list of tests. Look for the `glib.conf` shipped with the plugin to enable the unsafe mode.
- Avocado can now use tags inside Python Unittests, and not only on its own Instrumented tests. It's expected that other forms or providing tags for other types of tests will also be introduced in the near future.
- The HTML report will now show, as a handy pop-up, the contents of the test whiteboard. If you set, say, performance metrics there, you'll able to see straight from the report.
- The HTML report now has filtering support by test status, and can show all records in the table.
- The `avocado.utils.runtime` module, a badly designed mechanism for sharing Avocado runtime settings with the utility libraries, has been removed.
- The test runner feature that would allow binaries to be run transparently inside GDB was removed. The reason for dropping such a feature have to do with how it limits the test runner to run one test at a time, and the use of the `avocado.utils.runtime` mechanism, also removed.
- Initial examples for writing custom jobs, using the so called Job API, have been added to `examples/jobs`. These APIs are still non-public (under core), but they're supposed to become public and supported soon.
- By means of a new plugin (`merge_files`, of type `job.prepost`), when using the *output check record* features, duplicate files created by different tests/variants will be consolidated into unique files.

Bug Fixes

- The HTML plugin now correctly shows the date for tests that were never executed because of interrupted jobs.
- A temporarily workaround for a stack overflow problem in Python 3.7 has been addressed.
- The pict plugin (a varianter implementation) now properly yields the variants paths as a list.
- A Python 3 related fix to mod:*avocado.utils.software_manager*, that was using Python 2 `next` on `get_source`.
- A Python 3 related fix to the docker plugin, that wasn't caught earlier.

Utility APIs

- *`avocado.utils.partition`* now allows `mkfs` and `mount` flags to be set.
- *`avocado.utils.cpu.get_cpu_vendor_name()`* now returns the CPU vendor name for POWER9.
- *`avocado.utils.asset`* now allows a given location, as well as a list, to be given, simplifying the most common use case.

- *avocado.utils.process.SubProcess.stop()* now supports setting a timeout. Please refer to the documentation for the important details on its behavior.
- *avocado.utils.memory* now properly handles hugepages for POWER platform.

Internal Changes

- Removal of the `stevedore` library dependency (previously used for the dispatcher/plugins infrastructure).
- `make check` now runs selftests using the experimental N(ext) Runner.
- Formal support for Python 3.7, which is now on our CI checks, documentation and module information.
- The Yaml to Mux plugin now uses a safe version of the Yaml loader, so that the execution of arbitrary Python code from Yaml input is now no longer possible.
- Codecov coverage reports for have been enabled for Avocado, and can be seen on every pull request.
- New tests have been added to many of the optional plugins.
- Various pylint compliance improvements, including w0231, w0235, w0706, w0715 and w0221.
- Avocado's selftests now use tempfile.TemporaryDirectory instead of mkdtemp and shutil.rmtree.
- *avocado.core.job.Job* instantiation now takes a `config` dictionary parameter, instead of a `argparse.Namespace` instance, and keeps it in a `config` attribute.
- *avocado.core.job.Job* instances don't have a `references` attribute anymore. That information is available in the `config` attribute, that is, `myjob.config['references']`.
- Basic checks for Fedora and RHEL 8 using Cirrus CI have been added, and will be shown on every pull request.

For more information, please check out the complete Avocado changelog.

70.0 The Man with the Golden Gun

The Avocado team is proud to present another release: Avocado 70.0, AKA "The Man with the Golden Gun", is now available!

Release documentation: Avocado 70.0

Users/Test Writers

- A completely new implementation of the CIT Varianter plugin implementation, now with support for constraints. Refer to *CIT Varianter Plugin* for more information.
- Python 2 support has been removed. Support Python versions include 3.4, 3.5, 3.6 and 3.7. An effort to support Python 3.8 is also underway. If you require Python 2 support, the 69.0 LTS series (currently at version 69.1) should be used. For more information on what a LTS release means, please read *RFC: Long Term Stability*.
- Improved safeloader support for Python unittests, including support for finding test classes that use multiple inheritance. As an example, Avocado's safeloader is now able to properly find all of its own tests (around 700 of them).
- Removal of old and redundant command line options, such as `--silent` and `--show-job-log` in favor of `--show=none` and `--show=test`, respectively.
- Job result categorization support, by means of the `--job-category` option to the `run` command, allows a user to create an easy to find directory, within the job results directory, for a given type of executed jobs.

Bug Fixes

- Log files could have been saved as "hidden" files files (`.INFO`, `.DEBUG`, `.WARN`, `.ERROR`) because the root logger's name is an empty string. Now, those are saved with a `log` prefix if one is not given.
- The second time Avocado crashes, a "crash" directory is created to hold the backtrace. On a subsequent crash, if the directory already exists, an exception would be raised for the failed attempted to create an existing directory, confusing users on the nature of the crash. Now a proper handling for the possibly existing directory is in place.
- The CIT Varianter plugin was returning variants in an invalid form to the runner. This caused the plugin to fail when actually used to run tests. A functional test has also been aded to avoid a regression here.
- The *avocado.utils.distro* module now properly detects RHEL 8 systems.
- The safeloader would fail to identify Python module names when a relative import was used. This means that the experience with `$ avocado list` and `$ avocado run` would suffer when trying to list and run tests that either directly or indirectly imported modules containing a relative import such as `from . import foo`.
- The *avocado.utils.vmimage* can now find Fedora images for s390x.
- The *avocado.utils.vmimage* now properly makes use of the build option.
- `avocado list` will now show the contents of the "key:val" tags.
- The Avocado test loader will correctly apply filters with multiple "key:val" tags.

Utility APIs

- Two simple utility APIs, *avocado.utils.genio.append_file()* and *avocado.utils.genio.append_one_line()* have been added to the benefit of some *avocado-mist-tests <https://github.com/avocado-framework-tests/avocado-misc-tests>*.
- The new *avocado.utils.datadrainer* provide an easy way to read from and write to various input/output sources without blocking a test (by spawning a thread for that).
- The new *avocado.utils.diff_validator* can help test writers to make sure that given changes have been applied to files.

Internal Changes

- Removal of the `six` library dependency (previously used for simultaneous Python 2 and 3 support).
- Removal of the `sphinx` module and local "build doc" test, in favor of increased reliance on readthedocs.org.
- Removal of the `pillow` module used when running very simple example tests as a selftests, which in reality added very little value.
- All selftests are now either Python unittests or standalone executables scripts that can be run with Avocado itself natively. This was done (also) because of the N(ext) Runner proposal.
- Build improvements and fixes, supporting packaging for Fedora 30 and beyond.

For more information, please check out the complete Avocado changelog.

69.0 The King's Choice

The Avocado team is proud to present another LTS (Long Term Stability) release: Avocado 69.0, AKA "The King's Choice", is now available!

Release documentation: Avocado 69.0

LTS Release

For more information on what a LTS release means, please read *RFC: Long Term Stability*.

For a complete list of changes from the last LTS release to this one, please refer to *69.0 LTS*.

The major changes introduced on this version (when compared to 68.0) are listed below, roughly categorized into major topics and intended audience:

Bug Fixes

- INSTRUMENTED tests would not send content to the test's individual log files when the logger name was not `avocado.test`. Now tests can declare and use their own logger (with their own names) and the content will be directed to the test's own log files.
- The JSON result plugin would store empty failure data as a string representation of Python's `None`, instead of JSON's own `null`. Because the JSON file is used internally between the local and remote runners, the Human UI would show a `"None"` "failure" reason when tests succeeded.

Internal Changes

- Document the Copr repo, including the repository build status for our packages on our README and Getting Started pages.
- Documentation improvements with a more accurate list of available plugins.
- Deployment checks for a setup of Avocado and Avocado-VT installed via PIP from the latest sources were added.
- Deployment checks for a setup of Avocado and Avocado-VT installed via the Copr repository packages were added.
- Reliability improvements for the unittest `selftests.test_utils.ProcessTest.test_process_start`.
- Skip the unittest `selftests.test_utils_network` when the Python netifaces library is not available.

For more information, please check out the complete Avocado changelog.

68.0 The Marvelous Mrs. Maisel

The Avocado team is proud to present another release: Avocado version 68.0, AKA "The Marvelous Mrs. Maisel", is now available!

Release documentation: Avocado 68.0

The major changes introduced on this version are listed below, roughly categorized into major topics and intended audience:

Users/Test Writers

- The Avocado test loader, which does not load or execute Python source code that may contain tests for security reasons, now operates in a way much more similar to the standard Python object inheritance model. Before, classes containing tests that would not directly inherit from *avocado.Test* would require a docstring statement (either `:avocado: enable` or `:avocado: recursive`). This is not necessary for most users anymore, as the recursive detection is now the default behavior.
- The xUnit plugin now should produce output that is more compatible with other implementations, specifically newer Jenkin's as well as Ant and Maven. The specific change was to format the time field with 3 decimal places.
- A new *avocado.utils.cpu.get_pid_cpus()* utility function allows one to get all the CPUs being used by a given process and its threads.
- The *avocado.utils.process* module now exposes the `timeout` parameter to users of the *avocado.utils.process.SubProcess* class. It allows users to define a timeout, and the type of signal that will be used to attempt to kill the process after the timeout is reached.
- The location of the Avocado configuration files can now be influenced by third parties by means of a new plugin.
- The configuration files that have been effectively parsed are now displayed as part of `avocado config` command output.

Bug Fixes

- A bug that would crash Avocado while listing simple or "broken" tests has been fixed.
- A bug on the asset fetcher cache system would prevent files with the same name, but from different locations, to be kept in the cache at the same, causing overwrites and new download attempts.
- The robot framework plugin would print errors and warnings to the console, confusing Avocado users as to the origin and reason for those messages. The plugin will now disable all robot framework logging operations on the console.
- Test directories won't be silently created on system wide locations any longer, as this is a packaging and/or installation step, and not an Avocado test runner runtime step.
- The *avocado.utils.ssh* module would not properly establish master sessions due to the lack of a `ControlPath` option.
- A possible infinite hang of the test runner, due to a miscalculation of the timeout, was fixed.
- The `avocado.utils.archive.extract_lzma()` now properly opens files in binary mode.

Internal Changes

- An optimization and robustness improvement on the func:*avocado.utils.memory.read_from_meminfo* was added.
- The required version of the PyYAML library has been updated to 4.2b2 because of CVE-2017-18342. Even though Avocado doesn't use the exact piece of code that was subject to the vulnerability, it's better to be on the safe side.
- Rules to allow a SRPM (and consequently RPM) packages to be built on the COPR build service have been added.
- The documentation on the `--mux-inject` feature and command line option has been improved, showing the behavior of the `path` component when inserting content and fetching parameters later on.

- A new test was added to cover the behavior of unittest's assertRaises when used in an Avocado test was added.
- A fix was added to selftests/unit/test_utils_vmimage.py to not depend or assume a given host architecture.
- The *avocado.utils.ssh.Session* will now perform a more extensive check for an usable master connection, instead of relying on just the SSH process status code.
- The upstream and Fedora versions of the SPEC files are now virtually in sync.
- Building the the docs as part of the selftests now works on Python 3.
- The Avocado test loader, when returning Python unittest results, will now return a proper ordered dictionary that matches the order in which they were found on the source code files.

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.

67.0 A Beautiful Mind

The Avocado team is proud to present another release: Avocado version 67.0, AKA "A Beautiful Mind", is now available!

Release documentation: Avocado 67.0

The major changes introduced on this version are listed below, roughly categorized into major topics and intended audience:

Users/Test Writers

- The *avocado.utils.archive* module now supports the handling of gzip files that are not compressed tarballs.
- The xunit output now names the job after the Avocado job results directory. This should make the correlation of results displayed in UIs such as Jenkins and the complete Avocado results much easier.
- A number of improvements to the *avocado.utils.lv_utils* module now allows users to choose if they want or not to use ramdisks, and allows for a more concise experience when creating Thin Provisioning LVs.
- New utility function in the *avocado.utils.genio* that allows for easy matching of patterns in files. See `avocado.utils.is_pattern_in_file()` for more information.
- New utility functions are available to deal with filesystems, such as *avocado.utils.disk.get_available_filesystems()* and *avocado.utils.disk.get_filesystem_type()*.
- The test filtering mechanism using tags now support "key:val" assignments for further categorization. See *Python unittest Compatibility Limitations And Caveats* for more details.
- The *Avocado Test class* now exposes the *tags* to the test. The test may use that information, for instance, to decide on default behavior.
- The *avocado.utils.process.kill_process_tree()* now supports waiting a given timeout, and returns the PIDs of all process that had signals delivered to.

- The `avocado.utils.network.is_port_free()` utility function now supports IPv6 in addition to IPv4, as well as UDP in addition to TCP.

Bug Fixes

- Fixed the lack of initialization of the logging system that would, on some unittests, cause an infinity recursion.

Internal Changes

- The template engine that powers the HTML report has been replaced, and now jinja2 is being used and pystache has been dropped. The reason is the lack of activity in the pystache project, and lack of Python 3.7 support.
- A number of refactors and improvements on the selftests have increased the number of test to the 650 mark.
- The mechanism used to list selftests to be run is now the same when running tests in serial or in parallel mode, and is exposed in the `selftests/list` script.

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.

66.0 Les Misérables

The Avocado team is proud to present another release: Avocado version 66.0, AKA "Les Misérables", is now available!

Release documentation: Avocado 66.0

The major changes introduced on this version are listed below, roughly categorized into major topics and intended audience:

Users/Test Writers

- The *`avocado.utils.vmimage`* library got a provider implementation for OpenSUSE. The limitation is that it tracks the general releases, and not the rolling releases (called Tumbleweed).
- Users of the *`avocado.utils.kernel`* module can now properly specify the base URL from which to download the kernel sources.

Bug Fixes

- The YAML to Mux plugins now properly deals with text encoding and work as intended on Python 3. These were the last existing tests that were being skipped in the Python 3 environment, so now all existing tests run equally on all Python versions.

Internal Changes

- Development environments now default to Python 3, that is, if you download the Avocado source code, and run `make develop` or related targets, Python 3 will be favored if available on your system. You can force the Python interpreter version with `make PYTHON=/path/to/python develop`.
- The *avocado.utils.partition* implementation for the `/etc/mtab` lock is now based on the *avocado.utils.filelock* module.

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/RbIV6bDp/1442-sprint-theme

65.0 Back to the Future

The Avocado team is proud to present another release: Avocado version 65.0, AKA "Back to the Future", is now available!

Release documentation: Avocado 65.0

The major changes introduced on this version are listed below, roughly categorized into major topics and intended audience:

Users/Test Writers

- A new utility library, *avocado.utils.ssh*, has been introduced. It's a simple wrapper around the OpenSSH client utilities (your regular `/usr/bin/ssh`) and allows a connection/session to be easily established, and commands to be executed on the remote endpoint using that previously established connection.
- Passing parameters to tests is now possible directly on the Avocado command line, without the use of any varianter plugin. In fact, when using variants, these parameters are (currently) ignored. To pass one parameter to a test, use `-p NAME=VAL`, and repeat it for other parameters.
- The `timeout` feature on the various *avocado.utils.process* functions is now respected for processes started with `sudo=True`. Sending general signals to processes that have also been started in privileged mode (and killing them) is now possible and is the basis of this improvement.
- The *avocado.utils.cloudinit* module now adds support for instances to be configured to allow `root` logins and authentication configuration via SSH keys.
- The *avocado.utils.distro* module introduced a probe for the Ubuntu distros.
- New *avocado.utils.disk.get_disk_blocksize()* and *avocado.utils.disk.get_disks()* disk related utilities.
- New *avocado.utils.process.get_parent_pid()* and *avocado.utils.process.get_owner_id()* process related functions

Bug Fixes

- The *avocado.utils.vmimage* had an issue when dealing with bytes and strings on Python 3. Now the expected encoding on the parsed web pages is explicitly given and used.
- The *avocado.utils.linux_modules.get_submodules()* function now returns unique modules names, instead of possibly having duplicate modules names.
- The system information collection, known in Avocado as "sysinfo", now properly collects information after failed and errored tests finish.
- The INSTRUMENTED test loader now properly finds all tests when, within the same module, either the Avocado library or the *avocado.Test* class is imported more than once, and with different names.
- The INSTRUMENTED test loader now won't crash when specific multi inheritance happens on test classes.
- The external test runner feature now supports relative paths given on the command line when used in conjunction with `--external-runner-chdir=runner`.

Internal Changes

- A number of utility libraries, including *avocado.utils.process* and *avocado.utils.linux_modules* have been modified to use system files (such as the ones from `/proc/`) instead of depending and executing command line utilities whenever possible. This type of change is expected to continue happening on Avocado.
- Tests depending on the presence of the HTML and remote plugin have been moved to the plugin themselves.
- A number of refactors and general improvements, usually accompanied by new tests, have increased the number of self tests from 549 to the 590 mark.
- Continuing from the past release, another large number of warnings checks have been enabled in the "lint" check, making the Avocado source code better now, and avoiding best practices regressions.
- Fixes to self tests that require privileged execution (tests covering the mount support in *avocado.utils.vmimage* and general operation of the *avocado.utils.lv_utils* module).

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/lhw9hO0L/1416-sprint-theme-back-to-the-future-1985

64.0 The man who would be king

The Avocado team is proud to present another release: Avocado version 64.0, AKA "The man who would be king", is now available!

Release documentation: Avocado 64.0

The major changes introduced on this version are listed below, roughly categorized into major topics and intended audience:

Users/Test Writers

- A new varianter plugin, the *CIT Varianter Plugin*. This plugin implements a "Pair-Wise", also known as "Combinatorial Independent Testing" algorithm, in pure Python. This exciting new functionality is provided thanks to a collaboration with the Czech Technical University in Prague.
- The *avocado.utils.distro* module has dropped the probe that depended on the Python standard library `platform.dist()`. The reason is the `platform.dist()` has been deprecated since Python 2.6, and has been removed on the upcoming Python 3.8.
- All optional plugins available on Python 2 RPM packages are now also available on Python 3 based RPM packages.
- The *avocado.utils.iso9660* module gained a pycdlib based backend, which is very capable, and pure Python ISO9660 library. This allows us to have a working *avocado.utils.iso9660* backend on environments in which other backends may not be easily installable.
- The *avocado.utils.iso9660.iso9660()* function gained a capabilities mechanism, in which users may request a backend that implement a given set of features.
- The *avocado.utils.iso9660* module, gained "create" and "write" capabilities, currently implemented on the pycdlib based backend. This allows users of the *avocado.utils.iso9660* module to create ISO images programmatically - a task that was previously done by running `mkisofs` and similar tools.
- The *avocado.utils.vmimage.get()* function now provides a directory in which to put the snapshot file, which is usually discarded. Previously, the snapshot file would always be kept in the cache directory, resulting in its pollution.
- The *avocado.utils.download* module, and the various utility functions that use it, will have extended logging, including the file size, time stamp information, etc.
- A brand new module, *avocado.utils.cloudinit*, that aides in the creation of ISO files containing configuration for the virtual machines compatible with cloudinit. Besides authentication credentials, it's also possible to define a "phone home" address, which is complemented by a simple phone home server implementation. On top of that, a very easy to use function to wait on the phone home is available as *avocado.utils.cloudinit.wait_for_phone_home()*.
- The Human UI plugin, will now show the "reason" behind test failures, cancellations and others right along the test result status. This hopefully will give more information to users without requiring them to resort to logs every single time.

Bug Fixes

- The *avocado.utils.partition* now behaves better when the system is missing the `lsof` utility.

Internal Changes

- Fixes generators on Python 3.7, according to PEP479.
- Other enablements for Python 3.7 environments were added, including RPM build fixes for Fedora 29.
- A large number of warnings checks have been enabled in the "lint" check, making the Avocado source code better now, and avoiding best practices regressions.

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/dTc5HtrX/1382-sprint-theme-the-man-who-would-be-king-1975

63.0 Greed in the Sun

The Avocado team is proud to present another release: Avocado version 63.0, AKA "Greed in the Sun", is now available!

Release documentation: Avocado 63.0

The major changes introduced on this version are listed below, roughly categorized into major topics and intended audience:

Users/Test Writers

- Including test logs in TAP plugin is disabled by default and can be enabled using `--tap-include-logs`.
- Performance is improved for the TAP plugin by only using `fsync()` after writes of important content, instead of doing it for all content, including the logs from tests.
- The command line options `--filter-by-tags` and `--filter-by-tags-include-empty` are now white listed for the remote runner plugin.
- The remote runner plugin will now respect `~/.ssh/config` configuration.
- The asset fetcher, available to a test via `avocado.core.Test.fetch_asset()`, will prevent clashes from downloaded files with the same name (when no hash is given), by using a directory named after the hash of the location.
- The identification of PCI bridge devices in *`avocado.utils.pci`* is now more precise by using its class.
- A smarter wait, instead of a sleep, is now used on *`avocado.utils.multipath`*.

Bug Fixes

- The recording of output, used by the output check functionality, is done as text, via a `RawFileHandler` logger. Now, instead of failing to encode data (depending on its content) and crashing, data is escaped using the `xmlcharrefreplace` handling.
- Avocado won't crash on systems without the `less` binary to be used as the paginator.

Internal Changes

- Self tests load failures are now caught on Python 3.4 environments (a workaround was needed due to Python 3.4 specific behavior, not necessary for 3.5+).
- Various build fixes related to the new Fabric packages and naming conventions.
- The *avocado.core.loader* module now makes use of better named symbolic values (based on enums), such as *avocado.core.loader.DiscoverMode.DEFAULT*.

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/EqauNWfL/1349-sprint-theme-greed-in-the-sun-1964

62.0 Farewell

The Avocado team is proud to present another release: Avocado version 62.0, AKA "Farewell", is now available!

Release documentation: Avocado 62.0

The major changes introduced on this version are listed below, roughly categorized into major topics and intended audience:

Users/Test Writers

- The `avocado.Test.srcdir` attribute has been removed, and with it, the `AVOCADO_TEST_SRCDIR` environment variable set by Avocado. This was done after a deprecation period, so tests should have been modified by now to make use of the *avocado.Test.workdir* instead.
- The `avocado.Test.datadir` attribute has been removed, and with it, the `AVOCADO_TEST_DATADIR` environment variable set by Avocado. This was done after a deprecation period, so tests should have been modified by now to make use of the *avocado.Test.get_data()* instead.
- The *avocado.utils.cpu.set_cpuidle_state()* function now takes a boolean value for its `disable` parameter (while still allowing the previous integer (0/1) values to be used). The goal is to have a more Pythonic interface, and to drop support legacy integer (0/1) use in the upcoming releases.
- *avocado.utils.astring.ENCODING* is a new addition, and holds the encoding used on many other Avocado utilities. If your test needs to convert between binary data and text, we recommend you use it as the default encoding (unless your test knows better).
- *avocado.utils.astring.to_text()* now supports setting the error handler. This means that when a perfect decoding is not possible, users can choose how to handle it, like, for example, ignoring the offending characters.

- When running a process by means of the *avocado.utils.process* module utilities, the output of such a process is captured and can be logged in a `stdout`/`stderr` (or combined `output`) file. The logging is now more resilient to decode errors, and will use the `replace` error handler by default. Please note that the downside is that this *may* produce different content in those files, from what was actually output by the processes if decoding error conditions happen.
- The *avocado.utils.astring.tabular_output()* will now properly strip trailing whitespace from lines that don't contain data for all "columns". This is also reflected in the (tabular) output of commands such as `avocado list -v`.

Bug Fixes

- Users of the *avocado.utils.service* module can now safely instantiate the service manager multiple times. It was previously limited to a single instance per interpreter.
- The *avocado.utils.vmimage* library default usage broke with the release of Fedora 28, which added a different directory layout for its cloud images. This has now been fixed and should allow for a successful `image = avocado.utils.vmimage()` usage.

Internal Changes

- Refactor of the *avocado.utils.asset* module, in preparation for new functionality.
- The *avocado.utils.cpu* module now treats reads/writes to/from `/proc/*` and `/sys/*` as binary data.
- The selftests for the *avocado.utils.cpu* module will now run under Python 3 (>= 3.6), due to more detailed checks of capable mock versions.
- The test that serves as the example for the *whiteboard* feature has been simplified, and the more complex test moved to `selftests`.
- Package builds with `make rpm` are now done with the systemd-nspawn based chroot implementation for `mock`.

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/4KtpSeGT/1305-sprint-theme-farewell-2009

61.0 Seven Pounds

The Avocado team is proud to present another release: Avocado version 60.0, AKA "Seven Pounds", is now available!

Release documentation: Avocado 61.0

The major changes introduced on this version are listed below, roughly categorized into major topics and intended audience:

Users/Test Writers

- The `xunit` result plugin can now limit the amount of output generated by individual tests that will make into the XML based output file. This is intended for situations where tests can generate prohibitive amounts of output that can render the file too large to be reused elsewhere (such as imported by Jenkins).
- SIMPLE tests can also finish with `SKIP` OR `WARN` status, depending on the output produced, and the Avocado test runner configuration. It now supports patterns that span across multiple lines. For more information, refer to *SIMPLE Tests Status*.
- Simple bytes and "unicode strings" utility functions have been added to *avocado.utils.astring*, and can be used by extension and test writers that need consistent results across Python major versions.
- All of core Avocado and all but one plugin (`yaml-to-mux`) now have all their tests enabled on Python 3. This means that for virtually all use cases, the experience of Python 3 users should be on par to the Python 2 experience. Please refer to https://trello.com/c/Q8QVmj8E/1254-bug-non-ascii-character-breaks-yaml2mux and https://trello.com/c/eFY9Vw1R/1282-python-3-functional-tests-checklist for the outstanding issues.

Bug Fixes

- The TAP plugin was omitting the output generated by the test from its own output. Now, that functionality is back, and commented out output will be shown after the `ok` or `not ok` lines.
- Packaging issues which prevented proper use of RPM packages installations, due to the lack dependencies, were fixed. Now, on both Python 2 and 3 packages, the right dependencies should be fulfilled.
- Replaying jobs that use the "YAML loader" is now possible. The fix was the implementation of the `fingerprint` method, previously missing from the *avocado.core.tree.TreeNodeEnvOnly* class.

Internal Changes

- The `glib` test loader plugin won't attempt to execute test references to list the glib tests, unless the test reference is an executable file.
- Files created after the test name, which include the `;` character, will now be properly mapped to a filesystem safe _;
- A number of improvements to the code quality, as a result of having more "warning" checks enabled on our lint check.
- A significant reduction in the default timeout used when waiting for hotplug operations on memory devices, as part of the utility module *avocado.utils.memory*.
- Improved support for non-ASCII input, including the internal use of "unicode" string types for *avocado.utils.process.run()* and similar functions. The command parameter given to those functions are now expected to be "unicode" strings.

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/4KtpSeGT/1305-sprint-theme-farewell-2009

60.0 Better Call Saul

The Avocado team is proud to present another release: Avocado version 60.0, AKA "Better Call Saul", is now available!

Release documentation: Avocado 60.0

The major changes introduced on this version are listed below, roughly categorized into major topics and intended audience:

Users/Test Writers

- The legacy options `--filter-only`, `--filter-out` and `--multiplex` have now been removed. Please adjust your usage, replacing those options with `--mux-filter-only`, `--mux-filter-out` and `--mux-yaml` respectively.
- The deprecated `skip` method, previously part of the *avocado.Test* API, has been removed. To skip a test, you can still use the *avocado.skip()*, *avocado.skipIf()* and *avocado.skipUnless()* decorators.
- The `avocado.Test.srcdir()` property has been deprecated, and will be removed in the next release. Please use *avocado.Test.workdir()* instead.
- Python 3 RPM packages are now available for the core Avocado and for many of the plugins. Users can install both versions side by side, and they'll share the same configuration. To run the Python 3 version, run `avocado-3` (or `avocado-3.x`, which x is the minor Python version) instead of `avocado`.
- The *avocado.utils.kernel* library now supports setting the URL that will be used to fetch the Linux kernel from, and can also build installable packages on supported distributions (such as `.deb` packages on Ubuntu).
- The *avocado.utils.process* library now contains helper functions similar to the Python 2 `commands.getstatusoutput()` and `commands.getoutput()` which can be of help to people porting code from Python 2 to Python 3.

Bug Fixes

- Each job now gets its own temporary directory, which allows multiple jobs to be used in a single interpreter execution.
- On some situations, Avocado would, internally, attempt to operate on a closed file, resulting in `ValueError: I/O operation on closed file`. This has been fixed in the *avocado.utils.process.FDDrainer* class, which will not only check if the file is not closed, but if the file-like object is capable of operations such as `fsync()`.
- Avocado can now (again) run tests that will produce output in encoding different than the Python standard one. This has been implemented as an Avocado-wide, hard-coded setting, that defines the default encoding to be `utf-8`. This may be made configurable in the future.

Internal Changes

- A memory optimization was applied, and allows test jobs with a large number of tests to run smoothly. Previously, Avocado would save the *avocado.Test.params* attribute, a *avocado.core.parameters.AvocadoParams* instance to the test results. Now, it just keeps the relevant contents of the test parameters instead.
- A number of warnings have been enabled on Avocado's "lint" checks, and consequently a number of mistakes have been fixed.
- The usage of the *avocado.core.job.Job* class now requires the use of *avocado.core.job.Job.setup()* and *avocado.core.job.Job.cleanup()*, either explicitly or as a context manager. This makes sure the temporary files are properly cleaned up after the job finishes.
- The exception raised by the utility functions in *avocado.utils.memory* has been renamed from `MemoryError` and became *avocado.utils.memory.MemError*. The reason is that `MemoryError` is a Python standard exception, that is intended to be used on different situations.
- A number of small improvements to the *avocado.Test* implementation, including making *avocado.Test.workdir()* creation more consistent with other test temporary directories, extended logging of test metadata, logging of test initialization (look for `INIT` in your test logs) in addition to the already existing start of test execution (logged as `START`), etc.

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/6a7jrsxA/1292-sprint-theme-better-call-saul

59.0 The Lobster

The Avocado team is proud to present another release: Avocado version 59.0, AKA "The Lobster", is now available!

Release documentation: Avocado 59.0

The major changes introduced on this version are listed below, roughly categorized into major topics and intended audience:

Users/Test Writers

- A new plugin enables users to list and execute tests based on the GLib test framework. This plugin allows individual tests inside a single binary to be listed and executed.
- Users of the YAML test loader have now access to a few special keys that can tweak test attributes, including adding prefixes to test names. This allows users to easily differentiate among execution of the same test, but executed different configurations. For more information, look for "special keys" in the YAML Loader plugin documentation.

- Users can now dump variants to a (JSON) file, and also reuse a previously created file in their future jobs execution. This allows users to avoid recomputing the variants on every job, which might bring significant speed ups in job execution or simply better control of the variants used during a job. Also notice that even when users do not manually dump a variants file to a specific location, Avocado will automatically save a suitable file at `jobdata/variants.json` as part of a Job results directory structure.
- SIMPLE tests were limited to returning PASS, FAIL and WARN statuses. Now SIMPLE tests can now also return SKIP status. At the same time, SIMPLE tests were previously limited in how they would flag a WARN or SKIP from the underlying executable. This is now configurable by means of regular expressions.
- The *avocado.utils.process* has seen a number of changes related to how it handles data from the executed processes. In a nutshell, process output (on both `stdout` and `stderr`) is now considered binary data. Users that need to deal with text instead, should use the newly added *avocado.utils.process.CmdResult.stdout_text* and *avocado.utils.process.CmdResult.stderr_text*, which are convenience properties that will attempt to decode the `stdout` or `stderr` data into a string-like type using the encoding set, and if none is set, falling back to the system default encoding. This change of behavior was needed to accommodate Python's 2 and Python's 3 differences in bytes and string-like types and handling.
- The TAP result format plugin received improvements, including support for reporting Avocado tests with CANCEL status as SKIP (which is the closest status available in the TAP specification), and providing more visible warning information in case Avocado tests finish with WARN status (while maintaining the test as a PASS, since TAP doesn't define a WARN status).
- Removal of a number of already deprecated features related to the 36.0 LTS series, which reached End-Of-Life during this sprint.
- Redundant (and deprecated) fields in the test sections of the JSON result output were removed. Now, instead of `url`, `test` and `id` carrying the same information, only `id` remains.
- Python 3 (beta) support. After too many changes to mention individually, Avocado can now run satisfactorily on Python 3. The Avocado team is aware of a small number of issues, which maps to a couple of functional tests, and is conscientious of the fact that many other issues may come up as users deploy and run it on Python 3. Please notice that all code on Avocado already goes through the Python 3 versions of `inspekt lint`, `inspekt style` and runs all unittests. Because of the few issues mentioned earlier, functional tests do yet run on Avocado's own CI, but are expected to be enable shortly after this release. For this release, expect packages to be available on PyPI (and consequently installable via `pip`). RPM packages should be available in the next release.

Bug Fixes

- Avocado won't crash when attempting, and not succeeding, to create a user-level configuration file `~/.config/avocado.conf`. This is useful in restricted environments such as in containers, where the user may not have its own home directory. Avocado also won't crash, but will report failure and exit, when it's not able to create the job results directory.
- Avocado will now properly respect the configuration files shipped in the Python module location, then the system wide (usually in `/etc`) configuration file, and finally the user level configuration files.
- The YAML test loader will now correctly log messages intended to go the log files, instead of printing them in the UI.
- Linux distributions detection code has been fixed for SuSE systems.
- The *avocado.utils.kernel* library now supports fetching all major versions of the Linux kernel, and not only kernels from the 3.x series.

Internal Changes

- Tests that perform checks on core Avocado features should not rely on upper level Avocado code. The `functional/test_statuses.py` selftest was changed in such a way, and doesn't require the `varianter_yaml_to_mux` plugin anymore.
- The Avocado assets and repository server now supports HTTPS connections. The documentation and code that refers to these services have been updated to use secure connections.

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/OTRQpSs7/1228-sprint-theme-the-lobster

58.0 Journey to the Christmas Star

The Avocado team is proud to present another release: Avocado version 58.0, AKA "Journey to the Christmas Star", is now available!

Release documentation: Avocado 58.0

The major changes introduced on this version are listed below, roughly categorized into major topics and intended audience:

Users/Test Writers

- The *`avocado.utils.vmimage`* library now contains support for Avocado's own JeOS image. A nice addition given the fact that it's the default image used in Avocado-VT and the latest version is available in the following architectures: x86_64, aarch64, ppc64, ppc64le and s390x.
- Avocado packages are now available in binary "wheel" format on PyPI. This brings faster, more convenient and reliable installs via `pip`. Previously, the source-only tarballs would require the source to be built on the target system, but the wheel package install is mostly an unpack of the already compiled files.
- The installation of Avocado from sources has improved and moved towards a more "Pythonic" approach. Installation of files in "non-Pythonic locations" such as `/etc` are no longer attempted by the Python `setup.py` code. Configuration files, for instance, are now considered package data files of the `avocado` package. The end result is that installation from source works fine outside virtual environments (in addition to installations *inside* virtual environments).
- Python 3 has been enabled, in "allow failures mode" in Avocado's CI environment. All static source code checks pass, and most of the unittests (*not* the functional tests) also pass. It's yet another incremental steps towards full Python 3 support.

Bug Fixes

- The *avocado.utils.software_manager* library received improvements with regards to downloads of source packages, working around bugs in older `yumdownloader` versions.

Internal Changes

- Spelling exceptions and fixes were added throughout and now `make spell` is back to a good shape.
- The Avocado CI checks (Travis-CI) are now run in parallel, similar to the stock `make check` behavior.

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/lHnzJT06/1208-sprint-theme-journey-to-the-christmas-star

57.0 Star Trek: Discovery

The Avocado team is proud to present another release: Avocado version 57.0, AKA "Star Trek: Discovery", is now available!

Release documentation: Avocado 57.0

The major changes introduced on this version are listed below, roughly categorized into major topics and intended audience:

Users/Test Writers

- A new (optional) plugin is available, the "result uploader". It allows job results to be copied over to a centralized results server at the end of job execution. Please refer to *Results Upload Plugin* for more information.
- The *avocado.utils.cpu* functions, such as `avocado.utils.cpu.cpu_oneline_list()` now support the S390X architecture.
- The `default_parameters` mechanism for setting default parameters on tests has been removed. This was introduced quite early in the Avocado development, and allowed users to set a dictionary at the class level with keys/values that would serve as default parameter values. The recommended approach now, is to just provide default values when calling `self.parameters.get` within a test method, such as `self.parameters.get("key", default="default_value_for_key")`.
- The `__getattr__` interface for `self.params` has been removed. It used to allow users to use a syntax such as `self.params.key` when attempting to access the value for key `key`. The supported syntax is `self.params.get("key")` to achieve the same thing.

- Yet another batch of progress towards Python 3 support. On this release, we have only 3 unittests that FAIL on a Python 3 environment. We even got bug reports of Avocado on Python 3, which makes us believe that it's already being used. Still, keep in mind that *there are still issues*, which will hopefully be iron out on the upcoming release(s).

Bug Fixes

- The *avocado.utils.crypto.hash_file()* function received fixes for a bug caused by a badly indented block.
- The *Golang Plugin* now won't report a test as found if the GO binary is not available to subsequently run those tests.
- The output record functionality receives fixes at the API level, so that it's now possible to enable and disable at the each API call.
- The subtests filter, that can be added to test references, was fixed and now works properly when added to directories and SIMPLE tests.
- The *avocado.utils.process.FDDrainer* now properly flushes its contents and the once ocurring data loss (last line read) is now fixed.

Internal Changes

- The "multiplexer" related code is being moved outside of the core Avocado. Only the variant plugin interface and support code (but not such an implementation) will remain in core Avocado.
- A new core `avocado.core.parameter` module was added and it's supposed to contain just the implementation of parameters, but no variants and/or multiplexer related code.
- The `sysinfo` feature implementation received a code clean up and now relies on the common *avocado.utils.process* code, to run the commands that will be collected, instead of having its own custom code for handling with output, timeouts, etc.

Other Changes

- The Avocado project now has a new server that hosts its RPM package repository and some other assets, including the JeOS images used on Avocado-VT. The documentation now points towards the new server and its updated URLs.

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/fJ1ilSuA/1198-sprint-theme-star-trek-discovery

56.0 The Second Mother

The Avocado team is proud to present another release: Avocado version 56.0, AKA "The Second Mother", is now available!

Release documentation: Avocado 56.0

The major changes introduced on this version are listed below, roughly categorized into major topics and intended audience:

Users/Test Writers

- The `avocado.core.utils.vmimage` library now allows users to expand the builtin list of image providers. If you have a local cache of public images, or your own images, you can quickly and easily register your own providers and thus use your images on your tests.
- A documentation on how to create your own base classes for your tests, kind of like you own Avocado-based test framework, was introduced. This should help users put common tasks into base classes and get even more productive test development.
- Avocado can record the output generated from a test, which can then be used to determine if the test passed or failed. This feature is commonly known as "output check". Traditionally, users would choose to record the output from `STDOUT` and/or `STDERR` into separate streams, which would be saved into different files. Some tests suites actually put all content of `STDOUT` and `STDERR` together, and unless we record them together, it'd be impossible to record them in the right order. This version introduces the `combined` option to `--output-check-record` option, which does exactly that: it records both `STDOUT` and `STDERR` into a single stream and into a single file (named `output` in the test results, and `output.expected` in the test data directory).
- A new varianter plugin has been introduced, based on PICT. PICT is a "Pair Wise" combinatorial tool, that can generate optimal combination of parameters to tests, so that (by default) at least a unique pair of parameter values will be tested at once.
- Further progress towards Python 3 support. While this version does not yet advertise full Python 3 support, the next development cycle will tackle any Python 3 issue as a critical bug. On this release, some optional plugins, including the remote and docker runner plugins, received attention and now execute correctly on a Python 3 stack.

Bug Fixes

- The remote plugin had a broken check for the timeout when executing commands remotely. It meant that the out-most timeout loop would never reach a second iteration.
- The remote and docker plugins had issues on how they were checking the installed Avocado versions.

Internal Changes

- The CI checks on Travis received a lot of attention, and a new script that and should be used by maintainers was introduced. `contrib/scripts/avocado-check-pr.sh` runs tests on all commits in a PR, and sends the result over to GitHub, showing other developers that no regression was introduced within the series.

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/s1WobkdE/1157-sprint-theme-the-second-mother-2015

55.0 Never Let Me Go

The Avocado team is proud to present another release: Avocado version 55.0, aka, "Never Let Me Go" is now available!

Release documentation: Avocado 55.0

The major changes introduced on this version are listed below, roughly categorized into major topics and intended audience:

Users/Test Writers

- Improvements in the serialization of TestIDs allow test result directories to be properly stored and accessed on Windows based filesystems.
- Support for listing and running golang tests has been introduced. Avocado can now discover tests written in Go, and if Go is properly installed, Avocado can run them.
- The support for test data files has been improved to support more specific sources of data. For instance, when a test file used to contain more than one test, all of them shared the same `datadir` property value, thus the same directory which contained data files. Now, tests should use the newly introduced *`get_data()`* API, which will attempt to locate data files specific to the variant (if used), test name, and finally file name. For more information, please refer to the section *Accessing test data files*.
- The output check feature will now use the to the most specific data source location available, which is a consequence of the switch to the use of the `get_data()` API discussed previously. This means that two tests in a single file can generate different output, generate different `stdout.expected` or `stderr.expected`.
- When the output check feature finds a mismatch between expected and actual output, will now produce a unified diff of those, instead of printing out their full content. This makes it a lot easier to read the logs and quickly spot the differences and possibly the failure cause(s).
- Sysinfo collection can now be enabled on a test level basis.
- Progress towards Python 3 support. Avocado can now run most commands on a Python 3 environment, including listing and running tests. The goal is to make Python 3 a "top tier" environment in the next release, being supported in the same way that Python 2 is.

Bug Fixes

- Avocado logs its own version as part of a job log. In some situations Avocado could log the version of a source repository, if the current working directory was an Avocado git source repo. That means that even when running, say, from RPM packages, the version number based on the source code would be registered.

- The output check record feature used to mistakenly add a newline to the end of the record stdout/stderr files.
- Problems with newline based buffering prevented Avocado from properly recording test stdout/stderr. If no newline was given at the end of a line, it would never show up in the stdout/stderr files.

Internal Changes

- The reference to `examples/*.yaml`, which isn't a valid set of files, was removed from the package manifest.
- The flexmock library requirement, used on some unittests, has been removed. Those tests were rewritten using `mock`, which is standard on Python 3 (`unittest.mock`) and available on Python 2 as a standalone module.

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/Oplm42c0/1132-sprint-theme-never-let-me-go

54.1 House of Cards (minor release)

Right on the heels of the 54.0 release, the Avocado team would like to apologize for a mistake that made into that version. The following change, as documented on 54.0 has been **reverted** on this 54.1 release:

- Test ID format Avocado has been using for a while received a minor tweak, to allow for better serialization into some filesystem types, such as Microsoft Windows' ones. Basically, the character that precedes the variant name, a separator, used to be `;`, which is not allowed on some filesystems. Now, a + character is used. A Test ID `sleeptest.py:SleepTest.test;short-beaf` on a previous Avocado version is now `sleeptest.py:SleepTest.test+short-beaf`.

The reason for the revert and the new release, is that the actual character causing trouble in Windows filesystems was "lost in translation". The culprit was the `:` character, and not `;`. This means that the Variant ID separator character change was unnecessary, and another fix is necessary.

Release documentation: Avocado 54.1

The major changes introduced on this version are listed below, roughly categorized into major topics and intended audience:

For more information, please check out the complete Avocado changelog.

54.0 House of Cards

The Avocado team is proud to present another release: Avocado version 54.0, aka, "House of Cards" is now available!

Release documentation: Avocado 54.0

The major changes introduced on this version are listed below, roughly categorized into major topics and intended audience:

Users/Test Writers

- Avocado can now run list and run standard Python unittests, that is, tests written in Python that use the `unittest` library alone. This should help streamline the execution of tests on projects that use different test types. Or, it may just be what plain `unittest` users were waiting for to start running them with Avocado.
- The Test ID format Avocado has been using for a while received a minor tweak, to allow for better serialization into some filesystem types, such as Microsoft Windows' ones. Basically, the character that precedes the variant name, a separator, used to be `;`, which is not allowed on some filesystems. Now, a + character is used. A Test ID `sleeptest.py:SleepTest.test;short-beaf` on a previous Avocado version is now `sleeptest.py:SleepTest.test+short-beaf`.
- The full path of the filename that holds the currently running test is now output is the test log, under the heading `Test metadata:`.
- The `yaml_to_mux` varianter plugin, while parsing the YAML files, would convert objects into *avocado.core.tree.TreeNode*. This caused when the variants were serialized (such as part of the job replay support). Objects are now converted into ordered dictionaries, which, besides supporting a proper serialization are also more easily accessible as test parameters.
- The test profilers, which are defined by default in `/etc/avocado/sysinfo/profilers`, are now executed without a backing shell. While Avocado doesn't ship with examples of shell commands as profilers, or suggests users to do so, it may be that some users could be using that functionality. If that's the case, it will now be necessary to write a script that wraps you previous shell command. The reason for doing so, was to fix a bug that could leave profiler processes after the test had already finished.
- The newly introduced *avocado.utils.vmimage* library can immensely help test writers that need access to virtual machine images in their tests. The simplest use of the API, `vmimage.get()` returns a ready to use disposable image (snapshot based, backed by a complete base image). Users can ask for more specific images, such as `vmimage.get(arch='aarch64')` for a image with a ARM OS ready to run.
- When installing and using Avocado in a Python virtual environment, the ubiquitous "venvs", the base data directory was one defined outside the virtual environment. Now, Avocado respects the virtual environment also in this aspect.
- A new network related utility function, `avocado.utils.network.PortTracker` was ported from Avocado-Virt, given the perceived general value in a variety of tests.
- A new memory utility utility, *avocado.utils.memory.MemInfo*, and its ready to use instance `avocado.utils.memory.meminfo`, allows easy access to most memory related information on Linux systems.
- The complete output of tests, that is the combination of `STDOUT` and `STDERR` is now also recorded in the test result directory as a file named `output`.

Bug Fixes

- As mentioned before, test profiler processes could be left running in the system, even after the test had already finished.
- The change towards serializing YAML objects as ordered dicts, instead of as *:class:'avocado.core.tree.TreeNode*, also fixed a bug, that manifested itself in the command line application UI.
- When the various `skip*` decorators were applied to `setUp` test methods, they would not be effective, and `tearDown` would also be called.
- When a job was replayed, tests without variants in the original (AKA "source" job, would appear to have a variant named `None` in the replayed job.

Internal Changes

- Avocado is now using the newest inspektor version 0.4.5. Developers should also update their installed versions to have comparable results to the CI checks.
- The old `avocado.test.TestName` class was renamed to `avocado.core.test.TestID`, and its member attributes updated to reflect the fact that it covers the complete Test ID, and not just a Test Name.

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/fA4RL1eo/1100-sprint-theme-house-of-cards

53.0 Rear Window

The Avocado team is proud to present another release: Avocado version 53.0, aka, "Rear Window" now available!

Release documentation: Avocado 53.0

The major changes introduced on this version are listed below, roughly categorized into major topics and intended audience:

Users/Test Writers

- A new loader implementation, that reuses (and resembles) the YAML input used for the varianter yaml_to_mux plugin. It allows the definition of test suite based on a YAML file, including different variants for different tests. For more information refer to yaml_loader.
- A better handling of interruption related signals, such as `SIGINT` and `SIGTERM`. Avocado will now try harder to not leave test processes that don't respond to those signals, and will itself behave better when it receives them. For a complete description refer to *signal_handlers*.
- The output generated by tests on `stdout` and `stderr` are now properly prefixed with `[stdout]` and `[stderr]` in the `job.log`. The prefix is **not** applied in the case of `$test_result/stdout` and `$test_result/stderr` files, as one would expect.
- Test writers will get better protection against mistakes when trying to overwrite *`avocado.core.test.Test`* "properties". Some of those were previously implemented using *`avocado.utils.data_structures.LazyProperty()`* which did not prevent test writers from overwriting them.

Internal Changes

- Some *`avocado.core.test.Test`* "properties" were implemented as lazy properties, but without the need to be so. Those have now be converted to pure Python properties.

- The deprecated `jobdata/urls` link to `jobdata/test_references` has been removed.
- The `avocado` command line argument parser is now invoked before plugins are initialized, which allows the use of `--config` with configuration file that influence plugin behavior.

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/SfBg9gdl/1072-sprint-theme-rear-window-1954

52.0 Pat & Mat

The Avocado team is proud to present another LTS (Long Term Stability) release: Avocado version 52.0, aka, "Pat & Mat" is now available!

Release documentation: Avocado 52.0

LTS Release

For more information on what a LTS release means, please read *RFC: Long Term Stability*.

For a complete list of changes from the last LTS release to this one, please refer to *52.0 LTS*.

Changes

The major changes introduced on this version are listed below, roughly categorized into major topics and intended audience:

Bugfixes

- The job replay option would not work with the `--execution-order` feature, but has now been fixed.
- The `avocado variants --system-wide` command is supposed to return one variant with the default parameter tree. This was not functional on the last few releases, but has now been fixed.
- The replay of jobs executed with Avocado 36.4 is now possible with this release.

Documentation

A lot of the activity on *this specific* sprint was on documentation. It includes these new topics:

- A list of all differences that users should pay attention to, from the 36.X release to this one.
- The steps to take when migrating from 36.X to 52.0.

- A review guide, with the list of steps to be followed by developers when taking a look at Pull Requests.
- The environment in which a test runs (a different process) and its peculiarities.
- The interface for the pre/post plugins for both jobs and tests.

Other Changes

- The HTML reports (generated by an optional plugin) now output a single file containing all the resources needed (JS, CSS and images). The original motivation of this change was to let users quickly access the HTML when they are stored as test results artifacts on servers that compress those files. With multiple files, multiple files had to be decompressed. If the process wasn't automatic (server and client support decompression) this would require a tedious process.
- Better examples of YAML files (to be used with the `yaml_to_mux` plugin) have been given. The other "example" files where really files intended to be used by selftests, and having thus been moved to the selftests data 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/6PuGdjJd/1054-sprint-theme-pat-mat-1976

51.0 The White Mountains

The Avocado team is proud to present another release: Avocado version 51.0, aka, "The White Mountains" now available!

