salomeTools Documentation

Release 5.0.0dev

CEA DEN/DANS/DM2S/STMF/LGLS

Jun 19, 2018

CONTENTS

Warning: This documentation is under construction.

The **Sa**lome**T**ools (sat) is a suite of commands that can be used to perform operations on SALOME[1].

For example, sat allows you to compile SALOME's codes (prerequisites, products) create application, run tests, create package, etc.

This utility code is a set of Python[2] scripts files.

Find a pdf version of this documentation

[1] http://www.salome-platform.org
[2] https://docs.python.org/2.7

CHAPTER

ONE

QUICK START

1.1 Installation

Usually user could find (and use) command **sat** directly after a 'detar' installation of SALOME.

```
tar -xf .../SALOME_xx.tgz
cd SALOME_xx
ls -l sat       # sat -> salomeTools/sat
```

Python package (scripts of salomeTools) actually remains in directory *salomeTools*.

1.2 Configuration

salomeTools uses files to store its configuration parameters.

There are several configuration files which are loaded by salomeTools in a specific order. When all the files are loaded a *config* object is created. Then, this object is passed to all command scripts.

1.2.1 Syntax

The configuration files use a python-like structure format (see config module[3] for a complete description).

- **{}** define a dictionary,
- **[]** define a list,
- **@** can be used to include a file,
- **$prefix** reference to another parameter (ex: `$PRODUCT.name`),
- **#** comments.

Note: in this documentation a reference to a configuration parameter will be noted `XXX.YYY`.

1.2.2 Description

VARS section

This section is dynamically created by salomeTools at run time.
It contains information about the environment: date, time, OS, architecture etc.

[3] http://www.red-dove.com/config-doc/

```
# to get the current setting
sat config --value VARS
```

PRODUCTS section

This section is defined in the product file.
It contains instructions on how to build a version of SALOME (list of prerequisites-products and versions)

```
# to get the current setting
sat config SALOME-xx --value PRODUCTS
```

APPLICATION section

This section is optional, it is also defined in the product file.
It gives additional parameters to create an application based on SALOME, as versions of products to use.

```
# to get the current setting
sat config SALOME-xx --value APPLICATION
```

USER section

This section is defined by the user configuration file, `~/.salomeTools/salomeTools.pyconf`.

The `USER` section defines some parameters (not exhaustive):

- **workDir** :

  The working directory.
  Each product will be usually installed here (in sub-directories).

- **browser** : The web browser to use (*firefox*).
- **editor** : The editor to use (*vi, pluma*).
- and other user preferences.

```
# to get the current setting
sat config SALOME-xx --value USER
```

1.3 Usage of SAlomeTools

1.3.1 Usage

sat usage is a Command Line Interface (CLI[4]).

```
sat [generic_options] [command] [product] [command_options]
```

Options of sat

Useful *not exhaustive* generic options of *sat* CLI.

–help or -h

Get help as simple text.

```
sat --help          # get the list of existing commands
sat --help compile  # get the help on a specific command 'compile'
```

–debug or -g

Execution in debug mode allows to see more trace and *stack* if an exception is raised.

–verbose or -v

Change verbosity level (default is 3).

```
# for product 'SALOME_xx' for example
# execute compile command in debug mode with trace level 4
sat -g -v 4 compile SALOME_xx
```

1.3.2 Build a SALOME product

Get the list of available products

To get the list of the current available products in your context:

```
sat config --list
```

Prepare sources of a product

To prepare (get) *all* the sources of a product (*SALOME_xx* for example):

```
sat prepare SALOME_xx
```

The sources are usually copied in directories
$USER.workDir + SALOME_xx. . . + SOURCES + $PRODUCT.name

[4] https://en.wikipedia.org/wiki/Command-line_interface

Compile SALOME

To compile products:

```
# compile all prerequisites/products
sat compile SALOME_xx

# compile only 2 products (KERNEL and SAMPLES), if not done yet
sat compile SALOME_xx --products KERNEL,SAMPLES

# compile only 2 products, unconditionaly
sat compile SALOME_xx ---products SAMPLES --clean_all
```

The products are usually build in the directories
$USER.workDir + SALOME_xx… + BUILD + $PRODUCT.name

The products are usually installed in the directories
$USER.workDir + SALOME_xx… + INSTALL + $PRODUCT.name

CHAPTER

TWO

LIST OF COMMANDS

2.1 Command doc

2.1.1 Description

The **doc** command displays sat documentation.

2.1.2 Usage

- Show (in a web browser) the sat documentation:

  ```
  sat doc --html
  ```

- Show (in evince, for example) the (same) sat documentation in format pdf:

  ```
  sat doc --pdf
  ```

- Edit and modify in your preference user editor the sources files (rst) of sat documentation:

  ```
  sat doc --edit
  ```

- get information how to compile locally sat documentation (from the sources files):

  ```
  sat doc --compile
  ```

2.1.3 Some useful configuration pathes

- **USER**
  - **browser** : The browser used to show the html files (*firefox* for example).
  - **pdf_viewer** : The viewer used to show the pdf files (*evince* for example).
  - **editor** : The editor used to edit ascii text files (*pluma or gedit* for example).

2.2 Command config

2.2.1 Description

The **config** command manages sat configuration. It allows display, manipulation and operation on configuration files

2.2.2 Usage

- Edit the user personal configuration file `$HOME/.salomeTools/SAT.pyconf`. It is used to store the user personal choices, like the favorite editor, browser, pdf viewer:

  ```
  sat config --edit
  ```

- List the available applications (they come from the sat projects defined in `data/local.pyconf`:

  ```
  sat config --list
  ```

- Edit the configuration of an application:

  ```
  sat config <application> --edit
  ```

- Copy an application configuration file into the user personal directory:

  ```
  sat config <application> --copy [new_name]
  ```

- Print the value of a configuration parameter.
  Use the automatic completion to get recursively the parameter names.
  Use *–no_label* option to get *only* the value, *without* label (useful in automatic scripts).
  Examples (with *SALOME-xx* as *SALOME-8.4.0*):

  ```
  # sat config --value <parameter_path>
  sat config --value .          # all the configuration
  sat config --value LOCAL
  sat config --value LOCAL.workdir

  # sat config <application> --value <parameter_path>
  sat config SALOME-xx --value APPLICATION.workdir
  sat config SALOME-xx --no_label --value APPLICATION.workdir
  ```

- Print in one-line-by-value mode the value of a configuration parameter,
  with its source *expression*, if any.
  This is a debug mode, useful for developers.
  Prints the parameter path, the source expression if any, and the final value:

  ```
  sat config SALOME-xx -g USER
  ```

  Note: And so, *not only for fun*, to get **all expressions** of configuration

  ```
  sat config SALOME-xx -g . | grep -e "-->"
  ```

- Print the patches that are applied:

  ```
  sat config SALOME-xx --show_patchs
  ```

- Get information on a product configuration:

```
# sat config <application> --info <product>
sat config SALOME-xx --info KERNEL
sat config SALOME-xx --info qt
```

2.2.3 Some useful configuration pathes

Exploring a current configuration.

- **PATHS**: To get list of directories where to find files.
- **USER**: To get user preferences (editor, pdf viewer, web browser, default working dir).

sat commands:

```
sat config SALOME-xx -v PATHS
sat config SALOME-xx -v USERS
```

2.3 Command prepare

2.3.1 Description

The **prepare** command brings the sources of an application in the *sources application directory*, in order to compile them with the compile command.

The sources can be prepared from VCS software (*cvs, svn, git*), an archive or a directory.

Warning: When sat prepares a product, it first removes the existing directory, except if the development mode is activated. When you are working on a product, you need to declare in the application configuration this product in **dev** mode.

2.3.2 Remarks

VCS bases (git, svn, cvs)

The *prepare* command does not manage authentication on the cvs server. For example, to prepare modules from a cvs server, you first need to login once.

To avoid typing a password for each product, you may use a ssh key with passphrase, or store your password (in .cvspass or .gitconfig files). If you have security concerns, it is also possible to use a bash agent and type your password only once.

Dev mode

By default *prepare* uses *export* mode: it creates an image of the sources, corresponding to the tag or branch specified, without any link to the VCS base. To perform a *checkout* (svn, cvs) or a *git clone* (git), you need to declare the product in dev mode in your application configuration: edit the application configuration file (pyconf) and modify the product declaration:

```
sat config <application> -e
# and edit the product section:
#   <product> : {tag : "my_tag", dev : "yes", debug : "yes"}
```

The first time you will execute the *sat prepare* command, your module will be downloaded in *checkout* mode (inside the SOURCES directory of the application. Then, you can develop in this repository, and finally push them in the base when they are ready. If you type during the development process by mistake a *sat prepare* command, the sources in dev mode will not be altered/removed (Unless you use -f option)

2.3.3 Usage

- Prepare the sources of a complete application in SOURCES directory (all products):

  ```
  sat prepare <application>
  ```

- Prepare only some modules:

  ```
  sat prepare <application>  --products <product1>,<product2> ...
  ```

- Use –force to force to prepare the products in development mode (this will remove the sources and do a new clone/checkout):

  ```
  sat prepare <application> --force
  ```

- Use –force_patch to force to apply patch to the products in development mode (otherwise they are not applied):

  ```
  sat prepare <application> --force_patch
  ```

2.3.4 Some useful configuration pathes

Command *sat prepare* uses the *pyconf file configuration* of each product to know how to get the sources.

Note: to verify configuration of a product, and get name of this *pyconf files configuration*

```
sat config <application> --info <product>
```

- **get_method**: the method to use to prepare the module, possible values are cvs, git, archive, dir.
- **git_info** : (used if get_method = git) information to prepare sources from git.
- **svn_info** : (used if get_method = svn) information to prepare sources from cvs.
- **cvs_info** : (used if get_method = cvs) information to prepare sources from cvs.
- **archive_info** : (used if get_method = archive) the path to the archive.
- **dir_info** : (used if get_method = dir) the directory with the sources.

2.4 Command compile

2.4.1 Description

The **compile** command allows compiling the products of a SALOME[5] application.

2.4.2 Usage

- Compile a complete application:

```
sat compile <application>
```

- Compile only some products:

```
sat compile <application> --products <product1>,<product2> ...
```

- Use *sat -t* to duplicate the logs in the terminal (by default the log are stored and displayed with *sat log* command):

```
sat -t compile <application> --products <product1>
```

- Compile a module and its dependencies:

```
sat compile <application> --products med --with_fathers
```

- Compile a module and the modules depending on it (for example plugins):

```
sat compile <application> --products med --with_children
```

- Clean the build and install directories before starting compilation:

```
sat compile <application> --products GEOM  --clean_all
```

Note:

a warning will be shown if option –*products* is missing
(as it will clean everything)

- Clean only the install directories before starting compilation:

```
sat compile <application> --clean_install
```

- Add options for make:

```
sat compile <application> --products <product> --make_flags <flags>
```

- Use the –*check* option to execute the unit tests after compilation:

```
sat compile <application> --check
```

- Remove the build directory after successful compilation (some build directory like qt are big):

```
sat compile <application> --products qt --clean_build_after
```

- Stop the compilation as soon as the compilation of a module fails:

[5] http://www.salome-platform.org

```
sat compile <product> --stop_first_fail
```

- Do not compile, just show if products are installed or not, and where is the installation:

```
sat compile <application> --show
```

2.4.3 Some useful configuration pathes

The way to compile a product is defined in the *pyconf file configuration*. The main options are:

- **build_source** : the method used to build the product (cmake/autotools/script)
- **compil_script** : the compilation script if build_source is equal to "script"
- **cmake_options** : additional options for cmake.
- **nb_proc** : number of jobs to use with make for this product.

2.5 Command launcher

2.5.1 Description

The **launcher** command creates a SALOME launcher, a python script file to start SALOME[6].

2.5.2 Usage

- Create a launcher:

  ```
  sat launcher <application>
  ```

  Generate a launcher in the application directory, i.e `$APPLICATION.workdir`.

- Create a launcher with a given name (default name is `APPLICATION.profile.launcher_name`)

  ```
  sat launcher <application> --name ZeLauncher
  ```

  The launcher will be called *ZeLauncher*.

- Set a specific resources catalog:

  ```
  sat launcher <application>  --catalog  <path of a salome resources catalog>
  ```

  Note that the catalog specified will be copied to the profile directory.

- Generate the catalog for a list of machines:

  ```
  sat launcher <application> --gencat <list of machines>
  ```

  This will create a catalog by querying each machine (memory, number of processor) with ssh.

