# PCVS-rt Documentation

***Release 0.5.0.-dev***

**Julien Adam**

**Jul 21, 2022**

# BASICS

**Parallel Computing Validation System** (shorten to **PCVS**) is a Validation Orchestrator designed by and for software at scale. Its primary target is HPC applications & runtimes but can flawlessly address smaller use cases. PCVS can help users to create their test scenarios and reuse them among multiples implementations, an high value when it comes to validating Programmation standards (like APIs & ABIs). No matter the number of programs, benchmarks, languages, or tech non-regression bases use, PCVS gathers in a single execution, and, with a focus on interacting with HPC batch managers efficiently, run jobs concurrently to reduce the time-to-result overall. Through basic YAML-based configuration files, PCVS handles more than hundreds of thousands of tests and helps developers to ensure code production is moving forward.

While PCVS is a validation engine, not providing any benchmarks on its own, it provides configurations to the most widely used MPI/OpenMP test applications (benchmarks & proxy apps), constituting a 300,000+ test base, offering a new way to compare implementations standard compliance.

PCVS acts as a bridge putting at scale regular test bases, by making the most of the parallelism brought by HPC architectures, able to run thousands of tests at the same time. This decrease in the time to result boosts productivity of development, leading to a better software. By separating tests (=code to execute) from application to validate, the effort of writing tests and setting validation plaform is strongly reduced, for instance:

1. The dev team writes tests and describes scenarii.
2. Cluster admins sets up the architecture for testing (compilers, tools…).
3. Q/A dep. executes application tests on given resource.

Splitting the effort will enforce reusability and flexibility. In many cases, multiple test bases share same compilers, runtimes, tools or even machine partition, it is then convenient (for administration purposes) to gather at a higher level this type of responsability.*

Using PCVS may have a slight cost as test logic must be converted to a new semantics (not the tests themselves but the testing system behind it), without being destructive for the original tests (PCVS can go together with other test frameworks).

---

**Note:** PCVS does not act as a test framework on its own. It does not have all the features from a test reporting interface either. Its purposes is to provide a new approach when building test-bases to dissociate testers from tested, to increase flexbility, reusability, ease maintenance and speed up test scheduling to minimize time to result. It is compatible with multiple build systems and test frameworks and can be integrated in a non-destructive way inside already-existing projects.

---

CHAPTER

ONE

# FEATURE OVERVIEW

Here is a quick overview of features that make PCVS a robust solution different than other validation engines.

## 1.1 Run the validation from anywhere

After installation, PCVS can be configured in a minute for the current directory:

```
$ pcvs scan .
$ pcvs run
```

## 1.2 Scale a test-suite depending on the target machine

From PCVS approach, a benchmarks is a collection of programs. coupled with a compiler and a launcher, sets of tests can be build to dynamically adapt the machine to test. With a single edition, a test-suite can be ported from a validating an MPI implementation on a simple workstation (no tests requiring more than one node & 4-8 MPI processes) to largest supercomputers (thousands of nodes). PCVS allow this thanks to **criterions**, a variadic component to apply to a program to build tests. It may be populated as follows in a profile:

```
criterion:
    iterators:
        number_of_mpi_processes:
            values: [1, 2, 4, 8, 16, 32]
```

## 1.3 Automatic Test-suite builder

PCVS relies on a specific test description syntax in order to build an efficient test-suite. To help with that process, PCVS can pre-generate templates:

```
$ pcvs scan /dir
```

## 1.4 Definition - Execution - Reporting in one place

One main advantage of PCVS is the capability to gather all validation modules in one single place, easy to install as a single user. Among others:

- Highly customizable test generation framework
- Orchestrator designed to run tests at scale
- Autonomous reporting web platform.
- Store results persistently as a Git repository for easy imports/exports & validatino progression.

CHAPTER

TWO

# GETTING STARTED

## 2.1 Installation

The simplest way to install PCVS is through **PyPI** repositories. It will fetch the lastest release but a specific version can be specified (detailed documentation in *Installation Guide*):

```
$ pip3 install pcvs
# OR
$ pip3 install pcvs<=0.5.0
$ pcvs
Usage: pcvs [OPTIONS] COMMAND [ARGS]...

PCVS main program.
...
```

Full completion (options & arguments) is provide an can be activated with:

```
# ZSH support
$ eval "$(_PCVS_COMPLETE=zsh_source pcvs)"
# BASH support
$ eval "$(_PCVS_COMPLETE=bash_source pcvs)"
```

## 2.2 PCVS-formatted test-suite

### 2.2.1 Test-suite layout

While PCVS is highly customizable, it comes with templates to locally test it without any prior knowledge. Before using PCVS, let's consider a provided test-suite as any `tests/` directory ( `all-reduce.c` & `wave.c` provided for convenience):

```
$ tree tests
tests
├── coll
│   └── all-reduce.c
└── pt2pt
    └── wave.c
2 directories, 2 files
```

PCVS needs rules to know how to parse the test-suite above to create tests. This will be done through `pcvs.yml` specification file. Such a file can be placed anywhere in the file tree. Consider putting it directly under the `tests/` directory for this example. Here is the content of this file:

**Note:** A test is the combination of a program, its arguments and the environment used to execute it. from PCVS' point of view, a test file does not carry the whole test environment. the orchestrator itself manage to build it directly from specification. Thus `pcvs.yml` expects the user to decribe programs to be used to build the test-suite.

```
# put this in test/pcvs.yml:
all_reduce_test:
        build:
                files: ["coll/all-reduce.c"]
        run:
                program: "a.out"
pt2pt_test:
        build:
                files: ["pt2pt/wave.c"]
        run:
                program: "a.out"
```

This file specifies two root nodes referred as *Test Expressions* (TE) or *Test Descriptors* (TD). It contains subondes describing how to build programs. A `build` gives informations about how to build the program. `files` (a list *or* a string) contains the whole list of files required to build the program (in case of a C file for instance). With no other information, PCVS will assume the program to be built with a compiler (no invocation to a build system here). A `run` subnode instructs PCVS to execute this program. The expected program name is `a.out`. This is the simplest way to integrate tests to PCVS. For a complete list of nodes to be used in a `pcvs.yml`, please consult *TE nodes: The complete list*

**Warning:** Beware of tabulations, YAML indentations only supports spaces !

## 2.3 Execute the test-suite

PCVS relies on (1) test specifications and (2) execution profile to create and execute a full benchmarks. Building a valid profile may be complex at first but offer a huge flexibility to solve complex validation scenarios. Still, most scenarios share similarities, like, in that case, running MPI programs. PCVS comes with default profiles for default scenarios. Here, we select the *mpi* base profile to build our own:

```
$ pcvs profile create user.my-profile --base mpi
$ pcvs profile list
```

By specifying `user.my-profile`, it will save the profile under `~/.pcvs/` and make it available for the whole `$USER`, no matter the current working directory used when running PCVS. To learn more about profile scope, please see *Scope*.

**Note:** As this profile uses MPI, we need to source an MPI implementation in the environment. Please use the method suiting your needs (spack/module/source). If interested by autoloading spack-or-module-based MPI implementation, please read *Profiles*.

Now, start PCVS. You must provide the profile & the directory where tests are located:

```
$ pcvs run --profile my-profile ./tests/
```

**Note:** the `user.` prefix to the profile name may be removed as there is no name ambiguity, PCVS will detect the proper scope.

## 2.4 Access the results

Results are stored in `$PWD/.pcvs-build/rawdata/*.json` by default. the default output directory may be changed with *pcvs run –output*. JSON files can directly processed by this-party tools. The `scheme` can be used to update the input parser with compliant output. Currently PCVS only provides specific JSON format. It is planned to support common validation format (like JUnit).

If no third-party tool is available, PCVS comes with a lightweight web server (=Flask) to serve results in a web browser:

```
# where pcvs run has been run:
$ pcvs report
# OR you may specify the run path
$ pcvs report <path>
```

Then, browse http://localhost:5000/ to browse your results.

CHAPTER

THREE

# INSTALLATION GUIDE

## 3.1 System Prerequisites

PCVS requires **python3.5+** before being installed. We encourage users to use virtual environment before installing PCVS, especially if targeting a single-user usage. To create a virtual environment, create & actiate it **before** using pip3. Please check venv (native), virtualenv (external package) or even pyenv (third-party tool) to manage them.

Here some quickstarts for each approach:

```
$ python3 -m venv ./env/
# to use it:
$ source env/bin/activate
# Work in virtual environment
$ deactivate
```

```
# install first:
$ pip3 install virtualenv
$ virtualenv ./env/
# to use it:
$ source ./env/bin/activate
# work...
$ deactivate
```

```
# install first:
$ curl https://pyenv.run | bash
$ pyenv virtualenv my_project
# to use it:
$ pyenv activate my_project
# work...
$ pyenv deactivate
```

## 3.2 Installation from PyPI

The simplest way to install PCVS is through **PyPI** repositories. It will fetch the lastest release but a specific version can be specified:

```
$ pip3 install pcvs
# OR
$ pip3 install pcvs<=0.5.0
```

## 3.3 Installation from sources

The source code is also available on Github, based on Setuptools, the manual installation is pretty simple. The latest release (and any previous archive) is also available on the website. To checkout the latest release:

```
$ git clone https://github.com/cea-hpc/pcvs.git pcvs_latest
$ pip3 install ./pcvs_latest
# OR
$ python3 ./pcvs_latest/setup.py install
```

## 3.4 Managing Dependencies

Installing only dependencies come in two way: `requirements.txt` gathers production-side deps, required for PCVS to work, while `requirements-dev.txt` contains (in addition to the base) the validation toolkit (pytest, coverage, etc.):

```
$ pip3 install -r requirements.txt
$ pip3 install -r requirements-dev.txt
# allowing to use:
$ coverage run
```

### 3.4.1 Dealing with offline networks

In some scenarios, it may not be possible to access PyPI mirrors to download dependencies (or even PCVS itself). Procedures below will describe how to download dep archives locally on a machine with internet access and then make them available for installation once manually moved to the 'offline' network. It consists in two steps. First, download the deps and create and archive (considering the project is already cloned locally):

```
$ git clone https://github.com/cea-hpc/pcvs.git # if not already done
    $ pip3 download . -d ./pcvs_deps
    # OR, select a proper requirements file
    $ pip3 download -r requirements-dev.txt -d ./pcvs_deps
    $ tar czf pcvs_deps.tar.gz ./pcvs_deps
```

Once the archive moved to the offline network (=where one wants to install PCVS), we are still considering PCVS is cloned locally:

```
$ tar xf ./pcvs_deps.tar.gz
$ pip3 install . --find-links ./pcvs_deps --no-index
# or any installation variations (-e ...)
```

**Warning:** Please use extra caution when using this method with different architectures between source & destination. By default, pip will download source-compatible wheel/source package, which may not be suited for the target machine.

pip provides options to select a given platform/target python version, which differ from the current one. Note in that case no intermediate source package will be used, only distributed versions (compiled one). To 'accept' it, you must specify `--only-binary=:all:` to force downloading distrution packages (but will failed if not provided) or `--no-deps` to exclude any dependencies to be downloade (and should be taken care manually):

```
$ pip3 download -r ... -d ... --platform x86_64 --python-version 3.5.4 [--only-
↪binary=:all:|--no-deps]
```

### 3.4.2 Important note

- PCVS requires Click >= 8.0, latest versions changed a critical keyword (to support completion) not backward compatible. Furthermore, Flask also have a dep to Click>7.1.
- To manage dict-based configuration object, PCVS relies on Addict. Not common, planned to be replaced but still required to ease configuration management process through PCVS.
- Banks are managed through Git repositories. Thus, PCVS relies on pygit2. One major issue is when pygit2 deployement requires to be rebuilt, as a strong dep to libgit2 development headers is required and may not be always provided. As a workaround for now:
  - Install a more recent pip version, able to work with wheel package (>20.x). This way, the pygit2 package won't have to be reinstalled.
  - install libgit2 headers manually

---

**Note:** A quick fix to install pygit2/libgit2 is to rely on Spack. Both are available for installation: `libgit2` & `py-pygit2`. Be sure to take a proper version above **1.x**.

---

CHAPTER

# FOUR

# BASIC USAGE

Once PCVS is installed through the *Installation Guide*, the `pcvs` is available in `PATH`. This program is the only entry point to PCVS:

```
$ pcvs
Usage: pcvs [OPTIONS] COMMAND [ARGS]...

PCVS main program.

Options:
-v, --verbose               Verbosity (cumulative)  [env var: PCVS_VERBOSE]
-c, --color / --no-color    Use colors to beautify the output  [env var:
                             PCVS_COLOR]
-g, --glyph / --no-glyph    enable/disable Unicode glyphs  [env var:
                             PCVS_ENCODING]
-C, --exec-path DIRECTORY   [env var: PCVS_EXEC_PATH]
-V, --version               Display current version
-w, --width INTEGER         Terminal width (autodetection if omitted
-P, --plugin-path PATH      Default Plugin path prefix  [env var:
                             PCVS_PLUGIN_PATH]
-m, --plugin TEXT
-help, -h, --help           Show this message and exit.

Commands:
bank     Persistent data repository management
check    Ensure future input will be conformant to standards
clean    Remove artifacts generated from PCVS
config   Manage Configuration blocks
exec     Running a specific test
profile  Manage Profiles
report   Manage PCVS result reporting interface
run      Run a validation
scan     Analyze directories to build up test conf. files
session  Manage multiple validations
```

## 4.1 Build a profile

A profile contains the whole PCVS configuration in a single component. While this approach allow deeply complex approaches, we'll target a simple MPI implementatino for this example. To create the most basic profile able to run MPI programs, we may herit ours from pre-generated called a template:

```
$ pcvs profile create -t mpi user.newprofile
```

A new profile is created and stored in the user space and will be available for any further `pcvs` invocations. It is also possible to set this profile as `local` (only for the current directory) or `global` (anyone able to use this PCVS installation. You may replace `newprofile` by a name of your choice. For a complete list of available templates, please check `pcvs profile list --all`.

