salomeTools Documentation

Release 5.0.0dev

CEA DEN/DANS/DM2S/STMF/LGLS

May 04, 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 config

2.1.1 Description

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

2.1.2 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
```

2.1.3 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
```

2.2 Command prepare

2.2.1 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.

2.2.2 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)

2.2.3 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
  ```

2.2.4 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.

2.3 Command compile

2.3.1 Description

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

2.3.2 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
```

2.3.3 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.

2.4 Command launcher

2.4.1 Description

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

2.4.2 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
  ```

2.4.3 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

2.5 Command application

2.5.1 Description

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

2.5.2 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
  ```

2.5.3 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

2.6 Command log

2.6.1 Description

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

2.6.2 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
  ```

2.6.3 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

2.7 Command environ

2.7.1 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.

2.7.2 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>, ...
```

2.7.3 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.

2.8 Command clean

2.8.1 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

2.8.2 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
```

2.8.3 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).

2.8.4 Some useful configuration pathes

No specific configuration.

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

2.9 Command package

2.9.1 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.

2.9.2 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].

2.9.3 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

2.10 Command generate

2.10.1 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`.

2.10.2 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)

2.10.3 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

Warning: ALL THIS IS OBSOLETE FOR SAT 5.1

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.")
```

(continues on next page)

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

(continued from previous page)

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

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.getConfig()* 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 python `logging` package class. It gives access to `debug, info, warning, error, critical` methods.

Using these methods, the message passed as parameter will be displayed in the terminal and written in an xml log file.

```
logger.info("My message")
```

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.info('HELLO! WORLD!\n')
    else:
        logger.writeinfo('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!

# 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 at 0x3504970>`

`ANSI_OSC_RE = <_sre.SRE_Pattern object at 0x350dc10>`

`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

src.example package

Submodules

src.example.essai_logging_1 module

http://sametmax.com/ecrire-des-logs-en-python/ https://docs.python.org/3/library/time.html#time.strftime

essai utilisation logger plusieurs handler format different

> /usr/lib/python2.7/logging/__init__.pyc
>
> init MyLogger, fmt='%(asctime)s :: %(levelname)-8s :: %(message)s', level='20'
>
> 2018-03-11 18:51:21 :: INFO :: test logger info 2018-03-11 18:51:21 :: WARNING :: test logger warning 2018-03-11 18:51:21 :: ERROR :: test logger error 2018-03-11 18:51:21 :: CRITICAL :: test logger critical
>
> init MyLogger, fmt='None', level='10'
>
> 2018-03-11 18:51:21 :: DEBUG :: test logger debug test logger debug 2018-03-11 18:51:21 :: INFO :: test logger info test logger info 2018-03-11 18:51:21 :: WARNING :: test logger warning test logger warning 2018-03-11 18:51:21 :: ERROR :: test logger error test logger error 2018-03-11 18:51:21 :: CRITICAL :: test logger critical test logger critical

`src.example.essai_logging_1.`**`getMyLogger`**`()`

`src.example.essai_logging_1.`**`initMyLogger`**`(`*fmt=None*, *level=None*`)`

`src.example.essai_logging_1.`**`testLogger1`**`()`

src.example.essai_logging_2 module

http://sametmax.com/ecrire-des-logs-en-python/ https://docs.python.org/3/library/time.html#time.strftime

essai utilisation logger un handler format different sur info() pas de format et su other format

> /usr/lib/python2.7/logging/__init__.pyc
>
> init MyLogger, fmt='%(asctime)s :: %(levelname)-8s :: %(message)s', level='20'
>
> test logger info 2018-03-11 18:51:51 :: WARNING :: test logger warning 2018-03-11 18:51:51 :: ERROR :: test logger error 2018-03-11 18:51:51 :: CRITICAL :: test logger critical

`class` `src.example.essai_logging_2.`**`MyFormatter`**`(`*fmt=None*, *datefmt=None*`)`
Bases: `logging.Formatter`

> **`format`**`(`*record*`)`
> Format the specified record as text.
>
> The record's attribute dictionary is used as the operand to a string formatting operation which yields the returned string. Before formatting the dictionary, a couple of preparatory steps are carried out. The message attribute of the record is computed using LogRecord.getMessage(). If the formatting string uses the time (as determined by a call to usesTime(), formatTime() is called to format the event time. If there is exception information, it is formatted using formatException() and appended to the message.

`src.example.essai_logging_2.`**`getMyLogger`**`()`

`src.example.essai_logging_2.`**`initMyLogger`**`(`*fmt=None*, *level=None*`)`

`src.example.essai_logging_2.`**`testLogger1`**`()`

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

Contains 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

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

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

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

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

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

`src.architecture.`**`get_nb_proc`**()
Gets the number of processors of the machine on which salomeTools is running.

Returns (str) The number of processors.

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

Returns (str) The version of the running python.

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

Returns (str) environ var USERNAME

`src.architecture.`**`is_windows`**()
Checks windows OS

Returns (bool) True if system is Windows

src.catchAll module

define class as a simple dictionary with keys with pretty print __str__ and __repr__ (indented as recursive) and jsonDumps()

Usage:
>> import catchAll as CAA
>> a = CAA.CatchAll()
>> a.tintin = "reporter"
>> a.milou = "dog"
>> print("a=%s" % a)
>> print("tintin: %s" % a.tintin)

`class` `src.catchAll.`**`CatchAll`**
Bases: `object`

class as simple dynamic dictionary with predefined keys as properties in inherited classes through __init__ method. Or NOT. with pretty print __str__ and __repr__ (indented as recursive) with jsonDumps()

Usage:
>> import catchAll as CAA
>> a = CAA.CatchAll()
>> a.tintin = “reporter”
>> a.milou = “dog”
>> print(“a=%s” % a)
>> print(“tintin: %s” % a.tintin)

as

>> a = {}
>> a[“tintin”] = “reporter”
>> a[“milou”] = “dog”
>> print(“tintin: %s” % a[“tintin”]

jsonDumps()

src.catchAll.**dumper**(*obj*)
to json explore subclass object as dict

src.catchAll.**dumperType**(*obj*)
to get a “_type” to trace json subclass object, but ignore all attributes begining with ‘_’

src.catchAll.**jsonDumps**(*obj*)
to get direct default jsonDumps method

src.coloringSat module

simple tagging as ‘<color>’ for simple coloring log messages on terminal(s) window or unix or ios using backend colorama. Using ‘<color>’ because EZ human readable, So ‘<color>’ are not supposed existing in log message. “{}”.format() is not choosen because “{}” are present in log messages of contents of python dict (as JSON) etc.

Usage:
>> import src.coloringSat as COLS

Example:
>> log(“this is in <green>color green<reset>, OK is in blue: <blue>OK?”)

class src.coloringSat.**ColoringStream**
Bases: object

write my stream class only write and flush are used for the streaming https://docs.python.org/2/library/logging.handlers.html https://stackoverflow.com/questions/31999627/storing-logger-messages-in-a-string

flush()

write(*astr*)

src.coloringSat.**cleanColors**(*msg*)
clean the message of color tags ‘<red> . . .

src.coloringSat.**indent**(*msg*, *nb*, *car=' '*)
indent nb car (spaces) multi lines message except first one

src.coloringSat.**log**(*msg*)
elementary log stdout for debug if _verbose

`src.coloringSat.`**`replace`**(*msg*, *tags*)

`src.coloringSat.`**`toColor`**(*msg*)

automatically clean the message of color tags '<red> … if the terminal output stdout is redirected by user if not, replace tags with ansi color codes

example: >> sat compile SALOME > log.txt

`src.coloringSat.`**`toColor_AnsiToWin32`**(*msg*)

for test debug no wrapping

src.compilation module

Utilities to build and compile

Usage:
>> import src.compilation as COMP

`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*)

Performs a build with a script.

`do_script_build`(*script*, *number_of_proc=0*)

`hack_libtool`()

`install`()

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

Shortcut method to log in log file.

`log_command`(*command*)

Shortcut method to log a command.

`make`(*nb_proc*, *make_opt=''*)

`prepare`()

Prepares the environment. Build two environment: one for building and one for testing (launch).

`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 **`file_name`** – (str) The name of the file to write

`wmake`(*nb_proc*, *opt_nb_proc=None*)

src.configManager module

class src.configManager.**ConfigManager**(*runner*)
Class that manages the read of all the config .pyconf files of salomeTools

create_config_file(*config*)
This method is called when there are no user config file. It build it from scratch.

Parameters **config** – (Config) The global config.

Returns (Config) The config corresponding to the file created.

get_command_line_overrides(*options*, *sections*)
get all the overwrites that are in the command line

Parameters

- **options** – The options from salomeTools class initialization (as '-l5' or '–overwrite')
- **sections** – (str) The config section to overwrite.

Returns (list) The list of all the overwrites to apply.

get_config(*application=None*, *options=None*, *command=None*, *datadir=None*)
get the config from all the configuration files.

Parameters

- **application** – (str) The application for which salomeTools is called.
- **options** – (Options) The general salomeTools options (as '–overwrite' or '-v5')
- **command** – (str) The command that is called.
- **datadir** – (str) The repository that contain external data for salomeTools.

Returns (Config) The final config.

get_user_config_file()
Get the user config file

Returns (str) path to the user config file.

set_user_config_file(*config*)
Set the user config file name and path. If necessary, build it from another one or create it from scratch.

Parameters **config** – (Config) The global config (containing all pyconf).

class src.configManager.**ConfigOpener**(*pathList*)
Class that helps to find an application pyconf in all the possible directories (pathList)

get_path(*name*)
The method that returns the entire path of the pyconf searched

Parameters **name** – (str) The name of the searched pyconf.

src.configManager.**check_path**(*path*, *ext=[]*)
Construct a text with the input path and colorized critical '** not found' if it does not exist. '** bad extension' if extension problem.

Parameters

- **path** – (str) The path to check.
- **ext** – (list) Verify that the path extension is in the list

Returns (str) The string of the path with information

src.configManager.**getConfigColored**(*config*, *path*, *stream*, *show_label=False*, *level=0*, *show_full_path=False*)
Get a colored representation value from a config pyconf instance. used recursively from the initial path.

Parameters

- **`config`** – (Config) The configuration from which the value is displayed.
- **`path`** – (str) The path in the configuration of the value to print.
- **`show_label`** – (bool) If True, do a basic display. (useful for bash completion)
- **`stream`** – The output stream used
- **`level`** – (int) The number of spaces to add before display.
- **`show_full_path`** – (bool) Display full path, else relative

`src.configManager.`**`get_config_children`**(*config*, *args*)

Gets the names of the children of the given parameter. Useful only for completion mechanism

Parameters

- **`config`** – (Config) The configuration where to read the values
- **`args`** – The path in the config from which get the keys

`src.configManager.`**`get_products_list`**(*self*, *options*, *cfg*, *logger*)

Gives the product list with their informations from configuration regarding the passed options.

Parameters

- **`options`** – (Options) The Options instance that stores the commands arguments
- **`config`** – (Config) The global configuration
- **`logger`** – (Logger) The logger instance to use for the display and logging

Returns (list) The list of (product name, product_informations).

