Contents

TAF is an integrated cross-platform system that sets the guidelines and provides tools for automation testing of a specific product in Networking.

TAF allows you to write a series of tests without worrying about the constraints or limitation of underlying test tools.

Chapter 1

Goals:

- create libraries with all necessary functionality;
- define the way how environment configuration description will be supplied to test suites;
- create the reporting processing functionality;
- create the smart regression analysis functionality

Chapter 2

Guide

2.1 TAF architecture

2.1.1 Directory structure of TAF

The current implementation of the testing framework has the following directory structure (only high-level directories are shown):

```
+ taf-repo
   + docs
   + reporting
   + taf
     - plugins
     - testlib
   + tests
   + unittests
   + utils
```

Directory **docs** contains documentation generated using sphinx and readthedocs tools.

Directory **plugins** contains TAF related plugins wich extends py.test and TAF testlib functionality and could be enabled/disabled for particular tests ot group of tests.

Directory **testlib** contains common functionality for the majority of tests, platform-specific libraries and various helper functions, described in System Architecture.

Directory **unittests** contains TAF unittests and TAF functional tests.

2.1.2 TAF plugins

Available **TAF plugins** located in **plugins** sub-directory. The most useful of plugins are:

plugins.pytest_reportingserver

- Starts Reporting Server as a separate process.
- Collects information about test case duration split by stages (setup/call/teardown).
- **Options:** `--xml_html` - create html report file at given path.

plugins.pytest_returns

Sometimes user wants to get specific information from the test instead of PASS/FAIL status. TAF allows to include returned information from the test into pytest . User must modify test case:

- Add return statement as test step in order to return necessary information.
- Add *@pytest.mark.returns* decorator to the test case or test class.

plugins.pytest_syslog

- Send notifications about test case start/end to the remote syslog server.
- Separate device must be specified in the environment Json file with syslog_settings instance_type value, e.g.:

```
[
  {"name": "std_syslog_settings", "entry_type": "settings", "instance_type": "syslog_settings", "id": "4",
        "ip": "X.X.X.X", "proto": "Udp", "port": 514, "localport": 514, "transport": "Tcp", "facility": -1,
        "severity": "Debug",
        "syslog_usr": "user", "syslog_passw": "password", "path_to_log": "/var/log/switches/"},
]
```

- This device must be included in the **related** devices of the DUT (**related_id:** [**6**]), e.g.:

```
[
  {"name": "simswitch1_lxc", "entry_type": "switch", "instance_type": "lxc", "id": "16",
        "ip_host": "X.X.X.X", "ip_port": "8081", "ports_count": 32,
        "cli_user": "lxc_user", "cli_user_passw": "password", "cli_user_prompt": "Switch ",
        "cli_img_path": "usr/lib/ons/cli_img/",
        "ports": [1, 2, 3],
        "related_id": ["6"]},
]
```

- **Options:** `--syslog` – enable syslog plugin. False by default.

plugins.pytest_pidchecker

- TAF gets info about ONS process IDs on test setup and teardown.
- TAF verifies PIDs are not changed during test case execution. In other case tests teardown fails, TAF provides information about restarted processes.
- **Options:** `--pidcheck_disable` – disable process IDs verification.

Note: During specific tests some processes could be restarted by design or device could be restarted. TAF has a special marker for these test cases that allows to skip process ID validation: *@pytest.mark.skip_pidchecker, @pytest.mark.skip_pidchecker(process1, process2)*

---

plugins.pytest_caselogger

- Stores devices logs on the remote host after test execution.

- **Options:** `--log_enable` – enable/disable log tool for test (False | True).

plugins.pytest_multiple_run

- Execute test cases N times in a loop. N=1 by default.
- **Options:** `--multiple_run=N`

plugins.pytest_start_from_case

- Run test suite starting from specific test case.
- **Options:** `--start_from_case`

User may use strict test names or patterns, e.g.:

```
--start_from_case  test_my_func
--start_from_case  test*func
--start_from_case  *func
--start_from_case  test*
```

plugins.pytest_smartrerun

- Reruns Test Cases with Failed and Cant Test status from custom Test Plan.
- **Options:** `--sm_rerun` –custom Test Plan name.

plugins.pytest_heat_checker

- TAF gets info about CPU temperature from ONS Sensors table and adds it into the test run logs.
- **Options:** `--heat_check` – enable/disable tool for temperature logging (False | True).

plugins.pytest_onsenv

- Initializes environment from common3.py module:
  - Reads environment json file
  - Reads setup json file.
  - Loads dev_* modules.
  - Creates instances of used devices according to setup json file.
- **Options:** `--env` – path to environment json file. None by default.

  `--setup` – path to setup json file. None by default.

plugins.pytest_skip_filter

- Remove skipped test cases from list of collected items.

Note: Skip reason must be specified for all skipif markers

---

plugins.pytest_loganalyzer

- Performs analysis for ONPSS devices logs, checks for duplicates and errors.
- **Options:** `--log_analyzer` – enable/disable log tool for test (False | True).