A profile can be edited if necessary with `pcvs profile edit newprofile`. It will open an `$EDITOR`. When exiting, the profile is validated to ensure coherency. In case it does not fulfill a proper format, a rejection file is crated in the current directory. Once fixed, the profile can be saved as a replacement with:

```
$ pcvs profile import newprofile --force --source file.yml
```

**Warning:** The `--force` option will overwrite any profile with the same name, if it exists. Please use this option with care. In case of a rejection, the import needs to be forced in order to replace the old one.

## 4.2 Implement test descriptions

For a short example of implementing test descriptions, please refer to the *Test-suite layout* shown in the Getting-Started guide. A more detailed presentation of test description capabilities is available in its own documentation page.

The most basic `pcvs.yml` file may look like this:

```yaml
my_program:
    build:
        files: ['main.c']
    run:
        program: ['a.out']
```

PCVS also support building programs through Make, CMake & Autotools, each system having its own set of keys to configure:

- `build.make.target`: allow to configure a Make target to invoke.
- `build.cmake.vars`: variables to forward to cmake (to be prefixed w/ `-D`)
- `build.autotools.params`: configure script flags
- `build.autotools.autogen`: boolean whether to execute autogen.sh first.

Proper YAML formats can be checked before running a test-suite with:

```
$ pcvs check --directory /path/to/dir
$ pcvs check --profiles
```

## 4.3 Run a test-suite

Start a run from the local directory with our profile is as simple as:

```
$ pcvs run --profile newprofile
```

A list of directories can also be given. Once started, the validation process is logged under `$PWD/.pcvs-build` directory. If the directory already exists, it is cleaned up and reused. A lock is put in that directory to protect against concurrent PCVS execution in the same directory.

CHAPTER

FIVE

# KNOWN ISSUES

## 5.1 Error installing Pygit2

Status: **Unresolved**

**Typical Error message**: While running `pip3 install`, Such error messages might be raised:

```
-> error: git2.h: No such file or directory
-> error: #error You need a compatible libgit2 version (1.1.x)
```

### 5.1.1 Solution / Workaround

- Manually install libgit2, version >= 1.0.0
- Use wheel package to install third-party tools (only Python3.7+)

CHAPTER

SIX

# COMMON WORKFLOW

## 6.1 Per-project test infrastructure

## 6.2 Dedicated benchmark repository

### 6.2.1 Benchmark decription

PCVS is a program built for current HPC structures, it allows the launch of programs with a large coverage of parameters. Moreover, PCVS allows users to log in and out streams and handle sessions. In order to do that, PCVS needs an exhaustive configuration handled in files and profiles.

### 6.2.2 Validation profile

Validation profiles are configuration files used at launch in pcvs run. A PCVS validation profile can be generated with the following command :

```
pcvs profile build tutorial_profile
```

A pcvs profile is made of blocks that can be customized, you can export a profile to a file :

```
pcvs profile export tutorial_profile tutorial_profile.yml
```

you should have a tutorial_profile.yml file which has the following nodes :

- compiler
- criterion
- group
- machine
- runtimes

### 6.2.3 Launch tests

Tests launches are done in this case by the following command :

```
pcvs run -f -p tutorial_profile .
```

The generic command is :

```
pcvs run [option] -p [profile] [directory]
```

PCVS will scan the target directory and find any "pcvs.yml" or "pcvs.setup" file within the directory or its subdirectories, and launch the benchmark on the corresponding files.

"pcvs.setup" files must return a yaml-structured character string describing a pcvs configuration described in pcvs.yml files.

The pcvs run configuration is also structured in nodes, here is a typical example:

```yaml
tutorial_test:
    build:
        files: "@SRCPATH@/tutorial_program.c"
        sources:
            binary: "tutorial_binary"
    run:
        program: "tutorial_binary"
```

With a directory like such :

```
├── pcvs.yml
└── tutorial_program.c
```

The run will :

- build `tutorial_binary` by compiling `tutorial.c` using gcc (as specified earlier)
- run the `tutorial_binary` file

Many other options are available such as tags, flags, etc, these are referenced in the documentation of PCVS.

### 6.2.4 Visualize results

PCVS owns an html report generator, it can be used with :

```
pcvs report
```

pcvs report must be used on a directory on which tests have been run.

CHAPTER

SEVEN

# RUNNING TEST-SUITES

CHAPTER

EIGHT

# VISUALIZING RESULTS

## 8.1 Real-time progress reports

## 8.2 Post-mortem analysis

CHAPTER

NINE

# VALIDATION SETUP

## 9.1 Generalities

### 9.1.1 Setup files

Like profiles, setup configurations have nodes to describe different steps of the process. These nodes are splitted into subnodes to describe the course of the run.

The validation configuration is specified using setup files. These files can be in the yml format, or be an executable files generating a yml configuration in stdout. The informations of this configuration are crossed with the profile informations to

When PCVS is launched in a directory, it browses every subdirectory to find any `pcvs.yml` or `pcvs.setup` file and launches itself with the corresponding configuration.

#### example

```
exampletree/
├── subdir1
│   └── pcvs.yml
└── subdir2
    └── pcvs.yml
```

Launching `pcvs run exampletree` will generate tests for subdir1/pcvs.yml **and** for subdir2/pcvs.yml. There is no need to put a setup configuration in the root of `exampletree`, but it is possible to add a setup here.

## 9.2 Structure

The yml input must have one node per test. Each test can describe the following configurations :

- build
- run
- validate
- group
- tag
- artifact

### 9.2.1 Build

The build node describe how a binary file should be built depending on its sources. It contains the following subndoes :

```
build:
    files: path/to/the/file/to/build
    sources:
        binary: name of the binary to be built (if necessary)
    depends_on: ["list of test names it depends on"]

    cflags: extra cflags
    ldflags: extra ldflags
    cwd: directory where the binary should be built
    variants: [list of variants (CF Configuration basic blocks -> compiler
    node)]

    autotools:
        params: [list of options for autotools]
    cmake:
        params: [list of options for cmake]
    make:
        target: target for make command
```

### 9.2.2 Run

The run node describes how a binary file should be launched. It contains the following nodes :

```
run:
    cwd: path to build directory
    depends_on:
        test: [list of tests on which it depends]
        spack: [list of spack dependencies used by this test]
        module: [list of installed modules this test needs]
    program: name of the binary file
```

The run node owns the `iterate` subnode which can contain custom iterators desribed in the `criterion` node in the selected profile. Moreover, the run.iterate node can define custom iterators without defining them in `criterion` by writing them in the run.iterate.program node.

```
run:
    iterate:
        iterator_described_in_profile.runtime.criterion:
            values: [list of values for the corresponding iterator]
        program:
            custom_iterator:
                numeric: true/false
                type: "argument" or "environment"
                values: [list of values taken by the iterator]
                subtitle: string chosen to identify this iterator
```

### 9.2.3 Validate

The validate node describes the expected test behaviour, including exit, time and matching output.

```
validate:
    expect_exit: expected exit code (integer)
    time:
        mean: expected time to compute the test (seconds / float) tolerance:
        standard deviation for expected time (seconds / float)
        kill_after: maximum time after which process has to be killed
        (seconds / float)
    match:
        label:
            expr:
            expect:
    script:
        path: Path to a validating script
```

### 9.2.4 Group

Groups are described in profiles. They can contain `build`, `run`, `tag`, `validate`, and `artifact` subnodes. Once a group is defined in the used profile it can be called in the validation setup file.

```
group: name of the group defined in the profile
```

### 9.2.5 Tag

Tags get in the results and tests can be sorted tag-wise. A test can have multiple tags and tags do not have to be defined upstream.

```
tag:
    - tag1
    - tag2
```

### 9.2.6 Artifact

The artifact node contains anything the output should have in addition to the results of tests.

```
artifact:
    obj1: "path/to/obj1"
    obj2: "path/to/obj2"
```

CHAPTER

TEN

# CONFIGURATION BASIC BLOCKS

## 10.1 Generalities

Configuration blocks define settings for PCVS. There are 5 configurable blocks which are :

- compiler
- criterion
- group
- machine
- runtime

The configuration block is a virtual object, it doesn't exist per se, configuration blocks are used to build profiles which can be imported/exported. It is possible however to share configuration blocks by addressing them in a scope that is large enough to reach other users.

Each configuration block contains sub-blocks in order to isolate and classify informations.

## 10.2 Scope

PCVS allows 3 scopes :

- **global** for everyone on the machine having access to the PCVS installation
- **user** accessible from everywhere for the corresponding user
- **local** accessible only from a directory

## 10.3 Blocks description

### 10.3.1 compiler node

The compiler node describes the building sequence of tests, from the compiler command to options, arguments, tags, libraries, etc.

This node can contain the subnodes **comands** and **variants**

#### commands

The compiler.commands block contains a collection of compiler commands.

```
cc: compilation command for C code
cxx: compilation command for C++ code
f77: compilation command for Fortran77 code
f90: compilation command for Fortran90 code
fc: compilation command for generic Fortran code
```

#### variants

The `variants` block can contain any custom variant. The variant must have a **name**, and **arguments** as such :

```
example_variant:
    args: additionnal arguments for the example variant
openmp:
    args:  -fopenmp
strict :
    args: -Werror -Wall -Wextra
```

I this example the variants "example_variant", "openmp", and "strict" have to be specified in the validation setup where the user wants to use them.

### 10.3.2 criterion node

the criterion node contains a collection of iterators that describe the tests. PCVS can iterate over custom parameters as such :

```
iterators :
    n_[iterator] :
        **subtitle** : string used to indicate the number of [iterator] in
        the test description

        **values** : values that [iterator] allowed to take
```

#### Example

```
iterators:
    n_core:
        subtitle: C
        values:
        - 1
        - 2
```

In this case the program has to iterate on the core number and has to take the values 1 and 2. The name `n_core` is arbitrary and has to be put in the validation setup file.

### 10.3.3 group node

The group node contains group definitions that describe tests. A group description can contain any node present in the Configuration basic blocks (CF *Validation Setup* section).

#### Example

```
GRPMPI:
    run:
        iterate:
            n_omp:
                **values**: null
```

### 10.3.4 machine node

The machine node describes the constraints of the physical machine.

**machine :**

nodes : number of accessible nodes

cores_per_node : number of accessible cores per node

concurrent_run : maximum number of processes that can coexist

### 10.3.5 runtime node

The runtime node specifies entries that must be passed to the launch command. It contains subnodes such as `args`, `iterators`, etc. The `iterator` node contains arguments passed to the launching command. For example, if prterun takes the "-np" argument, which corresponds to the number of MPI threads, let's say `n_mpi`, we will get the following runtime profile :

args : arguments for the launch command

```
iterators:
    n_mpi:
        numeric : true

        option : "-np "

        type : argument

        aliases :
            [dictionary of aliases for the option]

plugins
```

CHAPTER

ELEVEN

# PROFILES

## 11.1 Generalitites

A PCVS profile defines a configuration in which PCVS will launch. This configuration is divided in nodes, and it can be customized within pcvs or in yaml files that can be imported/exported via the command line interface.

This configuration is separated in 5 nodes :

- compiler
- criterion
- group
- machine
- runtimes

each node is separated in subnodes and can be defined separately in multiple ways. As files, profiles are written with the yml syntax.

## 11.2 Scope

Profiles can have different scopes depending on which user should have access or which project should be affected by it. The 3 scopes are the following :

- Local : The profile is only seeable from a specific folder
- User : The profile is seeable from everywhere in an userspace.
- Global : The profile is accessible to everyone and from everywhere.

## 11.3 Building a new Profile

To create a blank profile, one can use the command :

```
pcvs profile build example_profile
```

To export this profile in a file format, use the command :

```
pcvs profile export example_profile example_profile.yml
```

This profile is fully customizable with any text editor, to import the profile back into PCVS use the command :

```
pcvs profile import example_profile example_profile.yml
```

Without arguments, the `pcvs profile build` command builds blocks as default, but a profile can be built with custom configuration blocks.

### 11.3.1 Building a profile with existing configuration blocks

Buiding a profile based on configuration blocks can be done in two ways :

- in the CLI
- with the interactive mode

#### In the CLI

```
pcvs profile build example_profile -b [scope].[block-name].[block-type]
```

This command has to include either 0 or 5 `-b` blocks (default or complete configuration).

#### With the interactive mode

```
pcvs profile build -i
```

For the configuration blocks setting please refer to the Configuration blocks section.

## 11.4 Managing Profiles

Besides being built, exported or imported, profiles can be altered or destroyed with the corresponding commands.

Use pcvs profile list to see every available profiles.

`pcvs profile alter [profile]` launches a text editor in order to manually change the profile. PCVS scans editors to give a choice to users.