`src.configManager.`**`print_debug`**(*config*, *aPath*, *logger*, *show_label=False*, *level=0*, *show_full_path=False*)

logger output for debugging a config/pyconf lines contains: path : expression –> 'evaluation'

Example:

PROJECTS.projects.salome.project_path : $PWD –> '/tmp/SALOME'

`src.configManager.`**`print_value`**(*config*, *path*, *logger*, *show_label=False*, *level=0*, *show_full_path=False*)

print a colored representation value from a config pyconf instance. used recursively from the initial path.

Param as getConfigColored

`src.configManager.`**`show_patchs`**(*config*, *logger*)

Prints all the used patchs in the application.

Parameters

- **`config`** – (Config) the global configuration.
- **`logger`** – (Logger) The logger instance to use for the display

`src.configManager.`**`show_product_info`**(*config*, *name*, *logger*)

Display on the terminal and logger information about a product.

Parameters

- **`config`** – (Config) the global configuration.
- **`name`** – (str) The name of the product
- **`logger`** – (Logger) The logger instance to use for the display

src.debug module

This file assume DEBUG functionalities use. Print salomeTools debug messages in sys.stderr. Show pretty print debug representation from instances of SAT classes (pretty print src.pyconf.Config)

Warning: 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

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.`**`format_color_exception`**(*msg, limit=None, trace=None*)
Format a stack trace and the exception information. as traceback.format_exception(), with color with traceback only if (_debug) or (DBG._user in DBG._developpers)

`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.`**`isTypeConfig`**(*var*)
To know if var is instance from Config/pyconf

`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)

`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 an environment context

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

Parameters

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

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

Parameters

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

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

Parameters

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

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

Parameters key – (str) the environment variable

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

Parameters key – (str) the environment variable to check

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

Parameters

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

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

Parameters

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

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

Parameters

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

- **value** – (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

- **filename** – (str) the file path
- **additional_env** – (dict) a dictionary of additional variables to add to the environment
- **for_package** – (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

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

Returns (str) The path to the generated file

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

add_line(*nb_line*)

Add empty lines to the out stream (in case of file generation)

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

add_warning(*warning*)

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

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

append(*key*, *value*, *sep=':'*)

append value to key using sep

Parameters

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

dump(*out*)

Write the environment to out

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

finish(*required*)

Add a final instruction in the out file (in case of file generation)

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

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

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

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

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

load_cfg_environment (*cfg_env*)
Loads environment defined in cfg_env

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

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

Parameters

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

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

Parameters

- **product_info** – (Config) The product description
- **logger** – (Logger) The logger instance to display messages
- **native** – (bool) 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

- **script_path** – (str) 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

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

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

Parameters

- **product** – (str) 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 **product_info** – (Config) 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
- **env_info** – (list) 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_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 **pi** – (Config) 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

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

src.environs module

Utility for print environment variables

Examples:
- split all or specific environment variables $XXX(s)…
  >> environs.py -> all
  >> environs.py SHELL PATH -> specific $SHELL $PATH

- split all or specific environment variables on pattern $*XXX*(s)…
  >> environs.py –pat ROOT -> specific $*ROOT*

- split search specific substrings in contents of environment variables $XXX(s)…
  >> environs.py –grep usr -> all specific environment variables containing usr

Tips:
- create unix alias as shortcut for bash console
  >> alias envs="…/environs.py"

`src.environs.`**`print_grep_environs`**(*args=[]*)

`src.environs.`**`print_split_environs`**(*args=[]*)

`src.environs.`**`print_split_pattern_environs`**(*args=[]*)

src.exceptionSat module

`exception` `src.exceptionSat.`**`ExceptionSat`**
Bases: `exceptions.Exception`

rename Exception Class for sat convenience (for future…)

src.fileEnviron module

`class` `src.fileEnviron.`**`BashFileEnviron`**(*output*, *environ=None*)
Bases: *`src.fileEnviron.FileEnviron`* (page 47)

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

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

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

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

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

Parameters

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

`class` `src.fileEnviron.`**`BatFileEnviron`**(*output*, *environ=None*)
Bases: *`src.fileEnviron.FileEnviron`* (page 47)

for Windows batch shell.

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

Parameters **`comment`** – (str) 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

- **key** – (str) the environment variable
- **command** – (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 **required** – (bool) Do nothing if required is False

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

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

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

Parameters

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

class src.fileEnviron.**ContextFileEnviron** (*output*, *environ=None*)
Bases: *src.fileEnviron.FileEnviron* (page 47)

Class for a salome context configuration file.

add_echo (*text*)
Add a comment

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

add_warning (*warning*)
Add a warning

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

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

Parameters

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

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

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

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

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

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

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

Parameters

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

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

Parameters

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

`class` `src.fileEnviron.`**`FileEnviron`** (*output*, *environ=None*)
Base class for shell environment

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

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

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

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

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

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

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

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

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

Parameters

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

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

Parameters

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

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

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

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

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

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

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

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

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

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

Parameters

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

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

Parameters

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

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

Parameters

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

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

`add_comment`(*comment*)

`add_echo`(*text*)
Add a comment

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

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

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

`add_warning`(*warning*)
Add a warning

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

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

Parameters

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

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

Parameters

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

change_to_launcher (*value*)
obsolete? do nothing

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

Parameters

- **key** – (str) the environment variable
- **command** – (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 **required** – (bool) Do nothing if required is False

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

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

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

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

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

Parameters

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

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

Parameters

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

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

Parameters

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

class src.fileEnviron.**ScreenEnviron** (*output*, *environ=None*)
Bases: *src.fileEnviron.FileEnviron* (page 47)

add_comment (*comment*)

add_echo (*text*)

add_line (*number*)

add_warning (*warning*)

append (*name*, *value*, *sep=':'*)

command_value (*key*, *command*)

get (*name*)

is_defined (*name*)

prepend (*name*, *value*, *sep=':'*)

run_env_script (*module*, *script*)

set (*name*, *value*)

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

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

Parameters

- **output** – (file) the output file stream.
- **shell** – (str) the type of shell syntax to use.
- **environ** – (dict) 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 **name** – (str) 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*)
Launch a batch

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

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

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

Parameters

- **logger** – (Logger) The logging instance

- **top** – (int) the number to display
- **delai** – (int) the number max
- **ss** – (str) the string to display

`src.fork.`**`write_back`**(*logger*, *message*)

shortcut function to write at the begin of the line

Parameters

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

src.loggingSat module

http://sametmax.com/ecrire-des-logs-en-python/ https://docs.python.org/3/library/time.html#time.strftime

use logging package for salometools

handler: on info() no format on other formatted indented on multi lines messages

class `src.loggingSat.`**`DefaultFormatter`**(*fmt=None*, *datefmt=None*)

Bases: `logging.Formatter`

`format`(*record*)

Format the specified record as text.

The record's attribute dictionary is used as the operand to a string formatting operation which yields the returned string. Before formatting the dictionary, a couple of preparatory steps are carried out. The message attribute of the record is computed using LogRecord.getMessage(). If the formatting string uses the time (as determined by a call to usesTime(), formatTime() is called to format the event time. If there is exception information, it is formatted using formatException() and appended to the message.

`setColorLevelname`(*levelname*)

class `src.loggingSat.`**`UnittestFormatter`**(*fmt=None*, *datefmt=None*)

Bases: `logging.Formatter`

`format`(*record*)

Format the specified record as text.

The record's attribute dictionary is used as the operand to a string formatting operation which yields the returned string. Before formatting the dictionary, a couple of preparatory steps are carried out. The message attribute of the record is computed using LogRecord.getMessage(). If the formatting string uses the time (as determined by a call to usesTime(), formatTime() is called to format the event time. If there is exception information, it is formatted using formatException() and appended to the message.

class `src.loggingSat.`**`UnittestStream`**

Bases: `object`

write my stream class only write and flush are used for the streaming https://docs.python.org/2/library/logging.handlers.html https://stackoverflow.com/questions/31999627/storing-logger-messages-in-a-string

`flush`()

`getLogs`()

`getLogsAndClear`()

`write`(*astr*)

`src.loggingSat.`**`dirLogger`**(*logger*)

`src.loggingSat.`**`getDefaultLogger`**()

`src.loggingSat.`**`getUnittestLogger`**()

`src.loggingSat.`**`indent`**`(`*msg*, *nb*, *car=' '*`)`
indent nb car (spaces) multi lines message except first one

`src.loggingSat.`**`indentUnittest`**`(`*msg*, *prefix=' | '*`)`
indent car multi lines message except first one car default is less spaces for size logs files keep human readable

`src.loggingSat.`**`initLoggerAsDefault`**`(`*logger*, *fmt=None*, *level=None*`)`
init logger as prefixed message and indented message if multi line exept info() outed 'as it' without any format

`src.loggingSat.`**`initLoggerAsUnittest`**`(`*logger*, *fmt=None*, *level=None*`)`
init logger as silent on stdout/stderr used for retrieve messages in memory for post execution unittest https://docs.python.org/2/library/logging.handlers.html

`src.loggingSat.`**`log`**`(`*msg*`)`
elementary log when no logger yet

`src.loggingSat.`**`testLogger_1`**`(`*logger*`)`
small test

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:
>> print(options.level)
>> 5

`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``()`

`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

src.product module

Contains the methods relative to the product notion of salomeTools.

Usage:
>> import src.product as PROD

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

Returns (tuple) as (boolean, str) True or false is the installation is found or not and if it is found, the path of the found installation

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

Returns (bool) True if it is well installed

src.product.**get_base_install_dir**(*config*, *prod_info*, *version*)
Compute the installation directory of a product in base

Parameters

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

Returns (str) The path of the product installation

src.product.**get_install_dir**(*config*, *base*, *version*, *prod_info*)
Compute the installation directory of a given product

Parameters

- **config** – (Config) The global configuration

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

Returns (str) The path of the product installation

src.product.**get_product_components**(*product_info*)
Get the component list to generate with the product

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

Returns (list) The list of names of the components

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

Returns (Config) The specific configuration of the product

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

Parameters

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

Returns (list) the list of products in dependence

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

Parameters

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

Returns (Config) The product description

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

Parameters

- **products** – (list) The list of product names
- **config** – (Config) The global configuration

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

src.product.**product_compiles**(*product_info*)

Know if a product compiles or not (some products do not have a compilation procedure)

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

Returns (bool) True if the product compiles, else False

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

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

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

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

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

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

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

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

Returns (bool) True if the product has one or more patches

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

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

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

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

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

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

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

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

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

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

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

Returns (bool) True if the product is autotools, else False

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

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

Returns (bool) True if the product is cmake, else False

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

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

Returns (bool) True if the product is a cpp, else False

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

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

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

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

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

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

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

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

Returns (bool) True if the product is fixed, else False

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

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

Returns (bool) True if the product is generated

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

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

Returns (bool) True if the product has openmpi inits dependencies

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

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

Returns (bool) True if the product is native, else False

src.product.**product_is_salome**(*product_info*)
Know if a product is of type salome

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

Returns (bool) True if the product is salome, else False

src.product.**product_is_sample**(*product_info*)
Know if a product has the sample type

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

Returns (bool) True if the product has the sample type, else False

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

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

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

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

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

Returns (bool) True if the product is vcs, else False

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 61)

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 58)

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 at 0x3b22aa0>*)
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 58)

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 61)

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 61)

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 a dot 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.returnCode module

This file contains ReturnCode class

Usage:
>> import returnCode as RCO

class src.returnCode.**ReturnCode**(*status=None*, *why=None*, *value=None*)
Bases: `object`

assume simple return code for methods, with explanation as 'why' obviously why is why it is not OK, but also why is why it is OK (if you want). and optionnaly contains a return value as self.getValue()

Usage: >> import returnCode as RCO

>> aValue = doSomethingToReturn() >> return RCO.ReturnCode("KO", "there is no problem here", aValue) >> return RCO.ReturnCode("KO", "there is a problem here because etc", None) >> return RCO.ReturnCode("TIMEOUT_STATUS", "too long here because etc") >> return RCO.ReturnCode("NA", "not applicable here because etc")

>> rc = doSomething() >> print("short returnCode string", str(rc)) >> print("long returnCode string with value", repr(rc))

>> rc1 = RCO.ReturnCode("OK", ...) >> rc2 = RCO.ReturnCode("KO", ...) >> rcFinal = rc1 + rc2 >> print("long returnCode string with value", repr(rcFinal)) # KO!