2.1.3 TAF features overview

Support for:

1. Cross-connection solutions (Vlab, static links)
2. Traffic generators (Ixia, TRex)
3. Switches (ONS, ONPSS, Simulated)
4. OVS controllers (OFTest, Floodlight)
5. Power boards (APC)
6. Terminal servers

Integration with:

1. Test Case Management Systems (Jira, SynapseRT)
2. Defect Trackers (Jira)

Available **TAF features** located in **testlib** sub-directory. The most useful of them are:

TAF devices

common3.py	main environment file
dev_switch_*.py	switch functionality
dev_ixia.py	TG functionality
dev_chef.py	chef functionality
dev_*cross.py	cross connector functionality
dev_ovscontroller.py	OVS functionality
dev_linux_host.py	Linux host functionality

TAF commons

entry_template.py	generic code for all devices
switch_general.py, switch_ons.py	generic code for switches
testlib/Ixia/*	Ixia related files
packet_processor.py	generic packet operations
clissh.py, clitelnet.py	ssh, Telnet connection
powerboard.py	APC functionality

TAF UIs

ui_wrapper.py	generic code for all UIs
ui_ons_xmlrpc.py	wrappers for ONS XmlRpc calls
ui_ons_cli.py	wrappers for ONS ClI calls
ui_onpss_shell.py	wrappers for ONPSS Shell calls
ui_onpss_jsonrpc.py	wrappers for ONPSS JsonRpc

TAF helpers

ui_helpers.py	general switch operations
helpers.py	general tests operations

2.2 Test execution preconfiguration

All TAF test cases are parameterized in a way that they have access to **env** object which is built from environment description. For example typical **env** object contains already initialized switch(es), traffic generator and cross connection tool. You just need to describe in your conftest.py file and test which methods and in which order should be called on different stages of test execution.

2.2.1 SETUP config - testcases/config/setup/*.json

Setup config contains **env** and **cross** parts.

- The **env** part is a list of devices in the current setup. Its format is the same as the environment config, except that you need define only entry IDs that corresponds to the ones in the env conf file.
- The **cross** part is a dictionary with cross ids as the keys and the appropriate list of connections as values.

Env Part

In general, you should define only ids of necessary devices, but you can also override parts of the device configuration from **env** config.

Examples:

```
{
"env": [
        {"id": 22, "related_id": ["33"]},
        {"id": "33", "related_id": [16]},
        {"id": 16}
       ],
}
```

```
{
"env": [
        {"id": 0, "ports": [[1, 1, 6], [1, 1, 7], [1, 1, 8], [1, 1, 9], [1, 1, 10], [1, 1, 11], [1,
↪ 1, 12], [1, 1, 13], [1, 1, 14]]},
```

(continues on next page)

(continued from previous page)

```
        {"id": 1, "ports": [24, 25, 26, 33, 48, 39, 34, 35]},
        {"id": 4, "ports": [28, 29, 24, 25, 26, 27]},
        {"id": 3, "ports": [24, 25, 28, 29, 34, 35]},
        {"id": 2, "ports": [24, 25, 33, 48, 39, 34, 35]},
        {"id": "33"}
       ],
}
```

Cross Part

This list of connections is a list of lists. Each connection is a list of four elements: *[<device1 id>=>, <port id>=>, <device2 id>=>, <port id>=>]* :

- Port ID is the ID for a port in the list of real ports in the device configuration (ids starts from 1).
- Device id is the value of the id field in the environment config.

Examples:

```
{
"cross": {"2": [[0,1,1,1], [0,2,1,2], [0,3,1,3], [0,4,1,4], [0,5,1,5]]}
}
```

```
{
"cross": {"4": [[0,1,1,1], [0,2,1,2], [0,3,1,3], [0,4,1,4], [0,5,1,5], [0,10,3,1],
                [0,11,3,2], [0,12,3,3], [0,6,2,1], [0,7,2,2],
                [0,8,2,3], [0,9,2,4]],
          "5": [[1,16,2,16], [1,17,2,17], [1,18,2,18], [1,19,2,19],
                [1,20,2,20], [1,21,2,21],
                [1,22,2,22], [1,23,2,23], [1,24,2,24], [1,11,3,11],
                [1,12,3,12], [1,13,3,13], [1,14,3,14],
                [2,11,3,5], [2,12,3,6], [2,13,3,7], [2,14,3,8]]}
}
```

Complete Configuration

Examples:

```
{
"env": [
        {"id": 0, "ports": [[1, 1, 6], [1, 1, 7], [1, 1, 8], [1, 1, 9], [1, 1, 10]]},
        {"id": 1},
        {"id": "30"}
       ],
"cross": {"30": [[0,1,1,1], [0,2,1,2], [0,3,1,3], [0,4,1,4], [0,5,1,5]]}
}
```

```
{
"env": [
        {"id": 22, "related_id": ["33"]},
        {"id": "33", "related_id": [16, 17, 18]},
        {"id": 16},
        {"id": 17},
        {"id": 18}
       ],
```