- Generate a mesa launcher (if mesa and llvm are parts of the application). Use this option only if you have to use salome through ssh and have problems with ssh X forwarding of OpengGL modules (like Paravis):

  ```
  sat launcher <application> --use_mesa
  ```

2.5.3 Configuration

Some useful configuration pathes:

- **APPLICATION.profile**
  - **product** : the name of the profile product (the product in charge of holding the application stuff, like logos, splashscreen)
  - **launcher_name** : the name of the launcher.

[6] http://www.salome-platform.org

2.6 Command application

2.6.1 Description

The **application** command creates a virtual SALOME[7] application. Virtual SALOME applications are used to start SALOME when distribution is needed.

2.6.2 Usage

- Create an application:

  ```
  sat application <application>
  ```

  Create the virtual application directory in the salomeTool application directory `$APPLICATION.workdir`.

- Give a name to the application:

  ```
  sat application <application> --name <my_application_name>
  ```

  Remark: this option overrides the name given in the virtual_app section of the configuration file `$APPLICATION.virtual_app.name`.

- Change the directory where the application is created:

  ```
  sat application <application> --target <my_application_directory>
  ```

- Set a specific SALOME[8] resources catalog (it will be used for the distribution of components on distant machines):

  ```
  sat application <application> --catalog <path_to_catalog>
  ```

  Note that the catalog specified will be copied to the application directory.

- Generate the catalog for a list of machines:

  ```
  sat application <application> --gencat machine1,machine2,machine3
  ```

  This will create a catalog by querying each machine through ssh protocol (memory, number of processor) with ssh.

- Generate a mesa application (if mesa and llvm are parts of the application). Use this option only if you have to use salome through ssh and have problems with ssh X forwarding of OpengGL modules (like Paravis):

  ```
  sat launcher <application> --use_mesa
  ```

2.6.3 Some useful configuration pathes

The virtual application can be configured with the virtual_app section of the configutation file.

- **APPLICATION.virtual_app**
  - **name** : name of the launcher (to replace the default runAppli).
  - **application_name** : (optional) the name of the virtual application directory, if missing the default value is `$name + _appli`.

[7] http://www.salome-platform.org
[8] http://www.salome-platform.org

2.7 Command log

2.7.1 Description

The **log** command displays sat log in a web browser or in a terminal.

2.7.2 Usage

- Show (in a web browser) the log of the commands corresponding to an application:

```
sat log <application>
```

- Show the log for commands that do not use any application:

```
sat log
```

- The –terminal (or -t) display the log directly in the terminal, through a CLI[9] interactive menu:

```
sat log <application> --terminal
```

- The –last option displays only the last command:

```
sat log <application> --last
```

- To access the last compilation log in terminal mode, use –last_terminal option:

```
sat log <application> --last_terminal
```

- The –clean (int) option erases the n older log files and print the number of remaining log files:

```
sat log <application> --clean 50
```

2.7.3 Some useful configuration pathes

- **USER**
  - **browser** : The browser used to show the log (by default *firefox*).
  - **log_dir** : The directory used to store the log files.

[9] https://en.wikipedia.org/wiki/Command-line_interface

2.8 Command environ

2.8.1 Description

The **environ** command generates the environment files used to run and compile your application (as SALOME[10] is an example).

Note: these files are **not** required, salomeTool set the environment himself, when compiling. And so does the salome launcher.

These files are useful when someone wants to check the environment. They could be used in debug mode to set the environment for *gdb*.

The configuration part at the end of this page explains how to specify the environment used by sat (at build or run time), and saved in some files by *sat environ* command.

2.8.2 Usage

- Create the shell environment files of the application:

```
sat environ <application>
```

- Create the environment files of the application for a given shell. Options are bash, bat (for windows) and cfg (the configuration format used by SALOME[11]):

```
sat environ <application> --shell [bash|cfg|all]
```

- Use a different prefix for the files (default is 'env'):

```
# This will create file <prefix>_launch.sh, <prefix>_build.sh
sat environ <application> --prefix <prefix>
```

- Use a different target directory for the files:

```
# This will create file env_launch.sh, env_build.sh
# in the directory corresponding to <path>
sat environ <application> --target <path>
```

- Generate the environment files only with the given products:

```
# This will create the environment files only for the given products
# and their prerequisites.
# It is useful when you want to visualise which environment uses
# sat to compile a given product.
sat environ <application> --product <product1>,<product2>, ...
```

2.8.3 Configuration

The specification of the environment can be done through several mechanisms.

1. For salome products (the products with the property `is_SALOME_module` as `yes`) the environment is set automatically by sat, in respect with SALOME[12] requirements.

[10] http://www.salome-platform.org
[11] http://www.salome-platform.org
[12] http://www.salome-platform.org

2. For other products, the environment is set with the use of the environ section within the pyconf file of the product. The user has two possibilities, either set directly the environment within the section, or specify a python script which wil be used to set the environment programmatically.

Within the section, the user can define environment variables. He can also modify PATH variables, by appending or prepending directories. In the following example, we prepend *<install_dir>/lib* to `LD_LIBRARY_PATH` (note the *left first* underscore), append *<install_dir>/lib* to `PYTHONPATH` (note the *right last* underscore), and set `LAPACK_ROOT_DIR` to *<install_dir>*:

```
environ :
{
  _LD_LIBRARY_PATH : $install_dir + $VARS.sep + "lib"
  PYTHONPATH_ : $install_dir + $VARS.sep + "lib"
  LAPACK_ROOT_DIR : $install_dir
}
```

It is possible to distinguish the build environment from the launch environment: use a subsection called *build* or *launch*. In the example below, `LD_LIBRARY_PATH` and `PYTHONPATH` are only modified at run time, not at compile time:

```
environ :
{
  build :
  {
    LAPACK_ROOT_DIR : $install_dir
  }
  launch :
  {
    LAPACK_ROOT_DIR : $install_dir
    _LD_LIBRARY_PATH : $install_dir + $VARS.sep + "lib"
    PYTHONPATH_ : $install_dir + $VARS.sep + "lib"
  }
}
```

3. The last possibility is to set the environment with a python script. The script should be provided in the *products/env_scripts* directory of the sat project, and its name is specified in the environment section with the key `environ.env_script`:

```
environ :
{
  env_script : 'lapack.py'
}
```

Please note that the two modes are complementary and are both taken into account. Most of the time, the first mode is sufficient.

The second mode can be used when the environment has to be set programmatically. The developer implements a handle (as a python method) which is called by sat to set the environment. Here is an example:

```python
#!/usr/bin/env python
#-*- coding:utf-8 -*-

import os.path
import platform

def set_env(env, prereq_dir, version):
    env.set("TRUST_ROOT_DIR",prereq_dir)
    env.prepend('PATH', os.path.join(prereq_dir, 'bin'))
    env.prepend('PATH', os.path.join(prereq_dir, 'include'))
    env.prepend('LD_LIBRARY_PATH', os.path.join(prereq_dir, 'lib'))
    return
```

SalomeTools defines four handles:

- **set_env(env, prereq_dir, version)** : used at build and run time.
- **set_env_launch(env, prereq_dir, version)** : used only at run time (if defined!)
- **set_env_build(env, prereq_dir, version)** : used only at build time (if defined!)
- **set_native_env(env)** : used only for native products, at build and run time.

2.9 Command clean

2.9.1 Description

The **clean** command removes products in the *source, build, or install* directories of an application. Theses directories are usually named `SOURCES, BUILD, INSTALL`.

Use the options to define what directories you want to suppress and to set the list of products

2.9.2 Usage

- Clean all previously created *build* and *install* directories (example application as *SALOME_xx*):

```
# take care, is long time to restore, sometimes
sat clean SALOME-xx --build --install
```

- Clean previously created *build* and *install* directories, only for products with property *is_salome_module*:

```
sat clean SALOME-xxx --build --install \
                     --properties is_salome_module:yes
```

2.9.3 Availables options

- **–products** : Products to clean.
- **–properties** :

  Filter the products by their properties.
  Syntax: *–properties <property>:<value>*

- **–sources** : Clean the product source directories.
- **–build** : Clean the product build directories.
- **–install** : Clean the product install directories.
- **–all** : Clean the product source, build and install directories.
- **–sources_without_dev** :

  Do not clean the products in development mode,
  (they could have VCS[13] commits pending).

2.9.4 Some useful configuration pathes

No specific configuration.

[13] https://en.wikipedia.org/wiki/Version_control

2.10 Command package

2.10.1 Description

The **package** command creates a SALOME[14] archive (usually a compressed Tar[15] file .tgz). This tar file is used later to intall SALOME on other remote computer.

Depending on the selected options, the archive includes sources and binaries of SALOME products and prerequisites.

Usually utility *salomeTools* is included in the archive.

Note: By default the package includes the sources of prerequisites and products. To select a subset use the *–without_property* or *–with_vcs* options.

2.10.2 Usage

- Create a package for a product (example as *SALOME_xx*):

  ```
  sat package SALOME_xx
  ```

  This command will create an archive named `SALOME_xx.tgz` in the working directory (`USER.workDir`). If the archive already exists, do nothing.

- Create a package with a specific name:

  ```
  sat package SALOME_xx --name YourSpecificName
  ```

Note: By default, the archive is created in the working directory of the user (`USER.workDir`).

If the option *–name* is used with a path (relative or absolute) it will be used.

If the option *–name* is not used and binaries (prerequisites and products) are included in the package, the OS[16] architecture will be appended to the name (example: `SALOME_xx-CO7.tgz`).

Examples:

```
# Creates SALOME_xx.tgz in $USER.workDir
sat package SALOME_xx

# Creates SALOME_xx_<arch>.tgz in $USER.workDir
sat package SALOME_xx --binaries

# Creates MySpecificName.tgz in $USER.workDir
sat package SALOME_xx --name MySpecificName
```

- Force the creation of the archive (if it already exists):

  ```
  sat package SALOME_xx --force
  ```

- Include the binaries in the archive (products and prerequisites):

  ```
  sat package SALOME_xx --binaries
  ```

  This command will create an archive named `SALOME_xx _<arch>.tgz` where <arch> is the OS[17] ar-

[14] http://www.salome-platform.org
[15] https://en.wikipedia.org/wiki/Tar_(computing)
[16] https://en.wikipedia.org/wiki/Operating_system
[17] https://en.wikipedia.org/wiki/Operating_system

chitecture of the machine.

- Do not delete Version Control System (VCS[18]) informations from the configurations files of the embedded salomeTools:

  ```
  sat package SALOME_xx --with_vcs
  ```

  The version control systems known by this option are CVS[19], SVN[20] and Git[21].

2.10.3 Some useful configuration pathes

No specific configuration.

[18] https://en.wikipedia.org/wiki/Version_control
[19] https://fr.wikipedia.org/wiki/Concurrent_versions_system
[20] https://en.wikipedia.org/wiki/Apache_Subversion
[21] https://git-scm.com

2.11 Command generate

2.11.1 Description

The **generate** command generates and compile SALOME modules from cpp modules using YACSGEN.

Note: This command uses YACSGEN to generate the module. It needs to be specified with *–yacsgen* option, or defined in the product or by the environment variable `$YACSGEN_ROOT_DIR`.

2.11.2 Remarks

- This command will only apply on the CPP modules of the application, those who have both properties:

```
cpp : "yes"
generate : "yes"
```

- The cpp module are usually computational components, and the generated module brings the CORBA layer which allows distributing the compononent on remore machines. cpp modules should conform to YACSGEN/hxx2salome requirements (please refer to YACSGEN documentation)

2.11.3 Usage

- Generate all the modules of a product:

```
sat generate <application>
```

- Generate only specific modules:

```
sat generate <application> --products <list_of_products>
```

  Remark: modules which don't have the *generate* property are ignored.

- Use a specific version of YACSGEN:

```
sat generate <application> --yacsgen <path_to_yacsgen>
```

CHAPTER THREE

DEVELOPER DOCUMENTATION

3.1 Add a user custom command

3.1.1 Introduction

Note: This documentation is for Python[22] developers.

The salomeTools product provides a simple way to develop commands. The first thing to do is to add a file with *.py* extension in the `commands` directory of salomeTools.

Here are the basic requirements that must be followed in this file in order to add a command.

3.1.2 Basic requirements

By adding a file *mycommand.py* in the `commands` directory, salomeTools will define a new command named `mycommand`.