`pcvs profile destroy [profile]` deletes a profile

TBW

Using Profiles

Profiles are used at runtime, they are specified with the `-p` option.

```
pcvs run -p example_profile
```

CHAPTER

TWELVE

# BANK MANAGEMENT

## 12.1 About

## 12.2 Create/manage/delete a bank

## 12.3 Feed a bank with results

CHAPTER

# THIRTEEN

# SESSIONS

## 13.1 About

## 13.2 Start a new session

```
$ pcvs run <...> # start an interactive session
$ pcvs run <...> --detach # start a background session
```

## 13.3 List sessions

```
$ pcvs session
$ pcvs session -l
```

## 13.4 Cleanup

Only for background sessions:

```
$ pcvs session --ack <sid>
$ pcvs session --ack-all # only completed sessions
```

CHAPTER

# FOURTEEN

# UTILITIES

## 14.1 Run a single test

```
$ pcvs exec <fully-qualified test-name>
$ pcvs exec --show [cmd|env|loads|all] <test-name>
```

## 14.2 Input correctness

### 14.2.1 Profiles

```
$ pcvs check --profiles
```

### 14.2.2 Configuration blocks

```
$ pcvs check --configs
```

### 14.2.3 Test-suites

```
$ pcvs check --directory <path>
```

## 14.3 Discover tests & generate proper configurations

```
$ pcvs scan <path>
```

## 14.4 Build & artifacts cleanup

```
$ pcvs clean --dry-run # list content to be deleted, recursively
$ pcvs clean --force # actually deleting content
```

CHAPTER

FIFTEEN

# PLUGINS

## 15.1 About

### 15.1.1 Possible passes

- START_BEFORE
- START_AFTER
- TFILE_BEFORE
- TDESC_BEFORE
- TEST_EVAL
- TDESC_AFTER
- TFILE_AFTER
- SCHED_BEFORE
- SCHED_SET_BEFORE
- SCHED_SET_EVAL
- SCHED_SET_AFTER
- SCHED_PUBLISH_BEFORE
- SCHED_PUBLISH_AFTER
- SCHED_AFTER
- END_BEFORE
- END_AFTER

## 15.2 Create a new plugin

## 15.3 Debug a plugin

## 15.4 A special case: runtime Plugins

CHAPTER

SIXTEEN

# INPUT EXAMPLES

## 16.1 TE nodes: The complete list

```yaml
---
.this_is_a_job:
  group: "GRPSERIAL"
  tag: ["a", "b"]
  build:
    # source files to include (if needed)
    files: "@SRCPATH@/*.c"
    autotools:
      params: ['--disable-bootstrap']
    cmake:
      vars: ['CMAKE_VERBOSE_MAKEFILE=ON']
    make:
      target: all
    sources:
      # program binary (if needed)
      binary: "a.out"
    # dependency scheme
    depends_on: ["this_is_another_test"]
    # extra cflags
    cflags: "extra cflags"
    # extra ldflags
    ldflags: "extra ldflags"
    # directory where program should be built
    cwd: "dir/to/build"
    # variants describing the job
    variants:
      - openmp
      - accel

  run: &run_part
    program: "./a.out"
    iterate:
      # runtime iterators
      n_mpi:
        values: [2, 4]
      n_omp:
        values: [1, 2]
```

(continues on next page)

(continued from previous page)

```
      program:
        # name will be used as part of final test-name
        give_it_a_name:
          numeric: true
          type: "argument"
          values: ["-iter 1000", "-fast"]
          subtitle: "lol"
    # directory where program should be built
    cwd: "dir/to/build"
    # dependency scheme
    depends_on: ["this_is_another_run_test_in_the_same_file"]
    package_manager:
      spack:
        - protobuf@3.1.1
        - gcc@7.3.0
      module:
        - protobuf
        - gnu/gcc/7.3.0
  artifact:
    # relative to $BUILDPATH
    obj1: "./path/1"
    obj2: "./path/2"

  # this is a copy/paste from pav2
  validate:
    expect_exit: 0
    time:
      mean: 10.0
      tolerance: 2.0
      kill_after: 20
    match:
      label:
        expr: '^\d+(\.\d+) received$'
        expect: true|false

      label2: 'Total Elapsed: \d+\.\d+ sec.$'
    script:
      path: "/path/to/script"

############################################################

# depicts an inheritance mechanism.
real_test:
  build:
    make:
      target: all
  run:
    <<: *run_part
```

## 16.2 Profile: The complete list

CHAPTER

SEVENTEEN

# PCVS ORCHESTRATION

## 17.1 Run Context initialization

## 17.2 Test Load Generation

## 17.3 Job scheduling

## 17.4 Post-mortem & live Reporting

CHAPTER

# EIGHTEEN

# CONTRIBUTION GUIDE

CHAPTER

# NINETEEN

# PACKAGE DOCUMENTATION

## 19.1 Subpackages

### 19.1.1 pcvs.backend package

#### Submodules

#### pcvs.backend.bank module

pcvs.backend.bank.**BANKS**: Dict[str, str] = {}

**Variables**
**BANKS** – list of available banks when PCVS starts up

**class** pcvs.backend.bank.**Bank**(*path: Optional[str] = None*, *token: str = ''*)

Bases: object

Representation of a PCVS result datastore.

Stored as a Git repo, a bank hold multiple results to be scanned and used to analyse benchmarks result over time. A single bank can manipulate namespaces (referred as 'projects'). The namespace is provided by suffixing `@proj` to the original name.

**Parameters**

- **root** (*str*) – the root bank directory
- **repo** (Pygit2.Repository) – the Pygit2 handle
- **config** (MetaDict) – when set, configuration file of the just-submitted archive
- **rootree** (Pygit2.Object) – When set, root handler to the next commit to insert
- **locked** (*bool*) – Serialize Bank manipulation among multiple processes
- **preferred_proj** (*str*) – extracted default-proj from initial token

**connect_repository**() → None

Connect to the bank repo, making it exclusive to the current process.

**Two scenarios:**

- the path is empty -> create a new bank
- the path is not empty -> detect a bank repo.

In any cases, lock the directory to prevent multiple accesses.

**Raises**

- ***AlreadyExistError*** – A bank is already built
- ***NotFoundError*** – the given path does not contain a Git directory.

**create_test_blob**(*data: str*) → Object

Create a small hashed object, to be stored into a bank.

**Parameters**

- **data** – any type of data to be stored. In PCVS context, it is mainly json-formatted strings.
- **data** – str

**Returns**
the pygit2-hashed representation id

**Return type**
pygit2.Object

**disconnect_repository**() → None

Free the bank repo, to be reused by other instance.

**exists**() → bool

Check if the bank is stored in PATH_BANK file.

Verification is made either on name **or** path.

**Returns**
True if both the bank exist and globally registered

**Return type**
bool

**extract_data**(*key*, *start*, *end*, *format*)

Extract information from the bank given specifications.

**Note:** This function is still WIP.

**Parameters**

- **key** (*str*) – the requested key
- **start** (*date*) – start time
- **end** (*date*) – end time
- **format** (*Not relevant yet*) – Not relevant yet

**Raises**
***ProjectNameError*** – Targeted project does not exist

**finalize_snapshot**(*tag: str*) → None

Finalize result submission into the bank.

After walking through build directory, finalize the Git tree to insert a commit on top of the created tree.

**Parameters**
**tag** (*str*) – overridable default project (if different)

**insert**(*treebuild: TreeBuilder*, *path: List[str]*, *obj: any*) → Object

Associate an object to a given tag (=path).

The result is stored into the parent subtree (treebuild). The path is an array of subrefixes, identifying the subtree where the object will be stored under the bank. This function associates the path & the object together, write the result in the parent and returns its Oid.

This function is called recursively to build the whole tree. The stop condition is when the function reaches the file (=basename), which create the real blob object.

**Parameters**

- **treebuild** (Pygit2.TreeBuilder) – the parent Oid where this association will be stored
- **path** (*list of str*) – the subpath where to store the object
- **obj** (*any*) – the actual data to store

**Returns**
the actual parent id

**Return type**
Pygit2.Oid

**list_projects**() → List[str]

Given the bank, list projects with at least one run.

In a bank, each branch is a project, just list available branches. *master* branch is not a valid project.

**Returns**
A list of available projects

**Return type**
list of str

**load_config_from_file**(*path: str*) → None

Load the configuration file associated with the archive to process.

**Parameters**
**path** (*str*) – the configuration file path

**load_config_from_str**(*s: str*) → None

Load the configuration data associated with the archive to process.

**Parameters**
**s** (*str*) – the configuration data

**property name: str**

Get bank name.

**Returns**
the exact label (without default-project suffix)

**Return type**
str

**name_exist**() → bool

Check if the bank name is registered into PATH_BANK file.

**Returns**
True if the name (lowered) is in the keys()

**Return type**
bool

**path_exist**() → bool

Check if the bank path is registered into PATH_BANK file.

**Returns**
True if the path is known.

**Return type**
bool

**property preferred_proj: Optional[str]**

Get default-project tag.

**Returns**
the exact project (without bank label prefix)

**Return type**
str

**property prefix: Optional[str]**

Get path to bank directory.

**Returns**
absolute path to directory

**Return type**
str

**save_from_archive**(*tag: str*, *archivepath: str*) → None

Extract results from the archive, if used to export results.

This is basically the same as BanK.save_from_buildir() except the archive is extracted first.

**Parameters**

- **tag** (*str*) – overridable default project (if different)
- **archivepath** (*str*) – archive path

**save_from_buildir**(*tag: str*, *buildpath: str*) → None

Extract results from the given build directory & store into the bank.

**Parameters**

- **tag** (*str*) – overridable default project (if different)
- **buildpath** (*str*) – the directory where PCVS stored results

**save_test_from_json**(*jtest: str*) → Object

Store data to a bank directly from JSON representation.

This is mainly used when a bank is directly connected to a run instance, not intermediate file is required.

**Parameters**
**jtest** (*str*) – the JSON-formatted test result

**Returns**
the blob object id

**Return type**
Pygit2.Oid

**save_to_global**() → None

Store the current bank into PATH_BANK file.

**show**() → None

Print the bank on stdout.

**Note:** This function does not use `log.IOManager`

pcvs.backend.bank.**add_banklink**(*name: str*, *path: str*) → None

Store a new bank to the global system.

**Parameters**

- **name** (*str*) – bank label
- **path** (*str*) – path to bank directory

pcvs.backend.bank.**flush_to_disk**() → None

Update the PATH_BANK file with in-memory object.

**Raises**

***IOError*** – Unable to properly manipulate the tree layout

pcvs.backend.bank.**init**() → None

Bank interface detection.

Called when program initializes. Detects defined banks in PATH_BANK

pcvs.backend.bank.**list_banks**() → dict

Accessor to bank dict (outside of this module).

**Returns**

dict of available banks.

**Return type**

dict

pcvs.backend.bank.**rm_banklink**(*name: str*) → None

Remove a bank from the global management system.

**Parameters**

**name** (*str*) – bank name

### pcvs.backend.config module

**class** pcvs.backend.config.**ConfigurationBlock**(*kind*, *name*, *scope=None*)

Bases: `object`

Handle the basic configuration block, smallest part of a profile.

From a user persperctive, a basic block is a dict, gathering in a Python object informations relative to the configuration of a single component. In PCVS, there is 5 types of components:

- Compiler-oriented (defining compiler commands)
- Runtime-oriented (setting runtime & parametrization)
- Machine-oriented (Defining resources used for execution)
- Group-oriented (used a templates to globally describe tests)
- Criterion-oriented (range of parameters used to run a test-suite)

This class helps to manage any of these config blocks above. The distinction between them is carried over by an instance attribute `_kind`.

---

**Note:** This object can easily be confused with `system.Config`. While ConfigurationBlocks are from a user perspective, system.Config handles the internal configuration tree, on which runs rely. Nonetheless, both could be merged into a single representation in later versions.

---

**Parameters**

- **_kind** (*str*) – which component this object describes
- **_name** (*str*) – block name
- **details** (*dict*) – block content
- **_scope** (*str*) – block scope, may be None
- **_file** (*str*) – absolute path for the block on disk
- **_exists** (*bool*) – True if the block exist on disk

**check**() → None

Validate a single configuration block according to its scheme.

**clone**(*clone:* ConfigurationBlock) → None

Copy the current object to create an identical one.

Mainly used to mirror two objects from different scopes.

**Parameters**
**clone** (*ConfigurationBlock*) – the object to mirror

**delete**() → None

Delete a configuration block from disk

**display**() → None

Configuration block pretty printer

**dump**() → dict

Convert the configuration Block to a regular dict.

This function first load the last version, to ensure being in sync.

**Returns**
a regular dict() representing the config blocK

**Return type**
dict

**edit**(*e=None*) → None

Open the current block for edition.

**Raises**
**Exception** – Something occured on the edited version.

**Parameters**
**e** (*str*) – the EDITOR to use instead of default.

**edit_plugin**(*e=None*) → None

Special case to handle 'plugin' key for 'runtime' blocks.

This allows to edit a de-serialized version of the 'plugin' field. By default, data are stored as a base64 string. In order to let user edit the code, the string need to be decoded first.

**Parameters**
**e** (*str*) – the editor to use instead of defaults

**fill**(*raw*) → None

Populate the block content with parameters.

**Parameters**
**raw** (*dict*) – the data to fill.

**flush_to_disk**() → None

write the configuration block to disk

**property full_name: str**

Return complete block label (scope + kind + name)

**Returns**
the fully-qualified name.

**Return type**
str

**is_found**() → bool

Check if the current config block is present on fs.

**Returns**
True if it exists

**Return type**
bool

**load_from_disk**() → None

load the configuration file to populate the current object.

**Raises**

- ***BadTokenError*** – the scope/kind/name tuple does not refer to a valid file.
- ***NotFoundError*** – The target file does not exist

**load_template**(*name=None*) → None

load from the specific template, to create a new config block

**property ref_file: str**

Return filepath associated with current config block.

**Returns**
the filepath, may be None

**Return type**
str

**retrieve_file**() → None

Associate the actual filepath to the config block.

From the stored kind, scope, name, attempt to detect configuration block on the file system (i.e. detected during module init())

property scope: str

Return block scope.

**Returns**
the scope, resolved if needed.

**Return type**
str

property short_name: str

Return the block label only.

**Returns**
the short name (may conflict with other config block)

**Return type**
str

pcvs.backend.config.check_valid_kind(*s*)

Assert the parameter is a valid kind.

Kind are defined by CONFIG_BLOCKS module attribute.

**Parameters**
**s** (*str*) – the kind to validate

**Raises**

- *BadTokenError* – Kind is None
- *BadTokenError* – Kind is not in allowed values.

pcvs.backend.config.init() → None

Load configuration tree available on disk.

This function is called when PCVS starts to load 3-scope configuration trees.

pcvs.backend.config.list_blocks(*kind*, *scope=None*)

Get available configuration blocks, as present on disk.