(continues on next page)

(continued from previous page)

```
"cross": {"33": [[22,1,16,1], [22,2,16,2], [22,3,16,3], [22,4,16,4],[22,5,16,5],
                 [16,16,17,16], [16,17,17,17], [16,18,17,18], [16,19,17,19], [16,20,17,20],
                 [16,21,17,21], [16,22,17,22], [16,23,17,23],[16,24,17,24],
                 [16,11,18,11], [16,12,18,12], [16,13,18,13],[16,14,18,14],
                 [22,10,18,1], [22,11,18,2], [22,12,18,3], [17,11,18,5], [17,12,18,6], [17,13,18,
↪7],
                 [17,14,18,8],[22,6,17,1], [22,7,17,2], [22,8,17,3], [22,9,17,4]]
        }
}
```

2.2.2 ENVIRONMENT config - testcases/config/env/*.json

Environment config describes all allowed devices for setups. Which devices will be used in current run is defined in the setup configuration.

Environment configuration is a list of dictionaries. Each dictionary is a record for one device in SUT.

Each device type has its own fields, but entry_type, instance_type and id are obligatory for all devices.

For each DUT type, the following **entry_type** - **instance_type** variants are possible:

- **tg - traffic generators:**
  - ixiahl;
  - ixload;
  - trex
- **switch - switches:**
  - lxc - simswitch in lxc container for L3 testing;
  - <*chipname*> - real devices based on particular hardware
- **cross - cross connection devices:**
  - vlab - virtual cross connection tool;
  - static_ons - just stub for static connections
- **linux_host - Linux bases hosts:**
  - generic - real Linux host;
  - netns - Linux network namespase which emulates real Lunux Host behaviour
- **hub - Hubs:**
  - real;
  - simulated

Note: The id field values must be unique for all devices in config

In some cases, one device type needs a part of the config of another device. In this case you can use the related_id key (list type).

This key contains IDs of other devices, for which the config is necessary for the current one. For example {*entry_type: tg, instance_type: trex*} uses vlab interfaces for sending and sniffing packets, and therefore *related_id: [<vlab_id>]* should be added.

Real port names must be described only in appropriate device config. Then in other config and cross parts only port IDs from that device config will be used.

Examples:

TG:

- Ixia traffic generator:

```
[
  {"entry_type": "tg", "instance_type": "ixia", "id": 0,
   "ip_host": "X.X.X.X", "ports": [[1, 1, 1], [1, 1, 2], [1, 1, 3], [1, 1, 4], [1, 1, 5]]},
]
```

Linux Hosts:

- Network namespace:

```
[
  {"name": "Namespace_Simulated_2", "entry_type": "linux_host", "instance_type": "netns", "id":
↪1520,
   "ipaddr": "X.X.X.X", "ssh_user": "User", "ssh_pass": "pAsSwD",
   "ports": ["veth0", "veth1", "veth2"]
  },
]
```

- Real Linux Host:

```
[
  {"name": "Localhost1", "entry_type": "linux_host", "instance_type": "generic", "id": 999,
   "ipaddr": "localhost", "ssh_user": "User", "ssh_pass": "pAsSwD",
   "ports": ["lo"]
  },
]
```

Switch:

- Simulated switch in LXC container:

```
[
  {"entry_type": "switch", "instance_type": "lxc", "id": 1,
   "ip_host": "X.X.X.X", "ip_port": "8081",
   "ports_count": "32",
   "ports": [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
↪24],
   "related_id": [5]},
]
```

- Real switch:

```
[
  {"name": "some_real_device", "entry_type": "switch", "instance_type": "real", "id": 415,
   "ip_host": "192.168.1.20", "ip_port": "8081",
   "use_sshtun": 1, "sshtun_user": "xmluser", "sshtun_pass": "password", "sshtun_port": 22,
```

(continues on next page)

(continued from previous page)

```
    "default_gw": "192.168.1.1", "net_mask": "255.255.255.0",
    "pwboard_host": "192.168.1.100", "pwboard_port": "15", "halt": 0,
    "portserv_host": "192.168.1.101", "portserv_user": "root", "portserv_pass": "pass", "portserv_
↪tty": 15, "portserv_port": 2015,
    "telnet_loginprompt": "switch login:", "telnet_passprompt": "Password:", "telnet_user": "admin",
    "telnet_pass": "password", "telnet_prompt": "[admin@switch ~]$",
    "cli_user": "admin", "cli_user_passw": "admin" "cli_user_prompt": "Switch",
    "ports_count": "52",
    "ports": [40, 41, 43, 44, 45, 46, 47, 11, 12, 36, 37, 10, 23, 20, 39, 38],
    "related_id": [210]},
]
```

Cross:

- Vlab:

```
[
  {"entry_type": "cross", "instance_type": "vlab", "id": "2",
    "ip_host": "localhost", "ip_port": "8050",
    "ports": ["vlab0", "vlab1", "vlab2", "vlab3", "vlab4"],
    "tgmap": [0],
    "related_id": [1]},
]
```

2.3 Test execution

2.3.1 Launch Examples

Test cases can be run with supplying of py.test options:

1. All standard **py.test** options can be used for performing test cases.
2. General TAF specific options.
3. Additional ons-specific options.
4. Options provided by plugins.