In *mycommand.py*, there must be the following method:

```
def run(args, runner, logger):
    # your algorithm ...
    pass
```

In fact, at this point, the command will already be functional. But there are some useful services provided by salomeTools :

- You can give some options to your command:

```
import src

# Define all possible option for mycommand command :  'sat mycommand <options>'
parser = src.options.Options()
parser.add_option('m', 'myoption', \
                  'boolean', 'myoption', \
                  'My option changes the behavior of my command.')

def run(args, runner, logger):
    # Parse the options
    (options, args) = parser.parse_args(args)
    # algorithm
```

- You can add a *description* method that will display a message when the user will call the help:

```
import src

# Define all possible option for mycommand command : 'sat mycommand <options>'
parser = src.options.Options()
parser.add_option('m', 'myoption', \
                  'boolean', 'myoption', \
                  'My option changes the behavior of my command.')

def description():
    return _("The help of mycommand.")

def run(args, runner, logger):
    # Parse the options
    (options, args) = parser.parse_args(args)
    # algorithm
```

[22] https://docs.python.org/2.7

3.1.3 HowTo access salomeTools config and other commands

The *runner* variable is an python instance of *Sat* class. It gives access to *runner.cfg* which is the data model defined from all *configuration pyconf files* of salomeTools For example, *runner.cfg.APPLICATION.workdir* contains the root directory of the current application.

The *runner* variable gives also access to other commands of salomeTools:

```
# as CLI_ 'sat prepare ...'
runner.prepare(runner.cfg.VARS.application)
```

3.1.4 HowTo logger

The logger variable is an instance of the Logger class. It gives access to the write method.

When this method is called, the message passed as parameter will be displayed in the terminal and written in an xml log file.

```
logger.write("My message", 3) # 3 as default
```

The second argument defines the level of verbosity that is wanted for this message. It has to be between 1 and 5 (the most verbose level).

3.1.5 HELLO example

Here is a *hello* command, file *commands/hello.py*:

```
import src

"""
hello.py
Define all possible options for hello command:
sat hello <options>
"""

parser = src.options.Options()
parser.add_option('f', 'french', 'boolean', 'french', "french set hello message in␣
↪french.")

def description():
    return _("The help of hello.")

def run(args, runner, logger):
    # Parse the options
    (options, args) = parser.parse_args(args)
    # algorithm
    if not options.french:
        logger.write('HELLO! WORLD!\n')
    else:
        logger.write('Bonjour tout le monde!\n')
```

A first call of hello:

```
# Get the help of hello:
./sat --help hello

# To get bonjour
./sat hello --french
Bonjour tout le monde!
```

(continues on next page)

(continued from previous page)

```
# To get hello
./sat hello
HELLO! WORLD!

# To get the log
./sat log
```

CHAPTER

FOUR

CODE DOCUMENTATION

4.1 src

4.1.1 src package

Subpackages

src.colorama package

Submodules

src.colorama.ansi module

This module generates ANSI character codes to printing colors to terminals. See: http://en.wikipedia.org/wiki/ANSI_escape_code

class src.colorama.ansi.**AnsiBack**

Bases: *src.colorama.ansi.AnsiCodes* (page 29)

BLACK = 40

BLUE = 44

CYAN = 46

GREEN = 42

LIGHTBLACK_EX = 100

LIGHTBLUE_EX = 104

LIGHTCYAN_EX = 106

LIGHTGREEN_EX = 102

LIGHTMAGENTA_EX = 105

LIGHTRED_EX = 101

LIGHTWHITE_EX = 107

LIGHTYELLOW_EX = 103

MAGENTA = 45

RED = 41

RESET = 49

WHITE = 47

YELLOW = 43

class `src.colorama.ansi.`**`AnsiCodes`**

Bases: `object`

class `src.colorama.ansi.`**`AnsiCursor`**

Bases: `object`

`BACK`(*n=1*)

`DOWN`(*n=1*)

`FORWARD`(*n=1*)

`POS`(*x=1*, *y=1*)

`UP`(*n=1*)

class `src.colorama.ansi.`**`AnsiFore`**

Bases: *`src.colorama.ansi.AnsiCodes`* (page 29)

`BLACK = 30`

`BLUE = 34`

`CYAN = 36`

`GREEN = 32`

`LIGHTBLACK_EX = 90`

`LIGHTBLUE_EX = 94`

`LIGHTCYAN_EX = 96`

`LIGHTGREEN_EX = 92`

`LIGHTMAGENTA_EX = 95`

`LIGHTRED_EX = 91`

`LIGHTWHITE_EX = 97`

`LIGHTYELLOW_EX = 93`

`MAGENTA = 35`

`RED = 31`

`RESET = 39`

`WHITE = 37`

`YELLOW = 33`

class `src.colorama.ansi.`**`AnsiStyle`**

Bases: *`src.colorama.ansi.AnsiCodes`* (page 29)

`BRIGHT = 1`

`DIM = 2`

`NORMAL = 22`

`RESET_ALL = 0`

`src.colorama.ansi.`**`clear_line`**(*mode=2*)

`src.colorama.ansi.`**`clear_screen`**(*mode=2*)

`src.colorama.ansi.`**`code_to_chars`**(*code*)

`src.colorama.ansi.`**`set_title`**(*title*)

src.colorama.ansitowin32 module

class src.colorama.ansitowin32.**AnsiToWin32**(*wrapped*, *convert=None*, *strip=None*, *autoreset=False*)

Bases: object

Implements a 'write()' method which, on Windows, will strip ANSI character sequences from the text, and if outputting to a tty, will convert them into win32 function calls.

ANSI_CSI_RE = <_sre.SRE_Pattern object>

ANSI_OSC_RE = <_sre.SRE_Pattern object>

call_win32(*command*, *params*)

convert_ansi(*paramstring*, *command*)

convert_osc(*text*)

extract_params(*command*, *paramstring*)

get_win32_calls()

reset_all()

should_wrap()

True if this class is actually needed. If false, then the output stream will not be affected, nor will win32 calls be issued, so wrapping stdout is not actually required. This will generally be False on non-Windows platforms, unless optional functionality like autoreset has been requested using kwargs to init()

write(*text*)

write_and_convert(*text*)

Write the given text to our wrapped stream, stripping any ANSI sequences from the text, and optionally converting them into win32 calls.

write_plain_text(*text*, *start*, *end*)

class src.colorama.ansitowin32.**StreamWrapper**(*wrapped*, *converter*)

Bases: object

Wraps a stream (such as stdout), acting as a transparent proxy for all attribute access apart from method 'write()', which is delegated to our Converter instance.

write(*text*)

src.colorama.ansitowin32.**is_a_tty**(*stream*)

src.colorama.ansitowin32.**is_stream_closed**(*stream*)

src.colorama.initialise module

src.colorama.initialise.**colorama_text**(**args*, ***kwds*)

src.colorama.initialise.**deinit**()

src.colorama.initialise.**init**(*autoreset=False*, *convert=None*, *strip=None*, *wrap=True*)

src.colorama.initialise.**reinit**()

src.colorama.initialise.**reset_all**()

src.colorama.initialise.**wrap_stream**(*stream*, *convert*, *strip*, *autoreset*, *wrap*)

src.colorama.win32 module

src.colorama.win32.**SetConsoleTextAttribute**(*_)

src.colorama.win32.**winapi_test**(*_)

src.colorama.winterm module

class src.colorama.winterm.**WinColor**

Bases: object

BLACK = 0

BLUE = 1

CYAN = 3

GREEN = 2

GREY = 7

MAGENTA = 5

RED = 4

YELLOW = 6

class src.colorama.winterm.**WinStyle**

Bases: object

BRIGHT = 8

BRIGHT_BACKGROUND = 128

NORMAL = 0

class src.colorama.winterm.**WinTerm**

Bases: object

back(*back=None*, *light=False*, *on_stderr=False*)

cursor_adjust(*x*, *y*, *on_stderr=False*)

erase_line(*mode=0*, *on_stderr=False*)

erase_screen(*mode=0*, *on_stderr=False*)

fore(*fore=None*, *light=False*, *on_stderr=False*)

get_attrs()

get_position(*handle*)

reset_all(*on_stderr=None*)

set_attrs(*value*)

set_console(*attrs=None*, *on_stderr=False*)

set_cursor_position(*position=None*, *on_stderr=False*)

set_title(*title*)

style(*style=None*, *on_stderr=False*)

Module contents

Submodules

src.ElementTree module

`src.ElementTree.`**`Comment`**(*text=None*)

`src.ElementTree.`**`dump`**(*elem*)

`src.ElementTree.`**`Element`**(*tag*, *attrib={}*, ***extra*)

class `src.ElementTree.`**`ElementTree`**(*element=None*, *file=None*)

> **`find`**(*path*)
>
> **`findall`**(*path*)
>
> **`findtext`**(*path*, *default=None*)
>
> **`getiterator`**(*tag=None*)
>
> **`getroot`**()
>
> **`parse`**(*source*, *parser=None*)
>
> **`write`**(*file*, *encoding='us-ascii'*)

`src.ElementTree.`**`fromstring`**(*text*)

`src.ElementTree.`**`iselement`**(*element*)

class `src.ElementTree.`**`iterparse`**(*source*, *events=None*)

> **`next`**()

`src.ElementTree.`**`parse`**(*source*, *parser=None*)

`src.ElementTree.`**`PI`**(*target*, *text=None*)

`src.ElementTree.`**`ProcessingInstruction`**(*target*, *text=None*)

class `src.ElementTree.`**`QName`**(*text_or_uri*, *tag=None*)

`src.ElementTree.`**`SubElement`**(*parent*, *tag*, *attrib={}*, ***extra*)

`src.ElementTree.`**`tostring`**(*element*, *encoding=None*)

class `src.ElementTree.`**`TreeBuilder`**(*element_factory=None*)

> **`close`**()
>
> **`data`**(*data*)
>
> **`end`**(*tag*)
>
> **`start`**(*tag*, *attrs*)

`src.ElementTree.`**`XML`**(*text*)

class `src.ElementTree.`**`XMLTreeBuilder`**(*html=0*, *target=None*)

> **`close`**()
>
> **`doctype`**(*name*, *pubid*, *system*)
>
> **`feed`**(*data*)

src.architecture module

In this file : all the stuff that can change with the architecture on which SAT is running

`src.architecture.`**`get_distrib_version`**(*distrib*, *codes*)
Gets the version of the distribution

Parameters

- **`str`** (*`distrib`*) – The distribution on which the version will be found.
- **`L{Mapping}`** (*`codes`*) – The map containing distribution correlation table.

Returns The version of the distribution on which salomeTools is running, regarding the distribution correlation table contained in codes variable.

Return type str

`src.architecture.`**`get_distribution`**(*codes*)
Gets the code for the distribution

Parameters **`L{Mapping}`** (*`codes`*) – The map containing distribution correlation table.

Returns The distribution on which salomeTools is running, regarding the distribution correlation table contained in codes variable.

Return type str

`src.architecture.`**`get_nb_proc`**()

Gets the number of processors of the machine on which salomeTools is running.

Returns the number of processors.

Return type str

`src.architecture.`**`get_python_version`**()
Gets the version of the running python.

Returns the version of the running python.

