

IOT Test Automation Framework (ITAF) WALKTHROUGH

Revision 1.0

Ramya Ranganathan

Contact: Ramya.ranganathan@intel.com

INTRODUCTION.....	4
TWO MAJOR CATEGORIES OF TESTS	4
WHY Test Structure is needed.....	5
Test Framework Goals	5
WHY ITAF	5
ITAF Terminologies	6
ITAF Architecture Choices	6
1: Separation of configuration data from test case application code	6
2: Abstraction of test run code from test application code	6
3: Separate test logs/reports from test application code	7
4: Common test code utilities	7
5: Documented code	7
6: Embrace other test aid tools and test scope	7
7: Common Directory Structure for ITAF	7
ITAF Tool Choices.....	8
Python (http://python.org)	8
Selenium.....	8
Additional Libraries	8
ITAF : Test Execution Flow	9
ITAF COMPONENTS.....	9
iTAF Directory Structure	9
testCaseApps.....	12
testScenarios	12
utils.....	12
Robot Framework.....	12
TestSettings and SettingsInfo	13
Parameterization and Metrics	14
Logging.....	14
Documentation	15
DETAILED WALKTHROUGH.....	15
First Things First.....	15
Robot File	16
What Happens When Robot Starts Executing This File?.....	19

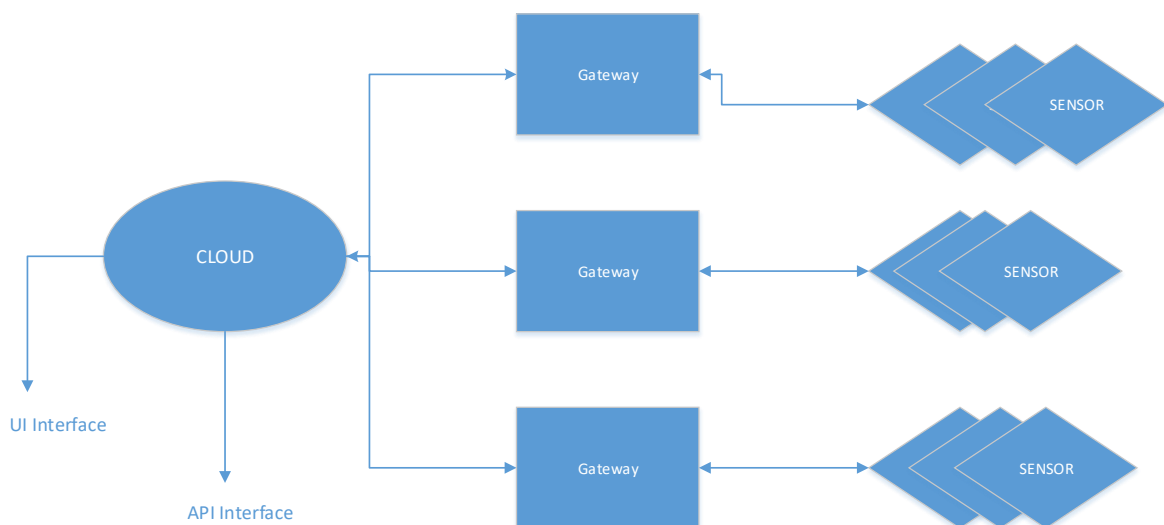
Robot “Cell-based” Parsing	20
The “import” Problem.....	21
Setup And Teardown	23
Robot Flow Of Execution.....	25
Keyword <i>SET CONFIG LOCATIONS</i>	25
TestSettings.....	29
What Is SettingsInfo?	30
Python Test Case File	32
Summary Of Walkthrough - Guidelines.....	35
DOCUMENTING THE CODE	36
Introduction	36
Creating Configuration File	36
Document the Code	37
Run the Doxygen	39

INTRODUCTION

IOT Test Automation Framework (ITAF) is an automation test framework for IOT product validation. It is especially useful for automating end-to-end UI and API tests of an IOT product. This document will walkthrough test code structuring and tool choices for ITAF.

TWO MAJOR CATEGORIES OF TESTS

The IOT products which ITAF supports usually have the following high-level architecture where there is a Sensor component generating the data, Gateway compiles and abstracts the data and sends interesting data to cloud where the customers can interpret and make insightful decisions. There is also a manageability / configurability flow that sources from the Cloud and gets propagated to GW and Sensors. For instance, over the air updates of the GW and Sensor SW happen from the Cloud.



From the cloud, there are usually two types of interfaces for customers to interact with; these are the UI and the API. There is also a third type of interface called the CLI

The UI interface is a human-driven interface that presents a set of web pages for humans to interact with the IOT system. In this case, iTAF testing would use the python selenium library to automate human-like interaction with the web pages.

The API interface is a programmatic interface intended so that customers can write applications to interact with the IOT system. In this case, iTAF would automate API calls.

CLI - Command Line Interface is for Personas with technical skillset who interact with the edge devices via cmd line console. Although current IOT solutions relies more on cloud based interfaces for remote edge management and UX, CLI does exist for obvious reasons.

Some products might require both or all 3 types of testing to be performed.

WHY Test Structure is needed

Structured test development which is modular and supports test reuse is crucial to get tests developed quickly and with less redundancy which in turn help organizations deliver better Software.

In addition to being modular, the structured test development approach establishes a boundary between test data and test scripts allowing to create test automation scripts by passing different sets of test data. This gives more test coverage with reusable tests and by changing only the input test data as a parameter.

Also since in many organizations, the test development team will be different than test execution team, the test script complexity need to be abstracted to the test executor wherein testers can work with keywords to develop any test scenario, testers with less programming knowledge would also be able to work on the test scripts.

Test Framework Goals

The following goals were taken into consideration while coming up with Test Framework

- ✓ Conformity among various IOT products
- ✓ Integration with automated CD system
- ✓ Separation of configuration data from test case application code
- ✓ Separation of test drivers from test application code
- ✓ Isolate test logs/reports from test application code
- ✓ Develop common test code utilities whenever possible
- ✓ Documented code

WHY ITAF

IOT Test Automation Framework (ITAF) establishes the above goals by creating a directory structure for test architecture. IOT Test Automation Framework (ITAF) is a generic validation framework designed for “any” IOT solution Validation.

- Architecture Adheres to IOT Use Case terminologies
- Addresses standard IOT use case scenarios
 - Commissioning / Deployment / Decommissioning
 - Initialization / Configuration/Registration
 - E2E Telemetry Flow / Health Monitoring / Security
- Factors in Metrics for IOT requirements
 - Robustness / Longevity/Scalability/ Reliability
 - Performance
 - Negative / Failure scenarios

ITAF Terminologies

ITAF Terminology	IOT Use Case Terminology	For Instance
TestApps	Use Case Types	Appliance Provisioning, Remote updates, Telemetry data flow etc..
Test App	Use Case E2E Transaction	OTA Kernel Upgrade
Test Case	Use Case Transaction break down of multiple Test Conditions (Action & Verify)	Click button / Verify Dialog box appears
Test Scenario	Use Case Scenario aka (customer scenario) under which a Test Condition can be applied	UI – click button when <ol style="list-style-type: none"> 1. server is down 2. during client system upgrade 3. When simultaneous multiple hits to the API
Test Scenario with Config	Use Case Scenario w/ different configurations	Vary Distance, Allowable Transaction Delay, Time Metrics, etc.

ITAF Architecture Choices

1: Separation of configuration data from test case application code

Test code if made configuration-aware enables diverse combination of tests with minimal test code. However it does require additional discipline in test planning and test code implementation to achieve maximum benefit.