Release documentation: Avocado 51.0

The major changes introduced on this version are listed below, roughly categorized into major topics and intended audience:

Users/Test Writers

- Users will be given more information when a test reference is not recognized by a given test loader.
- Users can now choose to proceed with the execution of a job even if one or more test references have not been resolved by one Avocado test loader (AKA a test resolver). By giving the command line option `--ignore-missing-references=on`, jobs will be executed (provided the job's test suite has at least one test).
- The `yaml-to-mux` varianter implementation (the only one at this point) is now an optional plugin. Basically, this means that users deploying this (and later) version of Avocado, should also explicitly install it. For `pip` users, the module name is `avocado-framework-plugin-varianter-yaml-to-mux`. The RPM package name is `python-avocado-plugins-varianter-yaml-to-mux`.

- Users can now choose in which order the job will execute tests (from its suite) and variants. Previously, users would always get one test executed with all its variants, than the second tests with all variants, and so on. Now, users can give the `--execution-order=tests-per-variant` command line option and all tests on the job's test suite will be executed with the first variant, then all tests will be executed with the second variant and so on. The original (still the current default behavior) can also be available explicitly selected with the command line option `--execution-order=variants-per-test`.
- Test methods on parent classes are now found upon the use of the new *recursive <docstring-directive-recursive>* docstring directive. While `:avocado: enable` enables Avocado to find INSTRUMENTED tests that do not look like one (more details *here*), `recursive` will do that while also finding test methods present on parent classes.
- The docstring directives now have a properly defined *format*. This applies to `:avocado: tags=` docstring directives, used for *categorizing tests*.
- Users can now see the tags set on `INSTRUMENTED` test when listing tests with the `-V` (verbose) option.

Internal Changes

- The `jobdata` file responsible for keeping track of the variants on a given job (saved under `$JOB_RESULTS/jobdata/multiplex`) is now called `variants.json`. As it names indicates, it's now a JSON file that contains the *result* of the variants generation. The previous file format was based on Python's pickle, which was not reliable across different Avocado versions and/or environments.
- Avocado is one step closer to Python 3 compatibility. The basic `avocado` command line application runs, and loads some plugins. Still, the very much known `byte` versus `string` issues plague the code enough to prevent tests from being loaded and executed. We anticipate that once the `byte` versus `string` is tackled, most functionality will be available.
- Avocado now uniformly uses *`avocado.core.output.LOG_UI`* for outputting to the UI and *`avocado.core.output.LOG_JOB`* to output to the job log.
- Some classes previously regarded as "test types" to flag error conditions have now be rewritten to *not* inherit from *`avocado.core.test.Test`*. It's now easier to identify real Avocado test types.

Improvements for Developers

- Developers now will also get Python "eggs" cleaned up when running `make clean`.
- Developers can now run `make requirements-plugins` to (attempt to) install external plugins dependencies, provided they are located at the same base directory where Avocado is.

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.

Next Release

The next Avocado release, 52.0, will be a LTS (Long Term Stability Release). For more information please read *RFC: Long Term Stability*.

Sprint theme: https://trello.com/c/dDou6uk0/1034-sprint-theme-the-white-mountains-the-tripods

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 conditions.

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 across 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 at *Job Diff*.
- 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 running. 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 achieved 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/user/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/user/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

9.7 BP000

Number BP000

Title Blueprint specification and requirements

Author Beraldo Leal <bleal@redhat.com>

Reviewers Cleber Rosa <crosa@redhat.com>, Jan Richter <jarichte@redhat.com>, Plamen Dimitrov <pdimitrov@pevogam.com>, Willian Rampazzo <willianr@redhat.com>

Created 29-Sep-2020

Type Meta Blueprint

Status Approved

Table of Contents

9.7.1 TL;DR

Having better records of architectural decisions in our repository is a good way to socialize our technical decisions, improve the code review process, avoid unnecessary code, and balance the workload among other developers. We are already using the "blueprint" documents for a while and still learning how to write them. Although we have a basic idea of what is a blueprint, some may not understand the concept because we are missing one meta-document that describes the blueprint's basic notions. This document is a blueprint to specify how we should write blueprints from now on.

9.7.2 Motivation

Depending on the project size, having very well defined and structured documents about the architecture decisions seems like an overkill, but could help projects of any size, including Avocado, save time, make better decisions and improve the way we socialize those decisions.

Today in the Avocado project we have the good practice to submit an RFC to our mailing list or to use/open a GitHub issue when we have a new idea. RFCs are a widespread way to disclose the architecture decisions, but they are just one part of a longer process. During the RFC phase, we argue in favor of a proposal, and we are mostly concerned about collecting feedback. After this phase, we could go one step forward and consolidate the discussion in Blueprints (sometimes called ADRs - Architecture Decision Records). This could be the next step so we could better socialize our decisions for future readers. A very well defined and structured document has some advantages over an RFC, but it is not intended to replace it, just be a later stage to follow from it.

With blueprints, we could not only, but mainly:

- Create better documents for future members and future reference, when we are trying to answer the following questions:

a) *why* are you doing this? (the "problem" or the "motivation");

b) *what* are you proposing to solve the problem? (your "solution")?

c) And *how* are you going to implement the proposed solution?

Depending on the type of your blueprint, the answer for the last question (c) could be written in pseudocode, general text or might even not be necessary (although desired) — more details on the section named Specification.

When using RFCs as email threads, there are no sections or headers, each contributor will try to send the RFC without following formal sections and headers. RFCs, as we use them today are just thread discussions and are not focused on future review/reading.

- Make sure that another peer will be able to implement a feature/requirement.

Blueprints are not for you; they are for the community. If you know that a problem exists and know how to fix it, the most natural course of action would be to start coding this fix and submitting a Pull Request. While this is still valid for most of the cases, some important architectural changes must be discussed first to explain the "why", "what" and "how" to keep everyone on the same page, avoid unnecessary coding, and most importantly: allow others to implement it in case you are not available.

- Improve the code review quality and time.

Having a better understanding of the problem and the big picture is better for code review. It is harder to capture that from the Pull Request (PR) description and PR changes. Developers that are aware of the problem tend to review your changes with the problem in mind and hence more quickly.

- Reduce onboarding time for new members.

Having a history of how we made an architectural decision and why we are implementing it this way will give new members reading material to understand our project, avoiding unnecessary discussions and meetings to explain something.

- Create a common standard that will make it easier for readers.

With an open RFC, authors tend to organize the ideas in different ways with different sections. Having a very well structured document with common sections will make it easier for readers to understand the problem and the solution.

- Track the progress of a significant implementation.

We could use the blueprints header "status" line to track the progress of some features. We could even have a page parsing and listing all the blueprints with the title, author, status, and target milestone for that feature.

- Find the middle ground between "overthinking" and "auto-pilot."

Last but not least: we are *not trying* to overthink here and/or slow down our development processes. The idea is to have a lightweight document, very objective that will save us time in the medium and long run. We don't have to overthink by trying to handle any possible scenario outside of ones we actually have a use case for. But we should also avoid the "auto-pilot" mode in our contributions where we are fixing issues as quick as possible without thinking about the big picture, it is not healthy for the project.

9.7.3 Specification

One blueprint per topic

Try to follow the minimalist approach and be concise with content relevant to one particular topic. If you have a more general topic to discuss, you should set the type as "Epic Blueprint" (more below) but still try to be concise and focused on the subject.

File format and location

Our current documentation already uses ReStructuredText (.rst) format, so we will adopt .rst format here too. All blueprints will be located inside *docs/source/blueprints* with the filename *BPXXX.rst*, where XXX is the number of the blueprint. Just pick the next number available for your blueprint.

It's recommended that you use `docs/source/blueprints/template.rst` as a starting point.

Write for your audience

As mentioned before, your blueprint will be read by your peers, future members, and future yourself. Keep in mind that your audience is developers with a minimal understanding of the Avocado internals and be kind providing any necessary context to understand the problem.

Blueprints types

Currently, we have the following blueprint types:

- Architectural Blueprint: Any blueprint changing or introducing a new core feature or architectural change to Avocado.
- Process Blueprint: Any blueprint that is not implementing a new core feature, but changing how the project works. This could be, for instance, related to the repositories or processes.
- Meta Blueprint: A blueprint about blueprints. Like this one and any future blueprint that changes our blueprint's styles and methods.
- Epic Blueprint: A blueprint that is touching on multiple areas and is too big to have all the documentation in one single blueprint. We could split epic blueprints into smaller blueprints or issues (if they are small and easy to understand). Epic Blueprints are not a merge of all sub-blueprints. Like an epic issue, epic blueprints don't need to detail "how" (or provide details) that the sub-blueprints could have.
- Component Blueprint: A blueprint with the intent to describe a new utility module or a new plugin.

Headers

Python PEPs (Python Enhancement Proposals) uses RFC822 for describing the headers. This could be useful here too, especially when parsing those headers to display our list of blueprints with the current status.

The current list of items of our blueprint headers is below:

- Number: Usually, the blueprint number in the format BPXXX
- Title: A short descriptive title, limited to 80 characters
- Author: The author or authors of blueprint. Following the format: *[FIRST NAME] [LAST NAME] - <email@domain>*
- Reviewers: All reviewers that approved and helped during the review process
- Created: Date string when the blueprint first draft was submitted. Please use the following format: DD-MMM-YYYY.
- Type: One of the types described during the previous section
- Status: One of the types described during the next section

Here is an example of a header:

```
:Number: BP001
:Title: Configuration by convention
:Author: Beraldo Leal <bleal@redhat.com>
:Reviewers: Cleber Rosa, Lukáš Doktor and Plamen Dimitrov
:Created: 06-Dec-2019
:Type: Epic Blueprint
:Status: WIP
```

Blueprint statuses

- Draft: All blueprints should be created in this state. This means the blueprint is accepting comments, and probably there is a discussion happening. Blueprints in draft mode can be part of our repository.
- Approved: Blueprint was approved after discussions, and all suggestions are already incorporated on the document. Nobody has started working on this yet.
- Assigned: This status is not about the blueprint itself, but about the proposal that is the subject of the BP. This means that the blueprint was approved, and someone is already working on implementing it. A BP status can change from Draft to Assigned if the work has started already.
- WIP: Blueprint was approved and someone is working on it. Work in Progress.
- Implemented: This means the BP is already implemented and delivered to the Avocado's master branch.
- Rejected: Rejected status means the idea was not implemented because it wasn't approved by everyone or has some technical limitations.
- Deprecated: Deprecated means it was approved, implemented, and at some point, makes no more sense to have it. For example, anything related to the legacy runner. Usually, Deprecated means that it was replaced by something else.

As you can see, there is no status to accommodate any future change in a blueprint. Blueprints should not be "voided." Any improvement on an old blueprint should be presented as a new blueprint, changing the status of the original to "deprecated".

Sections

In order to facilitate the reading and understanding of the problem, all blueprints must have the following sections:

- TL;DR
- Motivation

- Specification
- Backwards Compatibility
- Security Implications
- How to Teach This
- Related Issues
- References

Below you can find a brief description of what you should write in each section:

- TL;DR: Should be a short description of your blueprint. Like an abstract. We recommend writing this at the end of your first draft. This will give you a better overview of it.
- Motivation: This should be the motivation of your proposed solution, not the motivation of the blueprint itself. It describes the problem. Here, you should answer "why" your solution is needed.
- Specification: In this section, you should describe how you are going to solve the problem. You can create subsections here to organize your ideas better. Please keep in mind that it is useful to mention the details, with code snippets, examples, and/or references. This will save you time, making sure that everyone is in agreement with the proposed solution.
- Backwards Compatibility: How is your proposal going to affect older versions of Avocado? Should we deprecate some modules, classes, or methods? Are we going to keep backwards compatibility or not?
- Security Implications: Do you have any concerns about security with your proposed solution and what are they? If there's functionality that is insecure but highly convenient, consider how to make it "opt-in", disabled by default.
- How to Teach This: What is the best way to inform our devs and users about your new feature/solution? Consider both "how-to" and reference style documentation, and if appropriate, examples (under `examples/`) using the feature.
- Related Issues: Here, you should mention Github links for both: a) current open issues that are blocking while waiting for your BP and b) all open issues that will render this BP as "implemented" when closed.

  1. Issues to address this BP

     Would be nice, if possible, to open issues on GH that covers all aspects of your Blueprint.

  2. Issues this BP will solve

     What are the issues already existent on Avocado project that your proposal will solve?

- References: Any external reference for helping understand the problem and your solution.

9.7.4 Backwards Compatibility

So far, we are on our 3rth blueprint (BP003 was the last one). This BP000 should have been released before those blueprints. So probably those three blueprints are not 100% compliaant with this meta blueprint, and that is fine. We were learning on the fly. We don't need to change any of those blueprints after BP000 gets approved.

9.7.5 Security Implications

No security implications found so far.

9.7.6 How to Teach This

Getting used to writing blueprints is not an easy task. And probably we are going to find unplanned issues with this process on the way. The general rule of thumb is to use common sense. To make this more public, we could consider the following:

- If approved, BP000 should be on top of our blueprints lists for reference.
- We could also have a template inside the *blueprints* directory to help people when submitting their own blueprints.
- Also, we could include pointers and instructions in our development guide for this BP.
- Another good practice would be to make comments in Avocado's source code with some pointers to specific blueprints.

9.7.7 Related Issues

None.

9.7.8 References

None.

9.8 BP001

Number BP001

Title Configuration by convention

Author Beraldo Leal <bleal@redhat.com>

Discussions-To avocado-devel@redhat.com

Reviewers Cleber Rosa, Lukáš Doktor and Plamen Dimitrov

Created 06-Dec-2019

Type Epic Blueprint

Status Approved

Table of Contents

9.8.1 TL;DR

The number of plugins made by many people and the lack of some name, config options, and argument type conventions may turn Avocado's usability difficult. This also makes it challenging to create a future API for executing more complex jobs. Even without plugins the lack of convention (or another type or order setting mechanism) can induce growth pains.

After an initial discussion on avocado-devel, we came up with this "blueprint" to change some config file settings and argparse options in Avocado.

This document has the intention to list the requirements before coding. And note that, since this is a relatively big change, this blueprint will be broken into small cards/issues. At the end of this document you can find a list of all issues that we should solve in order to solve this big epic Blueprint.

9.8.2 Motivation

An Avocado Job is primarily executed through the *avocado run* command line. The behavior of such an Avocado Job is determined by parsing the following settings (listed in parsed order):

1) Default values in source code

2) Configuration file contents

3) Command-line options

Currently, the Avocado config file is an .ini file that is parsed by Python's *configparser* library and this config is broken into sections. Each Avocado plugin has its dedicated section.

Today, the parsing of the command line options is made by *argparse* library and produces a dictionary that is given to the *avocado.core.job.Job()* class as its *config* parameter.

There is a lack of convention/order in the item 1. For instance, we have "avocado/core/defaults.py" with some defaults, but there are other such defaults scattered around the project, with ad-hoc names.

There is also no convention on the naming pattern used either on configuration files or on command-line options. Besides the name convention, there is also a lack of convention for some argument types. For instance:

```
$ avocado run -d
```

and:

```
$ avocado run --sysinfo on
```

Both are boolean variables, but with different "execution model" (the former doesn't need arguments and the latter needs *on* or *off* as argument).

Since the Avocado trend is to have more and more plugins, we need to design a name convention on command-line arguments and settings to avoid chaos.

But, most important: It would be valuable for our users if Avocado provides a Python API in such a way that developers could write more complex jobs programmatically and advanced users that know the configuration entries used on jobs, could do a quick one-off execution on command-line.

Example:

```
import sys
from avocado.core.job import Job

config = {'references': ['tests/passtest.py:PassTest.test']}

with Job(config) as j:
  sys.exit(j.run())
```

Before we address this API use-case, it is important to create this convention so we can have an intuitive use of Avocado config options.

We understand that, plugin developers have the flexibility to configure they options as desired but inside Avocado core and plugin, settings should have a good naming convention.

9.8.3 Specification

Basics on Defaults

The Oxford dictionary lists the following as one of the meanings of the word "default" (noum):

> *"a preselected option adopted by a computer program or other mechanism when no alternative is specified by the user or programmer."*

The basic behavior on defaults values vs config files vs command line arguments should be:

1. Avocado has all default values inside the source code;
2. Avocado parses the config files and override the defined values;
3. Avocado parses the command-line options and override the defined values;

If the config files or configuration options are missing, Avocado should still be able to use the default values. Users can only change 2 and 3.

Note: New Issue: Converte all "currently configured settings" into a default value.

Mapping between configuration options

Currently, Avocado has the following options to configure it:

1. Default values;
2. Configuration files;
3. Command-line options;

Soon, we will have a fourth option:

4. Job API config argument;

Although we should keep an eye on item 4 while implementing this blueprint, it is not intended to address the API at this time.

The default values (within the source code) should have an 1:1 mapping to the configuration file options. Must follow the same naming convention and sections. Example:

```
#avocado.conf:
[core]
foo = bar
[core.sysinfo]
foo = bar
[pluginx]
foo = bar
```

Should generate a dictionary or object in memory with a 1:1 mapping, respecting chained sections:

```
{'core': {'foo': 'bar',
          'sysinfo': {'foo': 'bar'}},
 'pluginx': {'foo': 'bar'}}
```

Again, if the config file is missing or some option is missing the result should be the same, but with the default values.

Since the command-line options are only the most used and basic ones, there is no need to have a 1:1 mapping between item 2 and item 3.

When naming subcommands options you don't have to worry about name conflicts outside the subcommand scope, just keep them short, simple and intuitive.

When naming a command-line option on the core functionality we should remove the "core" word section and replace "_" by "-". For instance:

```
[core]
execution_timeout = 30
```

Should be:

```
avocado --execution-timeout 30
```

When naming plugin options, we should try to use the following standard:

```
[pluginx]
foo = bar
```

Becomes:

```
avocado --pluginx-foo bar
```

This only makes sense if the plugins' names are short.

Warning: Maybe I have to get more used with all the Avocado options to understand better. Or someone could help here.

Standards for Command Line Interface

When it comes to the command line interface, a very interesting recommendation is the POSIX Standard's recommendation for arguments[1]. Avocado should try to follow this standard and its recommendations.

This pattern does not cover long options (starting with –). For this, we should also embrace the GNU extension[2].

One of the goals of this extension, by introducing long options, was to make command-line utilities user-friendly. Also, another aim was to try to create a norm among different command-line utilities. Thus, –verbose, –debug, –version (with other options) would have the same behavior in many programs. Avocado should try to, where applicable, use the GNU long options table[3] as reference.

Note: New Issue: Review the command line options to see if we can use the GNU long options table.

Many of these recommendations are obvious and already used by Avocado or enforced by default, thanks to libraries like *argparse*.

However, those libraries do not force the developer to follow all recommendations.

Besides the basic ones, there is a particular case to pay attention: "option-arguments".

Option-arguments should not be optional (Guideline 7, from POSIX). So we should avoid this:

```
avocado run --loaders [LOADERS [LOADERS ...]]
```

or:

```
avocado run --store-logging-stream [STREAM[:LEVEL] [STREAM[:LEVEL] ...]]
```

As discussed we should try to have this:

```
avocado run --loaders LOADERS [LOADERS ...]
```

Note: New Issue: Make the option-arguments not optional.

Argument Types

Basic types, like strings and integers, are clear how to use. But here is a list of what should expect when using other types:

1. **Booleans**: Boolean options should be expressed as "flags" args (without the "option-argument"). Flags, when present, should represent a True/Active value. This will reduce the command line size. We should avoid using this:

```
avocado run --json-job-result {on,off}
```

So, if the default it is enabled, we should have only one option on the command-line:

```
avocado run --disable-json-job-result
```

This is just an example, the name and syntax may be different.

Note: New Issue: Fix boolean command line options

2. **Lists**: When an option argument has multiple values we should use the space as the separator.

Note: New Issue: Review if we have any command line list using non space as separator.

Presentation

Finding options easily, either in the manual or in the help, favor usability and avoids chaos.

We can arrange the display of these options in alphabetical order within each section.

Standards for Config File Interface

Many other config file options could be used here, but since that this is another discussion, we are assuming that we are going to keep using *configparser* for a while.

As one of the main motivations of this Blueprint is to create a convention to avoid chaos and make the job execution API use as straightforward as possible, We believe that the config file should be as close as possible to the dictionary that will be passed to this API.

For this reason, this may be the most critical point of this blueprint. We should create a pattern that is intuitive for the developer to convert from one format to another without much juggling.

Nested Sections

While the current *configparser* library does not support nested sections, Avocado can use the dot character as a convention for that. i.e: *[runner.output]*.

This convention will be important soon, when converting a dictionary into a config file and vice-versa.

And since almost everything in Avocado is a plugin, each plugin section should **not** use the "plugins" prefix and **must** respect the reserved sections mentioned before. Currently, we have a mix of sections that start with "plugins" and sections that don't.

Note: New Issue: Remove "plugins" from the configuration section names.

Plugin section name

Most plugins currently have the same name as the python module. Example: human, diff, tap, nrun, run, journal, replay, sysinfo, etc.

These are examples of "good" names.

However, some other plugins do not follow this convention. Ex: runnable_run, runnable_run_recipe, task_run, task_run_recipe, archive, etc.

We believe that having a convention here helps when writing more complex tests, configfiles, as well as easily finding plugins in various parts of the project, either on a manual page or during the installation procedure.

We understand that the name of the plugin is different from the module name in python, but in any case we should try to follow the PEP8:

> From PEP8: *Modules should have short, all-lowercase names. Underscores can be used in the module name if it improves readability. Python packages should also have short, all-lowercase names, although the use of underscores is discouraged.*

Let's get the *human* example:

- Python module name: human
- Plugin name: human

Let's get the *task_run_recipe* example:

- Python module name: task_run_recipe
- Plugin name: task-run-recipe

Let's get another example:

- Python module name: archive
- Plugin name: zip_archive

One suggestion should be to have a namespace like *resolvers.tests.exec*, *resolvers.tests.unit.python*.

And all the duplicated code could be imported from a common module inside the plugin. But yes, it is a "delicate issue".

Note: New Issue: Rename the plugins modules and names. This might be tricky.

Reserved Sections

We should have one reserved section, the *core* section for the Avocado's core functionalities.

All plugin code that it is considered "core" should be inside core as a "nested section". Example:

```
[core]
foo = bar

[core.sysinfo]
collect_enabled = True
```

Note: New Issue: Move all 'core' related settings to the core section.

Config Types

configparser do not guess datatypes of values in configuration files, always storing them internally as strings. This means that if you need other datatypes, you should convert on your own

There are few methods on this library to help us: *getboolean()*, *getint()* and *getfloat()*. Basic types here, are also straightforward.

Regarding boolean values, *getboolean()* can accept *yes/no*, *on/off*, *true/false* or *1/0*. But we should adopt one style and stick with it.

Note: New Issue: Create a simple but effective type system for configuration files and argument options.

Presentation

As the avocado trend is to have more and more plugins, We believe that to make it easier for the user to find where each configuration is, we should split the file into smaller files, leaving one file for each plugin. Avocado already supports that with the conf.d directory. What do you think?

Note: New Issue: Split config files into small ones (if necessary).

9.8.4 Backwards Compatibility

In order to keep a good naming convention, this set of changes probably will rename some args and/or config file options.

While some changes proposed here are simple and do not affect Avocado's behavior, others are critical and may break Avocado jobs.

Command line syntax changes

These command-line conversions will lead to a "syntax error". We should have a transition period with a "deprecated message".

Plugin name changes

Changing the modules names and/or the 'name' attribute of plugins will require to change the config files inside Avocado as well. This will not break unless the user is using an old config file. In that case, we should also have a "deprecated message" and accept the old config file option for some time.

9.8.5 Security Implications

Avocado users should have the warranty that their jobs are running on isolated environment.

We should consider this and keep in mind that any moves here should continue with this assumption.

9.8.6 How to Teach This

We should provide a complete configuration reference guide section in our User's Documentation.

Note: New Issue: Create a complete configuration reference.

In the future, the Job API should also be very well detailed so sphinx could generate good documentation on our Test Writer's Guide.

Besides a good documentation, there is no better way to learn than by example. If our plugins, options and settings follow a good convention it will serve as template to new plugins.

If these changes are accepted by the community and implemented, this RFC could be adapted to become a section on one of our guides, maybe something like the a Python PEP that should be followed when developing new plugins.

Note: New Issue: Create a new section in our Contributor's Guide describing all the conventions on this blueprint.

9.8.7 Related Issues

Here a list of all issues related to this blueprint:

1. Create a new section in our Contributor's Guide describing all the conventions on this blueprint.
2. Create a complete configuration reference.
3. Split config files into small ones (if necessary).
4. Create a simple but effective type system for configuration files and argument options.
5. Move all 'core' related settings to the core section.
6. Rename the plugins modules and names. This might be tricky.
7. Remove "plugins" from the configuration section names.
8. Review if we have any command line list using non space as separator.
9. Fix boolean command line options.
10. Make the option-arguments not optional.
11. Review the command line options to see if we can use the GNU long options table.
12. Converte all "currently configured settings" into a default value.

Warning: After this blueprint get approved, I will open all issues on GH, add links here and remove all the notes.

9.8.8 References

[1] - https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap12.html

[2] - https://www.gnu.org/prep/standards/html_node/Command_002dLine-Interfaces.html

[3] - https://www.gnu.org/prep/standards/html_node/Option-Table.html#Option-Table

9.9 BP002

Number BP002

Title Requirements resolver

Author Willian Rampazzo <willianr@redhat.com>

Discussions-To https://github.com/avocado-framework/avocado/issues/3455

Reviewers Beraldo Leal, Cleber Rosa

Created 27-Jan-2020

Type Architecture Blueprint

Status WIP

Table of Contents

9.9.1 TL;DR

The current management of test assets is handled manually by the test developer. It is usual to have a set of repetitive code blocks to define the name, location, and other attributes of an asset, download it or signal an error condition if a problem occurred and the download failed.

Based on use cases compiled from the discussion on qemu-devel mailing-list [1] and discussions during Avocado meetings, this blueprint describes the architecture of a requirements resolver aiming the extensibility and flexibility when handling different types of assets, like a file, a cloud image, a package, a Git repository, source codes or Operating System parameters.

9.9.2 Motivation

Implementing a test that gathers its requirements while executing may lead to a wrong interpretation of the test results if a requirement is not satisfied. The failure of a test because of a missing requirement does not mean the test itself failed. During its execution, the test has never reached the core test code; still, it may be considered a failing test.

Fulfilling all the test requirements beforehand can be an efficient way to handle requirements problems and can improve the trustworthiness of the test result. It means that if a test ran and failed, the code responsible for the failure is related to the core test and not with one of its requirements.

Regardless of how the test defines a requirement, an architecture capable of identifying them is beneficial. Storing its references and delegating to the code responsible for handling each different type of requirement makes the overall architecture of Avocado and the requirement definition of a test more flexible.

A requirements resolver can bring the necessary flexibility to the Avocado architecture, as well as managing support for different types of requirements.

This blueprint discusses the architecture of a requirements resolver responsible for handling the different requirements types.

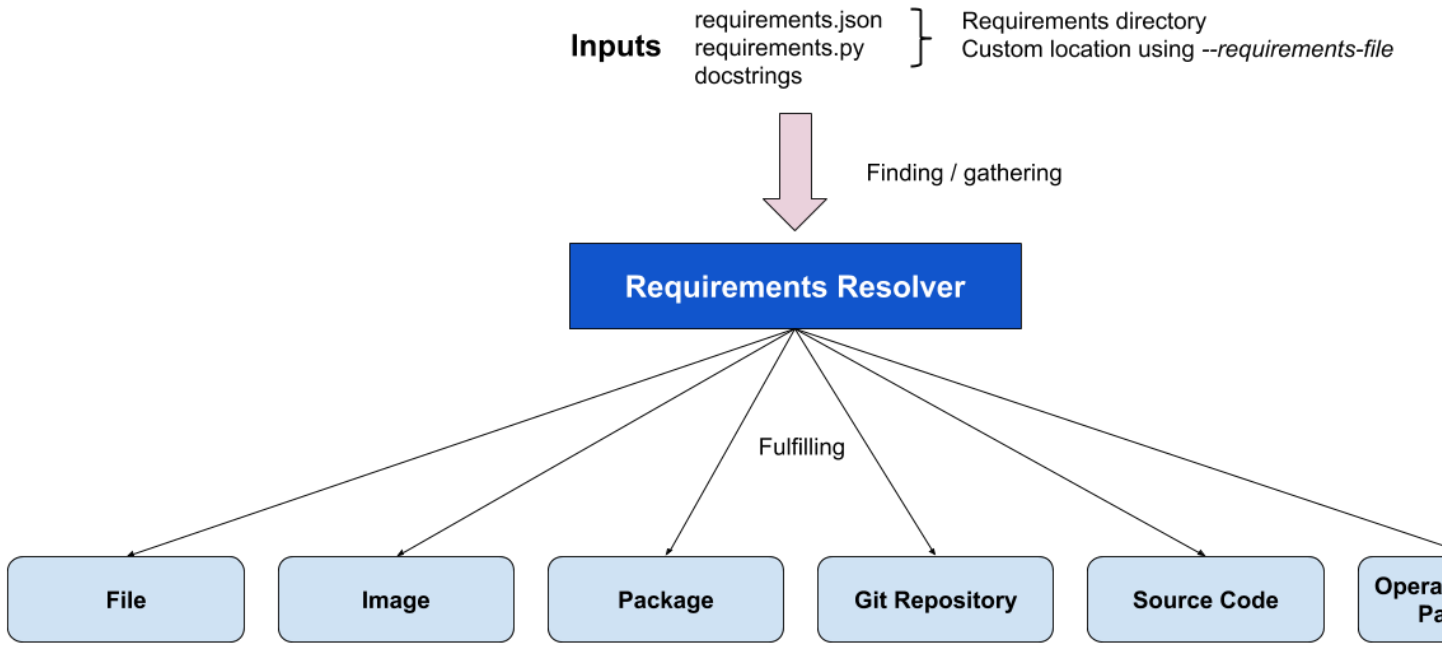
9.9.3 Specification

Basics

The strict meaning of a resolver is related to something responsible for creating resolutions from a given representation. When there is a well-defined way to declare something, a resolver can translate this representation to another well-defined representation. The classic example is a Domain Name Server (DNS), which resolves the hostname into an Internet Protocol (IP) address. The use of the word *resolver* in this text means a code responsible for gathering and fulfilling well-know representations with little or no transformation.

The definition of requirements resolver in this blueprint is a code responsible for gathering well-known formats of requirements, possibly from different sources, and centralizing in one place, or fulfilling them. The requirements fulfillment can take place starting from the centralized collection of requirements as input to one of several modules responsible for handling each specific type of requirement, like, for example, files, images, packages, git repositories, source code or operating system parameters.

The following diagram shows the underlying architecture of a requirements resolver proposed in this blueprint. The next sessions describes, in detail, each part of the resolver, its inputs, and outputs.



Requirements representations

Define how to represent a requirement is the first step to define the architecture of a resolver. This blueprint defines the following ways to represent a requirement:

1. JavaScript Object Notation (JSON) file;
2. Python executable that produces a JSON file;
3. Metadata included in the test docstring.

Requirements representation as JSON files

JSON is a lightweight data-interchange format [2] supported by the Python standard library. Using it to represent requirements is flexible and straightforward.

The standard proposed way to represent requirements with JSON is defining one requirement per entry. Each entry should start with the requirement type, followed by other keyword arguments related to that type. Example:

```
[
    {"type": "file", "uri": "https://cabort.com/cabort.c", "hash": "deadbeefdeadbeef"}
↪,
    {"type": "vmimage", "distro": "fedora", "version": 31, "arch": "x86_64"},
    {"type": "package", "package": "lvm"}
]
```

The requirement *type* should match the module responsible for that type of requirement.

Requirements representation as Python executable

Another way to create the requirements representation as JSON files is by writing a Python executable. This approach makes the requirements representation flexible, by allowing the use of Python variables and code that may change the

parameters values for the requirements, depending on the environment the Python code runs.

The following example shows a requirement that depends on the architecture the test is running:

```
#!/usr/bin/python3

import os
import json

requirements = [
    {"type": "file", "uri": "https://cabort.com/cabort.c", "hash": "deadbeefdeadbeef"}
↪,
    {"type": "vmimage", "distro": "fedora", "version": 31, "arch": os.uname()[4]},
    {"type": "package", "package": "lvm"}
]

print(json.dumps(requirements))
```

Requirements representation as Metadata on test docstring

Test writers may want to add the requirements of a test into de test code. The option proposed here allows the use of metadata on test docstrings to represent the requirements list.

Below is an example of how to define requirements as metadata on docstrings:

```
def test_something(self):
'''
    :avocado: requirement={"type": "file", "uri": "https://cabort.com/cabort.c", "hash
↪": "deadbeefdeadbeef"}
    :avocado: requirement={"type": "vmimage", "distro": "fedora", "version": 31, "arch
↪": "x86_64"}
    :avocado requirement={"type": "package", "package": "lvm"}
'''
   <test code>
```

Requirements files location

It may be useful for test writers to define a standard source location for the requirements JSON files and the requirements Python executable.

This blueprint defines the default location for a job-wide requirements file in the same directory of the test files or test-specific requirements files into a requirements directory preceded by the test file name. It is also possible to use sub-directories with the name of a specific test to define requirements for that test.

The following file tree is an example of possible use for requirements directories for a test:

```
requirements.json
cabort.py
cabort.py.requirements/
├── CAbort.test_2
│   └── requirements.py
└── requirements.json
```

In this case, all the tests on *cabort.py*, except for *CAbort.test_2*, use the *requirements.json* file located at *cabort.py.requirements*. The *CAbort.test_2* test uses its own *requirements.py* located at *CAbort.test_2* directory in-

side the requirements directory. The tests located at the same directory of *cabort.py* use the *requirements.json* in the root directory.

Requirements files command-line parameter

It is also possible to use a command-line parameter to define the location of the requirements file. The command-line parameter supersedes all the other possible uses of requirements files. For that, this blueprint defines the parameter *–requirements-file* followed by the location of the requirements file. As a command-line example, we have:

```
avocado --requirements-file requirements.json run passtest.py
```

Note: New Issue: Add the support for *–requirements-file* command-line parameter.

9.9.4 Backward Compatibility

The implementation of the requirements resolver, proposed here, affects Avocado's behavior related to the tasks executed before a test execution starts.

To make the requirements resolver as flexible as possible, the implementation of this blueprint may change the utility APIs related to a requirement type.

9.9.5 Security Implications

Avocado users should have the warranty that their jobs are running in an isolated environment, but Avocado can, conservatively, create mechanisms to protect the users from running unintended code.

The use of a Python executable to build the requirements file is subject to security considerations. A malicious code distributed as a Python executable to build the requirements file can lead to security implications. This blueprint proposes a security flag in a general Avocado configuration file to avoid Python executable code to run by default. Users can change this flag anytime to be able to use the ability to run Python executable codes to generate the requirements JSON file.

Following is an example of how this flag can look like:

```
[resolver.requirements]
# Whether to run Python executables to build the requirements file
unsafe = False
```

Note: New Issue: Add the unsafe flag support for the requirements resolver.

9.9.6 How to Teach This

We should provide a complete and detailed explanation of how to handle test requirements in the User's Documentation.

Note: New Issue: Create a complete section in the User's Guide on how to handle test requirements.

Also, we should address how to create utility modules to handle new types of requirements in the Contributor's Guide.

Note: New Issue: Create a new section in the Contributor's Guide on how to develop modules to handle new types of requirements.

9.9.7 Related Issues

Here a list of all issues related to this blueprint:

1. [DONE] Implement the initial architecture for a requirements resolver based on BP002 and add support for packages
2. [DONE] Create a complete section in the User's Guide on how to handle test requirements.
3. [DONE] BP002: implement the file runner to support file requirements
4. [OPEN] BP002: implement the image runner to support image requirements
5. [OPEN] BP002: implement the git repository runner to support Git repository requirements
6. [OPEN] BP002: add support to JSON representation for requirements
7. [OPEN] BP002: add support to Python executable that produces JSON representation for requirements
8. [OPEN] BP002: add command-line parameter to support requirements file
9. [OPEN] BP002: add support to auto-detect requirements files based on file location
10. [OPEN] BP002 add support to run unsafe Python code that produces JSON representation for requirements
11. [OPEN] BP002: Create a new section in the Contributor's Guide on how to implement requirements runners

9.9.8 References

[1] - https://lists.gnu.org/archive/html/qemu-devel/2019-11/msg04074.html

[2] - https://docs.python.org/3/library/json.html

9.10 BP003

Number BP003

Title N(ext)Runner Task Life-Cycle

Author Cleber Rosa <crosa@redhat.com>

Discussions-To avocado-devel@redhat.com

Reviewers Beraldo Leal <bleal@redhat.com>, Willian Rampazzo <willianr@redhat.com>

Created 20-July-2020

Type Architecture Blueprint

Status WIP

Table of Contents

9.10.1 TL;DR

The N(ext) Runner has been used as Avocado's runner for selftests for over a year. The implementation used is based on the `avocado nrun` command, that is, outside of the Avocado's traditional `avocado run` entrypoint. Under the hood, it means that the N(ext) Runner is not integrated well enough with an Avocado Job.

A partial implementation of the N(ext) Runner integration with an Avocado Job is available at `avocado/plugins/runner_nrunner.py` but it has a number limitations.

The N(ext) Runner executes tests as *Tasks*. This blueprint describes the phases that a Task can be in throughout its life-cycle, and how the handling of these phases or states, will power the tests execution mechanism within the context of an Avocado Job.

9.10.2 Motivations

Propose an architecture for integrating the N(ext) Runner concepts and features into an Avocado Job. Because the N(ext) Runner contains distinguishing features that the original Avocado Job did not anticipate, a proxy layer is necessary.

The current runner (and Job) is built on the premises that there's a "currently executing test", and thus, does not need to keep track of various running tests states at once. The N(ext) Runner, on the other hand, support for running tests in parallel, and thus needs supporting code for keeping track of their state and forward their relevant information to an Avocado Job.

9.10.3 Goals of this BluePrint

1. Propose an architecture based on the life-cycle phases that an N(ext) Runner Task can go through while running under an Avocado Job.
2. Describe how the proposed architecture can power an implementation suitable for the next Avocado LTS release (82.0), having feature completeness when compared to the current runner, while still making its distinguishing features available to users who opt in. This also means that the current Avocado Job interface will continue to support the current runner implementation.
3. Prove that the current runner can be removed without significant user impact after the LTS release (within the 83.0 development cycle), based on the feature completeness of the N(ext) Runner with regards to its integration with an Avocado Job.
4. Allow for future extension of the Task life-cycle phases architecture, such as into a more capable and further reaching scheduler for Tasks. This means that this BluePrint is focused on short term integration issues, as describe in the motivation, but at the same tries to not impose future limitations to have new features implemented for other use cases.

9.10.4 Requirements

This section describes the requirements to manage the Task's life-cycle. It also describes the phases of a Task life-cycle and includes an example.

Task Execution Requirements Verification

For a Task to actually be executed, there needs to be a minimal number of requirements present. For instance, it's pointless to attempt to execute a Task of kind "custom" without either:

1. An `avocado-runner-custom` runner script that is compatible with the Avocado interface, OR
2. A `CustomRunner` runner class that is compatible with the *`avocado.core.nrunner.BaseRunner`* interface

Other types of Task Execution Requirements checks may be added in the future, but the core concept that a Task can not always be executed remains.

Currently, as per the `avocado nrun` implementation, this verification is done in a synchronous way, and it's of limited visibility to the user.

Requirements:

1. The verification of one Task's requirement should not block other Tasks from progressing to other phases.
2. The user interface should provide more information on tasks that either failed the verification or that still going through the verification process.

Parallelization and Result Events

The N(ext) Runner allows for the parallel execution of tasks. When integrated into a Job, it means there can be more than one test running at a given time.

Currently, plugins that implement the *avocado.core.plugin_interfaces.ResultEvents* interface may contain logic that assumes that the same test will have `start_test`, `test_progress` and then `end_test` methods called in that particular order, and only then another test will have any of those called on its behalf.

For instance, the Human UI plugin will currently:

1. Print a line such as `(1/1) /bin/true:` when a test starts, that is, when *avocado.core.plugin_interfaces.ResultEvents.start_test()* is called.
2. Add a throbber and/or change its state whenever a progress update is received, that is, when *avocado.core.plugin_interfaces.ResultEvents.test_progress()* is called.
3. Add a test result such as `PASS (0.01 s)` when the test finishes, that is, when *avocado.core.plugin_interfaces.ResultEvents.end_test()* is called.

Other implementations, such as the TAP result plugin, will only print a line when the final test result is known.

Requirement: have no conflicts of test information when more than one is running in parallel.

Requirement example: provide the test progress notification and the final test result information "in line" with the correct test indication (if given earlier).

Note: Ideally, this shouldn't require a change to the interface, but only within the implementation so that the presentation of coherent test result events is achieved.

Non-blocking Parallelization

As stated earlier, the N(ext) Runner allows for the parallel execution of tasks. A given Task should be allowed to be executed as early as possible, provided:

1. Its requirements (such as its specific test runners) are available.
2. A limit for concurrently running tasks has not been reached.

Requirement: there should be no artificial and unnecessary blocking of the parallelization level.

Requirement example: if an hypothetical Result Events plugin interacts with a high latency server, and such interaction takes 2 minutes, the execution of new tasks should not be affected by it.

Note: There are a number of strategies for concurrent programming in Python these days, and the "avocado nrun" command currently makes use of asyncio to have coroutines that spawn tasks and collect results concurrently (in a preemptive cooperative model). The actual tools/libraries used in the implementation shall be discussed later.

Passive Task Status Collection

The N(ext) Runner architecture allows tests to run in a much more decoupled way, because of a number of its characteristics, including the fact that Tasks communicate their status by sending asynchronous messages.

Note: The current implementation uses network sockets as the transport for these messages, in part for its universal aspect, and in part to enforce this decoupling. Future implementations may provide alternate transports, such as file descriptors, serial connections, etc.

There currently is a component used for a similar role used in `avocado nrun`: `avocado.core.nrunner.StatusServer`, but it exceeds what's needed here in some aspects, and lacks in others aspects.

Requirement: have a mechanism that can receive and collect in an organized manner, all the state messages coming from tasks that are part of an Avocado Job.

Requirement example: the Avocado Job should be able to use the collection of task status information to ask questions such as the following.

1. When was the last time that task "123-foobar" gave an status update? Such information would be useful to determine if the task should be abandoned or destroyed as part of a timeout handling, as described in the later section about Task Monitoring and Termination.
2. Has the task "123-foobar" given a final status update? That is, can we conclude that, as a Task, regardless of the success or failure of what it ran, it finished its execution? Such information would be useful to post the final test result to the Job results and `ResultEvent` plugins, as described in the next section.

Proxy from Task Status To Job Result

An Avocado Job contains an *`avocado.core.result.Result`* which tallies the overall job results. But, the state messages coming from Tasks are not suitable to being given directly to methods such as `avocado.core.result.Result.check_state()`. A mechanism is needed to proxy and convert the relevant message and events to the current Avocado job result and `ResultEvents` plugins.

Requirements:

1. Proxy Task Status messages and convert them into the appropriate information suitable for *`avocado.core.result.Result`*.
2. Allow *`ResultEvents`* plugins to act as soon as possible on relevant status messages;

Task Monitoring and Termination

The N(ext) Runner architecture, as stated before, can have tasks running without much, if any, contact with an Avocado Job. But, an Avocado Job must have a beginning and end, and with that it's necessary to monitor tasks, and if their situation is not clear, decide their fate.

For instance, a Task started as part of an Avocado Job may communicate the following messages:

```
{'status': 'started', 'time': 1596680574.8790667, 'output_dir': '/tmp/.avocado-task-
↪d8w0k9s1', 'id': '1-/bin/sleep'}
{'status': 'running', 'time': 1596680574.889258, 'id': '1-/bin/sleep'}
```

Then it may go offline for eternity. The possible reasons are varied, and despite them, the Job will eventually have to deal the non-final, unknown state of tasks and given them a resolution.

Note: The Spanwer may be able to provide additional information that will help to decide the handling given to such as Task (or its recorded final status). For instance, if a Task running on a container is not communicating its status, and its verified that the container has finished its execution, it may be wise to not wait for the timeout.

Requirements:

1. Monitor the execution of a task (from an external PoV).
2. Unless it proves to be, say because of complexity or impossibilities when interacting with the spawners, tasks that are unresponsive should attempt to be terminated.
3. Notify the user if stray tasks should be clean up manually. This may be, for instance, necessary if a Task on a container seems to be stuck, and the container can not be destroyed. The same applies to a process in an uninterruptible sleep state.
4. Update Job result with the information about monitored tasks.

Note: Tasks going through the usual phases will end up having their final state in the going through the task status collection described earlier, and from there have them proxied/converted into the Job result and plugins. At first sight, it seems that the task monitoring should use the same repository of status and update it in a similar way, but on behalf of the "lost/exterminated task".

9.10.5 Suggested Terminology for the Task Phases

Task execution has been requested

A Task whose execution was requested by the user. All of the tasks on a Job's `test_suite` attribute are requested tasks.

If a software component deals with this type of task, it's advisable that it refers to `TASK_REQUESTED` or `requested_tasks` or a similar name that links to this definition.

Task is being triaged

The details of the task are being analyzed, including, and most importantly, the ability of the system to run it. A task that leaves triage, and it's either considered `FINISHED` because it can not be executed, or is `READY` and waits to be executed.

If a software component deals with this type of task, for instance, if a "task scheduler" is looking for runners matching the Task's kind, it should keep it under a `tasks_under_triage` or mark the tasks as `TASK_UNDER_TRIAGE` or `TASK_TRIAGING` a similar name that links to this definition.

Task is ready to be started

Task has been triaged, and as much as the system knows, it's ready to be executed. A task may be in this phase for any amount of time, given that the capacity to have an additional task started is dynamic and may be enforced here.

If a software component deals with this type of task, it should keep it under a `tasks_ready` or mark the tasks as `TASK_READY` or a similar name that links to this definition.

Task has been started

A task was successfully started by a spawner.

Note that it does *not* mean that the test that the task runner (say, an `avocado-runner-$kind task-run` command) will run has already started. This will be signaled by a runner, say `avocado-runner-$kind` producing an `status: started` kind of status message.

If a software component deals with this type of task, it should keep it under a `tasks_started` or mark the tasks as `TASK_STARTED` or a similar name that links to this definition.

Task is finished

This means that there's no longer any activity or a new phase for this task to move to.

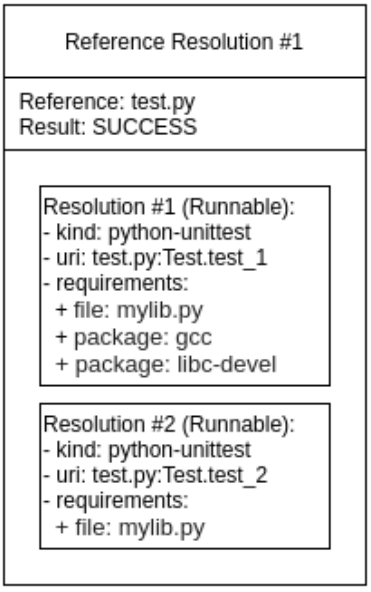
It's expected that extra information will be available explaining how/why the task arrived in this phase. For instance, it may have come from the `TASK_TRIAGING` phase and never gone through the `TASK_STARTED` phase. Alternatively, it may been in the `TASK_STARTED` phase and finished without any errors.

It should be kept under a `tasks_finished` structure or be marked as `TASK_FINISHED` or a similar name that links to this definition.

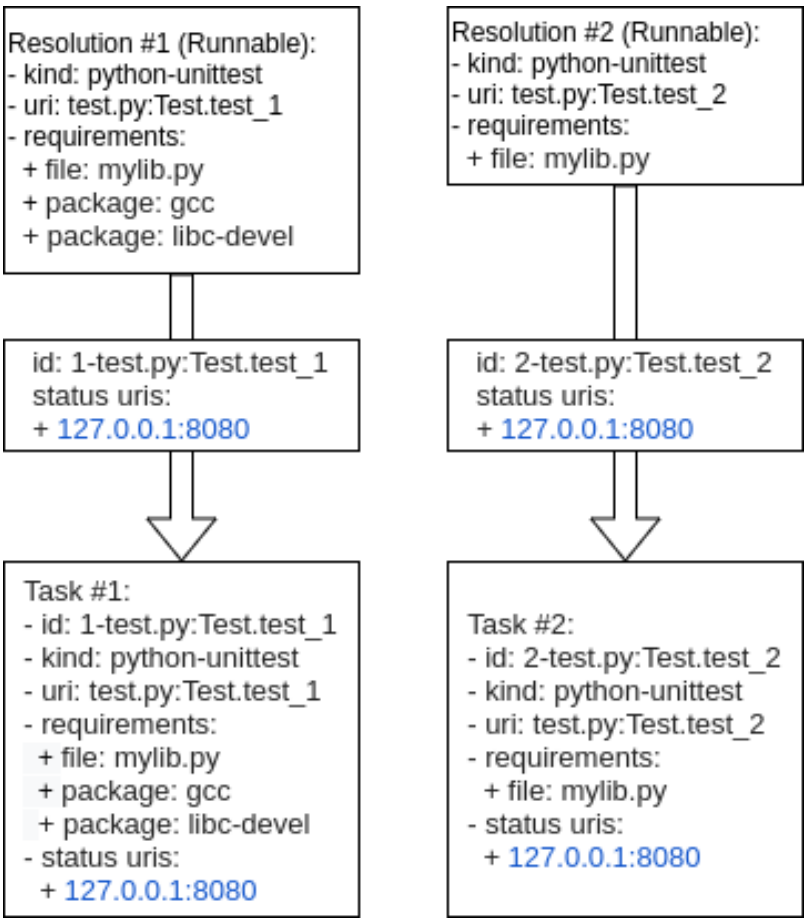
Note: There's no associated meaning here about the pass/fail output of the test payload executed by the task.