Return type str

`src.architecture.`**`get_user`**()
method that gets the username that launched sat

Return type str

`src.architecture.`**`is_windows`**()
method that checks windows OS

Return type boolean

src.compilation module

`class` `src.compilation.`**`Builder`**(*config*, *logger*, *product_info*, *options=OptResult( )*, *check_src=True*)
Class to handle all construction steps, like cmake, configure, make, …

`build_configure`(*options=''*)

`check`(*command=''*)

`cmake`(*options=''*)

`complete_environment`(*make_options*)

`configure`(*options=''*)

`do_batch_script_build`(*script*, *nb_proc*)

do_default_build(*build_conf_options=”, configure_options=”, show_warning=True*)

do_python_script_build(*script, nb_proc*)

do_script_build(*script, number_of_proc=0*)

hack_libtool()

install()

log(*text, level, showInfo=True*)

log_command(*command*)

make(*nb_proc, make_opt=”*)

prepare()

put_txt_log_in_appli_log_dir(*file_name*)

Put the txt log (that contain the system logs, like make command output) in the directory <APPLICATION DIR>/LOGS/<product_name>/

Parameters **Str** (*file_name*) – the name of the file to write

wmake(*nb_proc, opt_nb_proc=None*)

src.debug module

This file assume DEBUG functionalities use

- print debug messages in sys.stderr for salomeTools
- show pretty print debug representation from instances of SAT classes (pretty print src.pyconf.Config), and python dict/list etc. (as ‘aVariable’)

WARNING: obviously supposedly show messages in SAT development phase, not production

usage: >> import debug as DBG >> DBG.write(“aTitle”, aVariable) # not shown in production >> DBG.write(“aTitle”, aVariable, True) # unconditionaly shown (as show=True)

to set show message as development phase: >> DBG.push_debug(True)

to set no show message as production phase: >> DBG.push_debug(False)

to set show message temporary as development phase, only in a method: >> def aMethodToDebug(…): >> DBG.push_debug(True) #force show as appended status >> etc. method code with some DBG.write() >> DBG.pop_debug() #restore previous status (show or not show) >> return

to set a message for future fix, as temporary problem to not forget: DBG.tofix(“aTitle”, aVariable, True/False) #True/False in production shown, or not

in command line interface you could redirect stderr to file ‘myDebug.log’: >> sat compile … 2> myDebug.log # only stderr >> sat compile … &> myDebug.log # stdout and stderr

class src.debug.**InStream**(*buf=”*)
Bases: StringIO.StringIO

utility class for pyconf.Config input iostream

class src.debug.**OutStream**(*buf=”*)
Bases: StringIO.StringIO

utility class for pyconf.Config output iostream

close()
because Config.__save__ calls close() stream as file keep value before lost as self.value

src.debug.**getLocalEnv**()
get string for environment variables representation

`src.debug.`**`getStrConfigDbg`**(*config*)
set string as saveConfigDbg, as (path expression evaluation) for debug

`src.debug.`**`getStrConfigStd`**(*config*)
set string as saveConfigStd, as file .pyconf

`src.debug.`**`indent`**(*text*, *amount=2*, *ch=' '*)
indent multi lines message

`src.debug.`**`pop_debug`**()
restore previous debug outputs status

`src.debug.`**`push_debug`**(*aBool*)
set debug outputs activated, or not

`src.debug.`**`saveConfigDbg`**(*config*, *aStream*, *indent=0*, *path="*)
pyconf returns multilines (path expression evaluation) for debug

`src.debug.`**`saveConfigStd`**(*config*, *aStream*)
returns as file .pyconf

`src.debug.`**`tofix`**(*title*, *var="*, *force=None*)
write sys.stderr a message if _debug[-1]==True or optionaly force=True use this only if no logger accessible for classic logger.warning(message) or logger.debug(message)

`src.debug.`**`write`**(*title*, *var="*, *force=None*, *fmt='\n#### DEBUG: %s:\n%s\n'*)
write sys.stderr a message if _debug[-1]==True or optionaly force=True

src.environment module

`class` `src.environment.`**`Environ`**(*environ=None*)
Class to manage the environment context

`append`(*key*, *value*, *sep=':'*)
Same as append_value but the value argument can be a list

Parameters

- **`str`** (*`sep`*) – the environment variable to append
- **`str or list`** (*`value`*) – the value(s) to append to key
- **`str`** – the separator string

`append_value`(*key*, *value*, *sep=':'*)
append value to key using sep

Parameters

- **`str`** (*`sep`*) – the environment variable to append
- **`str`** – the value to append to key
- **`str`** – the separator string

`command_value`(*key*, *command*)
Get the value given by the system command "command" and put it in the environment variable key

Parameters

- **`str`** (*`command`*) – the environment variable
- **`str`** – the command to execute

`get`(*key*)
Get the value of the environment variable "key"

Parameters **`str`** (*`key`*) – the environment variable

is_defined(*key*)
Check if the key exists in the environment

Parameters **str** (*key*) – the environment variable to check

prepend(*key*, *value*, *sep=':'*)
Same as prepend_value but the value argument can be a list

Parameters

- **str** (*sep*) – the environment variable to prepend
- **str or list** (*value*) – the value(s) to prepend to key
- **str** – the separator string

prepend_value(*key*, *value*, *sep=':'*)
prepend value to key using sep

Parameters

- **str** (*sep*) – the environment variable to prepend
- **str** – the value to prepend to key
- **str** – the separator string

set(*key*, *value*)
Set the environment variable "key" to value "value"

Parameters

- **str** (*value*) – the environment variable to set
- **str** – the value

class src.environment.**FileEnvWriter**(*config*, *logger*, *out_dir*, *src_root*, *env_info=None*)
Class to dump the environment to a file.

write_cfgForPy_file(*filename*, *additional_env={}*, *for_package=None*, *with_commercial=True*)
Append to current opened aFile a cfgForPy environment (SALOME python launcher).

Parameters

- **str** (*for_package*) – the file path
- **dict** (*additional_env*) – a dictionary of additional variables to add to the environment
- **str** – If not None, produce a relative environment designed for a package.

write_env_file(*filename*, *forBuild*, *shell*, *for_package=None*)
Create an environment file.

Parameters

- **str** (*shell*) – the file path
- **bool** (*forBuild*) – if true, the build environment
- **str** – the type of file wanted (.sh, .bat)

Returns The path to the generated file

Return type str

class src.environment.**SalomeEnviron**(*cfg*, *environ*, *forBuild=False*, *for_package=None*, *enable_simple_env_script=True*)
Class to manage the environment of SALOME.

add_comment(*comment*)
Add a commentary to the out stream (in case of file generation)

Parameters **`str`** (*`comment`*) – the commentary to add

`add_line`(*nb_line*)
Add empty lines to the out stream (in case of file generation)

Parameters **`int`** (*`nb_line`*) – the number of empty lines to add

`add_warning`(*warning*)
Add a warning to the out stream (in case of file generation)

Parameters **`str`** (*`warning`*) – the warning to add

`append`(*key*, *value*, *sep=':'*)
append value to key using sep

Parameters

- **`str`** (*`sep`*) – the environment variable to append
- **`str`** – the value to append to key
- **`str`** – the separator string

`dump`(*out*)
Write the environment to out

Parameters **`file`** (*`out`*) – the stream where to write the environment

`finish`(*required*)
Add a final instruction in the out file (in case of file generation)

Parameters **`bool`** (*`required`*) – Do nothing if required is False

`get`(*key*)
Get the value of the environment variable "key"

Parameters **`str`** (*`key`*) – the environment variable

`get_names`(*lProducts*)
Get the products name to add in SALOME_MODULES environment variable It is the name of the product, except in the case where the is a component name. And it has to be in SALOME_MODULES variable only if the product has the property has_salome_hui = "yes"

Parameters **`list`** (*`lProducts`*) – List of products to potentially add

`is_defined`(*key*)
Check if the key exists in the environment

Parameters **`str`** (*`key`*) – the environment variable to check

`load_cfg_environment`(*cfg_env*)
Loads environment defined in cfg_env

Parameters **`Config`** (*`cfg_env`*) – A config containing an environment

`prepend`(*key*, *value*, *sep=':'*)
prepend value to key using sep

Parameters

- **`str`** (*`sep`*) – the environment variable to prepend
- **`str`** – the value to prepend to key
- **`str`** – the separator string

`run_env_script`(*product_info*, *logger=None*, *native=False*)
Runs an environment script.

Parameters

- **`Config`** (*`product_info`*) – The product description

- **Logger** (*logger*) – The logger instance to display messages
- **Boolean** (*native*) – if True load set_native_env instead of set_env

run_simple_env_script (*script_path*, *logger=None*)
Runs an environment script. Same as run_env_script, but with a script path as parameter.

Parameters

- **str** (*script_path*) – a path to an environment script
- **Logger** (*logger*) – The logger instance to display messages

set (*key*, *value*)
Set the environment variable "key" to value "value"

Parameters

- **str** (*value*) – the environment variable to set
- **str** – the value

set_a_product (*product*, *logger*)
Sets the environment of a product.

Parameters

- **str** (*product*) – The product name
- **Logger** (*logger*) – The logger instance to display messages

set_application_env (*logger*)
Sets the environment defined in the APPLICATION file.

Parameters **Logger** (*logger*) – The logger instance to display messages

set_cpp_env (*product_info*)
Sets the generic environment for a SALOME cpp product.

Parameters **Config** (*product_info*) – The product description

set_full_environ (*logger*, *env_info*)
Sets the full environment for products specified in env_info dictionary.

Parameters

- **Logger** (*logger*) – The logger instance to display messages
- **list** (*env_info*) – the list of products

set_products (*logger*, *src_root=None*)
Sets the environment for all the products.

Parameters

- **Logger** (*logger*) – The logger instance to display messages
- **src** (*src_root*) – the application working directory

set_python_libdirs ()
Set some generic variables for python library paths

set_salome_generic_product_env (*pi*)
Sets the generic environment for a SALOME product.

Parameters **Config** (*pi*) – The product description

set_salome_minimal_product_env (*product_info*, *logger*)
Sets the minimal environment for a SALOME product. xxx_ROOT_DIR and xxx_SRC_DIR

Parameters

- **Config** (*product_info*) – The product description

- **Logger** (*logger*) – The logger instance to display messages

class src.environment.**Shell** (*name*, *extension*)
Definition of a Shell.

src.environment.**load_environment** (*config*, *build*, *logger*)
Loads the environment (used to run the tests, for example).

Parameters

- **Config** (*config*) – the global config
- **bool** (*build*) – build environement if True
- **Logger** (*logger*) – The logger instance to display messages

src.fileEnviron module

class src.fileEnviron.**BashFileEnviron** (*output*, *environ=None*)
Bases: *src.fileEnviron.FileEnviron* (page 42)

Class for bash shell.

command_value (*key*, *command*)
Get the value given by the system command "command" and put it in the environment variable key. Has to be overwritten in the derived classes This can be seen as a virtual method

Parameters

- **str** (*command*) – the environment variable
- **str** – the command to execute

finish (*required=True*)
Add a final instruction in the out file (in case of file generation)

Parameters bool (*required*) – Do nothing if required is False

set (*key*, *value*)
Set the environment variable "key" to value "value"

Parameters

- **str** (*value*) – the environment variable to set
- **str** – the value

class src.fileEnviron.**BatFileEnviron** (*output*, *environ=None*)
Bases: *src.fileEnviron.FileEnviron* (page 42)

for Windows batch shell.

add_comment (*comment*)
Add a comment in the shell file

Parameters str (*comment*) – the comment to add

command_value (*key*, *command*)
Get the value given by the system command "command" and put it in the environment variable key. Has to be overwritten in the derived classes This can be seen as a virtual method

Parameters

- **str** (*command*) – the environment variable
- **str** – the command to execute

finish (*required=True*)
Add a final instruction in the out file (in case of file generation) In the particular windows case, do nothing

Parameters **`bool`** (`required`) – Do nothing if required is False

`get` (*key*)
Get the value of the environment variable "key"

Parameters **`str`** (`key`) – the environment variable

`set` (*key*, *value*)
Set the environment variable "key" to value "value"

Parameters

- **`str`** (`value`) – the environment variable to set
- **`str`** – the value

class `src.fileEnviron.`**`ContextFileEnviron`** (*output*, *environ=None*)
Bases: *`src.fileEnviron.FileEnviron`* (page 42)

Class for a salome context configuration file.

