Rossum User Manual

Table of Contents

This document is the user-manual for Rossum Test Framework.

1. General Package Information

1.1. Release Information

Table 1. Release Information

Date	19-Jan-2021
Version	1.0.0
Status	Stable
Notes	First release

2. Rossum Installation

2.1. System requirements

OS	Linux (Ubuntu 16.04, Ubuntu 20.04), Windows 10
Python	3.6 or above

2.2. For Linux

2.2.1. Installation

Un-tar the package in some path, say `/home/<username>/rossum`

Install the package:

```
$ /home/<username>/rossum/set_vars.sh install
```

2.2.2. Un-Installation

Remove changes made by `set_vars.sh install`:

```
$ /home/<username>/rossum/set_vars.sh uninstall
```

Then, the folder can be deleted:

```
$ rm -rf /home/<username>/rossum
```

3. Introduction

Rossum Test Framework was initially developed in 2018 and it's released under Vayavya Permissive License (VPL). Rossum is extensible and can be integrated with virtually any other tool to create powerful and flexible automation solutions.

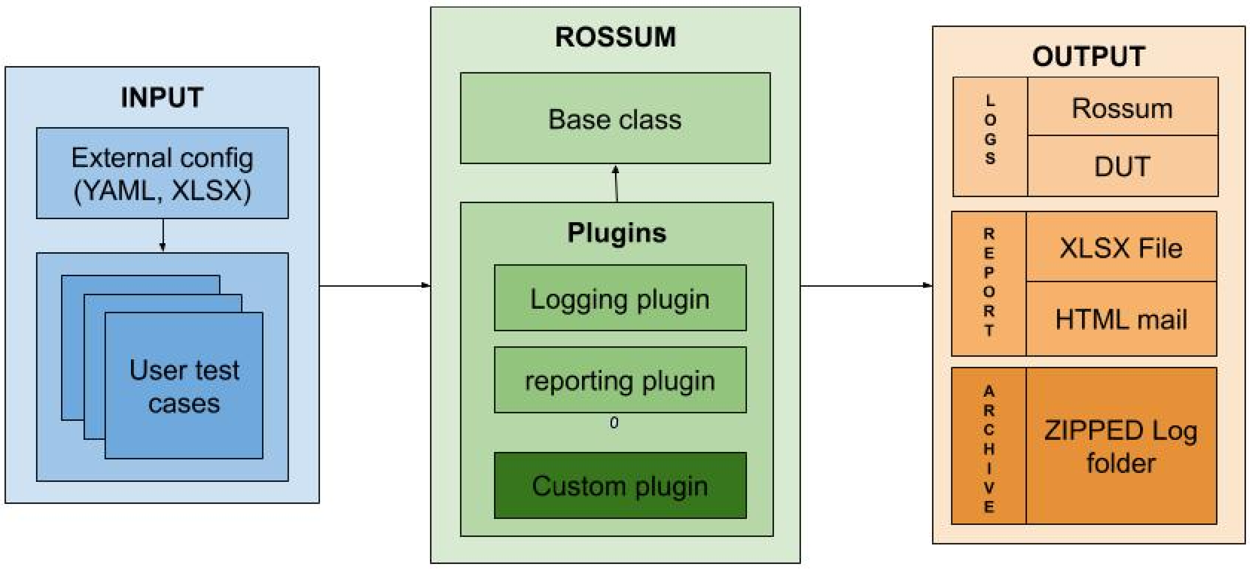
Its capabilities can be easily be extended via plugins and it already has plugins for logging, reporting, email notification, ssh, scpi and more.

3.1. Support

Support requests can be raised on the product's bugzilla page at: http://svn.vayavyalabs.com/bugzilla/enter_bug.cgi?product=rossum

4. Getting started

Rossum Test framework allows testcase writers easy way to interact with it's plugin objects.



4.1. User test-cases

- `run-rossum.py` imports `TestCase` class which has all functions required by test-case
- `test_case` has common `setup` and `teardown` for all test-cases.
- Functions are part of `setup`, `teardown` or `testcase` and they are identified by decorators assigned to them.