9.10.6 Task life-cycle example

A task will usually be created from a Runnable. A Runnable will, in turn, almost always be created as part of the "avocado.core.resolver" module. Let's consider the following output of a resolution:

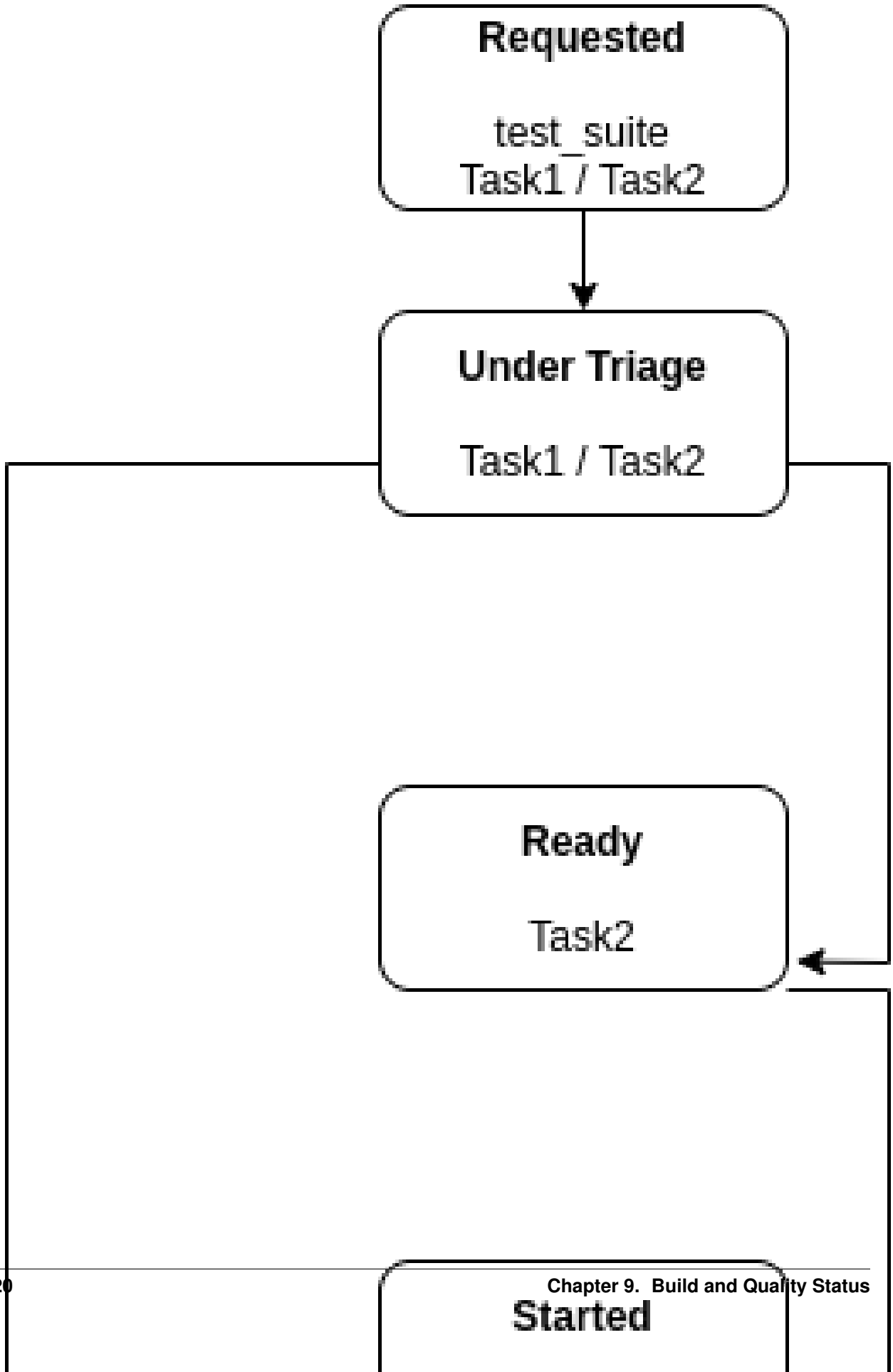


Two Runnables here will be transformed into Tasks. The process usually includes adding an identification and a status URI:



In the end, a job will contain a `test_suite` with "Task #1" and "Task #2". It means that the execution of both tasks was requested by the Job owner.

These tasks will now be triaged. A suitable implementation will move those tasks to a `tasks_under_triage` queue, mark them as `TASK_UNDER_TRIAGE` or some other strategy to differentiate the tasks at this stage.


Requested
test_suite
Task1 / Task2
Under Triage
Task1 / Task2
Ready
Task2
Started

Iteration I

Task #1 is selected on the first iteration, and it's found that:

1. A suitable runner for tasks of kind `python-unittest` exists;
2. The `mylib.py` requirement is already present on the current environment;
3. The `gcc` and `libc-devel` packages are not installed in the current environment;

Task #1 is not ready to be executed, so it's moved to `TASK_FINISHED` and it's reason is recorded.

No further action is performed on the first iteration because no other relevant state exists (Task #2, the only other requested task, has not progressed beyond its initial stage).

Iteration II

On the second iteration, Task #2 is selected, and it's found that:

1. A suitable runner for tasks of kind `python-unittest` exists;
2. The `mylib.py` requirement is already present on the current environment.

Task #2 is now ready to be started.

As a reminder, Task #1 has not passed triaging and is `TASK_FINISHED`.

Iteration III

On the third iteration, there are no tasks left under triage, so the action is now limited to tasks being prepared and ready to be started.

Note: As an optimization, supposing that the "status uri" 127.0.0.1:8080, was set by the job, as its internal status server, it must be started before any task, to avoid any status message being lost. Without such an optimization, the status server could be started earlier.

At this stage, Task #2 has been started.

Final Iteration

On the fifth iteration, the spawner reports that Task #2 is not alive anymore, and the status server has received a message about it.

Because of that, Task #2 is now considered `TASK_FINISHED`.

Tallying results

The nrunner plugin should be able to provide meaningful results to the Job, and consequently, to the user, based on the resulting information on the final iteration.

Notice that some information, such as the `PASS` for the second test, will come from the "result" given in a status message from the task itself. Some other status, such as the `CANCEL` status for the first test will not come from a status message received, but from a realization of the actual management of the task execution. It's expected to other information will also have to be inferred, and "filled in" by the nrunner plugin implementation.

In the end, it's expected that results similar to this would be presented:

```
JOB ID     : f59bd40b8ac905864c4558dc02b6177d4f422ca3
JOB LOG    : /home/user/avocado/job-results/job-2020-05-20T17.58-f59bd40/job.log
 (1/2) tests.py:Test.test_1: CANCEL (0 s)
 (1/2) tests.py:Test.test_2: PASS (2.56 s)
RESULTS    : PASS 1 | ERROR 0 | FAIL 0 | SKIP 0 | WARN 0 | INTERRUPT 0 | CANCEL 1
JOB TIME   : 0.19 s
JOB HTML   : /home/user/avocado/job-results/job-2020-05-20T17.58-f59bd40/results.html
```

Notice that Task #2 may show up before Task #1. There may be issues associated with the current UI to deal with regarding out of order task status updates.

9.10.7 Implementation Example

The following implementation example uses random sleeps and random (but biased) results from operations expected to happen on different phases, to simulate the behavior of real tasks.

The enforcement of some artificial limits (such as the number of tasks `TASK_STARTED`) is also exemplified. As a general rule, all tasks are attempted to be moved further into their life-cycle and a number of "workers" doing that should not conflict with each other.

This implementation uses Python's `asyncio` library very crudely. The final implementation may use other tools, such as a `asyncio.Queue` instead of plain lists with a `asyncio.Lock`. It may also use individual `Tasks` for each work in each phase.

```
import asyncio
import itertools
import random
import time

from avocado.utils.astring import tabular_output

DEBUG = False


def debug(msg):
    if DEBUG:
        print(msg)


async def sleep_random():
    await asyncio.sleep(random.random())


def true_or_false(handicap=3):
    """Returns a random positive or negative outcome, with some bias."""
    if handicap > 1:
        choices = [True] + ([False] * handicap)
    else:
        choices = [False] + ([True] * abs(handicap))
    return random.choice(choices)


def mock_check_task_requirement():
    # More success than failures, please
    return true_or_false(-8)
```

(continues on next page)

(continued from previous page)

```
def mock_check_task_start():
    # More success than failures, please
    return true_or_false(-6)


def mock_monitor_task_finished():
    # More failures than successes, please
    return true_or_false(5)


class Task:
    """Used here as a placeholder for an avocado.core.nrunner.Task."""

    def __init__(self, identification):
        self._identification = identification


class TaskInfo(Task):
    """Task with extra status information on its life-cycle.

    The equivalent of a StatusServer will contain this information
    in the real implementation."""

    def __init__(self, identification):
        super(TaskInfo, self).__init__(identification)
        self._status = None
        self._timeout = None

    @property
    def status(self):
        return self._status

    @status.setter
    def status(self, status):
        self._status = status

    @property
    def timeout(self):
        return self._timeout

    @timeout.setter
    def timeout(self, timeout):
        self._timeout = timeout

    def __repr__(self):
        if self._status is None:
            return '%s' % self._identification
        else:
            return '%s (%s)' % (self._identification,
                                self.status)


class TaskStateMachine:
    """Represents all phases that a task can go through its life."""

    def __init__(self, tasks):
```

(continues on next page)

(continued from previous page)

```
        self._requested = tasks
        self._triaging = []
        self._ready = []
        self._started = []
        self._finished = []
        self._lock = asyncio.Lock()

    @property
    def requested(self):
        return self._requested

    @property
    def triaging(self):
        return self._triaging

    @property
    def ready(self):
        return self._ready

    @property
    def started(self):
        return self._started

    @property
    def finished(self):
        return self._finished

    @property
    def lock(self):
        return self._lock

    @property
    async def complete(self):
        async with self._lock:
            pending = any([self._requested, self._triaging,
                           self._ready, self._started])
        return not pending

    def __str__(self):
        headers = ("|_REQUESTED_|", "|_TRIAGING__|",
                   "|___READY___|", "|__STARTED__|",
                   "|_______FINISHED________|")
        data = itertools.zip_longest(self._requested, self._triaging, self._ready,
                                     self._started, self._finished, fillvalue="")
        matrix = [_ for _ in data]
        return tabular_output(matrix, headers)


async def bootstrap(lc):
    """Reads from requested, moves into triaging."""
    # fake some rate limiting
    if true_or_false(10):
        return
    try:
        async with lc.lock:
            task = lc.requested.pop()
            lc.triaging.append(task)
```

(continues on next page)

(continued from previous page)

```
            debug('Moved Task %s: REQUESTED => TRIAGING' % task)
    except IndexError:
        debug('BOOTSTRAP: nothing to do')
        return


async def triage(lc):
    """Reads from triaging, moves into either: ready or finished."""
    await sleep_random()
    try:
        async with lc.lock:
            task = lc.triaging.pop()
    except IndexError:
        debug('TRIAGE done')
        return

    if mock_check_task_requirement():
        async with lc.lock:
            lc.ready.append(task)
            debug('Moving Task %s: TRIAGING => READY' % task)
    else:
        async with lc.lock:
            lc.finished.append(task)
            task.status = 'FAILED ON TRIAGE'
            debug('Moving Task %s: TRIAGING => FINISHED' % task)


async def start(lc):
    """Reads from ready, moves into either: started or finished."""
    await sleep_random()
    try:
        async with lc.lock:
            task = lc.ready.pop()
    except IndexError:
        debug('START: nothing to do')
        return

    # enforce a rate limit on the number of started (currently running) tasks.
    # this is a global limit, but the spawners can also be queried with regards
    # to their capacity to handle new tasks
    MAX_RUNNING_TASKS = 8
    async with lc.lock:
        if len(lc.started) >= MAX_RUNNING_TASKS:
            lc.ready.insert(0, task)
            task.status = 'WAITING'
            return

    # suppose we're starting the tasks
    if mock_check_task_start():
        async with lc.lock:
            task.status = None
            # Let's give each task 15 seconds from start time
            task.timeout = time.monotonic() + 15
            lc.started.append(task)
            debug('Moving Task %s: READY => STARTED' % task)
    else:
        async with lc.lock:
```

(continues on next page)

(continued from previous page)

```
            lc.finished.append(task)
            task.status = 'FAILED ON START'
            debug('Moving Task %s: READY => FINISHED (ERRORED ON START)' % task)


async def monitor(lc):
    """Reads from started, moves into finished."""
    await sleep_random()
    try:
        async with lc.lock:
            task = lc.started.pop()
    except IndexError:
        debug('MONITOR: nothing to do')
        return

    if time.monotonic() > task.timeout:
        async with lc.lock:
            task.status = 'FAILED W/ TIMEOUT'
            lc.finished.append(task)
            debug('Moving Task %s: STARTED => FINISHED (FAILED ON TIMEOUT)' % task)
    elif mock_monitor_task_finished():
        async with lc.lock:
            lc.finished.append(task)
            debug('Moving Task %s: STARTED => FINISHED (COMPLETED AFTER STARTED)' %␣
↪task)
    else:
        async with lc.lock:
            lc.started.insert(0, task)
        debug('Task %s: has not finished yet' % task)


def print_lc_status(lc):
    print("\033c", end="")
    print(str(lc))


async def worker(lc):
    """Pushes Tasks forward and makes them do something with their lives."""
    while True:
        complete = await lc.complete
        debug('Complete? %s' % complete)
        if complete:
            break
        await bootstrap(lc)
        print_lc_status(lc)
        await triage(lc)
        print_lc_status(lc)
        await start(lc)
        print_lc_status(lc)
        await monitor(lc)
        print_lc_status(lc)


if __name__ == '__main__':
    NUMBER_OF_TASKS = 40
    NUMBER_OF_LIFECYCLE_WORKERS = 4
    tasks_info = [TaskInfo("%03i" % _) for _ in range(1, NUMBER_OF_TASKS - 1)]
```

(continues on next page)

(continued from previous page)

```
    state_machine = TaskStateMachine(tasks_info)
    loop = asyncio.get_event_loop()
    workers = [loop.create_task(worker(state_machine))
               for _ in range(NUMBER_OF_LIFECYCLE_WORKERS)]
    loop.run_until_complete(asyncio.gather(*workers))
    print("JOB COMPLETED")
```

9.10.8 Backwards Compatibility

The compatibility of the resulting Job compatible runner implementation with the current runner is to be verified by running the same set of "Job API feature tests", but with this runner selected instead.

There are no compatibility issues with the previous versions of itself, or with the non-Job compatible `nrun` implementation.

9.10.9 Security Implications

None that we can determine at this point.

9.10.10 How to Teach This

The distinctive features that the N(ext) Runner provides should be properly documented.

Users should not be required to learn about the N(ext) Runner features to use it just as an alternative to the current runner implementation.

9.10.11 Related Issues

Current issues that are expected to be solved when this blueprint is implemented:

1. Have a passive Task Status collection server implementation.
2. Have a Task Life Cycle / State Machine implementation.
3. Have Spawner features to check the status (alive or not) for Tasks. This is intended to be used in place or in addition of the status messages from Tasks, when they failed to be generated by tasks or received by the Task Status collection server.
4. Have Spawner features to destroy (best effort) stray Tasks.
5. Fully integrate the N(ext) Runner into the Avocado Job and command line app, that is, as a general rule all features of the current runner should be present when the N(ext) Runner is used in a job.

9.10.12 Future work

These are possible future improvements to the Task phases, and may be a partial list of addition towards a more comprehensive "Task scheduler". They are provided for discussion only and do not constitute hard requirements for this or future work.

Tasks' requirements fulfillment

1. Prepare for the execution of a task, such as the fulfillment of extra task requirements. The requirements resolver is one, if not the only way, component that should be given a chance to act here;

2. Executes a task in a prepared environment;

Active Task Status Collection

Some environments and use cases may require disconnected execution of tasks. In such cases, a Job will have to activelly poll for tasks' statuses, which may be:

1. an operation that happens along the task execution.
2. only at the end of the task execution, as signalled by the termination of the environment in which a task is running on.

9.10.13 References

- RFC: https://www.redhat.com/archives/avocado-devel/2020-May/msg00015.html
- Early implementation: https://github.com/avocado-framework/avocado/pull/3765
- Requirement check prototype: https://github.com/avocado-framework/avocado/pull/4015

9.11 Other Resources

9.11.1 Open Source Projects Relying on Avocado

The following is a partial list of projects that use Avocado as either the framework for their tests, or the Avocado test runner to run other regular tests.

Fedora Modularity

The Fedora Modularity project is about building a modular Linux OS with multiple versions of components on different lifecycles.

It uses Avocado in its meta test family subproject.

QEMU

QEMU is a generic and open source machine emulator and virtualizer.

It uses Avocado in functional level tests.

SoS

SoS is an extensible, portable, support data collection tool primarily aimed at Linux distributions and other UNIX-like operating systems.

It uses Avocado in its functional level tests.

DAOS

The Distributed Asynchronous Object Storage (DAOS) is an open-source object store designed from the ground up for massively distributed Non Volatile Memory (NVM).

It uses Avocado in its ftest test suite.

Falco

Falco, the cloud-native runtime security project, is the de facto Kubernetes threat detection engine.

It uses Avocado in its regression test suite.

RUDDER

RUDDER is a European, open source and multi-platform solution allowing you to manage configurations and compliance of your systems.

It uses Avocado in its ncf project, which is a framework that runs in pure CFEngine language, to help structure your CFEngine policy and provide reusable, single purpose components.

POK

POK is a real-time embedded operating system for safety-critical systems.

It uses Avocado in its unitary and multiprocessing unitary tests.

9.11.2 Avocado extensions

The following are extensions of the Avocado framework specifically designed to enhance Avocado with more targeted testing capabilities.

Avocado-VT

Avocado-VT lets you execute virtualization related tests (then known as virt-test), with all conveniences provided by Avocado.

Together with its various test providers (QEMU, LibVirt) it provides literally dozens of thousands of virtualization related tests.

Avocado-I2N

Avocado-I2N is a plugin that extends Avocado-VT with automated vm state setup, inheritance, and traversal.

Avocado-cloud

Avocado-cloud is a cloud test suite for RHEL guests on various clouds such as Alibaba, AWS, Azure, Huawei, IBM Cloud and OpenStack.

Test specific repositories

These repositories contain a multitude of tests for specific different purposes.

- Avocado Misc Tests: a repository dedicated to host tests initially ported from autotest client tests repository, but not limited to those.
- OpenPOWER Host OS and Guest Virtual Machine (VM) stability tests

9.11.3 Presentations

This is a collection of some varied Avocado related presentations on the web:

- Testing Framework Internals (DevConf 2017)
- Auto Testing for AArch64 Virtualization (Linaro connect San Francisco 2017)
- libvirt integration and testing for enterprise KVM/ARM (Linaro Connect Budapest 2017)
- Automated Testing Framework (PyCon CZ 2016)
- Avocado and Jenkins (DevConf 2016)
- Avocado: Next Gen Testing Toolbox (DevConf 2015)
- Avocado workshop (DevConf 2015) mindmap with all commands/content and a partial video
- Avocado: Open Source Testing Made Easy (LinuxCon 2015)

9.12 Avocado's Configuration Reference

This is current Avocado Configuration reference. You can adjust the values by two ways:

- Configuration file options;
- Command-line options (when available)

Some options that are used often are available for your convenience also at the command-line. This list has all options registered with Avocado so far.

Note: Please, keep in mind that we are in constant evolution and doing a huge improvements on how to configure Avocado, some options here can be changed in the near future.

9.12.1 assets.fetch.ignore_errors

always return success for the fetch command.

- Default: False
- Type: <class 'bool'>

9.12.2 assets.fetch.references

Path to avocado instrumented test

- Default: []
- Type: <class 'list'>

9.12.3 assets.fetch.timeout

Timeout to be used when download an asset.

- Default: 300
- Type: <class 'int'>

9.12.4 assets.list.days

How old (in days) should Avocado look for assets?

- Default: None
- Type: <class 'int'>

9.12.5 assets.list.overall_limit

Filter will be based on a overall system limit threshold in bytes (with assets ordered by last access) or with a suffix unit. Valid suffixes are: b,k,m,g,t

- Default: None
- Type: <class 'str'>

9.12.6 assets.list.size_filter

Apply action based on a size filter (comparison operator + value) in bytes. Ex '>20', '<=200'. Supported operators: ==, <, >, <=, >=

- Default: None
- Type: <class 'str'>

9.12.7 assets.purge.days

How old (in days) should Avocado look for assets?

- Default: None
- Type: <class 'int'>

9.12.8 assets.purge.overall_limit

Filter will be based on a overall system limit threshold in bytes (with assets ordered by last access) or with a suffix unit. Valid suffixes are: b,k,m,g,t

- Default: None
- Type: <class 'str'>

9.12.9 assets.purge.size_filter

Apply action based on a size filter (comparison operator + value) in bytes. Ex '>20', '<=200'. Supported operators: ==, <, >, <=, >=

- Default: None
- Type: <class 'str'>

9.12.10 assets.register.name

Unique name to associate with this asset.

- Default: None
- Type: <class 'str'>

9.12.11 assets.register.sha1_hash

SHA1 hash of this asset.

- Default: None
- Type: <class 'str'>

9.12.12 assets.register.url

Path to asset that you would like to register manually.

- Default: None
- Type: <class 'str'>

9.12.13 config.datadir

Shows the data directories currently being used by Avocado

- Default: False
- Type: <class 'bool'>

9.12.14 core.input_encoding

The encoding used by default on all data input

- Default: utf-8
- Type: <class 'str'>

9.12.15 core.paginator

Turn the paginator on. Useful when output is too long.

- Default: False
- Type: <class 'bool'>

9.12.16 core.show

List of comma separated builtin logs, or logging streams optionally followed by LEVEL (DEBUG,INFO,...). Builtin streams are: "app": application output; "test": test output; "debug": tracebacks and other debugging info; "early": early logging of other streams, including test (very verbose); "all": all builtin streams; "none": disables regular output (leaving only errors enabled). By default: 'app'

- Default: {'app'}
- Type: <function register_core_options.<locals>.<lambda> at 0x7f20f061e5f0>

9.12.17 core.verbose

Some commands can produce more information. This option will enable the verbosity when applicable.

- Default: False
- Type: <class 'bool'>

9.12.18 datadir.paths.base_dir

Base directory for Avocado tests and auxiliary data

- Default: /home/docs/avocado
- Type: <function prepend_base_path at 0x7f20eca167a0>

9.12.19 datadir.paths.cache_dirs

Cache directories to be used by the avocado test

- Default: ['/home/docs/avocado/data/cache']
- Type: <class 'list'>

9.12.20 datadir.paths.data_dir

Data directory for Avocado

- Default: /home/docs/avocado/data
- Type: <function prepend_base_path at 0x7f20eca167a0>

9.12.21 datadir.paths.logs_dir

Logs directory for Avocado

- Default: /home/docs/avocado/job-results
- Type: <function prepend_base_path at 0x7f20eca167a0>

9.12.22 datadir.paths.test_dir

Test directory for Avocado tests

- Default: /usr/share/doc/avocado/tests
- Type: <function prepend_base_path at 0x7f20eca167a0>

9.12.23 diff.create_reports

Create temporary files with job reports to be used by other diff tools

- Default: False
- Type: <class 'bool'>

9.12.24 diff.filter

Comma separated filter of diff sections: (no)cmdline,(no)time,(no)variants,(no)results, (no)config,(no)sysinfo (defaults to all enabled).

- Default: ['cmdline', 'time', 'variants', 'results', 'config', 'sysinfo']
- Type: <function Diff._validate_filters at 0x7f20e93fe9e0>

9.12.25 diff.html

Enable HTML output to the FILE where the result should be written.

- Default: None
- Type: <class 'str'>

9.12.26 diff.jobids

A job reference, identified by a (partial) unique ID (SHA1) or test results directory.

- Default: []
- Type: <class 'list'>

9.12.27 diff.open_browser

Generate and open a HTML report in your preferred browser. If no –html file is provided, create a temporary file.

- Default: False
- Type: <class 'bool'>

9.12.28 diff.strip_id

Strip the "id" from "id-name;variant" when comparing test results.

- Default: False
- Type: <class 'bool'>

9.12.29 distro.distro_def_arch

Primary architecture that the distro targets

- Default:
- Type: <class 'str'>

9.12.30 distro.distro_def_create

Creates a distro definition file based on the path given.

- Default: False
- Type: <class 'bool'>

9.12.31 distro.distro_def_name

Distribution short name

- Default:
- Type: <class 'str'>

9.12.32 distro.distro_def_path

Top level directory of the distro installation files

- Default:
- Type: <class 'str'>

9.12.33 distro.distro_def_release

Distribution release version number

- Default:
- Type: <class 'str'>

9.12.34 distro.distro_def_type

Distro type (one of: rpm, deb)

- Default:
- Type: <class 'str'>

9.12.35 distro.distro_def_version

Distribution major version name

- Default:
- Type: <class 'str'>

9.12.36 filter.by_tags.include_empty

Include all tests without tags during filtering. This effectively means they will be kept in the test suite found previously to filtering.

- Default: False
- Type: <class 'bool'>

9.12.37 filter.by_tags.include_empty_key

Include all tests that do not have a matching key in its key:val tags. This effectively means those tests will be kept in the test suite found previously to filtering.

- Default: False
- Type: <class 'bool'>

9.12.38 filter.by_tags.tags

Filter tests based on tags

- Default: []
- Type: <class 'list'>

9.12.39 human_ui.omit.statuses

Status that will be omitted from the Human UI. Valid statuses: SKIP, ERROR, FAIL, WARN, PASS, INTERRUPTED, CANCEL, STARTED

- Default: []
- Type: <class 'list'>

9.12.40 job.output.loglevel

Sets the base log level of the output generated by the job, which is also the base logging level for the –show command line option. Any of the Python logging levels names are allowed here. Examples: DEBUG, INFO, WARNING, ERROR, CRITICAL. For more information refer to: https://docs.python.org/3/library/logging.html#levels

- Default: DEBUG
- Type: <class 'str'>

9.12.41 job.output.testlogs.logfiles

The specific log files that will be shown for tests whose exit status match the ones defined in the "job.output.testlogs.statuses" configuration.

- Default: ['debug.log']
- Type: <class 'list'>

9.12.42 job.output.testlogs.statuses

Status that will trigger the output of a test's logs after the job ends. Valid statuses: SKIP, ERROR, FAIL, WARN, PASS, INTERRUPTED, CANCEL

- Default: []
- Type: <class 'list'>

9.12.43 job.replay.source_job_id

Replays a job, identified by: complete or partial Job ID, "latest" for the latest job, the job results path.

- Default: latest
- Type: <class 'str'>

9.12.44 job.run.result.html.enabled

Enables default HTML result in the job results directory. File will be named "results.html".

- Default: True
- Type: <class 'bool'>

9.12.45 job.run.result.html.open_browser

Open the generated report on your preferred browser. This works even if –html was not explicitly passed, since an HTML report is always generated on the job results dir.

- Default: False
- Type: <class 'bool'>

9.12.46 job.run.result.html.output

Enable HTML output to the FILE where the result should be written. The value - (output to stdout) is not supported since not all HTML resources can be embedded into a single file (page resources will be copied to the output file dir)

- Default: None
- Type: <class 'str'>

9.12.47 job.run.result.json.enabled

Enables default JSON result in the job results directory. File will be named "results.json".

- Default: True
- Type: <class 'bool'>

9.12.48 job.run.result.json.output

Enable JSON result format and write it to FILE. Use "-" to redirect to the standard output.

- Default: None
- Type: <class 'str'>

9.12.49 job.run.result.tap.enabled

Enables default TAP result in the job results directory. File will be named "results.tap"

- Default: True
- Type: <class 'bool'>

9.12.50 job.run.result.tap.include_logs

Include test logs as comments in TAP output.

- Default: False
- Type: <class 'bool'>

9.12.51 job.run.result.tap.output

Enable TAP result output and write it to FILE. Use "-" to redirect to standard output.

- Default: None
- Type: <class 'str'>

9.12.52 job.run.result.xunit.enabled

Enables default xUnit result in the job results directory. File will be named "results.xml".

- Default: True
- Type: <class 'bool'>

9.12.53 job.run.result.xunit.job_name

Override the reported job name. By default uses the Avocado job name which is always unique. This is useful for reporting in Jenkins as it only evaluates first-failure from jobs of the same name.

- Default: None
- Type: <class 'str'>

9.12.54 job.run.result.xunit.max_test_log_chars

Limit the attached job log to given number of characters (k/m/g suffix allowed)

- Default: 100000
- Type: <function XUnitInit.initialize.<locals>.<lambda> at 0x7f20eb5d2290>

9.12.55 job.run.result.xunit.output

Enable xUnit result format and write it to FILE. Use "-" to redirect to the standard output.

- Default: None
- Type: <class 'str'>

9.12.56 job.run.store_logging_stream

Store given logging STREAMs in "$JOB_RESULTS_DIR/$STREAM.$LEVEL."

- Default: ['avocado.core:DEBUG']
- Type: <class 'list'>

9.12.57 job.run.timeout

Set the maximum amount of time (in SECONDS) that tests are allowed to execute. Values <= zero means "no timeout". You can also use suffixes, like: s (seconds), m (minutes), h (hours).

- Default: 0
- Type: <function time_to_seconds at 0x7f20eb373e60>

9.12.58 jobs.get.output_files.destination

Destination path

- Default: None
- Type: <class 'str'>

9.12.59 jobs.get.output_files.job_id

JOB id

- Default: None
- Type: <class 'str'>

9.12.60 jobs.show.job_id

JOB id

- Default: latest
- Type: <class 'str'>

9.12.61 json.variants.load

Load the Variants from a JSON serialized file

- Default: None
- Type: <class 'str'>

9.12.62 list.compatiblity_with_resolver_noop

Uses the Avocado resolver method (part of the nrunner architecture) to detect tests. This is enabled by default and exists only for compatibility purposes, and will be removed soon. To use the legacy (loader) method for finding tests, set the "–loader" option

- Default: True
- Type: <class 'bool'>

9.12.63 list.external_runner

Path to an specific test runner that allows the use of its own tests. This should be used for running tests that do not conform to Avocado's SIMPLE test interface and can not run standalone. Note: the use of –external-runner overwrites the –loaders to 'external_runner'

- Default: None
- Type: <class 'str'>

9.12.64 list.external_runner_chdir

Change directory before executing tests. This option may be necessary because of requirements and/or limitations of the external test runner. If the external runner requires to be run from its own base directory, use 'runner' here. If the external runner runs tests based on files and requires to be run from the directory where those files are located, use 'test' here and specify the test directory with the option '–external-runner-testdir'.

- Default: None
- Type: <class 'str'>

9.12.65 list.external_runner_testdir

Where test files understood by the external test runner are located in the filesystem. Obviously this assumes and only applies to external test runners that run tests from files

- Default: None
- Type: <class 'str'>

9.12.66 list.loaders

Overrides the priority of the test loaders. You can specify either @loader_name or TEST_TYPE. By default it tries all available loaders according to priority set in settings->plugins.loaders.

- Default: ['file', '@DEFAULT']
- Type: <class 'list'>

9.12.67 list.recipes.write_to_directory

Writes runnable recipe files to a directory. Valid only when using –resolver.

- Default: None
- Type: <class 'str'>

9.12.68 list.references

List of test references (aliases or paths). If empty, Avocado will list tests on the configured test source, (see "avocado config –datadir") Also, if there are other test loader plugins active, tests from those plugins might also show up (behavior may vary among plugins)

- Default: []
- Type: <class 'list'>

9.12.69 list.resolver

Uses the Avocado legacy (loader) method for finding tests. This option will exist only for a transitional period until the legacy (loader) method is deprecated and removed

- Default: True
- Type: <class 'bool'>

9.12.70 list.write_to_json_file

Writes output to a json file.

- Default: None
- Type: <class 'str'>

9.12.71 nrunner.max_parallel_tasks

Number of maximum number tasks running in parallel. You can disable parallel execution by setting this to 1. Defaults to the amount of CPUs on this machine.

- Default: 2
- Type: <class 'int'>

9.12.72 nrunner.shuffle

Shuffle the tasks to be executed

- Default: False
- Type: <class 'bool'>

9.12.73 nrunner.spawner

Spawn tasks in a specific spawner. Available spawners: 'process' and 'podman'

- Default: process
- Type: <class 'str'>

9.12.74 nrunner.status_server_auto

If the status server should automatically choose a "status_server_listen" and "status_server_uri" configuration. Default is to auto configure a status server.

- Default: True
- Type: <class 'bool'>

9.12.75 nrunner.status_server_buffer_size

Buffer size that status server uses. This should generally not be a concern to most users, but it can be tuned in case a runner generates very large status messages, which is common if a test generates a lot of output. Default is 33554432 (32MiB)

- Default: 33554432
- Type: <class 'int'>

9.12.76 nrunner.status_server_listen

URI for listing the status server. Usually a "HOST:PORT" string

- Default: 127.0.0.1:8888
- Type: <class 'str'>

9.12.77 nrunner.status_server_uri

URI for connecting to the status server, usually a "HOST:PORT" string. Use this if your status server is in another host, or different port

- Default: 127.0.0.1:8888
- Type: <class 'str'>

9.12.78 plugins.cli.cmd.order

Execution order for "plugins.cli.cmd" plugins

- Default: []
- Type: <class 'list'>

9.12.79 plugins.cli.order

Execution order for "plugins.cli" plugins

- Default: []
- Type: <class 'list'>

9.12.80 plugins.disable

Plugins that will not be loaded and executed

- Default: []
- Type: <class 'list'>

9.12.81 plugins.init.order

Execution order for "plugins.init" plugins

- Default: []
- Type: <class 'list'>

9.12.82 plugins.job.prepost.order

Execution order for "plugins.job.prepost" plugins

- Default: []
- Type: <class 'list'>

9.12.83 plugins.jobscripts.post

Directory with scripts to be executed after a job is run

- Default: /etc/avocado/scripts/job/post.d/
- Type: <function prepend_base_path at 0x7f20eca167a0>

9.12.84 plugins.jobscripts.pre

Directory with scripts to be executed before a job is run

- Default: /etc/avocado/scripts/job/pre.d/
- Type: <function prepend_base_path at 0x7f20eca167a0>

9.12.85 plugins.jobscripts.warn_non_existing_dir

Warn if configured (or default) directory does not exist

- Default: False
- Type: <class 'bool'>

9.12.86 plugins.jobscripts.warn_non_zero_status

Warn if any script run return non-zero status

- Default: True
- Type: <class 'bool'>

9.12.87 plugins.resolver.order

Execution order for "plugins.resolver" plugins

- Default: []
- Type: <class 'list'>

9.12.88 plugins.result.order

Execution order for "plugins.result" plugins

- Default: []
- Type: <class 'list'>

9.12.89 plugins.result_events.order

Execution order for "plugins.result_events" plugins

- Default: []
- Type: <class 'list'>

9.12.90 plugins.result_upload.cmd

Specify the command to upload results

- Default: None
- Type: <class 'str'>

9.12.91 plugins.result_upload.url

Specify the result upload url

- Default: None
- Type: <class 'str'>

9.12.92 plugins.resultsdb.api_url

Specify the resultsdb API url

- Default: None
- Type: <class 'str'>

9.12.93 plugins.resultsdb.logs_url

Specify the URL where the logs are published

- Default: None
- Type: <class 'str'>

9.12.94 plugins.resultsdb.note_size_limit

Maximum note size limit

- Default: 0
- Type: <class 'int'>

9.12.95 plugins.runnable.runner.order

Execution order for "plugins.runnable.runner" plugins

- Default: []
- Type: <class 'list'>

9.12.96 plugins.runner.order

Execution order for "plugins.runner" plugins

- Default: []
- Type: <class 'list'>

9.12.97 plugins.skip_broken_plugin_notification

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. (e.g. "avocado_result_html")

- Default: []
- Type: <class 'list'>

9.12.98 plugins.spawner.order

Execution order for "plugins.spawner" plugins

- Default: []
- Type: <class 'list'>

9.12.99 plugins.varianter.order

Execution order for "plugins.varianter" plugins

- Default: []
- Type: <class 'list'>

9.12.100 run.cit.combination_order

Order of combinations. Maximum number is 6

- Default: 2
- Type: <class 'int'>

9.12.101 run.cit.parameter_file

Paths to a parameter file

- Default: None
- Type: <class 'str'>

9.12.102 run.dict_variants

Load the Variants from Python dictionaries

- Default: []
- Type: <class 'list'>

9.12.103 run.dry_run.enabled

Instead of running the test only list them and log their params.

- Default: False
- Type: <class 'bool'>

9.12.104 run.dry_run.no_cleanup

Do not automatically clean up temporary directories used by dry-run

- Default: False
- Type: <class 'bool'>

9.12.105 run.execution_order

Defines the order of iterating through test suite and test variants

- Default: variants-per-test
- Type: <class 'str'>

9.12.106 run.external_runner

Path to an specific test runner that allows the use of its own tests. This should be used for running tests that do not conform to Avocado's SIMPLE test interface and can not run standalone. Note: the use of –external-runner overwrites the –loaders to 'external_runner'

- Default: None
- Type: <class 'str'>

9.12.107 run.external_runner_chdir

Change directory before executing tests. This option may be necessary because of requirements and/or limitations of the external test runner. If the external runner requires to be run from its own base directory, use 'runner' here. If the external runner runs tests based on files and requires to be run from the directory where those files are located, use 'test' here and specify the test directory with the option '–external-runner-testdir'.

- Default: None
- Type: <class 'str'>

9.12.108 run.external_runner_testdir

Where test files understood by the external test runner are located in the filesystem. Obviously this assumes and only applies to external test runners that run tests from files

- Default: None
- Type: <class 'str'>

9.12.109 run.failfast

Enable the job interruption on first failed test.

- Default: False
- Type: <class 'bool'>

9.12.110 run.ignore_missing_references

Force the job execution, even if some of the test references are not resolved to tests. "on" and "off" will be deprecated soon.

- Default: False
- Type: <class 'bool'>

9.12.111 run.job_category

Categorizes this within a directory with the same name, by creating a link to the job result directory

- Default: None
- Type: <class 'str'>

9.12.112 run.journal.enabled

Records test status changes (for use with avocado-journal-replay and avocado-server)

- Default: False
- Type: <class 'bool'>

9.12.113 run.keep_tmp

Keep job temporary files (useful for avocado debugging).

- Default: False
- Type: <class 'bool'>

9.12.114 run.loaders

Overrides the priority of the test loaders. You can specify either @loader_name or TEST_TYPE. By default it tries all available loaders according to priority set in settings->plugins.loaders.

- Default: ['file', '@DEFAULT']
- Type: <class 'list'>

9.12.115 run.log_test_data_directories

Logs the possible data directories for each test. This is helpful when writing new tests and not being sure where to put data files. Look for "Test data directories" in your test log

- Default: False
- Type: <class 'bool'>

9.12.116 run.output_check

Disables test output (stdout/stderr) check. If this option is given, no output will be checked, even if there are reference files present for the test.

- Default: True
- Type: <class 'bool'>

9.12.117 run.output_check_record

Record the output produced by each test (from stdout and stderr) into both the current executing result and into reference files. Reference files are used on subsequent runs to determine if the test produced the expected output or not, and the current executing result is used to check against a previously recorded reference file. Valid values: "none" (to explicitly disable all recording) "stdout" (to record standard output *only*), "stderr" (to record standard error *only*), "both" (to record standard output and error in separate files), "combined" (for standard output and error in a single file). "all" is also a valid but deprecated option that is a synonym of "both".

- Default: None
- Type: <class 'str'>

9.12.118 run.pict_binary

Where to find the binary version of the pict tool. Tip: download it from "https://github.com/Microsoft/pict" and run *make* to build it

- Default: None
- Type: <class 'str'>

9.12.119 run.pict_combinations_order

Order of combinations. Maximum number is specific to parameter file content

- Default: 2
- Type: <class 'int'>

9.12.120 run.pict_parameter_file

Paths to a pict parameter file

- Default: None
- Type: <class 'str'>

9.12.121 run.pict_parameter_path

Default path for parameters generated on the Pict based variants

- Default: /run
- Type: <class 'str'>

9.12.122 run.references

List of test references (aliases or paths)

- Default: []
- Type: <class 'list'>

9.12.123 run.replay.ignore

Ignore variants and/or configuration from the source job.

- Default: []
- Type: <function Replay._valid_ignore at 0x7f20e997c0e0>

9.12.124 run.replay.job_id

Replay a job identified by its (partial) hash id. Use "–replay" latest to replay the latest job.

- Default: None
- Type: <class 'str'>

9.12.125 run.replay.resume

Resume an interrupted job

- Default: False
- Type: <class 'bool'>

9.12.126 run.replay.test_status

Filter tests to replay by test status.

- Default: []
- Type: <function Replay._valid_status at 0x7f20e997c050>

9.12.127 run.results.archive

Archive (ZIP) files generated by tests

- Default: False
- Type: <class 'bool'>

9.12.128 run.results_dir

Forces to use of an alternate job results directory.

- Default: None
- Type: <class 'str'>

9.12.129 run.test_parameters

Parameter name and value to pass to all tests. This is only applicable when not using a varianter plugin. This option format must be given in the NAME=VALUE format, and may be given any number of times, or per parameter.

- Default: []
- Type: <function Run._test_parameter at 0x7f20eb1ed830>

9.12.130 run.test_runner

Selects the runner implementation from one of the installed and active implementations. You can run "avocado plugins" and find the list of valid runners under the "Plugins that run test suites on a job (runners) section. Defaults to "nrunner", which is the new runner. To use the conventional and traditional runner, use "runner".

- Default: nrunner
- Type: <class 'str'>

9.12.131 run.unique_job_id

Forces the use of a particular job ID. Used internally when interacting with an avocado server. You should not use this option unless you know exactly what you're doing

- Default: None
- Type: <class 'str'>

9.12.132 run.wrapper.wrappers

Use a script to wrap executables run by a test. The wrapper is either a path to a script (AKA a global wrapper) or a path to a script followed by colon symbol (:), plus a shell like glob to the target EXECUTABLE. Multiple wrapper options are allowed, but only one global wrapper can be defined.

- Default: []
- Type: <class 'list'>

9.12.133 runner.exectest.exitcodes.skip

Use a custom exit code list to consider a test as skipped. This is only used by exec-test runners. Default is [].

- Default: []
- Type: <class 'list'>

9.12.134 runner.output.color

Whether to force colored output to non-tty outputs (e.g. log files). Allowed values: auto, always, never

- Default: auto
- Type: <class 'str'>

9.12.135 runner.output.colored

Whether to display colored output in terminals that support it

- Default: True
- Type: <class 'bool'>

9.12.136 runner.output.utf8

Use UTF8 encoding (True or False)

- Default: True
- Type: <class 'bool'>

9.12.137 runner.timeout.after_interrupted

The amount of time to give to the test process after it it has been interrupted (such as with CTRL+C)

- Default: 60
- Type: <class 'int'>

9.12.138 runner.timeout.process_alive

The amount of time to wait after a test has reported status but the test process has not finished

- Default: 60
- Type: <class 'int'>

9.12.139 runner.timeout.process_died

The amount of to wait for a test status after the process has been noticed to be dead

- Default: 10
- Type: <class 'int'>

9.12.140 simpletests.status.failure_fields

Fields to include in the presentation of SIMPLE test failures. Accepted values: status, stdout, stderr.

- Default: ['status', 'stdout', 'stderr']
- Type: <class 'list'>

9.12.141 simpletests.status.skip_location

Location to search the regular expression on. Accepted values: all, stdout, stderr.

- Default: all
- Type: <class 'str'>

9.12.142 simpletests.status.skip_regex

Python regular expression that will make the test status SKIP when matched.

- Default: ^SKIP$
- Type: <class 'str'>

9.12.143 simpletests.status.warn_location

Location to search the regular expression on. Accepted values: all, stdout, stderr.

- Default: all
- Type: <class 'str'>

9.12.144 simpletests.status.warn_regex

Python regular expression that will make the test status WARN when matched.

- Default: ^WARN$
- Type: <class 'str'>

9.12.145 spawner.podman.bin

Path to the podman binary

- Default: /usr/bin/podman
- Type: <class 'str'>

9.12.146 spawner.podman.image

Image name to use when creating the container. The first default choice is a container image matching the current OS. If unable to detect, default becomes the latest Fedora release. Default on this system: Ubuntu:18

- Default: Ubuntu:18
- Type: <class 'str'>

9.12.147 sysinfo.collect.commands_timeout

Overall timeout to collect commands, when <=0no timeout is enforced

- Default: -1
- Type: <class 'int'>

9.12.148 sysinfo.collect.enabled

Enable or disable sysinfo information. Like hardware details, profiles, etc.

- Default: True
- Type: <class 'bool'>

9.12.149 sysinfo.collect.installed_packages

Whether to take a list of installed packages previous to avocado jobs

- Default: False
- Type: <class 'bool'>

9.12.150 sysinfo.collect.locale

Force LANG for sysinfo collection

- Default: C
- Type: <class 'str'>

9.12.151 sysinfo.collect.optimize

Optimize sysinfo collected so that duplicates between pre and post re not stored in post

- Default: False
- Type: <class 'bool'>

9.12.152 sysinfo.collect.per_test

Enable sysinfo collection per-test

- Default: False
- Type: <class 'bool'>

9.12.153 sysinfo.collect.profiler

Whether to run certain commands in bg to give extra job debug information

- Default: False
- Type: <class 'bool'>

9.12.154 sysinfo.collect.sysinfodir

Directory where Avocado will dump sysinfo data. If one is not given explicitly, it will default to a directory named "sysinfo-" followed by a timestamp in the current working directory.

- Default: None
- Type: <class 'str'>

9.12.155 sysinfo.collectibles.commands

File with list of commands that will be executed and have their output collected

- Default: /home/docs/checkouts/readthedocs.org/user_builds/avocado-framework/envs/92.0/lib/python3.7/site-packages/avocado_framework-92.0-py3.7.egg/avocado/etc/avocado/sysinfo/commands
- Type: <function prepend_base_path at 0x7f20eca167a0>

9.12.156 sysinfo.collectibles.fail_commands

File with list of commands that will be executed and have their output collected, in case of failed test

- Default: /home/docs/checkouts/readthedocs.org/user_builds/avocado-framework/envs/92.0/lib/python3.7/site-packages/avocado_framework-92.0-py3.7.egg/avocado/etc/avocado/sysinfo/fail_commands
- Type: <function prepend_base_path at 0x7f20eca167a0>

9.12.157 sysinfo.collectibles.fail_files

File with list of files that will be collected verbatim, in case of failed test

- Default: /home/docs/checkouts/readthedocs.org/user_builds/avocado-framework/envs/92.0/lib/python3.7/site-packages/avocado_framework-92.0-py3.7.egg/avocado/etc/avocado/sysinfo/fail_files
- Type: <function prepend_base_path at 0x7f20eca167a0>

9.12.158 sysinfo.collectibles.files

File with list of files that will be collected verbatim

- Default: /home/docs/checkouts/readthedocs.org/user_builds/avocado-framework/envs/92.0/lib/python3.7/site-packages/avocado_framework-92.0-py3.7.egg/avocado/etc/avocado/sysinfo/files
- Type: <function prepend_base_path at 0x7f20eca167a0>

9.12.159 sysinfo.collectibles.profilers

File with list of commands that will run alongside the job/test

- Default: /home/docs/checkouts/readthedocs.org/user_builds/avocado-framework/envs/92.0/lib/python3.7/site-packages/avocado_framework-92.0-py3.7.egg/avocado/etc/avocado/sysinfo/profilers
- Type: <function prepend_base_path at 0x7f20eca167a0>

9.12.160 task.timeout.running

The amount of time a test has to complete in seconds.

- Default: None
- Type: <class 'int'>

9.12.161 variants.cit.combination_order

Order of combinations. Maximum number is 6

- Default: 2
- Type: <class 'int'>

9.12.162 variants.cit.parameter_file

Paths to a parameter file

- Default: None
- Type: <class 'str'>

9.12.163 variants.contents

[obsoleted by –variants] Shows the node content (variables)

- Default: False
- Type: <class 'bool'>

9.12.164 variants.debug

Use debug implementation to gather more information.

- Default: False
- Type: <class 'bool'>

9.12.165 variants.inherit

[obsoleted by –summary] Show the inherited values

- Default: False
- Type: <class 'bool'>

9.12.166 variants.json_variants_dump

Dump the Variants to a JSON serialized file

- Default: None
- Type: <class 'str'>

9.12.167 variants.pict_binary

Where to find the binary version of the pict tool. Tip: download it from "https://github.com/Microsoft/pict" and run *make* to build it

- Default: None
- Type: <class 'str'>

9.12.168 variants.pict_combinations_order

Order of combinations. Maximum number is specific to parameter file content

- Default: 2
- Type: <class 'int'>

9.12.169 variants.pict_parameter_file

Paths to a pict parameter file

- Default: None
- Type: <class 'str'>

9.12.170 variants.pict_parameter_path

Default path for parameters generated on the Pict based variants

- Default: /run
- Type: <class 'str'>

9.12.171 variants.summary

Verbosity of the variants summary. (positive integer - 0, 1, ... - or none, brief, normal, verbose, full, max)

- Default: 0
- Type: <function map_verbosity_level at 0x7f20e9418e60>

9.12.172 variants.tree

[obsoleted by –summary] Shows the multiplex tree structure

- Default: False
- Type: <class 'bool'>

9.12.173 variants.variants

Verbosity of the list of variants. (positive integer - 0, 1, ... - or none, brief, normal, verbose, full, max)

- Default: 1
- Type: <function map_verbosity_level at 0x7f20e9418e60>

9.12.174 vmimage.get.arch

Image architecture

- Default: None
- Type: <class 'str'>

9.12.175 vmimage.get.distro

Name of image distribution

- Default: None
- Type: <class 'str'>

9.12.176 vmimage.get.version

Image version

- Default: None
- Type: <class 'str'>

9.12.177 yaml_to_mux.files

Location of one or more Avocado multiplex (.yaml) FILE(s) (order dependent)

- Default: []
- Type: <class 'list'>

9.12.178 yaml_to_mux.filter_only

Filter only path(s) from multiplexing

- Default: []
- Type: <class 'list'>

9.12.179 yaml_to_mux.filter_out

Filter out path(s) from multiplexing

- Default: []
- Type: <class 'str'>

9.12.180 yaml_to_mux.inject

Inject [path:]key:node values into the final multiplex tree.

- Default: []
- Type: <class 'list'>

9.12.181 yaml_to_mux.parameter_paths

List of default paths used to determine path priority when querying for parameters

- Default: ['/run/*']
- Type: <class 'list'>

CHAPTER 10

Test API

10.1 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.

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.

This is the bare minimum set of APIs that users should use, and can rely on, while writing tests.