KFSYS = 4

KNOWNFAILURE_STATUS = 'KF'

KOSYS = 1

KO_STATUS = 'OK'

NASYS = 2

NA_STATUS = 'NA'

NDSYS = 3

OKSYS = 0

OK_STATUS = 'OK'

TIMEOUT_STATUS = 'TIMEOUT'

TOSYS = 5

UNKNOWN_STATUS = 'ND'

getValue()

getWhy()
return why as str or list if sum or some ReturnCode

indent(*text*, *amount=5*, *ch=' '*)
indent multi lines message

isOk()
return True if ok

raiseIfKo()
raise an exception with message why if not ok

setStatus(*status*, *why=None*, *value=None*)

setValue(*value*)

setWhy (*why*)

toSys ()
return system return code as bash or bat

src.salomeTools module

This file is the main entry file to salomeTools NO __main__ entry allowed, use 'sat' (in parent directory)

class `src.salomeTools.`**Sat** (*logger*)
Bases: `object`

The main class that stores all the commands of salomeTools (usually known as 'runner' argument in Command classes)

assumeAsList (*strOrList*)

execute_cli (*cli_arguments*)
select first argument as a command in directory 'commands', and launch on arguments

Parameters **cli_arguments** – (str or list) The sat cli arguments (as sys.argv)

getColoredVersion ()
get colored salomeTools version message

getCommandAndAppli (*arguments*)

getCommandInstance (*name*)
returns inherited instance of Command(_BaseCmd) for command 'name' if module not loaded yet, load it.

getConfig ()

getConfigManager ()

getLogger ()

getModule (*name*)
returns only-one-time loaded module Command 'name' assume load if not done yet

get_help ()
get general help colored string

parseArguments (*arguments*)

print_help ()
prints salomeTools general help

`src.salomeTools.`**assumeAsList** (*strOrList*)
return a list as sys.argv if string

`src.salomeTools.`**find_command_list** (*dirPath*)
Parse files in dirPath that end with '.py' : it gives commands list

Parameters **dirPath** – (str) The directory path where to search the commands

Returns (list) the list containing the commands name

`src.salomeTools.`**getCommandsList** ()
Gives commands list (as basename of files .py in directory commands

`src.salomeTools.`**getVersion** ()
get version number as string

`src.salomeTools.`**launchSat** (*command*)
launch sat as subprocess.Popen command as string ('sat –help' for example) used for unittest, or else…

Returns (stdout, stderr) tuple of subprocess.Popen output src.salomeTools.**setLocale**()

reset initial locale at any moment 'fr' or else (TODO) from initial environment var '$LANG' 'i18n' as 'internationalization'

src.salomeTools.**setNotLocale**()

force english at any moment

src.system module

All utilities method doing a system call, like open a browser or an editor, or call a git command

Usage:
>> import src.system as SYSS

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

Extracts sources from an archive.

Parameters

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

Returns (bool) True if the extraction is successful

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

Extracts sources from a cvs repository.

Parameters

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

Returns (bool) True if the extraction is successful

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

Extracts sources from a git repository.

Parameters

- **from_what** – (str) The remote git repository.
- **tag** – (str) The tag.
- **where** – (str) The path where to extract.
- **logger** – (Logger) The logger instance to use.

- **environment** – (Environ) The environment to source when extracting.

Returns (bool) True if the extraction is successful

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

Parameters

- **editor** – (str) The editor to use.
- **filePath** – (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

- **user** – (str) The user to be used.
- **from_what** – (str) The remote git repository.
- **tag** – (str) The tag.
- **where** – (str) The path where to extract.
- **logger** – (Logger) The logger instance to use.
- **checkout** – (bool) If true use checkout svn.
- **environment** – (Environ) The environment to source when extracting.

Returns (bool) True if the extraction is successful

src.template module

template for substitute strings in template files

Usage:
>> import src.template as TPLATE

class `src.template.`**`MyTemplate`**(*template*)
Bases: `string.Template`

`delimiter = '\xc2\xa4'`

`pattern = <_sre.SRE_Pattern object at 0x40e4170>`

`src.template.`**`substitute`**(*template_file*, *subst_dic*)

src.test_module module

test_module for test base dir/git/svn etc

Usage:
>> import src.test_module as TMOD

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*)
Generates the script to be run by Salome. This python script includes init and close statements and a loop calling all the scripts of a single directory.

get_test_timeout (*test_name*, *default_value*)

get_tmp_dir ()
Find the getTmpDir function that gives access to xxxpidict file directory. (the xxxpidict file exists when SALOME is launched)

prepare_testbase (*test_base_name*)
Configure tests base.

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*)
Read the xxx.result.py files.

run_all_tests ()

run_grid_tests ()
Runs all tests of a grid.

run_script (*script_name*)

run_session_tests ()
Runs all tests of a session.

run_testbase_tests ()
Runs test testbase

run_tests (*listTest*, *ignoreList*)
Runs tests of a session (using a single instance of Salome).

search_known_errors (*status*, *test_grid*, *test_session*, *test*)
Searches if the script is declared in known errors pyconf. Update the status if needed.

write_test_margin (*tab*)
Write margin to show test results. indent with '| ... +'

src.test_module.**getTmpDirDEFAULT** ()
Get directory to be used for the temporary files.

src.utilsSat module

utilities for sat general useful simple methods all-in-one import srs.utilsSat as UTS

Usage:
>> import srsc.utilsSat as UTS
>> UTS.ensure_path_exists(path)

class src.utilsSat.**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*)

src.utilsSat.**black**(*msg*)

src.utilsSat.**blue**(*msg*)

src.utilsSat.**check_config_has_application**(*config*)
Check that the config has the key APPLICATION. Else raise an exception.

> **Parameters** **config** – (Config) The config.

src.utilsSat.**check_config_has_profile**(*config*)
Check that the config has the key APPLICATION.profile. Else, raise an exception.

> **Parameters** **config** – (Config) The config.

src.utilsSat.**check_has_key**(*inConfig*, *key*)
Check that the in-Config node has the named key (as an attribute)

> **Parameters**
>
> - **inConfig** – (Config or Mapping etc) The in-Config node
> - **key** – (str) The key to check presence in in-Config node
>
> **Returns** (RCO.ReturnCode) 'OK' if presence

src.utilsSat.**critical**(*msg*)

src.utilsSat.**cyan**(*msg*)

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

> **Parameters** **date** – (str) The date in format YYYYMMDD_HHMMSS
>
> **Returns** (tuple) as (str,str,str,str,str,str) The same date and time in separate variables.

src.utilsSat.**deepcopy_list**(*input_list*)
Do a deep copy of a list

> **Parameters** **input_list** – (list) The list to copy
>
> **Returns** (list) The copy of the list

src.utilsSat.**ensure_path_exists**(*path*)
Create a path if not existing

> **Parameters** **path** – (str) The path.

src.utilsSat.**error**(*msg*)

`src.utilsSat.`**`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, 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

- **`file_name`** – (str) The file name to search
- **`lpath`** – (list) The list of directories where to search
- **`additional_dir`** – (str) The name of the additional directory

Returns (str) The full path of the file or False if not found

`src.utilsSat.`**`formatTuples`**(*tuples*)
Format 'label = value' the tuples in a tabulated way.

Parameters **`tuples`** – (list) The list of tuples to format

Returns (str) The tabulated text. (as mutiples lines)

`src.utilsSat.`**`formatValue`**(*label*, *value*, *suffix=''*)
format 'label = value' with the info color

Parameters

- **`label`** – (int) the label to print.
- **`value`** – (str) the value to print.
- **`suffix`** – (str) the optionnal suffix to add at the end.

`src.utilsSat.`**`get_CONFIG_FILENAME`**()
get initial config.pyconf

`src.utilsSat.`**`get_base_path`**(*config*)
Returns the path of the products base.

Parameters **`config`** – (Config) The global Config instance.

Returns (str) The path of the products base.

`src.utilsSat.`**`get_config_key`**(*inConfig*, *key*, *default*)
Search for key value in config node 'inConfig[key]' as 'inConfig.key' If key is not in inCconfig, then return default, else, return the found value

Parameters

- **`inConfig`** – (Config or Mapping etc) The in-Config node.
- **`key`** – (str) the name of the parameter to get the value
- **`default`** – (str) The value to return if key is not in-Config

Returns (if supposedly leaf (str),else (in-Config Node)

`src.utilsSat.`**`get_launcher_name`**(*config*)
Returns the name of application file launcher, 'salome' by default.

Parameters **`config`** – (Config) The global Config instance.

Returns (str) The name of salome launcher.

`src.utilsSat.`**`get_log_path`**(*config*)
Returns the path of the logs.

Parameters **`config`** – (Config) The global Config instance.

Returns (str) The path of the logs.

`src.utilsSat.`**`get_property_in_product_cfg`**(*product_cfg*, *pprty*)

`src.utilsSat.`**`get_salome_version`**(*config*) src.utilsSat.**get_tmp_filename**(*config*, *name*)

src.utilsSat.**green**(*msg*)

src.utilsSat.**handleRemoveReadonly**(*func*, *path*, *exc*)

src.utilsSat.**header**(*msg*)

src.utilsSat.**info**(*msg*)

src.utilsSat.**label**(*msg*)

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

Parameters

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

Returns (list) the list of files path and informations about it

src.utilsSat.**log_res_step**(*logger*, *res*)

src.utilsSat.**log_step**(*logger*, *header*, *step*)

src.utilsSat.**logger_info_tuples**(*logger*, *tuples*)
For convenience format as formatTuples() and call logger.info()

src.utilsSat.**magenta**(*msg*)

src.utilsSat.**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.utilsSat.**normal**(*msg*)

src.utilsSat.**only_numbers**(*str_num*)

src.utilsSat.**parse_date**(*date*)
Transform YYYYMMDD_hhmmss into YYYY-MM-DD hh:mm:ss.