**Parameters**

- **kind** (str, one of CONFIG_BLOCKS values) – configBlock kind (see CONFIG_BLOCKS for possible values)
- **scope** (*'user', 'global' or 'local', optional*) – where the configblocks is located, defaults to None

**Returns**
list blocks with specified kind, restricted by scope (if any)

**Return type**
dict of config blocks

pcvs.backend.config.list_templates()

List available templates to be used for boostraping config. blocks.

**Returns**
a list of valid templates.

**Return type**
list

### pcvs.backend.profile module

**class** pcvs.backend.profile.**Profile**(*name*, *scope=None*)

Bases: object

A profile represents the most complete object the user can provide.

It is built upon 5 components, called configuration blocks (or basic blocks), one of each kind (compiler, runtime, machine, criterion & group) and gathers all required information to start a validation process. A profile object is the basic representation to be manipulated by the user.

---

**Note:** A profile object can be confused with *pcvs.helpers.system.MetaConfig*. While both are carrying the whole user configuration, a Profile object is used to build/manipulate it, while a Metaconfig is the actual internal representation of a complete run config.

---

**Parameters**

- **_name** (*str*) – profile name, should be unique for a given scope
- **_scope** (*str*) – profile scope, allowed values in *storage_order()*, defaults to None
- **_exists** (*bool*) – return True if the profile exists on disk.
- **_file** (*str*) – profile file absolute path

**check**()

Ensure profile meets scheme requirements, as a concatenation of 5 configuration block schemes.

**Raises**
*FormatError* – A 'kind' is missing from profile OR incorrect profile.

**clone**(*clone*)

Duplicate a valid profile into the current one.

**Parameters**
**clone** (*Profile*) – a valid profile object

**property compiler**

Access the 'compiler' section.

**Returns**
the 'compiler' dict segment

**Return type**
dict

**property criterion**

Access the 'criterion' section.

**Returns**
the 'criterion' dict segment

**Return type**
dict

**delete**()

Remove the current profile from disk.

It does not destroy the Python object, though.

`display`()

Display profile data into stdout/file.

`dump`()

Return the full profile content as a single regular dict.

This function loads the profile on disk first.

**Returns**
a regular dict for this profile

**Return type**
dict

`edit`(*e=None*)

Open the editor to manipulate profile content.

**Parameters**
**e** (*str*) – an editor program to use instead of defaults

**Raises**
**Exception** – Something happened while editing the file

> **Warning:** If the edition failed (validation failed) a rejected file is created in the current directory containing the rejected profile. Once manually edited, it may be submitted again through *pcvs profile import*.

`edit_plugin`(*e=None*)

Edit the 'runtime.plugin' section of the current profile.

**Parameters**
**e** (*str*) – an editor program to use instead of defaults

**Raises**
**Exception** – Something happened while editing the file.

> **Warning:** If the edition failed (validation failed) a rejected file is created in the current directory containing the rejected profile. Once manually edited, it may be submitted again through *pcvs profile import*.

`fill`(*raw*)

Update the given profile with content stored in parameter.

**Parameters**
**raw** (*dict*) – tree of (key, values) pairs to update

`flush_to_disk`()

Write down profile to disk.

Also, ensure the filepath is valid and profile content is compliant with schemes.

`property full_name`

Return fully-qualified profile name (scope + name).

**Returns**
the unique profile name.

**Return type**
str

`get_unique_id`()

Compute unique hash string identifying a profile.

This is required to make distinction between multiple profiles, based on its content (banks relies on such unicity).

**Returns**
an hashed version of profile content

**Return type**
str

`property group`

Access the 'group' section.

**Returns**
the 'group' dict segment

**Return type**
dict

`is_found`()

Check if the current profile exists on disk.

**Returns**
True if the file exist on disk

**Return type**
bool

`load_from_disk`()

Load the profile from its representation on disk.

**Raises**

- *NotFoundError* – profile does not exist
- *NotFoundError* – profile path is not valid

`load_template`(*name='default'*)

Populate the profile from templates of 5 basic config. blocks.

Filepath still need to be determined via *retrieve_file()* call.

`property machine`

Access the 'machine' section.

**Returns**
the 'machine' dict segment

**Return type**
dict

`property runtime`

Access the 'runtime' section.

**Returns**
the 'runtime' dict segment

**Return type**
dict

**property scope**

Return the profile scope.

**Returns**
profile scope

**Return type**
str

**split_into_configs**(*prefix*, *blocklist*, *scope=None*)

Convert the given profile into a list of basic blocks.

This is the reverse operation of creating a profile (not the 'opposite').

**Parameters**

- **prefix** (*str*) – common prefix name used to name basic blocks.
- **blocklist** (*list*) – list of config.blocks to generate (all 5 by default but can be retrained)
- **scope** (*str, optional*) – config block scope, defaults to None

**Raises**
*AlreadyExistError* – the created configuration block name already exist

**Returns**
list of created ConfigurationBlock

**Return type**
list

pcvs.backend.profile.**init**()

Initialization callback, loading available profiles on disk.

pcvs.backend.profile.**list_profiles**(*scope=None*)

Return a list of valid profiles found on disk.

**Parameters**
**scope** (*str, optional*) – restriction on scope, defaults to None

**Returns**
dict of 3 dicts ('user', 'local' & 'global') or a single dict (if 'scope' was set), containing, for each profile name, the filepath.

**Return type**
dict

pcvs.backend.profile.**list_templates**()

List available templates to be used for boostraping profiles.

**Returns**
a list of valid templates.

**Return type**
list

### pcvs.backend.report module

pcvs.backend.report.**build_static_pages**(*buildir*)

From a given build directory, generate static pages.

This can be used only for already run test-suites (no real-time support) and when Flask cannot/don't want to be used.

**Parameters**
**buildir** (*str*) – the build directory to load

pcvs.backend.report.**locate_json_files**(*path*)

Locate where json files are stored under the given prefix.

**Parameters**
**path** (*[type]*) – [description]

**Returns**
[description]

**Return type**
[type]

pcvs.backend.report.**start_server**()

Initialize the Flask server, default to 5000.

A random port is picked if the default is already in use.

**Returns**
the application handler

**Return type**
class:*Flask*

pcvs.backend.report.**upload_buildir_results**(*buildir*)

Upload a whole test-suite from disk to the server data model.

**Parameters**
**buildir** (*str*) – the build directory

### pcvs.backend.run module

pcvs.backend.run.**anonymize_archive**()

Erase from results any undesired output from the generated archive.

This process is disabled by default as it may increase significantly the validation process on large test bases. .. note:

```
It does not alter results in-place, only the generated archive. To
preserve the anonymization, only the archive must be exported/shared,
not the actual build directory.
```

pcvs.backend.run.**build_env_from_configuration**(*current_node*, *parent_prefix='pcvs'*)

create a flat dict of variables mapping to the actual configuration.

In order to "pcvs.setup" to read current configuration, the whole config is serialized into shell variables. Purpose of this function is to flatten the configuration tree into env vars, each tree level being divided with an underscore.

This function is called recursively to walk through the whole tree.

**Example**
The *compiler.commands.cc* config node become *$compiler_commands_cc=<...>*

**Parameters**

- **current_node** (*dict*) – current node to flatten
- **parent_prefix** (*str, optional*) – prefix used to name vars at this depth, defaults to "pcvs"

**Returns**
a flat dict of the whole configuration, keys are shell variables.

**Return type**
dict

pcvs.backend.run.**display_summary**(*the_session*)

Display a summary for this run, based on profile & CLI arguments.

pcvs.backend.run.**dup_another_build**(*build_dir*, *outdir*)

Clone another build directory to start this validation upon it.

It allows to save test-generation time if the validation is re-run under the exact same terms (identical configuration & tests).

**Parameters**

- **build_dir** (*str*) – the build directory to copy resource from
- **outdir** (*str*) – where data will be copied to.

**Returns**
the whole configuration loaded from the dup'd build directory

**Return type**
dict

pcvs.backend.run.**find_files_to_process**(*path_dict*)

Lookup for test files to process, from the list of paths provided as parameter.

The given *path_dict* is a dict, where keys are path labels given by the user, while values are the actual path. This function then returns a two-list tuple, one being files needing preprocessing (setup), the other being static configuration files (pcvs.yml)

**Each list element is a tuple:**

- origin label
- subtree from this label leading to the actual file
- file basename (either "pcvs.setup" or "pcvs.yml")

**Parameters**
**path_dict** (*dict*) – tree of paths to look for

**Returns**
a tuple with two lists

**Return type**
tuple

pcvs.backend.run.**prepare**()

Prepare the environment for a validation run.

This function prepares the build dir, create trees...

pcvs.backend.run.**print_progbar_walker**(*elt*)

Walker used to pretty-print progress bar element within Click.

**Parameters**
**elt** (*tuple*) – the element to pretty-print, containing the label & subprefix

**Returns**
the formatted string

**Return type**
str

pcvs.backend.run.**process**()

Process the test-suite generation.

It includes walking through user directories to find definitions AND generating the associated tests.

**Raises**
*TestUnfoldError* – An error occured while processing files

pcvs.backend.run.**process_dyn_setup_scripts**(*setup_files*)

Process dynamic test files and generate associated tests.

This function executes pcvs.setup files after deploying the environment (to let these scripts access it). It leads to generate "pcvs.yml" files, then processed to construct tests.

**Parameters**
**setup_files** (*tuple*) – list of tuples, each mapping a single pcvs.setup file

**Returns**
list of errors encountered while processing.

**Return type**
list

pcvs.backend.run.**process_main_workflow**(*the_session=None*)

Main run.py entry point, triggering a PCVS validation run.

This function is called by session management and may be run within an active terminal or as a detached process.

**Parameters**
**the_session** (Session, optional) – the session handler this run is connected to, defaults to None

pcvs.backend.run.**process_static_yaml_files**(*yaml_files*)

Process 'pcvs.yml' files to contruct the test base.

**Parameters**
**yaml_files** (*list*) – list of tuples, each describing a single input file.

**Returns**
list of encountered errors while processing

**Return type**
list

pcvs.backend.run.**save_for_export**(*f*, *dest=None*)

Add a resource to the archive to be exported.

Copy a source file to a destination prefix. The root build directory is replaced with $buildir/save_for_export, the relative subtree is preserved.

If 'dest' is set, the default target directory may be changed.

**Parameters**

- **f** (*str*) – the source file to be saved (absolute path)
- **dest** (*str, optional*) – the destination directory, defaults to None

**Raises**

- ***UnclassifiableError*** – input file is not a file or a directory
- ***NotFoundError*** – source or target resource cannot be determined

pcvs.backend.run.**stop_pending_jobs**(*exc=None*)

pcvs.backend.run.**str_dict_as_envvar**(*d*)

Convert a dict to a list of shell-compliant variable strings.

The final result is a regular multiline str, each line being an entry.

**Parameters**
**d** (*dict*) – the dict containing env vars to serialize

**Returns**
the str, containing mulitple lines, each of them being a var.

**Return type**
str

pcvs.backend.run.**terminate**()

Finalize a validation run.

This include generating & anonymizing (if needed) the archive.

**Raises**
***ProgramError*** – Problem occured while invoking the archive tool.

## pcvs.backend.session module

**class** pcvs.backend.session.**Session**(*date=None*, *path='.'*)

Bases: object

Object representing a running validation (detached or not).

Despite the fact it is designed for manage concurrent runs, it takes a callback and can be derived for other needs.

**Parameters**

- **_func** (*Callable*) – user function to be called once the session starts
- **_sid** (*int*) – session id, automatically generated
- **_session_infos** (*dict*) – session infos dict

class State(*value*)

Bases: IntEnum

Enum of possible Session states.

COMPLETED = 2

ERROR = 3

IN_PROGRESS = 1

WAITING = 0

classmethod from_yaml(*constructor*, *node*)

Construct a *Session.State* from its YAML representation.

Relies on the fact the node contains a 'Session.State' tag. :param loader: the YAML loader :type loader: `yaml.FullLoader` :param node: the YAML representation :type node: Any :return: The session State as an object :rtype: *Session.State*

classmethod to_yaml(*representer*, *data*)

Convert a Test.State to a valid YAML representation.

A new tag is created: 'Session.State' as a scalar (str). :param dumper: the YAML dumper object :type dumper: `YAML().dumper` :param data: the object to represent :type data: class:*Session.State* :return: the YAML representation :rtype: Any

property id

Getter to session id.

**Returns**
session id

**Return type**
int

property infos

Getter to session infos.

**Returns**
session infos

**Return type**
dict

load_from(*sid*, *data*)

Update the current object with session infos read from global file.

**Parameters**

- **sid** (*int*) – session id read from file
- **data** (*dict*) – session infos read from file

property(*kw*)

Access specific data from the session stored info session.yml.

**Parameters**
**kw** (*str*) – the information to retrieve. kw must be a valid key

**Returns**
the requested session infos if exist

**Return type**
Any

property rc

Gett to final RC.

**Returns**
rc

**Return type**
int

register_callback(*callback*, *io_file=None*)

Register the callback used as main function once the session is started.

**Parameters**

- **callback** (*Callable*) – function to invoke
- **io_file** (*str, optional*) – Where I/O will be redirected once session is started

register_io_file(*pathfile=None*)

Register the I/O file when stdout/stderr will be flushed once the session is started.

**Parameters**
**pathfile** (*str, optional*) – the path to redirect I/Os, may be None

run(**args*, ***kwargs*)

Run the session normally, without detaching the focus.

Arguments are user function ones. This function is also in charge of redirecting I/O properly (stdout, file, logs)

**Parameters**
**args** (*tuple*) – user function positional arguments

:param kwargs user function keyword-based arguments. :type kwargs: tuple

run_detached(**args*, ***kwargs*)

Run the session is detached mode.