4.1.1. Example Test-case

```
# import all the functions from test_case
from test_case import *
```

```
# TestCase is inherited by the class
class my_broadcast_test(TestCase):

   #### Start of boiler plate code ####
    def __init__(self):

        """
        Init base class (makes tag list)
        """
        super().__init__()

   #### End of boiler plate code ####

   #### Start of test-case code ####

    """
    Setup decorator which takes care of default setup to be done.
    """
    @TestCase.setup
    def my_setup(self):

        self.log.info("print from my setup")

    """
    TestCase testgen decorator has logic to parse the user input
    excel file
    """
    @TestCase.testgen(path="Path of the excel file",sheet="Sheet name to be accessed")

    """
    TestCase test decorator has logic to pick test-case based on tags and user-input
    """
    @TestCase.test(tags=[["short_test", "broadcast"]], timeout="600", retry="2")
    def test_case_1(self):

    #### User-defined test code to be mentioned here ####

        self.log.info('print from test_case_1')

        """
        Default skip_teardown = False
        If skip_teardown = True, it will skip the
        teardown for current test and setup for next test
        """
        self.skip_teardown = False

        """
        Return value is based on whether test-case passes or fails
        """
        return (True)
    """
    Teardown decorator will take care of reseting the values created in setup
decorator
    """


    @TestCase.teardown
    def my_teardown(self):
        self.log.info("print from my teardown")

   #### End of test-case code ####
```

4.1.2. Example command line

```
run-rossum.py -tf <path_to_user_test> -l 1
```

4.1.3. Logs & Reports

- Logs for the test-case run would be created in current working directory with timestamp
- Each log folder is separately archieved into a zip file
- Excel report is generated at the end of test execution

5. Guides

5.1. Command line interface

```
usage: run-rossum.py [-h] [-v]
                     (-tf TESTFILE [TESTFILE ...] | -td TESTDIR [TESTDIR ...] | -af
ARGSFILE)
                     [-t TAG_LIST [TAG_LIST ...]] [-dt DEFAULT_TIMEOUT]
                     [-dr DEFAULT_RETRY] [-l LOG_TYPE]

required arguments:
  -tf TESTFILE [TESTFILE ...], --testfile TESTFILE [TESTFILE ...]
                        path to test case file or files
  -td TESTDIR [TESTDIR ...], --testdir TESTDIR [TESTDIR ...]
                        path to folder with test case
  -af ARGSFILE, --argsfile ARGSFILE
                        pass arguments using yaml file

optional arguments:
  -h, --help            show this help message and exit
  -v, --version         show program's version number and exit
  -t TAG_LIST [TAG_LIST ...], --tag TAG_LIST [TAG_LIST ...]
                        Only run test cases that match tag (default: all)
  -dt DEFAULT_TIMEOUT, --default_timeout DEFAULT_TIMEOUT
                        Default timeout for each testcase in secs (default: 10800)
  -dr DEFAULT_RETRY, --default_retry DEFAULT_RETRY
                        Default retry for each testcase (default: 2)
  -l LOG_TYPE, --log LOG_TYPE
                        Logging Levels:
                        0 - DEBUG
                        1 - INFO (Default)
                        2 - WARN
                        3 - ERROR
```

Mandatory args:

- `testfile` or `testdir`
    - `testfile`: Accepts one or more files with .py extension
    - `testdir`: Accepts one or more dirs; will pick files with .py extension
    - `argsfile`: Accepts one yaml file as input that can have all the arguments mentioned in it

Optional args:

- `tag`: Expects list of tags that match at least one tag in testcases, default values is `All` that matches all tags
- `default_timeout`: Timeout is time in secs after which testcase should exit. Timeout specified in testcase takes precedence over command line. `0` means no timeout (default: `10800` secs).
- `default_retry`: Retry is number of times testcase should retry if previous run time-outs. Retry value in testcase takes precedence over command line. '0' means no retry (default: `2` extra tries).
- `log`: Log level for log prints. Supported log levels are:

    - `0` - DEBUG
    - `1` - INFO (Default)
    - `2` - WARN
    - `3` - ERROR

5.2. Testcase class and decorated functions

5.2.1. Decorators

- Decorators allow you to inject or modify code in functions or classes.
- A function decorator is applied to a function definition by placing it on the line before that function definition begins
- Functions are part of `setup`,`testcase` or `teardown` and they are identified by decorators assigned to them
- `Setup` decorator is mandatory along with the setup function
- `Test` decorator is mandatory along with the test-case function
- `@TestCase.test` decorator is the only decorator that takes arguments:
    - `tag`: argument has a list which contains tags supported by test-case. User should pass tag to `run-rossum.py` such that it matches `tag` param of the decorator for the test-case they wish to run.
    - `timeout`: argument takes integer; which is number of seconds after which testcase should timeout
    - `retry`: argument takes integer; which is number of times testcase should retry on timeout
- `Teardown` decorator is mandatory along with the teardown function
- Some more information on decorator is available at https://www.artima.com/weblogs/viewpost.jsp?thread=240808

5.2.2. setup

- All the common code or environments settings to be done can be part of setup

5.2.3. test

- The complete test-case logic should be part of this function
- Each testfile can have multiple test-cases present and each test-cases can be individual or inter-related
- The return value at the end of each test-case should be either True/False, this would be the result of the test execution
- Logs for each test will created separately
- **Testcase result**:
    - **Pass**: When test-case function returns `True`

    - **Fail**: When test-case function returns `False`
    - **Abort**: When test-case or its setup raises following exception conditions:
        - Test-case raises an exception
        - Syntax error in the script
    - **Error**: When there is an exception before executing the test-case
    - **Timeout**: When test-case time-outs before there is an exception or before executing the test-case:

5.2.4. teardown

- Code to reset of values created in setup and test-case will be present here

5.2.5. testgen

- `@TestCase.testgen`: decorator takes two arguments
    - `path`: argument takes one parameter, i.e., the path to the excel file
    - `sheet`: argument takes one parameter, i.e., the sheet name to be accessed
    - `@TestCase.testgen` decorator should only decorate a function which is already decorated by `@TestCase.test` decorator

5.3. Extending Testcase class

5.3.1. Sample Base class

```
from test_case import *


class BaseClass(TestCase):

    def __init__(self):

        super().__init__()

    def log_init(self, base_log):

        res = super().log_init(base_log)
        return(res)

    def pre_setup(self):

        res = super().pre_setup()
        self.__result__ = [False]

        return (res)

    def post_teardown(self):

        res = super().post_teardown()

        return (res)

    def end_logs(self, tc_res):

        res = super().end_logs(tc_res)

        return(res)

    def test_extend(self, tc_fn_info):

        res = super().test_extend(tc_fn_info)
        return(res)

    def pre_retry(self, retry_counter):

        res = super().pre_retry(retry_counter)
        return(res)

    def testgen_logic(self, path=None, sheet=None):
        res = super().testgen_logic(path=path, sheet=sheet)
        return(res)

    '''
    User functions
    '''
```

5.3.2. Pre-setup

- This gets called before the `setup` function mentioned in the user test-case
- This is made available in the base class
- Any envirornment related settings that is required for execution of all test-cases can be added to this function

5.3.3. Extending testgen logic

- Each row in the input excel file is considered as a separate test-case
- `Test ID` and `Tags` are mandatory columns
- Additional columns can be added based on project requirements

5.3.4. Extending end_logs

- A call can directly be made to this function from the user test-case.
- Commands can be added to this function to copy any additional logs or files to be made available in the log folder

5.3.5. Post-teardown

- This gets called after thet `teardown` function mentioned in the user test-case
- This is made available in the base class
- Any values that needs to be reset can be done using this function

5.4. Helper functions and attributes

5.4.1. self.log

Below log messages will be printed to the screen and also get logged to a file based on the log level mentioned as part of the run-rossum command line input

- self.log.debug("Additional prints for debug")
- self.log.info("Something you might like to know")
- self.log.warn("Not an error but you should know this")
- self.log.error("Something failed")
- self.log.status("This is always printed")

5.4.2. add_to_report

- This function can be called in the user test-case or base class to insert additional columns and values in the excel report that will be generated at the end of test-run
- Below is the sample command to be used

```
self.add_to_report({"column_name": "value"})
```

5.4.3. Python-Constraint

- The Python constraint module offers solvers for Constraint Solving Problems (CSPs) over finite domains in simple and pure Python
- Constraints are relationships between variables or unknowns, each taking a value in a given domain: constraints restrict possible values that variables can take. Constraints need to be identified in order to solve problems by means of constraint solving techniques.