`add_echo` (*text*)
Add a comment

Parameters **`str`** (`text`) – the comment to add

`add_warning` (*warning*)
Add a warning

Parameters **`str`** (`text`) – the warning to add

`append_value` (*key*, *value*, *sep=':'*)
append value to key using sep

Parameters

- **`str`** (`sep`) – the environment variable to append
- **`str`** – the value to append to key
- **`str`** – the separator string

`command_value` (*key*, *command*)
Get the value given by the system command "command" and put it in the environment variable key. Has to be overwritten in the derived classes This can be seen as a virtual method

Parameters

- **`str`** (`command`) – the environment variable
- **`str`** – the command to execute

`finish` (*required=True*)
Add a final instruction in the out file (in case of file generation)

Parameters **`bool`** (`required`) – Do nothing if required is False

`get` (*key*)
Get the value of the environment variable "key"

Parameters **`str`** (`key`) – the environment variable

`prepend_value` (*key*, *value*, *sep=':'*)
prepend value to key using sep

Parameters

- **`str`** (`sep`) – the environment variable to prepend
- **`str`** – the value to prepend to key
- **`str`** – the separator string

set (*key*, *value*)
Set the environment variable "key" to value "value"

Parameters

- **str** (`value`) – the environment variable to set
- **str** – the value

class `src.fileEnviron.`**FileEnviron** (*output*, *environ=None*)
Bases: `object`

Base class for shell environment

add_comment (*comment*)
Add a comment in the shell file

Parameters **str** (`comment`) – the comment to add

add_echo (*text*)
Add a "echo" in the shell file

Parameters **str** (`text`) – the text to echo

add_line (*number*)
Add some empty lines in the shell file

Parameters **int** (`number`) – the number of lines to add

add_warning (*warning*)
Add a warning "echo" in the shell file

Parameters **str** (`warning`) – the text to echo

append (*key*, *value*, *sep=':'*)
Same as append_value but the value argument can be a list

Parameters

- **str** (`sep`) – the environment variable to append
- **str or list** (`value`) – the value(s) to append to key
- **str** – the separator string

append_value (*key*, *value*, *sep=':'*)
append value to key using sep

Parameters

- **str** (`sep`) – the environment variable to append
- **str** – the value to append to key
- **str** – the separator string

command_value (*key*, *command*)
Get the value given by the system command "command" and put it in the environment variable key. Has to be overwritten in the derived classes This can be seen as a virtual method

Parameters

- **str** (`command`) – the environment variable
- **str** – the command to execute

finish (*required=True*)
Add a final instruction in the out file (in case of file generation)

Parameters **bool** (`required`) – Do nothing if required is False

get (*key*)
Get the value of the environment variable "key"

Parameters **`str`** (*`key`*) – the environment variable

`is_defined`(*key*)
Check if the key exists in the environment

Parameters **`str`** (*`key`*) – the environment variable to check

`prepend`(*key*, *value*, *sep=':'*)
Same as prepend_value but the value argument can be a list

Parameters

- **`str`** (*`sep`*) – the environment variable to prepend
- **`str or list`** (*`value`*) – the value(s) to prepend to key
- **`str`** – the separator string

`prepend_value`(*key*, *value*, *sep=':'*)
prepend value to key using sep

Parameters

- **`str`** (*`sep`*) – the environment variable to prepend
- **`str`** – the value to prepend to key
- **`str`** – the separator string

`set`(*key*, *value*)
Set the environment variable 'key' to value 'value'

Parameters

- **`str`** (*`value`*) – the environment variable to set
- **`str`** – the value

`class` `src.fileEnviron.`**`LauncherFileEnviron`**(*output*, *environ=None*)
Class to generate a launcher file script (in python syntax) SalomeContext API

`add`(*key*, *value*)
prepend value to key using sep

Parameters

- **`str`** (*`value`*) – the environment variable to prepend
- **`str`** – the value to prepend to key

`add_comment`(*comment*)

`add_echo`(*text*)
Add a comment

Parameters **`str`** (*`text`*) – the comment to add

`add_line`(*number*)
Add some empty lines in the launcher file

Parameters **`int`** (*`number`*) – the number of lines to add

`add_warning`(*warning*)
Add a warning

Parameters **`str`** (*`text`*) – the warning to add

`append`(*key*, *value*, *sep=':'*)
Same as append_value but the value argument can be a list

Parameters

- **`str`** (*`sep`*) – the environment variable to append

- **str or list** (*value*) – the value(s) to append to key
- **str** – the separator string

append_value (*key*, *value*, *sep=':'*)
append value to key using sep

Parameters

- **str** (*sep*) – the environment variable to append
- **str** – the value to append to key
- **str** – the separator string

change_to_launcher (*value*)

command_value (*key*, *command*)
Get the value given by the system command "command" and put it in the environment variable key.

Parameters

- **str** (*command*) – the environment variable
- **str** – the command to execute

finish (*required=True*)
Add a final instruction in the out file (in case of file generation) In the particular launcher case, do nothing

Parameters **bool** (*required*) – Do nothing if required is False

get (*key*)
Get the value of the environment variable "key"

Parameters **str** (*key*) – the environment variable

is_defined (*key*)
Check if the key exists in the environment

Parameters **str** (*key*) – the environment variable to check

prepend (*key*, *value*, *sep=':'*)
Same as prepend_value but the value argument can be a list

Parameters

- **str** (*sep*) – the environment variable to prepend
- **str or list** (*value*) – the value(s) to prepend to key
- **str** – the separator string

prepend_value (*key*, *value*, *sep=':'*)
prepend value to key using sep

Parameters

- **str** (*sep*) – the environment variable to prepend
- **str** – the value to prepend to key
- **str** – the separator string

set (*key*, *value*)
Set the environment variable "key" to value "value"

Parameters

- **str** (*value*) – the environment variable to set
- **str** – the value

class src.fileEnviron.**ScreenEnviron**(*output*, *environ=None*)
Bases: *src.fileEnviron.FileEnviron* (page 42)

add_comment(*comment*)
Add a comment in the shell file

Parameters **str** (*comment*) – the comment to add

add_echo(*text*)
Add a "echo" in the shell file

Parameters **str** (*text*) – the text to echo

add_line(*number*)
Add some empty lines in the shell file

Parameters **int** (*number*) – the number of lines to add

add_warning(*warning*)
Add a warning "echo" in the shell file

Parameters **str** (*warning*) – the text to echo

append(*name*, *value*, *sep=':'*)
Same as append_value but the value argument can be a list

Parameters

- **str** (*sep*) – the environment variable to append
- **str or list** (*value*) – the value(s) to append to key
- **str** – the separator string

command_value(*key*, *command*)
Get the value given by the system command "command" and put it in the environment variable key. Has to be overwritten in the derived classes This can be seen as a virtual method

Parameters

- **str** (*command*) – the environment variable
- **str** – the command to execute

get(*name*)
Get the value of the environment variable "key"

Parameters **str** (*key*) – the environment variable

is_defined(*name*)
Check if the key exists in the environment

Parameters **str** (*key*) – the environment variable to check

prepend(*name*, *value*, *sep=':'*)
Same as prepend_value but the value argument can be a list

Parameters

- **str** (*sep*) – the environment variable to prepend
- **str or list** (*value*) – the value(s) to prepend to key
- **str** – the separator string

run_env_script(*module*, *script*)

set(*name*, *value*)
Set the environment variable 'key' to value 'value'

Parameters

- **str** (*value*) – the environment variable to set

- **str** – the value

write (*command*, *name*, *value*, *sign='='*)

src.fileEnviron.**get_file_environ** (*output*, *shell*, *environ=None*)
Instantiate correct FileEnvironment sub-class.

Parameters

- **file** (*output*) – the output file stream.
- **str** (*shell*) – the type of shell syntax to use.
- **dict** (*environ*) – a potential additional environment.

src.fileEnviron.**special_path_separator** (*name*)
TCLLIBPATH, TKLIBPATH, PV_PLUGIN_PATH environments variables need some exotic path separator. This function gives the separator regarding the name of the variable to append or prepend.

Parameters **str** (*name*) – The name of the variable to find the separator

src.fork module

src.fork.**batch** (*cmd*, *logger*, *cwd*, *args=[]*, *log=None*, *delai=20*, *sommeil=1*)

src.fork.**batch_salome** (*cmd*, *logger*, *cwd*, *args*, *getTmpDir*, *pendant='SALOME_Session_Server'*, *fin='killSalome.py'*, *log=None*, *delai=20*, *sommeil=1*, *delaiapp=0*)

src.fork.**launch_command** (*cmd*, *logger*, *cwd*, *args=[]*, *log=None*)

src.fork.**show_progress** (*logger*, *top*, *delai*, *ss=''*)
shortcut function to display the progression

Parameters

- **Logger** (*logger*) – The logging instance
- **int** (*delai*) – the number to display
- **int** – the number max
- **str** (*ss*) – the string to display

src.fork.**write_back** (*logger*, *message*, *level*)
shortcut function to write at the begin of the line

Parameters

- **Logger** (*logger*) – The logging instance
- **str** (*message*) – the text to display
- **int** (*level*) – the level of verbosity

src.logger module

Implements the classes and method relative to the logging

class src.logger.**Logger** (*config*, *silent_sysstd=False*, *all_in_terminal=False*, *micro_command=False*)
Bases: object

Class to handle log mechanism.

add_link (*log_file_name*, *command_name*, *command_res*, *full_launched_command*)
Add a link to another log file.

Parameters

- **str** (*command_res*) – The file name of the link.

- **str** – The name of the command linked.
- **str** – The result of the command linked. "0" or "1"

Parma full_launched_command str The full lanch command ("sat command ...")

end_write (*attribute*)
Called just after command end: Put all fields corresponding to the command end context (time). Write the log xml file on the hard drive. And display the command to launch to get the log

Parameters **dict** (`attribute`) – the attribute to add to the node "Site".

error (*message*)
Print an error.

Parameters **str** (`message`) – The message to print.

flush ()
Flush terminal

put_initial_xml_fields ()
Called at class initialization: Put all fields corresponding to the command context (user, time, ...)

write (*message*, *level=None*, *screenOnly=False*)
function used in the commands to print in the terminal and the log file.

Parameters

- **str** (`message`) – The message to print.
- **int** (`level`) – The output level corresponding to the message 0 < level < 6.
- **boolean** (`screenOnly`) – if True, do not write in log file.

src.logger.**date_to_datetime** (*date*)
From a string date in format YYYYMMDD_HHMMSS returns list year, mon, day, hour, minutes, seconds

Parameters **str** (`date`) – The date in format YYYYMMDD_HHMMSS

Returns the same date and time in separate variables.

Return type (str,str,str,str,str,str)

src.logger.**list_log_file** (*dirPath*, *expression*)
Find all files corresponding to expression in dirPath

Parameters

- **str** (`expression`) – the directory where to search the files
- **str** – the regular expression of files to find

Returns the list of files path and informations about it

Return type list

src.logger.**show_command_log** (*logFilePath*, *cmd*, *application*, *notShownCommands*)
Used in updateHatXml. Determine if the log xml file logFilePath has to be shown or not in the hat log.

Parameters

- **str** (`application`) – the path to the command xml log file
- **str** – the command of the log file
- **str** – the application passed as parameter to the salomeTools command
- **list** (`notShownCommands`) – the list of commands that are not shown by default

Returns True if cmd is not in notShownCommands and the application in the log file corresponds to application

Return type boolean

`src.logger.`**`timedelta_total_seconds`**(*timedelta*)
Replace total_seconds from datetime module in order to be compatible with old python versions

Parameters **`datetime.timedelta`** (*`timedelta`*) – The delta between two dates

Returns The number of seconds corresponding to timedelta.

Return type float

`src.logger.`**`update_hat_xml`**(*logDir*, *application=None*, *notShownCommands=[]*)
Create the xml file in logDir that contain all the xml file and have a name like YYYYMMDD_HHMMSS_namecmd.xml

Parameters

- **`str`** (*`application`*) – the directory to parse
- **`str`** – the name of the application if there is any

src.options module

The Options class that manages the access to all options passed as parameters in salomeTools command lines

`class` `src.options.`**`OptResult`**
Bases: `object`

An instance of this class will be the object manipulated in code of all salomeTools commands The aim of this class is to have an elegant syntax to manipulate the options.

Example:
>> options, remainderArgs = command.parseArguments(args)
>> print(options.output_verbose_level)
>> 'INFO'

`class` `src.options.`**`Options`**
Bases: `object`

Class to manage all salomeTools options

`add_option`(*shortName*, *longName*, *optionType*, *destName*, *helpString=''*, *default=None*)
Add an option to a command. It gets all attributes of an option and append it in the options field

Parameters

- **`shortName`** – (str) The short name of the option (as '-l' for level option).
- **`longName`** – (str) The long name of the option (as '–level' for level option).
- **`optionType`** – (str) The type of the option (ex "int").
- **`destName`** – (str) The name that will be used in the code.
- **`helpString`** – (str) The text to display when user ask for help on a command.

Returns None

`debug_write`()

`filterLevel`(*aLevel*)
filter level logging values

`filterList2`(*aStr*)
filter a list as 'KERNEL,YACS,etc.'

`getDetailOption`(*option*)
for convenience

Returns (tuple) 4-elements (shortName, longName, optionType, helpString)

get_help()
Returns all options stored in self.options as help message colored string

Returns (str) colored string

indent(*text*, *amount*, *car=' '*)
indent multi lines message

parse_args(*argList=None*)
Instantiates the class OptResult that gives access to all options in the code

Parameters **argList** – (list) the raw list of arguments that were passed

Returns (OptResult, list) as (optResult, args) optResult is the option instance to manipulate in the code. args is the full raw list of passed options

print_help()
Method that display all options stored in self.options and there help

Returns None

src.printcolors module

In this file is stored the mechanism that manage color prints in the terminal

src.printcolors.**cleancolor**(*message*)
remove color from a colored text.

Parameters **str** (*message*) – The text to be cleaned.

Returns The cleaned text.