Arguments are for user function only. :param args: user function positional arguments :type args: tuple :param kwargs user function keyword-based arguments. :type kwargs: tuple

**Returns**
the Session id created for this run.

**Return type**
int

property state

Getter to session status.

**Returns**
session status

**Return type**
int

pcvs.backend.session.**list_alive_sessions**()

Load and return the complete dict from session.yml file

**Returns**
the session dict

**Return type**
dict

pcvs.backend.session.**lock_session_file**(*timeout=None*)

Acquire the lockfil before manipulating the session.yml file.

This ensure safety between multiple PCVS instances. Be sure to call *unlock_session_file()* once completed

**Parameters**
**timeout** (*int*) – return from blocking once timeout is expired (raising TimeoutError)

pcvs.backend.session.**main_detached_session**(*sid*, *user_func*, **args*, ***kwargs*)

Main function processed when running in detached mode.

This function is called by Session.run_detached() and is launched from cloned process (same global env, new main function).

**Raises**
**Exception** – any error occuring during the main process is re-raised.

**Parameters**

- **sid** – the session id
- **user_func** – the Python function used as the new main()
- **args** (*tuple*) – user_func() arguments
- **kwargs** (*dict*) – user_func() arguments

pcvs.backend.session.**remove_session_from_file**(*sid*)

clear a session from logs.

**Parameters**
**sid** (*int*) – the session id to remove.

pcvs.backend.session.**store_session_to_file**(*c*)

Save a new session into the session file (in HOME dir).

**Parameters**
**c** (*dict*) – session infos to store

**Returns**
the sid associated to new create session id.

**Return type**
int

pcvs.backend.session.**unlock_session_file**()

Release the lock after manipulating the session.yml file.

The call won't fail if the lockfile is not taken before unlocking.

pcvs.backend.session.**update_session_from_file**(*sid*, *update*)

Update data from a running session from the global file.

This only add/replace keys present in argument dict. Other keys remain.

**Parameters**

- **sid** (*int*) – the session id

- **update** – the keys to update. If already existing, content is replaced

**Type**
dict

### pcvs.backend.utilities module

**class** pcvs.backend.utilities.**AutotoolsBuildSystem**(*root*, *dirs=None*, *files=None*)

Bases: *BuildSystem*

Derived BuildSystem targeting Autotools projects.

**fill**()

Populate the dict relatively to the build system to build the proper YAML representation.

**class** pcvs.backend.utilities.**BuildSystem**(*root*, *dirs=None*, *files=None*)

Bases: object

Manage a generic build system discovery service.

**Variables**

- **_root** – the root directory the discovery service is attached to.
- **_dirs** – list of directory found in _root.
- **_files** – list of files found in _root
- **_stream** – the resulted dict, representing targeted YAML architecture

**fill**()

This function should be overriden by overriden classes.

Nothing to do, by default.

**generate_file**(*filename='pcvs.yml'*, *force=False*)

Build the YAML test file, based on path introspection and build model.

**Parameters**

- **filename** (*str*) – test file suffix
- **force** (*bool*) – erase target file if exist.

**class** pcvs.backend.utilities.**CMakeBuildSystem**(*root*, *dirs=None*, *files=None*)

Bases: *BuildSystem*

Derived BuildSystem targeting CMake projects.

**fill**()

Populate the dict relatively to the build system to build the proper YAML representation.

**class** pcvs.backend.utilities.**MakefileBuildSystem**(*root*, *dirs=None*, *files=None*)

Bases: *BuildSystem*

Derived BuildSystem targeting Makefile-based projects.

**fill**()

Populate the dict relatively to the build system to build the proper YAML representation.

pcvs.backend.utilities.**compute_scriptpath_from_testname**(*testname*, *output=None*)

Locate the proper 'list_of_tests.sh' according to a fully-qualified test name.

**Parameters**

- **testname** (*str*) – test name belonging to the script
- **output** (*str, optional*) – prefix to walk through, defaults to current directory

**Returns**
the associated path with testname

**Return type**
str

pcvs.backend.utilities.**locate_scriptpaths**(*output=None*)

Path lookup to find all 'list_of_tests' script within a given prefix.

**Parameters**
**output** (*str, optional*) – prefix to walk through, defaults to current directory

**Returns**
the list of scripts found in prefix

**Return type**
List[str]

pcvs.backend.utilities.**process_check_configs**()

Analyse available configurations to ensure their correctness relatively to their respective schemes.

**Returns**
caught errors, as a dict, where the keys is the errmsg base64

**Return type**
dict

pcvs.backend.utilities.**process_check_directory**(*dir*, *pf_name='default'*)

Analyze a directory to ensure defined test files are valid.

**Parameters**
**dir** (*str*) – the directory to process.

**Returns**
a dict of caught errors

**Return type**
dict

pcvs.backend.utilities.**process_check_profiles**()

Analyse availables profiles and check their correctness relatively to the base scheme.

**Returns**
list of caught errors as a dict, where keys are error msg base64

**Return type**
dict

pcvs.backend.utilities.**process_check_setup_file**(*filename*, *prefix*, *run_configuration*)

Check if a given pcvs.setup could be parsed if used in a regular process.

**Parameters**

- **filename** (*str*) – the pcvs.setup filepath

- **prefix** (*str*) – the subtree the setup is extract from (used as argument)

**Returns**
a tuple (err msg, icon to print, parsed data)

**Return type**
tuple

pcvs.backend.utilities.**process_check_yaml_stream**(*data*)

Analyze a pcvs.yml stream and check its correctness relatively to standard. :param data: the stream to process :type data: str :return: a tuple (err_msg, load status icon, yaml format status icon) :rtype: tuple

pcvs.backend.utilities.**process_discover_directory**(*path*, *override=False*, *force=False*)

Path discovery to detect & intialize build systems found.

**Parameters**

- **path** (*str*) – the root path to start with
- **override** (*bool*) – True if test files should be generated, default to False
- **force** (*bool*) – True if test files should be replaced if exist, defaut to False

#### Module contents

### 19.1.2 pcvs.cli package

#### Submodules

#### pcvs.cli.cli_bank module

pcvs.cli.cli_bank.**compl_bank_projects**(*ctx*, *args*, *incomplete*)

bank project completion function.

**Parameters**

- **ctx** (Click.Context) – Click context
- **args** (*str*) – the option/argument requesting completion.
- **incomplete** (*str*) – the user input

pcvs.cli.cli_bank.**compl_list_banks**(*ctx*, *args*, *incomplete*)

bank name completion function.

**Parameters**

- **ctx** (Click.Context) – Click context
- **args** (*str*) – the option/argument requesting completion.
- **incomplete** (*str*) – the user input

### pcvs.cli.cli_config module

pcvs.cli.cli_config.**compl_list_templates**(*ctx*, *args*, *incomplete*) → list

Config template completion.

**Parameters**

- **ctx** (Click.Context) – Click context
- **args** (*str*) – the option/argument requesting completion.
- **incomplete** (*str*) – the user input

pcvs.cli.cli_config.**compl_list_token**(*ctx*, *args*, *incomplete*) → list

config name completion function.

**Parameters**

- **ctx** (Click.Context) – Click context
- **args** (*str*) – the option/argument requesting completion.
- **incomplete** (*str*) – the user input

pcvs.cli.cli_config.**config_list_single_kind**(*kind*, *scope*) → None

Related to 'config list' command, handling a single 'kind' at a time.

**Parameters**

- **kind** (*str*) – config kind
- **scope** (*str*) – config scope

### pcvs.cli.cli_profile module

pcvs.cli.cli_profile.**compl_list_templates**(*ctx*, *args*, *incomplete*)

the profile template completion.

**Parameters**

- **ctx** (Click.Context) – Click context
- **args** (*str*) – the option/argument requesting completion.
- **incomplete** (*str*) – the user input

pcvs.cli.cli_profile.**compl_list_token**(*ctx*, *args*, *incomplete*)

profile name completion function.

**Parameters**

- **ctx** (Click.Context) – Click context
- **args** (*str*) – the option/argument requesting completion.
- **incomplete** (*str*) – the user input

pcvs.cli.cli_profile.**profile_interactive_select**()

Interactive selection of config blocks to build a profile.

Based on user input, this function displays, for each kind, possible blocks and waits for a selection. A final profile is built from them.

**Returns**
concatenation of basic blokcs

**Return type**
dict

### pcvs.cli.cli_report module

### pcvs.cli.cli_run module

pcvs.cli.cli_run.**compl_list_dirs**(*ctx*, *args*, *incomplete*) → list
directory completion function.

**Parameters**

- **ctx** (Click.Context) – Click context
- **args** (*str*) – the option/argument requesting completion.
- **incomplete** (*str*) – the user input

pcvs.cli.cli_run.**handle_build_lockfile**(*exc=None*)
Remove the file lock in build dir if the application stops abrubtly.

This function will automatically forward the raising exception to the next handler.

**Raises**
**Exception** – any exception triggering this handler

**Parameters**
**exc** (*Exception*) – The raising exception.

pcvs.cli.cli_run.**iterate_dirs**(*ctx*, *param*, *value*) → dict
Validate directories provided by users & format them correctly.

Set the defaul label for a given path if not specified & Configure default directories if none was provided.

**Parameters**

- **ctx** (Click.Context) – Click Context
- **param** (*str*) – The arg targeting the function
- **value** (*List[str] or str*) – The value given by the user:

**Returns**
properly formatted dict of user directories, keys are labels.

**Return type**
dict

#### pcvs.cli.cli_session module

pcvs.cli.cli_session.**compl_session_token**(*ctx*, *args*, *incomplete*) → list

Session name completion function.

**Parameters**

- **ctx** (Click.Context) – Click context
- **args** (*str*) – the option/argument requesting completion.
- **incomplete** (*str*) – the user input

#### pcvs.cli.cli_utilities module

#### Module contents

### 19.1.3 pcvs.testing package

#### Submodules

#### pcvs.testing.tedesc module

**class** pcvs.testing.tedesc.**TEDescriptor**(*name*, *node*, *label*, *subprefix*)

Bases: object

A Test Descriptor (named TD, TE or TED), maps a test prograzm representation, as defined by a root node in a single test files.

A TE Descriptor is not a test but a definition of a program (how to use it, to compile it…), leading to a collection once combined with a profile (providing on which MPI processes to run it, for instance).

**Variables**

- **_te_name** – YAML root node name, part of its unique id
- **_te_label** – which user directory this TE is coming from
- **_te_subtree** – subprefix, relative to label, where this TE is located
- **_full_name** – fully-qualified te-name
- **_srcdir** – absolute path pointing to the YAML testfile dirname
- **_buildir** – absolute path pointing to build equivalent of _srcdir
- **_skipped** – flag if this TE should be unfolded to tests or not
- **_effective_cnt** – number of tests created by this single TE
- **_program_criterion** – extra criterion defined by the TE
- **others** – used yaml node references.

**construct_tests**()

Construct a collection of tests (build & run) from a given TE.

This function will process a YAML node and, through a generator, will create each test coming from it.

get_debug()

Build information debug for the current TE.

**Returns**
the debug info

**Return type**
dict

classmethod init_system_wide(*base_criterion_name*)

Initialize system-wide information (to shorten accesses).

**Parameters**
**base_criterion_name** (*str*) – iterator name used as scheduling resource.

property name

Getter to the current TE name.

**Returns**
te_name

**Return type**
str

pcvs.testing.tedesc.**build_job_deps**(*deps_node*, *pkg_label*, *pkg_prefix*)

Build the dependency list from a given depenency YAML node.

A depends_on is used by test to establish their relationship. It looks like:

**Example**

**depends_on:**
["list_of_test_name"]

**Parameters**

- **deps_node** (*dict*) – the TE/job YAML node.
- **pkg_label** (*str*) – the label where this TE is from (to compute depnames)
- **pkg_prefix** – the subtree where this TE is from (to compute depnames)

:type pkg_prefix, str or NoneType

**Returns**
a list of dependencies, either as depnames or PManager objects

**Return type**
list

pcvs.testing.tedesc.**build_pm_deps**(*deps_node*)

Build the dependency list from a given YAML node.

This only initialize package-manager oriented deps. For job deps, see build_job_deps

**Parameters**
**deps_node** (*str*) – contains package_manager YAML information

**Returns**
a list of PM objects, one for each entry

**Return type**
List[PManager]

pcvs.testing.tedesc.**detect_source_lang**(*array_of_files*)

Determine compilation language for a target file (or list of files).

Only one language is detected at once.

**Parameters**
**array_of_files** (*list*) – list of files to identify

**Returns**
the language code

**Return type**
str

pcvs.testing.tedesc.**prepare_cmd_build_variants**(*variants=[]*)

Build the list of extra args to add to a test using variants.

Each defined variant comes with an `arg` option. When tests enable this variant, these definitions are additioned to test compilation command. For instance, the variant `omp` defines *-fopenmp* within GCC-based profile. When a test requests to be built we `omp` variant, the flag is appended to cflags.

**Parameters**
**variants** (*list*) – the list of variants to load

**Returns**
the string as the concatenation of variant args

**Return type**
str

### pcvs.testing.test module

**class** pcvs.testing.test.**Test**(***kwargs*)

Bases: `object`

Smallest component of a validation process.

A test is basically a shell command to run. Depending on its post-execution status, a success or a failure can be determined. To handle such component in a convenient way, more information can be attached to the command like a name, the elapsed time, the output, etc.

In order to make test content flexible, there is no fixed list of attributes. A Test() constructor is initialized via (*args, **kwargs), to populate a dict *_array*.

**Variables**

- **Timeout_RC** (*int*) – special constant given to jobs exceeding their time limit.
- **NOSTART_STR** (*str*) – constant, setting default output when job cannot be run.

**NOSTART_STR = b'This test cannot be started.'**

**class State**(*value*)

Bases: `IntEnum`

Provide Status management, specifically for tests/jobs.