2: Abstraction of test run code from test application code

Test Run code drives various configurations to be executed with same test code. Test Run Framework can produce nice test output like xml/html. Makes consistent test output possible. Test output can be made available to CD system. Email report links are made available to test stakeholders.

For the reasons, listed in ITAF Tool Choices section, the Test Run Tool for ITAF is Robot.

Copyright © Intel Corporation, 2019. All rights reserved. This work is licensed under a Creative Commons Attribution 4.0 International License.

3: Separate test logs/reports from test application code

testArtifacts contains subdirectories for framework (reports) and for test code (logs)

Test application code writes log output to common directory – testArtifacts/logs

Framework writes output to testArtifacts/reports

Easy integration with code repository – don't check-in testArtifacts files

Easy for CD system to serve up output for test stakeholders.

Challenge – Framework doesn't usually understand "performance" type output. Only deals with Pass/Fail type status. Solution – use test case logs to output performance metrics.

4: Common test code utilities

Many test cases do similar things, for instance logging into cloud, making API calls, parsing similar config files, reading data, verifying data. A test code that is generic for multiple people/projects to reuse is a great candidate to be stored in common utilities. This makes test case code cleaner and its benefits the entire team when generic code can be shared. Common utilities simplify test case code. Test case code is easier to review and understand. Utils library catalogued for easy viewing and porting.

In ITAF, Utils is where common utilities are stored.

5: Documented code

Documented code is better understood and easier to read for test stakeholders that do not directly write the test code. CD system can make test code documentation available to test stakeholders. This allows wider understanding of the testing so that stakeholders who do not write test code can participate in test development and test feedback. ITAF uses Doxygen for test documentation. Please refer to the ITAF Tool Choices for more details.

6: Embrace other test aid tools and test scope

ITAF does not limit itself to the tools that are chosen. The framework can embrace test tools/plugins that aid certain aspects of testing. For instance,

UI Test Automation tool eg : Selenium

Automated API test tools, e.g. Runscope, Postman, vRest, Jmeter etc

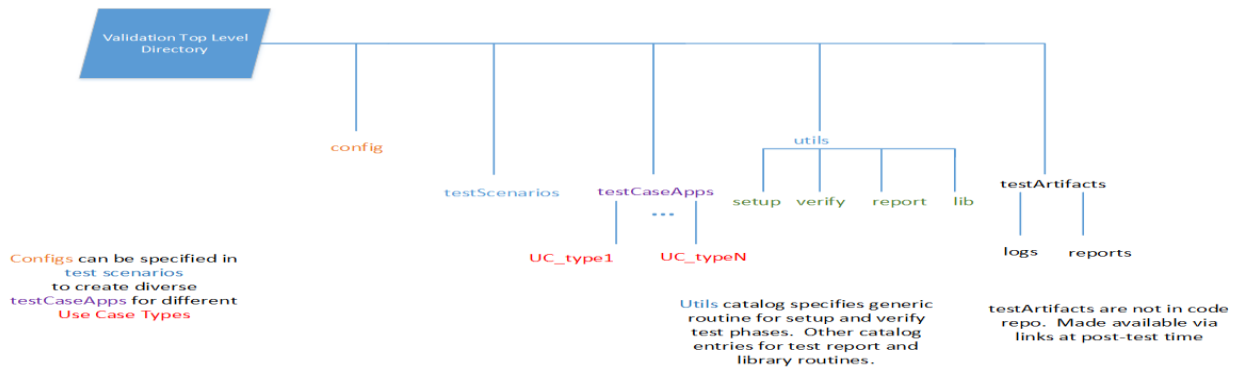
Simulators for Gateways/Sensors

Debug Hooks where applicable - Gateways / Cloud / Sensors

Diagnostics for Gateways / Sensors where applicable

7: Common Directory Structure for ITAF

A common directory structure allows easier start-up of new IOT products. It also allows common CD strategy



ITAF Tool Choices

Python (<http://python.org>)

Python has been chosen as the ITAF test scripting language as opposed to Java , for its simplicity and being open source and cross platform support. Although some claim that Python is slow since the code is executed by interpreter as opposed to a compiler, for validation needs Speed isn't a problem until it is a problem!

Robot Tool (<http://robotframework.org>)

Robot tool has been chosen as the test run framework for it being Open source, Very easy to install, platform independent and works well with Python. Also it supports Keyword driven, Data-driven and Behavior-driven (BDD) approaches. Its semantics are simple enough that User doesn't need a programming language to write a Robot Framework test case.

Doxygen (<http://doxygen.nl/>)

Doxygen is a documentation system for C++, C, Java, Python, Objective-C, IDL (Corba and Microsoft flavors) etc. It helps in generating on-line documentation or offline reference manual from documented source files. Extracting the Code Structure and visualising the relations between various elements by means of include dependency graphs, inheritance diagrams, and collaboration diagrams, which are all generated automatically.

Selenium

If you are testing an html-based UI, such as a cloud interface, then selenium may be required for your project. IMPORTANT NOTE: robot framework also has an internal selenium library, which can be included using the robot "Library" statement. iTAF does not use the robot Selenium library.

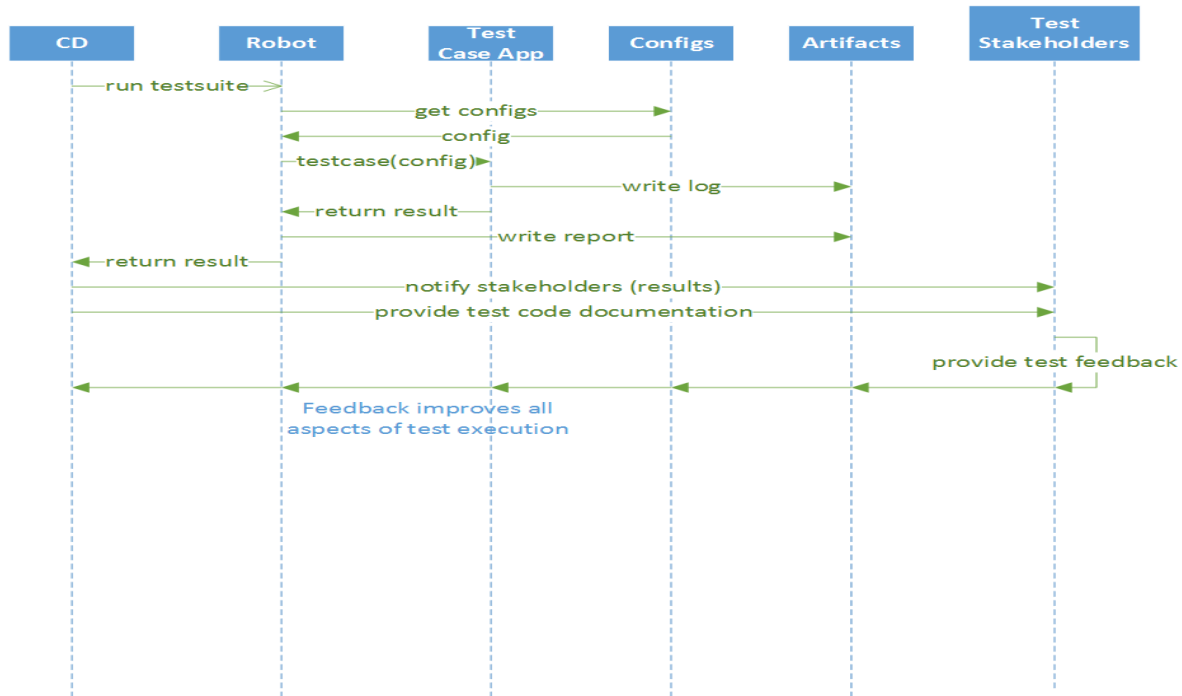
iTAF uses the standard python selenium library which can be installed using pip

Additional Libraries

If your specific project requires additional python libraries, they can usually be obtained through pip.