Parameters **date** – (str) The date to transform

Returns (str) The date in the new format

src.utilsSat.**read_config_from_a_file**(*filePath*)

src.utilsSat.**red**(*msg*)

src.utilsSat.**remove_item_from_list**(*input_list*, *item*)
Remove all occurences of item from input_list

Parameters **input_list** – (list) The list to modify

Returns (list) The without any item

src.utilsSat.**replace_in_file**(*file_in*, *str_in*, *str_out*)
Replace <str_in> by <str_out> in file <file_in>

Parameters

- **file_in** – (str) The file name
- **str_in** – (str) The string to search
- **str_out** – (str) The string to replace.

src.utilsSat.**reset**(*msg*)

src.utilsSat.**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

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

Returns (RCO.ReturnCode) OK if cmd is not in notShownCommands and the application in the log file corresponds to application ReturnCode value is tuple (appliLog, launched_cmd)

`src.utilsSat.`**`success`**(*msg*)

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

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

Returns (float) The number of seconds corresponding to timedelta.

`src.utilsSat.`**`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

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

`src.utilsSat.`**`warning`**(*msg*)

`src.utilsSat.`**`white`**(*msg*)

`src.utilsSat.`**`yellow`**(*msg*)

src.xmlManager module

Utilities to read xml logging files

Usage:
>> import src.xmlManager as XMLMGR

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 (dict) The attributes of the root node

`get_attrib`(*node_name*)
Get the attibutes of the node node_name in self.xmlroot

Parameters **node_name** – (str) the name of the node

Returns (dict) the attibutes of the node node_name in self.xmlroot

`get_node_text`(*node*)
Get the text of the first node that has name that corresponds to the parameter node

Parameters **node** – (str) the name of the node from which get the text

Returns (str) The text of the first node that has name that corresponds to the parameter node

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

- **node_name** – (str) the name of the node to add
- **text** – (str) the text of the node
- **attrib** – (dict) 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

- **node_name** – (str) The name of the node on which append text
- **attrib** – (dict) The attrib to append

append_node_text(*node_name*, *text*)

Append a new text to the node that has node_name as name

Parameters

- **node_name** – (str) The name of the node on which append text
- **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 **stylesheet** – (str) 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

- **root_node** – (etree.Element) the Etree element where to add the new node
- **node_name** – (str) the name of the node to add
- **text** – (str) the text of the node
- **attrib** – (dict) the dictionary containing the attribute(s) 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

- **root_node** – (etree.Element) the Etree element where to append the new attibutes
- **attrib** – (dict) The attrib to append

src.xmlManager.**find_node_by_attrib**(*xmlroot*, *name_node*, *key*, *value*)

Find the first node from xmlroot that has name name_node and that has in its attributes {key : value}. Return the node

Parameters

- **xmlroot** – (etree.Element) the Etree element where to search
- **name_node** – (str) the name of node to search
- **key** – (str) the key to search
- **value** – (str) the value to search

Returns (etree.Element) the found node

`src.xmlManager.`**`write_report`**(*filename*, *xmlroot*, *stylesheet*)
Writes a report file from a XML tree.

Parameters

- **`filename`** – (str) The path to the file to create
- **`xmlroot`** – (etree.Element) the Etree element to write to the file
- **`stylesheet`** – (str) The stylesheet to add to the begin of the file

Module contents

4.2 commands

4.2.1 commands package

Submodules

commands.application module

Is a salomeTools command module see Command class docstring, also used for help

`class` `commands.application.`**`Command`**(*runner*)
Bases: `src.salomeTools._BaseCommand`

The application command creates a SALOME application.

Warning:
It works only for SALOME 6.
Use the 'launcher' command for newer versions of SALOME

Examples:
>> sat application SALOME-6.6.0

`getParser`()
Define all options for command 'sat application <options>'

`name = 'application'`

`run`(*cmd_arguments*)
method called for command 'sat application <options>'

`commands.application.`**`add_module_to_appli`**(*out*, *module*, *has_gui*, *module_path*, *logger*, *flagline*)
add the definition of a module to out stream.

`commands.application.`**`create_application`**(*config*, *appli_dir*, *catalog*, *logger*, *display=True*)
reates a SALOME application.

`commands.application.`**`create_config_file`**(*config*, *modules*, *env_file*, *logger*)
Creates the config file to create an application with the list of modules.

`commands.application.`**`customize_app`**(*config*, *appli_dir*, *logger*)
Customizes the application by editing SalomeApp.xml.

`commands.application.`**`generate_application`**(*config*, *appli_dir*, *config_file*, *logger*)
Generates the application with the config_file.

`commands.application.`**`generate_catalog`**(*machines*, *config*, *logger*)
Generates the catalog from a list of machines.

`commands.application.`**`generate_launch_file`**(*config*, *appli_dir*, *catalog*, *logger*, *l_SALOME_modules*)
Obsolescent way of creating the application. This method will use appli_gen to create the application directory.

`commands.application.`**`get_SALOME_modules`**(*config*)

`commands.application.`**`get_step`**(*logger*, *message*, *pad=50*)
returns 'message .... ' with pad 50 by default avoid colors '<color>' for now in message

`commands.application.`**`make_alias`**(*appli_path*, *alias_path*, *force=False*)
Creates an alias for runAppli

commands.check module

`class` `commands.check.`**`Command`**(*runner*)
Bases: `src.salomeTools._BaseCommand`

The check command executes the 'check' command in the build directory of all the products of the application. It is possible to reduce the list of products to check by using the –products option

Examples:
>> sat check SALOME –products KERNEL,GUI,GEOM

`getParser`()
Define all options for the check command 'sat check <options>'

`name = 'check'`

`run`(*cmd_arguments*)
method called for command 'sat check <options>'

`commands.check.`**`check_all_products`**(*config*, *products_infos*, *logger*)
Execute the proper configuration commands in each product build directory.

Parameters

- **`config`** – (Config) The global configuration
- **`products_info`** – (list) List of (str, Config) => (product_name, product_info)
- **`logger`** – (Logger) The logger instance to use for the display and logging

Returns (int) the number of failing commands.

`commands.check.`**`check_product`**(*p_name_info*, *config*, *logger*)
Execute the proper configuration command(s) in the product build directory.

Parameters

- **`p_name_info`** – (tuple) (str, Config) => (product_name, product_info)
- **`config`** – (Config) The global configuration
- **`logger`** – (Logger) The logger instance to use for the display and logging

Returns (int) 1 if it fails, else 0.

`commands.check.`**`get_products_list`**(*options*, *cfg*, *logger*)
method that gives the product list with their informations from configuration regarding the passed options.

Parameters

- **options** – (Options) The Options instance that stores the commands arguments
- **cfg** – (Config) The global configuration
- **logger** – (Logger) The logger instance to use for the display and logging

Returns (list) The list of (product name, product_informations).

commands.clean module

class `commands.clean.`**`Command`**(*runner*)

Bases: `src.salomeTools._BaseCommand`

The clean command suppresses the source, build, or install directories of the application products. Use the options to define what directories you want to suppress and to reduce the list of products

Examples:
>> sat clean SALOME –build –install –properties is_salome_module:yes

`getParser`()

Define all options for the command 'sat clean <options>'

`name = 'clean'`

`run`(*cmd_arguments*)

method called for command 'sat clean <options>'

`commands.clean.`**`get_build_directories`**(*products_infos*)

Returns the list of directory build paths corresponding to the list of product information given as input.

Parameters **products_infos** – (list) The list of (name, config) corresponding to one product.

Returns (list) the list of build paths.

`commands.clean.`**`get_install_directories`**(*products_infos*)

Returns the list of directory install paths corresponding to the list of product information given as input.

Parameters **products_infos** – (list) The list of (name, config) corresponding to one product.

Returns (list) the list of install paths.

`commands.clean.`**`get_source_directories`**(*products_infos*, *without_dev*)

Returns the list of directory source paths corresponding to the list of product information given as input. If without_dev (bool), then the dev products are ignored.

Parameters

- **products_infos** – (list) The list of (name, config) corresponding to one product.
- **without_dev** – (boolean) If True, then ignore the dev products.

Returns (list) the list of source paths.

`commands.clean.`**`product_has_dir`**(*product_info*, *without_dev=False*)

Returns a boolean at True if there is a source, build and install directory corresponding to the product described by product_info.

Parameters **products_info** – (Config) The config corresponding to the product.

Returns (bool) True if there is a source, build and install directory corresponding to the product described by product_info.

`commands.clean.`**`suppress_directories`**(*l_paths*, *logger*)

Suppress the paths given in the list in l_paths.

Parameters

- **`l_paths`** – (list) The list of Path to be suppressed
- **`logger`** – (Logger) The logger instance to use for the display and logging

commands.compile module

class `commands.compile.`**`Command`**(*runner*)

Bases: `src.salomeTools._BaseCommand`

The compile command constructs the products of the application

Examples:

>> sat compile SALOME –products KERNEL,GUI,MEDCOUPLING –clean_all

`getParser`()

Define all options for the command 'sat compile <options>'

`name = 'compile'`

`run`(*cmd_arguments*)

method called for command 'sat compile <options>'

`commands.compile.`**`add_compile_config_file`**(*p_info*, *config*)

Execute the proper configuration command(s) in the product build directory.

Parameters

- **`p_info`** – (Config) The specific config of the product
- **`config`** – (Config) The global configuration

`commands.compile.`**`check_dependencies`**(*config*, *p_name_p_info*)

`commands.compile.`**`compile_all_products`**(*sat*, *config*, *options*, *products_infos*, *logger*)

Execute the proper configuration commands in each product build directory.

Parameters

- **`config`** – (Config) The global configuration
- **`products_info`** – (list) List of (str, Config) => (product_name, product_info)
- **`logger`** – (Logger) The logger instance to use for the display and logging

Returns (int) the number of failing commands.

`commands.compile.`**`compile_product`**(*sat*, *p_name_info*, *config*, *options*, *logger*, *header*, *len_end*)

Execute the proper configuration command(s) in the product build directory.

Parameters

- **`p_name_info`** – (tuple) (str, Config) => (product_name, product_info)
- **`config`** – (Config) The global configuration
- **`logger`** – (Logger) The logger instance to use for the display and logging
- **`header`** – (str) the header to display when logging
- **`len_end`** – (int) the lenght of the the end of line (used in display)

Returns (int) 1 if it fails, else 0.

commands.compile.**compile_product_cmake_autotools**(*sat*, *p_name_info*, *config*, *options*, *logger*, *header*, *len_end*)

Execute the proper build procedure for autotools or cmake in the product build directory.

Parameters

- **p_name_info** – (tuple) (str, Config) => (product_name, product_info)
- **config** – (Config) The global configuration
- **logger** – (Logger) The logger instance to use for the display and logging
- **header** – (str) the header to display when logging
- **len_end** – (int) the length of the the end of line (used in display)

Returns (int) 1 if it fails, else 0.

commands.compile.**compile_product_script**(*sat*, *p_name_info*, *config*, *options*, *logger*, *header*, *len_end*)

Execute the script build procedure in the product build directory.

Parameters

- **p_name_info** – (tuple) (str, Config) => (product_name, product_info)
- **config** – (Config) The global configuration
- **logger** – (Logger) The logger instance to use for the display and logging
- **header** – (str) the header to display when logging
- **len_end** – (int) the lenght of the the end of line (used in display)

Returns (int) 1 if it fails, else 0.

commands.compile.**extend_with_children**(*config*, *p_infos*)

commands.compile.**extend_with_fathers**(*config*, *p_infos*)

commands.compile.**get_children**(*config*, *p_name_p_info*)

commands.compile.**get_products_list**(*options*, *cfg*, *logger*)

method that gives the product list with their informations from configuration regarding the passed options.