Its easy to get all the information with short description using the command shown below:

```
$ py.test --help
```

General Options:

These options have to be registered in top level conftest.py file and they are checked by **testlib** modules.

--env=ENV	setting environment option, identify devices, *None* by default
--setup_file=SETUP	environment setup option, define environment configurations, *sim_lxc_simplified.json* by default
--loglevel=LOGLEVEL	logging level, print logging to console. *INFO* by default
--logfile=LOGFILE	logging file, store logs into file. *None* by default
--silent	do not print logging to console. *Default* - Disabled
--get_only	do not start environment, only connect to exists one. *False* by default
--leave_on	do not shutdown environment after the end of tests. *False* by default
--use_parallel_init	threads for simultaneous devices processing. *False* by default

Note: option –setup_file are obligatory

Additional Options:

These options are described and analyzed in top level conftest.py files.

--setup_scope=SETUP_SCOPE	Setup scope, select from session, module, class, function. *module* by default
--call_check=CALL_CHECK	check method for devices on test case call (*none*|*complete*|*fast*|*sanity_check_only*) *fast* by default
--teardown_check=TEARDOWN_CHECK	check method for devices on test case teardown (*none*|*complete*|*fast*|*sanity_check_only*) *sanity_check_only* by default

ONS-specific Options:

This options are analyzed in ONS specific **teslib** modules (e.g. switches module).

--fail_ctrl=FAIL_CTRL	device failure (*stop*|*restart*|*ignore*). *restart* by default
--build_path=BUILD_PATH	Path to build, */opt/simswitch* by default
--testenv	{none, simplified2, simplified3, simplified4, simplified5, golden, diamond, mixed} Verify environment before starting tests (*none* | *simplified2* | *simplified3* | *simplified4* | *simplified2* | *golden* | *diamond* | *mixed*) *none* by default

Plugin Options:

These options are provided by plugins.

--pidcheck_disable	disable pid check for test
--log_storage	{none, host, tms, both} where to store run logs (*none* | *host* | *tms* | *both*)
--log_type	{Failed, All} what kind of tests logs to store (*Failed* | *All*). *Failed* by default
--log_enable	{False, True} enable/disable log tool for test (*False* | *True*). *False* by default

You can **execute** the test cases using the following command:

```
/<host>/:~ /testcases$ sudo env PYTHONPATH=~taf/taf py.test --env=config/env/environment_examples.
↪jso --setup=config/setup/rr_simplified.json general/test_switch.py -m sanity --logdir=demo_logs -
↪-xml_html=demo.html
```

`env PYTHONPATH`	set up PYTHONPATH variable
`env`	provide path to the environment*.json file
`setup`	provide path to the setup*.json file

2.3.2 Test cases with traffic generators

Traffic generator is necessary to transmit and receive network traffic packets through the switch systems.

For the first you need to install 3rd party packages for Traffic Generator.

Installing 3rd party packages for IXIA traffic generator

IxTclHal API

TAF might require an IXIA TCL hal library and Python bindings for Tcl available in the Tkinter library depending on test scenarios. Ixia is one of traffic generators which generate traffic, or monitor the traffic transmitted by peers such as a switch. The latest Ixia software can be downloaded from Ixia website with eligible account.

Install Python Tk library:

```
$ sudo apt-get install python-tk
```

Get the IxOS binary (version X.XX) and install it (into **/opt** folder):

```
$ chmod +x ixosX.XX.XXX.XLinux.bin
$ sudo ./ixosX.XX.XXX.XLinux.bin
```

- Select Tcl version: **Tcl8.4**
- Choose install folder: **/opt/ixos**

Add symlink to **/opt/ixos/lib/ixTcl1.0** directory into **/usr/share/tcltk/tcl8.4/**:

```
$ sudo ln -s /opt/ixos/lib/ixTcl1.0 /usr/share/tcltk/tcl8.4/
```

Check if IXIA library was installed properly:

```
$ env IXIA_VERSION=X.XX tclsh
% package req IxTclHal
Tcl Client is running Ixia Software version: X.XX
X.XX
```

Note: Add export IXIA_VERSION=X.XX to your /etc/profile if necessary

Possible issues:

- Cant create directory **/opt/ixos/Logs**. Permission denied - create necessary directories manually with sudo:

```
$ sudo mkdir /opt/ixos/Logs && sudo mkdir /opt/ixos/Results
```

- In case youre getting Segmentation fault error - reinstall Tcl:

```
$ sudo apt-get purge tcl8.4
$ sudo apt-get install tcl
```

IxNetwork and HLTAPI

New (IxNetwork) API requires additional steps to be performed, as described in this section.