Defined as an enum, it represents different states a job can take during its lifetime. As tests are then serialized into a JSON file, there is no need for construction/representation (as done for Session states).

**Variables**

- **WAITING** (*int*) – Job is currently waiting to be scheduled
- **IN_PROGRESS** (*int*) – A running Set() handle the job, and is scheduled for run.
- **SUCCEED** (*int*) – Job successfully run and passes all checks (rc, matchers…)
- **FAILED** (*int*) – Job didn't suceed, at least one condition failed.
- **ERR_DEP** (*int*) – Special cases to manage jobs descheduled because at least one of its dependencies have failed to complete.
- **ERR_OTHER** (*int*) – Any other uncaught situation.

**ERR_DEP = 4**

**ERR_OTHER = 5**

**FAILURE = 3**

**IN_PROGRESS = 1**

**SUCCESS = 2**

**WAITING = 0**

**Timeout_RC = 127**

**been_executed**()

Cehck if job has been executed (not waiting or in progress).

**Returns**
False if job is waiting for scheduling or in progress.

**Return type**
bool

**property combination**

Getter to the test combination dict.

**Returns**
test comb dict.

**Return type**
dict

**property command**

Getter for the full command.

This is a real command, executed in a shell, coming from user's specificaition. It should not be confused with *wrapped_command*.

**Returns**
unescaped command line

**Return type**
str

**classmethod compute_fq_name**(*label*, *subtree*, *name*, *combination=None*, *suffix=None*)

Generate the fully-qualified (dq) name for a test, based on : - the label & subtree (original FS tree) - the name (the TE name it is originated) - a potential extra suffix - the combination PCVS computed for this iteration.

**display**()

Print the Test into stdout (through the manager).

**executed**(*state=None*)

Set current Test as executed.

**Parameters**

- **state** – give a special state to the test, defaults to FAILED
- **state** – *Test.State*, optional

**first_incomplete_dep**()

Retrive the first ready-for-schedule dep.

This is mainly used to ease the scheduling process by following the job dependency graph.

**Returns**
a Test object if possible, None otherwise

**Return type**
*Test* or NoneType

**from_json**(*test_json: str*) → None

Replace the whole Test structure based on input JSON.

**Parameters**
**json** (*test-result-valid JSON-formated str*) – the json used to set this Test

**generate_script**(*srcfile*)

Serialize test logic to its Shell representation.

This script provides the shell sequence to put in a shell script switch-case, in order to reach that test from script arguments.

**Parameters**
**srcfile** (*str*) – script filepath, to store the actual wrapped command.

**Returns**
the shell-compliant instruction set to build the test

**Return type**
str

**get_dim**(*unit='n_node'*)

Return the orch-dimension value for this test.

The dimension can be defined by the user and let the orchestrator knows what resource are, and how to 'count' them'. This accessor allow the orchestrator to exract the information, based on the key name.

**Parameters**
**unit** (*str*) – the resource label, such label should exist within the test

**Returns**
The number of resource this Test is requesting.

**Return type**
int

**has_completed_deps**()

Check if the test can be scheduled.

It ensures it hasn't been executed yet (or currently running) and all its deps are resolved and successfully run.

**Returns**
True if the job can be scheduled

**Return type**
bool

`has_failed_dep()`

Check if at least one dep is blocking this job from ever be scheduled.

**Returns**
True if at least one dep is shown a *Test.State.FAILURE* state.

**Return type**
bool

`property invocation_command`

Getter for the list_of_test.sh invocation leading to run the job.

This command is under the form: *sh /path/list_of_tests.sh <test-name>*

**Returns**
wrapper command line

**Return type**
str

`property job_depnames`

Getter to the list of deps, as an array of names.

This array is emptied when all deps are converted to objects.

**Returns**
the array of dep names

**Return type**
list

`property job_deps`

"Getter to the dependency list for this job.

The dependency struct is an array, where for each name (=key), the associated Job is stored (value) :return: the list of object-converted deps :rtype: list

`property label`

Getter to the test label.

**Returns**
the label

**Return type**
str

`property mod_deps`

Getter to the list of pack-manager rules defined for this job.

There is no need for a `_depnames` version as these deps are provided as PManager objects directly.

**Returns**
the list of package-manager based deps.

**Return type**
list

property name

Getter for fully-qualified job name.

**Returns**
test name.

**Return type**
str

pick()

Flag the job as picked up for scheduling.

res_scheme = <pcvs.helpers.system.ValidationScheme object>

resolve_a_dep(*name*, *obj*)

Resolve the dep object for a given dep name.

**Parameters**

- **name** (*str*) – the dep name to resolve, should be a valid dep.
- **obj** (*Test*) – the dep object, should be a Test()

save_final_result(*rc=0*, *time=0.0*, *out=b''*, *state=None*)

Build the final Test result node.

**Parameters**

- **rc** (*int, optional*) – return code, defaults to 0
- **time** (*float, optional*) – elapsed time, defaults to 0.0
- **out** (*bytes, optional*) – standard out/err, defaults to b”
- **state** (*Test.State*, optional) – Job final status (if override needed), defaults to FAILED

property state

Getter for current job state.

**Returns**
the job current status.

**Return type**
*Test.State*

property subtree

Getter to the test subtree.

**Returns**
test subtree.

**Return type**
str.

property tags

Getter for the full list of tags.

**Returns**
the list of of tags

**Return type**
list

property te_name

Getter to the test TE name.

**Returns**
test TE name.

**Return type**
str.

property timeout

Getter for Test timeout in seconds.

It cumulates timeout + tolerance, this value being passed to the subprocess.timeout.

**Returns**
an integer if a timeout is defined, None otherwise

**Return type**
int or NoneType

to_json(*strstate=False*)

Serialize the whole Test as a JSON object.

**Returns**
a JSON object mapping the test

**Return type**
str

### pcvs.testing.testfile module

class pcvs.testing.testfile.**TestFile**(*file_in*, *path_out*, *data=None*, *label=None*, *prefix=None*)

Bases: object

A TestFile manipulates source files to be processed as benchmarks (pcvs.yml & pcvs.setup).

It handles global informations about source imports & building one execution script (`list_of_tests.sh`) per input file.

**param _in**
YAML input file

**type _in**
str

**param _path_out**
prefix where to store output artifacts

**type _path_out**
str

**param _raw**
stream to populate the TestFile rather than opening input file

**type _raw**
dict

**param _label**
label the test file comes from

**type _label**
str

**param _prefix**
subtree the test file has been extracted

**type _prefix**
str

**param _tests**
list of tests handled by this file

**type _tests**
list

**param _debug**
debug instructions (concatenation of TE debug infos)

**type _debug**
dict

`cc_pm_string = ''`

`flush_sh_file()`
Store the given input file into their destination.

`generate_debug_info()`
Dump debug info to the appropriate file for the input object.

`load_from_str(data)`
Fill a File object from stream.

This allows reusability (by loading only once).

**Parameters**
**data** (*YAML-formatted str*) – the YAML-formatted input stream.

`process()`
Load the YAML file and map YAML nodes to Test().

`rt_pm_string = ''`

`val_scheme = None`

`pcvs.testing.testfile.load_yaml_file(f, source, build, prefix)`
Load a YAML test description file.

**Parameters**

- **f** (*str*) – YAML-based source testfile
- **source** (*str*) – source directory (used to replace placeholders)
- **build** (*str*) – build directory (placeholders)
- **prefix** (*str*) – file subtree (placeholders)

**Returns**
the YAML-to-dict content

**Return type**
dict

`pcvs.testing.testfile.replace_special_token(stream, src, build, prefix)`
Replace placeholders by their actual definition in a stream.

**Parameters**

- **stream** (*str*) – the stream to alter
- **src** (*str*) – source directory (replace SRCPATH)
- **build** (*str*) – build directory (replace BUILDPATH)
- **prefix** (*str*) – subtree for the current parsed stream

**Returns**
the modified stream

**Return type**
str

#### Module contents

### 19.1.4 pcvs.converter package

#### Submodules

#### pcvs.converter.yaml_converter module

pcvs.converter.yaml_converter.**check_if_key_matches**(*key*, *value*, *ref_array*) → tuple

list all matches for the current key in the new YAML description.

pcvs.converter.yaml_converter.**compute_new_key**(*k*, *v*, *m*) → str

replace in 'k' any pattern found in 'm'. 'k' is a string with placeholders, while 'm' is a match result with groups named after placeholders. This function will also expand the placeholder if 'call:' token is used to execute python code on the fly (complex transformation)

pcvs.converter.yaml_converter.**flatten**(*dd*, *prefix=''*) → dict

make the n-depth dict 'dd' a "flat" version, where the successive keys are chained in a tuple. for instance: {'a': {'b': {'c': value}}} –> {('a', 'b', 'c'): value}

pcvs.converter.yaml_converter.**print_version**(*ctx*, *param*, *value*) → None

print converter version number, tied to PCVS version number

pcvs.converter.yaml_converter.**process**(*data*, *ref_array=None*, *warn_if_missing=True*) → dict

Process YAML dict 'data' and return a transformed dict

pcvs.converter.yaml_converter.**process_modifiers**(*data*)

applies rules in-place for the data dict. Rules are present in the desc_dict['first'] sub-dict.

pcvs.converter.yaml_converter.**replace_placeholder**(*tmp*, *refs*) → dict

The given TMP should be a dict, where keys contain placeholders, wrapped with "<>". Each placeholder will be replaced (i.e. key will be changed) by the associated value in refs.

pcvs.converter.yaml_converter.**separate_key_and_value**(*s: str*, *c: str*) → tuple

helper to split the key and value from a string

pcvs.converter.yaml_converter.**set_with**(*data*, *klist*, *val*, *append=False*)

Add a value to a n-depth dict where the depth is declared as a list of intermediate keys. the 'append' flag indicates if the given 'value' should be appended or replace the original content

#### Module contents

### 19.1.5 pcvs.helpers package

#### Submodules

#### pcvs.helpers.criterion module

**class** pcvs.helpers.criterion.**Combination**(*crit_desc*, *dict_comb*)

Bases: object

A combination maps the actual concretization from multiple criterion.

For a given set of criterion, a Combination carries, for each kind, its associated value in order to generate the appropriate test

**get**(*k*, *dflt=None*)

Retrieve the actual value for a given combination element :param k: value to retrieve :type k: str :param dflt: default value if k is not a valid key :type: object

**items**()

Get the combination dict.

**Returns**
the whole combination dict.

**Return type**
dict

**translate_to_command**()

Translate the actual combination is tuple of three elements, based on the representation of each criterion in the test semantic. It builds tokens to provide to properly build the test command. It can either be:

1. an environment variable to export before the test to run (gathering system-scope and program-scope elements)
2. a runtime argument
3. a program-level argument (through custom-made iterators)

**translate_to_dict**()

Translate the combination into a dictionary.

**Returns**
configuration in the shape of a python dict

**Return type**
dict

**translate_to_str**()

Translate the actual combination in a pretty-format string. This is mainly used to generate actual test names

**class** pcvs.helpers.criterion.**Criterion**(*name*, *description*, *local=False*, *numeric=False*)

Bases: object

A Criterion is the representation of a component each program (i.e. test binary) should be run against. A criterion comes with a range of possible values, each leading to a different test

`aliased_value`(*val*)

Check if the given value has an alias for the current criterion. An alias is the value replacement to use instead of the one defined by test configuration. This allows to split test logic from runtime semantics.

For instance, TEs manipulate 'ib' as a value to depict the 'infiniband' network layer. But once the test has to be built, the term will change depending on the runtime carrying it, the value may be different from a runtime to another :param val: string with aliases to be replaced

`concretize_value`(*val=''*)

Return the exact string mapping this criterion, according to the specification. (is it aliased ? should the option be put before/after the value?...) :param val: value to add with prefix :type val: str :return: values with aliases replaced :rtype: str

`expand_values`()

Browse values for the current criterion and make it ready to generate combinations

`intersect`(*other*)

Update the calling Criterion with the interesection of the current range of possible values with the one given as a parameters.

This is used to refine overriden per-TE criterion according to system-wide's

`is_discarded`()

Should this criterion be ignored from the current TE generaiton ?

`is_empty`()

Is the current set of values empty May lead to errors, as it may indicates no common values has been found between user and system specifications

`is_env`()

Is this criterion targeting a component used as an env var ?

`is_local`()

Is the criterion local ? (program-scoped)

`property name`

Get the `name` attribute of this criterion.

**Returns**
name of this criterion

**Return type**
str

`property numeric`

Get the `numeric` attribute of this criterion.

**Returns**
numeric of this criterion

**Return type**
str

`override`(*desc*)

**Replace the value of the criterion using a descriptor containing the**
said value

**Parameters**
**desc** (*dict*) – descriptor supposedly containing a ``value``entry

**property subtitle**

Get the `subtitle` attribute of this criterion.

**Returns**
subtitle of this criterion

**Return type**
str

**property values**

Get the `value` attribute of this criterion.

**Returns**
values of this criterion

**Return type**
list

**class** pcvs.helpers.criterion.**Serie**(*dict_of_criterion*)

Bases: `object`

A serie ties a test expression (TEDescriptor) to the possible values which can be taken for each criterion to build test sets. A serie can be seen as the Combination generator for a given TEDescriptor

**generate**()

Generator to build each combination

**classmethod register_sys_criterion**(*system_criterion*)

copy/inherit the system-defined criterion (shortcut to global config)

pcvs.helpers.criterion.**initialize_from_system**()

Initialise system-wide criterions

TODO: Move this function elsewhere.

pcvs.helpers.criterion.**valid_combination**(*dic*)

Check if dict is a valid criterion combination .

**Parameters**
**dic** (*dict*) – dict to check

**Returns**
True if dic is a valid combination

**Return type**
bool

### pcvs.helpers.exceptions module

**class** pcvs.helpers.exceptions.**BankException**

Bases: *CommonException*

Bank-specific exceptions.