If you write a general-purpose library for iTAF, place it in one of the utils/src subdirectories. Then it will be available to everyone as a python import.

iTAF : Test Execution Flow



iTAF COMPONENTS

iTAF consists of several components:

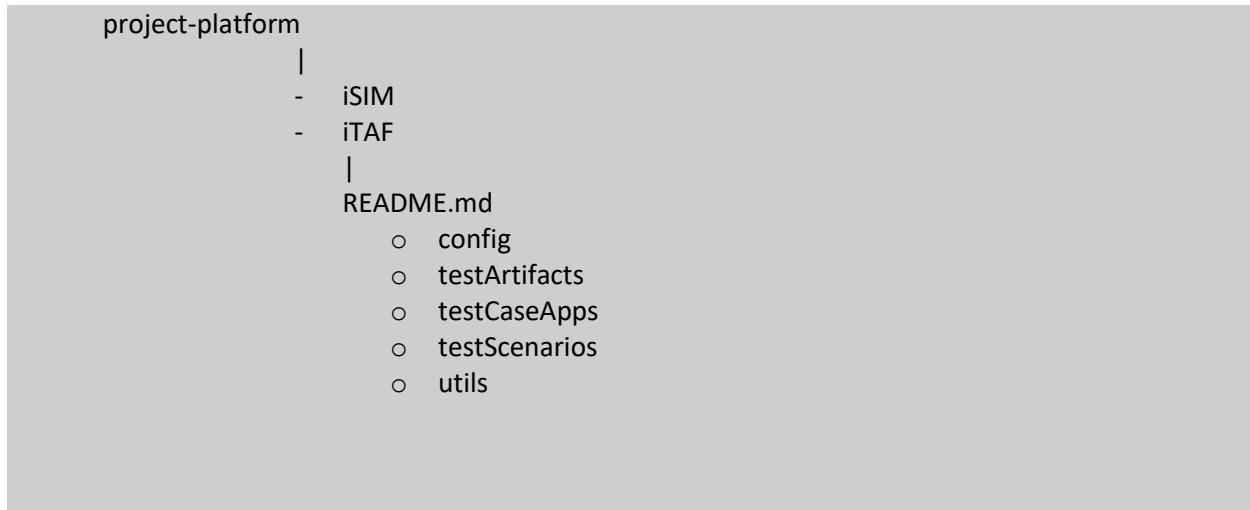
1. The iTAF directory structure
2. Python Libraries
3. Robot Framework
4. ConfigParser interface
5. Logging
6. Documentation

Each of these will be covered in this document.

iTAF Directory Structure

The iTAF directory structure is a template that is created in the validation git repository for each IOT product. <project>-platform is the normal naming convention for the repository. The iTAF template directory structure is a subdirectory of the project-platform

When this repository is checked out on a local machine, the directory structure will have the subdirectory structure as shown below.



Descriptions of the high level objects and subdirectories:

- **README.md** – the top level page of the Doxygen-generated web pages. This should be written for each IOT project.
- **config** – platform level configuration files are stored here. These configuration files contain data that is the same for all test cases in a project, for instance, the URL of the cloud.
- **testArtifacts** – this is where logs and robot output is stored. Logs files generated by test cases should have the same basename as the robot file. This will be discussed further in subsequent sections.
- **testCaseApps** – This is where test case apps are stored. It is subdivided by use case types. Each test case app should have it's own subdirectory within its use case type subdirectory. Specific test cases may also have their own specific config files there also.
- **testScenarios** – This is where robot files are stored. A test scenario is the condition when a test case app will be run.
- **utils** – This is where common utilities are stored. It is divided into subdirectories based upon the type of the utility. Test developers should always strive to write as generic code as possible.

Now, let us drill down and show the expanded directory structure.

The expanded directory structure is shown next:

```

config
├── demo-platform.cfg
├── profiles
│   └── API_paths.cfg
├── README
├── __init__.py
├── README.md
├── testArtifacts
│   ├── logs
│   │   └── UC_Demo.log
│   ├── README
│   ├── report
│   │   ├── log.html
│   │   ├── output.xml
│   │   └── report.html
│   └── robot
│       ├── log.html
│       ├── output.xml
│       └── report.html
├── testCaseApps
│   ├── AnalyticsDecision
│   │   └── README
│   ├── ConfigDeploy
│   │   └── README
│   ├── Decommission
│   │   └── README
│   ├── Demo
│   │   ├── Demo
│   │   ├── demo_setup_tearardown.py
│   │   ├── demo_setup_tearardown.pyc
│   │   ├── demo_tc_utils.py
│   │   ├── demo_tc_utils.pyc
│   │   ├── demo_testcases.py
│   │   ├── demo_testcases.pyc
│   │   └── README
│   ├── EdgeMgmt
│   │   └── README
│   ├── HealthMonitoring
│   │   └── README
│   ├── InitActivate
│   │   └── README
│   ├── __init__.py
│   ├── README
│   ├── Telemetry
│   │   └── README
├── testScenarios
│   ├── run.py
│   ├── UC_Demo
│   │   ├── demo.robot
│   │   ├── Test_Configs
│   │   │   └── Default.cfg
│   │   └── testResource
│   │       └── common.robot
├── utils
│   ├── doc
│   │   └── README
│   ├── __init__.py
│   ├── README
│   ├── src
│   │   ├── data
│   │   │   └── README
│   │   ├── exceptions
│   │   │   └── __init__.py
│   │   ├── __init__.py
│   │   ├── lib
│   │   ├── README
│   │   ├── report
│   │   │   ├── __init__.py
│   │   │   └── README
│   │   ├── setup
│   │   │   ├── __init__.py
│   │   │   └── README
│   │   ├── teardown
│   │   │   ├── __init__.py
│   │   │   └── README
│   │   └── verify
│   │       ├── __init__.py
│   │       └── README

```

testCaseApps

You can see that the testCaseApps contains subdirectories for different use case types:

Analytics/Decision Support: For test cases that involve data analytics functionality

Config/Deploy : Test cases supporting configuration and deployment functionality

Decommission: Test cases involving end-of-life cycle issues.

Edge Mgmt: Test cases for management of gateways and sensors from the cloud

Health Monitoring: These are for test functions which gather data about edge health (think SNMP type data)

InitActivate: Functionality related to initialization and activation of edge components.

Telemetry: Gathering data from sensors to the cloud.

testScenarios

testScenarios folder is where the robot files are stored. This is the subdirectory that iSIM will look for robot files to execute. Files in this have a naming convention which will be discussed later.

utils

Utils is where common utilities are stored. A test code that is generic for multiple people/projects to reuse is a great candidate to be stored in common utilities. This makes test case code cleaner and its benefits the entire team when generic code can be shared. If a code solution can be generalized, a library class is created and placed in the appropriate utils/src subdirectory.

Utils is subdivided into categories: data lib report setup teardown verify

This is where common code that can be used by many different test cases, should be placed. For instance: code used to access a cloud API or selenium code to login to a cloud web page.

As an example, the iTAF ColorLog logging library is in utils/src/report. It can be imported into any python script like this:

```
from utils.src.report.ColorLog import ColorLog

""" Example instantiating ColorLog """
""" This would likely be done in a setup routine """
testlog = ColorLog("/path/to/logfile")
```

Robot Framework

Robot Framework is used to automate execution of tests. Robot Framework is called by iSIM scripts to execute robot files within the *testScenarios* subdirectory.