Parameters

- **options** – (Options) The Options instance that stores the commands arguments
- **cfg** – (Config) The global configuration
- **logger** – (Logger) The logger instance to use for the display and logging

Returns (list) The list of (product name, product_informations).

commands.compile.**get_recursive_children**(*config*, *p_name_p_info*, *without_native_fixed=False*)

Get the recursive list of the product that depend on the product defined by prod_info

Parameters

- **config** – (Config) The global configuration
- **prod_info** – (Config) The specific config of the product
- **without_native_fixed** – (bool) If true, do not include the fixed or native products in the result

Returns (list) The list of product_informations.

commands.compile.**get_recursive_fathers**(*config*, *p_name_p_info*, *without_native_fixed=False*)

Get the recursive list of the dependencies of the product defined by prod_info

Parameters

- **`config`** – (Config) The global configuration
- **`prod_info`** – (Config) The specific config of the product
- **`without_native_fixed`** – (bool) If true, do not include the fixed or native products in the result

Returns (list) The list of product_informations.

`commands.compile.`**`sort_products`**`(`*config*, *p_infos*`)`
Sort the p_infos regarding the dependencies between the products

Parameters

- **`config`** – (Config) The global configuration
- **`p_infos`** – (list) List of (str, Config) => (product_name, product_info)

commands.config module

`class` `commands.config.`**`Command`**`(`*runner*`)`
Bases: `src.salomeTools._BaseCommand`

The config command allows manipulation and operation on config '.pyconf' files.

Examples:
>> sat config –list
>> sat config SALOME –edit
>> sat config SALOME –copy SALOME-new
>> sat config SALOME –value VARS
>> sat config SALOME –debug VARS
>> sat config SALOME –info ParaView
>> sat config SALOME –show_patchs

`getParser``()`
Define all options for command 'sat config <options>'

`name = 'config'`

`run``(`*cmd_arguments*`)`
method called for command 'sat config <options>'

commands.configure module

`class` `commands.configure.`**`Command`**`(`*runner*`)`
Bases: `src.salomeTools._BaseCommand`

The configure command executes in the build directory commands corresponding to the compilation mode of the application products. The possible compilation modes are 'cmake', 'autotools', or 'script'.

Here are the commands to be run:
autotools: >> build_configure and configure
cmake: >> cmake
script: (do nothing)

Examples:
>> sat configure SALOME –products KERNEL,GUI,PARAVIS

getParser()
Define all options for command 'sat configure <options>'

name = 'configure'

run(*cmd_arguments*)
method called for command 'sat configure <options>'

commands.configure.**configure_all_products**(*config*, *products_infos*, *conf_option*, *logger*)
Execute the proper configuration commands in each product build directory.

Parameters

- **config** – (Config) The global configuration
- **products_info** – (list) List of (str, Config) => (product_name, product_info)
- **conf_option** – (str) The options to add to the command
- **logger** – (Logger) The logger instance to use for the display and logging

Returns (int) the number of failing commands.

commands.configure.**configure_product**(*p_name_info*, *conf_option*, *config*, *logger*)
Execute the proper configuration command(s) in the product build directory.

Parameters

- **p_name_info** – (tuple) (str, Config) => (product_name, product_info)
- **conf_option** – (str) The options to add to the command
- **config** – (Config) The global configuration
- **logger** – (Logger) The logger instance to use for the display and logging

Returns (int) 1 if it fails, else 0.

commands.configure.**get_products_list**(*options*, *cfg*, *logger*)
method that gives the product list with their informations from configuration regarding the passed options.

Parameters

- **options** – (Options) The Options instance that stores the commands arguments
- **cfg** – (Config) The global configuration
- **logger** – (Logger) The logger instance to use for the display and logging

Returns (list) The list of (product name, product_informations).

commands.environ module

class commands.environ.**Command**(*runner*)
Bases: src.salomeTools._BaseCommand

The environ command generates the environment files of your application.

Examples:
>> sat environ SALOME

getParser()
Define all options for command 'sat environ <options>'

name = 'environ'

run (*cmd_arguments*)
method called for command 'sat environ <options>'

commands.environ.**write_all_source_files** (*config, logger, out_dir=None, src_root=None, silent=False, shells=['bash'], prefix='env', env_info=None*)
Generates the environment files.

Parameters

- **config** – (Config) The global configuration
- **logger** – (Logger) The logger instance to use for the display and logging
- **out_dir** – (str) The path to the directory where the files will be put
- **src_root** – (str) The path to the directory where the sources are
- **silent** – (bool) If True, do not print anything in the terminal
- **shells** – (list) The list of shells to generate
- **prefix** – (str) The prefix to add to the file names.
- **env_info** – (str) The list of products to add in the files.

Returns (list) The list of the generated files.

commands.find_duplicates module

class commands.find_duplicates.**Command** (*runner*)
Bases: src.salomeTools._BaseCommand

The find_duplicates command search recursively for all duplicates files in INSTALL directory (or the optionally given directory) and prints the found files to the terminal.

Examples:
>> sat find_duplicates –path /tmp

getParser ()
Define all options for command 'sat find_duplicates <options>'

name = 'find_duplicates'

run (*cmd_arguments*)
method called for command 'sat find_duplicates <options>'

class commands.find_duplicates.**Progress_bar** (*name, valMin, valMax, logger, length=50*)
Create a progress bar in the terminal

display_value_progression (*val*)
Display the progress bar.

Parameters val – (float) val must be between valMin and valMax.

commands.find_duplicates.**format_list_of_str** (*l_str*)
Make a list from a string

Parameters l_str – (list or str) The variable to format

Returns (list) the formatted variable

commands.find_duplicates.**list_directory** (*lpath, extension_ignored, files_ignored, directories_ignored*)
Make the list of all files and paths that are not filtered

Parameters

- **`lpath`** – (list) The list of path to of the directories where to search for duplicates
- **`extension_ignored`** – (list) The list of extensions to ignore
- **`files_ignored`** – (list) The list of files to ignore
- **`directories_ignored`** – (list) The list of directory paths to ignore

Returns (list, list) files_arb_out is the list of [file, path] and files_out is is the list of files

commands.generate module

class `commands.generate.`**`Command`**(*runner*)

Bases: `src.salomeTools._BaseCommand`

The generate command generates SALOME modules from 'pure cpp' products.

warning: this command NEEDS YACSGEN to run.

Examples:
>> sat generate SALOME –products FLICACPP

`getParser`()

Define all options for command 'sat generate <options>'

`name = 'generate'`

`run`(*cmd_arguments*)

method called for command 'sat generate <options>'

`commands.generate.`**`build_context`**(*config*, *logger*)

`commands.generate.`**`check_module_generator`**(*directory=None*)

Check if module_generator is available.

Parameters **`directory`** – (str) The directory of YACSGEN.

Returns (str) The YACSGEN path if the module_generator is available, else None

`commands.generate.`**`check_yacsgen`**(*config*, *directory*, *logger*)

Check if YACSGEN is available.

Parameters

- **`config`** – (Config) The global configuration.
- **`directory`** – (str) The directory given by option –yacsgen
- **`logger`** – (Logger) The logger instance

Returns (RCO.ReturnCode) with value The path to yacsgen directory if ok

`commands.generate.`**`generate_component`**(*config*, *compo*, *product_info*, *context*, *header*, *logger*)

`commands.generate.`**`generate_component_list`**(*config*, *product_info*, *context*, *logger*)

commands.init module

class `commands.init.`**`Command`**(*runner*)

Bases: `src.salomeTools._BaseCommand`

The init command Changes the local settings of SAT

getParser()
Define all options for command 'sat init <options>'

name = 'init'

run(*cmd_arguments*)
method called for command 'sat init <options>'

`commands.init.`**check_path**(*path_to_check*, *logger*)
Verify that the given path is not a file and can be created.

Parameters

- **path_to_check** – (str) The path to check.
- **logger** – (Logger) The logger instance.

`commands.init.`**display_local_values**(*config*, *logger*)
Display the base path

Parameters

- **config** – (Config) The global configuration.
- **key** – (str) The key from which to change the value.
- **logger** – (Logger) The logger instance.

`commands.init.`**set_local_value**(*config*, *key*, *value*, *logger*)
Edit the site.pyconf file and change a value.

Parameters

- **config** – (Config) The global configuration.
- **key** – (str) The key from which to change the value.
- **value** – (str) The path to change.
- **logger** – (Logger) The logger instance.

Returns (int) 0 if all is OK, else 1

commands.job module

class `commands.job.`**Command**(*runner*)
Bases: `src.salomeTools._BaseCommand`

The job command executes the commands of the job defined in the jobs configuration file | Examples: | >> sat job –jobs_config my_jobs –name my_job"

getParser()
Define all options for command 'sat job <options>'

name = 'job'

run(*cmd_arguments*)
method called for command 'sat job <options>'

commands.jobs module

class `commands.jobs.`**Command**(*runner*)
Bases: `src.salomeTools._BaseCommand`

The jobs command command launches maintenances that are described in the dedicated jobs configuration file.

Examples:

>> sat jobs –name my_jobs –publish

getParser()
Define all options for command 'sat jobs <options>'

name = 'jobs'

run(*cmd_arguments*)
method called for command 'sat jobs <options>'

class commands.jobs.**Gui**(*xml_dir_path*, *l_jobs*, *l_jobs_not_today*, *prefix*, *logger*, *file_boards=''*)
Bases: object

Class to manage the the xml data that can be displayed in a browser to see the jobs states

add_xml_board(*name*)
Add a board to the board list

Parameters **name** – (str) the board name

find_history(*l_jobs*, *l_jobs_not_today*)
find, for each job, in the existent xml boards the results for the job. Store the results in the dictionary self.history = {name_job : list of (date, status, list links)}

Parameters

- **l_jobs** – (list) the list of jobs to run today
- **l_jobs_not_today** – (list) the list of jobs that do not run today

find_test_log(*l_remote_log_files*)
Find if there is a test log (board) in the remote log files and the path to it. There can be several test command, so the result is a list.

Parameters **l_remote_log_files** – (list) the list of all remote log files

Returns (list) the list of tuples (test log files path, res of the command)

initialize_boards(*l_jobs*, *l_jobs_not_today*)
Get all the first information needed for each file and write the first version of the files

Parameters

- **l_jobs** – (list) the list of jobs that run today
- **l_jobs_not_today** – (list) the list of jobs that do not run today

last_update(*finish_status='finished'*)
update information about the jobs for the file xml_file

Parameters

- **l_jobs** – (list) the list of jobs that run today
- **xml_file** – (xmlManager.XmlLogFile) the xml instance to update

parse_csv_boards(*today*)
Parse the csv file that describes the boards to produce and fill the dict d_input_boards that contain the csv file contain

Parameters **today** – (int) the current day of the week

put_jobs_not_today(*l_jobs_not_today*, *xml_node_jobs*)
Get all the first information needed for each file and write the first version of the files

Parameters

- **xml_node_jobs** – (etree.Element) the node corresponding to a job

- **l_jobs_not_today** – (list) the list of jobs that do not run today

update_xml_file(*l_jobs*, *xml_file*)
update information about the jobs for the file xml_file

Parameters

- **l_jobs** – (list) the list of jobs that run today
- **xml_file** – (xmlManager.XmlLogFile) the xml instance to update

update_xml_files(*l_jobs*)
Write all the xml files with updated information about the jobs

Parameters **l_jobs** – (list) the list of jobs that run today

write_xml_file(*xml_file*, *stylesheet*)
Write one xml file and the same file with prefix

write_xml_files()
Write the xml files

class commands.jobs.**Job**(*name*, *machine*, *application*, *board*, *commands*, *timeout*, *config*, *job_file_path*, *logger*, *after=None*, *prefix=None*)
Bases: `object`

Class to manage one job

cancel()
In case of a failing job, one has to cancel every job that depend on it. This method put the job as failed and will not be executed.

check_time()
Verify that the job has not exceeded its timeout. If it has, kill the remote command and consider the job as finished.

get_log_files()
Get the log files produced by the command launched on the remote machine, and put it in the log directory of the user, so they can be accessible from

get_pids()
Get the pid(s) corresponding to the command that have been launched On the remote machine

Returns (list) The list of integers corresponding to the found pids

get_status()
Get the status of the job (used by the Gui for xml display)

Returns (str) The current status of the job

has_begun()
Returns True if the job has already begun

Returns (bool) True if the job has already begun

has_failed()
Returns True if the job has failed. A job is considered as failed if the machine could not be reached, if the remote command failed, or if the job finished with a time out.