Return type str

src.printcolors.**print_color_map**()
This method prints the color map

src.printcolors.**print_color_range**(*start*, *end*)
print possible range values for colors

Parameters

- **int** (*end*) – The smaller value.
- **int** – The bigger value.

src.printcolors.**print_value**(*logger*, *label*, *value*, *level=1*, *suffix=''*)
shortcut method to print a label and a value with the info color

Parameters

- **class logger** (*logger*) – the logger instance.
- **int** (*level*) – the label to print.
- **str** (*suffix*) – the value to print.
- **int** – the level of verboseness.
- **str** – the suffix to add at the end.

src.printcolors.**printc**(*txt*, *code=''*)
print a text with colors

Parameters

- **str** (*code*) – The text to be printed.
- **str** – The color to use.

Returns The colored text.

Return type str

`src.printcolors.`**`printcError`**`(`*txt*`)`
print a text error color

Parameters **`str`** (`txt`) – The text to be printed.

Returns The colored text.

Return type str

`src.printcolors.`**`printcHeader`**`(`*txt*`)`
print a text header color

Parameters **`str`** (`txt`) – The text to be printed.

Returns The colored text.

Return type str

`src.printcolors.`**`printcHighlight`**`(`*txt*`)`
print a text highlight color

Parameters **`str`** (`txt`) – The text to be printed.

Returns The colored text.

Return type str

`src.printcolors.`**`printcInfo`**`(`*txt*`)`
print a text info color

Parameters **`str`** (`txt`) – The text to be printed.

Returns The colored text.

Return type str

`src.printcolors.`**`printcLabel`**`(`*txt*`)`
print a text label color

Parameters **`str`** (`txt`) – The text to be printed.

Returns The colored text.

Return type str

`src.printcolors.`**`printcSuccess`**`(`*txt*`)`
print a text success color

Parameters **`str`** (`txt`) – The text to be printed.

Returns The colored text.

Return type str

`src.printcolors.`**`printcWarning`**`(`*txt*`)`
print a text warning color

Parameters **`str`** (`txt`) – The text to be printed.

Returns The colored text.

Return type str

src.product module

In this file are implemented the methods relative to the product notion of salomeTools `src.product.`**`check_config_exists`**(*config*, *prod_dir*, *prod_info*)

Verify that the installation directory of a product in a base exists Check all the config-<i> directory and verify the sat-config.pyconf file that is in it

Parameters

- **`Config`** (*`product_info`*) – The global configuration
- **`str`** (*`prod_dir`*) – The product installation directory path (without config-<i>)
- **`Config`** – The configuration specific to the product

Returns True or false is the installation is found or not and if it is found, the path of the found installation

Return type (boolean, str)

`src.product.`**`check_installation`**(*product_info*)

Verify if a product is well installed. Checks install directory presence and some additional files if it is defined in the config

Parameters **`Config`** (*`product_info`*) – The configuration specific to the product

Returns True if it is well installed

Return type boolean

`src.product.`**`check_source`**(*product_info*)

Verify if a sources of product is preset. Checks source directory presence

Parameters **`Config`** (*`product_info`*) – The configuration specific to the product

Returns True if it is well installed

Return type boolean

`src.product.`**`get_base_install_dir`**(*config*, *prod_info*, *version*)

Compute the installation directory of a product in base

Parameters

- **`Config`** (*`product_info`*) – The global configuration
- **`Config`** – The configuration specific to the product
- **`str`** (*`version`*) – The version of the product

Returns The path of the product installation

Return type str

`src.product.`**`get_install_dir`**(*config*, *base*, *version*, *prod_info*)

Compute the installation directory of a given product

Parameters

- **`Config`** (*`product_info`*) – The global configuration
- **`str`** (*`version`*) – This corresponds to the value given by user in its application.pyconf for the specific product. If “yes”, the user wants the product to be in base. If “no”, he wants the product to be in the application workdir
- **`str`** – The version of the product
- **`Config`** – The configuration specific to the product

Returns The path of the product installation

Return type str

`src.product.`**`get_product_components`**(*product_info*)

Get the component list to generate with the product

Parameters **`Config`** (*`product_info`*) – The configuration specific to the product

Returns The list of names of the components

Return type List

`src.product.`**`get_product_config`**(*config*, *product_name*, *with_install_dir=True*)
Get the specific configuration of a product from the global configuration

Parameters

- **`Config`** (*`config`*) – The global configuration
- **`str`** (*`product_name`*) – The name of the product
- **`boolean`** (*`with_install_dir`*) – If false, do not provide an install directory (at false only for internal use of the function check_config_exists)

Returns the specific configuration of the product

Return type *Config* (page 56)

`src.product.`**`get_product_dependencies`**(*config*, *product_info*)
Get recursively the list of products that are in the product_info dependencies

Parameters

- **`Config`** (*`product_info`*) – The global configuration
- **`Config`** – The configuration specific to the product

Returns the list of products in dependence

Return type list

`src.product.`**`get_product_section`**(*config*, *product_name*, *version*, *section=None*)
Get the product description from the configuration

Parameters

- **`Config`** (*`config`*) – The global configuration
- **`str`** (*`section`*) – The product name
- **`str`** – The version of the product
- **`str`** – The searched section (if not None, the section is explicitly given

Returns The product description

Return type *Config* (page 56)

`src.product.`**`get_products_infos`**(*lproducts*, *config*)
Get the specific configuration of a list of products

Parameters

- **`List`** (*`lproducts`*) – The list of product names
- **`Config`** (*`config`*) – The global configuration

Returns the list of tuples (product name, specific configuration of the product)

Return type [(str, *Config* (page 56))]

`src.product.`**`product_compiles`**(*product_info*)
Know if a product compiles or not (some products do not have a compilation procedure)

Parameters **`Config`** (*`product_info`*) – The configuration specific to the product

Returns True if the product compiles, else False

Return type boolean

`src.product.`**`product_has_env_script`**(*product_info*)
Know if a product has an environment script

Parameters **`Config`** (*`product_info`*) – The configuration specific to the product

Returns True if the product it has an environment script, else False

Return type boolean

`src.product.`**`product_has_logo`**(*product_info*)
Know if a product has a logo (YACSGEN generate)

Parameters **`Config`** (*`product_info`*) – The configuration specific to the product

Returns The path of the logo if the product has a logo, else False

Return type Str

`src.product.`**`product_has_patches`**(*product_info*)
Know if a product has one or more patches

Parameters **`Config`** (*`product_info`*) – The configuration specific to the product

Returns True if the product has one or more patches

Return type boolean

`src.product.`**`product_has_salome_gui`**(*product_info*)
Know if a product has a SALOME gui

Parameters **`Config`** (*`product_info`*) – The configuration specific to the product

Returns True if the product has a SALOME gui, else False

Return type Boolean

`src.product.`**`product_has_script`**(*product_info*)
Know if a product has a compilation script

Parameters **`Config`** (*`product_info`*) – The configuration specific to the product

Returns True if the product it has a compilation script, else False

Return type boolean

`src.product.`**`product_is_autotools`**(*product_info*)
Know if a product is compiled using the autotools

Parameters **`Config`** (*`product_info`*) – The configuration specific to the product

Returns True if the product is autotools, else False

Return type boolean

`src.product.`**`product_is_cmake`**(*product_info*)
Know if a product is compiled using the cmake

Parameters **`Config`** (*`product_info`*) – The configuration specific to the product

Returns True if the product is cmake, else False

Return type boolean

`src.product.`**`product_is_cpp`**(*product_info*)
Know if a product is cpp

Parameters **`Config`** (*`product_info`*) – The configuration specific to the product

Returns True if the product is a cpp, else False

Return type boolean

`src.product.`**`product_is_debug`**(*product_info*)
Know if a product is in debug mode

Parameters **`Config`** (*`product_info`*) – The configuration specific to the product

Returns True if the product is in debug mode, else False

Return type boolean

src.product.**product_is_dev**(*product_info*)
Know if a product is in dev mode

Parameters **Config** (*product_info*) – The configuration specific to the product

Returns True if the product is in dev mode, else False

Return type boolean

src.product.**product_is_fixed**(*product_info*)
Know if a product is fixed

Parameters **Config** (*product_info*) – The configuration specific to the product

Returns True if the product is fixed, else False

Return type boolean

src.product.**product_is_generated**(*product_info*)
Know if a product is generated (YACSGEN)

Parameters **Config** (*product_info*) – The configuration specific to the product

Returns True if the product is generated

Return type boolean

src.product.**product_is_mpi**(*product_info*)
Know if a product has openmpi in its dependencies

Parameters **Config** (*product_info*) – The configuration specific to the product

Returns True if the product has openmpi inits dependencies

Return type boolean

src.product.**product_is_native**(*product_info*)
Know if a product is native

Parameters **Config** (*product_info*) – The configuration specific to the product

Returns True if the product is native, else False

Return type boolean

src.product.**product_is_salome**(*product_info*)
Know if a product is a SALOME module

Parameters **Config** (*product_info*) – The configuration specific to the product

Returns True if the product is a SALOME module, else False

Return type boolean

src.product.**product_is_smesh_plugin**(*product_info*)
Know if a product is a SMESH plugin

Parameters **Config** (*product_info*) – The configuration specific to the product

Returns True if the product is a SMESH plugin, else False

Return type boolean

src.product.**product_is_vcs**(*product_info*)
Know if a product is download using git, svn or cvs (not archive)

Parameters **Config** (*product_info*) – The configuration specific to the product

Returns True if the product is vcs, else False

Return type boolean

src.product.**product_is_verbose**(*product_info*)
Know if a product is in verbose mode

Parameters **Config** (*product_info*) – The configuration specific to the product

Returns True if the product is in verbose mode, else False

Return type boolean

src.pyconf module

This is a configuration module for Python.

This module should work under Python versions >= 2.2, and cannot be used with earlier versions since it uses new-style classes.

Development and testing has only been carried out (so far) on Python 2.3.4 and Python 2.4.2. See the test module (test_config.py) included in the U{distribution<http://www.red-dove.com/python_config.html|_blank>} (follow the download link).

A simple example - with the example configuration file:

```
messages:
[
  {
    stream : `sys.stderr`
    message: 'Welcome'
    name: 'Harry'
  }
  {
    stream : `sys.stdout`
    message: 'Welkom'
    name: 'Ruud'
  }
  {
    stream : $messages[0].stream
    message: 'Bienvenue'
    name: Yves
  }
]
```

a program to read the configuration would be:

```
from config import Config

f = file('simple.cfg')
cfg = Config(f)
for m in cfg.messages:
    s = '%s, %s' % (m.message, m.name)
    try:
        print >> m.stream, s
    except IOError, e:
        print e
```

which, when run, would yield the console output:

```
Welcome, Harry
Welkom, Ruud
Bienvenue, Yves
```

See U{this tutorial<http://www.red-dove.com/python_config.html|_blank>} for more information.

#modified for salomeTools @version: 0.3.7.1

@author: Vinay Sajip

@copyright: Copyright (C) 2004-2007 Vinay Sajip. All Rights Reserved.

@var streamOpener: The default stream opener. This is a factory function which takes a string (e.g. filename) and returns a stream suitable for reading. If unable to open the stream, an IOError exception should be thrown.

The default value of this variable is L{defaultStreamOpener}. For an example of how it's used, see test_config.py (search for streamOpener).

class `src.pyconf.`**`Config`**(*streamOrFile=None*, *parent=None*, *PWD=None*)

Bases: *`src.pyconf.Mapping`* (page 60)

This class represents a configuration, and is the only one which clients need to interface to, under normal circumstances.

class Namespace

Bases: `object`

This internal class is used for implementing default namespaces.

An instance acts as a namespace.

`addNamespace`(*ns*, *name=None*)

Add a namespace to this configuration which can be used to evaluate (resolve) dotted-identifier expressions. @param ns: The namespace to be added. @type ns: A module or other namespace suitable for passing as an argument to vars(). @param name: A name for the namespace, which, if specified, provides an additional level of indirection. @type name: str

`getByPath`(*path*)

Obtain a value in the configuration via its path. @param path: The path of the required value @type path: str @return the value at the specified path. @rtype: any @raise ConfigError: If the path is invalid

`load`(*stream*)

Load the configuration from the specified stream. Multiple streams can be used to populate the same instance, as long as there are no clashing keys. The stream is closed. @param stream: A stream from which the configuration is read. @type stream: A read-only stream (file-like object). @raise ConfigError: if keys in the loaded configuration clash with existing keys. @raise ConfigFormatError: if there is a syntax error in the stream.