10.1.1 Module contents

class avocado.**Test**(*methodName='test'*, *name=None*, *params=None*, *base_logdir=None*, *config=None*, *runner_queue=None*, *tags=None*)

Bases: `unittest.case.TestCase`, *avocado.core.test.TestData*

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.TestID`) – 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 a temporary directory will be created.
- **config** (*dict*) – the job configuration, usually set by command line options and argument parsing

actual_time_end = -1
(unix) time when the test finished, actual one to be shown to users

actual_time_start = -1
(unix) time when the test started, actual one to be shown to users

basedir
The directory where this test (when backed by a file) is located at

cache_dirs
Returns a list of cache directories as set in config file.

static 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

Warning message This parameter will changed name to "msg" in the next LTS release because of lint W0221

static 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

Warning message This parameter will changed name to "msg" in the next LTS release because of lint W0221

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

Warning message This parameter will changed name to "msg" in the next LTS release because of lint W0221

fail_class

fail_reason

fetch_asset(*name*, *asset_hash=None*, *algorithm=None*, *locations=None*, *expire=None*, *find_only=False*, *cancel_on_missing=False*)
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 *avocado.utils.asset.DEFAULT_HASH_ALGORITHM*)
- **locations** – list of URLs from where the asset can be fetched (optional)
- **expire** – time for the asset to expire
- **find_only** – When *True*, *fetch_asset* only looks for the asset in the cache, avoiding the download/move action. Defaults to *False*.
- **cancel_on_missing** – whether the test should be canceled if the asset was not found in the cache or if *fetch* could not add the asset to the cache. Defaults to *False*.

Raises **OSError** – when it fails to fetch the asset or file is not in the cache and *cancel_on_missing* is *False*.

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

log
The enhanced test log

logdir
Path to this test's logging dir

logfile
Path to this test's main *debug.log* file

name
Returns the Test ID, which includes the test name

Return type *TestID*

outputdir
Directory available to test writers to attach files to the results

params
Parameters of this test (AvocadoParam instance)

phase
The current phase of the test execution

Possible (string) values are: INIT, SETUP, TEST, TEARDOWN and FINISHED

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

status
The result status of this test

tags
The tags associated with this test

tearDown()
Hook method for deconstructing the test fixture after testing it.

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, monotonic (could be forced from test)

time_start = -1
(unix) time when the test started, monotonic (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
This property returns a writable directory that exists during the entire test execution, but will be cleaned up once the test finishes.

It can be used on tasks such as decompressing source tarballs, building software, etc.

avocado.**fail_on**(*exceptions=None*)
Fail the test when decorated function produces exception of the specified type.

Parameters **exceptions** – Tuple or single exception to be assumed as test FAIL [Exception].

Note self.error, self.cancel and self.fail remain intact.

Note to allow simple usage param 'exceptions' must not be callable.

avocado.**cancel_on**(*exceptions=None*)
Cancel the test when decorated function produces exception of the specified type.

Parameters **exceptions** – Tuple or single exception to be assumed as test CANCEL [Exception].

Note self.error, self.cancel and self.fail remain intact.

Note to allow simple usage param 'exceptions' must not be callable.

avocado.**skip**(*message=None*)
Decorator to skip a test.

Parameters **message** (*str*) – the message given when the test is skipped

avocado.**skipIf**(*condition*, *message=None*)
Decorator to skip a test if a condition is True.

Parameters

- **condition** (*bool or callable*) – a condition that will be evaluated as either True or False, if it's a callable, it will be called with the class instance as a parameter
- **message** (*str*) – the message given when the test is skipped

avocado.**skipUnless**(*condition*, *message=None*)
Decorator to skip a test if a condition is False.

Parameters

- **condition** (*bool or callable*) – a condition that will be evaluated as either True or False, if it's a callable, it will be called with the class instance as a parameter
- **message** (*str*) – the message given when the test is skipped

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*, 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'

10.2 Internal (Core) APIs

Internal APIs that may be of interest to Avocado hackers.

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.

10.2.1 Subpackages

avocado.core.requirements package

Subpackages

avocado.core.requirements.cache package

Subpackages

avocado.core.requirements.cache.backends package

Submodules

avocado.core.requirements.cache.backends.sqlite module

Test requirements module.

avocado.core.requirements.cache.backends.sqlite.**CACHE_DATABASE_PATH = '/home/docs/avocado/**
The location of the requirements cache database

avocado.core.requirements.cache.backends.sqlite.**SCHEMA = ['CREATE TABLE IF NOT EXISTS requ**
The definition of the database schema

avocado.core.requirements.cache.backends.sqlite.**get_requirement**(*environment_type*, *environment*, *requirement_type*, *requirement*)

avocado.core.requirements.cache.backends.sqlite.**set_requirement**(*environment_type*, *environment*, *requirement_type*, *requirement*)

Module contents

Module contents

Submodules

avocado.core.requirements.resolver module

class avocado.core.requirements.resolver.**RequirementsResolver**
Bases: object

description = 'Requirements resolver for tests with requirements'

name = 'requirements'

static resolve(*runnable*)

Module contents

avocado.core.runners package

Subpackages

avocado.core.runners.utils package

Submodules

avocado.core.runners.utils.messages module

class avocado.core.runners.utils.messages.**FileMessage**
Bases: *avocado.core.runners.utils.messages.GenericRunningMessage*

Creates file message with all necessary information.

classmethod get(*msg*, *path*)
Creates running message with all necessary information.

Parameters **msg** (*str*) – log of running message

Returns running message

Return type dict

message_type = 'file'

class avocado.core.runners.utils.messages.**FinishedMessage**
Bases: *avocado.core.runners.utils.messages.GenericMessage*

classmethod get(*result*, *fail_reason=None*, *returncode=None*)
Creates finished message with all necessary information.

Parameters **result** – test result

:type result values for the statuses defined in

class avocado.core.teststatus.STATUSES

Parameters

- **fail_reason** (*str*) – parameter for brief specification, of the failed result.
- **returncode** – exit status of runner

Returns finished message

Return type dict

message_status = 'finished'

class avocado.core.runners.utils.messages.**GenericMessage**
Bases: object

classmethod get(***kwargs*)
Creates message base on it's type with all necessary information.

Returns message dict which can be send to avocado server

Return type dict

message_status = None

class avocado.core.runners.utils.messages.**GenericRunningMessage**
Bases: *avocado.core.runners.utils.messages.GenericMessage*

classmethod get(*msg*, ***kwargs*)
Creates running message with all necessary information.

Parameters **msg** (*str*) – log of running message

Returns running message

Return type dict

message_status = 'running'

message_type = None

class avocado.core.runners.utils.messages.**LogMessage**
Bases: *avocado.core.runners.utils.messages.GenericRunningMessage*

message_type = 'log'

class avocado.core.runners.utils.messages.**RunnerLogHandler**(*queue*, *message_type*)
Bases: logging.Handler

Runner logger which will put every log to the runner queue

Parameters

- **queue** (*multiprocessing.SimpleQueue*) – queue for the runner messages
- **message_type** (*string*) – type of the log

emit(*record*)
Do whatever it takes to actually log the specified logging record.

This version is intended to be implemented by subclasses and so raises a NotImplementedError.

class avocado.core.runners.utils.messages.**RunningMessage**
Bases: *avocado.core.runners.utils.messages.GenericMessage*

Creates running message without any additional info.

message_status = 'running'

class avocado.core.runners.utils.messages.**StartedMessage**
Bases: *avocado.core.runners.utils.messages.GenericMessage*

message_status = 'started'

class avocado.core.runners.utils.messages.**StderrMessage**
Bases: *avocado.core.runners.utils.messages.GenericRunningMessage*

Creates stderr message with all necessary information.

message_type = 'stderr'

class avocado.core.runners.utils.messages.**StdoutMessage**
Bases: *avocado.core.runners.utils.messages.GenericRunningMessage*

Creates stdout message with all necessary information.

message_type = 'stdout'

class avocado.core.runners.utils.messages.**StreamToQueue**(*queue*, *message_type*)
Bases: object

Runner Stream which will transfer data to the runner queue

Parameters

- **queue** (*multiprocessing.SimpleQueue*) – queue for the runner messages
- **message_type** (*string*) – type of the log

flush()

write(*buf*)

class avocado.core.runners.utils.messages.**WhiteboardMessage**
Bases: *avocado.core.runners.utils.messages.GenericRunningMessage*

Creates whiteboard message with all necessary information.

message_type = 'whiteboard'

avocado.core.runners.utils.messages.**start_logging**(*config*, *queue*)
Helper method for connecting the avocado logging with avocado messages.

It will add the logHandlers to the :class: avocado.core.output loggers, which will convert the logs to the avocado messages and sent them to processing queue.

Parameters

- **config** (*dict*) – avocado configuration
- **queue** (*multiprocessing.SimpleQueue*) – queue for the runner messages

Module contents

Submodules

avocado.core.runners.avocado_instrumented module

class avocado.core.runners.avocado_instrumented.**AvocadoInstrumentedTestRunner**(*runnable*)
Bases: *avocado.core.nrunner.BaseRunner*

Runner for Avocado INSTRUMENTED tests

Runnable attributes usage:

- uri: path to a test file, combined with an Avocado.Test inherited class name and method. The test file path and class and method names should be separated by a ":". One example of a valid uri is "mytest.py:Class.test_method".
- args: not used

DEFAULT_TIMEOUT = 86400

run()
Runner main method

Yields dictionary as output, containing status as well as relevant information concerning the results.

class avocado.core.runners.avocado_instrumented.**RunnerApp**(*echo=<built-in function print>*, *prog=None*, *description=None*)
Bases: *avocado.core.nrunner.BaseRunnerApp*

PROG_DESCRIPTION = 'nrunner application for avocado-instrumented tests'

PROG_NAME = 'avocado-runner-avocado-instrumented'

RUNNABLE_KINDS_CAPABLE = {'avocado-instrumented': <class 'avocado.core.runners.avocad

avocado.core.runners.avocado_instrumented.**main**()

avocado.core.runners.requirement_asset module

class avocado.core.runners.requirement_asset.**RequirementAssetRunner**(*runnable*)
Bases: *avocado.core.nrunner.BaseRunner*

Runner for requirements of type package

This runner handles the fetch of files using the Avocado Assets utility.

Runnable attributes usage:

- kind: 'requirement-asset'
- uri: not used
- args: not used
- **kwargs:**
  - name: the file name or uri (required)
  - asset_hash: hash of the file (optional)
  - algorithm: hash algorithm (optional)
  - locations: location(s) where the file can be fetched from (optional)
  - expire: time in seconds for the asset to expire (optional)

run()
Runner main method

Yields dictionary as output, containing status as well as relevant information concerning the results.

class avocado.core.runners.requirement_asset.**RunnerApp**(*echo=<built-in function print>*, *prog=None*, *description=None*)
Bases: *avocado.core.nrunner.BaseRunnerApp*

PROG_DESCRIPTION = 'nrunner application for requirements of type asset'

PROG_NAME = 'avocado-runner-requirement-asset'

RUNNABLE_KINDS_CAPABLE = {'requirement-asset': <class 'avocado.core.runners.requireme

avocado.core.runners.requirement_asset.**main**()

avocado.core.runners.requirement_package module

class avocado.core.runners.requirement_package.**RequirementPackageRunner**(*runnable*)
Bases: *avocado.core.nrunner.BaseRunner*

Runner for requirements of type package

This runner handles, the installation, verification and removal of packages using the avocado-software-manager.

Runnable attributes usage:

- kind: 'requirement-package'
- uri: not used
- args: not used
- **kwargs:**

- name: the package name (required)
- action: one of 'install', 'check', or 'remove' (optional, defaults to 'install')

run()
Runner main method

Yields dictionary as output, containing status as well as relevant information concerning the results.

class avocado.core.runners.requirement_package.**RunnerApp**(*echo=<built-in function print>*, *prog=None*, *description=None*)
Bases: *avocado.core.nrunner.BaseRunnerApp*

PROG_DESCRIPTION = 'nrunner application for requirements of type package'

PROG_NAME = 'avocado-runner-requirement-package'

RUNNABLE_KINDS_CAPABLE = {'requirement-package': <class 'avocado.core.runners.require

avocado.core.runners.requirement_package.**main**()

avocado.core.runners.sysinfo module

class avocado.core.runners.sysinfo.**PostSysInfo**(*config*, *sysinfo_config*, *queue*, *test_fail=False*)
Bases: *avocado.core.runners.sysinfo.PreSysInfo*

Log different system properties after end event.

An event may be a job, a test, or any other event with a beginning and end.

Parameters **test_fail** (*bool*) – flag for fail tests. Default False

sysinfo_dir = 'sysinfo/post'

class avocado.core.runners.sysinfo.**PreSysInfo**(*config*, *sysinfo_config*, *queue*)
Bases: object

Log different system properties before start event.

An event may be a job, a test, or any other event with a beginning and end.

Set sysinfo collectibles.

Parameters

- **config** (*dict*) – avocado configuration
- **sysinfo_config** (*dict*) – dictionary with commands/tasks which should be performed during the sysinfo collection.
- **queue** (*multiprocessing.SimpleQueue*) – queue for the runner messages

collect()
Log all collectibles at the start of the event.

installed_pkgs

sysinfo_dir = 'sysinfo/pre'

class avocado.core.runners.sysinfo.**RunnerApp**(*echo=<built-in function print>*, *prog=None*, *description=None*)
Bases: *avocado.core.nrunner.BaseRunnerApp*

PROG_DESCRIPTION = 'nrunner application for gathering sysinfo'

PROG_NAME = 'avocado-runner-sysinfo'

RUNNABLE_KINDS_CAPABLE = {'sysinfo': <class 'avocado.core.runners.sysinfo.SysinfoRunn

class avocado.core.runners.sysinfo.**SysinfoRunner**(*runnable*)
Bases: *avocado.core.nrunner.BaseRunner*

Runner for gathering sysinfo

Runnable attributes usage:

- **uri: sysinfo type pre/post. This variable decides if the sysinfo is** collected before or after the test.
- **kwargs: "sysinfo" dictionary with commands/tasks which should be** performed during the sysinfo collection.

run()
Runner main method

Yields dictionary as output, containing status as well as relevant information concerning the results.

avocado.core.runners.sysinfo.**main**()

avocado.core.runners.tap module

class avocado.core.runners.tap.**RunnerApp**(*echo=<built-in function print>*, *prog=None*, *description=None*)
Bases: *avocado.core.nrunner.BaseRunnerApp*

PROG_DESCRIPTION = 'nrunner application for executable tests that produce TAP'

PROG_NAME = 'avocado-runner-tap'

RUNNABLE_KINDS_CAPABLE = {'tap': <class 'avocado.core.runners.tap.TAPRunner'>}

class avocado.core.runners.tap.**TAPRunner**(*runnable*)
Bases: *avocado.core.nrunner.ExecTestRunner*

Runner for standalone executables treated as TAP

When creating the Runnable, use the following attributes:

- kind: should be 'tap';
- uri: path to a binary to be executed as another process. This must provides a TAP output.
- args: any runnable argument will be given on the command line to the binary given by path
- kwargs: you can specify multiple key=val as kwargs. This will be used as environment variables to the process.

Example:

runnable = Runnable(kind='tap', uri='tests/foo.sh', 'bar', # arg 1 DEBUG='false') # kwargs 1 (environment)

avocado.core.runners.tap.**main**()

Module contents

avocado.core.safeloader package

Submodules

avocado.core.safeloader.core module

exception avocado.core.safeloader.core.**ClassNotSuitable**
Bases: Exception

Exception raised when examination of a class should not proceed.

avocado.core.safeloader.core.**find_avocado_tests**(*path*)

avocado.core.safeloader.core.**find_python_tests**(*target_module*, *target_class*, *determine_match*, *path*)
Attempts to find Python tests from source files

A Python test in this context is a method within a specific type of class (or that inherits from a specific class).

Parameters

- **target_module** (*str*) – the name of the module from which a class should have come from. When attempting to find a Python unittest, the target_module will most probably be "unittest", as per the standard library module name. When attempting to find Avocado tests, the target_module will most probably be "avocado".
- **target_class** (*str*) – the name of the class that is considered to contain test methods. When attempting to find Python unittests, the target_class will most probably be "TestCase". When attempting to find Avocado tests, the target_class will most probably be "Test".
- **path** (*str*) – path to a Python source code file

Returns tuple where first item is dict with class name and additional info such as method names and tags; the second item is set of class names which look like Python tests but have been forcefully disabled.

Return type tuple

avocado.core.safeloader.core.**find_python_unittests**(*path*)

avocado.core.safeloader.core.**get_methods_info**(*statement_body*, *class_tags*, *class_requirements*)
Returns information on test methods.

Parameters

- **statement_body** – the body of a "class" statement
- **class_tags** – the tags at the class level, to be combined with the tags at the method level.
- **class_requirements** – the requirements at the class level, to be combined with the requirements at the method level.

avocado.core.safeloader.docstring module

avocado.core.safeloader.docstring.**DOCSTRING_DIRECTIVE_RE_RAW = '\\s*:avocado:[\\t]+(([a-zA**
Gets the docstring directive value from a string. Used to tweak test behavior in various ways

avocado.core.safeloader.docstring.**check_docstring_directive**(*docstring*, *directive*)
Checks if there's a given directive in a given docstring

Return type bool

avocado.core.safeloader.docstring.**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.docstring.**get_docstring_directives_requirements**(*docstring*)
Returns the test requirements from docstring patterns like *:avocado: requirement={}*.

Return type list

avocado.core.safeloader.docstring.**get_docstring_directives_tags**(*docstring*)
Returns the test categories based on a *:avocado: tags=category* docstring

Return type dict

avocado.core.safeloader.imported module

class avocado.core.safeloader.imported.**ImportedSymbol**(*module_path*, *symbol=''*, *importer_fs_path=None*, *module_alias=''*, *symbol_alias=''*)
Bases: object

A representation of an importable symbol.

Attributes:

module_path : str symbol : str importer_fs_path: str or None

classmethod from_statement(*statement*, *importer_fs_path=None*, *index=0*)

get_importable_spec(*symbol_is_module=False*)
Returns the specification of an actual importable module.

This is a check based on the limitations that we do not actually perform an import, and assumes a directory structure with modules.

Parameters **symbol_is_module** (*bool*) – if it's known that the symbol is also a module, include it in the search for an importable spec

static get_module_path_from_statement(*statement*)

get_parent_fs_path()

get_relative_module_fs_path()
Returns the module base dir, based on its relative path

The base dir for the module is the directory where one is expected to find the first module of the module path. For a module path of "..foo.bar", and its importer being at "/abs/path/test.py", the base dir where "foo" is supposed to be found would be "/abs". And as a consequence, "bar" would be found at "/abs/foo/bar".

This assumes that the module path is indeed related to the location of its importer. This may not be true if the namespaces match, but are distributed across different filesystem paths.

static get_symbol_from_statement(*statement*)

static get_symbol_module_path_from_statement(*statement*, *name_index=0*)

importer_fs_path = None
The full, absolute filesystem path of the module importing this symbol. This is used for relative path

calculations, but it's limited to relative modules that also share the filesystem location. An example is "/path/to/mytest.py", that can contain:

from .base import BaseTestClass

And thus will have a symbol of "BaseTestClass" and the module as ".base". The relative filesystem path of the module (which should contain the symbol) will be "/path/to".

And if "/path/to/common/test.py" contains:

from ..base import BaseTestClass

The relative filesystem path of the module (which should contain the symbol) will be "/path/to".

is_importable(*symbol_is_module=False*)
Checks whether this imported symbol seems to be importable.

This is a check based on the limitations that we do not actually perform an import, and assumes a directory structure with modules.

Parameters **symbol_is_module** (*bool*) – if it's known that the symbol is also a module, include it in the search for an importable spec

is_relative()
Returns whether the imported symbol is on a relative path.

module_alias = None
An optional alias for the module, such as when a "import os as operating_system" statement is given.

module_name
The final name of the module from its importer perspective.

If a alias exists, it will be the alias name. If not, it will be the original name.

module_path = None
Path from where the symbol was imported. On a statement such as "import os", module_path is "os" and there's no symbol. On a statement such as from unittest.mock import mock_open", the module_path is "unittest.mock". On a statement such as "from ..foo import bar", module_path is "..foo" (relative).

symbol = None
The name of the imported symbol. On a statement such as "import os", there's no symbol. On a statement such as "from unittest import mock"", the symbol is "mock" (even though it may actually also be a module, but it's impossible to know for sure). On a statement such as "from unittest.mock import mock_open", symbol is "mock_open".

symbol_alias = None
An optional alias the symbol, such as when a "from os import path as os_path" is given

symbol_name
The final name of the symbol from its importer perspective.

If a alias exists, it will be the alias name. If not, it will be the original name.

to_str()
Returns a string representation of the plausible statement used.

avocado.core.safeloader.module module

class avocado.core.safeloader.module.**PythonModule**(*path*, *module='avocado'*, *klass='Test'*)

Bases: object

Representation of a Python module that might contain interesting classes

By default, it uses module and class names that matches Avocado instrumented tests, but it's supposed to be agnostic enough to be used for, say, Python unittests.

Instantiates a new PythonModule representation

Parameters

- **path** (*str*) – path to a Python source code file
- **module** (*str*) – the original module name from where the possibly interesting class must have been imported from
- **klass** (*str*) – the possibly interesting class original name

add_imported_symbol(*statement*)
Keeps track of symbol names and importable entities

imported_symbols

interesting_klass_found

is_matching_klass(*klass*)
Detect whether given class directly defines itself as <module>.<klass>

It can either be a <klass> that inherits from a test "symbol", like:

```
`class FooTest(Test)`
```

Or from an <module>.<klass> symbol, like in:

```
`class FooTest(avocado.Test)`
```

Return type bool

iter_classes(*interesting_klass=None*)
Iterate through classes and keep track of imported avocado statements

klass

klass_imports

mod

mod_imports

module

path

avocado.core.safeloader.utils module

avocado.core.safeloader.utils.**get_statement_import_as**(*statement*)
Returns a mapping of imported module names whether using aliases or not

Parameters **statement** (*ast.Import*) – an AST import statement

Returns a mapping of names {<realname>: <alias>} of modules imported

Return type collections.OrderedDict

Module contents

Safe (AST based) test loader module utilities

avocado.core.safeloader.**find_avocado_tests**(*path*)

avocado.core.safeloader.**find_python_unittests**(*path*)

avocado.core.spawners package

Submodules

avocado.core.spawners.common module

class avocado.core.spawners.common.**SpawnMethod**
Bases: enum.Enum

The method employed to spawn a runnable or task.

ANY = <object object>
Spawns with any method available, that is, it doesn't declare or require a specific spawn method

PYTHON_CLASS = <object object>
Spawns by running executing Python code, that is, having access to a runnable or task instance, it calls its run() method.

STANDALONE_EXECUTABLE = <object object>
Spawns by running a command, that is having either a path to an executable or a list of arguments, it calls a function that will execute that command (such as with os.system())

class avocado.core.spawners.common.**SpawnerMixin**(*config=None*)
Bases: object

Common utilities for Spawner implementations.

METHODS = []

static bytes_from_file(*filename*)
Read bytes from a files in binary mode.

This is a helpful method to read *local* files bytes efficiently.

If the spawner that you are implementing needs access to local file, feel free to use this method.

static stream_output(*job_id*, *task_id*)
Returns output files streams in binary mode from a task.

This method will find for output files generated by a task and will return a generator with tuples, each one containing a filename and bytes.

You need to provide in your spawner a *stream_output()* method if this one is not suitable for your spawner. i.e: if the spawner is trying to access a remote output file.

avocado.core.spawners.exceptions module

exception avocado.core.spawners.exceptions.**SpawnerException**
Bases: Exception

avocado.core.spawners.mock module

class avocado.core.spawners.mock.**MockRandomAliveSpawner**
Bases: *avocado.core.spawners.mock.MockSpawner*

A mocking spawner that simulates randomness about tasks being alive.

is_task_alive(*runtime_task*)
Determines if a task is alive or not.

Parameters **runtime_task** (*avocado.core.task.runtime.RuntimeTask*) – wrapper for a Task with additional runtime information

class avocado.core.spawners.mock.**MockSpawner**
Bases: *avocado.core.plugin_interfaces.Spawner*

A mocking spawner that performs no real operation.

Tasks asked to be spawned by this spawner will initially reported to be alive, and on the next check, will report not being alive.

METHODS = [<SpawnMethod.PYTHON_CLASS: <object object>>, <SpawnMethod.STANDALONE_EXECUT

static check_task_requirements(*runtime_task*)
Checks if the requirements described within a task are available.

Parameters **runtime_task** (*avocado.core.task.runtime.RuntimeTask*) – wrapper for a Task with additional runtime information

is_task_alive(*runtime_task*)
Determines if a task is alive or not.

Parameters **runtime_task** (*avocado.core.task.runtime.RuntimeTask*) – wrapper for a Task with additional runtime information

spawn_task(*runtime_task*)
Spawns a task return whether the spawning was successful.

Parameters **runtime_task** (*avocado.core.task.runtime.RuntimeTask*) – wrapper for a Task with additional runtime information

wait_task(*runtime_task*)
Waits for a task to finish.

Parameters **runtime_task** (*avocado.core.task.runtime.RuntimeTask*) – wrapper for a Task with additional runtime information

Module contents

avocado.core.status package

Submodules

avocado.core.status.repo module

exception avocado.core.status.repo.**StatusMsgMissingDataError**
Bases: Exception

Status message does not contain the required data.

class avocado.core.status.repo.**StatusRepo**(*job_id*)
Bases: object

Maintains tasks' status related data and provides aggregated info.

Initializes a new StatusRepo

Parameters **job_id** (*str*) – the job unique identification for which the messages are destined to.

get_all_task_data(*task_id*)
Returns all data on a given task, by its ID.

get_latest_task_data(*task_id*)
Returns the latest data on a given task, by its ID.

get_result_set_for_tasks(*task_ids*)
Returns a set of results for the given tasks.

get_task_data(*task_id*, *index*)
Returns the data on the index of a given task, by its ID.

get_task_status(*task_id*)

process_message(*message*)

process_raw_message(*raw_message*)

result_stats

status_journal_summary

avocado.core.status.server module

class avocado.core.status.server.**StatusServer**(*uri*, *repo*)
Bases: object

Server that listens for status messages and updates a StatusRepo.

Initializes a new StatusServer.

Parameters

- **uri** (*str*) – either a "host:port" string or a path to a UNIX socket
- **repo** (*avocado.core.status.repo.StatusRepo*) – the repository to use to process received status messages

cb(*reader*, *_*)

close()

create_server()

serve_forever()

uri

avocado.core.status.utils module

exception avocado.core.status.utils.**StatusMsgInvalidJSONError**
Bases: Exception

Status message does not contain valid JSON.

avocado.core.status.utils.**json_base64_decode**(*dct*)
base64 decode object hook for custom JSON encoding.

avocado.core.status.utils.**json_loads**(*data*)
Loads and decodes JSON, with added base64 decoding.

Parameters **data** – either bytes or a string. If bytes, will be decoded using the current default encoding.

Raises

Returns decoded Python objects

Module contents

avocado.core.task package

Submodules

avocado.core.task.runtime module

class avocado.core.task.runtime.**RuntimeTask**(*task*)
Bases: object

Task with extra status information on its life cycle status.

The *avocado.core.nrunner.Task* class contains information that is necessary to describe its persistence and execution by itself.

This class wraps a *avocado.core.nrunner.Task*, with extra information about its execution by a spawner within a state machine.

Instantiates a new RuntimeTask.

Parameters **task** (*avocado.core.nrunner.Task*) – The task to keep additional information about

execution_timeout = None
Timeout limit for the completion of the task execution

spawner_handle = None
A handle that may be set by a spawner, and that may be spawner implementation specific, to keep track the task execution. This may be a PID, a container ID, a FQDN+PID etc.

spawning_result = None
The result of the spawning of a Task

status = None
Additional descriptive information about the task status

task = None
The *avocado.core.nrunner.Task*

avocado.core.task.statemachine module

class avocado.core.task.statemachine.**TaskStateMachine**(*tasks*, *status_repo*)
Bases: object

Represents all phases that a task can go through its life.

abort(*status_reason=None*)
Abort all non-started tasks.

This method will move all non-started tasks to finished with a specific reason.

Parameters **status_reason** – string reason. Optional.

abort_queue(*queue_name*, *status_reason=None*)
Abort all tasks inside a specific queue adding a status reason.

Parameters

- **queue_name** – a string with the queue name.
- **status_reason** – string reason. Optional.

complete

finish_task(*runtime_task*, *status_reason=None*)
Include a task to the finished queue with a specific reason.

This method is assuming that you have removed (pop) the task from the original queue.

Parameters

- **runtime_task** – A running task object.
- **status_reason** – string reason. Optional.

finished

lock

ready

requested

started

triaging

class avocado.core.task.statemachine.**Worker**(*state_machine*, *spawner*, *max_triaging=None*, *max_running=None*, *task_timeout=None*)

Bases: object

bootstrap()
Reads from requested, moves into triaging.

monitor()
Reads from started, moves into finished.

run()
Pushes Tasks forward and makes them do something with their lives.

start()
Reads from ready, moves into either: started or finished.

triage()
Reads from triaging, moves into either: ready or finished.

Module contents

10.2.2 Submodules

10.2.3 avocado.core.app module

The core Avocado application.

class avocado.core.app.**AvocadoApp**
Bases: object

Avocado application.

run()

10.2.4 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**(*base_dir=None*, *unique_id=None*)
Create a log directory for a job, or a stand alone execution of a test.

Parameters

- **base_dir** – Base log directory, if *None*, use value from configuration.
- **unique_id** – The unique identification. If *None*, create one.

Return type str

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_cache_dirs**()
Returns the list of cache dirs, according to configuration and convention.

This will be deprecated. Please use settings.as_dict() or self.config.

> **Warning:** This method is deprecated, get values from settings.as_dict() or self.config

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

> **Warning:** This method is deprecated, get values from settings.as_dict() or self.config

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_job_results_dir**(*job_ref*, *logs_dir=None*)
Get the job results directory from a job reference.

Parameters

- **job_ref** – job reference, which can be: * an valid path to the job results directory. In this case it is checked if 'id' file exists * the path to 'id' file * the job id, which can be 'latest' * an partial job id
- **logs_dir** – path to base logs directory (optional), otherwise it uses the value from settings.

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.

> **Warning:** This method is deprecated, get values from settings.as_dict() or self.config

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 from its source code 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

10.2.5 avocado.core.decorators module

avocado.core.decorators.**cancel_on**(*exceptions=None*)
Cancel the test when decorated function produces exception of the specified type.

Parameters **exceptions** – Tuple or single exception to be assumed as test CANCEL [Exception].

Note self.error, self.cancel and self.fail remain intact.

Note to allow simple usage param 'exceptions' must not be callable.

avocado.core.decorators.**deco_factory**(*behavior*, *signal*)
Decorator factory.

Returns a decorator used to signal the test when specified exception is raised. :param behavior: expected test result behavior. :param signal: delegating exception.

avocado.core.decorators.**fail_on**(*exceptions=None*)
Fail the test when decorated function produces exception of the specified type.

Parameters **exceptions** – Tuple or single exception to be assumed as test FAIL [Exception].

Note self.error, self.cancel and self.fail remain intact.

Note to allow simple usage param 'exceptions' must not be callable.

avocado.core.decorators.**skip**(*message=None*)
Decorator to skip a test.

Parameters **message** (*str*) – the message given when the test is skipped

avocado.core.decorators.**skipIf**(*condition*, *message=None*)
Decorator to skip a test if a condition is True.

Parameters

- **condition** (*bool or callable*) – a condition that will be evaluated as either True or False, if it's a callable, it will be called with the class instance as a parameter
- **message** (*str*) – the message given when the test is skipped

avocado.core.decorators.**skipUnless**(*condition*, *message=None*)
Decorator to skip a test if a condition is False.

Parameters

- **condition** (*bool or callable*) – a condition that will be evaluated as either True or False, if it's a callable, it will be called with the class instance as a parameter
- **message** (*str*) – the message given when the test is skipped

10.2.6 avocado.core.dispatcher module

Extensions/plugins dispatchers

Besides the dispatchers listed here, there's also a lower level dispatcher that these depend upon: *avocado.core.settings_dispatcher.SettingsDispatcher*

class avocado.core.dispatcher.**CLICmdDispatcher**
Bases: *avocado.core.enabled_extension_manager.EnabledExtensionManager*

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.enabled_extension_manager.EnabledExtensionManager*

Calls extensions on configure/run

Automatically adds all the extension with entry points registered under 'avocado.plugins.cli'

class avocado.core.dispatcher.**InitDispatcher**
Bases: *avocado.core.enabled_extension_manager.EnabledExtensionManager*

class avocado.core.dispatcher.**JobPrePostDispatcher**
Bases: *avocado.core.enabled_extension_manager.EnabledExtensionManager*

Calls extensions before Job execution

Automatically adds all the extension with entry points registered under 'avocado.plugins.job.prepost'

class avocado.core.dispatcher.**ResultDispatcher**
Bases: *avocado.core.enabled_extension_manager.EnabledExtensionManager*

class avocado.core.dispatcher.**ResultEventsDispatcher**(*config*)
Bases: *avocado.core.enabled_extension_manager.EnabledExtensionManager*

class avocado.core.dispatcher.**RunnerDispatcher**
Bases: *avocado.core.enabled_extension_manager.EnabledExtensionManager*

class avocado.core.dispatcher.**SpawnerDispatcher**(*config=None*)
Bases: *avocado.core.enabled_extension_manager.EnabledExtensionManager*

class avocado.core.dispatcher.**VarianterDispatcher**
Bases: *avocado.core.enabled_extension_manager.EnabledExtensionManager*

map_method_with_return(*method_name*, **args*, ***kwargs*)
The same as *map_method* but additionally reports the list of returned values and optionally deepcopies the passed arguments

Parameters

- **method_name** – Name of the method to be called on each ext
- **args** – Arguments to be passed to all called functions
- **kwargs** – Key-word arguments to be passed to all called functions if *"deepcopy" == True* is present in kwargs the args and kwargs are deepcopied before passing it to each called function.

map_method_with_return_copy(*method_name*, **args*, ***kwargs*)
The same as map_method_with_return, but use copy.deepcopy on each passed arg

10.2.7 avocado.core.enabled_extension_manager module

Extension manager with disable/ordering support

class avocado.core.enabled_extension_manager.**EnabledExtensionManager**(*namespace*, *invoke_kwds=None*)
Bases: *avocado.core.extension_manager.ExtensionManager*

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.

10.2.8 avocado.core.exceptions module

Exception classes, useful for tests, and other parts of the framework code.

exception avocado.core.exceptions.**JobBaseException**
Bases: 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.**JobTestSuiteDuplicateNameError**
Bases: *avocado.core.exceptions.JobTestSuiteError*

Error raised when a test suite name is not unique in a job

status = 'ERROR'

exception avocado.core.exceptions.**JobTestSuiteEmptyError**
Bases: *avocado.core.exceptions.JobTestSuiteError*

Error raised when the creation of a test suite results in an empty suite

status = 'ERROR'

exception avocado.core.exceptions.**JobTestSuiteError**
Bases: *avocado.core.exceptions.JobBaseException*

Generic error happened during the creation of a job's test suite

status = 'ERROR'

exception avocado.core.exceptions.**JobTestSuiteReferenceResolutionError**
Bases: *avocado.core.exceptions.JobTestSuiteError*

Test References did not produce a valid reference by any resolver

status = 'ERROR'

exception avocado.core.exceptions.**OptionValidationError**
Bases: 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: 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*, 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.**TestFailFast**
Bases: *avocado.core.exceptions.TestBaseException*

Indicates that the test has failed because failfast is enabled.

Should be thrown when a test has failed and failfast is enabled. This will indicate that other tests will be skipped.

status = 'SKIP'

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.**TestSkipError**
Bases: *avocado.core.exceptions.TestBaseException*

Indicates 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'

10.2.9 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

10.2.10 avocado.core.extension_manager module

Base extension manager

This is a mix of stevedore-like APIs and behavior, with Avocado's own look and feel.

class avocado.core.extension_manager.**Extension**(*name*, *entry_point*, *plugin*, *obj*)
Bases: object

This is a verbatim copy from the stevedore.extension class with the same name

class avocado.core.extension_manager.**ExtensionManager**(*namespace*, *invoke_kwds=None*)
Bases: object

NAMESPACE_PREFIX = 'avocado.plugins.'
Default namespace prefix for Avocado extensions

enabled(*extension*)
Checks if a plugin is enabled

Sub classes can change this implementation to determine their own criteria.

fully_qualified_name(*extension*)
Returns the Avocado fully qualified plugin name

Parameters **extension** (*Extension*) – an Extension instance

map_method(*method_name*, **args*)
Maps method_name on each extension in case the extension has the attr

Parameters

- **method_name** – Name of the method to be called on each ext
- **args** – Arguments to be passed to all called functions

map_method_with_return(*method_name*, **args*, ***kwargs*)
The same as *map_method* but additionally reports the list of returned values and optionally deepcopies the passed arguments

Parameters

- **method_name** – Name of the method to be called on each ext
- **args** – Arguments to be passed to all called functions
- **kwargs** – Key-word arguments to be passed to all called functions if *"deepcopy" == True* is present in kwargs the args and kwargs are deepcopied before passing it to each called function.

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

10.2.11 avocado.core.job module

Job module - describes a sequence of automated test operations.

class avocado.core.job.**Job**(*config=None*, *test_suites=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.

A job has multiple test suites attached to it. Please keep in mind that when creating jobs from the constructor (*Job()*), we are assuming that you would like to have control of the test suites and you are going to build your own TestSuites.

If you would like any help to create the job's test_suites from the config provided, please use *Job.from_config()* method and we are going to do our best to create the test suites.

So, basically, as described we have two "main ways" to create a job:

1. Automatic discovery, using *from_config()* method:

```
job = Job.from_config(job_config=job_config,
                      suites_configs=[suite_cfg1, suite_cfg2])
```

2. Manual or Custom discovery, using the constructor:

```
job = Job(config=config,
          test_suites=[suite1, suite2, suite3])
```

Creates an instance of Job class.

Note that *config* and *test_suites* are optional, if not passed you need to change this before running your tests. Otherwise nothing will run. If you need any help to create the test_suites from the config, then use the *Job.from_config()* method.

Parameters

- **config** (*dict*) – the job configuration, usually set by command line options and argument parsing
- **test_suites** (*list*) – A list with TestSuite objects. If is None the job will have an empty list and you can add suites after init accessing job.test_suites.

cleanup()
Cleanup the temporary job handlers (dirs, global setting, . . .)

create_test_suite()

classmethod from_config(*job_config*, *suites_configs=None*)
Helper method to create a job from config dicts.

This is different from the Job() initialization because here we are assuming that you need some help to build the test suites. Avocado will try to resolve tests based on the configuration information instead of assuming pre populated test suites.

Keep in mind that here we are going to replace the suite.name with a counter.

If you need create a custom Job with your own TestSuites, please use the Job() constructor instead of this method.

Parameters

- **job_config** (*dict*) – A config dict to be used on this job and also as a 'global' config for each test suite.

- **suites_configs** (*list*) – A list of specific config dict to be used on each test suite. Each suite config will be merged with the job_config dict. If None is passed then this job will have only one test_suite with the same config as job_config.

get_failed_tests()
Gets the tests with status 'FAIL' and 'ERROR' after the Job ended.

Returns List of failed tests

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.

render_results()
Render test results that depend on all tests having finished.

By default this runs the plugins that implement the *avocado.core.plugin_interfaces.Result* interface.

result_events_dispatcher

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()
The actual test execution phase

setup()
Setup the temporary job handlers (dirs, global setting, . . .)

size
Job size is the sum of all test suites sizes.

test_results_path

test_suite
This is the first test suite of this job (deprecated).

Please, use test_suites instead.

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.

timeout

unique_id

avocado.core.job.**register_job_options**()
Register the few core options that the support the job operation.

10.2.12 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

10.2.13 avocado.core.jobdata module

Record/retrieve job information

avocado.core.jobdata.**get_variants_path**(*resultsdir*)
Retrieves the variants path from the results directory.

avocado.core.jobdata.**record**(*job*, *cmdline=None*)
Records all required job information.

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_job_config**(*resultsdir*)
Retrieves the job config 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.

10.2.14 avocado.core.loader module

Test loader module.

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

class avocado.core.loader.**DiscoverMode**
Bases: enum.Enum

An enumeration.

ALL = <object object>
All tests (including broken ones)

AVAILABLE = <object object>
Available tests (for listing purposes)

DEFAULT = <object object>
Show default tests (for execution)

class avocado.core.loader.**ExternalLoader**(*config*, *extra_params*)
Bases: *avocado.core.loader.TestLoader*

External-runner loader class

discover(*reference*, *which_tests=<DiscoverMode.DEFAULT: <object object>>*)

Parameters

- **reference** – arguments passed to the external_runner
- **which_tests** (*DiscoverMode*) – Limit tests to be displayed

Returns list of matching tests

static get_decorator_mapping()
Get label mapping for display in test listing.

Returns Dict {TestClass: decorator function}

static get_type_label_mapping()
Get label mapping for display in test listing.

Returns Dict {TestClass: 'TEST_LABEL_STRING'}

name = 'external'

class avocado.core.loader.**FileLoader**(*config*, *extra_params*)
Bases: *avocado.core.loader.SimpleFileLoader*

Test loader class.

NOT_TEST_STR = 'Not an INSTRUMENTED (avocado.Test based), PyUNITTEST (unittest.TestCas

static get_decorator_mapping()
Get label mapping for display in test listing.

Returns Dict {TestClass: decorator function}

static get_type_label_mapping()
Get label mapping for display in test listing.

Returns Dict {TestClass: 'TEST_LABEL_STRING'}

name = 'file'

exception avocado.core.loader.**InvalidLoaderPlugin**
Bases: *avocado.core.loader.LoaderError*

Invalid loader plugin

exception avocado.core.loader.**LoaderError**
Bases: 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.**SimpleFileLoader**(*config*, *extra_params*)
Bases: *avocado.core.loader.TestLoader*

Test loader class.

NOT_TEST_STR = 'Not a supported test'

discover(*reference*, *which_tests=<DiscoverMode.DEFAULT: <object object>>*)
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** (*DiscoverMode*) – Limit tests to be displayed

Returns list of matching tests

static get_decorator_mapping()
Get label mapping for display in test listing.

Returns Dict {TestClass: decorator function}

static get_type_label_mapping()
Get label mapping for display in test listing.

Returns Dict {TestClass: 'TEST_LABEL_STRING'}

name = 'file'

class avocado.core.loader.**TapLoader**(*config*, *extra_params*)
Bases: *avocado.core.loader.SimpleFileLoader*

Test Anything Protocol loader class

static get_decorator_mapping()
Get label mapping for display in test listing.

Returns Dict {TestClass: decorator function}

static get_type_label_mapping()
Get label mapping for display in test listing.

Returns Dict {TestClass: 'TEST_LABEL_STRING'}

name = 'tap'

class avocado.core.loader.**TestLoader**(*config*, *extra_params*)
Bases: object

Base for test loader classes

discover(*reference*, *which_tests=<DiscoverMode.DEFAULT: <object object>>*)
Discover (possible) tests from an reference.

Parameters

- **reference** (*str*) – the reference to be inspected.
- **which_tests** (*DiscoverMode*) – Limit tests to be displayed

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()

get_full_decorator_mapping()
Allows extending the decorator-mapping after the object is initialized

get_full_type_label_mapping()
Allows extending the type-label-mapping after the object is initialized

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=<DiscoverMode.DEFAULT: <object object>>*, *force=None*)
Discover (possible) tests from test references.

Parameters

- **references** (*builtin.list*) – a list of tests references; if [] use plugin defaults
- **which_tests** (*DiscoverMode*) – Limit tests to be displayed
- **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(*config*)

static 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*, *section='run'*)

10.2.15 avocado.core.main module

avocado.core.main.**get_crash_dir**()

avocado.core.main.**handle_exception**(**exc_info*)

avocado.core.main.**main**()

10.2.16 avocado.core.messages module

class avocado.core.messages.**BaseMessageHandler**
Bases: object

Base interface for resolving runner messages.

This is the interface a job uses to deal with messages from runners.

handle(*message*, *task*, *job*)
Handle message from runner.

Parameters

- **message** (*dict*) – message from runner.
- **task** (*avocado.core.nrunner.Task*) – runtime_task which message is related to
- **job** (*avocado.core.job.Job*) – job which task is related to

process_message(*message*, *task*, *job*)
It transmits the message to the right handler.

Parameters

- **message** (*dict*) – message from runner
- **task** (*avocado.core.nrunner.Task*) – runtime_task which message is related to
- **job** (*avocado.core.job.Job*) – job which task is related to

class avocado.core.messages.**BaseRunningMessageHandler**
Bases: *avocado.core.messages.BaseMessageHandler*

Base interface for resolving running messages.

class avocado.core.messages.**FileMessageHandler**
Bases: *avocado.core.messages.BaseRunningMessageHandler*

Handler for file message.

In task directory will save log into the runner specific file. When the file doesn't exist, the file will be created. If the file exist, the message data will be appended at the end.

Parameters

- **status** – 'running'
- **type** – 'file'
- **path** (*string*) – relative path to the file. The file will be created under the Task directory and the absolute path will be created as *absolute_task_directory_path/relative_file_path*.
- **log** (*bytes*) – data to be saved inside file
- **time** (*float*) – Time stamp of the message

example: {'status': 'running', 'type': 'file', 'path':'foo/runner.log', 'log': 'this will be saved inside file', 'time': 18405.55351474}

handle (*message*, *task*, *job*)
Handle message from runner.

Parameters

- **message** (*dict*) – message from runner.
- **task** (*avocado.core.nrunner.Task*) – runtime_task which message is related to
- **job** (*avocado.core.job.Job*) – job which task is related to

class avocado.core.messages.**FinishMessageHandler**
Bases: *avocado.core.messages.BaseMessageHandler*

Handler for finished message.

It will report the test status and triggers the 'end_test' event.

This is triggered when the runner ends the test.

Parameters

- **status** – 'finished'
- **result** (*avocado.core.teststatus.STATUSES*) – test result
- **time** (*float*) – end time of the test
- **fail_reason** (*string*) – Optional parameter for brief specification, of the failed result.

example: {'status': 'finished', 'result': 'pass', 'time': 16444.819830573}

handle (*message*, *task*, *job*)
Handle message from runner.

Parameters

- **message** (*dict*) – message from runner.
- **task** (*avocado.core.nrunner.Task*) – runtime_task which message is related to
- **job** (*avocado.core.job.Job*) – job which task is related to

class avocado.core.messages.**LogMessageHandler**
Bases: *avocado.core.messages.BaseRunningMessageHandler*

Handler for log message.

It will save the log to the debug.log file in the task directory.

Parameters

- **status** – 'running'

- **type** – 'log'
- **log** (*string*) – log message
- **time** (*float*) – Time stamp of the message

example: {'status': 'running', 'type': 'log', 'log': 'log message', 'time': 18405.55351474}

handle(*message*, *task*, *job*)
Logs a textual message to a file.

This assumes that the log message will not contain a newline, and thus one is explicitly added here.

class avocado.core.messages.**MessageHandler**
Bases: *avocado.core.messages.BaseMessageHandler*

Entry point for handling messages.

process_message(*message*, *task*, *job*)
It transmits the message to the right handler.

Parameters

- **message** (*dict*) – message from runner
- **task** (*avocado.core.nrunner.Task*) – runtime_task which message is related to
- **job** (*avocado.core.job.Job*) – job which task is related to

class avocado.core.messages.**RunningMessageHandler**
Bases: *avocado.core.messages.BaseMessageHandler*

Entry point for handling running messages.

process_message(*message*, *task*, *job*)
It transmits the message to the right handler.

Parameters

- **message** (*dict*) – message from runner
- **task** (*avocado.core.nrunner.Task*) – runtime_task which message is related to
- **job** (*avocado.core.job.Job*) – job which task is related to

class avocado.core.messages.**StartMessageHandler**
Bases: *avocado.core.messages.BaseMessageHandler*

Handler for started message.

It will create the test base directories and triggers the 'start_test' event.

This have to be triggered when the runner starts the test.

Parameters

- **status** – 'started'
- **time** (*float*) – start time of the test

example: {'status': 'started', 'time': 16444.819830573}

handle(*message*, *task*, *job*)
Handle message from runner.

Parameters

- **message** (*dict*) – message from runner.

- **task** (*avocado.core.nrunner.Task*) – runtime_task which message is related to
- **job** (*avocado.core.job.Job*) – job which task is related to

class avocado.core.messages.**StderrMessageHandler**

Bases: *avocado.core.messages.BaseRunningMessageHandler*

Handler for stderr message.

It will save the stderr to the stderr and debug file in the task directory.

Parameters

- **status** – 'running'
- **type** – 'stderr'
- **log** (*bytes*) – stderr message
- **encoding** (*str*) – optional value for decoding messages
- **time** (*float*) – Time stamp of the message

example: {'status': 'running', 'type': 'stderr', 'log': 'stderr message', 'time': 18405.55351474}

handle(*message*, *task*, *job*)

Handle message from runner.

Parameters

- **message** (*dict*) – message from runner.
- **task** (*avocado.core.nrunner.Task*) – runtime_task which message is related to
- **job** (*avocado.core.job.Job*) – job which task is related to

class avocado.core.messages.**StdoutMessageHandler**

Bases: *avocado.core.messages.BaseRunningMessageHandler*

Handler for stdout message.

It will save the stdout to the stdout and debug file in the task directory.

Parameters

- **status** – 'running'
- **type** – 'stdout'
- **log** (*bytes*) – stdout message
- **encoding** (*str*) – optional value for decoding messages
- **time** (*float*) – Time stamp of the message

example: {'status': 'running', 'type': 'stdout', 'log': 'stdout message', 'time': 18405.55351474}

handle(*message*, *task*, *job*)

Handle message from runner.

Parameters

- **message** (*dict*) – message from runner.
- **task** (*avocado.core.nrunner.Task*) – runtime_task which message is related to
- **job** (*avocado.core.job.Job*) – job which task is related to

class avocado.core.messages.**WhiteboardMessageHandler**
Bases: *avocado.core.messages.BaseRunningMessageHandler*

Handler for whiteboard message.

It will save the stderr to the whiteboard file in the task directory.

Parameters

- **status** – 'running'
- **type** – 'whiteboard'
- **log** (*bytes*) – whiteboard message
- **encoding** (*str*) – optional value for decoding messages
- **time** (*float*) – Time stamp of the message

example: {'status': 'running', 'type': 'whiteboard', 'log': 'whiteboard message', 'time': 18405.55351474}

handle(*message*, *task*, *job*)
Handle message from runner.