Install IxNetwork Tcl Client

Get the IxOS binary (version X.XX or better X.XX) and Install it:

```
$ chmod +x IxNetworkTclClientX.X.XXX.XXLinux.bin
$ sudo ./IxNetworkTclClientX.X.XXX.XXLinux.bin
```

Note: Install the client software where IxOS is already installed!

IxNetwork adds environment variables into **/etc/profile**, so reload profiles:

```
$ source /etc/profile && source ~/.bashrc
```

How to check that IxNetwork Tcl Client has been installed properly:

```
$ /opt/ixos/bin/ixnetwish
% package req ix_tc
3.20
% package req tbcload
1.4
% package req Thread
2.6.5
%
```

Configure HLTAPI

Install additional tcl pckages:

```
$ sudo apt-get install tcl8.4 tclx8.4
```

Note: HLTAPI can work properly only under tcl8.4

Check if configuration is correct:

```
$ env IXIA_VERSION=X.XX tclsh
% package req Ixia
Tcl X.X is installed on 64bit architecture.
Using products based on HLTSET138
Tcl Client is running Ixia Software version: X.XX
Loaded IxTclHal X.XX
Loaded IxTclServices X.XX
Loaded IxTclProtocol X.XX.XXX.XX
Loaded IxTclNetwork X.X.XXX.XX
HLT release X.XX.X.XXX
Loaded ixia_hl_lib-X.X
X.XX
%
```

IxLoad

Install IxLoad libraries to the same location with IxTclClient:

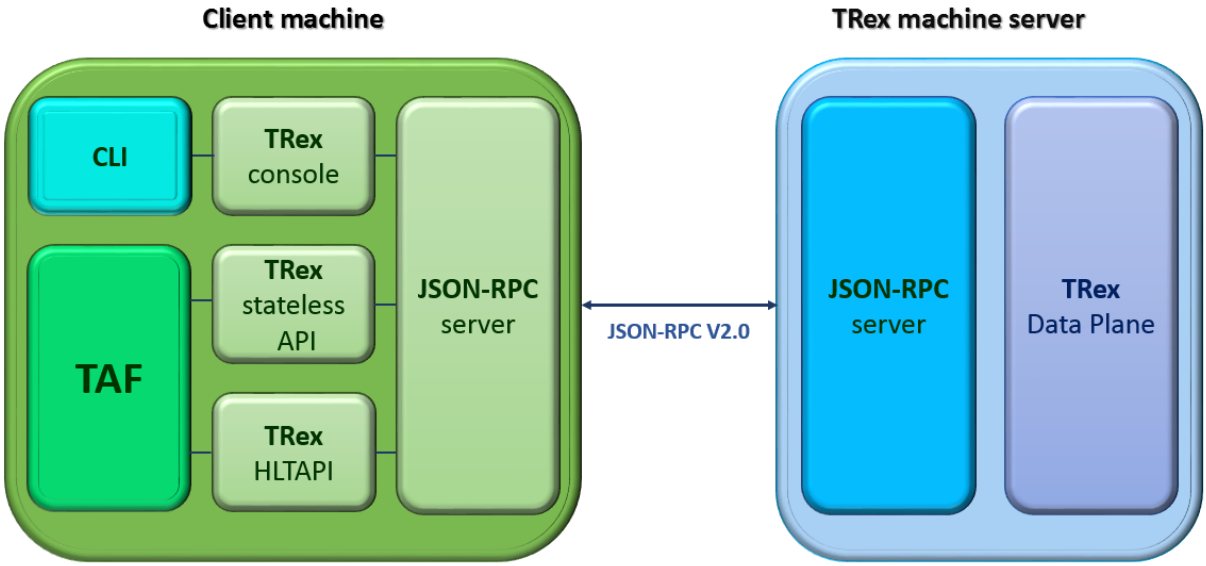
```
$ chmod +x IxLoadTclApiX.XX.XX.XLinux.bin
$ sudo ./IxLoadTclApiX.XX.XX.XLinux.bin
$ source /etc/profile && source ~/.bashrc
```

Check if IxLoad installed correctly:

```
$ env IXIA_VERSION=X.XX tclsh
% package req IxLoad
Tcl Client is running Ixia Software version: X.XX
X.XX.XX.X
```

Installing 3rd party packages for TRex traffic generator

TAF and TRex control plane



Download and installation TRex server

- Check environment:
  - hardware recommendation
  - supported Linux versions:
    - Fedora 18-21, 64-bit kernel (not 32-bit)
    - Ubuntu 16.04 LTS, 64-bit kernel (not 32-bit)
- Obtain the TRex package. Latest release:

```
$ mkdir trex
$ cd trex
$ wget --no-cache http://trex-tgn.cisco.com/trex/release/latest
$ tar -xzvf latest
```