`removeNamespace`(*ns*, *name=None*)

Remove a namespace added with L{addNamespace}. @param ns: The namespace to be removed. @param name: The name which was specified when L{addNamespace} was called. @type name: str

exception `src.pyconf.`**`ConfigError`**

Bases: `exceptions.Exception`

This is the base class of exceptions raised by this module.

exception `src.pyconf.`**`ConfigFormatError`**

Bases: *`src.pyconf.ConfigError`* (page 56)

This is the base class of exceptions raised due to syntax errors in configurations.

class `src.pyconf.`**`ConfigInputStream`**(*stream*)

Bases: `object`

An input stream which can read either ANSI files with default encoding or Unicode files with BOMs.

Handles UTF-8, UTF-16LE, UTF-16BE. Could handle UTF-32 if Python had built-in support.

`close`()

`read`(*size*)

`readline`()

class `src.pyconf.`**`ConfigList`**

Bases: `list`

This class implements an ordered list of configurations and allows you to try getting the configuration from each entry in turn, returning the first successfully obtained value.

getByPath (*path*)
Obtain a value from the first configuration in the list which defines it.

@param path: The path of the value to retrieve. @type path: str @return: The value from the earliest configuration in the list which defines it. @rtype: any @raise ConfigError: If no configuration in the list has an entry with the specified path.

class `src.pyconf.`**ConfigMerger** (*resolver=<function defaultMergeResolve>*)
Bases: `object`

This class is used for merging two configurations. If a key exists in the merge operand but not the merge target, then the entry is copied from the merge operand to the merge target. If a key exists in both configurations, then a resolver (a callable) is called to decide how to handle the conflict.

handleMismatch (*obj1*, *obj2*)
Handle a mismatch between two objects.

@param obj1: The object to merge into. @type obj1: any @param obj2: The object to merge. @type obj2: any

merge (*merged*, *mergee*)
Merge two configurations. The second configuration is unchanged, and the first is changed to reflect the results of the merge.

@param merged: The configuration to merge into. @type merged: L{Config}. @param mergee: The configuration to merge. @type mergee: L{Config}.

mergeMapping (*map1*, *map2*)
Merge two mappings recursively. The second mapping is unchanged, and the first is changed to reflect the results of the merge.

@param map1: The mapping to merge into. @type map1: L{Mapping}. @param map2: The mapping to merge. @type map2: L{Mapping}.

mergeSequence (*seq1*, *seq2*)
Merge two sequences. The second sequence is unchanged, and the first is changed to have the elements of the second appended to it.

@param seq1: The sequence to merge into. @type seq1: L{Sequence}. @param seq2: The sequence to merge. @type seq2: L{Sequence}.

overwriteKeys (*map1*, *seq2*)
Renint variables. The second mapping is unchanged, and the first is changed depending the keys of the second mapping. @param map1: The mapping to reinit keys into. @type map1: L{Mapping}. @param map2: The mapping container reinit information. @type map2: L{Mapping}.

class `src.pyconf.`**ConfigOutputStream** (*stream*, *encoding=None*)
Bases: `object`

An output stream which can write either ANSI files with default encoding or Unicode files with BOMs.

Handles UTF-8, UTF-16LE, UTF-16BE. Could handle UTF-32 if Python had built-in support.

close ()

flush ()

write (*data*)

class `src.pyconf.`**ConfigReader** (*config*)
Bases: `object`

This internal class implements a parser for configurations.

getChar ()
Get the next char from the stream. Update line and column numbers appropriately.

@return: The next character from the stream. @rtype: str

getToken()
Get a token from the stream. String values are returned in a form where you need to eval() the returned value to get the actual string. The return value is (token_type, token_value).

Multiline string tokenizing is thanks to David Janes (BlogMatrix)

@return: The next token. @rtype: A token tuple.

load(*stream*, *parent=None*, *suffix=None*)
Load the configuration from the specified stream.

@param stream: A stream from which to load the configuration. @type stream: A stream (file-like object). @param parent: The parent of the configuration (to which this reader belongs) in the hierarchy. Specified when the configuration is included in another one. @type parent: A L{Container} instance. @param suffix: The suffix of this configuration in the parent configuration. Should be specified whenever the parent is not None. @raise ConfigError: If parent is specified but suffix is not. @raise ConfigFormatError: If there are syntax errors in the stream.

location()
Return the current location (filename, line, column) in the stream as a string.

Used when printing error messages,

@return: A string representing a location in the stream being read. @rtype: str

match(*t*)
Ensure that the current token type matches the specified value, and advance to the next token.

@param t: The token type to match. @type t: A valid token type. @return: The token which was last read from the stream before this function is called. @rtype: a token tuple - see L{getToken}. @raise ConfigFormatError: If the token does not match what's expected.

parseFactor()
Parse a factor in an multiplicative expression (a * b, a / b, a % b)

@return: the parsed factor @rtype: any scalar @raise ConfigFormatError: if a syntax error is found.

parseKeyValuePair(*parent*)
Parse a key-value pair, and add it to the provided L{Mapping}.

@param parent: The mapping to add entries to. @type parent: A L{Mapping} instance. @raise ConfigFormatError: if a syntax error is found.

parseMapping(*parent*, *suffix*)
Parse a mapping.

@param parent: The container to which the mapping will be added. @type parent: A L{Container} instance. @param suffix: The suffix for the value. @type suffix: str @return: a L{Mapping} instance representing the mapping. @rtype: L{Mapping} @raise ConfigFormatError: if a syntax error is found.

parseMappingBody(*parent*)
Parse the internals of a mapping, and add entries to the provided L{Mapping}.

@param parent: The mapping to add entries to. @type parent: A L{Mapping} instance.

parseReference(*type*)
Parse a reference.

@return: the parsed reference @rtype: L{Reference} @raise ConfigFormatError: if a syntax error is found.

parseScalar()
Parse a scalar - a terminal value such as a string or number, or an L{Expression} or L{Reference}.

@return: the parsed scalar @rtype: any scalar @raise ConfigFormatError: if a syntax error is found.

parseSequence(*parent*, *suffix*)
Parse a sequence.

@param parent: The container to which the sequence will be added. @type parent: A L{Container} instance. @param suffix: The suffix for the value. @type suffix: str @return: a L{Sequence} instance representing the sequence. @rtype: L{Sequence} @raise ConfigFormatError: if a syntax error is found.

parseSuffix(*ref*)
Parse a reference suffix.

@param ref: The reference of which this suffix is a part. @type ref: L{Reference}. @raise ConfigFormatError: if a syntax error is found.

parseTerm()
Parse a term in an additive expression (a + b, a - b)

@return: the parsed term @rtype: any scalar @raise ConfigFormatError: if a syntax error is found.

parseValue(*parent*, *suffix*)
Parse a value.

@param parent: The container to which the value will be added. @type parent: A L{Container} instance. @param suffix: The suffix for the value. @type suffix: str @return: The value @rtype: any @raise ConfigFormatError: if a syntax error is found.

setStream(*stream*)
Set the stream to the specified value, and prepare to read from it.

@param stream: A stream from which to load the configuration. @type stream: A stream (file-like object).

exception src.pyconf.**ConfigResolutionError**
Bases: *src.pyconf.ConfigError* (page 56)

This is the base class of exceptions raised due to semantic errors in configurations.

class src.pyconf.**Container**(*parent*)
Bases: object

This internal class is the base class for mappings and sequences.

@ivar path: A string which describes how to get to this instance from the root of the hierarchy.

Example:

```
a.list.of[1].or['more'].elements
```

evaluate(*item*)
Evaluate items which are instances of L{Reference} or L{Expression}.

L{Reference} instances are evaluated using L{Reference.resolve}, and L{Expression} instances are evaluated using L{Expression.evaluate}.

@param item: The item to be evaluated. @type item: any @return: If the item is an instance of L{Reference} or L{Expression}, the evaluated value is returned, otherwise the item is returned unchanged.

setPath(*path*)
Set the path for this instance. @param path: The path - a string which describes how to get to this instance from the root of the hierarchy. @type path: str

writeToStream(*stream*, *indent*, *container*)
Write this instance to a stream at the specified indentation level.

Should be redefined in subclasses.

@param stream: The stream to write to @type stream: A writable stream (file-like object) @param indent: The indentation level @type indent: int @param container: The container of this instance @type container: L{Container} @raise NotImplementedError: If a subclass does not override this

writeValue(*value*, *stream*, *indent*)

class src.pyconf.**Expression**(*op*, *lhs*, *rhs*)
Bases: `object`

This internal class implements a value which is obtained by evaluating an expression.

evaluate(*container*)
Evaluate this instance in the context of a container.

@param container: The container to evaluate in from. @type container: L{Container} @return: The evaluated value. @rtype: any @raise ConfigResolutionError: If evaluation fails. @raise ZeroDivideError: If division by zero occurs. @raise TypeError: If the operation is invalid, e.g. subtracting one string from another.

class src.pyconf.**Mapping**(*parent=None*)
Bases: *src.pyconf.Container* (page 59)

This internal class implements key-value mappings in configurations.

addMapping(*key*, *value*, *comment*, *setting=False*)
Add a key-value mapping with a comment.

@param key: The key for the mapping. @type key: str @param value: The value for the mapping. @type value: any @param comment: The comment for the key (can be None). @type comment: str @param setting: If True, ignore clashes. This is set to true when called from L{__setattr__}. @raise ConfigFormatError: If an existing key is seen again and setting is False.

get(*key*, *default=None*)
Allows a dictionary-style get operation.

iteritems()

iterkeys()

keys()
Return the keys in a similar way to a dictionary.

writeToStream(*stream*, *indent*, *container*)
Write this instance to a stream at the specified indentation level.

Should be redefined in subclasses.

@param stream: The stream to write to @type stream: A writable stream (file-like object) @param indent: The indentation level @type indent: int @param container: The container of this instance @type container: L{Container}

class src.pyconf.**Reference**(*config*, *type*, *ident*)
Bases: `object`

This internal class implements a value which is a reference to another value.

addElement(*type*, *ident*)
Add an element to the reference.

@param type: The type of reference. @type type: BACKTICK or DOLLAR @param ident: The identifier which continues the reference. @type ident: str

findConfig(*container*)
Find the closest enclosing configuration to the specified container.

@param container: The container to start from. @type container: L{Container} @return: The closest enclosing configuration, or None. @rtype: L{Config}

resolve(*container*)
Resolve this instance in the context of a container.

@param container: The container to resolve from. @type container: L{Container} @return: The resolved value. @rtype: any @raise ConfigResolutionError: If resolution fails.

class src.pyconf.**Sequence**(*parent=None*)
Bases: *src.pyconf.Container* (page 59)

This internal class implements a value which is a sequence of other values.

class SeqIter(*seq*)
Bases: object

This internal class implements an iterator for a L{Sequence} instance.

next()

append(*item*, *comment*)
Add an item to the sequence.

@param item: The item to add. @type item: any @param comment: A comment for the item. @type comment: str

writeToStream(*stream*, *indent*, *container*)
Write this instance to a stream at the specified indentation level.

Should be redefined in subclasses.

@param stream: The stream to write to @type stream: A writable stream (file-like object) @param indent: The indentation level @type indent: int @param container: The container of this instance @type container: L{Container}

src.pyconf.**deepCopyMapping**(*inMapping*)

src.pyconf.**defaultMergeResolve**(*map1*, *map2*, *key*)
A default resolver for merge conflicts. Returns a string indicating what action to take to resolve the conflict.

@param map1: The map being merged into. @type map1: L{Mapping}. @param map2: The map being used as the merge operand. @type map2: L{Mapping}. @param key: The key in map2 (which also exists in map1). @type key: str

@return: One of "merge", "append", "mismatch" or "overwrite" indicating what action should be taken. This should be appropriate to the objects being merged - e.g. there is no point returning "merge" if the two objects are instances of L{Sequence}.

@rtype: str

src.pyconf.**defaultStreamOpener**(*name*)
This function returns a read-only stream, given its name. The name passed in should correspond to an existing stream, otherwise an exception will be raised.

This is the default value of L{streamOpener}; assign your own callable to streamOpener to return streams based on names. For example, you could use urllib2.urlopen().

@param name: The name of a stream, most commonly a file name. @type name: str @return: A stream with the specified name. @rtype: A read-only stream (file-like object)

src.pyconf.**isWord**(*s*)
See if a passed-in value is an identifier. If the value passed in is not a string, False is returned. An identifier consists of alphanumerics or underscore characters.

Examples:

```
isWord('a word') ->False
isWord('award') -> True
isWord(9) -> False
isWord('a_b_c_') ->True
```

@note: isWord('9abc') will return True - not exactly correct, but adequate for the way it's used here.