Returns (bool) True if the job has failed

has_finished()
Returns True if the job has already finished (i.e. all the commands have been executed) If it is finished, the outputs are stored in the fields out and err.

Returns (bool) True if the job has already finished

is_running()
Returns True if the job commands are running

Returns (bool) True if the job is running

is_timeout()
Returns True if the job commands has finished with timeout

Returns (bool) True if the job has finished with timeout

kill_remote_process(*wait=1*)
Kills the process on the remote machine.

Returns (str, str) the output of the kill, the error of the kill

run()
Launch the job by executing the remote command.

time_elapsed()
Get the time elapsed since the job launching

Returns The number of seconds

Return type int

total_duration()
Gives the total duration of the job

Returns (int) the total duration of the job in seconds

write_results()
Display on the terminal all the job's information

class commands.jobs.**Jobs**(*runner*, *logger*, *job_file_path*, *config_jobs*, *lenght_columns=20*)
Bases: `object`

Class to manage the jobs to be run

cancel_dependencies_of_failing_jobs()
Cancels all the jobs that depend on a failing one.

Returns None

define_job(*job_def*, *machine*)
Takes a pyconf job definition and a machine (from class machine) and returns the job instance corresponding to the definition.

Parameters

- **job_def** – (Mapping a job definition
- **machine** – (Machine) the machine on which the job will run

Returns (Job) The corresponding job in a job class instance

determine_jobs_and_machines()
Reads the pyconf jobs definition and instantiates all the machines and jobs to be done today.

Returns None

display_status(*len_col*)
Takes a lenght and construct the display of the current status of the jobs in an array that has a column for each host. It displays the job that is currently running on the host of the column.

Parameters **len_col** – (int) the size of the column

Returns None

find_job_that_has_name(*name*)
Returns the job by its name.

Parameters **name** – (str) a job name

Returns (Job) the job that has the name.

is_occupied(*hostname*)
Returns True if a job is running on the machine defined by its host and its port.

Parameters `hostname` – (str, int) the pair (host, port)

Returns (Job or bool) the job that is running on the host, or false if there is no job running on the host.

run_jobs()

The main method. Runs all the jobs on every host. For each host, at a given time, only one job can be running. The jobs that have the field after (that contain the job that has to be run before it) are run after the previous job. This method stops when all the jobs are finished.

Returns None

ssh_connection_all_machines(*pad=50*)

Do the ssh connection to every machine to be used today.

Returns None

str_of_length(*text*, *length*)

Takes a string text of any length and returns the most close string of length "length".

Parameters

- `text` – (str) any string
- `length` – (int) a length for the returned string

Returns (str) the most close string of length "length"

update_jobs_states_list()

Updates the lists that store the currently running jobs and the jobs that have already finished.

Returns None

write_all_results()

Display all the jobs outputs.

Returns None

class commands.jobs.**Machine**(*name*, *host*, *user*, *port=22*, *passwd=None*, *sat_path='salomeTools'*)

Bases: `object`

Manage a ssh connection on a machine

close()

Close the ssh connection

connect(*logger*)

Initiate the ssh connection to the remote machine

Parameters `logger` – The logger instance

Returns None

copy_sat(*sat_local_path*, *job_file*)

Copy salomeTools to the remote machine in self.sat_path

exec_command(*command*, *logger*)

Execute the command on the remote machine

Parameters

- `command` – (str) The command to be run
- `logger` – The logger instance

Returns (paramiko.channel.ChannelFile, etc) the stdin, stdout, and stderr of the executing command, as a 3-tuple

mkdir(*path*, *mode=511*, *ignore_existing=False*)

As mkdir by adding an option to not fail if the folder exists

put_dir(*source*, *target*, *filters=[]*)
Uploads the contents of the source directory to the target path. The target directory needs to exists. All sub-directories in source are created under target.

successfully_connected(*logger*)
Verify if the connection to the remote machine has succeed

Parameters **logger** – The logger instance

Returns (bool) True if the connection has succeed, False if not

write_info(*logger*)
Prints the informations relative to the machine in the logger (terminal traces and log file)

Parameters **logger** – The logger instance

Returns None

`commands.jobs.`**develop_factorized_jobs**(*config_jobs*)
update information about the jobs for the file xml_file

Parameters **config_jobs** – (Config) the config corresponding to the jos description

`commands.jobs.`**getParamiko**(*logger=None*)

`commands.jobs.`**get_config_file_path**(*job_config_name*, *l_cfg_dir*)

commands.launcher module

class `commands.launcher.`**Command**(*runner*)
Bases: `src.salomeTools._BaseCommand`

The launcher command generates a SALOME launcher.

Examples:
>> sat launcher SALOME

getParser()
Define all possible options for command 'sat launcher <options>'

name = 'launcher'

run(*cmd_arguments*)
method called for command 'sat launcher <options>'

`commands.launcher.`**copy_catalog**(*config*, *catalog_path*)
Copy the xml catalog file into the right location

Parameters

- **config** – (Config) The global configuration
- **catalog_path** – (str) the catalog file path

Returns (dict) The environment dictionary corresponding to the file path.

`commands.launcher.`**generate_catalog**(*machines*, *config*, *logger*)
Generates an xml catalog file from a list of machines.

Parameters

- **machines** – (list) The list of machines to add in the catalog
- **config** – (Config) The global configuration
- **logger** – (Logger) The logger instance to use for the display and logging

Returns (str) The catalog file path.

`commands.launcher.`**`generate_launch_file`**(*config*, *logger*, *launcher_name*, *pathlauncher*, *display=True*, *additional_env={}*)

Generates the launcher file.

Parameters

- **`config`** – (Config) The global configuration
- **`logger`** – (Logger) The logger instance to use for the display and logging
- **`launcher_name`** – (str) The name of the launcher to generate
- **`pathlauncher`** – (str) The path to the launcher to generate
- **`display`** – (bool) If False, do not print anything in the terminal
- **`additional_env`** – (dict) The dict giving additional environment variables

Returns (str) The launcher file path.

commands.log module

`class` `commands.log.`**`Command`**(*runner*)

Bases: `src.salomeTools._BaseCommand`

The log command gives access to the logs produced by the salomeTools commands.

Examples:
>> sat log

`getParser`()

Define all options for command 'sat log <options>'

`name = 'log'`

`run`(*cmd_arguments*)

method called for command 'sat log <options>'

`commands.log.`**`ask_value`**(*nb*)

Ask for an int n. 0<n<nb

Parameters **`nb`** – (int) The maximum value of the value to be returned by the user.

Returns (int) the value entered by the user. Return -1 if it is not as expected

`commands.log.`**`getMaxFormat`**(*aListOfStr*, *offset=1*)

returns format for columns width as '%-30s"' for example

`commands.log.`**`get_last_log_file`**(*logDir*, *notShownCommands*)

Used in case of last option. Get the last log command file path.

Parameters

- **`logDir`** – (str) The directory where to search the log files
- **`notShownCommands`** – (list) the list of commands to ignore

Returns (str) the path to the last log file

`commands.log.`**`print_log_command_in_terminal`**(*filePath*, *logger*)

Print the contain of filePath. It contains a command log in xml format.

Parameters

- **filePath** – The command xml file from which extract the commands context and traces
- **logger** – (Logger) the logging instance to use in order to print.

`commands.log.`**`remove_log_file`**(*filePath*, *logger*)
if it exists, print a warning and remove the input file

Parameters

- **filePath** – the path of the file to delete
- **logger** – (Logger) the logger instance to use for the print

`commands.log.`**`show_last_logs`**(*logger*, *config*, *log_dirs*)
Show last compilation logs

`commands.log.`**`show_product_last_logs`**(*logger*, *config*, *product_log_dir*)
Show last compilation logs of a product

commands.make module

class `commands.make.`**`Command`**(*runner*)
Bases: `src.salomeTools._BaseCommand`

The make command executes the 'make' command in the build directory.

Examples:
>> sat make SALOME –products Python,KERNEL,GUI

`getParser`()
Define all options for the command 'sat make <options>'

`name = 'make'`

`run`(*cmd_arguments*)
method called for command 'sat make <options>'

`commands.make.`**`get_nb_proc`**(*product_info*, *config*, *make_option*)

`commands.make.`**`get_products_list`**(*options*, *cfg*, *logger*)
method that gives the product list with their informations from configuration regarding the passed options.

Parameters

- **options** – (Options) The Options instance that stores the commands arguments
- **cfg** – (Config) The global configuration
- **logger** – (Logger) The logger instance to use for the display and logging

Returns (list) The list of tuples (product name, product_informations).

`commands.make.`**`make_all_products`**(*config*, *products_infos*, *make_option*, *logger*)
Execute the proper configuration commands in each product build directory.

Parameters

- **config** – (Config) The global configuration
- **products_info** – (list) List of (str, Config) => (product_name, product_info)
- **make_option** – (str) The options to add to the command
- **logger** – (Logger) The logger instance to use for the display and logging

Returns (int) the number of failing commands.

`commands.make.`**`make_product`**`(`*p_name_info*, *make_option*, *config*, *logger*`)`
Execute the proper configuration command(s) in the product build directory.

Parameters

- **`p_name_info`** – (tuple) (str, Config) => (product_name, product_info)
- **`make_option`** – (str) The options to add to the command
- **`config`** – (Config) The global configuration
- **`logger`** – (Logger) The logger instance to use for the display and logging

Returns (int) 1 if it fails, else 0.

commands.makeinstall module

`class` `commands.makeinstall.`**`Command`**`(`*runner*`)`
Bases: `src.salomeTools._BaseCommand`

The makeinstall command executes the 'make install' command in the build directory. In case of product constructed using a script (build_source : 'script'), then the makeinstall command do nothing.

Examples:
>> sat makeinstall SALOME –products KERNEL,GUI

`getParser``()`
Define all options for the command 'sat makeinstall <options>'

`name = 'makeinstall'`

`run``(`*cmd_arguments*`)`
method called for command 'sat makeinstall <options>'

`commands.makeinstall.`**`get_products_list`**`(`*options*, *cfg*, *logger*`)`
method that gives the product list with their informations from configuration regarding the passed options.