To obtain a specific version(X.XX=Version number):

```
$ wget --no-cache http://trex-tgn.cisco.com/trex/release/vX.XX.tar.gz
```

Run TRex server

- Identify the ports:

```
$ cd trex
$ sudo ./dpdk_setup_ports.py {s
```

- Create simple configuration file:
  - Copy a basic configuration file from cfg folder:

```
$ cp  cfg/simple_cfg.yaml /etc/trex_cfg.yaml
```

  - Edit the configuration file:

```
$ vim  /etc/trex_cfg.yaml
port_limit      : 2         # this option can limit the number of port of the platform
version         : 2
interfaces      : ["03:00.0","03:00.1"] #the interfaces using ./dpdk_setup_ports.py {s
```

- Run TRex for the first time:

  Use the following command to begin operation of a 2x10Gb/sec TRex:

```
$ sudo ./t-rex-64 -i -c 2 {cfg /etc/ trex_cfg.yaml
```

Download and installation Client Machine

- Obtain the TRex client package:

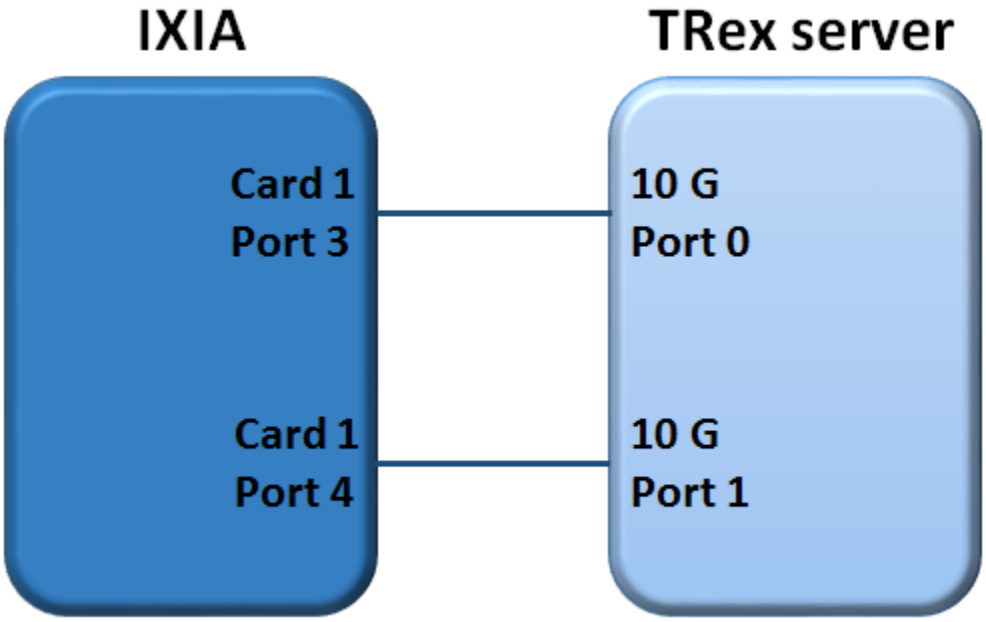
  Latest release:

```
$ mkdir trex_client
$ cd trex_client
$ wget --no-cache http://trex-tgn.cisco.com/trex/release/latest
$ tar -xzvf latest
$ cd vX.XX
# X.XX=Version number
$ tar -xzvf trex_client_v2.03.tar.gz
```

- Clone TAF and TCs repositories:

```
$ git clone https://github.com/taf3/taf.git
$ git clone https://github.com/taf3/testcases.git
```

- Create setup file and update env file based on diagram bellow:



TRex config in ENV JSON file:

```
{"name": "TRex", "entry_type": "tg", "instance_type": "trex", "id": "1001", "ipaddr": "X.X.X.X",
↪"ssh_user": "",  "ssh_pass": "", "ports": [0, 1]}
```

TRex setup file:

```
{"env": [
        {"id": "02", "port_list": [[[1, 1, 3], 10000], [[1, 1, 4], 10000]]},
        {"id": "1001"},
        {"id": "5", "related_id": ["02", "1001"]}],
"cross": { "5": [["02",1,"1001",1], ["02",2,"1001",2]] }}
```

2.3.3 Traffic Generator attributes and methods

All traffic generator objects have to be child of interface class **testlib::tg_template::GenericTG**.

Please read reference above for all available methods.