The files in *testScenarios* have extension .robot and also have a specific required naming convention such that iSIM knows to execute them. The current naming convention for each robot file is that it must begin with UC_ and the first character after the underscore is capitalized. An example of this is

UC_Basic_Api.robot. This name meets the naming convention rule because it begins with UC_ immediately followed by a capital 'B' and it has the extension .robot.

It is important to note that Robot is only used to control execution of the test cases. Test cases are not written in the robot framework. Test cases are written in python. Robot has hooks to startup the python environment, variables that encompass the iTAF directory structure, pass configuration and logging variables to python test case code, and call setup, testcase execution, and teardown python methods. It also produces html execution status pages.

TestSettings and SettingsInfo

Python's ConfigParser is used to parse configuration files. In iTAF this is implemented in `utils/src/data/TestSettings.py`. Platform level configuration files are stored in the `config` subdirectory as shown in Figure 1.

Test level configuration files may also be stored close to test case apps in the `testCaseApps` subdirectories.

Python ConfigParser is a clean, simple configuration file parser. A configuration file consists of sections. Each section then consists of name=value pairs.

```
# My comment for this config file
```

```
[ThisIsOneSection]
```

```
myvar1=myval1
```

```
myvar2=myval2
```

```
[AnotherSection]
```

```
another1=someval1
```

```
thisvar=thatval
```

One or more configuration files will normally be specified in the robot file and passed to the test case app. iSIM will have one platform level configuration for items that are common for all test cases for that product. For instance, login credentials for a web page, or cloud URLs.

To assist test cases with the storage and retrieval of configuration information and other needed objects, `SettingInfo()` was developed as a generic name/object pair storage class. It is a test suite-level singleton that can be used to store objects by name. These objects can be configuration section dictionaries, helper classes like `ColorLog` for logfiles, or utility classes. Once stored in `SettingsInfo`, these objects can be retrieved anywhere in the testsuite by name reference. More on this later.

Parameterization and Metrics

iTAF should always be looking for parameters and metrics that should be used in testing. For instance, is there a limit to how long API calls should take?, how much data is stored regarding a particular sensor, how responsive, in terms of time, should a web page response be?

These kinds of parameters should be gathered at the beginning of the project. Never hardcode items in python code, they should, instead, be placed in configuration files. If they are not appropriate for the iSIM high level platform configuration, then place them in a separate configuration file with the test case application subdirectory.

Two classes in iTAF, TestSettings and SettingsInfo, both in `utils/src/data`, have been developed to make parsing these files and storing the key/value pairs very straightforward and simple.

The robot file is used to specify the config files, the setup routine can then get these sections parsed, and their data stored in SettingsInfo. Then, test cases are assured to have all the data they need before they begin executing. Furthermore, SettingsInfo can store any object by name, not just a configuration section dictionary.

Logging

For logging, iTAF uses the ColorLog class which wraps python's logging module. This class was obtained from the original Moon Island PSV validation team. PSV was running only manual tests in the console, so iTAF made some minor modifications to suppress console output for all but ERROR level messages.

The name ColorLog is used because it produces messages that are colored based upon the level of the log message. Usage is straightforward as shown below.

```
from utils.src.report.ColorLog import ColorLog

testlog = ColorLog(logfile)

testlog.info("This is an informational message")

testlog.error("This will log an error, colored red")
```

If you want log variable values, ints, floats, strings, etc, use the new style python string formatting. For instance, you have metric variables called `expected_api_response_time` and `actual_api_response_time`, and you want to log a failure.

```
testlog.error('ERROR: actual api response time {0}
              exceeded expected response time {1}'.format(actual_api_response_time,
              expected_api_response_time))
```

Since `expected_response_time` is probably a metric, it should come from a config file. For example

```
time_metrics = SettingsInfo().MYTIMEMETRICS
expected_api_response_time = time_metrics['api_response_time_max']
```

In this case, the testcase would return False and robot would log the failure at the iSIM level. iSIM would send you email with a link to the logfile, and you will be able to debug the failure. In this case, you might write a JIRA issue against the product.

Documentation

Doxygen is used as the main documentation tool. To enhance doxygen formatting with python, doxypy is used as a plugin.

The README.md in the main iTAF directory serves as the main page for the html documentation.

DETAILED WALKTHROUGH

In this section, we'll go through a detailed walkthrough of execution beginning when iSIM kicks off robot.

First Things First

When writing code we must observe the following rules:

1 . TABS are evil !

Never, ever, use a tab character in a source file. Before using any editor to edit a python or robot file, check the tab settings. The editor must either not use tabs or be able to convert tab key presses to 4 spaces. Some editors do auto-indenting to help the programmer. This is ok as long as the indent is converted to 4 spaces.

2. Use 4-space indenting

In python indenting is an inherit part of the language. Using 4-space indenting through-out iTAF is important for readability and consistency.

3. Remove extra white-space at end of line

It is good practice to ensure that source file lines end with a non-whitespace character before the newline. This is part of the PEP8 python formatting standard. When doing tool based code reviews ie gerrit, these extra whitespaces will show up in bold red blocks and you will receive comments, so it is better just to go ahead and make sure they are not there before checking code in.

Robot File

Test suites are executed by calling robot with the name of a .robot file as argument.

```
user@linuxmachine: robot robotfile.robot
```

Below is an example of a complete robot file. Notice we have a comment header at the beginning of the file.


```

# @file UC_API_TestSuite.robot
#   robot file for API functional tests
#
# @author : Tony Brown
# @created: Jan 2017
#
# DESCRIPTION:
# This is a robot test file for API functional tests
#

*** Settings ***
Documentation   Test Suite API Functional

## setup_python LIBRARY must be first in list
## it add all iTAF directories to sys.path so
## imports "just work" in test files
Library   ${TOPDIR}/utils/src/setup/setup_python.py
Library   ${TESTCASEAPPDIR}/API_TestSuite_setup.py
Library   ${TESTCASEAPPDIR}/API_Test.py
Library   ${TESTCASEAPPDIR}/API_TestSuite_teardown.py
Library   OperatingSystem
Library   Collections

Suite Setup   Common Setup
Suite Teardown   Teardown

*** Variables ***

${TOPDIR}      ../
${LIBDIR}      ../utils/src/lib

## the test case appdir is where the testapp is located
${TESTCASEAPPDIR}  ../testCaseApps/AnalyticsDecision/API_TestSuite
${TESTCASELOGDIR}  ../testArtifacts/Logs

## the test config dir is where the test configuration files are located
${PLATCONFIGDIR}  ../config

## TESTLOGDIR - note that ../testArtifacts/logs is part of the code directory structure
${TESTLOGDIR}      ../testArtifacts/logs
${cfg_platform}    ${PLATCONFIGDIR}/project-platform.cfg

${api_log}         ${TESTLOGDIR}/UC_API_TestSuite.log

${iterations}     2