Parameters

- **message** (*dict*) – message from runner.
- **task** (*avocado.core.nrunner.Task*) – runtime_task which message is related to
- **job** (*avocado.core.job.Job*) – job which task is related to

10.2.17 avocado.core.nrunner module

class avocado.core.nrunner.**BaseRunner**(*runnable*)
Bases: abc.ABC

Base interface for a Runner

static prepare_status(*status_type*, *additional_info=None*)
Prepare a status dict with some basic information.

This will add the keyword 'status' and 'time' to all status.

Param status_type: The type of event ('started', 'running', 'finished')

Param addional_info: Any additional information that you would like to add to the dict. This must be a dict.

Return type dict

run()
Runner main method

Yields dictionary as output, containing status as well as relevant information concerning the results.

class avocado.core.nrunner.**BaseRunnerApp**(*echo=<built-in function print>*, *prog=None*, *description=None*)
Bases: object

Helper base class for common runner application behavior

CMD_RUNNABLE_RUN_ARGS = ((('-k', '--kind'), {'type': <class 'str'>, 'help': 'Kind of
The command line arguments to the "runnable-run" command

CMD_RUNNABLE_RUN_RECIPE_ARGS = ((('recipe',), {'type': <class 'str'>, 'help': 'Path

CMD_STATUS_SERVER_ARGS = ((('uri',), {'type': <class 'str'>, 'help': 'URI to bind a

CMD_TASK_RUN_ARGS = ((('-i', '--identifier'), {'type': <class 'str'>, 'required': Tr

CMD_TASK_RUN_RECIPE_ARGS = ((('recipe',), {'type': <class 'str'>, 'help': 'Path to t

PROG_DESCRIPTION = ''
The description of the command line application given to the command line parser

PROG_NAME = ''
The name of the command line application given to the command line parser

RUNNABLE_KINDS_CAPABLE = {}
The types of runnables that this runner can handle. Dictionary key is a name, and value is a class that inherits from *BaseRunner*

command_capabilities(_)
Outputs capabilities, including runnables and commands

The output is intended to be consumed by upper layers of Avocado, such as the Job layer selecting the right runner script to handle a runnable of a given kind, or identifying if a runner script has a given feature (as implemented by a command).

command_runnable_run(*args*)
Runs a runnable definition from arguments

This defines a Runnable instance purely from the command line arguments, then selects a suitable Runner, and runs it.

Parameters **args** (*dict*) – parsed command line arguments turned into a dictionary

command_runnable_run_recipe(*args*)
Runs a runnable definition from a recipe

Parameters **args** (*dict*) – parsed command line arguments turned into a dictionary

command_task_run(*args*)
Runs a task from arguments

Parameters **args** (*dict*) – parsed command line arguments turned into a dictionary

command_task_run_recipe(*args*)
Runs a task from a recipe

Parameters **args** (*dict*) – parsed command line arguments turned into a dictionary

get_capabilities()
Returns the runner capabilities, including runnables and commands

This can be used by higher level tools, such as the entity spawning runners, to know which runner can be used to handle each runnable type.

Return type dict

get_commands()
Return the command names, as seen on the command line application

For every method whose name starts with "command", and the name of the command follows, with underscores replaced by dashes. So, a method named "command_foo_bar", will be a command available on the command line as "foo-bar".

Return type list

get_runner_from_runnable(*runnable*)
Returns a runner that is suitable to run the given runnable

Return type instance of class inheriting from `BaseRunner`

Raises ValueError if runnable is now supported

run()
Runs the application by finding a suitable command method to call

class avocado.core.nrunner.**ConfigDecoder**(*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *strict=True*, *object_pairs_hook=None*)

Bases: `json.decoder.JSONDecoder`

JSON Decoder for config options.

`object_hook`, if specified, will be called with the result of every JSON object decoded and its return value will be used in place of the given `dict`. This can be used to provide custom deserializations (e.g. to support JSON-RPC class hinting).

`object_pairs_hook`, if specified will be called with the result of every JSON object decoded with an ordered list of pairs. The return value of `object_pairs_hook` will be used instead of the `dict`. This feature can be used to implement custom decoders. If `object_hook` is also defined, the `object_pairs_hook` takes priority.

`parse_float`, if specified, will be called with the string of every JSON float to be decoded. By default this is equivalent to float(num_str). This can be used to use another datatype or parser for JSON floats (e.g. decimal.Decimal).

`parse_int`, if specified, will be called with the string of every JSON int to be decoded. By default this is equivalent to int(num_str). This can be used to use another datatype or parser for JSON integers (e.g. float).

`parse_constant`, if specified, will be called with one of the following strings: -Infinity, Infinity, NaN. This can be used to raise an exception if invalid JSON numbers are encountered.

If `strict` is false (true is the default), then control characters will be allowed inside strings. Control characters in this context are those with character codes in the 0-31 range, including `'\t'` (tab), `'\n'`, `'\r'` and `'\0'`.

decode(*config_str*)
Return the Python representation of `s` (a `str` instance containing a JSON document).

static decode_set(*config_dict*)

class avocado.core.nrunner.**ConfigEncoder**(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

Bases: `json.encoder.JSONEncoder`

JSON Encoder for config options.

Constructor for JSONEncoder, with sensible defaults.

If skipkeys is false, then it is a TypeError to attempt encoding of keys that are not str, int, float or None. If skipkeys is True, such items are simply skipped.

If ensure_ascii is true, the output is guaranteed to be str objects with all incoming non-ASCII characters escaped. If ensure_ascii is false, the output can contain non-ASCII characters.

If check_circular is true, then lists, dicts, and custom encoded objects will be checked for circular references during encoding to prevent an infinite recursion (which would cause an OverflowError). Otherwise, no such check takes place.

If allow_nan is true, then NaN, Infinity, and -Infinity will be encoded as such. This behavior is not JSON specification compliant, but is consistent with most JavaScript based encoders and decoders. Otherwise, it will be a ValueError to encode such floats.

If sort_keys is true, then the output of dictionaries will be sorted by key; this is useful for regression tests to ensure that JSON serializations can be compared on a day-to-day basis.

If indent is a non-negative integer, then JSON array elements and object members will be pretty-printed with that indent level. An indent level of 0 will only insert newlines. None is the most compact representation.

If specified, separators should be an (item_separator, key_separator) tuple. The default is (', ', ': ') if *indent* is `None` and (',', ': ') otherwise. To get the most compact JSON representation, you should specify (',', ':') to eliminate whitespace.

If specified, default is a function that gets called for objects that can't otherwise be serialized. It should return a JSON encodable version of the object or raise a `TypeError`.

default(*config_option*)

Implement this method in a subclass such that it returns a serializable object for `o`, or calls the base implementation (to raise a `TypeError`).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

class avocado.core.nrunner.**DryRunRunner**(*runnable*)

Bases: *avocado.core.nrunner.BaseRunner*

Runner for –dry-run.

It performs no action before reporting FINISHED status with cancel result.

Runnable attributes usage:

- uri: not used
- args: not used

run()

Runner main method

Yields dictionary as output, containing status as well as relevant information concerning the results.

class avocado.core.nrunner.**ExecTestRunner**(*runnable*)

Bases: *avocado.core.nrunner.BaseRunner*

Runner for standalone executables treated as tests

This is similar in concept to the Avocado "SIMPLE" test type, in which an executable returning 0 means that a test passed, and anything else means that a test failed.

Runnable attributes usage:

- uri: path to a binary to be executed as another process
- args: arguments to be given on the command line to the binary given by path

- kwargs: key=val to be set as environment variables to the process

run()
Runner main method

Yields dictionary as output, containing status as well as relevant information concerning the results.

class avocado.core.nrunner.**NoOpRunner**(*runnable*)
Bases: *avocado.core.nrunner.BaseRunner*

Sample runner that performs no action before reporting FINISHED status

Runnable attributes usage:

- uri: not used
- args: not used

run()
Runner main method

Yields dictionary as output, containing status as well as relevant information concerning the results.

class avocado.core.nrunner.**PythonUnittestRunner**(*runnable*)
Bases: *avocado.core.nrunner.BaseRunner*

Runner for Python unittests

The runnable uri is used as the test name that the native unittest TestLoader will use to find the test. A native unittest test runner (TextTestRunner) will be used to execute the test.

Runnable attributes usage:

- **uri: a single test reference, that is "a test method within a test case** class" within a test module. Example is: "./tests/foo.py:ClassFoo.test_bar".
- args: not used
- kwargs: not used

module_class_method
Return a dotted name with module + class + method.

Important to note here that module is only the module file without the full path.

module_path
Path where the module is located.

Ex: uri = './avocado.dev/selftests/.data/unittests/test.py:Class.test_foo' It will return './avocado.dev/selftests/.data/unittests/'

run()
Runner main method

Yields dictionary as output, containing status as well as relevant information concerning the results.

unittest
Returns the unittest part of an uri as tuple.

Ex:

uri = './avocado.dev/selftests/.data/unittests/test.py:Class.test_foo' It will return ("test", "Class", "test_foo")

avocado.core.nrunner.**RUNNERS_REGISTRY_PYTHON_CLASS = {'dry-run': <class 'avocado.core.nrun**
All known runner Python classes. This is a dictionary keyed by a runnable kind, and value is a class that inherits from *BaseRunner*. Suitable for spawners compatible with SpawnMethod.PYTHON_CLASS

avocado.core.nrunner.**RUNNERS_REGISTRY_STANDALONE_EXECUTABLE = {}**
All known runner commands, capable of being used by a SpawnMethod.STANDALONE_EXECUTABLE compatible spawners

avocado.core.nrunner.**RUNNER_RUN_CHECK_INTERVAL = 0.01**
The amount of time (in seconds) between each internal status check

avocado.core.nrunner.**RUNNER_RUN_STATUS_INTERVAL = 0.5**
The amount of time (in seconds) between a status report from a runner that performs its work asynchronously

class avocado.core.nrunner.**Runnable**(*kind*, *uri*, **args*, *config=None*, ***kwargs*)
Bases: object

Describes an entity that be executed in the context of a task

A instance of *BaseRunner* is the entity that will actually execute a runnable.

classmethod from_args(*args*)
Returns a runnable from arguments

classmethod from_recipe(*recipe_path*)
Returns a runnable from a runnable recipe file

Parameters **recipe_path** – Path to a recipe file

Return type instance of *Runnable*

get_command_args()
Returns the command arguments that adhere to the runner interface

This is useful for building 'runnable-run' and 'task-run' commands that can be executed on a command line interface.

Returns the arguments that can be used on an avocado-runner command

Return type list

get_dict()
Returns a dictionary representation for the current runnable

This is usually the format that will be converted to a format that can be serialized to disk, such as JSON.

Return type collections.OrderedDict

get_json()
Returns a JSON representation

Return type str

get_serializable_tags()

is_kind_supported_by_runner_command(*runner_command*)
Checks if a runner command that seems a good fit declares support.

pick_runner_class(*runners_registry=None*)
Selects a runner class from the registry based on kind.

This is related to the SpawnMethod.PYTHON_CLASS

Parameters

- **runners_registry** – a registry with previously registered runner classes, keyed by runnable kind
- **runners_registry** – dict

Returns a class that inherits from *BaseRunner*

Raises ValueError if kind there's no runner from kind of runnable

pick_runner_class_from_entry_point()
Selects a runner class from entry points based on kind.

This is related to the `SpawnMethod.PYTHON_CLASS`. This complements the *RUNNERS_REGISTRY_PYTHON_CLASS* on systems that have setuptools available.

Returns a class that inherits from *BaseRunner* or None

pick_runner_command(*runners_registry=None*)
Selects a runner command based on the runner.

And when finding a suitable runner, keeps found runners in registry.

This utility function will look at the given task and try to find a matching runner. The matching runner probe results are kept in a registry (that is modified by this function) so that further executions take advantage of previous probes.

This is related to the `SpawnMethod.STANDALONE_EXECUTABLE`

Parameters

- **runners_registry** – a registry with previously found (and not found) runners keyed by runnable kind
- **runners_registry** – dict

Returns command line arguments to execute the runner

Return type list of str or None

write_json(*recipe_path*)
Writes a file with a JSON representation (also known as a recipe)

class avocado.core.nrunner.**RunnerApp**(*echo=<built-in function print>*, *prog=None*, *description=None*)
Bases: *avocado.core.nrunner.BaseRunnerApp*

PROG_DESCRIPTION = 'nrunner base application'

PROG_NAME = 'avocado-runner'

RUNNABLE_KINDS_CAPABLE = {'dry-run': <class 'avocado.core.nrunner.DryRunRunner'>, 'ex

class avocado.core.nrunner.**StatusEncoder**(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)
Bases: `json.encoder.JSONEncoder`

Constructor for JSONEncoder, with sensible defaults.

If skipkeys is false, then it is a TypeError to attempt encoding of keys that are not str, int, float or None. If skipkeys is True, such items are simply skipped.

If ensure_ascii is true, the output is guaranteed to be str objects with all incoming non-ASCII characters escaped. If ensure_ascii is false, the output can contain non-ASCII characters.

If check_circular is true, then lists, dicts, and custom encoded objects will be checked for circular references during encoding to prevent an infinite recursion (which would cause an OverflowError). Otherwise, no such check takes place.

If allow_nan is true, then NaN, Infinity, and -Infinity will be encoded as such. This behavior is not JSON specification compliant, but is consistent with most JavaScript based encoders and decoders. Otherwise, it will be a ValueError to encode such floats.

If sort_keys is true, then the output of dictionaries will be sorted by key; this is useful for regression tests to ensure that JSON serializations can be compared on a day-to-day basis.

If indent is a non-negative integer, then JSON array elements and object members will be pretty-printed with that indent level. An indent level of 0 will only insert newlines. None is the most compact representation.

If specified, separators should be an (item_separator, key_separator) tuple. The default is (', ', ': ') if *indent* is `None` and (',', ': ') otherwise. To get the most compact JSON representation, you should specify (',', ':') to eliminate whitespace.

If specified, default is a function that gets called for objects that can't otherwise be serialized. It should return a JSON encodable version of the object or raise a `TypeError`.

default(*o*)

Implement this method in a subclass such that it returns a serializable object for `o`, or calls the base implementation (to raise a `TypeError`).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

avocado.core.nrunner.**TASK_DEFAULT_CATEGORY = 'test'**

The default category for tasks, and the value that will cause the task results to be included in the job results

class avocado.core.nrunner.**Task**(*runnable*, *identifier=None*, *status_uris=None*, *known_runners=None*, *category='test'*, *job_id=None*)

Bases: `object`

Wraps the execution of a runnable

While a runnable describes what to be run, and gets run by a runner, a task should be a unique entity to track its state, that is, whether it is pending, is running or has finished.

Instantiates a new Task.

Parameters

- **runnable** (*avocado.core.nrunner.Runnable*) – the "description" of what the task should run.
- **identifier** – any identifier that is guaranteed to be unique within the context of a Job. A recommended value is a *avocado.core.test_id.TestID* instance when a task represents a test, because besides the uniqueness aspect, it's also descriptive. If an identifier is not given, an automatically generated one will be set.
- **status_uri** (*list*) – the URIs for the status servers that this task should send updates to.
- **known_runners** (*dict*) – a mapping of runnable kinds to runners.
- **category** (*str*) – category of this task. Defaults to *TASK_DEFAULT_CATEGORY*.
- **job_id** (*str*) – the ID of the job, for authenticating messages that get sent to the destination job's status server and will make into the job's results.

are_requirements_available(*runners_registry=None*)
Verifies if requirements needed to run this task are available.

This currently checks the runner command only, but can be expanded once the handling of other types of requirements are implemented. See *BP002*.

category = None
Category of the task. If the category is not "test", it will not be accounted for on a Job's test results.

classmethod from_recipe(*task_path*, *known_runners*)
Creates a task (which contains a runnable) from a task recipe file

Parameters

- **task_path** – Path to a recipe file
- **known_runners** – Dictionary with runner names and implementations

Return type instance of *Task*

get_command_args()
Returns the command arguments that adhere to the runner interface

This is useful for building 'task-run' commands that can be executed on a command line interface.

Returns the arguments that can be used on an avocado-runner command

Return type list

run()

setup_output_dir()

class avocado.core.nrunner.**TaskStatusService**(*uri*)
Bases: object

Implementation of interface that a task can use to post status updates

TODO: make the interface generic and this just one of the implementations

close()

post(*status*)

avocado.core.nrunner.**check_runnables_runner_requirements**(*runnables*, *runners_registry=None*)
Checks if runnables have runner requirements fulfilled

Parameters

- **runnables** – the tasks whose runner requirements will be checked
- **runners_registry** (*dict*) – a registry with previously found (and not found) runners keyed by a task's runnable kind. Defaults to *RUNNERS_REGISTRY_STANDALONE_EXECUTABLE*

Returns two list of tasks in a tuple, with the first being the tasks that pass the requirements check and the second the tasks that fail the requirements check

Return type tuple of (list, list)

avocado.core.nrunner.**json_dumps**(*data*)

avocado.core.nrunner.**main**(*app_class=<class 'avocado.core.nrunner.RunnerApp'>*)

10.2.18 avocado.core.output module

Manages output and logging in avocado applications.

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*)
Determine if the specified record is to be logged.

Is the specified record to be logged? Returns 0 for no, nonzero for yes. If deemed appropriate, the record may be modified in-place.

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*)
Determine if the specified record is to be logged.

Is the specified record to be logged? Returns 0 for no, nonzero for yes. If deemed appropriate, the record may be modified in-place.

avocado.core.output.**LOG_JOB = <Logger avocado.test (WARNING)>**
Pre-defined Avocado job/test logger

avocado.core.output.**LOG_UI = <Logger avocado.app (WARNING)>**
Pre-defined Avocado human UI logger

class avocado.core.output.**LoggingFile**(*prefixes=None*, *level=10*, *loggers=None*)
Bases: object

File-like object that will receive messages pass them to logging.

Constructor. Sets prefixes and which loggers are going to be used.

Parameters

- **prefixes** – Prefix per logger to be prefixed to each line.
- **level** – Log level to be used when writing messages.
- **loggers** – Loggers into which write should be issued. (list)

add_logger(*logger*, *prefix=''*)

flush()

static isatty()

rm_logger(*logger*)

write(*data*)
" Splits the line to individual lines and forwards them into loggers with expected prefixes. It includes the tailing newline <lf> as well as the last partial message. Do configure your logging to not to add newline <lf> automatically. :param data - Raw data (a string) 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*)
Emit a record.

If a formatter is specified, it is used to format the record. The record is then written to the stream with a trailing newline. If exception information is present, it is formatted using traceback.print_exception and appended to the stream. If the stream has an 'encoding' attribute, it is used to determine how to do the output to the stream.

flush()
This is in-mem object, it does not require flushing

log = []

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*)
Emit a record.

If a formatter is specified, it is used to format the record. The record is then written to the stream with a trailing newline. If exception information is present, it is formatted using traceback.print_exception and appended to the stream. If the stream has an 'encoding' attribute, it is used to determine how to do the output to the stream.

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

configured
Determines if a configuration of any sort has been performed

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

avocado.core.output.**TEST_STATUS_DECORATOR_MAPPING = {'CANCEL': <bound method TermSupport.sl**
A collection of mapping from test status to formatting functions to be used consistently across the various plugins

avocado.core.output.**TEST_STATUS_MAPPING = {'CANCEL': '', 'ERROR': '', 'FAIL': '', 'INTERRU**
A collection of mapping from test statuses to colors to be used consistently across the various plugins

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',
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(*msg='ERROR'*, *move='\x1b[1D'*)
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(*msg='FAIL'*, *move='\x1b[1D'*)
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(*msg='INTERRUPT'*, *move='\x1b[1D'*)
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(*msg='PASS'*, *move='\x1b[1D'*)
Print a pass string (green colored).

If the output does not support colors, just return the original string.

skip_str(*msg='SKIP'*, *move='\x1b[1D'*)
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(*msg='WARN'*, *move='\x1b[1D'*)
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=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*, *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.`**del_last_configuration**`()`

`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.

10.2.19 avocado.core.parameters module

Module related to test parameters

class `avocado.core.parameters.`**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.parameters.`**AvocadoParams**`(`*leaves*, *paths*, *logger_name=None*`)`
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.

Parameters

- **leaves** – List of TreeNode leaves defining current variant

- **paths** – list of entry points
- **logger_name** (*str*) – the name of a logger to use to record attempts to get parameters

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.

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.parameters.**NoMatchError**
Bases: KeyError

10.2.20 avocado.core.parser module

Avocado application command line parsing.

class avocado.core.parser.**ArgumentParser**(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *parents=[]*, *formatter_class=<class 'argparse.HelpFormatter'>*, *prefix_chars='-'*, *fromfile_prefix_chars=None*, *argument_default=None*, *conflict_handler='error'*, *add_help=True*, *allow_abbrev=True*)
Bases: argparse.ArgumentParser

Class to override argparse functions

error(*message: string*)
Prints a usage message incorporating the message to stderr and exits.

If you override this in a subclass, it should not return – it should either exit or raise an exception.

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.**HintParser**(*filename*)
Bases: object

get_resolutions()
Return a list of resolutions based on the file definitions.

validate_kind_section(*kind*)
Validates a specific "kind section".

This method will raise a *settings.SettingsError* if any problem is found on the file.

Parameters **kind** – a string with the specific section.

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 on attribute *config*.

start()
Start to parsing arguments.

At the end of this method, the support for subparsers is activated. Side effect: update attribute *args* (the namespace).

10.2.21 avocado.core.parser_common_args module

avocado.core.parser_common_args.**add_tag_filter_args**(*parser*)

10.2.22 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(*config*)
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(*config*)
Entry point for actually running the command.

class avocado.core.plugin_interfaces.**Discoverer**(*config=None*)
Bases: *avocado.core.plugin_interfaces.Plugin*, *avocado.core.plugin_interfaces.ResolverMixin*

Base plugin interface for discovering tests without reference.

discover()
Discovers a test resolutions

It will be used when the *test.references* variable is empty, but the discoverer will be able to use another data for gathering test resolutions. It work same as the Resolver, but without the test reference.

Returns the result of the resolution process, containing the success, failure or error, along with zero or more *avocado.core.nrunner.Runnable* objects

Return type *avocado.core.resolver.ReferenceResolution*

class avocado.core.plugin_interfaces.**Init**
Bases: *avocado.core.plugin_interfaces.Plugin*

Base plugin interface for plugins that needs to initialize itself.

initialize()
Entry point for the plugin to perform its initialization.

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: abc.ABC

Base for all plugins.

class avocado.core.plugin_interfaces.**Resolver**(*config=None*)
Bases: *avocado.core.plugin_interfaces.Plugin*, *avocado.core.plugin_interfaces.ResolverMixin*

Base plugin interface for resolving test references into resolutions.

resolve(*reference*)
Resolves the given reference into a reference resolution.

Parameters **reference** (*str*) – a specification that can eventually be resolved into a test (in the form of a *avocado.core.nrunner.Runnable*)

Returns the result of the resolution process, containing the success, failure or error, along with zero or more *avocado.core.nrunner.Runnable* objects

Return type *avocado.core.resolver.ReferenceResolution*

class avocado.core.plugin_interfaces.**ResolverMixin**(*config=None*)
Bases: object

Common utilities for Resolver implementations.

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.**Runner**
Bases: *avocado.core.plugin_interfaces.Plugin*

Base plugin interface for test runners.

This is the interface a job uses to drive the tests execution via compliant test runners.

NOTE: This interface is not to be confused with the internal interface or idiosyncrasies of the *The "nrunner" and "legacy runner" test runner*.

run_suite(*job*, *test_suite*)
Run one or more tests and report with test result.

Parameters

- **job** – an instance of *avocado.core.job.Job*.
- **test_suite** – an instance of TestSuite with some tests to run.

Returns a set with types of test failures.

class avocado.core.plugin_interfaces.**Settings**
Bases: *avocado.core.plugin_interfaces.Plugin*

Base plugin to allow modifying settings.

Currently it only supports to extend/modify the default list of paths to config files.

adjust_settings_paths(*paths*)
Entry point where plugin can modify the list of configuration paths.

class avocado.core.plugin_interfaces.**Spawner**
Bases: *avocado.core.plugin_interfaces.Plugin*

Base plugin interface spawners of tasks.

A spawner implementation will spawn a runner in its intended location, and isolation model. It's supposed to be generic enough that it can perform that in the local machine using a process as an isolation model, or in a virtual machine, using the virtual machine itself as the isolation model.

static check_task_requirements(*runtime_task*)
Checks if the requirements described within a task are available.

Parameters **runtime_task** (*avocado.core.task.runtime.RuntimeTask*) – wrapper for a Task with additional runtime information

static is_task_alive(*runtime_task*)
Determines if a task is alive or not.

Parameters **runtime_task** (*avocado.core.task.runtime.RuntimeTask*) – wrapper for a Task with additional runtime information

spawn_task(*runtime_task*)
Spawns a task return whether the spawning was successful.

Parameters **runtime_task** (*avocado.core.task.runtime.RuntimeTask*) – wrapper for a Task with additional runtime information

wait_task(*runtime_task*)
Waits for a task to finish.

Parameters **runtime_task** (*avocado.core.task.runtime.RuntimeTask*) – wrapper for a Task with additional runtime information

class avocado.core.plugin_interfaces.**Varianter**
Bases: *avocado.core.plugin_interfaces.Plugin*

Base plugin interface for producing test variants.

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

10.2.23 avocado.core.references module

Test loader module.

avocado.core.references.**reference_split**(*reference*)
Splits a test reference into a path and additional info

This should be used dependent on the specific type of resolver. If a resolver is not expected to support multiple test references inside a given file, then this is not suitable.

Returns (path, additional_info)

Type (str, str or None)

10.2.24 avocado.core.resolver module

Test resolver module.

class avocado.core.resolver.**Discoverer**(*config=None*)
Bases: *avocado.core.enabled_extension_manager.EnabledExtensionManager*

Secondary test reference resolution utility.

When the user didn't provide any test references, Discoverer will discover tests from different data according to active discoverer plugins.

discover()

class avocado.core.resolver.**ReferenceResolution**(*reference*, *result*, *resolutions=None*, *info=None*, *origin=None*)
Bases: object

Represents one complete reference resolution

Note that the reference itself may result in many resolutions, or none.

Parameters

- **reference** (*str*) – a specification that can eventually be resolved into a test (in the form of a *avocado.core.nrunner.Runnable*)
- **result** (*ReferenceResolutionResult*) – if the complete resolution was a success, failure or error

- **resolutions** (list of *avocado.core.nrunner.Runnable*) – the runnable definitions resulting from the resolution
- **info** (*str*) – free form information the resolver may add
- **origin** (*str*) – the name of the resolver that performed the resolution

class avocado.core.resolver.**ReferenceResolutionAction**
Bases: enum.Enum

An enumeration.

CONTINUE = <object object>
Continue to resolve the given reference

RETURN = <object object>
Stop trying to resolve the reference

class avocado.core.resolver.**ReferenceResolutionResult**
Bases: enum.Enum

An enumeration.

ERROR = <object object>
Internal error in the resolution process

NOTFOUND = <object object>
Given test reference was not properly resolved

SUCCESS = <object object>
Given test reference was properly resolved

class avocado.core.resolver.**Resolver**(*config=None*)
Bases: *avocado.core.enabled_extension_manager.EnabledExtensionManager*

Main test reference resolution utility.

This performs the actual resolution according to the active resolver plugins and a resolution policy.

DEFAULT_POLICY = {<ReferenceResolutionResult.SUCCESS: <object object>>: <ReferenceRes

resolve(*reference*)

avocado.core.resolver.**check_file**(*path*, *reference*, *suffix='.py'*, *type_check=<function isfile>*, *type_name='regular file'*, *access_check=4*, *access_name='readable'*)

avocado.core.resolver.**resolve**(*references*, *hint=None*, *ignore_missing=True*, *config=None*)

10.2.25 avocado.core.result module

Contains the Result class, used for result accounting.

class avocado.core.result.**Result**(*job_unique_id*, *job_logfile*)
Bases: object

Result class, holder for job (and its tests) result information.

Creates an instance of Result.

Parameters

- **job_unique_id** – the job's unique ID, usually from *avocado.core.job.Job.unique_id*

- **job_logfile** – the job's unique ID, usually from `avocado.core.job.Job.logfile`

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.

rate

start_test(*state*)
Called when the given test is about to run.

Parameters **state** (*dict*) – result of *avocado.core.test.Test.get_state*.

10.2.26 avocado.core.runner module

Test runner module.

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

10.2.27 avocado.core.settings module

This module is a new and experimental configuration handler.

This will handle both, command line args and configuration files. Settings() = configparser + argparser

Settings() is an attempt to implement part of BP001 and concentrate all default values in one place. This module will read the Avocado configuration options from many sources, in the following order:

1. Default values: This is a "source code" defined. When plugins or core needs a settings, basically needs to call settings.register_option() with default value as argument. Developers only need to register the default value once, here when calling this methods.
2. User/System configuration files (/etc/avocado or ~/.avocado/): This is configured by the user, on a more "permanent way".
3. Command-line options parsed in runtime. This is configured by the user, on a more "temporary way";

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.**ConfigOption**(*namespace*, *help_msg*, *key_type=<class 'str'>*, *default=None*, *parser=None*, *short_arg=None*, *long_arg=None*, *positional_arg=False*, *choices=None*, *nargs=None*, *metavar=None*, *required=None*, *action=None*, *argparse_type=None*)
Bases: object

action

add_argparser(*parser*, *long_arg*, *short_arg=None*, *positional_arg=False*, *choices=None*, *nargs=None*, *metavar=None*, *required=None*, *action=None*, *argparse_type=None*)
Add an command-line argparser to this option.

arg_parse_args

argparse_type

key

metavar

name_or_tags

section

set_value(*value*, *convert=False*)

value

exception avocado.core.settings.**DuplicatedNamespace**
Bases: *avocado.core.settings.SettingsError*

Raised when a namespace is already registered.

exception avocado.core.settings.**NamespaceNotRegistered**
Bases: *avocado.core.settings.SettingsError*

Raised when a namespace is not registered.

class avocado.core.settings.**Settings**
Bases: object

Settings is the Avocado configuration handler.

It is a simple wrapper around configparser and argparse.

Also, one object of this class could be passed as config to plugins and modules.

Basically, if you are going to have options (configuration options), either via config file or via command line, you should use this class. You don't need to instantiate a new settings, just import and use *register_option()*.

from avocado.core.settings import settings settings.register_option(...)

And when you needs get the current value, check on your configuration for the namespace (section.key) that you registered. i.e:

value = config.get('a.section.with.subsections.key')

Note: Please, do not use a default value when using *get()* here. If you are using an existing namespace, get will always return a value, either the default value, or the value set by the user.

Please, note that most of methods and attributes here are private. Only public methods and attributes should be used outside this module.

Constructor. Tries to find the main settings files and load them.

add_argparser_to_option(*namespace*, *parser*, *long_arg=None*, *short_arg=None*, *positional_arg=False*, *choices=None*, *nargs=None*, *metavar=None*, *required=None*, *action=None*, *allow_multiple=False*, *argparse_type=None*)
Add a command-line argument parser to an existing option.

This method is useful to add a parser when the option is registered without any command-line argument options. You should call the "register_option()" method for the namespace before calling this method.

Arguments

namespace [str] What is the namespace of the option (section.key)

parser [argparser parser] Since that you would like to have a command-line option, you should specify what is the parser or parser group that we should add this option.

long_arg: [str] A long option for the command-line. i.e: *–debug* for debug.

short_arg [str] A short option for the command-line. i.e: *-d* for debug.

positional_arg [bool] If this option is an positional argument or not. Default is *False*.

choices [tuple] If you would like to limit the option to a few choices. i.e: ('foo', 'bar')

nargs [int or str] The number of command-line arguments that should be consumed. Could be a int, '?', '*' or '+'. For more information visit the argparser documentation.

metavar [str] String presenting available sub-commands in help, if None we will use the section+key as metavar.

required [bool] If this is a required option or not when on command-line. Default is False.

action : The basic type of action to be taken when this argument is encountered at the command line. For more information visit the argparser documentation.

allow_multiple : Whether the same option may be available on different parsers. This is useful when the same option is available on different commands, such as "avocado run" or "avocado list".

argparse_type : A possibly different type for the command line handling of an option. For instance, when an option has a "key_type" of "list", its respective configuration file entry will expect a string that can be evaluated to a Python list, but that is far from convenient to set on the command line. With this argument, a function that will, for instance, split a comma separated list may be used, resulting in command line users being able to provide convenient input.

as_dict(*regex=None*)
Return an dictionary with the current active settings.

This will return a dict with all parsed options (either via config file or via command-line). If regex is not None, this method will filter the current config matching regex with the namespaces.

Parameters **regex** – A regular expression to be used on the filter.

as_full_dict()

as_json(*regex=None*)
Return a JSON with the current active settings.

This will return a JSON with all parsed options (either via config file or via command-line). If regex is not None, it will be used to filter namespaces.

Parameters **regex** – A regular expression to be used on the filter.

static filter_config(*config*, *regex*)
Utility to filter a config by namespaces based on a regex.

Parameters

- **config** – dict object with namespaces and values
- **regex** – regular expression to use against the namespace

merge_with_arguments(*arg_parse_config*)
Merge the current settings with the command-line args.

After parsing argument options this method should be executed to have an unified settings.

Parameters **arg_parse_config** – argparse.config dictionary with all command-line parsed arguments.

merge_with_configs()
Merge the current settings with the config file options.

After parsing config file options this method should be executed to have an unified settings.

process_config_path(*path*)
Update list of config paths and process the given path.

register_option(*section*, *key*, *default*, *help_msg*, *key_type=<class 'str'>*, *parser=None*, *positional_arg=False*, *short_arg=None*, *long_arg=None*, *choices=None*, *nargs=None*, *metavar=None*, *required=False*, *action=None*, *allow_multiple=False*)
Method used to register a configuration option inside Avocado.

This should be used to register a settings option (either config file option or command-line option). This is the central point that plugins and core should use to register a new configuration option.

This method will take care of the 'under the hood dirt', registering the configparse option and, if desired, the argparse too. Instead of using argparse and/or configparser, Avocado's contributors should use this method.

Using this method, you need to specify a "section", "key", "default" value and a "help_msg" always. This will create a relative configuration file option for you.

For instance:

settings.register_option(section='foo', key='bar', default='hello', help_msg='this is just a test')

This will register a 'foo.bar' namespace inside Avocado internals settings. And this could be now, be changed by the users or system configuration option:

[foo] bar = a different message replacing 'hello'

If you would like to provide also the flexibility to the user change the values via command-line, you should pass the other arguments.

Arguments

section [str] The configuration file section that your option should be present. You can specify subsections with dots. i.e: run.output.json

key [str] What is the key name of your option inside that section.

default [typeof(key_type)] The default value of an option. It sets the option value when the key is not defined in any configuration files or via command-line. The default value should be "processed". It means the value should match the type of key_type. Due to some internal limitations, the Settings module will not apply key_type to the default value.

help_msg [str] The help message that will be displayed at command-line (-h) and configuration file template.

key_type [any method] What is the type of your option? Currently supported: int, list, str or a custom method. Default is *str*.

parser [argparser parser] Since that you would like to have a command-line option, you should specify what is the parser or parser group that we should add this option.

positional_arg [bool] If this option is an positional argument or not. Default is *False*.

short_arg [str] A short option for the command-line. i.e: *-d* for debug.

long_arg: [str] A long option for the command-line. i.e: *–debug* for debug.

choices [tuple] If you would like to limit the option to a few choices. i.e: ('foo', 'bar')

nargs [int or str] The number of command-line arguments that should be consumed. Could be a int, '?', '*' or '+'. For more information visit the argparser documentation.

metavar [str] String presenting available sub-commands in help, if None we will use the section+key as metavar.

required [bool] If this is a required option or not when on command-line. Default is False.

action : The basic type of action to be taken when this argument is encountered at the command line. For more information visit the argparser documentation.

allow_multiple : Whether the same option may be available on different parsers. This is useful when the same option is available on different commands, such as "avocado run" or "avocado list".

Note: Most of the arguments here (like parser, positional_arg, short_arg, long_arg, choices, nargs, metavar, required and action) are only necessary if you would like to add a command-line option.

update_option(*namespace*, *value*, *convert=False*)
Convenient method to change the option's value.

This will update the value on Avocado internals and if necessary the type conversion will be realized.

For instance, if an option was registered as bool and you call:

settings.register_option(namespace='foo.bar', value='true', convert=True)

This will be stored as True, because Avocado will get the 'key_type' registered and apply here for the conversion.

This method is useful when getting values from config files where everything is stored as string and a conversion is needed.

Arguments

namespace [str] Your section plus your key, separated by dots. The last part of the namespace is your key. i.e: run.outputs.json.enabled (section is *run.outputs.json* and key is *enabled*)

value [any type] This is the new value to update.

convert [bool] If Avocado should try to convert the value and store it as the 'key_type' specified during the register. Default is False.

exception avocado.core.settings.**SettingsError**
Bases: Exception

Base settings error.

avocado.core.settings.**sorted_dict**(*dict_object*)

10.2.28 avocado.core.settings_dispatcher module

Settings Dispatcher

This is a special case for the dispatchers that can be found in *avocado.core.dispatcher*. This one deals with settings that will be read by the other dispatchers, while still being a dispatcher for configuration sources.

class avocado.core.settings_dispatcher.**SettingsDispatcher**
Bases: *avocado.core.extension_manager.ExtensionManager*

Dispatchers that allows plugins to modify settings

It's not the standard "avocado.core.dispatcher" because that one depends on settings. This dispatcher is the bare-stevedore dispatcher which is executed before settings is parsed.

10.2.29 avocado.core.streams module

avocado.core.streams.**BUILTIN_STREAMS = {'app': 'application output', 'debug': 'tracebacks**
Builtin special keywords to enable set of logging streams

avocado.core.streams.**BUILTIN_STREAM_SETS = {'all': 'all builtin streams', 'none': 'disabl**
Groups of builtin streams

10.2.30 avocado.core.suite module

class avocado.core.suite.**TestSuite**(*name*, *config=None*, *tests=None*, *job_config=None*, *resolutions=None*)

Bases: object

classmethod from_config(*config*, *name=None*, *job_config=None*)

Helper method to create a TestSuite from config dicts.

This is different from the TestSuite() initialization because here we are assuming that you need some help to build the test suite. Avocado will try to resolve tests based on the configuration information instead of assuming pre populated tests.

If you need to create a custom TestSuite, please use the TestSuite() constructor instead of this method.

Parameters

- **config** (*dict*) – A config dict to be used on the desired test suite.
- **name** (*str*) – The name of the test suite. This is optional and default is a random uuid.
- **job_config** (*dict*) – The job config dict (a global config). Use this to avoid huge configs per test suite. This is also optional.

references

run(*job*)

Run this test suite with the job context in mind.

Parameters **job** – A *avocado.core.job.Job* instance.

Return type set

runner

size

The overall length/size of this test suite.

stats

Return a statistics dict with the current tests.

status

tags_stats

Return a statistics dict with the current tests tags.

test_parameters

Placeholder for test parameters.

This is related to –test-parameters command line option or (run.test_parameters).

variants

exception avocado.core.suite.**TestSuiteError**

Bases: Exception

class avocado.core.suite.**TestSuiteStatus**

Bases: enum.Enum

An enumeration.

RESOLUTION_NOT_STARTED = <object object>

TESTS_FOUND = <object object>

TESTS_NOT_FOUND = <object object>

UNKNOWN = <object object>

avocado.core.suite.**resolutions_to_runnables**(*resolutions*, *config*)

Transforms resolver resolutions into runnables suitable for a suite

A resolver resolution (*avocado.core.resolver.ReferenceResolution*) contains information about the resolution process (if it was successful or not) and in case of successful resolutions a list of resolutions. It's expected that the resolution contain one or more *avocado.core.nrunner.Runnable*.

This function sets the runnable specific configuration for each runnable. It also performs tag based filtering on the runnables for possibly excluding some of the Runnables.

Parameters

- **resolutions** (list of *avocado.core.resolver.ReferenceResolution*) – possible multiple resolutions for multiple references
- **config** (*dict*) – job configuration

Returns the resolutions converted to runnables

Return type list of *avocado.core.nrunner.Runnable*

10.2.31 avocado.core.sysinfo module

class avocado.core.sysinfo.**SysInfo**(*basedir=None*, *log_packages=None*, *profiler=None*)

Bases: object

Log different system properties at some key control points.

Includes support for a start and stop event, with daemons running in between. An event may be a job, a test, or any other event with a beginning and end.

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** – Whether to use the profiler. If not given explicitly, tries to look in the config files.

end(*status=''*)

Logging hook called whenever a job finishes.

start()

Log all collectibles at the start of the event.

avocado.core.sysinfo.**collect_sysinfo**(*basedir*)

Collect sysinfo to a base directory.

avocado.core.sysinfo.**gather_collectibles_config**(*config*)

10.2.32 avocado.core.tags module

Test tags utilities module

avocado.core.tags.**filter_test_tags**(*test_suite*, *filter_by_tags*, *include_empty=False*, *include_empty_key=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
- **include_empty_key** (*bool*) – if true tests "keys" on key:val tags will be included in the filtered results

avocado.core.tags.**filter_test_tags_runnable**(*runnable*, *filter_by_tags*, *include_empty=False*, *include_empty_key=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
- **include_empty_key** (*bool*) – if true tests "keys" on key:val tags will be included in the filtered results

10.2.33 avocado.core.tapparser module

class avocado.core.tapparser.**TapParser**(*tap_io*)

Bases: object

class Bailout(*message*)

Bases: tuple

Create new instance of Bailout(message,)

message

Alias for field number 0

class Error(*message*)

Bases: tuple

Create new instance of Error(message,)

message

Alias for field number 0

class Plan(*count*, *late*, *skipped*, *explanation*)

Bases: tuple

Create new instance of Plan(count, late, skipped, explanation)

count
Alias for field number 0

explanation
Alias for field number 3

late
Alias for field number 1

skipped
Alias for field number 2

class Test(*number*, *name*, *result*, *explanation*)
Bases: tuple

Create new instance of Test(number, name, result, explanation)

explanation
Alias for field number 3

name
Alias for field number 1

number
Alias for field number 0

result
Alias for field number 2

class Version(*version*)
Bases: tuple

Create new instance of Version(version,)

version
Alias for field number 0

parse()

parse_test(*ok*, *num*, *name*, *directive*, *explanation*)

class avocado.core.tapparser.**TestResult**
Bases: enum.Enum

An enumeration.

FAIL = 'FAIL'

PASS = 'PASS'

SKIP = 'SKIP'

XFAIL = 'XFAIL'

XPASS = 'XPASS'

10.2.34 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

filename
Returns the name of the file (path) that holds the current test

setUp()
Hook method for setting up the test fixture before exercising it.

class avocado.core.test.**ExternalRunnerSpec**(*runner*, *chdir=None*, *test_dir=None*)
Bases: object

Defines the basic options used by ExternalRunner

class avocado.core.test.**ExternalRunnerTest**(*name*, *params=None*, *base_logdir=None*, *config=None*, *external_runner=None*, *external_runner_argument=None*)
Bases: *avocado.core.test.SimpleTest*

filename
Returns the name of the file (path) that holds the current test

test()
Run the test and postprocess the results

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()

class avocado.core.test.**PythonUnittest**(*name*, *params=None*, *base_logdir=None*, *config=None*, *test_dir=None*, *python_unittest_module=None*, *tags=None*)
Bases: *avocado.core.test.ExternalRunnerTest*

Python unittest test

test()
Run the test and postprocess the results

class avocado.core.test.**RawFileHandler**(*filename*, *mode='a'*, *encoding=None*, *delay=False*)
Bases: logging.FileHandler

File Handler that doesn't include arbitrary characters to the logged stream but still respects the formatter.

Open the specified file and use it as the stream for logging.

emit(*record*)
Modifying the original emit() to avoid including a new line in streams that should be logged in its purest form, like in stdout/stderr recordings.

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()

class avocado.core.test.**SimpleTest**(*name*, *params=None*, *base_logdir=None*, *config=None*, *executable=None*)

Bases: *avocado.core.test.Test*

Run an arbitrary command that returns either 0 (PASS) or !=0 (FAIL).

DATA_SOURCES = ['variant', 'file']

filename

Returns the name of the file (path) that holds the current test

test()

Run the test and postprocess the results

avocado.core.test.**TEST_STATE_ATTRIBUTES = ('name', 'logdir', 'logfile', 'status', 'running**

The list of test attributes that are used as the test state, which is given to the test runner via the queue they share

class avocado.core.test.**TapTest**(*name*, *params=None*, *base_logdir=None*, *config=None*, *executable=None*)

Bases: *avocado.core.test.SimpleTest*

Run a test command as a TAP test.

class avocado.core.test.**Test**(*methodName='test'*, *name=None*, *params=None*, *base_logdir=None*, *config=None*, *runner_queue=None*, *tags=None*)

Bases: unittest.case.TestCase, *avocado.core.test.TestData*

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.TestID) – 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 a temporary directory will be created.
- **config** (*dict*) – the job configuration, usually set by command line options and argument parsing

actual_time_end = -1

(unix) time when the test finished, actual one to be shown to users

actual_time_start = -1

(unix) time when the test started, actual one to be shown to users

basedir

The directory where this test (when backed by a file) is located at

cache_dirs

Returns a list of cache directories as set in config file.

static 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

Warning message This parameter will changed name to "msg" in the next LTS release because of lint W0221

static 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

Warning message This parameter will changed name to "msg" in the next LTS release because of lint W0221

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

Warning message This parameter will changed name to "msg" in the next LTS release because of lint W0221

fail_class

fail_reason

fetch_asset(*name*, *asset_hash=None*, *algorithm=None*, *locations=None*, *expire=None*, *find_only=False*, *cancel_on_missing=False*)
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 *avocado.utils.asset.DEFAULT_HASH_ALGORITHM*)
- **locations** – list of URLs from where the asset can be fetched (optional)
- **expire** – time for the asset to expire
- **find_only** – When *True*, *fetch_asset* only looks for the asset in the cache, avoiding the download/move action. Defaults to *False*.
- **cancel_on_missing** – whether the test should be canceled if the asset was not found in the cache or if *fetch* could not add the asset to the cache. Defaults to *False*.

Raises **OSError** – when it fails to fetch the asset or file is not in the cache and *cancel_on_missing* is *False*.

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

log
The enhanced test log

logdir
Path to this test's logging dir

logfile
Path to this test's main *debug.log* file

name
Returns the Test ID, which includes the test name

Return type *TestID*

outputdir
Directory available to test writers to attach files to the results

params
Parameters of this test (AvocadoParam instance)

phase
The current phase of the test execution

Possible (string) values are: INIT, SETUP, TEST, TEARDOWN and FINISHED

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

status
The result status of this test

tags
The tags associated with this test

tearDown()
Hook method for deconstructing the test fixture after testing it.

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, monotonic (could be forced from test)

time_start = -1
(unix) time when the test started, monotonic (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
This property returns a writable directory that exists during the entire test execution, but will be cleaned up once the test finishes.

It can be used on tasks such as decompressing source tarballs, building software, etc.

class avocado.core.test.**TestData**
Bases: object

Class that adds the ability for tests to have access to data files

Writers of new test types can change the completely change the behavior and still be compatible by providing an *DATA_SOURCES* attribute and a meth:*get_data* method.

DATA_SOURCES = ['variant', 'test', 'file']
Defines the name of data sources that this implementation makes available. Users may choose to pick data file from a specific source.

get_data(*filename*, *source=None*, *must_exist=True*)
Retrieves the path to a given data file.

This implementation looks for data file in one of the sources defined by the *DATA_SOURCES* attribute.

Parameters

- **filename** (*str*) – the name of the data file to be retrieved
- **source** (*str*) – one of the defined data sources. If not set, all of the *DATA_SOURCES* will be attempted in the order they are defined
- **must_exist** (*bool*) – whether the existence of a file is checked for

Return type str or None

class avocado.core.test.**TestError**(**args*, ***kwargs*)
Bases: *avocado.core.test.Test*

Generic test error.

test()

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()

10.2.35 avocado.core.test_id module

class avocado.core.test_id.**TestID**(*uid*, *name*, *variant=None*, *no_digits=None*)
Bases: object

Test ID construction and representation according to specification

This class wraps the representation of both Avocado's Test ID specification and Avocado's Test Name, which is part of a Test ID.

Constructs a TestID instance

Parameters

- **uid** – unique test id (within the job)
- **name** – test name, as returned by the Avocado test resolver (AKA as test loader)
- **variant** (*dict*) – the variant applied to this Test ID
- **no_digits** – number of digits of the test uid

classmethod from_identifier(*identifier*)
It wraps an identifier by the TestID class.

Parameters **identifier** – Any identifier that is guaranteed to be unique within the context of an avocado Job.

Returns TestID with *uid* as string representation of *identifier* and *name* "test".

Return type *avocado.core.test_id.TestID*

str_filesystem
Test ID in a format suitable for use in file systems

The string returned should be safe to be used as a file or directory name. This file system version of the test ID may have to shorten either the Test Name or the Variant ID.

The first component of a Test ID, the numeric unique test id, AKA "uid", will be used as a an stable identifier between the Test ID and the file or directory created based on the return value of this method. If the filesystem can not even represent the "uid", than an exception will be raised.

For Test ID "001-mytest;foo", examples of shortened file system versions include "001-mytest;f" or "001-myte;foo".

Raises RuntimeError if the test ID cannot be converted to a filesystem representation.

10.2.36 avocado.core.teststatus module

Valid test statuses and whether they signal success (or failure).

avocado.core.teststatus.**STATUSES = ['SKIP', 'ERROR', 'FAIL', 'WARN', 'PASS', 'INTERRUPTED',**
Valid test statuses, if a returned status is not listed here, it should be handled as error condition.

avocado.core.teststatus.**STATUSES_MAPPING = {'CANCEL': True, 'ERROR': False, 'FAIL': False,**
Maps the different status strings in avocado to booleans.

10.2.37 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*)
Add an element to a set.

This has no effect if the element is already present.

update(*items*)
Update a set with the union of itself and others.

class avocado.core.tree.**TreeEnvironment**
Bases: dict

TreeNode environment with values, origins and filters

copy() → a shallow copy of D

to_text(*sort=False*)
Human readable representation

Parameters **sort** – Sorted to provide stable output

Return type str

class avocado.core.tree.**TreeNode**(*name=''*, *value=None*, *parent=None*, *children=None*)
Bases: object

Class for bounding nodes into tree-structure.