**exception ProjectNameError**(*err_msg='Unkown error'*, *help_msg='Please check pcvs --help for more information.'*, *dbg_info={}*)

Bases: *GenericError*

name is not a valid project under the given bank.

class pcvs.helpers.exceptions.**CommonException**

Bases: object

Gathers exceptions commonly encountered by more specific namespaces.

**exception AlreadyExistError**(*err_msg='Unkown error'*, *help_msg='Please check pcvs --help for more information.'*, *dbg_info={}*)

Bases: *GenericError*

The content already exist as it should.

**exception BadTokenError**(*err_msg='Unkown error'*, *help_msg='Please check pcvs --help for more information.'*, *dbg_info={}*)

Bases: *GenericError*

Badly formatted string, unable to parse.

**exception IOError**(*err_msg='Unkown error'*, *help_msg='Please check pcvs --help for more information.'*, *dbg_info={}*)

Bases: *GenericError*

Communication error (FS, process) while processing data.

**exception NotFoundError**(*err_msg='Unkown error'*, *help_msg='Please check pcvs --help for more information.'*, *dbg_info={}*)

Bases: *GenericError*

Content haven't been found based on specifications.

**exception NotImplementedError**(*err_msg='Unkown error'*, *help_msg='Please check pcvs --help for more information.'*, *dbg_info={}*)

Bases: *GenericError*

Missing implementation for this particular feature.

**exception TimeoutError**(*err_msg='Unkown error'*, *help_msg='Please check pcvs --help for more information.'*, *dbg_info={}*)

Bases: *GenericError*

The parent class timeout error.

**exception UnclassifiableError**(*err_msg='Unkown error'*, *help_msg='Please check pcvs --help for more information.'*, *dbg_info={}*)

Bases: *GenericError*

Unable to classify this common error.

**exception WIPError**(*err_msg='Unkown error'*, *help_msg='Please check pcvs --help for more information.'*, *dbg_info={}*)

Bases: *GenericError*

Work in Progress, not a real error.

class pcvs.helpers.exceptions.**ConfigException**

Bases: *CommonException*

Config-specific exceptions.

exception pcvs.helpers.exceptions.**GenericError**(*err_msg='Unkown error'*, *help_msg='Please check pcvs --help for more information.'*, *dbg_info={}*)

Bases: Exception

Generic error (custom errors will inherit of this).

property **dbg**

returns the extra infos of the exceptions (if any).

**Returns**
only the debug infos.

**Return type**
str

property **dbg_str**

Stringify the debug infos. These infos are stored as a dict initially.

**return**
a itemized string.

**rtype**
str

property **err**

returns the error part of the exceptions.

**Returns**
only the error part

**Return type**
str

property **help**

returns the help part of the exceptions.

**Returns**
only the help part

**Return type**
str

class pcvs.helpers.exceptions.**LockException**

Bases: *CommonException*

Lock-specific exceptions.

exception **BadOwnerError**(*err_msg='Unkown error'*, *help_msg='Please check pcvs --help for more information.'*, *dbg_info={}*)

Bases: *GenericError*

Attempt to manipulate the lock while the current process is not the owner.

exception **TimeoutError**(*err_msg='Unkown error'*, *help_msg='Please check pcvs --help for more information.'*, *dbg_info={}*)

Bases: *GenericError*

Timeout reached before lock.

class pcvs.helpers.exceptions.**OrchestratorException**

Bases: *CommonException*

Execution-specific errors.

**exception CircularDependencyError**(*err_msg='Unkown error'*, *help_msg='Please check pcvs --help for more information.'*, *dbg_info={}*)

Bases: *GenericError*

Circular dep detected while processing job dep tree.

**exception UndefDependencyError**(*err_msg='Unkown error'*, *help_msg='Please check pcvs --help for more information.'*, *dbg_info={}*)

Bases: *GenericError*

Declared job dep cannot be fully qualified, not defined.

class pcvs.helpers.exceptions.**PluginException**

Bases: *CommonException*

Plugin-related exceptions.

**exception BadStepError**(*err_msg='Unkown error'*, *help_msg='Please check pcvs --help for more information.'*, *dbg_info={}*)

Bases: *GenericError*

targeted pass does not exist.

**exception LoadError**(*err_msg='Unkown error'*, *help_msg='Please check pcvs --help for more information.'*, *dbg_info={}*)

Bases: *GenericError*

Unable to load plugin directory.

class pcvs.helpers.exceptions.**ProfileException**

Bases: *CommonException*

Profile-specific exceptions.

**exception IncompleteError**(*err_msg='Unkown error'*, *help_msg='Please check pcvs --help for more information.'*, *dbg_info={}*)

Bases: *GenericError*

A configuration block is missing to build the profile.

class pcvs.helpers.exceptions.**RunException**

Bases: *CommonException*

Run-specific exceptions.

**exception InProgressError**(*msg='Execution in progress in this build directory'*, ***kwargs*)

Bases: *GenericError*

A run is currently occuring in the given dir.

**exception ProgramError**(*msg='Program cannot be found'*, ***kwargs*)

Bases: *GenericError*

The given program cannot be found.

exception **TestUnfoldError**(*msg='Issue(s) while parsing test input'*, ***kwargs*)

Bases: *GenericError*

Issue raised during processing test files.

class pcvs.helpers.exceptions.**TestException**

Bases: *CommonException*

Test-specific exceptions.

exception **DynamicProcessError**(*err_msg='Unkown error'*, *help_msg='Please check pcvs --help for more information.'*, *dbg_info={}*)

Bases: *GenericError*

Test File is not properly formatted.

exception **TDFormatError**(*err_msg='Unkown error'*, *help_msg='Please check pcvs --help for more information.'*, *dbg_info={}*)

Bases: *GenericError*

Test description is wrongly formatted.

class pcvs.helpers.exceptions.**ValidationException**

Bases: *CommonException*

Validation-specific exceptions.

exception **FormatError**(*msg='Invalid format'*, ***kwargs*)

Bases: *GenericError*

The content does not comply the required format (schemes).

exception **SchemeError**(*msg='Invalid Scheme provided'*, ***kwargs*)

Bases: *GenericError*

The content is not a valid format (scheme).

### pcvs.helpers.git module

pcvs.helpers.git.**generate_data_hash**(*data*) → str

Hash data with git protocol.

**Parameters**
**data** (*str*) – data to hash

**Returns**
hashed data

**Return type**
str

pcvs.helpers.git.**get_current_usermail**()

Get the git user mail.

**Returns**
git user mail

**Return type**
str

pcvs.helpers.git.**get_current_username**() → str

Get the git username.

**Returns**
git username

**Return type**
str

pcvs.helpers.git.**request_git_attr**(*k*) → str

Get a git configuration.

**Parameters**
**k** (*str*) – parameter to get

**Returns**
a git configuration

**Return type**
str

### pcvs.helpers.log module

**class** pcvs.helpers.log.**IOManager**(*verbose=0*, *enable_unicode=True*, *length=80*, *logfile=None*, *tty=True*)

Bases: object

Manager for Input/Output streams.

Contains methods for logging and printing in PCVS. IOManager handles multiple outputs (file + standard output), logging levels (warning, error, info, etc) and pretty banners. Colors are handled by click and color tags are written in files (use less -r).

**Parameters**

- **special_chars** (*dict*) – dictionary for fancy bullet characters
- **verb_levels** (*list*) – verbosity level (normal, info, debug)
- **color_list** (*list*) – list of colors used by PCVS

**avail_chars**()

lists allowed bullet characters

**Returns**
a list of characters

**Return type**
list

**capture_exception**(*e_type*, *user_func=None*)

wraps functions to capture unhandled exceptions for high-level function not to crash. :param *e_type: errors to be caught

**color_list = ['black', 'red', 'green', 'yellow', 'blue', 'magenta', 'cyan', 'white', 'bright_black', 'bright_red', 'bright_green', 'bright_yellow', 'bright_blue', 'bright_magenta', 'bright_cyan', 'bright_white']**

**debug**(*msg*)

prints a debug message

**disable_tty**()

disables tty

**enable_tty**()

enables tty

**enable_unicode**(*e=True*)

enables/disables unicode alphabet usage

**Parameters**
**e** (*bool, optional*) – True to enable unicode usage, defaults to True

**err**(*msg*)

prints an error message

**get_verbosity_str**()

[summary]

**Returns**
[description]

**Return type**
[type]

**has_verb_level**(*match*)

returns true if the verbosity level is activated.

**Parameters**
**match** (*str or int*) – verbosity level to check

**Returns**
True if "match" verbosity level is supposed to be printed by the IOManager

**Return type**
bool

**info**(*msg*)

prints an info message

**property log_filename**

getter for logfile path

**Returns**
logfile path

**Return type**
str

**nimpl**(**msg*)

prints the "not implemented" error

**print**(**msg*)

prints a raw line. Takes multiple arguments.

**print_banner**()

prints a large banner

**print_header**(*s*, *out=True*)

prints a header

**Parameters**

- **s** (*str*) – header content
- **out** (*bool, optional*) – True if the header has to be logged, False if it has to be returned, defaults to True

**Returns**
header string if out=False, Nothing otherwise

**Return type**
str

**print_item**(*s*, *depth=1*, *out=True*, *with_bullet=True*)

prints an item

**Parameters**

- **s** (*str*) – item content
- **depth** (*int, optional*) – number of tabulations used for indentation, defaults to 1
- **out** (*bool, optional*) – True if the item has to be logged, False if it has to be returned, defaults to True
- **with_bullet** (*bool, optional*) – True if the item should have a bullet, defaults to True

**Returns**
item string if out=False, Nothing otherwise

**Return type**
str

**print_job**(*label*, *time*, *name*, *colorname='red'*, *icon=None*)

prints a job description

**Parameters**

- **label** (*str*) – job label
- **time** (*float*) – time elapsed since the job launch
- **name** (*str*) – name of the job
- **colorname** (*str, optional*) – color of the job log, defaults to "red"
- **icon** (*str, optional*) – bullet, defaults to None

**print_n_stop**(***kwargs*)

prints a message, then exits the program

**print_section**(*s*, *out=True*)

prints a section

**Parameters**

- **s** (*str*) – content of the section
- **out** (*bool, optional*) – True if the section has to be logged, False if it has to be returned, defaults to True

**Returns**
section string if out=False, Nothing otherwise

**Return type**
str

**print_short_banner**(*string=False*)

prints a little banner

**Parameters**
**string** (*bool*) – True if the banner has to be returned, False if it has to be logged

**set_logfile**(*enable*, *logfile=None*)

setter for logfile path

**Parameters**
**logfile** (*str, optional*) – logfile name, defaults to None

**set_tty**(*enable*)

[summary]

**Parameters**
**enable** (*[type]*) – [description]

**special_chars = {'ascii': {'copy': '(c)', 'empty_pg': '-', 'fail': 'X', 'full_pg': '#', 'git': '(git)', 'hdr': '=', 'item': '*', 'none': '-', 'sec': '#', 'sep_h': '-', 'sep_v': ' | ', 'star': '*', 'succ': 'V', 'time': '(time)'}, 'unicode': {'copy': '©', 'empty_pg': '\x1b[90m\x1b[1m\x1b[0m', 'fail': '', 'full_pg': '\x1b[36m\x1b[1m\x1b[0m', 'git': '', 'hdr': '', 'item': '', 'none': '', 'sec': '', 'sep_h': '', 'sep_v': '  ', 'star': '', 'succ': '✓', 'time': ''}}**

**style**(**args*, ***kwargs*)

returns a string style using click

**Returns**
a string style

**Return type**
click.style

**property tty**

getter for tty information

**Returns**
False if tty not used, 1 if tty used

**Return type**
bool

**utf**(*k*)

returns the corresponding character to a bullet character

**Parameters**
**k** (*char*) – character used as bullet character

**Returns**
fancy bullet character

**Return type**
char

**verb_levels = [(0, 'normal'), (1, 'info'), (2, 'debug')]**

**property verbose**

getter for verbosity level

**Returns**
verbosity level (0, 1, 2)

**Return type**
int

**warn**(*msg*)

prints a warning message

**write**(*txt*)

print a string.

**Parameters**
**txt** (*str*) – message to be printed

pcvs.helpers.log.**init**(*v=0*, *e=False*, *l=100*, *quiet=False*)

initializes a global manager for everyone to use

**Parameters**

- **v** (*int, optional*) – verbosity level, defaults to 0
- **e** (*bool, optional*) – True to enable unicode alphabet, False to use ascii, defaults to False
- **l** (*int, optional*) – length of the terminal, defaults to 100
- **quiet** (*bool, optional*) – False to write to stdout, defaults to False

pcvs.helpers.log.**pretty_print_exception**(*e:* GenericError)

Display exceptions in a fancy way.