```

```

${api_log}    ${TESTLOGDIR}/UC_API_TestSuite.log

${iterations}  2

${one} =      1

## END of *** Variables *** section

*** Keywords ***

## Called before test cases execute
Common Setup
    ## Setup the configuration name/cfg_file dictionary, so it is available to setup routine
    Set Config Locations
    ## Now call the setup routine
    ${setup_status} =  API Test Suite Setup  ${api_log}
    ## Make sure the setup routine was successful
    Should Be True    ${setup_status}      API Test Suite setup Failed

## Called after test cases execute
Teardown
    ${teardown_status} =  API Test Suite Teardown
    Should Be True    ${teardown_status}

## Called from 'Common Setup' above
##
## This creates a dictionary of config file section names and the config file in which that section can be
found
## see demo-platform.cfg for examples of sections such as Login and Gateway
## see testCaseApps/AnalyticsDecision/API_TestSuite_setup.py (function _get_objects)
## The ... syntax, which is used to continue the variable definition on multiple lines, requires minimum
robot 2.9
Set Config Locations
    ${CONFIG_PARAMS} =  Create Dictionary
        # SECTION NAME #          # CONFIG FILE LOCATION #
    ...  Login                ${PLATCONFIGDIR}/demo-platform.cfg
    ...  Gateway              ${PLATCONFIGDIR}/demo-platform.cfg
    ...  SimGateway           ${PLATCONFIGDIR}/demo-platform.cfg
    ...  Params_APITest1     ${TESTCASEAPPDIR}/Params_APITest1.cfg
    ...  Api_Post             ${TESTCASEAPPDIR}/API_paths.cfg
    ...  Api_Put              ${TESTCASEAPPDIR}/API_paths.cfg
    ...  Api_Delete          ${TESTCASEAPPDIR}/API_paths.cfg
    ...  TestSuite           ${TESTCASEAPPDIR}/Suite.cfg

    Set Suite Variable  ${CONFIG_PARAMS}

## End of *** Keywords *** section

```

```

...   Api_Delete           ${TESTCASEAPPDIR}/API_paths.cfg
...   TestSuite            ${TESTCASEAPPDIR}/Suite.cfg

Set Suite Variable  ${CONFIG_PARAMS}

## End of *** Keywords *** section

*** Test Cases ***

##
# ** API Inventory Tests
#

DO_API_TESTS
[Tags]    python
    Log To Console    API FUNCTIONAL TESTS
:FOR      ${index} IN RANGE  ${iterations}
\        ${one} = set variable  ${1}
\        ${plus} = Evaluate  ${index}+${one}
\        Log To Console  Running API Test1 (${plus} of ${iterations}) ...
\        ${status} = test get API Test1
\        Should Be True  ${status}          Failed API Test1

```

What Happens When Robot Starts Executing This File?

Robot parses the file into “Sections”.

Robot understands various formats for .robot files. iTAF uses the Plain text formatting. In Plain text formatting the section names used by iTAF are:

```

*** Settings ***

*** Variables ***

*** Keywords ***

*** Test Cases ***

```

It is important to understand that robot executes keywords. That is strange you might say, since the robot file above has only about 3 names in the *** Keywords *** section. The answer to this puzzle is in the *** Settings *** section where we have specified additional Libraries to load. When robot loads

Copyright © Intel Corporation, 2019. All rights reserved. This work is licensed under a Creative Commons Attribution 4.0 International License.

a python file specified in the `*** Settings ***` section, all the functions defined in those files become additional keywords. Notice in the `*** Settings ***` section, we have specified a Library named `API_APITest1.py`. In that file, there is the following function:

```
def test_get_API_Test1():
```

Robot will add `test_get_API_Test1` to its list of keywords. Please note: when robot loads python files to get keywords, it will not execute or verify the functions in the python code. If the python file has syntax errors robot will fail. If there is python code that is outside of any function, it may be executed, though, so please remember this.

Robot “Cell-based” Parsing

Notice the name of the routine in the python file; it is `test_get_API_Test1`.

In the `*** Test Cases ***` section of the robot file however, we are calling:

```
_${status} = test get API Test1
```

In the robot file we have spaces, not underscores. Robot does a kind of “cell-based” matching. It figures out that these two are the same thing.

A “cell” in robot doesn’t end until at least two spaces are encountered. That is why we have only one space between `_${status}` and the corresponding equal sign.

```
_${status} = <- this is all one cell
```

If it helps, you can imagine pipes.

```
|_${status} = | test get API Test1 |
```

If we were to put two spaces between `test` and `get` in the previous line, then Robot would complain that no keyword ‘test’ was found. Robot would see it this way:

```
|_${status} = | test | get API Test1 |
```

Remember when we were talking about Libraries? Notice that Robot also has its own built-in libraries of keywords. The robot file above is including the Robot built-in `OperatingSystem` and `Collections` libraries. although none of its keywords are actually being executed in this robot file. Robot also has a

BuiltIn library, which is always available and need not be explicitly specified. This is where the Create Dictionary function is located.

The “import” Problem

When iTAF was first conceived and the directory structure was first built, there was an issue with python imports. Look at the top of a typical python test case file:

```
from utils.src.report.ColorLog import ColorLog
from utils.src.data.SettingsInfo import SettingsInfo
from utils.src.data.TestSettings import TestSettings
from utils.src.lib.API import API
```

These Imports are in the same python test file that has the keyword “test get API Test1”. When robot would attempt to execute the test_get_API_Test1 python function, it would fail with import errors. Python could not find these imports. The reason was due to the complex iTAF directory structure. The test case file is in iTAF/testCaseApps/Telemetry/Basic_API whereas the ColorLog.py file is in iTAF/utils/src/report. The problem was that iTAF/utils/src/report is not in the PYTHONPATH, so python cannot find it.

Also, we did not want to hardcode some PYTHONPATH variable. An automated solution was found as follows:

1. When Robot loads a Library from the `*** Settings ***` section, if that library has python script that is not enclosed in a function (i.e. inside a def), it will execute that code.
2. Using this fact, a script was developed that will add the iTAF subdirectories to the PYTHONPATH. Note that inside a python script, the PYTHONPATH can be augmented by appending directories to sys.path.

Note the very first Library loaded in a robot file is:

```
Library  ${TOPDIR}/utils/src/setup/setup_python.py
```

In the `*** Variables ***` section `${TOPDIR}` is set to ..

So, there is one assumption, that the robot file is one level below iTAF

The setup_python.py file is shown next:

```

"""
This script sets up the python environment for iTAF

@package utils.src.setup.setup_python
"""

import os
import sys

"""
setup python environment

*** IMPORTANT ***
This module should be the first Library
included in the robot file. Robot should
be executed from testScenarios subdir.

Given a toplevel directory, add all subdirs to the pythonpath.
Note that sys.path.append is equivalent to specifying PYTHONPATH.

TOPLEVEL *MUST BE* iTAF, which is one directory above robots
current directory which is iTAF/testScenarios.
"""
TOPLEVEL = "../"
""" We need to list the subdirs directly
under the TOPLEVEL, so we can exclude
any git dir, since the test code might
reside in a git repo
"""
ABS_TOP = os.path.abspath(TOPLEVEL)
top_subdirs = os.listdir(ABS_TOP)
sys.path.append(ABS_TOP)

""" EXCLUDE ANY GIT DIRS """
top_dirs_no_git = []
for sdir in top_subdirs:
    if str(sdir) != '.git':
        top_dirs_no_git.append(ABS_TOP + os.path.sep + sdir)

""" Now we can walk the top level subdirs
and build the python path
"""
for tdir in top_dirs_no_git:
    for root, subdirs, files in os.walk(tdir, topdown=True):
        if root not in sys.path:

```

```

top_dirs_no_git.append(ABS_TOP + os.path.sep + sdir)

''' Now we can walk the top level subdirs
and build the python path
'''
for tdir in top_dirs_no_git:
    for root, subdirs, files in os.walk(tdir, topdown=True):
        if root not in sys.path:
            if 'pycache' not in str(root):
                sys.path.append(root)

def iTAF_dummy_keyword():
    """ This just avoids a robot WARN message.
    Without this, Robot WARNS this file
    contains no keywords.
    """
    pass

```

Notice it is just script with one additional function. Without the additional function, robot WARNS that the library contains no keywords. To suppress this message, a dummy keyword was created, iTAF_dummy_keyword.