Note: Support of some methods could be unavailable in some tg types

Sniffing

Usage example:

```
tg_ports = [ports[('tg1', 'sw1')][1], ports[('tg1', 'sw1')][2]]
env.tg[1].start_sniff(tg_ports, sniffing_time=10, filter_layer="STP")
#...
# Other steps in case
```

(continues on next page)

(continued from previous page)

```
#...
data = env.tg[1].stop_sniff(tg_ports)
```

Description/Features

- Sniffing is performed in background.
- Stop_sniff() can be called separately for each port.
- Real packets capturing could be started with some delay. We found that different sniffer types dont start capturing right after receiving the command but with some delay. Based on experience this delay is set to 1.5 seconds. Therefore additional time sleep 1.5 seconds is introduced in start_sniff method, which are included in sniffing_time. You have to take this in to account and dont use sniffing time less then 3 seconds (5 is recommended). For benchmarking test cases you have to use special environment like IxNetwork or IxLoad.
- You can stop sniffers immediately using force=True parameter. By default stop_sniff returns control only after sniffing_time is elapsed.

Full methods references

- testlib::tg_template::GenericTG::start_sniff
- testlib::tg_template::GenericTG::stop_sniff

Packets filtering

You can check list of run-time filter layers in tg object attribute flt_patterns (testlib::packet_processor::PacketProcessor::flt_patterns).

If you need more complicated filter condition you should use post processing:

```
lfilter = lambda x: x.haslayer("STP") and x.get_lfield("STP", "rootid") == 4096
data1 = {}
for key in data.keys():
    data1[key] = filter(lfilter, data[key])
```

This filter returns the selection of STP BPDU packets with the rootid field equal to 4096.

Using comprehensive lists, the previous example becomes:

```
lfilter = lambda x: x.haslayer("STP") and x.get_lfield("STP", "rootid") == 4096
data1 = dict((key, filter(lfilter, data[key])) for key in data.keys())
```

Sending Packets

Trivial Packet Sending

Method **send_stream()** returns control after packet sending is finished:

```
tg_port = ports[('tg1', 'sw1')][1]
stream_id = env.tg[1].set_stream(packet_definition, count=5, inter=2, iface=tg_port)
env.tg[1].send_stream(stream_id)
```

Note: You should define streams at the beginning of the test case if possible. Then in steps, use only the send_stream method. This manner of packet sending improves test case performance as far as setting stream takes additional time for execution

Threaded Packet Sending

Method **start_streams()** returns control immediately after call. Packets will be sent in the background:

```
tg_port = ports[('tg1', 'sw1')][1]
stream_id_1 = env.tg[1].set_stream(packet_definition_1, count=5, inter=2, iface=tg_port)
stream_id_2 = env.tg[1].set_stream(packet_definition_2, count=15, inter=1, iface=tg_port)
env.tg[1].start_streams([stream_id_1, stream_id_2, ])
#...
# Some staff
#...
env.tg[1].stop_streams([stream_id_1, stream_id_2, ])
```

Warning: If youll try to start another stream without stopping the previous one, the first streams will be re-started. This behavior depends on TG type. Dont use it as a feature

The **set_stream()** method has several parameters that allow you to build a packet stream with incremented parameters.

Full methods references

- testlib::tg_template::GenericTG::set_stream
- testlib::tg_template::GenericTG::send_stream
- testlib::tg_template::GenericTG::start_streams
- testlib::tg_template::GenericTG::stop_streams

Working with Packets

Traffic Generator objects include set of methods to work with packets (please see testlib::packet_processor::PacketProcessor). But you also can access packet fields directly using internal packets methods.

Get Packet Field

- **TG method**
  - testlib::packet_processor::PacketProcessor::get_packet_field()

Get Packet Layer in Necessary Format

- **TG method**
  - testlib::packet_processor::PacketProcessor::get_packet_layer()

Check packet field value

- **TG method**
  - testlib::packet_processor::PacketProcessor::check_packet_field()
  - testlib::packet_processor::PacketProcessor::check_packet_field_multilayer()

Get packet dictionary

Reverse packet building: pypacker packet to packet dictionary.

- **TG method**
  - testlib::packet_processor::PacketProcessor::packet_dictionary()

Packet fragmenting and assembling

- **TG method**
  - testlib::packet_processor::PacketProcessor::packet_fragment
  - testlib::packet_processor::PacketProcessor::assemble_fragmented_packets

Statistics

Traffic Generator object contains number of methods to work with statistics.

- testlib::tg_template::GenericTG::clear_statistics()
- testlib::tg_template::GenericTG::get_sent_frames_count()
- testlib::tg_template::GenericTG::get_received_frames_count()
- testlib::tg_template::GenericTG::get_filtered_frames_count()
- testlib::tg_template::GenericTG::get_uds_3_frames_count()
- testlib::tg_template::GenericTG::get_qos_frames_count()

Note: Different types of TG could dont have support of some counters. Read appropriate tg type docs