Example:

```
from constraint import *

>>> problem = Problem()
>>> problem.addVariable("a", [1,2,3])
>>> problem.addVariable("b", [4,5,6])
>>> problem.getSolutions()

#### Solution for the problem ####

[{'a': 3, 'b': 6}, {'a': 3, 'b': 5}, {'a': 3, 'b': 4},
 {'a': 2, 'b': 6}, {'a': 2, 'b': 5}, {'a': 2, 'b': 4},
 {'a': 1, 'b': 6}, {'a': 1, 'b': 5}, {'a': 1, 'b': 4}]

#### Adding a constraint to get better solutions ####

>>> problem.addConstraint(lambda a, b: a*2 == b, ("a", "b"))
>>> problem.getSolutions()
[{'a': 3, 'b': 6}, {'a': 2, 'b': 4}]
```

More information on python constraints is available at https://labix.org/python-constraint.

5.4.4. wait

- This function is used to add delay in the test-case execution

```
wait(time in seconds)
```

6. Rossum plugins:

6.1. Rossum plugins overview

- We use `pluggy` python module to add plugin support to Rossum, pluggy is a minimalist production ready plugin system which is used by pytest, tox and several other projects
- User can use `Rossum Plugin Template` to write Rossum plugin file and save it at rossum_core/plugins/user_plugins folder
- Rossum will use all plugins present in the rossum_core/plugins/core_plugins & rossum_core/plugins/user_plugins folder automatically.

6.2. How to write a plugin

Read comments in code to understand how plugin works.

```
# ##########################
# # Rossum Plugin Template
# ##########################

from plugin_module import hookimpl

class feature_plugin(object):
    '''
    Class name and class function names are all
    part of template and must not be changed.
    '''

    @hookimpl
    def service_start_hook(self, evars):
        '''
        User code in this function is executed in run_rossum.py
        before any testcases are even imported.

        Provides access to evars variable and expects user to return Bool
        '''
        # ## User code Start
        print("feature_cmd_prefix - Print from service_start_hook")
        return True
        # ## User code End

    @hookimpl
    def service_end_hook(self, evars, report):
        '''
        User code in this function is executed in run_rossum.py
        after all testcases execution ends.

        Provides access to evars & report variable and expects user to return Bool
        '''
        # ## User code Start
        print("feature_cmd_prefix - Print from service_end_hook")
        return True
        # ## User code End

    @hookimpl
    def argparse_hook(self, parser):
        '''
        User code in this function is executed in test_case.py baseclass.
        Plugin specific user arguments can be added here.

        Provides user parser object and expects user to return it after adding
arguments.
        '''
        # ## User code Start
        parser.add_argument(
            '--cmd-prefix', dest='cmd_prefix',
            choices=['echo', 'gdb', 'valgrind'],
            help='Cmd prefix for running DUT binary')
        # ## User code End

        return parser

    @hookimpl
    def pre_setup_hook(self, selfo):
        '''
        User code in this function is executed in each test case's pre_setup function

        Provides access to testcase object ie. selfo and expects user to return Bool
        '''
        # ## User code Start
        print("feature_cmd_prefix - Print from pre_setup hook")
        if selfo.evars.interact or selfo.evars.cmd_prefix:
            ret_val = selfo.debug_test_binary()
        else:
            ret_val = True

        return ret_val
        # ## User code End

    @hookimpl
    def post_teardown_hook(self, selfo):
        '''
        User code in this function is executed in each test case's port_teardown
function

        Provides access to testcase object ie. selfo and expects user to return Bool
        '''
        # ## User code Start
        print("feature_cmd_prefix - Print from post_teardown hook")
        return True
        # ## User code End
```

6.3. Existing plugin repo

6.3.1. excel_maker_plugin.py

- This plugin is used for creation of excel reports
- Using this plugin the user can create and format columns in the excel report

6.3.2. local_logging_plugin.py

- This plugin is used for creation of log files

6.3.3. print_summary_plugin.py

- This plugin is used for printing and logging the test-case execution summary
- This plugin gets executed at the end of the test-case run and it is being called in run-rossum.py