From this, we can see that a python function is the same thing as a robot keyword, as far as iTAF is concerned.

Remember the line in the robot file which loaded the setup_python Library? It used the variable \${TOPDIR}. That variable was defined in the ***** Variables *****, but the Variables section appears later in the file than the ***** Settings ***** section. This shows that variables defined in the ***** Variables ***** section can be used in any other section, they don't need to be defined "before" being used.

Setup And Teardown

Looking again at the ***** Settings ***** section we also see two other items:

```

Suite Setup    Common Setup
Suite Teardown Teardown

```

In the lefthand cell of these two lines, “Suite Setup” and “Suite Teardown”, are Robot-defined Settings. They are not required, but Robot, like most test systems, has the concept of “Setting up” test suites and “Tearing down” test suites.

The rightmost “cell” of these two definitions are “Common Setup” and “Teardown”. Robot assumes these are keywords, so it looks in the `*** Keywords ***` section, and they must be there or robot will fail with a “Keyword not found” error.

If we did not have explicit Suite Setup and/or Suite Teardown functions, then we could either delete those two lines or comment them out.

Looking at the `*** Keywords ***` section, we have “Common Setup” and “Teardown” keywords.

```
## Called before test cases execute
Common Setup
    ## Setup the configuration name/cfg_file dictionary, so it is available to setup routine
    Set Config Locations
    ## Now call the setup routine
    ${setup_status} = API Test Suite Setup  ${api_log}
    ## Make sure the setup routine was successful
    Should Be True    ${setup_status}    API Test Suite setup Failed
```

Don’t forget the cell concept, here is what robot sees:

```
Common Setup
| Set Config Locations |
| ${setup_status} =   | API Test Suite Setup | ${api_log}
| Should Be True     | ${setup_status}    | API Test Suite setup Failed
```

In Common Setup, we are telling robot that it should first call ‘Set Config Locations’ and then assign to `${setup_status}` the result of calling keyword “Setup Basic Api” with the argument stored in variable `${api_log}`

And, we do have these routines in our `*** Keywords ***` section, either directly in the robot file or dynamically loaded from one of the Library files. Notice that ‘Set Config Locations’ is defined directly in the robot file, and that we are calling it before the ‘API Test Suite Setup’ routine. We will discuss this in more detail shortly.

Note in “Common Setup” that robot expects to get a return value from “API Test Suite Setup” and then it tests whether that return value is True. If not, it will fail and not run any test cases in the `*** Test cases ***` section.

Note also that Teardown returns a status, and robot will check that as well. This will also cause a robot failure. If the test case passed, Robot will show the test case passed, but robot will report that teardown failed.

At this point, we can outline the robot flow of execution:

Robot Flow Of Execution

ROBOT FLOW OF EXECUTION

- Parse the `*** Variables ***` section and make the variables defined there available to the entire script.
- Load the `*** Settings ***` section and import all the Library files. Any code which is not enclosed in a `def`, gets executed immediately. Any functions, which are defined by python `def` statements, are added to the list of keywords in the `*** Keywords ***` section.
- If there is a Suite Setup defined in `*** Settings ***`, find the associated keywords and execute them. If there is a pass/fail condition tied to the Suite Setup, then stop and report the error if that condition has failed.
- If the Suite Setup condition has passed, then proceed to execute the test cases in the `*** Test cases ***` section. Unless instructed otherwise, run all test cases and report their status.
- If a Suite Teardown exists in the `*** Settings ***` section, then, after all test cases have been executed, run the keyword associated with it.

Keyword *SET CONFIG LOCATIONS*

An important aspect of iTAF is the ability to specify various configuration parameters. This allows the automation system to dynamically iterate over test cases using different configurations each time. In iTAF configuration files, as mentioned previously, the typical python ConfigParser format is used.

During test design, the test designer should determine all the applicable parameters that will need to be iterated over during the test case execution. Various configurations within or among different robot files which reflect different mixes of configurations can then be designed.

We can see a simple example of how this works in the robot file. The *Set Config Locations* keyword is shown again below:

Set Config Locations

```

${CONFIG_PARAMS} = Create Dictionary
# SECTION NAME #           # CONFIG FILE LOCATION #
... Login                  ${PLATCONFIGDIR}/project-platform.cfg
... Gateway                ${PLATCONFIGDIR}/project-platform.cfg
... SimGateway             ${PLATCONFIGDIR}/project-platform.cfg
... Params_APITest1       ${TESTCASEAPPDIR}/Params_APITest1.cfg
... Params_APITest2       ${TESTCASEAPPDIR}/Params_APITest2.cfg
... Api_Post               ${TESTCASEAPPDIR}/API_paths.cfg
... Api_Put                ${TESTCASEAPPDIR}/API_paths.cfg
... Api_Delete             ${TESTCASEAPPDIR}/API_paths.cfg
... TestSuite              ${TESTCASEAPPDIR}/Suite.cfg

Set Suite Variable  ${CONFIG_PARAMS}
```

Set Config Locations is a robot keyword, when it is executed it creates a dictionary. Each key in the dictionary is a config file section name, and each value is the path of a config file where a section with that name can be found. Once the dictionary is created, *Set Suite Variable* is called to set the name of the dictionary as a variable in the suite. Then, the *API Test Suite setup* routine can access that variable and parse the config file.

Now, look at the *API Test Suite setup* routine (this file is located in directory iTAF/testCaseApps/AnalyticsDecision/API_TestSuite):

```

from robot.libraries.BuiltIn import BuiltIn

def api_testsuite_setup(logname):
    """
    sets up the objects needed for the test suite
    """
    @param logname logfile for the suite
    @return Boolean pass/fail condition to Robot
    """
    testlog = None

    try:
        _get_objects(logname)

        testlog = SettingsInfo().TESTLOG
        testlog.info('api_testsuite_setup : log: {0}'.format(logname))

    except Exception:
        """ If we get an exception, we cannot depend upon
           testlog being available, otherwise log to robot console
        """
        if testlog:
            testlog.log_exception('api_testsuite_setup')
        else:
            logger.console('Exception in api_testsuite_setup: {0};{1}'.format(
                sys.exc_info()[0], sys.exc_info()[1]))

    return False

```

The first thing the setup routine tries to do is `_get_objects(logname)`. Notice we have included robots BuiltIn library. We will need this to get access to the CONFIG_PARAMS dictionary we created in the robot file.