**Parameters**
**e** (*exceptions.GenericError.*) – the execption to print

pcvs.helpers.log.**progbar**(*it*, *print_func=None*, *man=None*, ***kargs*)

prints a progress bar using click

**Parameters**

- **it** (*iterable*) – iterable on which the progress bar has to iterate
- **print_func** (*function, optional*) – method used to show the item next to the progress bar, defaults to None
- **man** (log.IOManager, *optional*) – manager used to describe the bullets, defaults to None

**Returns**
a click progress bar (iterable)

**Return type**
click.ProgressBar

### pcvs.helpers.pm module

class pcvs.helpers.pm.**ModuleManager**(*spec*)

Bases: *PManager*

handles Module package manager

**get**(*load=True*, *install=False*)

get the command to install the specified package

**Parameters**

- **load** (*bool, optional*) – load the specified package, defaults to True
- **install** (*bool, optional*) – install the specified package, defaults to False

**Returns**

command to install/load the package

**Return type**

str

class pcvs.helpers.pm.**PManager**(*spec=None*)

Bases: object

generic Package Manager

**get**(*load*, *install*)

Get specified packages for this manager

**Parameters**

- **load** (*bool*) – True to load the package
- **install** (*bool*) – True to install the package

**install**()

install specified packages

class pcvs.helpers.pm.**SpackManager**(*spec*)

Bases: *PManager*

handles Spack package manager

**get**(*load=True*, *install=True*)

get the commands to install the specified package

**Parameters**

- **load** (*bool, optional*) – load the specified package, defaults to True
- **install** (*bool, optional*) – install the specified package, defaults to True

**Returns**

command to install/load the package

**Return type**

str

pcvs.helpers.pm.**identify**(*pm_node*)

identifies where

**Parameters**

**pm_node** (*[type]*) – [description]

**Returns**
[description]

**Return type**
[type]

### pcvs.helpers.system module

class pcvs.helpers.system.**Config**(*d={}*, **args*, ***kwargs*)

Bases: *MetaDict*

a 'Config' is a dict extension (an MetaDict), used to manage all configuration fields. While it can contain arbitrary data, the whole PCVS configuration is composed of 5 distinct 'categories', each being a single Config. These are then gathered in a *MetaConfig* object (see below)

**from_dict**(*d*)

Fill the current Config from a given dict :param d: dictionary to add :type d: dict

**from_file**(*filename*)

Fill the current config from a given file

**Raises**
*CommonException.IOError* – file does not exist OR badly formatted

**isset**(*k*)

check key existence in config dict :param k: name of param to check :type k: str

**set_ifdef**(*k*, *v*)

shortcut function: init self[k] only if v is not None :param k: name of value to add :type k: str :param v: value to add :type v: str

**set_nosquash**(*k*, *v*)

shortcut function: init self[k] only if v is not already set :param k: name of value to add :type k: str :param v: value to add :type v: str

**to_dict**()

Convert the Config() to regular dict.

**validate**(*kw*)

Check if the Config instance matches the expected format as declared in schemes/. As the 'category' is not carried by the object itself, it is provided by the function argument.

**Parameters**
**kw** (*str*) – keyword describing the configuration to be validated (scheme)

class pcvs.helpers.system.**MetaConfig**(**args*, ***kwargs*)

Bases: *MetaDict*

Root configuration object. It is composed of Config(), categorizing each configuration blocks. This MetaConfig() contains the whole profile along with any validation and current run information. This configuration is used as a dict extension.

To avoid carrying a global instancied object over the whole code, a class-scoped attribute allows to browse the global configuration from anywhere through *Metaconfig.root*"

**bootstrap_compiler**(*node*)

"Specific initialize for compiler config block :param node: compiler block to initialize :type node: dict :return: added node :rtype: dict

**bootstrap_criterion**(*node*)

"Specific initialize for criterion config block :param node: criterion block to initialize :type node: dict :return: initialized node :rtype: dict

**bootstrap_generic**(*subnode*, *node*)

"Initialize a Config() object and store it under name 'node' :param subnode: node name :type subnode: str :param node: node to initialize and add :type node: dict :return: added subnode :rtype: dict

**bootstrap_group**(*node*)

"Specific initialize for group config block. There is currently nothing to here but calling bootstrap_generic() :param node: runtime block to initialize :type node: dict :return: added node :rtype: dict

**bootstrap_machine**(*node*)

"Specific initialize for machine config block :param node: machine block to initialize :type node: dict :return: initialized node :rtype: dict

**bootstrap_runtime**(*node*)

"Specific initialize for runtime config block :param node: runtime block to initialize :type node: dict :return: added node :rtype: dict

**bootstrap_validation**(*node*)

"Specific initialize for validation config block :param node: validation block to initialize :type node: dict :return: initialized node :rtype: dict

**bootstrap_validation_from_file**(*filepath*)

Specific initialize for validation config block. This function loads a file containing the validation dict.

**Parameters**
**filepath** (*os.path*, *str*) – path to file to be validated

**Raises**
*CommonException.IOError* – file is not found or badly formatted

**dump_for_export**()

Export the whole configuration as a dict. Prune any __internal node beforehand.

**get_internal**(*k*)

manipulate the internal MetaConfig() node to load not-exportable data :param k: value to get :type k: str

**root = None**

**set_internal**(*k*, *v*)

manipulate the internal MetaConfig() node to store not-exportable data :param k: name of value to add :type k: str :param v: value to add :type v: str

**validation_default_file = '/home/docs/.pcvs/validation.yml'**

**class** pcvs.helpers.system.**MetaDict**(**args*, ***kwargs*)

Bases: Dict

Helps with managing large configuration sets, based on dictionaries.

Once instanciated, an arbitraty subnode can be initialized like:

o = MetaDict() o.field_a.subnode2 = 4

Currently, this class is just derived from addict.Dict. It is planned to remove this adherence.

to_dict()

Convert the object to a regular dict.

**Returns**
a regular Python dict

**Return type**
Dict

class pcvs.helpers.system.ValidationScheme(*name*)

Bases: object

Object manipulating schemes (yaml) to enforce data formats. A validationScheme is instancied according to a 'model' (the format to validate). This instance can be used multiple times to check multiple streams belonging to the same model.

avail_list = None

classmethod available_schemes()

return list of currently supported formats to be validated. The list is extracted from INSTALL/schemes/<model>-scheme.yml

validate(*content*, *filepath=None*)

Validate a given datastructure (dict) agasint the loaded scheme.

**Parameters**

- **content** (*dict*) – json to validate
- **filepath** –

**Raises**

- *ValidationException.FormatError* – data are not valid
- *ValidationException.SchemeError* – issue while applying scheme

## pcvs.helpers.utils module

pcvs.helpers.utils.check_valid_program(*p*, *succ=None*, *fail=None*, *raise_if_fail=True*)

Check if p is a valid program, using the `which` function.

**Parameters**

- **p** (*str*) – program to check
- **succ** (*optional*) – function to call in case of success, defaults to None
- **fail** (*optional*) – function to call in case of failure, defaults to None
- **raise_if_fail** (*bool, optional*) – Raise an exception in case of failure, defaults to True

**Raises**
*RunException.ProgramError* – p is not a valid program

**Returns**
True if p is a program, False otherwise

**Return type**
bool

pcvs.helpers.utils.**check_valid_scope**(*s*)

Check if argument is a valid scope (local, user, global).

**Parameters**
**s** (*str*) – scope to check

**Raises**
*CommonException.BadTokenError* – the argument is not a valid scope

pcvs.helpers.utils.**copy_file**(*src*, *dest*)

Copy a source file into a destination directory.

**Parameters**

- **src** (*str*) – source file to copy.
- **dest** (*str*) – destination directory, may not exist yet.

pcvs.helpers.utils.**create_home_dir**()

Create a home directory

pcvs.helpers.utils.**create_or_clean_path**(*prefix*, *dir=False*)

Create a path or cleans it if it already exists.

**Parameters**

- **prefix** (*os.path, str*) – prefix of the path to create
- **dir** (*bool, optional*) – True if the path is a directory, defaults to False

pcvs.helpers.utils.**cwd**(*path*)

Change the working directory.

**Parameters**
**path** (*os.path, str*) – new working directory

pcvs.helpers.utils.**extract_infos_from_token**(*s*, *pair='right'*, *single='right'*, *maxsplit=3*)

Extract fields from tokens (a, b, c) from user's string.

**Parameters**

- **s** (*str*) – the input string
- **pair** (*str, optional*) – padding side when only 2 tokens found, defaults to "right"
- **single** (*str, optional*) – padding side when only 1 token found, defaults to "right"
- **maxsplit** (*int, optional*) – maximum split number for s, defaults to 3

**Returns**
3-string-tuple: mapping (scope, kind, name), any of them may be null

**Return type**
tuple

pcvs.helpers.utils.**find_buildir_from_prefix**(*path*)

Find the build directory from the `path` prefix.

**Parameters**
**path** (*os.path, str*) – path to search the build directory from

**Raises**
*CommonException.NotFoundError* – the build directory is not found

**Returns**
the path of the build directory

**Return type**
os.path

pcvs.helpers.utils.**generate_local_variables**(*label*, *subprefix*)

Return directories from PCVS working tree :

- the base source directory
- the current source directory
- the base build directory
- the current build directory

**Parameters**

- **label** (*str*) – name of the object used to generate paths
- **subprefix** (*str*) – path to the subdirectories in the base path

**Raises**
*CommonException.NotFoundError* – the label is not recognized as to be validated

**Returns**
paths for PCVS working tree

**Return type**
tuple

pcvs.helpers.utils.**get_lock_owner**(*f*)

The lock file will contain the process ID owning the lock. This function returns it.

**Parameters**
**f** (*str*) – the original file to mutex

**Returns**
the process ID

**Return type**
int

pcvs.helpers.utils.**get_lockfile_name**(*f*)

From a file to mutex, return the file lock name associated with it.

For instance for /a/b.yml, the lock file name will be /a/.b.yml.lck

**Parameters**
**f** (*str*) – the file to mutex

pcvs.helpers.utils.**is_locked**(*f*)

Is the given file locked somewhere else ?

**Parameters**
**f** (*str*) – the file to test

**Returns**
a boolean indicating wether the lock is hold or not.

**Return type**
bool

pcvs.helpers.utils.**lock_file**(*f*, *reentrant=False*, *timeout=None*, *force=True*)

Try to lock a directory.

**Parameters**

- **f** (*os.path*) – name of lock
- **reentrant** (*bool, optional*) – True if this process may have locked this file before, defaults to False
- **timeout** (*int (seconds), optional*) – time before timeout, defaults to None

**Raises**

*LockException.TimeoutError* – timeout is reached before the directory is locked

**Returns**

True if the file is reached, False otherwise

**Return type**

bool

pcvs.helpers.utils.**program_timeout**(*sig*, *frame*)

pcvs.helpers.utils.**set_local_path**(*path*)

Set the prefix for the `local` storage.

**Parameters**

**path** (*os.path*) – path of the `local` storage

pcvs.helpers.utils.**start_autokill**(*timeout=None*)

pcvs.helpers.utils.**storage_order**()

Return scopes in order of searching.

**Returns**

a list of scopes

**Return type**

list

pcvs.helpers.utils.**trylock_file**(*f*, *reentrant=False*)

Try to lock a file (used in lock_file).

**Parameters**

- **f** (*os.path*) – name of lock
- **reentrant** (*bool, optional*) – True if this process may have locked this file before, defaults to False

**Returns**

True if the file is reached, False otherwise

**Return type**

bool

pcvs.helpers.utils.**unlock_file**(*f*)

Remove lock from a directory.

**Parameters**

**f** (*os.path*) – file locking the directory

#### Module contents

### 19.1.6 pcvs.orchestration package

#### Submodules

#### pcvs.orchestration.publishers module

class pcvs.orchestration.publishers.**Publisher**(*prefix='.'*)

Bases: object

Manage result publication and storage on disk.

Jobs are submitted to a publisher, forming a set of ready-to-be-flushed elements. Every time *generate_file* is invoked, tests are dumped to a file named after `pcvs_rawdat<id>.json`, where `id` is a automatic increment. Then, pool is emptied and waiting for new tests. This way, a single manager manages multiple files.

**Variables**

- **scheme** – path to test result scheme
- **increment** – used within filename
- **fn_fmt** – filename format string
- **_layout** – hierarchical representation of tests within the file
- **_destpath** – target filepath

**add**(*json*)

Add a new job to be published.

**Parameters**
**json** (*json*) – the Test() JSON.

**empty_entries**()

Empty the publisher from all saved jobs.

**flush**()

Flush down saved JSON-based jobs to a single file.

The Publisher is then reset for the next flush (next file).

**fn_fmt = 'pcvs_rawdat{:>04d}.json'**

**property format**

Return format type (currently only 'json' is supported).

**Returns**
format as printable string

**Return type**
str

**increment = 0**

**scheme = None**

**validate**(*stream*)

Ensure the test results layout saved is compliant with standards.

**Parameters**
**stream** – content to validate against publisher scheme.

**Type**
stream: dict or str

### Module contents

**class** pcvs.orchestration.**Orchestrator**

Bases: object

The job orchestrator, managing test processing through the whole test base.

**Variables**

- **_conf** – global configuration object
- **_pending_sets** – started Sets not completed yet
- **_max_res** – number of resources allowed to be used
- **_publisher** – result publisher
- **_manager** – job manager
- **_maxconcurrent** – Max number of sets started at the same time.

**add_new_job**(*job*)

Append a new job to be scheduled.

**Parameters**
**job** (Test) – job to append

**print_infos**()

display pre-run infos.

**run**(*session*)

Start the orchestrator.

**Parameters**
**session** (Session) – container owning the run.

**start_new_runner**()

Start a new Runner thread & register comm queues.

**start_run**(*the_session=None*, *restart=False*)

Start the orchestrator.

**Parameters**

- **the_session** (Session) – container owning the run.
- **restart** (*False for a brand new run.*) – whether the run is starting from scratch

**classmethod stop**()

Request runner threads to stop.

**stop_runners**()

Stop all previously started runners.

Wait for their completion.

pcvs.orchestration.**global_stop**()

### 19.1.7 pcvs.webview package

#### Module contents

pcvs.webview.**create_app**()

Start and run the Flask application.

**Returns**
the application

**Return type**
Flask

## 19.2 Submodules

### 19.2.1 pcvs.main module

pcvs.main.**print_version**(*ctx*, *param*, *value*)

Print current version.

This is used as an option formatter, PCVS is not event loaded yet.

**Parameters**

- **ctx** (Click.Context) – Click Context.
- **param** (*str*) – the option triggering the callback (unused here)
- **value** – the value provided with the option (unused here)

### 19.2.2 pcvs.version module

## 19.3 Module contents

# PYTHON MODULE INDEX

## p

# INDEX

## D



## E



## F

## G



## H



## I

## J



## L



## M

## N



## O



## P

## R



## S

## T



## U



## V

## W