Parameters

- **name** (*str*) – a name for this node that will be used to define its path according to the name of its parents
- **value** (*dict*) – a collection of keys and values that will be made into this node environment.
- **parent** (*TreeNode*) – the node that is directly above this one in the tree structure
- **children** (*builtin.list*) – the nodes that are directly beneath this one in the 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.**TreeNodeEnvOnly**(*path*, *environment=None*)
Bases: object

Minimal TreeNode-like class providing interface for AvocadoParams

Parameters

- **path** – Path of this node (must not end with '/')

- **environment** – List of pair/key/value items

fingerprint()

get_environment()

get_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

10.2.38 avocado.core.utils module

avocado.core.utils.**get_avocado_git_version**()

avocado.core.utils.**prepend_base_path**(*value*)

avocado.core.utils.**system_wide_or_base_path**(*file_path*)
Returns either a system wide path, or one relative to the base.

If "etc/avocado/avocado.conf" is given as input, it checks for the existence of "/etc/avocado/avocado.conf". If that path does not exist, then a path starting with the avocado's Python's distribution is returned. In that case it'd return something like "/usr/lib/python3.9/site-packages/avocado/etc/avocado/avocado.conf".

Parameters **file_path** (*str*) – a filesystem path that can either be absolute, or relative. If relative, the absolute equivalent (that is, by prefixing the filesystem root location) is checked for existence. If it does not exist, a path relative to the Python's distribution base path is returned.

Return type str

10.2.39 avocado.core.varianter module

Base classes for implementing the varianter interface

class avocado.core.varianter.**FakeVariantDispatcher**(*state*)
Bases: object

This object can act instead of VarianterDispatcher to report loaded variants.

map_method_with_return(*method*, **args*, ***kwargs*)
Reports list containing one result of map_method on self

to_str(*summary=0*, *variants=0*, ***kwargs*)

class avocado.core.varianter.**Varianter**(*debug=False*, *state=None*)
Bases: object

This object takes care of producing test variants

Parameters

- **debug** – Store whether this instance should debug varianter
- **state** – Force-varianter state

Note it's necessary to check whether variants debug is enable in order to provide the right results.

dump()
Dump the variants in loadable-state

This is lossy representation which takes all yielded variants and replaces the list of nodes with TreeNodeEnvOnly representations:

```
[{'path': path,
  'variant_id': variant_id,
  'variant': dump_tree_nodes(original_variant)},
 {'path': [str, str, ...],
  'variant_id': str,
  'variant': [(str, [(str, str, object), ...])],
 {'path': ['/run/*'],
  'variant_id': 'cat-26c0'
  'variant': [('/pig/cat',
               [('/pig', 'ant', 'fox'),
                ('/pig/cat', 'dog', 'bee')])]}
 ...]
```

where *dump_tree_nodes* looks like:

```
[(node.path, environment_representation),
 (node.path, [(path1, key1, value1), (path2, key2, value2), ...]),
 ('/pig/cat', [('/pig', 'ant', 'fox')])
```

Returns loadable Varianter representation

classmethod from_resultsdir(*resultsdir*)
Retrieves the job variants objects from the results directory.

This will return a list of variants since a Job can have multiple suites and the variants is per suite.

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)
- paths - default path(s)

:yield variant

load(*state*)
Load the variants state

Current implementation supports loading from a list of loadable variants. It replaces the VariantDispatcher with fake implementation which reports the loaded (and initialized) variants.

Parameters **state** – loadable Varianter representation

parse(*config*)
Apply options defined on the cmdline and initialize the plugins.

Parameters **config** (*dict*) – Configuration received from configuration files, command line parser, etc.

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.varianter.**dump_ivariants**(*ivariants*)
Walks the iterable variants and dumps them into json-serializable object

avocado.core.varianter.**dump_variant**(*variant*)
Dump a variant into a json-serializable representation

Parameters **variant** – Valid variant (list of TreeNode-like objects)

Returns json-serializable representation

avocado.core.varianter.**generate_variant_id**(*variant*)
Basic function to generate variant-id from a variant

Parameters **variant** – Avocado test variant (list of TreeNode-like objects)

Returns String compounded of ordered node names and a hash of all values.

avocado.core.varianter.**is_empty_variant**(*variant*)
Reports whether the variant contains any data

Parameters **variant** – Avocado test variant (list of TreeNode-like objects)

Returns True when the variant does not contain (any useful) data

avocado.core.varianter.**variant_to_str**(*variant*, *verbosity*, *out_args=None*, *debug=False*)
Reports human readable representation of a variant

Parameters

- **variant** – Valid variant (list of TreeNode-like objects)
- **verbosity** – Output verbosity where 0 means brief
- **out_args** – Extra output arguments (currently unused)
- **debug** – Whether the variant contains and should report debug info

Returns Human readable representation

10.2.40 avocado.core.version module

10.2.41 Module contents

avocado.core.**initialize_plugin_infrastructure**()

avocado.core.**initialize_plugins**()

avocado.core.**register_core_options**()

10.3 Utilities APIs

Avocado gives to you more than 40 python utility libraries (so far), that can be found under the *avocado.utils*. You can use these libraries to avoid having to write necessary routines for your tests. 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.

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.

10.3.1 Subpackages

avocado.utils.external package

Submodules

avocado.utils.external.gdbmi_parser module

class avocado.utils.external.gdbmi_parser.**AST**(*ast_type*)
Bases: object

class avocado.utils.external.gdbmi_parser.**GdbDynamicObject**(*dict_*)
Bases: object

graft(*dict_*)

class avocado.utils.external.gdbmi_parser.**GdbMiInterpreter**(*ast*)
Bases: *avocado.utils.external.spark.GenericASTTraversal*

static n_list(*node*)

static n_record_list(*node*)

static n_result(*node*)

n_result_header(*node*)

static n_result_list(*node*)

static n_result_record(*node*)

n_stream_record(*node*)

static n_tuple(*node*)

static n_value_list(*node*)

class avocado.utils.external.gdbmi_parser.**GdbMiParser**
Bases: *avocado.utils.external.spark.GenericASTBuilder*

error(*token*, *i=0*, *tokens=None*)

nonterminal(*token_type*, *args*)

p_output(*args*)
output ::= record_list record_list ::= generic_record record_list ::= generic_record record_list generic_record ::= result_record generic_record ::= stream_record result_record ::= result_header result_list nl result_record ::= result_header nl result_header ::= token result_type class result_header ::= result_type class result_header ::= token = class result_header ::= = class stream_record ::= stream_type c_string nl result_list ::= , result result_list result_list ::= , result result_list ::= , tuple result ::= variable = value class ::= string variable ::= string value ::= const value ::= tuple value ::= list value_list ::= , value value_list ::= , value value_list const ::= c_string tuple ::= { } tuple ::= { result } tuple ::= { result result_list } list ::= [ ] list ::= [value] list ::= [value value_list] list ::= [result] list ::= [result result_list] list ::= { value } list ::= { value value_list }

terminal(*token*)

class avocado.utils.external.gdbmi_parser.**GdbMiRecord**(*record*)
Bases: object

class avocado.utils.external.gdbmi_parser.**GdbMiScanner**(*flags=0*)
Bases: *avocado.utils.external.gdbmi_parser.GdbMiScannerBase*

t_token(*s*)
d+

class avocado.utils.external.gdbmi_parser.**GdbMiScannerBase**(*flags=0*)
Bases: *avocado.utils.external.spark.GenericScanner*

t_c_string(*s*)
".*?(?<![\])"

t_default(*s*)
(. | n)+

t_nl(*s*)
n|rn

t_result_type(*s*)
*|+|^

t_stream_type(*s*)
@|&|~

t_string(*s*)
[w-]+

t_symbol(*s*)
,|{|\}|[|\]|=

t_whitespace(*s*)
[tfv]+

tokenize(*data_input*)

class avocado.utils.external.gdbmi_parser.**Token**(*token_type*, *value=None*)
Bases: object

class avocado.utils.external.gdbmi_parser.**session**
Bases: object

parse(*tokens*)

process(*data_input*)

scan(*data_input*)

avocado.utils.external.spark module

class avocado.utils.external.spark.**GenericASTBuilder**(*AST*, *start*)
Bases: *avocado.utils.external.spark.GenericParser*

buildASTNode(*args*, *lhs*)

nonterminal(*token_type*, *args*)

preprocess(*rule*, *func*)

static terminal(*token*)

class avocado.utils.external.spark.**GenericASTMatcher**(*start*, *ast*)
Bases: *avocado.utils.external.spark.GenericParser*

static foundMatch(*args*, *func*)

match(*ast=None*)

match_r(*node*)

preprocess(*rule*, *func*)

resolve(*input_list*)

class avocado.utils.external.spark.**GenericASTTraversal**(*ast*)
Bases: object

default(*node*)

postorder(*node=None*)

preorder(*node=None*)

static prune()

static typestring(*node*)

exception avocado.utils.external.spark.**GenericASTTraversalPruningException**
Bases: Exception

class avocado.utils.external.spark.**GenericParser**(*start*)
Bases: object

add(*input_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*)

static 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*)

static preprocess(*rule*, *func*)

static resolve(*input_list*)

skip(*hs*, *pos=0*)

static typestring(*token*)

class avocado.utils.external.spark.**GenericScanner**(*flags=0*)

Bases: object

static error(*s*, *pos*)

makeRE(*name*)

reflect()

static t_default(*s*)

(. | n)+

tokenize(*s*)

Module contents

avocado.utils.network package

Submodules

avocado.utils.network.common module

avocado.utils.network.common.**run_command**(*command*, *host*, *sudo=False*)

avocado.utils.network.exceptions module

exception avocado.utils.network.exceptions.**NWException**

Bases: Exception

Base Exception Class for all exceptions

avocado.utils.network.hosts module

This module provides an useful API for hosts in a network.

class avocado.utils.network.hosts.**Host**(*host*)
Bases: object

This class represents a base Host and shouldn't be instantiated.

Use one of the child classes (LocalHost or RemoteHost).

During the initialization of a child, all interfaces will be detected and available via *interfaces* attribute. This could be accessed on LocalHost and RemoteHost instances.

So, for instance, you could have a local and a remote host:

```
remote = RemoteHost(host='foo', port=22,
                    username='foo', password='bar')
local = LocalHost()
```

You can iterate over the network interfaces of any host:

```
for i in remote.interfaces:
    print(i.name, i.is_link_up())
```

get_default_route_interface()
Get a list of default routes interfaces

Returns list of interface names

get_interface_by_ipaddr(*ipaddr*)
Return an interface that has a specific ipaddr.

interfaces

class avocado.utils.network.hosts.**LocalHost**(*host='localhost'*)
Bases: *avocado.utils.network.hosts.Host*

This class represents a local host and inherit from *Host*.

You should use this class when trying to get information about your localhost.

Example:

local = LocalHost()

class avocado.utils.network.hosts.**RemoteHost**(*host*, *username*, *port=22*, *key=None*, *password=None*)
Bases: *avocado.utils.network.hosts.Host*

This class represents a remote host and inherit from *Host*.

You must provide at least an username to establish a connection.

Example with password:

remote = RemoteHost(host='192.168.0.1', port=22, username='foo', password='bar')

You can also provide a key instead of a password.

avocado.utils.network.interfaces module

class avocado.utils.network.interfaces.**NetworkInterface**(*if_name*, *host*, *if_type='Ethernet'*)

Bases: object

This class represents a network card interface (NIC).

An "NetworkInterface" is attached to some host. This could be an instance of LocalHost or RemoteHost. If a RemoteHost then all commands will be executed on a remote_session (host.remote_session). Otherwise will be executed locally.

Here you will find a few methods to perform basic operations on a NIC.

add_ipaddr(*ipaddr*, *netmask*)
Add an IP Address (with netmask) to the interface.

This method will try to add a new ipaddr/netmask this interface, if fails it will raise a NWException.

You must have sudo permissions to run this method on a host.

Parameters

- **ipaddr** – IP Address
- **netmask** – Network mask

add_vlan_tag(*vlan_num*, *vlan_name=None*)
Configure 802.1Q VLAN tagging to the interface.

This method will attempt to add a VLAN tag to this interface. If it fails, the method will raise a NWException.

Parameters

- **vlan_num** – VLAN ID
- **vlan_name** – option to name VLAN interface, by default it is named <interface_name>.<vlan_num>

bring_down()
Shutdown the interface.

This will shutdown the interface link. Be careful, you might lost connection to the host.

You must have sudo permissions to run this method on a host.

bring_up()
"Wake-up the interface.

This will wake-up the interface link.

You must have sudo permissions to run this method on a host.

config_filename

get_hwaddr()
Get the Hardware Address (MAC) of this interface.

This method will try to get the address and if fails it will raise a NWException.

get_ipaddrs(*version=4*)
Get the IP addresses from a network interface.

Interfaces can hold multiple IP addresses. This method will return a list with all addresses on this interface.

Parameters **version** – Address Family Version (4 or 6). This must be a integer and default is 4.

Returns IP address as string.

get_link_state()
Method used to get the current link state of this interface.

This method will return 'up', 'down' or 'unknown', based on the network interface state. Or it will raise a NWException if is unable to get the interface state.

get_mtu()
Return the current MTU value of this interface.

This method will try to get the current MTU value, if fails will raise a NWException.

is_admin_link_up()
Check the admin link state is up or not.

Returns True or False, True if network interface state is 'UP' otherwise will return False.

is_available()
Check if interface is available.

This method checks if the interface is available.

rtype: bool

is_link_up()
Check if the interface is up or not.

Returns True or False. True if admin link state and operational link state is up otherwise will return False.

is_operational_link_up()
Check Operational link state is up or not.

Returns True or False. True if operational link state is LOWER_UP, otherwise will return False.

ping_check(*peer_ip*, *count=2*, *options=None*)
This method will try to ping a peer address (IPv4 or IPv6).

You should provide a IPv4 or IPV6 that would like to ping. This method will try to ping the peer and if fails it will raise a NWException.

Parameters

- **peer_ip** – Peer IP address (IPv4 or IPv6)
- **count** – How many packets to send. Default is 2
- **options** – ping command options. Default is None

remove_all_vlans()
Remove all VLANs of this interface.

This method will remove all the VLAN interfaces associated by the interface. If it fails, the method will raise a NWException.

remove_cfg_file()
Remove any config files that is created as a part of the test

remove_ipaddr(*ipaddr*, *netmask*)
Removes an IP address from this interface.

This method will try to remove the address from this interface and if fails it will raise a NWException. Be careful, you can lost connection.

You must have sudo permissions to run this method on a host.

remove_link()
Deletes virtual interface link.

This method will try to delete the virtual device link and the interface will no more be listed with 'ip a' and if fails it will raise a NWException. Be careful, you can lost connection.

You must have sudo permissions to run this method on a host.

remove_vlan_by_tag(*vlan_num*)
Remove the VLAN of the interface by tag number.

This method will try to remove the VLAN tag of this interface. If it fails, the method will raise a NWException.

Parameters **vlan_num** – VLAN ID

Returns True or False, True if it found the VLAN interface and removed it successfully, otherwise it will return False.

restore_from_backup()
Revert interface file from backup.

This method checks if a backup version is available for given interface then it copies backup file to interface file in /sysfs path.

save(*ipaddr*, *netmask*)
Save current interface IP Address to the system configuration file.

If the ipaddr is valid (currently being used by the interface) this will try to save the current settings into /etc/. This check is necessary to avoid inconsistency. Before save, you should add_ipaddr, first.

Currently, only RHEL, Fedora and SuSE are supported. And this will create a backup file of your current configuration if found.

:param ipaddr : IP Address which need to configure for interface :param netmask: Network mask which is associated to the provided IP

set_hwaddr(*hwaddr*)
Sets a Hardware Address (MAC Address) to the interface.

This method will try to set a new hwaddr to this interface, if fails it will raise a NWException.

You must have sudo permissions to run this method on a host.

Parameters **hwaddr** – Hardware Address (Mac Address)

set_mtu(*mtu*, *timeout=30*)
Sets a new MTU value to this interface.

This method will try to set a new MTU value to this interface, if fails it will raise a NWException. Also it will wait until the Interface is up before returning or until timeout be reached.

You must have sudo permissions to run this method on a host.

Parameters

- **mtu** – mtu size that need to be set. This must be an int.
- **timeout** – how many seconds to wait until the interface is up again. Default is 30.

vlans
Return all interface's VLAN.

This is a dict were key is the VLAN number and the value is the name of the VLAN interface.

rtype: dict

avocado.utils.network.ports module

Module with network related utility functions

avocado.utils.network.ports.**FAMILIES = (<AddressFamily.AF_INET: 2>, <AddressFamily.AF_INET**
Families taken into account in this class

avocado.utils.network.ports.**PROTOCOLS = (<SocketKind.SOCK_STREAM: 1>, <SocketKind.SOCK_DGRA**
Protocols taken into account in this class

class avocado.utils.network.ports.**PortTracker**
Bases: *avocado.utils.data_structures.Borg*

Tracks ports used in the host machine.

find_free_port(*start_port=None*)

register_port(*port*)

release_port(*port*)

avocado.utils.network.ports.**find_free_port**(*start_port=1024*, *end_port=65535*, *address='localhost'*, *sequent=False*)
Return a host free port in the range [start_port, end_port].

Parameters

- **start_port** – header of candidate port range, defaults to 1024
- **end_port** – ender of candidate port range, defaults to 65535
- **address** – Socket address to bind or connect
- **sequent** – Find port sequentially, random order if it's False

Return type int or None if no free port found

avocado.utils.network.ports.**find_free_ports**(*start_port*, *end_port*, *count*, *address='localhost'*, *sequent=False*)
Return count of host free ports in the range [start_port, end_port].

Parameters

- **start_port** – header of candidate port range
- **end_port** – ender of candidate port range
- **count** – Initial number of ports known to be free in the range.
- **address** – Socket address to bind or connect
- **sequent** – Find port sequentially, random order if it's False

avocado.utils.network.ports.**is_port_free**(*port*, *address*)
Return True if the given port is available for use.

Currently we only check for TCP/UDP connections on IPv4/6

Parameters

- **port** – Port number
- **address** – Socket address to bind or connect

Module contents

avocado.utils.software_manager package

Subpackages

avocado.utils.software_manager.backends package

Submodules

avocado.utils.software_manager.backends.apt module

class avocado.utils.software_manager.backends.apt.**AptBackend**
Bases: *avocado.utils.software_manager.backends.dpkg.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'

build_dep(*name*)
Installed build-dependencies of a given package [name].

Parameters **name** – parameter package to install build-dependencies for.

Return True If packages are installed properly

get_source(*name*, *path*)
Download source for provided package. Returns the path with source placed.

Parameters **name** – parameter wildcard package to get the source for

Return path path of ready-to-build source

install(*name*)
Installs package [name].

Parameters **name** – Package name.

provides(*name*)
Return a list of packages that provide [name of package/file].

Parameters **name** – File name.

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

avocado.utils.software_manager.backends.base module

class avocado.utils.software_manager.backends.base.**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.

avocado.utils.software_manager.backends.dnf module

class avocado.utils.software_manager.backends.dnf.**DnfBackend**
Bases: *avocado.utils.software_manager.backends.yum.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.

build_dep(*name*)
Install build-dependencies for package [name]

Parameters **name** – name of the package

Return True If build dependencies are installed properly

avocado.utils.software_manager.backends.dpkg module

class avocado.utils.software_manager.backends.dpkg.**DpkgBackend**
Bases: *avocado.utils.software_manager.backends.base.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*)

static extract_from_package(*package_path*, *dest_path=None*)
Extracts the package content to a specific destination path.

Parameters

- **package_path** (*str*) – path to the deb package.
- **dest_path** – destination path to extract the files. Default is the current directory.

Returns path of the extracted file

Returns the path of the extracted files.

Return type str

static is_valid(*package_path*)
Verifies if a package is a valid deb file.

Parameters **package_path** (*str*) – .deb package path.

Returns True if valid, otherwise false.

Return type bool

static 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.

avocado.utils.software_manager.backends.rpm module

class avocado.utils.software_manager.backends.rpm.**RpmBackend**
Bases: *avocado.utils.software_manager.backends.base.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.

static extract_from_package(*package_path*, *dest_path=None*)
Extracts the package content to a specific destination path.

Parameters

- **package_path** (*str*) – path to the rpm package.
- **dest_path** – destination path to extract the files. Default it will be the current directory.

Returns path of the extracted file

Returns the path of the extracted files.

Return type str

find_rpm_packages(*rpm_dir*)
Extract product dependencies from a defined RPM directory and all its subdirectories.

Parameters **rpm_dir** (*str*) – directory to search in

Returns found RPM packages

Return type [str]

static is_valid(*package_path*)
Verifies if a package is a valid rpm file.

Parameters **package_path** (*str*) – .rpm package path.

Returns True if valid, otherwise false.

Return type bool

list_all(*software_components=True*)
List all installed packages.

Parameters **software_components** – log in a format suitable for the SoftwareComponent schema

static list_files(*name*)
List files installed on the system by package [name].

Parameters **name** – Package name.

perform_setup(*packages*, *no_dependencies=False*)
General RPM setup with automatic handling of dependencies based on install attempts.

Parameters **packages** (*[str]*) – the RPM packages to install in dependency-friendly order

Returns whether setup completed successfully

Return type bool

static prepare_source(*spec_file*, *dest_path=None*)
Rpmbuild the spec path and return build dir

Parameters **spec_path** – spec path to install

Return path build directory

static rpm_erase(*package_name*)
Erase an RPM package.

Parameters **package_name** (*str*) – name of the erased package

Returns whether file is erased properly

Return type bool

static rpm_install(*file_path*, *no_dependencies=False*, *replace=False*)
Install the rpm file [file_path] provided.

Parameters

- **file_path** (*str*) – file path of the installed package
- **no_dependencies** (*bool*) – whether to add “nodeps” flag
- **replace** (*bool*) – whether to replace existing package

Returns whether file is installed properly

Return type bool

static rpm_verify(*package_name*)
Verify an RPM package with an installed one.

Parameters **package_name** (*str*) – name of the verified package

Returns whether the verification was successful

Return type bool

avocado.utils.software_manager.backends.yum module

class avocado.utils.software_manager.backends.yum.**YumBackend**(*cmd='yum'*)
Bases: *avocado.utils.software_manager.backends.rpm.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.

REPO_FILE_PATH = '/etc/yum.repos.d/avocado-managed.repo'
Path to the repository managed by Avocado

add_repo(*url*)
Adds package repository located on [url].

Parameters **url** – Universal Resource Locator of the repository.

static build_dep(*name*)
Install build-dependencies for package [name]

Parameters **name** – name of the package

Return True If build dependencies are installed properly

get_source(*name*, *dest_path*)
Downloads the source package and prepares it in the given dest_path to be ready to build.

Parameters

- **name** – name of the package
- **dest_path** – destination_path

Return final_dir path of ready-to-build directory

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.

repo_config_parser

upgrade(*name=None*)
Upgrade all available packages.

Optionally, upgrade individual packages.

Parameters **name** (*str*) – optional parameter wildcard spec to upgrade

yum_base

avocado.utils.software_manager.backends.zypper module

class avocado.utils.software_manager.backends.zypper.**ZypperBackend**
Bases: *avocado.utils.software_manager.backends.rpm.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.

build_dep(*name*)
Return True if build-dependencies are installed for provided package

Keyword argument: name – name of the package

get_source(*name*, *dest_path*)
Downloads the source package and prepares it in the given dest_path to be ready to build

Parameters

- **name** – name of the package
- **dest_path** – destination_path

Return final_dir path of ready-to-build directory

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

Module contents

Submodules

avocado.utils.software_manager.distro_packages module

avocado.utils.software_manager.distro_packages.**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.inspector module

avocado.utils.software_manager.inspector.**SUPPORTED_PACKAGE_MANAGERS = {'apt-get': <class**
Mapping of package manager name to implementation class.

class avocado.utils.software_manager.inspector.**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.

avocado.utils.software_manager.main module

avocado.utils.software_manager.main.**main**()

avocado.utils.software_manager.manager module

class avocado.utils.software_manager.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

static extract_from_package(*package_path*, *dest_path=None*)
Try to extract a package content into a destination directory.

It will try to see if the package is valid against all supported package managers and if any is found, then extracts its content into the extract_path.

Raises NotImplementedError when a non-supported package is used.

Parameters

- **package_path** (*str*) – package file path.
- **dest_path** (*str*) – destination path to extract. Default is the current directory.

Returns destination path were the package it was extracted.

is_capable()
Checks if environment is capable by initializing the backend.

Module contents

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).

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

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

static extract_from_package(*package_path*, *dest_path=None*)
Try to extract a package content into a destination directory.

It will try to see if the package is valid against all supported package managers and if any is found, then extracts its content into the extract_path.

Raises NotImplementedError when a non-supported package is used.

Parameters

- **package_path** (*str*) – package file path.
- **dest_path** (*str*) – destination path to extract. Default is the current directory.

Returns destination path were the package it was extracted.

is_capable()
Checks if environment is capable by initializing the backend.

10.3.2 Submodules

10.3.3 avocado.utils.ar module

Module to read UNIX ar files

class avocado.utils.ar.**Ar**(*path*)
Bases: object

An UNIX ar archive.

is_valid()
Checks if a file looks like an AR archive.

Parameters **path** – path to a file

Returns bool

list()
Return the name of the members in the archive.

read_member(*identifier*)
Returns the data for the given member identifier.

class avocado.utils.ar.**ArMember**(*identifier*, *size*, *offset*)
Bases: object

A member of an UNIX ar archive.

avocado.utils.ar.**FILE_HEADER_FMT = '16s12s6s6s8s10s2c'**
The header for each file in the archive

avocado.utils.ar.**MAGIC = b'!<arch>\n'**
The first eight bytes of all AR archives

10.3.4 avocado.utils.archive module

Module to help extract and create compressed archives.

exception avocado.utils.archive.**ArchiveException**
Bases: 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.

Returns the first member of the archive, a file or directory or None if the archive is empty

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.**GZIP_MAGIC = b'\x1f\x8b'**
The first two bytes that all gzip files start with

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.**gzip_uncompress**(*path*, *output_path*)
Uncompress a gzipped file at path, to either a file or dir at output_path

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.**is_gzip_file**(*path*)
Checks if file given by path has contents that suggests gzip file

avocado.utils.archive.**is_lzma_file**(*path*)
Checks if file given by path has contents that suggests lzma file

avocado.utils.archive.**lzma_uncompress**(*path*, *output_path=None*, *force=False*)
Extracts a XZ compressed file to the same directory.

avocado.utils.archive.**uncompress**(*filename*, *path*)
Extract files from an archive.

Parameters

- **filename** – archive file name.
- **path** – destination path to extract to.

10.3.5 avocado.utils.asset module

Asset fetcher from multiple locations

class avocado.utils.asset.**Asset**(*name=None*, *asset_hash=None*, *algorithm=None*, *locations=None*, *cache_dirs=None*, *expire=None*, *metadata=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. Default is ‘’.
- **asset_hash** – asset hash
- **algorithm** – hash algorithm
- **locations** – location(s) where the asset can be fetched from
- **cache_dirs** – list of cache directories
- **expire** – time in seconds for the asset to expire
- **metadata** – metadata which will be saved inside metadata file

asset_name

fetch(*timeout=None*)
Try to fetch the current asset.

First tries to find the asset on the provided cache_dirs list. Then tries to download the asset from the locations list provided.

Parameters **timeout** – timeout in seconds. Default is *avocado.utils.asset.DOWNLOAD_TIMEOUT*.

Raises **OSError** – When it fails to fetch the asset

Returns The path for the file on the cache directory.

Return type str

find_asset_file(*create_metadata=False*)
Search for the asset file in each one of the cache locations

Parameters **create_metadata** (*bool*) – Should this method create the metadata in case asset file found and metadata is not found? Default is False.

Returns asset path, if it exists in the cache

Return type str

Raises OSError

classmethod get_all_assets(*cache_dirs*, *sort=True*)
Returns all assets stored in all cache dirs.

classmethod get_asset_by_name(*name*, *cache_dirs*, *expire=None*, *asset_hash=None*)
This method will return a cached asset based on name if exists.

You don't have to instantiate an object of Asset class. Just use this method.

To be improved soon: cache_dirs should be not necessary.

Parameters

- **name** – the asset filename used during registration.
- **cache_dirs** – list of directories to use during the search.
- **expire** – time in seconds for the asset to expire. Expired assets will not be returned.
- **asset_hash** – asset hash.

Returns asset path, if it exists in the cache.

Return type str

Raises OSError

classmethod get_assets_by_size(*size_filter*, *cache_dirs*)
Return a list of all assets in cache based on its size in MB.

Parameters

- **size_filter** – a string with a filter (comparison operator + value). Ex ">20", "<=200". Supported operators: ==, <, >, <=, >=.
- **cache_dirs** – list of directories to use during the search.

classmethod get_assets_unused_for_days(*days*, *cache_dirs*)
Return a list of all assets in cache based on the access time.

This will check if the file's data wasn't modified N days ago.

Parameters

- **days** – how many days ago will be the threshold. Ex: "10" will return the assets files that *was not* accessed during the last 10 days.
- **cache_dirs** – list of directories to use during the search.

get_metadata()
Returns metadata of the asset if it exists or None.

Returns metadata

Return type dict or None

name_scheme
This property will return the scheme part of the name if is an URL.

Otherwise, will return None.

name_url
This property will return the full url of the name if is an URL.

Otherwise, will return None.

static parse_name(*name*)
Returns a ParseResult object for the given name.

parsed_name
Returns a ParseResult object for the currently set name.

classmethod read_hash_from_file(*filename*)
Read the CHECKSUM file and return the hash.

This method raises a FileNotFoundError if file is missing and assumes that filename is the CHECKSUM filename.

Return type list with algorithm and hash

relative_dir

classmethod remove_asset_by_path(*asset_path*)
Remove an asset and its checksum.

To be fixed: Due the current implementation limitation, this method will not remove the metadata to avoid removing other asset metadata.

Parameters **asset_path** – full path of the asset file.

classmethod remove_assets_by_overall_limit(*limit*, *cache_dirs*)
This will remove assets based on overall limit.

We are going to sort the assets based on the access time first. For instance it may be the case that a GitLab cache limit is 4 GiB, in that case we can sort by last access, and remove all that exceeds 4 GiB (that is, keep the last accessed 4 GiB worth of cached files).

Note: during the usage of this method, you should use bytes as limit.

Parameters

- **limit** – a integer limit in bytes.
- **cache_dirs** – list of directories to use during the search.

classmethod remove_assets_by_size(*size_filter*, *cache_dirs*)

classmethod remove_assets_by_unused_for_days(*days*, *cache_dirs*)

urls
Complete list of locations including name if is an URL.

avocado.utils.asset.**DEFAULT_HASH_ALGORITHM = 'sha1'**
The default hash algorithm to use on asset cache operations

avocado.utils.asset.**DOWNLOAD_TIMEOUT = 300**
The default timeout for the downloading of assets

exception avocado.utils.asset.**UnsupportedProtocolError**
Bases: OSError

Signals that the protocol of the asset URL is not supported

10.3.6 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.**ENCODING = 'UTF-8'**
On import evaluated value representing the system encoding based on system locales using locale.getpreferredencoding(). Use this value wisely as some files are dumped in different encoding.

avocado.utils.astring.**FS_UNSAFE_CHARS = '<>:"/\\|?*;'**
String containing all fs-unfriendly chars (Windows-fat/Linux-ext3)

avocado.utils.astring.**bitlist_to_string**(*data*)
Transform from bit list to ASCII string.

Parameters **data** – Bit list to be transformed

avocado.utils.astring.**is_bytes**(*data*)
Checks if the data given is a sequence of bytes

And not a "text" type, that can be of multi-byte characters. Also, this does NOT mean a bytearray type.

Parameters **data** – the instance to be checked if it falls under the definition of an array of bytes.

avocado.utils.astring.**is_text**(*data*)
Checks if the data given is a suitable for holding text

That is, if it can hold text that requires more than one byte for each character.

avocado.utils.astring.**iter_tabular_output**(*matrix*, *header=None*, *strip=False*)
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.
- **strip** – Optionally remove trailing whitespace from each row.

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**(*input_str*)
People tend to mix unicode streams with encoded strings. This function tries to replace any input with a valid utf-8 encoded ascii stream.

On Python 3, it's a terrible idea to try to mess with encoding, so this function is limited to converting other types into strings, such as numeric values that are often the members of a matrix.

Parameters **input_str** – possibly unsafe string or other object that can be turned into a string

Returns a 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**(*input_str*)
Convert string to a valid file/dir name.

This takes a string that may contain characters that are not allowed on FAT (Windows) filesystems and/or ext3 (Linux) filesystems, and replaces them for safe (boring) underlines.

It limits the size of the path to be under 255 chars, and make hidden paths (starting with ".") non-hidden by making them start with "_".

Parameters **input_str** – String to be converted

Returns 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*, *strip=False*)
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.

- **strip** – Optionally remove trailing whitespace from each row.

Returns String with the tabular output, lines separated by unix line feeds.

Return type str

avocado.utils.astring.**to_text**(*data*, *encoding='UTF-8'*, *errors='strict'*)

Convert anything to text decoded text

When the data is bytes, it's decoded. When it's not of string types it's re-formatted into text and returned. Otherwise (it's string) it's returned unchanged.

Parameters

- **data** (*either bytes or other data that will be returned unchanged*) – data to be transformed into text
- **encoding** – encoding of the data (only used when decoding is necessary)
- **errors** – how to handle encode/decode errors, see: https://docs.python.org/3/library/codecs.html#error-handlers

10.3.7 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.

10.3.8 avocado.utils.build module

avocado.utils.build.**configure**(*path*, *configure=None*)

Configures the source tree for a subsequent build

Most source directories coming from official released tarballs will have a "configure" script, but source code snapshots may have "autogen.sh" instead (which usually creates and runs a "configure" script itself). This function will attempt to run the first one found (if a configure script name not given explicitly).

Parameters **configure** (*str or None*) – the name of the configure script (None for trying to find one automatically)

Returns the configure script exit status, or None if no script was found and executed

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'*, *extra_args=''*, *process_kwargs=None*)
Run make, adding MAKEOPTS to the list of options.

Parameters

- **path** – directory from where to run make
- **make** – what make command name to use.
- **extra_args** – extra command line arguments to pass to make.
- **process_kwargs** – Additional key word arguments to the underlying process running the make.

Returns the make command result object

10.3.9 avocado.utils.cloudinit module

cloudinit configuration support

This module can be easily used with *avocado.utils.vmimage*, to configure operating system images via the cloudinit tooling.

Please, keep in mind that if you would like to create/write in ISO images, you need pycdlib module installed in your environment.

see http://cloudinit.readthedocs.io.

avocado.utils.cloudinit.**AUTHORIZED_KEY_TEMPLATE = '\nssh_authorized_keys:\n - {0}\n'**
An authorized key configuration for the default user

Positional template variables are: ssh_authorized_keys

avocado.utils.cloudinit.**METADATA_TEMPLATE = 'instance-id: {0}\nhostname: {1}\n'**
The meta-data file template

Positional template variables are: instance-id, hostname

avocado.utils.cloudinit.**PASSWORD_TEMPLATE = '\npassword: {0}\nchpasswd:\n expire: False\n**
A username configuration as per cloudinit/config/cc_set_passwords.py

Positional template variables are: password

avocado.utils.cloudinit.**PHONE_HOME_TEMPLATE = '\nphone_home:\n url: http://{0}:{1}/$INSTA**
A phone home configuration that will post just the instance id

Positional template variables are: address, port

class avocado.utils.cloudinit.**PhoneHomeServer**(*address*, *instance_id*)
Bases: http.server.HTTPServer

Implements the phone home HTTP server.

Wait the phone home from a given instance.

Initialize the server.

Parameters

- **address** (*tuple*) – a hostname or IP address and port, in the same format given to socket and other servers
- **instance_id** (*str*) – the identification for the instance that should be calling back, and the condition for the wait to end

class avocado.utils.cloudinit.**PhoneHomeServerHandler**(*request*, *client_address*, *server*)

Bases: http.server.BaseHTTPRequestHandler

Handles HTTP requests to the phone home server.

do_POST()

Handles an HTTP POST request.

Respond with status 200 if the instance phoned back.

log_message(*format_*, **args*)

Logs an arbitrary message.

Note It currently disables any message logging.

avocado.utils.cloudinit.**USERDATA_HEADER = '#cloud-config'**

The header expected to be found at the beginning of the user-data file

avocado.utils.cloudinit.**USERNAME_TEMPLATE = '\nssh_pwauth: True\n\nsystem_info:\n default_**

A username configuration as per cloudinit/config/cc_set_passwords.py

Positional template variables : username

avocado.utils.cloudinit.**iso**(*output_path*, *instance_id*, *username=None*, *password=None*, *phone_home_host=None*, *phone_home_port=None*, *authorized_key=None*)

Generates an ISO image with cloudinit configuration

The content always include the cloudinit metadata, and optionally the userdata content. On the userdata file, it may contain a username/password section (if both parameters are given) and/or a phone home section (if both host and port are given).

Parameters

- **output_path** – the location of the resulting (to be created) ISO image containing the cloudinit configuration
- **instance_id** – the ID of the cloud instance, a form of identification for the dynamically created executing instances
- **username** – the username to be used when logging interactively on the instance
- **password** – the password to be used along with username when authenticating with the login services on the instance
- **phone_home_host** – the address of the host the instance should contact once it has finished booting
- **phone_home_port** – the port acting as an HTTP phone home server that the instance should contact once it has finished booting
- **authorized_key** (*str*) – a SSH public key to be added as an authorized key for the default user, similar to "ssh-rsa ..."

Raises RuntimeError if the system can not create ISO images. On such a case, user is expected to install supporting packages, such as pycdlib.

avocado.utils.cloudinit.**wait_for_phone_home**(*address*, *instance_id*)
Sets up a phone home server and waits for the given instance to call

This is a shorthand for setting up a server that will keep handling requests, until it has heard from the specific instance requested.

Parameters

- **address** (*tuple*) – a hostname or IP address and port, in the same format given to socket and other servers
- **instance_id** (*str*) – the identification for the instance that should be calling back, and the condition for the wait to end

10.3.10 avocado.utils.configure_network module

Configure network when interface name and interface IP is available.

exception avocado.utils.configure_network.**NWException**
Bases: Exception

Base Exception Class for all exceptions

class avocado.utils.configure_network.**PeerInfo**(*host*, *port=None*, *peer_user=None*, *key=None*, *peer_password=None*)
Bases: object

class for peer function

create a object for accesses remote machine

get_peer_interface(*peer_ip*)
get peer interface from peer ip

set_mtu_peer(*peer_interface*, *mtu*)
Set MTU size in peer interface

avocado.utils.configure_network.**is_interface_link_up**(*interface*)
Checks if the interface link is up :param interface: name of the interface :return: True if the interface's link comes up, False otherwise.

avocado.utils.configure_network.**ping_check**(*interface*, *peer_ip*, *count*, *option=None*, *flood=False*)
Checks if the ping to peer works.

avocado.utils.configure_network.**set_ip**(*ipaddr*, *netmask*, *interface*, *interface_type=None*)
Gets interface name, IP, subnet mask and creates interface file based on distro.

avocado.utils.configure_network.**set_mtu_host**(*interface*, *mtu*)
Set MTU size in host interface

avocado.utils.configure_network.**unset_ip**(*interface*)
Gets interface name unassigns the IP to the interface

10.3.11 avocado.utils.cpu module

Get information from the current's machine CPU.

exception avocado.utils.cpu.**FamilyException**
Bases: Exception

avocado.utils.cpu.**VENDORS_MAP = {'amd': (b'AMD',), 'ibm': (b'POWER\\d', b'IBM/S390'), 'in**
Map vendor's name with expected string in /proc/cpuinfo.

avocado.utils.cpu.**cpu_has_flags**(*flags*)
Check if a list of flags are available on current CPU info.

Parameters **flags** (*list of str*) – A *list* of cpu flags that must exists on the current CPU.

Returns True if all the flags were found or False if not

Return type bool

avocado.utils.cpu.**cpu_online_list**(**args*, ***kwargs*)

avocado.utils.cpu.**get_arch**()
Work out which CPU architecture we're running on.

avocado.utils.cpu.**get_cpu_arch**(**args*, ***kwargs*)

avocado.utils.cpu.**get_cpu_vendor_name**(**args*, ***kwargs*)

avocado.utils.cpu.**get_cpufreq_governor**(**args*, ***kwargs*)

avocado.utils.cpu.**get_cpuidle_state**(**args*, ***kwargs*)

avocado.utils.cpu.**get_family**()
Get family name of the cpu like Broadwell, Haswell, power8, power9.

avocado.utils.cpu.**get_freq_governor**()
Get current cpu frequency governor.

avocado.utils.cpu.**get_idle_state**()
Get current cpu idle values.

Returns Dict of cpuidle states values for all cpus

Return type dict

avocado.utils.cpu.**get_pid_cpus**(*pid*)
Get all the cpus being used by the process according to pid informed.

Parameters **pid** (*str*) – process id

Returns A list include all cpus the process is using

Return type list

avocado.utils.cpu.**get_vendor**()
Get the current cpu vendor name.

Returns a key of *VENDORS_MAP* (e.g. 'intel') depending on the current CPU architecture. Return None if it was unable to determine the vendor name.

Return type str or None

avocado.utils.cpu.**get_version**()
Get cpu version.

Returns cpu version of given machine e.g.:- 'i5-5300U' for Intel and 'POWER9' for IBM machines in case of unknown/unsupported machines, return an empty string.

Return type str

avocado.utils.cpu.**offline**(*cpu*)
Offline given CPU.

avocado.utils.cpu.**online**(*cpu*)
Online given CPU.

avocado.utils.cpu.**online_count**()
Return Number of Online cpus in the system.

avocado.utils.cpu.**online_cpus_count**(**args*, ***kwargs*)

avocado.utils.cpu.**online_list**()
Reports a list of indexes of the online cpus.

avocado.utils.cpu.**set_cpufreq_governor**(**args*, ***kwargs*)

avocado.utils.cpu.**set_cpuidle_state**(**args*, ***kwargs*)

avocado.utils.cpu.**set_freq_governor**(*governor='random'*)
To change the given cpu frequency governor.

Parameters **governor** (*str*) – frequency governor profile name whereas *random* is default option to choose random profile among available ones.

avocado.utils.cpu.**set_idle_state**(*state_number='all'*, *disable=True*, *setstate=None*)
Set/Reset cpu idle states for all cpus.

Parameters

- **state_number** (*str*) – cpuidle state number, default: *all* all states
- **disable** (*bool*) – whether to disable/enable given cpu idle state, default is to disable.
- **setstate** (*dict*) – cpuidle state value, output of *get_idle_state()*

avocado.utils.cpu.**total_count**()
Return Number of Total cpus in the system including offline cpus.

avocado.utils.cpu.**total_cpus_count**(**args*, ***kwargs*)

10.3.12 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.
- **algorithm** – Method used to calculate the hash (default is md5).
- **size** – If provided, hash only the first size bytes of the file.

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

10.3.13 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

10.3.14 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**

Bases: object

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.**DataSize**(*data*)

Bases: object

Data Size object with builtin unit-converted attributes.

Parameters **data** (*str*) – Data size plus optional unit string. i.e. '10m'. No unit string means the data size is in bytes.

MULTIPLIERS = {'b': 1, 'g': 1073741824, 'k': 1024, 'm': 1048576, 't': 10995116277

b

g

k

m

t

unit

value

exception avocado.utils.data_structures.**InvalidDataSize**
Bases: ValueError

Signals that the value given to *DataSize* is not valid.

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.**comma_separated_ranges_to_list**(*string*)
Provides a list from comma separated ranges

Parameters **string** – string of comma separated range

Return list list of integer values in comma separated range

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')

10.3.15 avocado.utils.datadrainer module

data drainer

This module provides utility classes for draining data and dispatching it to different destinations. This is intended to be used concurrently with other code, usually test code producing the output to be drained/processed. A thread is started and maintained on behalf of the user.

class avocado.utils.datadrainer.**BaseDrainer**(*source*, *stop_check=None*, *name=None*)
Bases: abc.ABC

Base drainer, doesn't provide complete functionality to be useful.

Parameters

- **source** – where to read data from, this is intentionally abstract
- **stop_check** (*function*) – callable that should determine if the drainer should quit. If None is given, it will never stop.
- **name** (*str*) – instance name of the drainer, used for describing the name of the thread maintained by this instance

static data_available()
Checks if source appears to have data to be drained

name = 'avocado.utils.datadrainer.BaseDrainer'

read()
Abstract method supposed to read from the data source

start()
Starts a thread to do the data draining

wait()
Waits on the thread completion

write(*data*)
Abstract method supposed to write the read data to its destination

class avocado.utils.datadrainer.**BufferFDDrainer**(*source*, *stop_check=None*, *name=None*)
Bases: *avocado.utils.datadrainer.FDDrainer*

Drains data from a file descriptor and stores it in an internal buffer

data
Returns the buffer data, as bytes

name = 'avocado.utils.datadrainer.BufferFDDrainer'

write(*data*)
Abstract method supposed to write the read data to its destination

class avocado.utils.datadrainer.**FDDrainer**(*source*, *stop_check=None*, *name=None*)
Bases: *avocado.utils.datadrainer.BaseDrainer*

Drainer whose source is a file descriptor

This drainer uses select to efficiently wait for data to be available on a file descriptor. If the file descriptor is closed, the drainer responds by shutting itself down.

This drainer doesn't provide a write() implementation, and is consequently not a complete implementation users can pick and use.

Parameters

- **source** – where to read data from, this is intentionally abstract
- **stop_check** (*function*) – callable that should determine if the drainer should quit. If None is given, it will never stop.
- **name** (*str*) – instance name of the drainer, used for describing the name of the thread maintained by this instance

data_available()
Checks if source appears to have data to be drained

name = 'avocado.utils.datadrainer.FDDrainer'

read()
Abstract method supposed to read from the data source

write(*data*)
Abstract method supposed to write the read data to its destination

class avocado.utils.datadrainer.**LineLogger**(*source*, *stop_check=None*, *name=None*, *logger=None*)
Bases: *avocado.utils.datadrainer.FDDrainer*

name = 'avocado.utils.datadrainer.LineLogger'

write(*data*)
Abstract method supposed to write the read data to its destination

10.3.16 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)"

10.3.17 avocado.utils.diff_validator module

Diff validator: Utility for testing file changes

Some typical use of this utility would be:

```
>>> import diff_validator
>>> change = diff_validator.Change()
>>> change.add_validated_files(["/etc/somerc"])
>>> change.append_expected_add("/etc/somerc", "this is a new line")
>>> change.append_expected_remove("/etc/somerc", "this line is removed")
>>> diff_validator.make_temp_file_copies(change.get_target_files())
```

After making changes through some in-test operation:

```
>>> changes = diff_validator.extract_changes(change.get_target_files())
>>> change_success = diff_validator.assert_change(changes, change.files_dict)
```

If test fails due to invalid change on the system:

```
>>> if not change_success:
>>>     changes = diff_validator.assert_change_dict(changes, change.files_dict)
>>>     raise DiffValidationError("Change is different than expected:
%s" % diff_validator.create_diff_report(changes))
>>> else:
>>>     logging.info("Change made successfully")
>>> diff_validator.del_temp_file_copies(change.get_target_files())
```

class avocado.utils.diff_validator.**Change**
Bases: object

Class for tracking and validating file changes

Creates a change object.

add_validated_files(*filenames*)
Add file to change object.

Parameters **filenames** (*[str]*) – files to validate

append_expected_add(*filename*, *line*)
Append expected added line to a file.

Parameters

- **filename** (*str*) – file to append to
- **line** (*str*) – line to append to as an expected addition

append_expected_remove(*filename*, *line*)
Append removed added line to a file.

Parameters

- **filename** (*str*) – file to append to
- **line** (*str*) – line to append to as an expected removal

get_all_adds()
Return a list of the added lines for all validated files.

get_all_removes()
Return a list of the removed lines for all validated files.

get_target_files()
Get added files for change.

exception avocado.utils.diff_validator.**DiffValidationError**
Bases: Exception

avocado.utils.diff_validator.**assert_change**(*actual_result*, *expected_result*)
Condition wrapper of the upper method.

Parameters

- **actual_result** (*{str, ([str], [str])}*) – actual added and removed lines with filepath keys and a tuple of ([added_line, ...], [removed_line, ...])
- **expected_result** (*{str, ([str], [str])}*) – expected added and removed lines of type as the actual result

Returns whether changes were detected

Return type bool

avocado.utils.diff_validator.**assert_change_dict**(*actual_result*, *expected_result*)
Calculates unexpected line changes.

Parameters

- **actual_result** (*{file_path, ([added_line, ..], [removed_line, ..])}*) – actual added and removed lines
- **expected_result** (*{file_path, ([added_line, ..], [removed_line, ..])}*) – expected added and removed lines

Returns detected differences as groups of lines with filepath keys and a tuple of (unexpected_adds, not_present_adds, unexpected_removes, not_present_removes)

Return type {str, (str, str, str, str)}

avocado.utils.diff_validator.**create_diff_report**(*change_diffs*)
Pretty prints the output of the *change_diffs* variable.

Parameters **change_diffs** – detected differences as groups of lines with filepath keys and a tuple of (unexpected_adds, not_present_adds, unexpected_removes, not_present_removes)

Type {str, (str, str, str, str)}

Returns print string of the line differences

Return type str

avocado.utils.diff_validator.**del_temp_file_copies**(*file_paths*)
Deletes all the provided files.

Parameters **file_paths** (*[str]*) – deleted file paths (their temporary versions)

avocado.utils.diff_validator.**extract_changes**(*file_paths*, *compared_file_paths=None*)
Extracts diff information based on the new and temporarily saved old files.

Parameters

- **file_paths** (*[str]*) – original file paths (whose temporary versions will be retrieved)
- **compared_file_paths** (*[str] or None*) – custom file paths to use instead of the temporary versions

Returns file paths with corresponding diff information key-value pairs

Return type {str, ([str], [str])}

avocado.utils.diff_validator.**get_temp_file_path**(*file_path*)
Generates a temporary filename.

Parameters **file_path** (*str*) – file path prefix

Returns appended file path

Return type str

avocado.utils.diff_validator.**make_temp_file_copies**(*file_paths*)
Creates temporary copies of the provided files.

Parameters **file_paths** (*[str]*) – file paths to be copied

avocado.utils.diff_validator.**parse_unified_diff_output**(*lines*)
Parses the unified diff output of two files.

Parameters **lines** (*[str]*) – diff lines

Returns pair of adds and removes, where each is a list of trimmed lines

Return type ([str], [str])

10.3.18 avocado.utils.disk module

Disk utilities

exception avocado.utils.disk.**DiskError**
Bases: Exception

Generic DiskError

avocado.utils.disk.**create_loop_device**(*size*, *blocksize=4096*, *directory='./'*)
Creates a loop device of size and blocksize specified.