Parameters

- **`options`** – (Options) The Options instance that stores the commands arguments
- **`cfg`** – (Config) The global configuration
- **`logger`** – (Logger) The logger instance to use for the display and logging

Returns (list) The list of (product name, product_informations).

`commands.makeinstall.`**`makeinstall_all_products`**`(`*config*, *products_infos*, *logger*`)`
Execute the proper configuration commands in each product build directory.

Parameters

- **`config`** – (Config) The global configuration
- **`products_info`** – (list) List of (str, Config) => (product_name, product_info)
- **`logger`** – (Logger) The logger instance to use for the display and logging

Returns (int) the number of failing commands.

`commands.makeinstall.`**`makeinstall_product`**`(`*p_name_info*, *config*, *logger*`)`
Execute the proper configuration command(s) in the product build directory.

Parameters

- **`p_name_info`** – (tuple) (str, Config) => (product_name, product_info)
- **`config`** – (Config) The global configuration

- **logger** – (Logger) The logger instance to use for the display and logging

Returns (int) 1 if it fails, else 0.

commands.package module

class `commands.package.`**`Command`**(*runner*)

Bases: `src.salomeTools._BaseCommand`

The package command creates an archive.

There are 4 kinds of archive, which can be mixed:

1- The binary archive. It contains all the product installation directories and a launcher.

2- The sources archive. It contains the products archives, a project corresponding to the application and salomeTools.

3- The project archive. It contains a project (give the project file path as argument).

4- The salomeTools archive. It contains salomeTools.

Examples:

>> sat package SALOME –binaries –sources

`getParser`()

Define all options for command 'sat package <options>'

`name = 'package'`

`run`(*cmd_arguments*)

method called for command 'sat package <options>'

`commands.package.`**`add_files`**(*tar*, *name_archive*, *d_content*, *logger*, *f_exclude=None*)

Create an archive containing all directories and files that are given in the d_content argument.

Parameters

- **tar** – (tarfile) The tarfile instance used to make the archive.
- **name_archive** – (str) The name of the archive to make.
- **d_content** – (dict) The dictionary that contain all directories and files to add in the archive. d_content[label] = (path_on_local_machine, path_in_archive)
- **logger** – (Logger) the logging instance
- **f_exclude** – (function) the function that filters

Returns (int) 0 if success, 1 if not.

`commands.package.`**`add_readme`**(*config*, *options*, *where*)

`commands.package.`**`add_salomeTools`**(*config*, *tmp_working_dir*)

Prepare a version of salomeTools that has a specific local.pyconf file configured for a source package.

Parameters

- **config** – (Config) The global configuration.
- **tmp_working_dir** – (str) The temporary local directory containing some specific directories or files needed in the source package

Returns (str) The path to the local salomeTools directory to add in the package

`commands.package.`**`binary_package`**(*config*, *logger*, *options*, *tmp_working_dir*)

Prepare a dictionary that stores all the needed directories and files to add in a binary package.

Parameters

- **config** – (Config) The global configuration.
- **logger** – (Logger) the logging instance
- **options** – (OptResult) the options of the launched command
- **tmp_working_dir** – (str) The temporary local directory containing some specific directories or files needed in the binary package

Returns (dict) The dictionary that stores all the needed directories and files to add in a binary package. {label : (path_on_local_machine, path_in_archive)}

`commands.package.`**`create_project_for_src_package`**`(`*config*, *tmp_working_dir*, *with_vcs*`)`

Create a specific project for a source package.

Parameters

- **config** – (Config) The global configuration.
- **tmp_working_dir** – (str) The temporary local directory containing some specific directories or files needed in the source package
- **with_vcs** – (bool) True if the package is with vcs products (not transformed into archive products)

Returns (dict) The dictionary {“project” : (produced project, project path in the archive)}

`commands.package.`**`exclude_VCS_and_extensions`**`(`*filename*`)`

The function that is used to exclude from package the link to the VCS repositories (like .git)

Parameters **filename** – (str) The filname to exclude (or not).

Returns (bool) True if the file has to be exclude

`commands.package.`**`find_application_pyconf`**`(`*config*, *application_tmp_dir*`)`

Find the application pyconf file and put it in the specific temporary directory containing the specific project of a source package.

Parameters

- **config** – ‘Config) The global configuration.
- **application_tmp_dir** – (str) The path to the temporary application scripts directory of the project.

`commands.package.`**`find_product_scripts_and_pyconf`**`(`*p_name*, *p_info*, *config*, *with_vcs*, *compil_scripts_tmp_dir*, *env_scripts_tmp_dir*, *patches_tmp_dir*, *products_pyconf_tmp_dir*`)`

Create a specific pyconf file for a given product. Get its environment script, its compilation script and patches and put it in the temporary working directory. This method is used in the source package in order to construct the specific project.

Parameters

- **p_name** – (str) The name of the product.
- **p_info** – (Config) The specific configuration corresponding to the product
- **config** – (Config) The global configuration.
- **with_vcs** – (bool) True if the package is with vcs products (not transformed into archive products)
- **compil_scripts_tmp_dir** – (str) The path to the temporary compilation scripts directory of the project.

- **env_scripts_tmp_dir** – (str) The path to the temporary environment script directory of the project.
- **patches_tmp_dir** – (str) The path to the temporary patch scripts directory of the project.
- **products_pyconf_tmp_dir** – (str) The path to the temporary product scripts directory of the project.

`commands.package.`**`get_archives`**(*config*, *logger*)

Find all the products from an archive and all the products from a VCS (git, cvs, svn) repository.

Parameters

- **config** – (Config) The global configuration.
- **logger** – (Logger) The logging instance

Returns (Dict, List) The dictionary {name_product : (local path of its archive, path in the package of its archive)} and the list of specific configuration corresponding to the vcs products

`commands.package.`**`get_archives_vcs`**(*l_pinfo_vcs*, *sat*, *config*, *logger*, *tmp_working_dir*)

For sources package that require that all products from an archive, one has to create some archive for the vcs products. So this method calls the clean and source command of sat and then create the archives.

Parameters

- **l_pinfo_vcs** – (list) The list of specific configuration corresponding to each vcs product
- **sat** – (Sat) The Sat instance that can be called to clean and source the products
- **config** – (Config) The global configuration.
- **logger** – (Logger) The logging instance
- **tmp_working_dir** – (str) The temporary local directory containing some specific directories or files needed in the source package

Returns (dict) The dictionary that stores all the archives to add in the sourcepackage. {label : (path_on_local_machine, path_in_archive)}

`commands.package.`**`hack_for_distene_licence`**(*filepath*)

Replace the distene licence env variable by a call to a file.

Parameters **filepath** – (str) The path to the launcher to modify.

`commands.package.`**`make_archive`**(*prod_name*, *prod_info*, *where*)

Create an archive of a product by searching its source directory.

Parameters

- **prod_name** – (str) The name of the product.
- **prod_info** – (Config) The specific configuration corresponding to the product
- **where** – (str) The path of the repository where to put the resulting archive

Returns (str) The path of the resulting archive

`commands.package.`**`produce_install_bin_file`**(*config*, *logger*, *file_dir*, *d_sub*, *file_name*)

Create a bash shell script which do substitutions in BIRARIES dir in order to use it for extra compilations.

Parameters

- **config** – (Config) The global configuration.
- **logger** – (Logger) the logging instance
- **file_dir** – (str) the directory where to put the files
- **d_sub** – (dict) the dictionnary that contains the substitutions to be done

- **file_name** – (str) the name of the install script file

Returns (str) the produced file

`commands.package.`**`produce_relative_env_files`**(*config*, *logger*, *file_dir*, *binaries_dir_name*)

Create some specific environment files for the binary package. These files use relative paths.

Parameters

- **config** – (Config) The global configuration.
- **logger** – (Logger) the logging instance
- **file_dir** – (str) the directory where to put the files
- **binaries_dir_name** – (str) The name of the repository where the binaries are, in the archive.

Returns (list) The list of path of the produced environment files

`commands.package.`**`produce_relative_launcher`**(*config*, *logger*, *file_dir*, *file_name*, *binaries_dir_name*, *with_commercial=True*)

Create a specific SALOME launcher for the binary package. This launcher uses relative paths.

Parameters

- **config** – (Config) The global configuration.
- **logger** – (Logger) the logging instance
- **file_dir** – (str) the directory where to put the launcher
- **file_name** – (str) The launcher name
- **binaries_dir_name** – (str) the name of the repository where the binaries are, in the archive.

Returns (str) the path of the produced launcher

`commands.package.`**`product_appli_creation_script`**(*config*, *logger*, *file_dir*, *binaries_dir_name*)

Create a script that can produce an application (EDF style) in the binary package.

Parameters

- **config** – (Config) The global configuration.
- **logger** – (Logger) the logging instance
- **file_dir** – (str) the directory where to put the file
- **binaries_dir_name** – (str) The name of the repository where the binaries are, in the archive.

Returns (str) The path of the produced script file

`commands.package.`**`project_package`**(*project_file_path*, *tmp_working_dir*)

Prepare a dictionary that stores all the needed directories and files to add in a project package.

Parameters

- **project_file_path** – (str) The path to the local project.
- **tmp_working_dir** – (str) The temporary local directory containing some specific directories or files needed in the project package

Returns (dict) The dictionary that stores all the needed directories and files to add in a project package. {label : (path_on_local_machine, path_in_archive)}

`commands.package.`**`source_package`**(*sat*, *config*, *logger*, *options*, *tmp_working_dir*)

Prepare a dictionary that stores all the needed directories and files to add in a source package.

Parameters

- **config** – (Config) The global configuration.
- **logger** – (Logger) the logging instance
- **options** – (OptResult) the options of the launched command
- **tmp_working_dir** – (str) The temporary local directory containing some specific directories or files needed in the binary package

Returns (dict) the dictionary that stores all the needed directories and files to add in a source package. {label : (path_on_local_machine, path_in_archive)}

commands.package.**update_config**(*config*, *prop*, *value*)

Remove from config.APPLICATION.products the products that have the property given as input.

Parameters

- **config** – (Config) The global config.
- **prop** – (str) The property to filter
- **value** – (str) The value of the property to filter

commands.patch module

class commands.patch.**Command**(*runner*)

Bases: src.salomeTools._BaseCommand

The patch command apply the patches on the sources of the application products if there is any.

Examples:
>> sat patch SALOME –products qt,boost

getParser()

Define all options for command 'sat patch <options>'

name = 'patch'

run(*cmd_arguments*)

method called for command 'sat patch <options>'

commands.patch.**apply_patch**(*config*, *product_info*, *max_product_name_len*, *logger*)

The method called to apply patches on a product

Parameters

- **config** – (Config) The global configuration
- **product_info** – (Config) The configuration specific to the product to be patched
- **logger** – (Logger: The logger instance to use for the display and logging

Returns (RCO.ReturnCode)

commands.prepare module

class commands.prepare.**Command**(*runner*)

Bases: src.salomeTools._BaseCommand

The prepare command gets the sources of the application products and apply the patches if there is any.

examples:
>> sat prepare SALOME –products KERNEL,GUI

getParser()
Define all options for command 'sat prepare <options>'

name = 'prepare'

run(*cmd_arguments*)
method called for command 'sat prepare <options>'

commands.prepare.**find_products_already_getted**(*l_products*)
Returns the list of products that have an existing source directory.