Now, let's look at the `_get_objects` routine, which is in the same python file.

```

def _get_objects(logname):
    """
    Get the objects needed by the test cases
    """
    """
    The logname is handled separately, because we want to get it
    early, so we can start logging immediately.
    """
    testlog = ColorLog(logname)
    SettingsInfo().add_name('TESTLOG', testlog)

    """
    Populate SettingsInfo with all the section dictionaries from all the config files

    Use robot BuiltIn to read the PARAM_CONFIGS dictionary from the robot file.
    Each dict entry will have a config file section name, e.g. Login, as key,
    and the corresponding value will be a config file where that section is
    to be found. (See iTAF/configs/project-platform as an example config file).

    For each name key, the 'config file' value is passed to TestSettings. Then,
    we call it to parse a section by that name. If successful, it returns
    a dictionary of settings for that section name.

    Finally, we add the name and the dictionary to the SettingsInfo() singleton.
    The {name, dictionary pair} is then available to any test case that needs
    those values.
    """
    config_names = BuiltIn().get_variable_value('&{CONFIG_PARAMS}', None)
    testlog.info('APITestSuite_setup: cfg names val: {0}'.format(config_names))

    if isinstance(config_names, dict):
        for config_name, cfg_file in config_names.iteritems():
            try:
                testlog.info('get_objects: getting section {0} from {1}'.format(config_name, cfg_file))
                config_parser = TestSettings(cfg_file)
                config_dictionary = config_parser.get_section(config_name)
                SettingsInfo().add_name(config_name, config_dictionary)
            except:
                testlog.log_exception('API_TestSuite_setup')
                return False
    else:
        testlog.error('API_TestSuite_setup: PARAM_CONFIGS from robot file should ' +
            'be a dictionary but is type {0}'.format(type(config_names)))
    return False

```

For each dictionary entry, we instantiate a TestSettings object with the file name *cfg_file*. If successful we ask TestSettings to parse a section with name *config_name*. If that is successful, we add the name and section dictionary to the SettingsInfo() singleton. Then, it is available to any test case that needs it.

The test case can simply access the section dictionary by doing, for example, logindict = SettingsInfo().Login. Then, to access individual elements, for example, cloud_url = logindict['base_url'].

TestSettings

The TestSettings class is used to parse configuration files. The files have a format similar to Windows INI files. For instance, here is a config file, called Api_paths.cfg, that can be parsed with TestSettings:

```
## @file API_paths.cfg
#   configuration file for API test
#
#
# @author : Tony Brown
# @created:
#
# DESCRIPTION:
# configuration file for API test
#
# These are the API endpoints supported by the API
# These paths need to be run on SED line, there is no
# proxy support
#
[Api_Post]
API1=/API/query/API1
API2=/API/query/API2

[Api_Put]
API1=/API/query/API1

[Api_Delete]
API1=/API/query/API1
```

It contains three sections, Api_Post, Api_Put, and Api_Delete. We can parse it as follows:

```
cfg_parser = TestSettings(Api_paths.cfg)
postdict = cfg_parser.get_section('Api_Post')
putdict = cfg_parser.get_section('Api_Put')
```

```
deletedict = cfg_parser.get_section('Api_Delete')
```

Then, if we wanted to get the API path for 'Test1' calls, we could do:

```
Test1_post_path = postdict['Test1']
```

The TestSettings class consists of only two functions, the `__init__(self, cfg_file)` constructor and a function called `get_section(self, section_name)`. This function expects a section with name `section_name` to be found in the configuration file `cfg_file`.

What Is SettingsInfo?

This is a new feature in iTAF. This could also be backported to previous projects as well. The SettingsInfo class makes getting access to configuration data easier and cleaner.

Let us look at it:

```
"""
A singleton class to hold name, object pairs

@package  utils.src.setup.SettingSInfo
"""

class SettingsInfo(object):

    """
    class to hold setup objects
    so there is only one instance
    throughout a test suite

    @class SettingsInfo
    """
```

```

class __SettingsInfo(object):
    """
    Inner class represents the real object
    @param none
    @return __SettingsInfo
    """
    def __init__(self):
        """
        init need do nothing but return object
        """
        pass

    def __add_name(self, name, obj):
        """
        __add_name adds an object with the specified name
        @param name string
        @param obj object
        """
        setattr(self, name, obj)
## the single object instance
instance = None
def __init__(self):
    """
    __init__ instantiates new object or return existing object
    """
    if not SettingsInfo.instance:
        SettingsInfo.instance = SettingsInfo.__SettingsInfo()
def __getattr__(self, name):
    """
    __getattr__ returns the specified object if it exists
    """
    return getattr(self.instance, name)

def add_name(self, name, obj):
    """
    add_name wrapper for adding a name to the inner object
    """
    self.instance.__add_name(name, obj)

```

Note that it is a singleton, meaning there will be only one object instance throughout the execution of a test suite. It has one job, to hold name/object pairs.

It consists of an outer class, which provides the API, and an inner class, which is the object holding the name/object pairs.

Any python file can simply do:

```
from utils.src.data.SettingsInfo import SettingsInfo

""" To add a name """
myobject = Object()
SettingsInfo().add_name('Some_meaningful_name', myobject)

""" To get it back.
    Note: this could be in a totally separate function or
        in a different python source file from the add_name call
    """
theobject = SettingsInfo().Some_meaningful_name
```

Python Test Case File

Now let's look at one of the test case files for the API TestSuite. Recall in the robot file we have test cases in the `*** Test cases ***` part of the file. There, we have:

```
    ${status} = test get API Test1
```

At the top of the robot file we have

```
Library    ${TESTCASEAPPDIR}/API_APITest1.py
```

When robot loads the Library it will find a python function matching the name `test get API Test1`

That function is shown below. Currently, it using the Simulator to test the cloud 'Test1' API.


```

def test_get_API_Test1():
    """
    Test get API Test1

    @return Boolean This return goes back to the test runner (robot or main)
    """

    tlog = None
    simgw = None
    simargs = None
    try:
        """ get log first """
        tlog = SettingsInfo().TESTLOG

        _set_debug()

        params_cfg = SettingsInfo().Params_APITest1
        if params_cfg is None:
            tlog.info('test_get_API_Test1: params_cfg is NONE')
        else:
            tlog.info('test_get_API_Test1: params_cfg is {0}'.format(params_cfg))

        """ See if we're using a simulated gateway """
        try:
            simgw = SettingsInfo().SIMGW
        except:
            tlog.info('No simulated GW found')
            pass
        else:
            tlog.info('USING simulated GW')
            try:
                SettingsInfo().add_name('DOSIM', True)
                simargs = SettingsInfo().SimGateway
                if isinstance(simargs, dict):
                    result = _do_sim_Test1()
                    return result
            else:
                tlog.error('test_get_API_Test1: Unable to get simargs')
                return False
        except:
            tlog.log_exception('test_get_API_Test1: Exception in sim_Test1')
            return False

```

Notice how it uses SettingsInfo() to get the objects it needs to run the test case. We also make sure it returns a True or False back to robot for the test result status.

Summary Of Walkthrough - Guidelines

Let us summarize what we have so far.

1. A robot file normally corresponds with one python test case file.
2. All test case keywords will be in that file and it is made known to robot using the Robot Library call.
3. All Robot files have the *FIRST* Library call as iTAF/utls/src/setup/setup_python.py. This sets up the python environment so that imports 'just work'
4. All code that is common should go somewhere in the utls/src subdirectories. Currently utls/src is divided into these subdirectories: data lib report setup teardown verify
5. If you are writing code that you think could be shared within your project, consider placing it somewhere in the utls/src path. Decide what category it belongs to: data lib report setup teardown or verify. Let's expand on this a little. For instance, ColorLog is in util/src/report because it has to do with reporting things. The TestSettings class is in utls/src/data because it has to do with parsing and storing test case data, and anybody can use it. The SettingsInfo convenience class is also in utls/src/data because it has to with storing test data in an easy, lookup form. It is general purpose, so anybody can use it.
6. utls has been made into a true python package so importing is as easy as:
from utls.src.report.ColorLog import ColorLog
from utls.src.data.TestSettings import TestSettings
7. The <project>API is a general purpose API library that is useful for <project> test cases so it was put in utls.src.lib a place for more generic things.
8. Always observe the three rules. We will never use a TAB character in a python source file. We will always remove trailing whitespace. We will always use 4-space indentation.
9. We will document source files. More on this in a subsequent section. Install doxygen on your local machine.
10. The python test case file that robot is calling will use the following naming convention:
Function names called during the robot *** Suite Setup *** phase will begin with setup_
Function names called during the *** Test cases *** phase will begin with test_
Function names called during the *** Suite Teardown *** phase will begin with teardown_
Function names that are internal to the test case file will begin with _.
(Internal, private functions beginning with _ is already a python BKM)
11. Install the pep8 program (google it) on your local machine and use it as a guideline before checking in code. DO NOT 'fix' everything it reports. It is only a guideline that can help your programming.