Parameters

- **size** (*int*) – Size of loop device, in bytes
- **blocksize** (*int*) – block size of loop device, in bytes. Defaults to 4096
- **directory** (*str*) – Directory where the backing file will be created. Defaults to current directory.

Returns loop device name

Return type str

avocado.utils.disk.**delete_loop_device**(*device*)
Deletes the specified loop device.

Parameters **device** (*str*) – device to be deleted

Returns True if deleted.

Return type bool

avocado.utils.disk.**freespace**(*path*)

avocado.utils.disk.**get_available_filesystems**()
Return a list of all available filesystem types

Returns a list of filesystem types

Return type list of str

avocado.utils.disk.**get_disk_blocksize**(*path*)
Return the disk block size, in bytes

avocado.utils.disk.**get_disks**()
Returns the physical "hard drives" available on this system

This is a simple wrapper around *lsblk* and will return all the top level physical (non-virtual) devices return by it.

TODO: this is currently Linux specific. Support for other platforms is desirable and may be implemented in the future.

Returns a list of paths to the physical disks on the system

Return type list of str

avocado.utils.disk.**get_filesystem_type**(*mount_point='/'*)
Returns the type of the filesystem of mount point informed. The default mount point considered when none is informed is the root "/" mount point.

Parameters **mount_point** (*str*) – mount point to asses the filesystem type. Default "/"

Returns filesystem type

Return type str

avocado.utils.disk.**is_root_device**(*device*)
check for root disk

Parameters **device** – device to check

Returns True or False, True if given device is root disk otherwise will return False.

10.3.19 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**(*session=None*)
Bases: object

Probes the machine and does it best to confirm it's the right distro. If given an avocado.utils.ssh.Session object representing another machine, Probe will attempt to detect another machine's distro via an ssh connection.

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_for_remote_file(*file_name*)
Checks if provided file exists in remote machine

Parameters **file_name** (*str*) – name of file

Returns whether the file exists in remote machine or not

Return type bool

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()

Parameters **session** – ssh connection between another machine

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**(*session=None*)
Attempts to detect the Linux Distribution running on this machine.

If given an avocado.utils.ssh.Session object, it will attempt to detect the distro of another machine via an ssh connection.

Parameters **session** (avocado.utils.ssh.Session) – ssh connection between another machine

Returns the detected *LinuxDistro* or UNKNOWN_DISTRO

Return type *LinuxDistro*

10.3.20 avocado.utils.dmesg module

Module for manipulate dmesg while running test.

exception avocado.utils.dmesg.**DmesgError**
Bases: Exception

Base Exception Class for all dmesg utils exceptions.

exception avocado.utils.dmesg.**TestFail**
Bases: AssertionError, Exception

Indicates that the test failed.

This is here, just because of an impossible circular import.

status = 'FAIL'

avocado.utils.dmesg.**clear_dmesg**()
function clear dmesg.

The dmesg operation is a privileged user task. This function needs sudo permissions enabled on the target host

avocado.utils.dmesg.**collect_dmesg**(*output_file=None*)
Function collect dmesg and save in file.

The dmesg operation is a privileged user task. This function needs sudo permissions enabled on the target host

Parameters **output_file** (*str*) – File use for save dmesg output if not provided it use tmp file which located in system /tmp path

Returns file which contain dmesg

Return type str

avocado.utils.dmesg.**collect_errors_by_level**(*output_file=None*, *level_check=5*, *skip_errors=None*)
Verify dmesg having severity level of OS issue(s).

Parameters

- **output_file** (*str*) – The file used to save dmesg
- **level_check** (*int*) – level of severity of issues to be checked 1 - emerg 2 - emerg,alert 3 - emerg,alert,crit 4 - emerg,alert,crit,err 5 - emerg,alert,crit,err,warn

Skip_errors list of dmesg error messages which want skip

avocado.utils.dmesg.**collect_errors_dmesg**(*patterns*)
Check patterns in dmesg.

:param patterns : list variable to search in dmesg :returns: error log in form of list :rtype: list of str

avocado.utils.dmesg.**fail_on_dmesg**(*level=5*)
Dmesg fail method decorator

Returns a class decorator used to signal the test when DmesgError exception is raised.

Parameters **level** (*int*) – Dmesg Level based on which the test failure should be raised

Returns Class decorator

Return type class

avocado.utils.dmesg.**skip_dmesg_messages**(*dmesg_stdout*, *skip_messages*)
Remove some messages from a dmesg buffer.

This method will remove some lines in a dmesg buffer if some strings are present. Returning the same buffer, but with less lines (in case of match).

Dmesg_stdout dmesg messages from which filter should be applied. This must be a decoded output buffer with new lines.

Skip_messages list of strings to be removed

10.3.21 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. Please, be aware that timeout here is just for blocking operations during the connection setup, since this method doesn't read the file from the url.

Returns file-like object.

Raises *URLError*.

10.3.22 avocado.utils.exit_codes module

Avocado Utilities exit codes.

These codes are returned on the command-line and may be used by the Avocado command-line utilities.

avocado.utils.exit_codes.**UTILITY_FAIL = 1**
The utility ran, but needs to signalize a fail.

avocado.utils.exit_codes.**UTILITY_GENERIC_CRASH = -1**
Utility generic crash

avocado.utils.exit_codes.**UTILITY_OK = 0**
The utility finished successfully

10.3.23 avocado.utils.file_utils module

SUMMARY

Utilities for file tests.

INTERFACE

avocado.utils.file_utils.**check_owner**(*owner*, *group*, *file_name_pattern*, *check_recursive=False*)
Verifies that given file belongs to given owner and group.

Parameters

- **owner** (*str*) – user that owns of the file
- **group** (*str*) – group of the owner of the file
- **file_name_pattern** (*str*) – can be a glob
- **check_recursive** (*bool*) – if file_name_pattern matches a directory, recurse into that subdir or not

Raises RuntimeError if file has wrong owner or group

avocado.utils.file_utils.**check_permissions**(*perms*, *file_name_pattern*)
Verify that a given file has a given numeric permission.

Parameters

- **perms** (*int*) – best given in octal form, e.g. 0o755
- **file_name_pattern** (*str*) – can be a glob

Raises RuntimeError if file has wrong permissions

10.3.24 avocado.utils.filelock module

Utility for individual file access control implemented via PID lock files.

exception avocado.utils.filelock.**AlreadyLocked**
Bases: 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: Exception

10.3.25 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=None*)
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

static checksum(*input_message*)

Calculates a remote message checksum.

More details are available at: https://sourceware.org/gdb/current/onlinedocs/gdb/Overview.html

Parameters **input_message** (*bytes*) – the message input payload, without the start and end markers

Returns two byte checksum

Return type bytes

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

static decode(*data*)

Decodes a packet and returns its payload.

More details are available at: https://sourceware.org/gdb/current/onlinedocs/gdb/Overview.html

Parameters **command_data** (*bytes*) – the command data payload

Returns the encoded command, ready to be sent to a remote GDB

Return type bytes

static encode(*data*)

Encodes a command.

That is, add prefix, suffix and checksum.

More details are available at: https://sourceware.org/gdb/current/onlinedocs/gdb/Overview.html

Parameters **command_data** (*bytes*) – the command data payload

Returns the encoded command, ready to be sent to a remote GDB

Return type bytes

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

10.3.26 avocado.utils.genio module

Avocado generic IO related functions.

exception avocado.utils.genio.**GenIOError**
Bases: Exception

Base Exception Class for all IO exceptions

avocado.utils.genio.**append_file**(*filename*, *data*)
Append data to a file.

Parameters

- **filename** (*str*) – Path to the file.
- **line** (*str*) – Line to be written.

avocado.utils.genio.**append_one_line**(*filename*, *line*)
Append one line of text to filename.

Parameters

- **filename** (*str*) – Path to the file.
- **line** (*str*) – Line to be written.

avocado.utils.genio.**are_files_equal**(*filename*, *other*)
Comparison of two files line by line :param filename: path to the first file :type filename: str :param other: path to the second file :type other: str :return: equality of file :rtype: boolean

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.**is_pattern_in_file**(*filename*, *pattern*)
Check if a pattern matches in a specified file. If a non regular file be informed a GenIOError will be raised.

Parameters

- **filename** (*str*) – Path to file
- **pattern** (*str*) – Pattern that need to match in file

Returns True when pattern matches in file if not return False

Return type boolean

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.**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.

10.3.27 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

10.3.28 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*, *capabilities=None*)

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
- **capabilities** (*list*) – list of specific capabilities that are required for the selected implementation, such as "read", "copy" and "mnt_dir".

Returns an instance of any iso9660 capable tool

Return type *Iso9660IsoInfo*, *Iso9660IsoRead*, *Iso9660Mount*, *ISO9660PyCDLib* 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*)

Abstract method to read data from path

Parameters **path** – path to the file

Returns data content from the file

Return type str

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()
Cleanups and frees any resources being used

copy(*src*, *dst*)
Simplistic version of copy that relies on read()

Parameters

- **src** (*str*) – source path
- **dst** (*str*) – destination path

Return type None

read(*path*)
Abstract method to read data from path

Parameters **path** – path to the file

Returns data content from the file

Return type str

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
Returns a path to the browsable content of the iso

read(*path*)
Read data from path

Parameters **path** (*str*) – path to read data

Returns data content

Return type str

class avocado.utils.iso9660.**ISO9660PyCDLib**(*path*)
Bases: avocado.utils.iso9660.MixInMntDirMount, avocado.utils.iso9660.BaseIso9660

Represents a ISO9660 filesystem

This implementation is based on the pycdlib library

DEFAULT_CREATE_FLAGS = {'interchange_level': 3, 'joliet': 3}
Default flags used when creating a new ISO image

close()
Cleanups and frees any resources being used

copy(*src*, *dst*)
Simplistic version of copy that relies on read()

Parameters

- **src** (*str*) – source path
- **dst** (*str*) – destination path

Return type None

create(*flags=None*)
Creates a new ISO image

Parameters **flags** (*dict*) – the flags used when creating a new image

read(*path*)
Abstract method to read data from path

Parameters **path** – path to the file

Returns data content from the file

Return type str

write(*path*, *content*)
Writes a new file into the ISO image

Parameters

- **path** (*bytes*) – the path of the new file inside the ISO image
- **content** – the content of the new file

10.3.29 avocado.utils.kernel module

Provides utilities for the Linux kernel.

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/v{major}.x/'

build(*binary_package=False*, *njobs=2*)
Build kernel from source.

Parameters

- **binary_package** – when True, the appropriate platform package is built for install() to use
- **njobs** (*int or None*) – number of jobs. It is mapped to the -j option from make. If njobs is None then do not limit the number of jobs (e.g. uses -j without value). The -j is omitted if a value equal or less than zero is passed. Default value is set to *multiprocessing.cpu_count()*.

build_dir
Return the build path if the directory exists

configure(*targets='defconfig'*, *extra_configs=None*)
Configure/prepare kernel source to build.

Parameters

- **targets** (*list of str*) – configuration targets. Default is 'defconfig'.
- **extra_configs** (*list of str*) – additional configurations in the form of CONFIG_NAME=VALUE.

download(*url=None*)
Download kernel source.

Parameters **url** (*str or None*) – override the url from where to fetch the kernel source tarball

install()
Install built kernel.

uncompress()
Uncompress kernel source.

Raises Exception in case the tarball is not downloaded

vmlinux
Return the vmlinux path if the file exists

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

10.3.30 avocado.utils.linux module

Linux OS utilities

avocado.utils.linux.**enable_selinux_enforcing**()
Enable SELinux Enforcing in system

Returns True if SELinux enable in enforcing mode, False if not enabled

avocado.utils.linux.**get_proc_sys**(*key*)
Read values from /proc/sys

Parameters **key** – A location under /proc/sys

Returns The single-line sysctl value as a string.

avocado.utils.linux.**is_selinux_enforcing**()
Returns True if SELinux is in enforcing mode, False if permissive/disabled.

avocado.utils.linux.**set_proc_sys**(*key*, *value*)
Set values on /proc/sys

Parameters

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

Returns The single-line sysctl value as a string.

10.3.31 avocado.utils.linux_modules module

Linux kernel modules APIs

class avocado.utils.linux_modules.**ModuleConfig**
Bases: enum.Enum

An enumeration.

BUILTIN = <object object>
Config built-in to kernel (=*y*)

MODULE = <object object>
Config compiled as loadable module (=*m*)

NOT_SET = <object object>
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 *ModuleConfig*

avocado.utils.linux_modules.**get_loaded_modules**()
Gets list of loaded modules. :return: List of loaded modules.

avocado.utils.linux_modules.**get_modules_dir**()
Return the modules dir for the running kernel version

Returns path of module directory

Return type String

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 regex 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 regex 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.

10.3.32 avocado.utils.lv_utils module

exception avocado.utils.lv_utils.**LVException**
Bases: Exception

Base Exception Class for all exceptions

avocado.utils.lv_utils.**get_device_total_space**(*disk*)
Get the total device size.

Parameters **device** (*str*) – name of the device/disk to find the total size

Returns size in bytes

Return type int

Raises *LVException* on failure to find disk space

avocado.utils.lv_utils.**get_devices_total_space**(*devices*)
Get the total size of given device(s)/disk(s).

Parameters **devices** (*list*) – list with the names of devices separated with space.

Returns sizes in bytes

Return type int

Raises *LVException* on failure to find disk space

avocado.utils.lv_utils.**get_diskspace**(*disk*)
Get the entire disk space of a given disk.

Parameters **disk** (*str*) – name of the disk to find the free space of

Returns size in bytes

Return type str

Raises *LVException* on failure to find disk space

avocado.utils.lv_utils.**lv_check**(*vg_name*, *lv_name*)
Check whether provided logical volume exists.

Parameters

- **vg_name** (*str*) – name of the volume group
- **lv_name** (*str*) – name of the logical volume

Returns whether the logical volume was found

Return type bool

avocado.utils.lv_utils.**lv_create**(*vg_name*, *lv_name*, *lv_size*, *force_flag=True*, *pool_name=None*, *pool_size='1G'*)
Create a (possibly thin) logical volume in a volume group. The volume group must already exist.

A thin pool will be created if pool parameters are provided and the thin pool doesn't already exist.

The volume group must already exist.

Parameters

- **vg_name** (*str*) – name of the volume group
- **lv_name** (*str*) – name of the logical volume
- **lv_size** (*str*) – size for the logical volume to be created
- **force_flag** (*bool*) – whether to abort if volume already exists or remove and recreate it
- **pool_name** (*str*) – name of thin pool or None for a regular volume
- **pool_size** (*str*) – size of thin pool if it will be created

Raises *LVException* if preconditions or execution fails

avocado.utils.lv_utils.**lv_list**(*vg_name=None*)
List all info about available logical volumes.

Parameters **vg_name** (*str*) – name of the volume group or None to list all

Returns list of available logical volumes

Return type {str, {str, str}}

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** (*str*) – name of the volume group
- **lv_name** (*str*) – name of the logical volume
- **mount_loc** (*str*) – location to mount the logical volume to
- **create_filesystem** (*str*) – can be one of ext2, ext3, ext4, vfat or empty if the filesystem was already created and the mkfs process is skipped

Raises *LVException* if the logical volume could not be mounted

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** (*str*) – name of the volume group
- **lv_name** (*str*) – name of the logical volume
- **timeout** (*int*) – timeout between operations

Raises *LVException* if the logical volume is still active

avocado.utils.lv_utils.**lv_remove**(*vg_name*, *lv_name*)
Remove a logical volume.

Parameters

- **vg_name** (*str*) – name of the volume group
- **lv_name** (*str*) – name of the logical volume

Raises *LVException* if volume group or logical volume cannot be found

avocado.utils.lv_utils.**lv_revert**(*vg_name*, *lv_name*, *lv_snapshot_name*)
Revert the origin logical volume to a snapshot.

Parameters

- **vg_name** (*str*) – name of the volume group
- **lv_name** (*str*) – name of the logical volume
- **lv_snapshot_name** (*str*) – name of the snapshot to be reverted

Raises `process.CmdError` on failure to revert snapshot

Raises *LVException* if preconditions or execution fails

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** (*str*) – name of the volume group
- **lv_name** (*str*) – name of the logical volume
- **lv_snapshot_name** (*str*) – name of the snapshot to be reverted
- **lv_snapshot_size** (*str*) – size of the snapshot

avocado.utils.lv_utils.**lv_take_snapshot**(*vg_name*, *lv_name*, *lv_snapshot_name*, *lv_snapshot_size=None*, *pool_name=None*)
Take a (possibly thin) snapshot of a regular (or thin) logical volume.

Parameters

- **vg_name** (*str*) – name of the volume group
- **lv_name** (*str*) – name of the logical volume
- **lv_snapshot_name** (*str*) – name of the snapshot be to created
- **lv_snapshot_size** (*str*) – size of the snapshot or None for thin snapshot of an already thin volume

- **pool_name** – name of thin pool or None for regular snapshot or snapshot in the same thin pool like the volume

Raises `process.CmdError` on failure to create snapshot

Raises *LVException* if preconditions fail

avocado.utils.lv_utils.**lv_umount**(*vg_name*, *lv_name*)
Unmount a Logical volume from a mount location.

Parameters

- **vg_name** (*str*) – name of the volume group
- **lv_name** (*str*) – name of the logical volume

Raises *LVException* if the logical volume could not be unmounted

avocado.utils.lv_utils.**vg_check**(*vg_name*)
Check whether provided volume group exists.

Parameters **vg_name** (*str*) – name of the volume group

Returns whether the volume group was found

Return type bool

avocado.utils.lv_utils.**vg_create**(*vg_name*, *pv_list*, *force=False*)
Create a volume group from a list of physical volumes.

Parameters

- **vg_name** (*str*) – name of the volume group
- **pv_list** (*str or [str]*) – list of physical volumes to use
- **force** (*bool*) – create volume group with a force flag

Raises *LVException* if volume group already exists

avocado.utils.lv_utils.**vg_list**(*vg_name=None*)
List all info about available volume groups.

Parameters **vg_name** (*str or None*) – name of the volume group to list or or None to list all

Returns list of available volume groups

Return type {str, {str, str}}

avocado.utils.lv_utils.**vg_ramdisk**(*disk*, *vg_name*, *ramdisk_vg_size*, *ramdisk_basedir*, *ramdisk_sparse_filename*, *use_tmpfs=True*)
Create volume group on top of ram memory to speed up LV performance. When disk is specified the size of the physical volume is taken from existing disk space.

Parameters

- **disk** (*str*) – name of the disk in which volume groups are created
- **vg_name** (*str*) – name of the volume group
- **ramdisk_vg_size** (*str*) – size of the ramdisk virtual group (MB)
- **ramdisk_basedir** (*str*) – base directory for the ramdisk sparse file
- **ramdisk_sparse_filename** (*str*) – name of the ramdisk sparse file
- **use_tmpfs** (*bool*) – whether to use RAM or slower storage

Returns ramdisk_filename, vg_ramdisk_dir, vg_name, loop_device

Return type (str, str, str, str)

Raises *LVException* on failure at any stage

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*, *use_tmpfs=True*)

Clean up any stage of the VG ramdisk setup in case of test error.

This detects whether the components were initialized and if so tries to remove them. In case of failure it raises summary exception.

Parameters

- **ramdisk_filename** (*str*) – name of the ramdisk sparse file
- **vg_ramdisk_dir** (*str*) – location of the ramdisk file
- **vg_name** (*str*) – name of the volume group
- **loop_device** (*str*) – name of the disk or loop device
- **use_tmpfs** (*bool*) – whether to use RAM or slower storage

Returns ramdisk_filename, vg_ramdisk_dir, vg_name, loop_device

Return type (str, str, str, str)

Raises *LVException* on intolerable failure at any stage

avocado.utils.lv_utils.**vg_reactivate**(*vg_name*, *timeout=10*, *export=False*)

In case of unclean shutdowns some of the vgs 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** (*str*) – name of the volume group
- **timeout** (*int*) – timeout between operations

Raises *LVException* if the logical volume is still active

avocado.utils.lv_utils.**vg_remove**(*vg_name*)

Remove a volume group.

Parameters **vg_name** (*str*) – name of the volume group

Raises *LVException* if volume group cannot be found

10.3.33 avocado.utils.memory module

exception avocado.utils.memory.**MemError**

Bases: Exception

called when memory operations fails

class avocado.utils.memory.**MemInfo**

Bases: object

Representation of /proc/meminfo

avocado.utils.memory.**check_hotplug**()
Check kernel support for memory hotplug

Returns True if hotplug supported, else False

Return type 'bool'

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_blk_string**(*block*)
Format the given block id to string

Parameters **block** – memory block id or block string.

Returns returns string memory198 if id 198 is given

Return type string

avocado.utils.memory.**get_buddy_info**(*chunk_sizes*, *nodes='all'*, *zones='all'*)
Get the fragment 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_page_size**()
Get linux page size for this system.

:return Kernel page size (Bytes).

avocado.utils.memory.**get_supported_huge_pages_size**()
Get all supported huge page sizes for this system.

Returns list of Huge pages size (kB).

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.**hotplug**(*block*)
Online the memory for the given block id.

Parameters **block** – memory block id or or memory198

avocado.utils.memory.**hotunplug**(*block*)
Offline the memory for the given block id.

Parameters **block** – memory block id.

avocado.utils.memory.**is_hot_pluggable**(*block*)
Check if the given memory block is hotpluggable

Parameters **block** – memory block id.

Returns True if hotpluggable, else False

Return type 'bool'

avocado.utils.memory.**memtotal**()
Read Memtotal from meminfo.

avocado.utils.memory.**memtotal_sys**()
Reports actual memory size according to online-memory blocks available via "/sys"

Returns system memory in Kb as float

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.**numa_nodes_with_memory**()
Get a list of NUMA nodes present with memory on the system.

Returns List with nodes which has memory.

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

10.3.34 avocado.utils.multipath module

Module with multipath related utility functions. It needs root access.

exception avocado.utils.multipath.**MPException**
Bases: Exception

Base Exception Class for all exceptions

avocado.utils.multipath.**add_mpath**(*mpath*)
Add back the removed mpathX of multipath.

Parameters **mpath_name** – mpath names. Example: mpatha, mpathb.

Returns True or False

avocado.utils.multipath.**add_path**(*path*)
Add back the removed individual paths.

Parameters **path** (*str*) – disk path. Example: sda, sdb.

Returns True or False

avocado.utils.multipath.**device_exists**(*mpath*)
Checks if a given mpath exists.

Parameters **mpath** – The multipath path

Returns True if path exists, False if does not exist.

Return type bool

avocado.utils.multipath.**fail_path**(*path*)
Fail the individual paths.

Parameters **path** (*str*) – disk path. Example: sda, sdb.

Returns True if succeeded, False otherwise

Return type bool

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.

Return type str

avocado.utils.multipath.**get_mpath_status**(*mpath*)
Get the status of mpathX of multipaths.

Parameters **mpath** – mpath names. Example: mpatha, mpathb.

Returns state of mpathX eg: Active, Suspend, None

avocado.utils.multipath.**get_multipath_details**()
Get multipath details as a dictionary.

This is the output of the following command:

$ multipathd show maps json

Returns Dictionary of multipath output in json format

Return type dict

avocado.utils.multipath.**get_multipath_wwid**(*mpath*)
Get the wwid binding for given mpath name

Returns Multipath wwid

Return type str

avocado.utils.multipath.**get_multipath_wwids**()
Get list of multipath wwids.

Returns List of multipath wwids.

Return type list of str

avocado.utils.multipath.**get_path_status**(*disk_path*)
Return the status of a path in multipath.

Parameters **disk_path** – disk path. Example: sda, sdb.

Returns Tuple in the format of (dm status, dev status, checker status)

avocado.utils.multipath.**get_paths**(*wwid*)
Get list of paths, given a multipath wwid.

Returns List of paths.

Return type list of str

avocado.utils.multipath.**get_policy**(*wwid*)
Gets path_checker policy, given a multipath wwid.

Returns path checker policy.

Return type str

avocado.utils.multipath.**get_size**(*wwid*)
Gets size of device, given a multipath wwid.

Returns size of multipath device.

Return type str

avocado.utils.multipath.**get_svc_name**()
Gets the multipath service name based on distro.

avocado.utils.multipath.**is_mpath_dev**(*mpath*)
Check the give name is a multipath device name or not.

Returns True if device is multipath or False

Return type Boolean

avocado.utils.multipath.**is_path_a_multipath**(*disk_path*)
Check if given disk path is part of a multipath.

Parameters **disk_path** – disk path. Example: sda, sdb.

Returns True if part of multipath, else False.

avocado.utils.multipath.**reinstate_path**(*path*)
Reinstate the individual paths.

Parameters **path** (*str*) – disk path. Example: sda, sdb.

Returns True if succeeded, False otherwise

avocado.utils.multipath.**remove_mpath**(*mpath*)
Remove the mpathX of multipaths.

Parameters **mpath_name** – mpath names. Example: mpatha, mpathb.

Returns True or False

avocado.utils.multipath.**remove_path**(*path*)
Remove the individual paths.

Parameters **disk_path** – disk path. Example: sda, sdb.

Returns True or False

avocado.utils.multipath.**resume_mpath**(*mpath*)
Resume the suspended mpathX of multipaths.

Parameters **mpath_name** – mpath names. Example: mpatha, mpathb.

Returns True or False

avocado.utils.multipath.**suspend_mpath**(*mpath*)
Suspend the given mpathX of multipaths.

Parameters **mpath** – mpath names. Example: mpatha, mpathb.

Returns True or False

10.3.35 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.

10.3.36 avocado.utils.partition module

Utility for handling partitions.

class avocado.utils.partition.**MtabLock**(*timeout=60*)
Bases: object

device = '/etc/mtab'

class avocado.utils.partition.**Partition**(*device*, *loop_size=0*, *mountpoint=None*, *mkfs_flags=''*, *mount_options=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.
- **mkfs_flags** – Optional flags for mkfs
- **mount_options** – Add mount options optionally

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=''*, *mnt_check=True*)

Mount this partition to a mount point

Parameters

- **mountpoint** – If you have not provided a mountpoint to partition object or want to use a different one, you may specify it here.
- **fstype** – Filesystem type. If not provided partition object value will be used.
- **args** – Arguments to be passed to "mount" command.
- **mnt_check** – Flag to check/avoid checking existing device/mountpoint

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: Exception

Generic PartitionError

10.3.37 avocado.utils.path module

Avocado path related functions.

exception avocado.utils.path.**CmdNotFoundError**(*cmd*, *paths*)
Bases: 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.**check_readable**(*path*)
Verify that the given path exists and is readable

This should be used where an assertion makes sense, and is useful because it can provide a better message in the exception it raises.

Parameters **path** (*str*) – the path to test

Raises **OSError** – path does not exist or path could not be read

Return type None

avocado.utils.path.**find_command**(*cmd*, *default=None*, *check_exec=True*)
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.
- **check_exec** (*bool*) – if a check for permissions that render the command executable by the current user should be performed.

Raise *avocado.utils.path.CmdNotFoundError* in case the command was not found and no default was given.

Returns Returns an absolute path to the command or the default value if the command is not found

Return type str

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.

Check if a usable RO directory is there.

Parameters **directory** – Directory

avocado.utils.path.**usable_rw_dir**(*directory*, *create=True*)
Verify whether we can use this dir (read/write).

Checks for appropriate permissions, and creates missing dirs as needed.

Parameters

- **directory** – Directory
- **create** – whether to create the directory

10.3.38 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.

Return type dict

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.

Return type list of str

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.

Return type str

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.

Return type str

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.

Return type str

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.

Return type int

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)

Return type list of str

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.

Return type str

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 Removed port related details using re, only returns till physical slot of the adapter.

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.

Return type dict

10.3.39 avocado.utils.pmem module

class avocado.utils.pmem.**PMem**(*ndctl='ndctl'*, *daxctl='daxctl'*)
Bases: object

PMem class which provides function to perform ndctl and daxctl operations

This class can be used only if ndctl binaries are provided before hand

Initialize PMem object

Parameters

- **ndctl** – path to ndctl binary, defaults to ndctl
- **daxctl** – path to daxctl binary, defaults to ndctl

static check_buses()
Get buses from sys subsystem to verify persistent devices exist

Returns List of buses available

Return type list

check_daxctl_subcmd(*command*)
Check if given sub command is supported by daxctl

check_ndctl_subcmd(*command*)
Check if given sub command is supported by ndctl

static check_subcmd(*binary*, *command*)
Check if given sub command is supported by binary

Parameters **command** – sub command of ndctl to check for existence

Returns True if sub command is available

Return type bool

create_namespace(*region=''*, *bus=''*, *n_type='pmem'*, *mode='fsdax'*, *memmap='dev'*, *name=''*, *size=''*, *uuid=''*, *sector_size=''*, *align=''*, *reconfig=''*, *force=False*, *autolabel=False*)
Creates namespace with specified options

Parameters

- **region** – Region on which namespace has to be created
- **bus** – Bus with which namespace has to be created
- **n_type** – Type of namespace to be created [pmem/blk]
- **mode** – Mode of namespace to be created, defaults to fsdax

- **memmap** – Metadata mapping for created namespace
- **name** – Optional name provided for namespace
- **size** – Size with which namespace has to be created
- **uuid** – Optional uuid provided for namespace
- **sector_size** – Sector size with which namespace has to be created
- **align** – Alignment with which namespace has to be created
- **reconfig** – Optionally reconfigure namespace providing existing namespace/region name
- **force** – Force creation of namespace
- **autolabel** – Optionally autolabel the namespace

Returns True on success

Raise *PMemException*, if command fails.

destroy_namespace(*namespace='all'*, *region=''*, *bus=''*, *force=False*)
Destroy namespaces, skipped in case of legacy namespace

Parameters

- **namespace** – name of the namespace to be destroyed
- **region** – Filter namespace by region
- **bus** – Filter namespace by bus
- **force** – Force a namespace to be destroyed

Returns True on Success

Raise *PMemException*, if command fails.

disable_namespace(*namespace='all'*, *region=''*, *bus=''*, *verbose=False*)
Disable namespaces

Parameters

- **namespace** – name of the namespace to be disabled
- **region** – Filter namespace by region
- **bus** – Filter namespace by bus
- **verbose** – Enable True command with debug information

Returns True on success

Raise *PMemException*, if command fails.

disable_region(*name='all'*)
Disable given region

Parameters **name** – name of the region to be disabled

Returns True on success

Raise *PMemException*, if command fails.

enable_namespace(*namespace='all'*, *region=''*, *bus=''*, *verbose=False*)
Enable namespaces

Parameters

- **namespace** – name of the namespace to be enabled
- **region** – Filter namespace by region
- **bus** – Filter namespace by bus
- **verbose** – Enable True command with debug information

return: True on success :raise: *PMemException*, if command fails.

enable_region(*name='all'*)
Enable given region

Parameters **name** – name of the region to be enabled

Returns True on success

Raise *PMemException*, if command fails.

get_slot_count(*region*)
Get max slot count in the index area for a dimm backing a region We use region0 - > nmem0

Parameters **region** – Region for which slot count is found

Returns Number of slots for given region 0 in case region is not available/command fails

Return type int

static is_region_legacy(*region*)
Check whether we have label index namespace. If legacy we can't create new namespaces.

Parameters **region** – Region for which legacy check is made

Returns True if given region is legacy, else False

read_infoblock(*namespace=''*, *inp_file=''*, ***kwargs*)
Read an infoblock from the specified medium

Parameters

- **namespace** – Read the infoblock from given namespace
- **inp_file** – Input file to read the infoblock from
- **kwargs** –

Example: self.plib.read_infoblock(namespace=ns_name, json_form=True)

Returns By default return list of json objects, if json_form is True Return as raw data, if json_form is False Return file path if op_file is specified

Raise *PMemException*, if command fails.

reconfigure_dax_device(*device*, *mode='devdax'*, *region=None*, *no_online=False*, *no_movable=False*)
Reconfigure devdax device into devdax or system-ram mode

Parameters

- **device** – Device from which memory is to be online
- **mode** – Mode with which device is to be configured, default:devdax
- **region** – Optionally filter device by region
- **no_online** – Optionally don't online the memory(only system-ram)
- **no_movable** – Optionally mark memory non-movable(only system-ram)

Returns Property of configured device

Return type str

Raise *PMemException*, if command fails.

run_daxctl_list(*options=''*)
Get the json of each provided options

Parameters **options** – optional arguments to daxctl list command

Returns By default returns entire list of json objects

Return type list of json objects

run_ndctl_list(*option=''*)
Get the json of each provided options

Parameters **option** – optional arguments to ndctl list command

Returns By default returns entire list of json objects

Return type list of json objects

static run_ndctl_list_val(*json_op*, *field*)
Get the value of a field in given json

Parameters

- **json_op** – Input Json object
- **field** – Field to find the value from json_op object

Return type Found value type, None if not found

set_dax_memory_offline(*device*, *region=None*)
Set memory from a given devdax device offline

Parameters

- **device** – Device from which memory is to be offline
- **region** – Optionally filter device by region

Returns True if command succeeds

Return type bool

Raise *PMemException*, if command fails.

set_dax_memory_online(*device*, *region=None*, *no_movable=False*)
Set memory from a given devdax device online

Parameters

- **device** – Device from which memory is to be online
- **region** – Optionally filter device by region
- **no_movable** – Optionally make the memory non-movable

Returns True if command succeeds

Return type bool

Raise *PMemException*, if command fails.

write_infoblock(*namespace=''*, *stdout=False*, *output=None*, ***kwargs*)
Write an infoblock to the specified medium.

Parameters

- **namespace** – Write the infoblock to given namespace
- **stdout** – Write the infoblock to stdout if True
- **output** – Write the infoblock to the file path specified
- **kwargs** –

Example:

pmem.write_infoblock(namespace=ns_name, align=align, size=size, mode='devdax')

Returns True if command succeeds

Return type bool

Raise *PMemException*, if command fails.

exception avocado.utils.pmem.**PMemException**(*additional_text=None*)

Bases: Exception

Error raised for all PMem failures

10.3.40 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: Exception

exception avocado.utils.process.**CmdInputError**

Bases: Exception

Raised when the command given is invalid, such as an empty command.

class avocado.utils.process.**CmdResult**(*command='', stdout=b'', stderr=b'', exit_status=None, duration=0, pid=None, encoding=None*)

Bases: object

Command execution result.

Parameters

- **command** (*str*) – the command line itself
- **exit_status** (*int*) – exit code of the process
- **stdout** (*bytes*) – content of the process stdout
- **stderr** (*bytes*) – content of the process stderr
- **duration** (*float*) – elapsed wall clock time running the process
- **pid** (*int*) – ID of the process
- **encoding** (*str*) – the encoding to use for the text version of stdout and stderr, by default *avocado.utils.astring.ENCODING*

stderr = None
The raw stderr (bytes)

stderr_text

stdout = None
The raw stdout (bytes)

stdout_text

class avocado.utils.process.**FDDrainer**(*fd*, *result*, *name=None*, *logger=None*, *logger_prefix='%s'*, *stream_logger=None*, *ignore_bg_processes=False*, *verbose=False*)

Bases: object

Reads data from a file descriptor in a thread, storing locally in a file-like data object.

Parameters

- **fd** (*int*) – a file descriptor that will be read (drained) from
- **result** (a *CmdResult* instance) – a *CmdResult* instance associated with the process used to detect if the process is still running and if there's still data to be read.
- **name** (*str*) – a descriptive name that will be passed to the Thread name
- **logger** (logging.Logger) – the logger that will be used to (interactively) write the content from the file descriptor
- **logger_prefix** (*str with one %-style string formatter*) – the prefix used when logging the data
- **ignore_bg_processes** (*boolean*) – 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.
- **verbose** (*boolean*) – whether to log in both the logger and stream_logger

flush()

start()

avocado.utils.process.**NRUNNER_MODE = True**
When using the nrunner architecture we don't need to log messages into the stream_logger as well. By setting this to True, we will set stream_logger to None.

avocado.utils.process.**OUTPUT_CHECK_RECORD_MODE = None**
The current output record mode. It's not possible to record both the 'stdout' and 'stderr' streams, and at the same time in the right order, the combined 'output' stream. So this setting defines the mode.

class avocado.utils.process.**SubProcess**(*cmd*, *verbose=True*, *allow_output_check=None*, *shell=False*, *env=None*, *sudo=False*, *ignore_bg_processes=False*, *encoding=None*)

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 record the output from this process (from stdout and stderr) in the test's output record files. Valid values: 'stdout', for standard output *only*, 'stderr' for standard error *only*, 'both' for both standard output and error in separate files, 'combined' for standard output and error in a single file, and 'none' to disable all recording. 'all' is also a valid, but deprecated, option that is a synonym of 'both'. If an explicit value is not given to this parameter, that is, if None is given, it defaults to using the module level configuration, as set by *OUTPUT_CHECK_RECORD_MODE*. If the module level configuration itself is not set, it defaults to 'none'.
- **shell** (*bool*) – Whether to run the subprocess in 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** – 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.
- **encoding** (*str*) – the encoding to use for the text representation of the command result stdout and stderr, by default *avocado.utils.astring.ENCODING*

Raises ValueError if incorrect values are given to parameters

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

get_user_id()
Reports user id of this process

is_sudo_enabled()
Returns whether the subprocess is running with sudo enabled

kill()
Send a `signal.SIGKILL` to the process. Please consider using *stop()* instead if you want to do all that's possible to finalize the process and wait for it to finish.

poll()
Call the subprocess poll() method, fill results if rc is not None.

run(*timeout=None*, *sig=<Signals.SIGTERM: 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 it's children using `sig` and get a status. When the process refuses to die within 1s we use SIGKILL and report the status (be it exit_code or zombie)
- **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(*timeout=None*)
Stop background subprocess.

Call this method to terminate the background subprocess and wait for it results.

Parameters **timeout** – Time (seconds) we'll wait until the process is finished. If it's not, we'll try to terminate it and it's children using `sig` and get a status. When the process refuses to die within 1s we use SIGKILL and report the status (be it exit_code or zombie)

terminate()
Send a `signal.SIGTERM` to the process. Please consider using *stop()* instead if you want to do all that's possible to finalize the process and wait for it to finish.

wait(*timeout=None*, *sig=<Signals.SIGTERM: 15>*)
Call the subprocess poll() method, fill results if rc is not None.

Parameters

- **timeout** – Time (seconds) we'll wait until the process is finished. If it's not, we'll try to terminate it and it's children using `sig` and get a status. When the process refuses to die within 1s we use SIGKILL and report the status (be it exit_code or zombie)
- **sig** – Signal to send to the process in case it did not end after the specified timeout.

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=None*, *shell=False*, *env=None*, *wrapper=None*, *sudo=False*, *ignore_bg_processes=False*, *encoding=None*)

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** (*unicode string*) – 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)

Parameters **cmd** – unicode string with the commands

avocado.utils.process.**cmd_split**(*s*, *comments=False*, *posix=True*)

This is kept for compatibility purposes, but is now deprecated and will be removed in later versions. Please use `shlex.split()` instead.

avocado.utils.process.**get_capabilities**(*pid=None*)

Gets a list of all capabilities for a process.

In case the getpcaps command is not available, and empty list will be returned.

It supports getpcaps' two different formats, the current and the so called legacy/ugly.

Parameters **pid** (*int*) – the process ID (PID), if one is not given, the current PID is used (given by `os.getpid()`)

Returns all capabilities

Return type list

avocado.utils.process.**get_children_pids**(*parent_pid*, *recursive=False*)

Returns the children PIDs for the given process

Note This is currently Linux specific.

Parameters **parent_pid** – The PID of parent child process

Returns The PIDs for the children processes

Return type list of int

avocado.utils.process.**get_command_output_matching**(*command*, *pattern*)

Runs a command, and if the pattern is in in the output, returns it.

Parameters

- **command** (*str*) – the command to execute
- **pattern** (*str*) – pattern to search in the output, in a line by line basis

Returns list of lines matching the pattern

Return type list of str

avocado.utils.process.**get_owner_id**(*pid*)

Get the owner's user id of a process

Parameters **pid** – the process id

Returns user id of the process owner

avocado.utils.process.**get_parent_pid**(*pid*)
Returns the parent PID for the given process

Note This is currently Linux specific.

Parameters **pid** – The PID of child process

Returns The parent PID

Return type int

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.**getoutput**(*cmd*, *timeout=None*, *verbose=False*, *ignore_status=True*, *allow_output_check='combined'*, *shell=True*, *env=None*, *sudo=False*, *ignore_bg_processes=False*)
Because commands module is removed in Python3 and it redirect stderr to stdout, we port commands.getoutput to make code compatible Return output (stdout or stderr) of executing cmd in a shell.

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 record the output from this process (from stdout and stderr) in the test's output record files. Valid values: 'stdout', for standard output *only*, 'stderr' for standard error *only*, 'both' for both standard output and error in separate files, 'combined' for standard output and error in a single file, and 'none' to disable all recording. 'all' is also a valid, but deprecated, option that is a synonym of 'both'. If an explicit value is not given to this parameter, that is, if None is given, it defaults to using the module level configuration, as set by *OUTPUT_CHECK_RECORD_MODE*. If the module level configuration itself is not set, it defaults to 'none'.
- **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

Returns Command output(stdout or stderr).

Return type str

avocado.utils.process.**getstatusoutput**(*cmd*, *timeout=None*, *verbose=False*, *ignore_status=True*, *allow_output_check='combined'*, *shell=True*, *env=None*, *sudo=False*, *ignore_bg_processes=False*)

Because commands module is removed in Python3 and it redirect stderr to stdout, we port commands.getstatusoutput to make code compatible Return (status, output) of executing cmd in a shell.

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 record the output from this process (from stdout and stderr) in the test's output record files. Valid values: 'stdout', for standard output *only*, 'stderr' for standard error *only*, 'both' for both standard output and error in separate files, 'combined' for standard output and error in a single file, and 'none' to disable all recording. 'all' is also a valid, but deprecated, option that is a synonym of 'both'. If an explicit value is not given to this parameter, that is, if None is given, it defaults to using the module level configuration, as set by *OUTPUT_CHECK_RECORD_MODE*. If the module level configuration itself is not set, it defaults to 'none'.
- **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

Returns Exit status and command output(stdout and stderr).

Return type tuple

avocado.utils.process.**has_capability**(*capability*, *pid=None*)

Checks if a process has a given capability.

This is a simple wrapper around getpcaps, part of the libcap package. In case the getpcaps command is not available, the capability will be considered *not* to be available.

Parameters **capability** (*str*) – the name of the capability, refer to capabilities(7) man page for more information.

Returns whether the capability is available or not

Return type bool

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=None*, *send_sigcont=True*, *timeout=0*)

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, defaults to `signal.SIGKILL`
- **send_sigcont** – Send SIGCONT to allow killing stopped processes
- **timeout** – How long to wait for the pid(s) to die (negative=infinity, 0=don't wait, positive=number_of_seconds)

Returns list of all PIDs we sent signal to

Return type list

`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=None*, *shell=False*, *env=None*, *sudo=False*, *ignore_bg_processes=False*, *encoding=None*)

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 record the output from this process (from stdout and stderr) in the test's output record files. Valid values: 'stdout', for standard output *only*, 'stderr' for standard error *only*, 'both' for both standard output and error in separate files, 'combined' for standard output and error in a single file, and 'none' to disable all recording. 'all' is also a valid, but deprecated, option that is a synonym of 'both'. If an explicit value is not given to this parameter, that is, if None is given, it defaults to using the module level configuration, as set by *OUTPUT_CHECK_RECORD_MODE*. If the module level configuration itself is not set, it defaults to 'none'.
- **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.

- **encoding** (*str*) – the encoding to use for the text representation of the command result stdout and stderr, by default *avocado.utils.astring.ENCODING*

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_wrapper`**(*cmd*)
Whether 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.`**`system`**(*cmd*, *timeout=None*, *verbose=True*, *ignore_status=False*, *allow_output_check=None*, *shell=False*, *env=None*, *sudo=False*, *ignore_bg_processes=False*, *encoding=None*)
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 record the output from this process (from stdout and stderr) in the test's output record files. Valid values: 'stdout', for standard output *only*, 'stderr' for standard error *only*, 'both' for both standard output and error in separate files, 'combined' for standard output and error in a single file, and 'none' to disable all recording. 'all' is also a valid, but deprecated, option that is a synonym of 'both'. If an explicit value is not given to this parameter, that is, if None is given, it defaults to using the module level configuration, as set by *OUTPUT_CHECK_RECORD_MODE*. If the module level configuration itself is not set, it defaults to 'none'.
- **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.
- **encoding** (*str*) – the encoding to use for the text representation of the command result stdout and stderr, by default *avocado.utils.astring.ENCODING*

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=None*, *shell=False*, *env=None*, *sudo=False*, *ignore_bg_processes=False*, *strip_trail_nl=True*, *encoding=None*)

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 record the output from this process (from stdout and stderr) in the test's output record files. Valid values: 'stdout', for standard output *only*, 'stderr' for standard error *only*, 'both' for both standard output and error in separate files, 'combined' for standard output and error in a single file, and 'none' to disable all recording. 'all' is also a valid, but deprecated, option that is a synonym of 'both'. If an explicit value is not given to this parameter, that is, if None is given, it defaults to using the module level configuration, as set by *OUTPUT_CHECK_RECORD_MODE*. If the module level configuration itself is not set, it defaults to 'none'.
- **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
- **encoding** (*str*) – the encoding to use for the text representation of the command result stdout and stderr, by default *avocado.utils.astring.ENCODING*

Returns Command output.

Return type bytes

Raise *CmdError*, if `ignore_status=False`.

10.3.41 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"

avocado.utils.script.**READ_ONLY_MODE = 292**

What is commonly known as "0444" or "u=r,g=r,o=r"

class avocado.utils.script.**Script**(*path*, *content*, *mode=509*, *open_mode='w'*)
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*, *open_mode='w'*)
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()
Remove script from the file system.

Returns *True* if script has been removed, otherwise *False*.

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.

10.3.42 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.

Example of use:

```
# 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 service manager for a specific service.

Example of use:

```
# 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>*)
Internal function to determine what executable is PID 1

It does that by checking /proc/1/exe. Fall back to checking /proc/1/cmdline (local execution).

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.

Example of use:

```
# 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 service manager for a specific service.

Example of use:

```
# 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.

10.3.43 avocado.utils.softwareraid module

This module provides APIs to work with software raid.

class avocado.utils.softwareraid.**SoftwareRaid**(*name*, *level*, *disks*, *metadata*, *spare_disks=None*)

Bases: object

Perform software raid related operations.

Parameters

- **name** (*str*) – Name of the software raid to be created
- **level** – Level of software raid to be created
- **disks** (*list*) – List of disks for software raid
- **metadata** (*str*) – Metadata level for software raid
- **spare_disks** (*list*) – List of spare disks for software raid

add_disk(*disk*)

Adds disk specified to software raid.

Parameters **disk** (*str*) – disk to be added.

Returns True if add is successful, False otherwise.

Return type bool

assemble()

Assembles software raid.

Returns True if assembled, False otherwise.

Return type bool

clear_superblock()

Zeroes superblocks in member devices of raid.

Returns True if zeroed, False otherwise.

Return type bool

create()

Creates software raid.

Returns True if raid is created. False otherwise.

Return type bool

get_detail()

Returns mdadm details.

Returns mdadm –detail output

Return type str

is_recovering()

Checks if raid is recovering.

Returns True if recovering, False otherwise.

Return type bool

remove_disk(*disk*)

Removes disk specified from software raid.

Parameters **disk** (*str*) – disk to be removed.

Returns True if remove is successful, False otherwise.

Return type bool

stop()
Stops software raid.

Returns True if stopped, False otherwise.

Return type bool

10.3.44 avocado.utils.ssh module

Provides utilities to carry out an SSH session.

Example of use:

```
from avocado.utils import ssh

with ssh.Session(host, user='root', key='/path/to/file') as session:
    result = session.cmd('ls')
    if result.exit_status == 0:
        print(result.stdout_text)
```

exception avocado.utils.ssh.**NWException**
Bases: Exception

Base Exception Class for all exceptions

avocado.utils.ssh.**SSH_CLIENT_BINARY = '/usr/bin/ssh'**
The SSH client binary to use, if one is found in the system

class avocado.utils.ssh.**Session**(*host*, *port=None*, *user=None*, *key=None*, *password=None*)
Bases: object

Represents an SSH session to a remote system, for the purpose of executing commands remotely.

Session is also a context manager. On entering the context it tries to establish the connection, therefore on exiting that connection is closed.

Parameters

- **host** (*str*) – a host name or IP address
- **port** (*int*) – port number
- **user** (*str*) – the name of the remote user
- **key** (*str*) – path to a key for authentication purpose
- **password** (*str*) – password for authentication purpose

DEFAULT_OPTIONS = (('StrictHostKeyChecking', 'no'), ('UpdateHostKeys', 'no'), ('Contro

MASTER_OPTIONS = (('ControlMaster', 'yes'), ('ControlPersist', 'yes'))

cleanup_master()
Removes master file if exists.

cmd(*command*, *ignore_status=True*)
Runs a command over the SSH session

Parameters

- **command** (*str*) – the command to execute over the SSH session
- **ignore_status** (*bool*) – Whether to check the operation failed or not. If set to False then it raises an *avocado.utils.process.CmdError* exception in case of either the command or ssh connection returned with exit status other than zero.

Returns The command result object.

Return type A *avocado.utils.process.CmdResult* instance.

connect()
Establishes the connection to the remote endpoint

On this implementation, it means creating the master connection, which is a process that will live while and be used for subsequent commands.

Returns whether the connection is successfully established

Return type bool

control_master

copy_files(*source*, *destination*, *recursive=False*)
Copy Files to and from remote through scp session.

Parameters

- **source** – Source file
- **destination** – Destination file location
- **recursive** – Scp option for copy file. if set to True copy files inside directory recursively.

Type str

Type str

Type bool

Returns True if success and an exception if not.

Return type bool

get_raw_ssh_command(*command*)
Returns the raw command that will be executed locally

This should only be used if you need to interact with the ssh subprocess, and most users will *NOT* need to. Try to use the *cmd()* method instead.

Parameters **command** (*str*) – the command to execute over the SSH session

Returns The raw SSH command, that can be executed locally for the execution of a remote command.