@param s: The name to be tested @type s: any @return: True if a word, else False @rtype: bool

src.pyconf.**makePath**(*prefix*, *suffix*)
Make a path from a prefix and suffix.

Examples: makePath(‘’, ‘suffix’) -> ‘suffix’ makePath(‘prefix’, ‘suffix’) -> ‘prefix.suffix’ makePath(‘prefix’, ‘[1]’) -> ‘prefix[1]’

@param prefix: The prefix to use. If it evaluates as false, the suffix is returned. @type prefix: str @param suffix: The suffix to use. It is either an identifier or an index in brackets. @type suffix: str @return: The path concatenation of prefix and suffix, with adot if the suffix is not a bracketed index. @rtype: str

`src.pyconf.`**`overwriteMergeResolve`**(*map1*, *map2*, *key*)

An overwriting resolver for merge conflicts. Calls L{defaultMergeResolve}, but where a “mismatch” is detected, returns “overwrite” instead.

@param map1: The map being merged into. @type map1: L{Mapping}. @param map2: The map being used as the merge operand. @type map2: L{Mapping}. @param key: The key in map2 (which also exists in map1). @type key: str

src.system module

In this file : all functions that do a system call, like open a browser or an editor, or call a git command

`src.system.`**`archive_extract`**(*from_what*, *where*, *logger*)

Extracts sources from an archive.

Parameters

- **`str`** (*`where`*) – The path to the archive.
- **`str`** – The path where to extract.
- **`Logger`** (*`logger`*) – The logger instance to use.

Returns True if the extraction is successful

Return type boolean

`src.system.`**`cvs_extract`**(*protocol*, *user*, *server*, *base*, *tag*, *product*, *where*, *logger*, *checkout=False*, *environment=None*)

Extracts sources from a cvs repository.

Parameters

- **`str`** (*`where`*) – The cvs protocol.
- **`str`** – The user to be used.
- **`str`** – The remote cvs server.
- **`str`** – .
- **`str`** – The tag.
- **`str`** – The product.
- **`str`** – The path where to extract.
- **`Logger`** (*`logger`*) – The logger instance to use.
- **`boolean`** (*`checkout`*) – If true use checkout cvs.
- **`src.environment.Environ`** (*`environment`*) – The environment to source when extracting.

Returns True if the extraction is successful

Return type boolean

`src.system.`**`git_extract`**(*from_what*, *tag*, *where*, *logger*, *environment=None*)

Extracts sources from a git repository.

Parameters

- **`str`** (*`where`*) – The remote git repository.

- **str** – The tag.
- **str** – The path where to extract.
- **Logger** (*logger*) – The logger instance to use.
- **src.environment.Environ** (*environment*) – The environment to source when extracting.

Returns True if the extraction is successful

Return type boolean

`src.system.`**`show_in_editor`**(*editor*, *filePath*, *logger*)
open filePath using editor.

Parameters

- **str** (*filePath*) – The editor to use.
- **str** – The path to the file to open.

`src.system.`**`svn_extract`**(*user*, *from_what*, *tag*, *where*, *logger*, *checkout=False*, *environment=None*)
Extracts sources from a svn repository.

Parameters

- **str** (*where*) – The user to be used.
- **str** – The remote git repository.
- **str** – The tag.
- **str** – The path where to extract.
- **Logger** (*logger*) – The logger instance to use.
- **boolean** (*checkout*) – If true use checkout svn.
- **src.environment.Environ** (*environment*) – The environment to source when extracting.

Returns True if the extraction is successful

Return type boolean

src.template module

class `src.template.`**`MyTemplate`**(*template*)
Bases: `string.Template`

delimiter = '\xc2\xa4'

pattern = <_sre.SRE_Pattern object>

`src.template.`**`substitute`**(*template_file*, *subst_dic*)

src.test_module module

class `src.test_module.`**`Test`**(*config*, *logger*, *tmp_working_dir*, *testbase=”*, *grids=None*, *sessions=None*, *launcher=”*, *show_desktop=True*)

generate_launching_commands()

generate_script(*listTest*, *script_path*, *ignoreList*)

get_test_timeout(*test_name*, *default_value*)

get_tmp_dir()

prepare_testbase (*test_base_name*)

prepare_testbase_from_dir (*testbase_name*, *testbase_dir*)

prepare_testbase_from_git (*testbase_name*, *testbase_base*, *testbase_tag*)

prepare_testbase_from_svn (*user*, *testbase_name*, *testbase_base*)

read_results (*listTest*, *has_timed_out*)

run_all_tests ()

run_grid_tests ()

run_script (*script_name*)

run_session_tests ()

run_testbase_tests ()

run_tests (*listTest*, *ignoreList*)

search_known_errors (*status*, *test_grid*, *test_session*, *test*)

write_test_margin (*tab*)

`src.test_module.`**getTmpDirDEFAULT** ()

src.xmlManager module

class `src.xmlManager.`**ReadXmlFile** (*filePath*)

Bases: `object`

Class to manage reading of an xml log file

getRootAttrib ()

Get the attibutes of the self.xmlroot

Returns The attributes of the root node

Return type dict

get_attrib (*node_name*)

Get the attibutes of the node node_name in self.xmlroot

Parameters **str** (`node_name`) – the name of the node

Returns the attibutes of the node node_name in self.xmlroot

Return type dict

get_node_text (*node*)

Get the text of the first node that has name that corresponds to the parameter node

Parameters **str** (`node`) – the name of the node from which get the text

Returns the text of the first node that has name that corresponds to the parameter node

Return type str

class `src.xmlManager.`**XmlLogFile** (*filePath*, *rootname*, *attrib={}*)

Bases: `object`

Class to manage writing in salomeTools xml log file

add_simple_node (*node_name*, *text=None*, *attrib={}*)

Add a node with some attibutes and text to the root node.

Parameters

- **str** (`text`) – the name of the node to add

- **str** – the text of the node
- **dict** (*attrib*) – the dictionary containing the attribute of the new node

append_node_attrib (*node_name*, *attrib*)
Append a new attributes to the node that has node_name as name

Parameters

- **str** (*node_name*) – The name of the node on which append text
- **dixt** (*attrib*) – The attrib to append

append_node_text (*node_name*, *text*)
Append a new text to the node that has node_name as name

Parameters

- **str** (*text*) – The name of the node on which append text
- **str** – The text to append

write_tree (*stylesheet=None*, *file_path=None*)
Write the xml tree in the log file path. Add the stylesheet if asked.

Parameters **str** (*stylesheet*) – The stylesheet to apply to the xml file

src.xmlManager.**add_simple_node** (*root_node*, *node_name*, *text=None*, *attrib={}*)
Add a node with some attibutes and text to the root node.

Parameters

- **etree.Element** (*root_node*) – the Etree element where to add the new node
- **str** (*text*) – the name of the node to add
- **str** – the text of the node
- **dict** (*attrib*) – the dictionary containing the attribute of the new node

src.xmlManager.**append_node_attrib** (*root_node*, *attrib*)
Append a new attributes to the node that has node_name as name

Parameters

- **etree.Element** (*root_node*) – the Etree element where to append the new attibutes
- **dixt** (*attrib*) – The attrib to append

src.xmlManager.**find_node_by_attrib** (*xmlroot*, *name_node*, *key*, *value*)

Find the nfirst ode from xmlroot that has name name_node and that has in its attributes {key : value}. Return the node

Parameters

- **etree.Element** (*xmlroot*) – the Etree element where to search
- **str** (*value*) – the name of node to search
- **str** – the key to search
- **str** – the value to search

Returns the found node

Return type xmlroot etree.Element

src.xmlManager.**write_report** (*filename*, *xmlroot*, *stylesheet*)
Writes a report file from a XML tree.

Parameters

- **str** (*stylesheet*) – The path to the file to create
- **etree.Element** (*xmlroot*) – the Etree element to write to the file
- **str** – The stylesheet to add to the begin of the file

Module contents

initial imports and utilities methods for salomeTools

class src.**Path**(*path*)

> **base**()
>
> **chmod**(*mode*)
>
> **copy**(*path*, *smart=False*)
>
> **copydir**(*dst*, *smart=False*)
>
> **copyfile**(*path*)
>
> **copylink**(*path*)
>
> **dir**()
>
> **exists**()
>
> **isdir**()
>
> **isfile**()
>
> **islink**()
>
> **list**()
>
> **make**(*mode=None*)
>
> **readlink**()
>
> **rm**()
>
> **smartcopy**(*path*)
>
> **symlink**(*path*)

exception src.**SatException**

> Bases: `exceptions.Exception`
>
> rename Exception Class

src.**activate_mesa_property**(*config*)

> Add mesa property into application properties
>
> **Parameters** **Config** (*config*) – The global configuration. It must have an application!

src.**check_config_has_application**(*config*, *details=None*)

> check that the config has the key APPLICATION. Else raise an exception.
>
> **Parameters** **class 'common.pyconf.Config'** (*config*) – The config.

src.**check_config_has_profile**(*config*, *details=None*)

> check that the config has the key APPLICATION.profile. else, raise an exception.
>
> **Parameters** **class 'common.pyconf.Config'** (*config*) – The config.

src.**config_has_application**(*config*)

src.**deepcopy_list**(*input_list*)

> Do a deep copy of a list
>
> **Parameters** **List** (*input_list*) – The list to copy

Returns The copy of the list

Return type List

`src.`**`ensure_path_exists`**`(`*p*`)`
Create a path if not existing

Parameters **`str`** (*`p`*) – The path.

`src.`**`find_file_in_lpath`**`(`*file_name*, *lpath*, *additional_dir=''*`)`
Find in all the directories in lpath list the file that has the same name as file_name. If it is found then return the full path of the file else return False.

The additional_dir (optional) is the name of the directory to add to all paths in lpath.

Parameters

- **`str`** (*`additional_dir`*) – The file name to search
- **`List`** (*`lpath`*) – The list of directories where to search
- **`str`** – The name of the additional directory

Returns the full path of the file or False if not found

Return type str

`src.`**`get_base_path`**`(`*config*`)`
Returns the path of the products base.

Parameters **`Config`** (*`config`*) – The global Config instance.

Returns The path of the products base.

Return type str

`src.`**`get_cfg_param`**`(`*config*, *param_name*, *default*`)`
eearch for param_name value in config. if param_name is not in config then return default, else return the found value

Parameters

- **`class 'common.pyconf.Config'`** (*`config`*) – The config.
- **`str`** (*`default`*) – the name of the parameter to get the value
- **`str`** – The value to return if param_name is not in config

Returns see initial description of the function

Return type str

`src.`**`get_launcher_name`**`(`*config*`)`
Returns the name of salome launcher.

Parameters **`Config`** (*`config`*) – The global Config instance.

Returns The name of salome launcher.

Return type str

`src.`**`get_log_path`**`(`*config*`)`
Returns the path of the logs.

Parameters **`Config`** (*`config`*) – The global Config instance.

Returns The path of the logs.

Return type str

`src.`**`get_property_in_product_cfg`**`(`*product_cfg*, *pprty*`)`

`src.`**`get_salome_version`**`(`*config*`)`

`src.`**`get_tmp_filename`**`(`*cfg*, *name*`)` `src.`**`handleRemoveReadonly`**(*func*, *path*, *exc*)

`src.`**`merge_dicts`**(**dict_args*)
Given any number of dicts, shallow copy and merge into a new dict, precedence goes to key value pairs in latter dicts.

`src.`**`only_numbers`**(*str_num*)

`src.`**`parse_date`**(*date*)
Transform YYYYMMDD_hhmmss into YYYY-MM-DD hh:mm:ss.

Parameters **`str`** (*`date`*) – The date to transform

Returns The date in the new format

Return type str

`src.`**`print_info`**(*logger*, *info*)
Prints the tuples that are in info variable in a formatted way.

Parameters

- **`Logger`** (*`logger`*) – The logging instance to use for the prints.
- **`list`** (*`info`*) – The list of tuples to display

`src.`**`read_config_from_a_file`**(*filePath*)

`src.`**`remove_item_from_list`**(*input_list*, *item*)
Remove all occurences of item from input_list

Parameters **`List`** (*`input_list`*) – The list to modify

Returns The without any item

Return type List

`src.`**`replace_in_file`**(*filein*, *strin*, *strout*)
Replace <strin> by <strout> in file <filein>

CHAPTER

FIVE

RELEASE NOTES

5.1 Release notes

In construction.

PYTHON MODULE INDEX

INDEX