DOCUMENTING THE CODE

ITAF uses doxygen for its code documentation. Below sections will help us to understand the doxygen usage.

Introduction

- Doxygen is a documentation system for C++, C, Java, Python, Objective-C, IDL (Corba and Microsoft flavors) etc.
- It helps in
 - Generating on-line documentation or offline reference manual from documented source files.
 - Extracting the Code Structure and visualising the relations between various elements by means of include dependency graphs, inheritance diagrams, and collaboration diagrams, which are all generated automatically.

Creating Configuration File

- Doxygen determines settings from Configuration file.
 - A configuration file is a free-form ASCII text file with a structure that is similar to that of a Makefile, with the default name Doxyfile.
 - The statements in the file are case-sensitive. Comments may be placed anywhere within the file (except within quotes).
 - The file essentially consists of a list of assignment statements. Each statement consists of a TAG_NAME written in capitals, followed by the equal sign (=) and one or more values.
 - For tags that take a list as their argument, the += operator can be used instead of = to append new values to the list.
 - If the value should contain one or more blanks it must be surrounded by quotes ("...").
 - Multiple lines can be concatenated by inserting a backslash (\) as the last character of a line.
 - Environment variables can be expanded using the pattern \$(ENV_VARIABLE_NAME).
 - You can also include part of a configuration file from another configuration file using a @INCLUDE tag as follows:

```
@INCLUDE = config_file_name
```
 - Configuration File

```

# Doxyfile 1.8.3

# This file describes the settings to be used by the documentation system
# doxygen (www.doxygen.org) for a project.
#
# All text after a hash (#) is considered a comment and will be ignored.
# The format is:
#   TAG = value [value, ...]
# For lists items can also be appended using:
#   TAG += value [value, ...]
# Values that contain spaces should be placed between quotes (" ").

-----
# Project related configuration options
-----

# This tag specifies the encoding used for all characters in the config file
# that follow. The default is UTF-8 which is also the encoding used for all
# text before the first occurrence of this tag. Doxygen uses libiconv (or the
# iconv built into libc) for the transcoding. See
# http://www.gnu.org/software/libiconv for the list of possible encodings.

DOXYFILE_ENCODING      = UTF-8

# The PROJECT_NAME tag is a single word (or sequence of words) that should
# identify the project. Note that if you do not use Doxywizard you need
# to put quotes around the project name if it contains spaces.

PROJECT_NAME           = "Demo Doxygen Documentation"

# The PROJECT_NUMBER tag can be used to enter a project or revision number.
# This could be handy for archiving the generated documentation or
# if some version control system is used.

PROJECT_NUMBER         = 1.0

# Using the PROJECT_BRIEF tag one can provide an optional one line description
# for a project that appears at the top of each page and should give viewer
# a quick idea about the purpose of the project. Keep the description short.

PROJECT_BRIEF          =

# With the PROJECT_LOGO tag one can specify an logo or icon that is
# included in the documentation. The maximum height of the logo should not
# exceed 55 pixels and the maximum width should not exceed 200 pixels.
# Doxygen will copy the logo to the output directory.

PROJECT_LOGO           =

# The OUTPUT_DIRECTORY tag is used to specify the (relative or absolute)

```

- Doxywizard is a GUI front-end for configuring and running doxygen.
 - Currently we are not using this method.

Document the Code

- Python documentation sections
 - Document Blocks or Lines – Python example
 - Document members
 - Structural Commands
 - Create Lists
- For Python there is a standard way of documenting the code using so called documentation strings. Such strings are stored in **doc** and can be retrieved at runtime. Doxygen will extract such comments and assume they have to be represented in a preformatted way.

```

"""@package docstring
Documentation for this module.

More details.
"""

def func():
    """Documentation for a function.

    More details.
    """
    pass

class PyClass:
    """Documentation for a class.

    More details.
    """

    def __init__(self):
        """The constructor."""
        self._memVar = 0;

    def PyMethod(self):
        """Documentation for a method."""
        pass

```

- There is also another way to document Python code using comments that start with "##". These type of comment blocks are more in line with the way documentation blocks work for the other languages supported by doxygen and this also allows the use of special commands. Here is the same example again but now using doxygen style comments:

```

## @package pyexample
# Documentation for this module.
#
# More details.

## Documentation for a function.
#
# More details.
def func():
    pass

## Documentation for a class.
#
# More details.
class PyClass:

    ## The constructor.
    def __init__(self):
        self._memVar = 0;

    ## Documentation for a method.
    # @param self The object pointer.
    def PyMethod(self):
        pass

    ## A class variable.
    classVar = 0;

    ## @var _memVar
    # a member variable

```

- Creating Lists (Generic Example)
 - Bulleted Lists - Column aligned Minus Sign '-'
 - Numbered Lists - Column aligned Minus Sign followed by Hash '-#'
 - Nesting Level is maintained according to column alignment.
 - HTML Commands - Can be used inside comment blocks.

Syntax:

```

/*!
* List of events
* -Event1
* -#Subevent1
* -#Subevent2\n
* subevent2 cont.
* -#Subevent3
* -Event2
* -Event3
*/

```

- Structural Commands (Generic Example)
 - Starts with '^' or '@'.
 - Main commands
 - \author {list of authors}
 - \brief {brief description}
 - \date {date description}
 - \file [<name>]
 - \fn (function name)- used if comment block is not placed
 - \param <parameter-name> {parameter description}
 - \return {return value description}
 - Visual Enhancement Commands
 - \a - Special font
 - \b - bold
 - \c - Typewriter Font
 - \arg - Simple bulleted list, not nested
 - \e - Italics
 - \em - Emphasize word and in Italics

Run the Doxygen

To generate the documentation, use the below command.

```
doxygen <Configuration File>
```

Sample reports

Demo Doxygen Documentation 1.0

Main Page **Files**

File List

File List

Here is a list of all documented files with brief descriptions:

__init__.py	
testCaseApps/__init__.py	
utils/__init__.py	
utils/src/__init__.py	
utils/src/exceptions/__init__.py	
utils/src/report/__init__.py	
utils/src/setup/__init__.py	
utils/src/teardown/__init__.py	
utils/src/verify/__init__.py	
demo_setup_teardown.py	: This is a demo test file
demo_tc_utils.py	:
demo_testcases.py	:
run.py	

Generated by  1.8.6

Main Page **Namespaces** **Classes** Files

Class List Class Index Class Hierarchy Class Members

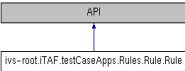
ivs-root ITAF testCaseApps Rules Rule Rule

ivs-root.ITAF.testCaseApps.Rules.Rule.Rule Class Reference

Public Member Functions | Public Attributes | Static Public Attributes | List of all members

This class is a subclass of API and extends the Rule API. More...

Inheritance diagram for ivs-root.ITAF.testCaseApps.Rules.Rule.Rule:



```
graph BT; API[API] --> Rule[ivs-root.ITAF.testCaseApps.Rules.Rule.Rule];
```

Public Member Functions

def `__init__(self, postfix=None, kwargs)`
Constructor. More...