Return type str

quit()
Attempts to gracefully end the session, by finishing the master process

Returns if closing the session was successful or not

Return type bool

10.3.45 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()

10.3.46 avocado.utils.sysinfo module

class avocado.utils.sysinfo.**Collectible**(*log_path*)
Bases: abc.ABC

Abstract class for representing sysinfo collectibles.

collect()

name

exception avocado.utils.sysinfo.**CollectibleException**
Bases: Exception

Base exception for all collectible errors.

class avocado.utils.sysinfo.**Command**(*cmd*, *timeout=-1*, *locale='C'*)
Bases: *avocado.utils.sysinfo.Collectible*

Collectible command.

Parameters

- **cmd** – String with the command.
- **timeout** – Timeout for command execution.
- **locale** – Force LANG for sysinfo collection

collect()
Execute the command as a subprocess and returns it's output. :raise CollectibleException

class avocado.utils.sysinfo.**Daemon**(**args*, ***kwargs*)
Bases: *avocado.utils.sysinfo.Command*

Collectible daemon.

Parameters

- **cmd** – String with the command.
- **timeout** – Timeout for command execution.
- **locale** – Force LANG for sysinfo collection

collect()
Stop daemon execution and returns it's logs. :raise OSError

run()
Start running the daemon as a subprocess. :raise CollectibleException

class avocado.utils.sysinfo.**JournalctlWatcher**(*log_path=None*)
Bases: *avocado.utils.sysinfo.Collectible*

Track the content of systemd journal.

Parameters **log_path** – Basename of the file where output is logged (optional).

collect()
Returns the content of systemd journal :raise CollectibleException

class avocado.utils.sysinfo.**LogWatcher**(*path*, *log_path=None*)
Bases: *avocado.utils.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.
- **log_path** – Basename of the file where output is logged (optional).

collect()
Collect all of the new data present in the log file. :raise CollectibleException

class avocado.utils.sysinfo.**Logfile**(*path*, *log_path=None*)
Bases: *avocado.utils.sysinfo.Collectible*

Collectible system file.

Parameters

- **path** – Path to the log file.
- **log_path** – Basename of the file where output is logged (optional).

collect()
Reads the log file. :raise CollectibleException

10.3.47 avocado.utils.vmimage module

Provides VM images acquired from official repositories

class avocado.utils.vmimage.**CentOSImageProvider**(*version='[0-9]+'*, *build='[0-9]{4}'*, *arch='x86_64'*)
Bases: *avocado.utils.vmimage.ImageProviderBase*

CentOS Image Provider

get_image_url()
Probes the higher image available for the current parameters.

name = 'CentOS'

class avocado.utils.vmimage.**CirrOSImageProvider**(*version='[0-9]+\.[0-9]+\.[0-9]+'*, *build=None*, *arch='x86_64'*)
Bases: *avocado.utils.vmimage.ImageProviderBase*

CirrOS Image Provider

CirrOS is a Tiny OS that specializes in running on a cloud.

name = 'CirrOS'

class avocado.utils.vmimage.**DebianImageProvider**(*version='[0-9]+.[0-9]+.[0-9]+.*'*, *build=None*, *arch='x86_64'*)
Bases: *avocado.utils.vmimage.ImageProviderBase*

Debian Image Provider

name = 'Debian'

class avocado.utils.vmimage.**FedoraImageProvider**(*version='[0-9]+'*, *build='[0-9]+.[0-9]+'*, *arch='x86_64'*)
Bases: *avocado.utils.vmimage.FedoraImageProviderBase*

Fedora Image Provider

name = 'Fedora'

class avocado.utils.vmimage.**FedoraImageProviderBase**(*version*, *build*, *arch*)
Bases: *avocado.utils.vmimage.ImageProviderBase*

Base Fedora Image Provider

HTML_ENCODING = 'iso-8859-1'

get_image_url()
Probes the higher image available for the current parameters.

url_old_images = None

class avocado.utils.vmimage.**FedoraSecondaryImageProvider**(*version='[0-9]+'*, *build='[0-9]+.[0-9]+'*, *arch='x86_64'*)
Bases: *avocado.utils.vmimage.FedoraImageProviderBase*

Fedora Secondary Image Provider

name = 'FedoraSecondary'

avocado.utils.vmimage.**IMAGE_PROVIDERS = {<class 'avocado.utils.vmimage.OpenSUSEImageProvide**

List of available providers classes

class avocado.utils.vmimage.**Image**(*name*, *url*, *version*, *arch*, *build*, *checksum*, *algorithm*, *cache_dir*, *snapshot_dir=None*)

Bases: object

Creates an instance of Image class.

Parameters

- **name** (*str*) – Name of image.
- **url** (*str*) – The url where the image can be fetched from.
- **version** (*int*) – Version of image.
- **arch** (*str*) – Architecture of the system image.
- **build** (*str*) – Build of the system image.
- **checksum** (*str*) – Hash of the system image to match after download.
- **algorithm** (*str*) – Hash type, used when the checksum is provided.
- **cache_dir** (*str or iterable*) – Local system path where the base images will be held.
- **snapshot_dir** (*str*) – Local system path where the snapshot images will be held. Defaults to cache_dir if none is given.

base_image

download()

classmethod from_parameters(*name=None*, *version=None*, *build=None*, *arch=None*, *checksum=None*, *algorithm=None*, *cache_dir=None*, *snapshot_dir=None*)

Returns an Image, according to the parameters provided.

Parameters

- **name** – (optional) Name of the Image Provider, usually matches the distro name.
- **version** – (optional) Version of the system image.
- **build** – (optional) Build number of the system image.
- **arch** – (optional) Architecture of the system image.
- **checksum** – (optional) Hash of the system image to match after download.
- **algorithm** – (optional) Hash type, used when the checksum is provided.
- **cache_dir** – (optional) Local system path where the base images will be held.
- **snapshot_dir** – (optional) Local system path where the snapshot images will be held. Defaults to cache_dir if none is given.

Returns Image instance that can provide the image according to the parameters.

get()

path

class avocado.utils.vmimage.**ImageProviderBase**(*version*, *build*, *arch*)
Bases: object

Base class to define the common methods and attributes of an image. Intended to be sub-classed by the specific image providers.

HTML_ENCODING = 'utf-8'

file_name

static get_best_version(*versions*)

get_image_parameters(*image_file_name*)
Computation of image parameters from image_pattern

Parameters **image_file_name** (*str*) – pattern with parameters

Returns dict with parameters

Return type dict or None

get_image_url()
Probes the higher image available for the current parameters.

get_version()
Probes the higher version available for the current parameters.

get_versions()
Return all available versions for the current parameters.

version

version_pattern

exception avocado.utils.vmimage.**ImageProviderError**
Bases: Exception

Generic error class for ImageProvider

class avocado.utils.vmimage.**JeosImageProvider**(*version='[0-9]+'*, *build=None*, *arch='x86_64'*)
Bases: *avocado.utils.vmimage.ImageProviderBase*

JeOS Image Provider

name = 'JeOS'

class avocado.utils.vmimage.**OpenSUSEImageProvider**(*version='[0-9]{2}.[0-9]{1}'*, *build=None*, *arch='x86_64'*)
Bases: *avocado.utils.vmimage.ImageProviderBase*

OpenSUSE Image Provider

HTML_ENCODING = 'iso-8859-1'

get_best_version(*versions*)

get_versions()
Return all available versions for the current parameters.

name = 'OpenSUSE'

version_pattern

avocado.utils.vmimage.**QEMU_IMG = None**
The "qemu-img" binary used when creating the snapshot images. If set to None (the default), it will attempt to

find a suitable binary with *avocado.utils.path.find_command()*, which uses the the system's PATH environment variable

class avocado.utils.vmimage.**UbuntuImageProvider**(*version='[0-9]+.[0-9]+'*, *build=None*, *arch='x86_64'*)

Bases: *avocado.utils.vmimage.ImageProviderBase*

Ubuntu Image Provider

get_versions()

Return all available versions for the current parameters.

name = 'Ubuntu'

class avocado.utils.vmimage.**VMImageHtmlParser**(*pattern*)

Bases: html.parser.HTMLParser

Custom HTML parser to extract the href items that match a given pattern

handle_starttag(*tag*, *attrs*)

avocado.utils.vmimage.**get**(*name=None*, *version=None*, *build=None*, *arch=None*, *checksum=None*, *algorithm=None*, *cache_dir=None*, *snapshot_dir=None*)

This method is deprecated. Use Image.from_parameters().

avocado.utils.vmimage.**get_best_provider**(*name=None*, *version=None*, *build=None*, *arch=None*)

Wrapper to get parameters of the best Image Provider, according to the parameters provided.

Parameters

- **name** – (optional) Name of the Image Provider, usually matches the distro name.
- **version** – (optional) Version of the system image.
- **build** – (optional) Build number of the system image.
- **arch** – (optional) Architecture of the system image.

Returns Image Provider

avocado.utils.vmimage.**list_providers**()

List the available Image Providers

10.3.48 avocado.utils.wait module

avocado.utils.wait.**wait_for**(*func*, *timeout*, *first=0.0*, *step=1.0*, *text=None*, *args=None*, *kwargs=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
- **step** – Time to sleep between attempts in seconds
- **text** – Text to print while waiting, for debug purposes
- **args** – Positional arguments to func
- **kwargs** – Keyword arguments to func

10.3.49 Module contents

10.4 Extension (plugin) APIs

Extension APIs that may be of interest to plugin writers.

10.4.1 Subpackages

avocado.plugins.legacy package

Submodules

avocado.plugins.legacy.replay module

class avocado.plugins.legacy.replay.**Replay**
Bases: *avocado.core.plugin_interfaces.CLI*

Replay a job

configure(*parser*)
Configures the command line parser with options specific to this plugin.

description = "Replay options for 'run' subcommand"

static load_config(*resultsdir*)

name = 'replay'

run(*config*)
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*.

Module contents

avocado.plugins.spawners package

Submodules

avocado.plugins.spawners.podman module

class avocado.plugins.spawners.podman.**PodmanCLI**
Bases: *avocado.core.plugin_interfaces.CLI*

configure(*parser*)
Configures the command line parser with options specific to this plugin.

description = 'podman spawner command line options for "run"'

name = 'podman'

run(*config*)
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.plugins.spawners.podman.**PodmanSpawner**(*config=None*)
Bases: *avocado.core.plugin_interfaces.Spawner*, *avocado.core.spawners.common.SpawnerMixin*

METHODS = [<SpawnMethod.STANDALONE_EXECUTABLE: <object object>>]

static check_task_requirements(*runtime_task*)
Check the runtime task requirements needed to be able to run

description = 'Podman (container) based spawner'

is_task_alive(*runtime_task*)
Determines if a task is alive or not.

Parameters **runtime_task** (*avocado.core.task.runtime.RuntimeTask*) – wrapper for a Task with additional runtime information

spawn_task(*runtime_task*)
Spawns a task return whether the spawning was successful.

Parameters **runtime_task** (*avocado.core.task.runtime.RuntimeTask*) – wrapper for a Task with additional runtime information

wait_task(*runtime_task*)
Waits for a task to finish.

Parameters **runtime_task** (*avocado.core.task.runtime.RuntimeTask*) – wrapper for a Task with additional runtime information

class avocado.plugins.spawners.podman.**PodmanSpawnerInit**
Bases: *avocado.core.plugin_interfaces.Init*

description = 'Podman (container) based spawner initialization'

initialize()
Entry point for the plugin to perform its initialization.

avocado.plugins.spawners.process module

class avocado.plugins.spawners.process.**ProcessSpawner**(*config=None*)
Bases: *avocado.core.plugin_interfaces.Spawner*, *avocado.core.spawners.common.SpawnerMixin*

METHODS = [<SpawnMethod.STANDALONE_EXECUTABLE: <object object>>]

static check_task_requirements(*runtime_task*)
Check the runtime task requirements needed to be able to run

description = 'Process based spawner'

static is_task_alive(*runtime_task*)
Determines if a task is alive or not.

Parameters **runtime_task** (*avocado.core.task.runtime.RuntimeTask*) – wrapper for a Task with additional runtime information

spawn_task(*runtime_task*)
Spawns a task return whether the spawning was successful.

Parameters **runtime_task** (*avocado.core.task.runtime.RuntimeTask*) – wrapper for a Task with additional runtime information

static wait_task(*runtime_task*)
Waits for a task to finish.

Parameters **runtime_task** (*avocado.core.task.runtime.RuntimeTask*) – wrapper for a Task with additional runtime information

Module contents

10.4.2 Submodules

10.4.3 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*)
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.plugins.archive.**ArchiveCLI**
Bases: *avocado.core.plugin_interfaces.CLI*

configure(*parser*)
Configures the command line parser with options specific to this plugin.

description = 'Result archive (ZIP) support to run command'

name = 'zip_archive'

run(*config*)
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*.

10.4.4 avocado.plugins.assets module

Assets subcommand

class avocado.plugins.assets.**Assets**
Bases: *avocado.core.plugin_interfaces.CLICmd*

Implements the avocado 'assets' subcommand

configure(*parser*)
Add the subparser for the assets action.

Parameters **parser** (*avocado.core.parser.ArgumentParser*) – The Avocado command line application parser

description = 'Manage assets'

static handle_fetch(*config*)

handle_list(*config*)

handle_purge(*config*)

static handle_register(*config*)

name = 'assets'

run(*config*)
Entry point for actually running the command.

class avocado.plugins.assets.**FetchAssetHandler**(*file_name*, *klass=None*, *method=None*)
Bases: ast.NodeVisitor

Handles the parsing of instrumented tests for *fetch_asset* statements.

PATTERN = 'fetch_asset'

visit_Assign(*node*)
Visit Assign on AST and build assignments.

This method will visit and build list of assignments that matches the pattern pattern *name* = *string*.

Parameters **node** (*ast.**) – AST node to be evaluated

visit_Call(*node*)
Visit Calls on AST and build list of calls that matches the pattern.

Parameters **node** (*ast.**) – AST node to be evaluated

visit_ClassDef(*node*)
Visit ClassDef on AST and save current Class.

Parameters **node** (*ast.**) – AST node to be evaluated

visit_FunctionDef(*node*)
Visit FunctionDef on AST and save current method.

Parameters **node** (*ast.**) – AST node to be evaluated

class avocado.plugins.assets.**FetchAssetJob**(*config=None*)
Bases: *avocado.core.plugin_interfaces.JobPreTests*

Implements the assets fetch job pre tests.

This has the same effect of running the 'avocado assets fetch INSTRUMENTED', but it runs during the test execution, before the actual test starts.

description = 'Fetch assets before the test run'

name = 'fetchasset'

pre_tests(*job*)
Entry point for job running actions before tests execution.

avocado.plugins.assets.**fetch_assets**(*test_file*, *klass=None*, *method=None*, *logger=None*)
Fetches the assets based on keywords listed on FetchAssetHandler.calls.

Parameters **test_file** – File name of instrumented test to be evaluated :type test_file: str

Returns list of names that were successfully fetched and list of fails.

10.4.5 avocado.plugins.config module

class avocado.plugins.config.**Config**
Bases: *avocado.core.plugin_interfaces.CLICmd*

Implements the avocado 'config' subcommand

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 = 'Shows avocado config keys'

static handle_default()

static handle_reference(*print_function*)

name = 'config'

run(*config*)
Entry point for actually running the command.

10.4.6 avocado.plugins.dict_variants module

class avocado.plugins.dict_variants.**DictVariants**
Bases: *avocado.core.plugin_interfaces.Varianter*

Turns (a list of) Python dictionaries into variants

description = 'Python Dictionary based varianter'

initialize(*config*)

name = 'dict_variants'

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

class avocado.plugins.dict_variants.**DictVariantsInit**
Bases: *avocado.core.plugin_interfaces.Init*

description = 'Python Dictionary based varianter'

initialize()
Entry point for the plugin to perform its initialization.

name = 'dict_variants'

10.4.7 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** (*avocado.core.parser.ArgumentParser*) – The Avocado command line application parser

description = 'Shows the difference between 2 jobs.'

name = 'diff'

run(*config*)
Entry point for actually running the command.

10.4.8 avocado.plugins.distro module

avocado.plugins.distro.**DISTRO_PKG_INFO_LOADERS = {'deb': <class 'avocado.plugins.distro.D**
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*)
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 = 'Shows detected Linux distribution'

name = 'distro'

run(*config*)
Entry point for actually running the command.

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*)
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

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.**DistroPkgInfoLoaderRpm**(*path*)
Bases: *avocado.plugins.distro.DistroPkgInfoLoader*

Loads package information for RPM files

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

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

10.4.9 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(*config*)
Print libexec path and finish

Parameters **config** – job configuration

10.4.10 avocado.plugins.expected_files_merge module

Functions for merging equal expected files together

class avocado.plugins.expected_files_merge.**FilesMerge**
Bases: *avocado.core.plugin_interfaces.JobPost*

Plugin for merging equal expected files together

description = 'Merge of equal expected files'

name = 'merge'

post(*job*)
Entry point for actually running the post job action.

avocado.plugins.expected_files_merge.**merge_expected_files**(*references*)
Cascade merge of equal expected files in job references from variant level to file level :param references: list of job references :type references: list

10.4.11 avocado.plugins.human module

Human result UI

avocado.plugins.human.**COMPLETE_STATUSES = ['SKIP', 'ERROR', 'FAIL', 'WARN', 'PASS', 'INTERR**
STATUSES contains the finished status, but lacks the novel concept of nrunner having tests STARTED (that is, in progress). This contains a more complete list of statuses that includes "STARTED"

class avocado.plugins.human.**Human**(*config*)
Bases: *avocado.core.plugin_interfaces.ResultEvents*

Human result UI

description = 'Human Interface UI'

end_test(*result*, *state*)
Event triggered when a test finishes running.

static get_colored_status(*status*, *extra=None*)

name = 'human'

post_tests(*job*)
Entry point for job running actions after the tests execution.

pre_tests(*job*)
Entry point for job running actions before tests execution.

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.plugins.human.**HumanInit**
Bases: *avocado.core.plugin_interfaces.Init*

description = 'Initialize human ui plugin settings'

initialize()
Entry point for the plugin to perform its initialization.

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*)
Entry point for actually running the post job action.

pre(*job*)
Entry point for actually running the pre job action.

10.4.12 avocado.plugins.jobs module

Jobs subcommand

class avocado.plugins.jobs.**Jobs**
Bases: *avocado.core.plugin_interfaces.CLICmd*

Implements the avocado 'jobs' subcommand

configure(*parser*)
Add the subparser for the assets action.

Parameters **parser** (*avocado.core.parser.ArgumentParser*) – The Avocado command line application parser

description = 'Manage Avocado jobs'

static handle_list_command(*jobs_results*)
Called when 'avocado jobs list' command is executed.

handle_output_files_command(*config*)
Called when 'avocado jobs get-output-files' command is executed.

handle_show_command(*config*)
Called when 'avocado jobs show' command is executed.

name = 'jobs'

run(*config*)
Entry point for actually running the command.

10.4.13 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*)
Entry point for actually running the post job action.

pre(*job*)
Entry point for actually running the pre job action.

class avocado.plugins.jobscripts.**JobScriptsInit**
Bases: *avocado.core.plugin_interfaces.Init*

description = 'Jobscripts plugin initialization'

initialize()
Entry point for the plugin to perform its initialization.

name = 'jobscripts-init'

10.4.14 avocado.plugins.journal module

Journal Plugin

class avocado.plugins.journal.**Journal**
Bases: *avocado.core.plugin_interfaces.CLI*

Test journal

configure(*parser*)
Configures the command line parser with options specific to this plugin.

description = "Journal options for the 'run' subcommand"

name = 'journal'

run(*config*)
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.plugins.journal.**JournalResult**(*config*)
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*)
Event triggered when a test finishes running.

lazy_init_journal(*state*)

name = 'journal'

post_tests(*job*)
Entry point for job running actions after the tests execution.

pre_tests(*job*)
Entry point for job running actions before tests execution.

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.

10.4.15 avocado.plugins.json_variants module

class avocado.plugins.json_variants.**JsonVariants**
Bases: *avocado.core.plugin_interfaces.Varianter*

Processes the serialized file into variants

description = 'JSON serialized based Varianter'

initialize(*config*)

name = 'json variants'

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

variants = None

class avocado.plugins.json_variants.**JsonVariantsCLI**
Bases: *avocado.core.plugin_interfaces.CLI*

Serialized based Varianter options

configure(*parser*)
Configures the command line parser with options specific to this plugin.

description = "JSON serialized based Varianter options for the 'run' subcommand"

name = 'json variants'

run(*config*)
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.plugins.json_variants.**JsonVariantsInit**
Bases: *avocado.core.plugin_interfaces.Init*

description = 'JSON serialized based varianter initialization'

initialize()
Entry point for the plugin to perform its initialization.

name = 'json_variants'

10.4.16 avocado.plugins.jsonresult module

JSON output module.

class avocado.plugins.jsonresult.**JSONCLI**
Bases: *avocado.core.plugin_interfaces.CLI*

JSON output

configure(*parser*)
Configures the command line parser with options specific to this plugin.

description = "JSON output options for 'run' command"

name = 'json'

run(*config*)
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.plugins.jsonresult.**JSONInit**
Bases: *avocado.core.plugin_interfaces.Init*

description = 'JSON job result plugin initialization'

initialize()
Entry point for the plugin to perform its initialization.

name = 'json'

class avocado.plugins.jsonresult.**JSONResult**
Bases: *avocado.core.plugin_interfaces.Result*

description = 'JSON result support'

name = 'json'

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

10.4.17 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** (*avocado.core.parser.ArgumentParser*) – The Avocado command line application parser

description = 'List available tests'

name = 'list'

run(*config*)
Entry point for actually running the command.

static save_recipes(*suite*, *directory*, *matrix_len*)

10.4.18 avocado.plugins.plugins module

Plugins information plugin

class avocado.plugins.plugins.**Plugins**
Bases: *avocado.core.plugin_interfaces.CLICmd*

Plugins information

description = 'Displays plugin information'

name = 'plugins'

run (*config*)
Entry point for actually running the command.

10.4.19 avocado.plugins.replay module

Replay Job Plugin

class avocado.plugins.replay.**Replay**
Bases: *avocado.core.plugin_interfaces.CLICmd*

Implements the avocado 'replay' subcommand.

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 = 'Runs a new job using a previous job as its configuration'

name = 'replay'

run (*config*)
Entry point for actually running the command.

10.4.20 avocado.plugins.resolvers module

Test resolver for builtin test types

class avocado.plugins.resolvers.**AvocadoInstrumentedResolver**(*config=None*)
Bases: *avocado.core.plugin_interfaces.Resolver*

description = 'Test resolver for Avocado Instrumented tests'

name = 'avocado-instrumented'

resolve (*reference*)
Resolves the given reference into a reference resolution.

Parameters **reference** (*str*) – a specification that can eventually be resolved into a test (in the form of a *avocado.core.nrunner.Runnable*)

Returns the result of the resolution process, containing the success, failure or error, along with zero or more *avocado.core.nrunner.Runnable* objects

Return type *avocado.core.resolver.ReferenceResolution*

class avocado.plugins.resolvers.**ExecTestResolver**(*config=None*)
Bases: *avocado.core.plugin_interfaces.Resolver*

description = 'Test resolver for executable files to be handled as tests'

name = 'exec-test'

resolve (*reference*)
Resolves the given reference into a reference resolution.

Parameters **reference** (*str*) – a specification that can eventually be resolved into a test (in the form of a *avocado.core.nrunner.Runnable*)

Returns the result of the resolution process, containing the success, failure or error, along with zero or more *avocado.core.nrunner.Runnable* objects

Return type *avocado.core.resolver.ReferenceResolution*

class avocado.plugins.resolvers.**PythonUnittestResolver**(*config=None*)
Bases: *avocado.core.plugin_interfaces.Resolver*

description = 'Test resolver for Python Unittests'

name = 'python-unittest'

resolve(*reference*)
Resolves the given reference into a reference resolution.

Parameters **reference** (*str*) – a specification that can eventually be resolved into a test (in the form of a *avocado.core.nrunner.Runnable*)

Returns the result of the resolution process, containing the success, failure or error, along with zero or more *avocado.core.nrunner.Runnable* objects

Return type *avocado.core.resolver.ReferenceResolution*

class avocado.plugins.resolvers.**TapResolver**(*config=None*)
Bases: *avocado.core.plugin_interfaces.Resolver*

description = 'Test resolver for executable files to be handled as TAP tests'

name = 'tap'

resolve(*reference*)
Resolves the given reference into a reference resolution.

Parameters **reference** (*str*) – a specification that can eventually be resolved into a test (in the form of a *avocado.core.nrunner.Runnable*)

Returns the result of the resolution process, containing the success, failure or error, along with zero or more *avocado.core.nrunner.Runnable* objects

Return type *avocado.core.resolver.ReferenceResolution*

avocado.plugins.resolvers.**python_resolver**(*name*, *reference*, *find_tests*, *config*)

10.4.21 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(*config*)
Run test modules or simple tests.

Parameters **config** (*dict*) – Configuration received from command line parser and possibly other sources.

class avocado.plugins.run.**RunInit**
Bases: *avocado.core.plugin_interfaces.Init*

description = 'Initializes the run options'

initialize()
Entry point for the plugin to perform its initialization.

name = 'run'

10.4.22 avocado.plugins.runner module

Conventional Test Runner Plugin

class avocado.plugins.runner.**TestRunner**
Bases: *avocado.core.plugin_interfaces.Runner*

A test runner class that displays tests results.

Creates an instance of TestRunner class.

DEFAULT_TIMEOUT = 86400

description = 'The conventional test runner'

name = 'runner'

run_suite(*job*, *test_suite*)
Run one or more tests and report with test result.

Parameters

- **job** – an instance of *avocado.core.job.Job*.
- **test_suite** – a list of tests to run.

Returns a set with types of test failures.

run_test(*job*, *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.

10.4.23 avocado.plugins.runner_nrunner module

NRunner based implementation of job compliant runner

class avocado.plugins.runner_nrunner.**Runner**
Bases: *avocado.core.plugin_interfaces.Runner*

description = 'nrunner based implementation of job compliant runner'

name = 'nrunner'

run_suite(*job*, *test_suite*)
Run one or more tests and report with test result.

Parameters

- **job** – an instance of *avocado.core.job.Job*.
- **test_suite** – an instance of TestSuite with some tests to run.

Returns a set with types of test failures.

class avocado.plugins.runner_nrunner.**RunnerCLI**
Bases: *avocado.core.plugin_interfaces.CLI*

configure(*parser*)
Configures the command line parser with options specific to this plugin.

description = 'nrunner command line options for "run"'

name = 'nrunner'

run(*config*)
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.plugins.runner_nrunner.**RunnerInit**
Bases: *avocado.core.plugin_interfaces.Init*

description = 'nrunner initialization'

initialize()
Entry point for the plugin to perform its initialization.

name = 'nrunner'

10.4.24 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** (*avocado.core.parser.ArgumentParser*) – The Avocado command line application parser

description = 'Collect system information'

name = 'sysinfo'

run(*config*)
Entry point for actually running the command.

class avocado.plugins.sysinfo.**SysInfoJob**(*config*)
Bases: *avocado.core.plugin_interfaces.JobPreTests*, *avocado.core.plugin_interfaces.JobPostTests*

description = 'Collects system information before/after the job is run'

name = 'sysinfo'

post_tests(*job*)
Entry point for job running actions after the tests execution.

pre_tests(*job*)
Entry point for job running actions before tests execution.

class avocado.plugins.sysinfo.**SysinfoInit**
Bases: *avocado.core.plugin_interfaces.Init*

description = 'Initializes sysinfo settings'

initialize()
Entry point for the plugin to perform its initialization.

name = 'sysinfo'

10.4.25 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*)
Configures the command line parser with options specific to this plugin.

description = 'TAP - Test Anything Protocol results'

name = 'TAP'

run(*config*)
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.plugins.tap.**TAPInit**
Bases: *avocado.core.plugin_interfaces.Init*

description = 'TAP - Test Anything Protocol - result plugin initialization'

initialize()
Entry point for the plugin to perform its initialization.

name = 'TAP'

class avocado.plugins.tap.**TAPResult**(*config*)
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*)
Entry point for job running actions after the tests execution.

pre_tests(*job*)
Log the test plan

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.

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

10.4.26 avocado.plugins.testlogs module

class avocado.plugins.testlogs.**TestLogging**(*config*)
Bases: *avocado.core.plugin_interfaces.ResultEvents*

TODO: The description should be changed when the legacy runner will be deprecated.

description = 'Nrunner specific Test logs for Job'

end_test(*result*, *state*)
Event triggered when a test finishes running.

post_tests(*job*)
Entry point for job running actions after the tests execution.

pre_tests(*job*)
Entry point for job running actions before tests execution.

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.plugins.testlogs.**TestLogsUI**
Bases: *avocado.core.plugin_interfaces.JobPre*, *avocado.core.plugin_interfaces.JobPost*

description = "Shows content from tests' logs"

post(*job*)
Entry point for actually running the post job action.

pre(*job*)
Entry point for actually running the pre job action.

class avocado.plugins.testlogs.**TestLogsUIInit**
Bases: *avocado.core.plugin_interfaces.Init*

description = 'Initialize testlogs plugin settings'

initialize()
Entry point for the plugin to perform its initialization.

10.4.27 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*)
Entry point for actually running the post job action.

pre(*job*)
Entry point for actually running the pre job action.

10.4.28 avocado.plugins.variants module

class avocado.plugins.variants.**Variants**
Bases: *avocado.core.plugin_interfaces.CLICmd*

Implements "variants" command to visualize/debug test variants and params

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 = 'Tool to analyze and visualize test variants and params'

name = 'variants'

run(*config*)
Entry point for actually running the command.

avocado.plugins.variants.**map_verbosity_level**(*level*)

10.4.29 avocado.plugins.vmimage module

class avocado.plugins.vmimage.**VMimage**
Bases: *avocado.core.plugin_interfaces.CLICmd*

Implements the avocado 'vmimage' subcommand

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 = 'Provides VM images acquired from official repositories'

name = 'vmimage'

run(*config*)
Entry point for actually running the command.

avocado.plugins.vmimage.**display_images_list**(*images*)
Displays table with information about images :param images: list with image's parameters :type images: list of dicts

avocado.plugins.vmimage.**download_image**(*distro*, *version=None*, *arch=None*)
Downloads the vmimage to the cache directory if doesn't already exist

Parameters

- **distro** (*str*) – Name of image distribution
- **version** (*str*) – Version of image
- **arch** (*str*) – Architecture of image

Raises **AttributeError** – When image can't be downloaded

Returns Information about downloaded image

Return type dict

avocado.plugins.vmimage.**list_downloaded_images**()
List the available Image inside avocado cache :return: list with image's parameters :rtype: list of dicts

10.4.30 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*)
Configures the command line parser with options specific to this plugin.

description = "Implements the '--wrapper' flag for the 'run' subcommand"

name = 'wrapper'

run(*config*)
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*.

10.4.31 avocado.plugins.xunit module

xUnit module.

class avocado.plugins.xunit.**XUnitCLI**
Bases: *avocado.core.plugin_interfaces.CLI*

xUnit output

configure(*parser*)
Configures the command line parser with options specific to this plugin.

description = 'xUnit output options'

name = 'xunit'

run(*config*)
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.plugins.xunit.**XUnitInit**
Bases: *avocado.core.plugin_interfaces.Init*

description = 'xUnit job result initialization'

initialize()
Entry point for the plugin to perform its initialization.

name = 'xunit'

class avocado.plugins.xunit.**XUnitResult**
Bases: *avocado.core.plugin_interfaces.Result*

PRINTABLE = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789!"#$%&\'()*

UNKNOWN = '<unknown>'

description = 'XUnit result support'

name = 'xunit'

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

10.4.32 Module contents

10.5 Optional Plugins API

The following pages document the private APIs of optional Avocado plugins.

10.5.1 avocado_resultsdb package

Module contents

Avocado Plugin to propagate Job results to Resultsdb

class avocado_resultsdb.**ResultsdbCLI**
Bases: *avocado.core.plugin_interfaces.CLI*

Propagate Job results to Resultsdb

configure(*parser*)
Configures the command line parser with options specific to this plugin.

description = "Resultsdb options for 'run' subcommand"

name = 'resultsdb'

run(*config*)
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_resultsdb.**ResultsdbResult**
Bases: *avocado.core.plugin_interfaces.Result*

ResultsDB render class

description = 'Resultsdb result support'

name = 'resultsdb'

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_resultsdb.**ResultsdbResultEvent**(*config*)
Bases: *avocado.core.plugin_interfaces.ResultEvents*

ResultsDB output class

description = 'Resultsdb result support'

end_test(*result*, *state*)
Create the ResultsDB result, which corresponds to one test from the Avocado Job

name = 'resultsdb'

post_tests(*job*)
Entry point for job running actions after the tests execution.

pre_tests(*job*)
Create the ResultsDB group, which corresponds to the Avocado Job

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.

10.5.2 avocado_golang package

Submodules

avocado_golang.runner module

class avocado_golang.runner.**GolangRunner**(*runnable*)
Bases: *avocado.core.nrunner.BaseRunner*

Runner for Golang tests.

When creating the Runnable, use the following attributes:

- kind: should be 'golang';
- uri: module name and optionally a test method name, separated by colon;
- args: not used
- kwargs: not used

Example:

runnable = Runnable(kind='golang', uri='countavocados:ExampleContainers')

run()
Runner main method

Yields dictionary as output, containing status as well as relevant information concerning the results.

class avocado_golang.runner.**RunnerApp**(*echo=<built-in function print>*, *prog=None*, *description=None*)
Bases: *avocado.core.nrunner.BaseRunnerApp*

PROG_DESCRIPTION = 'nrunner application for golang tests'

PROG_NAME = 'avocado-runner-golang'

RUNNABLE_KINDS_CAPABLE = {'golang': <class 'avocado_golang.runner.GolangRunner'>}

avocado_golang.runner.**main**()

Module contents

Plugin to run Golang tests in Avocado

class avocado_golang.**GolangCLI**
Bases: *avocado.core.plugin_interfaces.CLI*

Run Golang tests

configure(*parser*)
Configures the command line parser with options specific to this plugin.

description = "Golang options for 'run' subcommand"

name = 'golang'

run(*config*)
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_golang.**GolangLoader**(*config*, *extra_params*)
Bases: *avocado.core.loader.TestLoader*

Golang loader class

discover(*reference*, *which_tests=<DiscoverMode.DEFAULT: <object object>>*)
Discover (possible) tests from an reference.

Parameters

- **reference** (*str*) – the reference to be inspected.
- **which_tests** (DiscoverMode) – Limit tests to be displayed

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}

static get_type_label_mapping()
Get label mapping for display in test listing.

Returns Dict {TestClass: 'TEST_LABEL_STRING'}

name = 'golang'

class avocado_golang.**GolangResolver**(*config=None*)
Bases: *avocado.core.plugin_interfaces.Resolver*

description = 'Test resolver for Go language tests'

name = 'golang'

static resolve(*reference*)
Resolves the given reference into a reference resolution.

Parameters **reference** (*str*) – a specification that can eventually be resolved into a test (in the form of a *avocado.core.nrunner.Runnable*)

Returns the result of the resolution process, containing the success, failure or error, along with zero or more *avocado.core.nrunner.Runnable* objects

Return type *avocado.core.resolver.ReferenceResolution*

class avocado_golang.**GolangTest**(*name*, *params=None*, *base_logdir=None*, *job=None*, *sub-test=None*, *executable=None*)
Bases: *avocado.core.test.SimpleTest*

Run a Golang Test command as a SIMPLE test.

filename
Returns the path of the golang test suite.

test()
Create the Golang command and execute it.

class avocado_golang.**NotGolangTest**
Bases: object

Not a golang test (for reporting purposes)

avocado_golang.**find_files**(*path*, *recursive=True*)

avocado_golang.**find_tests**(*test_path*)

10.5.3 avocado_result_upload package

Module contents

Avocado Plugin to propagate Job results to remote host

class avocado_result_upload.**ResultUpload**
Bases: *avocado.core.plugin_interfaces.Result*

ResultsUpload output class

description = 'ResultUpload result support'

name = 'result_upload'

render(*result*, *job*)
Upload result, which corresponds to one test from the Avocado Job

if job.status == "RUNNING": return # Don't create results on unfinished jobs

class avocado_result_upload.**ResultUploadCLI**
Bases: *avocado.core.plugin_interfaces.CLI*

ResultsUpload output class

configure(*parser*)
Configures the command line parser with options specific to this plugin.

description = "ResultUpload options for 'run' subcommand"

name = 'result_upload'

run(*config*)
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*.

10.5.4 avocado_varianter_pict package

Module contents

class avocado_varianter_pict.**VarianterPict**
Bases: *avocado.core.plugin_interfaces.Varianter*

Processes the pict file into variants

description = 'PICT based Varianter'

initialize(*config*)

name = 'pict'

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

class avocado_varianter_pict.**VarianterPictCLI**
Bases: *avocado.core.plugin_interfaces.CLI*

Pict based Varianter options

configure(*parser*)
Configures the command line parser with options specific to this plugin.

description = "PICT based Varianter options for the 'run' subcommand"

name = 'pict'

run(*config*)
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*.

avocado_varianter_pict.**parse_pict_output**(*output*)

avocado_varianter_pict.**run_pict**(*binary*, *parameter_file*, *order*)

10.5.5 avocado_robot package

Submodules

avocado_robot.runner module

Avocado nrunner for Robot Framework tests

class avocado_robot.runner.**RobotRunner**(*runnable*)
Bases: *avocado.core.nrunner.BaseRunner*

run()
Runner main method

Yields dictionary as output, containing status as well as relevant information concerning the results.

class avocado_robot.runner.**RunnerApp**(*echo=<built-in function print>*, *prog=None*, *description=None*)
Bases: *avocado.core.nrunner.BaseRunnerApp*

PROG_DESCRIPTION = '*nrunner application for robot tests'

PROG_NAME = 'avocado-runner-robot'

RUNNABLE_KINDS_CAPABLE = {'robot': <class 'avocado_robot.runner.RobotRunner'>}

avocado_robot.runner.**main**()

Module contents

Plugin to run Robot Framework tests in Avocado

class avocado_robot.**NotRobotTest**
Bases: object

Not a robot test (for reporting purposes)

class avocado_robot.**RobotCLI**
Bases: *avocado.core.plugin_interfaces.CLI*

Run Robot Framework tests

configure(*parser*)
Configures the command line parser with options specific to this plugin.

description = "Robot Framework options for 'run' subcommand"

name = 'robot'

run(*config*)
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_robot.**RobotLoader**(*config*, *extra_params*)
Bases: *avocado.core.loader.TestLoader*

Robot loader class

discover(*reference*, *which_tests=<DiscoverMode.DEFAULT: <object object>>*)
Discover (possible) tests from an reference.

Parameters

- **reference** (*str*) – the reference to be inspected.
- **which_tests** (DiscoverMode) – Limit tests to be displayed

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}

static get_type_label_mapping()
Get label mapping for display in test listing.

Returns Dict {TestClass: 'TEST_LABEL_STRING'}

name = 'robot'

class avocado_robot.**RobotResolver**(*config=None*)
Bases: *avocado.core.plugin_interfaces.Resolver*

description = 'Test resolver for Robot Framework tests'

name = 'robot'

static resolve(*reference*)
Resolves the given reference into a reference resolution.

Parameters **reference** (*str*) – a specification that can eventually be resolved into a test (in the form of a *avocado.core.nrunner.Runnable*)

Returns the result of the resolution process, containing the success, failure or error, along with zero or more *avocado.core.nrunner.Runnable* objects

Return type *avocado.core.resolver.ReferenceResolution*

class avocado_robot.**RobotTest**(*name*, *params=None*, *base_logdir=None*, *config=None*, *executable=None*)
Bases: *avocado.core.test.SimpleTest*

Run a Robot command as a SIMPLE test.

filename
Returns the path of the robot test suite.

test()
Create the Robot command and execute it.

avocado_robot.**find_tests**(*reference*, *test_suite*)

10.5.6 avocado_varianter_yaml_to_mux package

Submodules

avocado_varianter_yaml_to_mux.mux module

This file contains mux-enabled implementations of parts useful for creating a custom Varianter plugin.

class avocado_varianter_yaml_to_mux.mux.**Control**(*code*, *value=None*)
Bases: object

Container used to identify node vs. control sequence

class avocado_varianter_yaml_to_mux.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*.

initialize_mux(*root*, *paths*)
Initialize the basic values

Note We can't use __init__ as this object is intended to be used via dispatcher with no __init__ arguments.

paths = None

root = None

to_str(*summary*, *variants*, ***kwargs*)
See *avocado.core.plugin_interfaces.Varianter.to_str()*

variant_ids = []

variants = None

class avocado_varianter_yaml_to_mux.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_varianter_yaml_to_mux.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()
Reports string which represents the value of this node.

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_varianter_yaml_to_mux.mux.**OutputList**(*values*, *nodes*, *yamls*)
Bases: list

List with some debug info

class avocado_varianter_yaml_to_mux.mux.**OutputValue**(*value*, *node*, *srcyaml*)
Bases: object

Ordinary value with some debug info

class avocado_varianter_yaml_to_mux.mux.**ValueDict**(*srcyaml*, *node*, *values*)
Bases: dict

Dict which stores the origin of the items

items()
Slower implementation with the use of __getitem__

avocado_varianter_yaml_to_mux.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_varianter_yaml_to_mux.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.

Module contents

Varianter plugin to parse yaml files to params

class avocado_varianter_yaml_to_mux.**ListOfNodeObjects**
Bases: list

Used to mark list as list of objects from whose node is going to be created

class avocado_varianter_yaml_to_mux.**YamlToMux**
Bases: *avocado_varianter_yaml_to_mux.mux.MuxPlugin*, *avocado.core.plugin_interfaces.Varianter*

Processes the mux options into varianter plugin

description = 'Multiplexer plugin to parse yaml files to params'

initialize(*config*)

name = 'yaml_to_mux'

class avocado_varianter_yaml_to_mux.**YamlToMuxCLI**
Bases: *avocado.core.plugin_interfaces.CLI*

Defines arguments for YamlToMux plugin

configure(*parser*)
Configures "run" and "variants" subparsers

description = "YamlToMux options for the 'run' subcommand"

name = 'yaml_to_mux'

run(*config*)
The YamlToMux varianter plugin handles these

class avocado_varianter_yaml_to_mux.**YamlToMuxInit**
Bases: *avocado.core.plugin_interfaces.Init*

YamlToMux initialization plugin

description = 'YamlToMux initialization plugin'

initialize()
Entry point for the plugin to perform its initialization.

name = 'yaml_to_mux'

avocado_varianter_yaml_to_mux.**create_from_yaml**(*paths*)
Create tree structure from yaml-like file.

Parameters **paths** – File object to be processed

Raises **SyntaxError** – When yaml-file is corrupted

Returns Root of the created tree structure

10.5.7 avocado_varianter_cit package

Submodules

avocado_varianter_cit.Cit module

class avocado_varianter_cit.Cit.**Cit**(*input_data*, *t_value*, *constraints*)
Bases: object

Creation of CombinationMatrix from user input

Parameters

- **input_data** – parameters from user
- **t_value** – size of one combination
- **constraints** – constraints of combinations

change_one_column(*matrix*)
Randomly choose one column of the matrix. In each cell of this column changes value. The row with the best coverage is the solution.

Parameters **matrix** – matrix to be changed

Returns solution, index of solution inside matrix and parameters which has been changed

change_one_value(*matrix*, *row_index=None*, *column_index=None*)
Change one cell inside the matrix

Parameters

- **matrix** – matrix to be changed
- **row_index** – row inside matrix. If it's None it is chosen randomly
- **column_index** – column inside matrix. If it's None it is chosen randomly

Returns solution, index of solution inside matrix and parameters which has been changed

compute()
Searching for the best solution. It creates one solution and from that, it tries to create smaller solution. This searching process is limited by ITERATIONS_SIZE. When ITERATIONS_SIZE is 0 the last found solution is the best solution.

Returns The best solution

compute_hamming_distance(*row*)

Returns hamming distance of row from final matrix

compute_row()
Computation of one row which covers most of combinations

Returns new solution row

compute_row_using_hamming_distance()

Returns row with the biggest hamming distance from final matrix

cover_missing_combination(*matrix*)
Randomly finds one missing combination. This combination puts into each row of the matrix. The row with the best coverage is the solution

Parameters **matrix** – matrix to be changed

Returns solution, index of solution inside matrix and parameters which has been changed

create_random_row_with_constraints()
Create a new test-case random row, and the row meets the constraints.

Returns new random row

Return type list

final_matrix_init()
Creation of the first solution. This solution is the start of searching for the best solution

Returns solution matrix (list(list))

find_better_solution(*counter*, *matrix*)
Changing the matrix to cover all combinations

Parameters

- **counter** – maximum number of changes in the matrix
- **matrix** – matrix to be changed

Returns new matrix and is changes have been successful?

get_missing_combination_random()
Randomly finds one missing combination.

Returns parameter of combination and values of combination

use_random_algorithm(*matrix*)
Applies one of these algorithms to the matrix. It chooses algorithm by random in proportion 1:1:8

Parameters **matrix** – matrix to be changed

Returns new row of matrix, index of row inside matrix and parameters which has been changed

avocado_varianter_cit.CombinationMatrix module

class avocado_varianter_cit.CombinationMatrix.**CombinationMatrix**(*input_data*, *t_value*)
Bases: object

CombinationMatrix object stores Rows of combinations into dictionary. And also stores which rows are not covered. Keys in dictionary are parameters of combinations and values are CombinationRow objects. CombinationMatrix object has information about how many combinations are uncovered and how many of them are covered more than ones.

Parameters

- **input_data** – list of data from user
- **t_value** – t number from user

cover_combination(*row*, *parameters*)
Cover combination of specific parameters by one row from possible solution

Parameters

- **row** – one row from solution
- **parameters** – parameters which has to be covered

Returns number of still uncovered combinations

cover_solution_row(*row*)
Cover all combination by one row from possible solution

Parameters **row** – one row from solution

Returns number of still uncovered combinations

del_cell(*parameters*, *combination*)
Disable one combination. If combination is disabled it means that the combination does not match the constraints

Parameters

- **parameters** – parameters whose combination is disabled
- **combination** – combination to be disabled

get_row(*key*)

Parameters **key** – identifier of row

Returns CombinationRow

is_valid_combination(*row*, *parameters*)
Is the specific parameters from solution row match the constraints.

Parameters

- **row** – one row from solution
- **parameters** – parameters from row

is_valid_solution(*row*)
Is the solution row match the constraints.

Parameters **row** – one row from solution

uncover()
Uncover all combinations

uncover_combination(*row*, *parameters*)
Uncover combination of specific parameters by one row from possible solution

Parameters

- **row** – one row from solution
- **parameters** – parameters which has to be covered

Returns number of uncovered combinations

uncover_solution_row(*row*)
Uncover all combination by one row from possible solution

Parameters **row** – one row from solution

Returns number of uncovered combinations

avocado_varianter_cit.CombinationRow module

class avocado_varianter_cit.CombinationRow.**CombinationRow**(*input_data*, *t_value*, *parameters*)

Bases: object

Row object store all combinations between two parameters into dictionary. Keys in dictionary are values of combinations and values in dictionary are information about coverage. Row object has information how many combinations are uncovered and how many of them are covered more than ones.

Parameters

- **input_data** – list of data from user
- **t_value** – t number from user
- **parameters** – the tuple of parameters whose combinations Row object represents

completely_uncover()
Uncover all combinations inside Row

cover_cell(*key*)
Cover one combination inside Row

Parameters **key** – combination to be covered

Returns number of new covered combinations and number of new covered combinations more than ones

del_cell(*key*)
Disable one combination. If combination is disabled it means that the combination does not match the constraints

Parameters **key** – combination to be disabled

Returns number of new covered combinations

get_all_uncovered_combinations()

Returns list of all uncovered combination

is_valid(*key*)
Is the combination match the constraints.

Parameters **key** – combination to valid

uncover_cell(*key*)
Uncover one combination inside Row

Parameters **key** – combination to be uncovered

Returns number of new covered combinations and number of new covered combinations more than ones

avocado_varianter_cit.Parser module

class avocado_varianter_cit.Parser.**Parser**
Bases: object

static parse(*file_object*)
Parsing of input file with parameters and constraints

Parameters **file_object** – input file for parsing

Returns array of parameters and set of constraints

avocado_varianter_cit.Solver module

class avocado_varianter_cit.Solver.**Parameter**(*name*, *values*)
Bases: object

Storage for constraints of one parameter.

This class stores the constraints which constrain the values of one parameter.

Parameters

- **name** (*int*) – identification of parameter
- **size** (*int*) – number of values
- **constraints** (*list*) – list for storing constraints

Parameter initialization.

Parameters

- **name** (*int*) – identification of parameter
- **values** – values of parameter

Type list

add_constraint(*constraint*, *value*, *index*)
Append new constraint to the parameter.

The constraint is placed under the parameter value which is affected by this constraint. And this value is also deleted from the constraint, because is defined by the index in the 'self.constraints' list.

Parameters

- **constraint** (*list*) – will be appended to the parameter constraints
- **value** (*int*) – parameter value which is is affected by new constraint

- **index** (*int*) – index of that value inside the constraint

is_full
Compute if constraints constrain every parameter value.

Return type bool

class avocado_varianter_cit.Solver.**Solver**(*data*, *constraints*)
Bases: object

CON_NAME = 0

CON_VAL = 1

clean_hash_table(*combination_matrix*, *t_value*)

compute_constraints()

get_possible_values(*row*, *parameter*)
Compute all possible values for the given parameter.

These values are based on constraints and already picked values of other parameters.

Parameters

- **row** (*list*) – row with picked values. -1 means an unpicked value.
- **parameter** (*int*) – index of the parameter whose we want to know the values

Returns all possible values for the given parameter

Return type list

read_constraints()

simplify_constraints()

Module contents

avocado_varianter_cit.**DEFAULT_ORDER_OF_COMBINATIONS = 2**
The default order of combinations

class avocado_varianter_cit.**VarianterCit**
Bases: *avocado.core.plugin_interfaces.Varianter*

Processes the parameters file into variants

description = 'CIT Varianter'

static error_exit(*config*)

initialize(*config*)

name = 'cit'

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

class avocado_varianter_cit.**VarianterCitCLI**
Bases: *avocado.core.plugin_interfaces.CLI*

CIT Varianter options

configure(*parser*)
Configures the command line parser with options specific to this plugin.

description = "CIT Varianter options for the 'run' subcommand"

name = 'cit'

run(*config*)
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*.

10.6 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

Q

R

S

T

U

V

W

X

Y

Z