Parameters **l_products** – (list) The list of products to check

Returns (list) The list of product configurations that have an existing source directory.

commands.prepare.**find_products_with_patchs**(*l_products*)
Returns the list of products that have one or more patches.

Parameters **l_products** – (list) The list of products to check

Returns (list) The list of product configurations that have one or more patches.

commands.prepare.**remove_products**(*arguments*, *l_products_info*, *logger*)
Removes the products in l_products_info from arguments list.

Parameters

- **arguments** – (str) The arguments from which to remove products
- **l_products_info** – (list) List of (str, Config) => (product_name, product_info)
- **logger** – (Logger) The logger instance to use for the display and logging

Returns (str) The updated arguments.

commands.profile module

class commands.profile.**Command**(*runner*)
Bases: src.salomeTools._BaseCommand

The profile command creates default profile.

Examples:
>> sat profile [PRODUCT]
>> sat profile –prefix (string)
>> sat profile –name (string)
>> sat profile –force
>> sat profile –version (string)
>> sat profile –slogan (string)

getParser()
Define all options for command 'sat profile <options>'

name = 'profile'

run(*cmd_arguments*)
method called for command 'sat profile <options>'

commands.profile.**generate_profile_sources**(*config*, *options*, *logger*)
Generates the sources of the profile

commands.profile.**get_profile_name**(*options*, *config*)

class `commands.profile.`**`profileConfigReader`**(*config*)

Bases: *`src.pyconf.ConfigReader`* (page 59)

`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.

class `commands.profile.`**`profileReference`**(*config*, *type*, *ident*)

Bases: *`src.pyconf.Reference`* (page 62)

`commands.profile.`**`update_pyconf`**(*config*, *options*, *logger*)

Updates the pyconf

commands.run module

class `commands.run.`**`Command`**(*runner*)

Bases: `src.salomeTools._BaseCommand`

The run command runs the application launcher with the given arguments.

Examples:

>> sat run SALOME

`getParser`()

Define all options for command 'sat run <options>'

`name = 'run'`

`run`(*cmd_arguments*)

method called for command 'sat run <options>'

commands.script module

class `commands.script.`**`Command`**(*runner*)

Bases: `src.salomeTools._BaseCommand`

The script command executes the script(s) of the the given products in the build directory. This is done only for the products that are constructed using a script (build_source : 'script'). Otherwise, nothing is done.

Examples:

>> sat script SALOME –products Python,numpy

`getParser`()

Define all options for the command 'sat script <options>'

`name = 'script'`

`run`(*cmd_arguments*)

method called for command 'sat script <options>'

`commands.script.`**`get_products_list`**(*options*, *cfg*, *logger*)

Gives the product list with their informations from configuration regarding the passed options.

Parameters

- **`options`** – (Options) The Options instance that stores the commands arguments

- **cfg** – (Config) The global configuration
- **logger** – (Logger) The logger instance to use for the display and logging

Returns (list) The list of (product name, product_informations).

`commands.script.`**`run_script_all_products`**(*config*, *products_infos*, *nb_proc*, *logger*)

Execute the script in each product build directory.

Parameters

- **config** – (Config) The global configuration
- **products_info** – (list) List of (str, Config) => (product_name, product_info)
- **nb_proc** – (int) The number of processors to use
- **logger** – (Logger) The logger instance to use for the display and logging

Returns (int) The number of failing commands.

`commands.script.`**`run_script_of_product`**(*p_name_info*, *nb_proc*, *config*, *logger*)

Execute the proper configuration command(s) in the product build directory.

Parameters

- **p_name_info** – (tuple) (str, Config) => (product_name, product_info)
- **nb_proc** – (int) The number of processors to use
- **config** – (Config) The global configuration
- **logger** – (Logger) The logger instance to use for the display and logging

Returns (int) 1 if it fails, else 0.

commands.shell module

class `commands.shell.`**`Command`**(*runner*)

Bases: `src.salomeTools._BaseCommand`

The shell command executes the shell command passed as argument.

Examples:
>> sat shell –command 'ls -lt /tmp'

getParser()

Define all options for the command 'sat shell <options>'

name = 'shell'

run(*cmd_arguments*)

method called for command 'sat shell <options>'

commands.source module

class `commands.source.`**`Command`**(*runner*)

Bases: `src.salomeTools._BaseCommand`

The source command gets the sources of the application products from cvs, git or an archive.

Examples:
>> sat source SALOME –products KERNEL,GUI

getParser()
Define all options for command 'sat source <options>'

name = 'source'

run(*cmd_arguments*)
method called for command 'sat source <options>'

`commands.source.`**check_sources**(*product_info*, *logger*)
Check that the sources are correctly get, using the files to be tested in product information

Parameters

- **product_info** – (Config) The configuration specific to the product to be prepared
- **logger** – (Logger) The logger instance to be used for the logging

Returns (bool) True if the files exists (or no files to test is provided).

`commands.source.`**get_all_product_sources**(*config*, *products*, *logger*)
Get all the product sources.

Parameters

- **config** – (Config) The global configuration
- **products** – (list) The list of tuples (product name, product informations)
- **logger** – (Logger) The logger instance to be used for the logging

Returns (int,dict) The tuple (number of success, dictionary product_name/success_fail)

`commands.source.`**get_product_sources**(*config*, *product_info*, *is_dev*, *source_dir*, *logger*, *pad*, *checkout=False*)
Get the product sources.

Parameters

- **config** – (Config) The global configuration
- **product_info** – (Config) The configuration specific to the product to be prepared
- **is_dev** – (bool) True if the product is in development mode
- **source_dir** – (Path) The Path instance corresponding to the directory where to put the sources
- **logger** – (Logger) The logger instance to use for the display and logging
- **pad** – (int) The gap to apply for the terminal display
- **checkout** – (bool) If True, get the source in checkout mode

Returns (bool) True if it succeed, else False

`commands.source.`**get_source_for_dev**(*config*, *product_info*, *source_dir*, *logger*, *pad*)
Called if the product is in development mode

Parameters

- **config** – (Config) The global configuration
- **product_info** – (Config) The configuration specific to the product to be prepared
- **source_dir** – (Path) The Path instance corresponding to the directory where to put the sources
- **logger** – (Logger) The logger instance to use for the display and logging
- **pad** – (int) The gap to apply for the terminal display

Returns (bool) True if it succeed, else False `commands.source.`**`get_source_from_archive`**(*product_info*, *source_dir*, *logger*)

The method called if the product is to be get in archive mode

Parameters

- **`product_info`** – (Config) The configuration specific to the product to be prepared
- **`source_dir`** – (Path) The Path instance corresponding to the directory where to put the sources
- **`logger`** – (Logger) The logger instance to use for the display and logging

Returns (bool) True if it succeed, else False

`commands.source.`**`get_source_from_cvs`**(*user*, *product_info*, *source_dir*, *checkout*, *logger*, *pad*, *environ=None*)

The method called if the product is to be get in cvs mode

Parameters

- **`user`** – (str) The user to use in for the cvs command
- **`product_info`** – (Config) The configuration specific to the product to be prepared
- **`source_dir`** – (Path) The Path instance corresponding to the directory where to put the sources
- **`checkout`** – (bool) If True, get the source in checkout mode
- **`logger`** – (Logger) The logger instance to use for the display and logging
- **`pad`** – (int) The gap to apply for the terminal display
- **`environ`** – (src.environment.Environ) The environment to source when extracting.

Returns (bool) True if it succeed, else False

`commands.source.`**`get_source_from_dir`**(*product_info*, *source_dir*, *logger*)

`commands.source.`**`get_source_from_git`**(*product_info*, *source_dir*, *logger*, *pad*, *is_dev=False*, *environ=None*)

Called if the product is to be get in git mode

Parameters

- **`product_info`** – (Config) The configuration specific to the product to be prepared
- **`source_dir`** – (Path) The Path instance corresponding to the directory where to put the sources
- **`Logger`** (*`logger`*) – (Logger) The logger instance to use for the display and logging
- **`pad`** – (int) The gap to apply for the terminal display
- **`is_dev`** – (bool) True if the product is in development mode
- **`environ`** – (src.environment.Environ) The environment to source when extracting.

Returns (bool) True if it succeed, else False

`commands.source.`**`get_source_from_svn`**(*user*, *product_info*, *source_dir*, *checkout*, *logger*, *environ=None*)

The method called if the product is to be get in svn mode

Parameters

- **`user`** – (str) The user to use in for the svn command
- **`product_info`** – (Config) The configuration specific to the product to be prepared
- **`source_dir`** – (Path) The Path instance corresponding to the directory where to put the sources
- **`checkout`** – (boolean) If True, get the source in checkout mode

- **logger** – (Logger) The logger instance to use for the display and logging
- **environ** – (src.environment.Environ) The environment to source when extracting.

Returns (bool) True if it succeed, else False

commands.template module

class commands.template.**Command**(*runner*)

Bases: src.salomeTools._BaseCommand

The template command creates the sources for a SALOME module from a template.

Examples:
>> sat template –name my_product_name –template PythonComponent –target /tmp

getParser()

Define all options for command 'sat template <options>'

name = 'template'

run(*cmd_arguments*)

method called for command 'sat template <options>'

class commands.template.**TParam**(*param_def*, *compo_name*, *dico=None*)

check_value(*val*)

class commands.template.**TemplateSettings**(*compo_name*, *settings_file*, *target*)

check_file_for_substitution(*file_*)

check_user_values(*values*)

get_parameters(*conf_values=None*)

get_pyconf_parameters()

has_pyconf()

commands.template.**get_dico_param**(*dico*, *key*, *default*)

commands.template.**get_template_info**(*config*, *template_name*, *logger*)

commands.template.**prepare_from_template**(*config*, *name*, *template*, *target_dir*, *conf_values*, *logger*)

Prepares a module from a template.

commands.template.**search_template**(*config*, *template*)

commands.test module

class commands.test.**Command**(*runner*)

Bases: src.salomeTools._BaseCommand

The test command runs a test base on a SALOME installation.

Examples:
>> sat test SALOME –grid GEOM –session light

check_option(*options*)
Check the options

Parameters **options** – (Options) The options

Returns None

getParser()
Define all options for command 'sat test <options>'

name = 'test'

run(*cmd_arguments*)
method called for command 'sat test <options>'

commands.test.**ask_a_path**()
interactive as using 'raw_input'

commands.test.**check_remote_machine**(*machine_name*, *logger*)

commands.test.**create_test_report**(*config*, *xml_history_path*, *dest_path*, *retcode*, *xmlname=''*)
Creates the XML report for a product.

commands.test.**generate_history_xml_path**(*config*, *test_base*)
Generate the name of the xml file that contain the history of the tests on the machine with the current APPLICATION and the current test base.

Parameters

- **config** – (Config) The global configuration
- **test_base** – (str) The test base name (or path)

Returns (str) the full path of the history xml file

commands.test.**move_test_results**(*in_dir*, *what*, *out_dir*, *logger*)

commands.test.**save_file**(*filename*, *base*)

Module contents

CHAPTER

FIVE

RELEASE NOTES

5.1 Release notes

In construction.

PYTHON MODULE INDEX

INDEX