2.3.4 Example of the test case with traffic generator

Sample Test:

```
class TestLinks(object):

    def test_links(self, env):
        """ Test links between Trex TG and IXIA"""
        # Define TGs
        ixia = env.tg[1].id
        trex = env.tg[2].id
        # Define active ports and packet
        ports = env.get_ports()
        packet_definition = (
             {"Ethernet": {"dst": "ff:ff:ff:ff:ff:ff",
                           "src": "00:00:00:00:00:02"}},
             {"IP": {"src": 'X.X.X.X', "dst": 'X.X.X.X'}})
        packet_count = 1
        # Set traffic stream on TRex
        stream_id = env.tg[2]. sey_stream(
                   packet_definition,
                   count=packet_count,
                   iface=ports[(trex, ixia)][1],
                   adjust_size=True)
```

(continues on next page)

(continued from previous page)

```
        # Start sniff on Ixia
        env.tg[1].start_sniff([ixia, trex)][1])
        # Send packet
        env.tg[2].start_streams([stream_id])
        time.sleep(1)
        # Stop sniff
        data = env.tg[1].stop_sniff([ports[(ixia, trex)][1]])
        # Verify that packet was received
        assert len(data[ports[(ixia, trex)][1]]) == 1
```

Example TAF command line, run in the testcase directory:

```
$ env PYTHONPATH=../taf/taf:<path to Trex client library  /trex_client/stl/ or use $TREX_CLIENT_
↪LIB> py.test --env=config/env/environment_examples.json --setup=config/setup/trex_ixia_
↪simplified.json testcases/test_links.py --loglevel=DEBUG
```

2.4 Contribution guidelines

2.4.1 Create an Issue

You can create an issue by the following link https://github.com/taf3/taf/issues if you:

- find a bug in a **TAF** project
- have trouble following the documentation
- have a question about the project

For more information on how issues work, check out GitHub Issues guide .

2.4.2 Code contribution

Pull Request

You can create a pull request by the following link https://github.com/taf3/taf/pulls if you are able to:

- patch the bug
- add the new feature

Before you will do pull request you have to:

- understand the license
- sign a Contributor Licence Agreement (CLA) if required

Note: All licenses and copyrights must be correct, and all code and information must have been pre-approved for public release.

GitHub workflow

1. Dev makes github account

2. Dev forks taf to repo in personal github account (e.g. github.com/rbbratta/taf)
3. Dev uploads new patches to personal taf repo as topic branch
4. Dev makes pull request to taf3/taf for topic branch
5. Travis-ci runs unittests on pull request
6. We review pull request in weekly meeting, leave comments on GitHub
7. Merge to GitHub
8. Internal Jenkins polls GitHub and builds internal TAF Docker image for Berta
9. We test internally with GitHub taf3 Docker image

For more information on how to create pull request, check out GitHub https://help.github.com/categories/collaborating-with-issues-and-pull-requests/ .

2.5 Development guide

2.5.1 TAF code naming convention

The heart of the design of python project is its high level of readability. One reason for code to be easily readable and understood is following set of code style guidelines. It is always advisable to maintain consistency in naming standards. This document describes the nomenclature suggested for use in TAF. TAF developers requires reading at PEP 8 – Style Guide for Python Code , and TAF library is conservative and requires limiting lines to 99 characters (and docstrings/comments to 92) although it exceeds PEP 8 standard. 99 characters will give enough to review side-by-side with multiple files, and visualize the difference between changes well in code review tools. Only exception is in writing function name in testcases which can exceed this limit as long as its helpful to understand the testcase.

Directory

- Referred to as packages in python project
- Short and all lowercase names
- Use of underscores is discouraged but can be used for better readability:

  e.g.: testlib, sanity_tests

File Names

- Referred to as modules in python project

Python Files

- Short and all lowercase names
- Use underscores for better readability
- Since module names are mapped to file names, and some file systems truncate long names, it is important that module names should be chosen to be fairly short

  e.g.: test_fdb.py

Data Files

Currently followed patterns for different data files are as follows:

- **Json**
  - lower_case_with_underscore

    e.g.: synapsert_client.json
  - CapsWords

    e.g.: StormControl.json
  - CapsWords_with_underscore

    e.g.: RouteTable_Prem
  - Startwithcapsletter

    e.g.: Fdb.json
  - lowerPlusCaps

    e.g.: ifType.json
- **TCL**
  - Lower_case_with_underscore
- **XML**
  - CapsWords
- **Dox**
  - lower_case_with_underscore
- **Java**
  - CapsWords
- **Vm**
  - lower_case_with_underscore
- **Txt**
  - lower_case_with_underscore

Note: DO NOT follow mixed standard mentioned above. It is always recommended to use lower_case_with_underscores for all file formats

---

Class Names

- Cap Words convention:

  e.g.: TestAclCopyToCpuAction

Some of the bad examples that are existing in TAF are as follows:

- Lowercase all

  e.g.: cv
- lowerThenUpper

  e.g.: helpersUI
- lower_with_underscores

e.g.: compare_color

- Beginning with underscore

  e.g.: _EncryptAndVerify

- Caps_With_Underscore

  e.g.: DHCP6_Decline

Functions

Test cases

- Test case function names should be lowercase, with words separated by underscores as necessary to improve readability, and it must starts with word test

  e.g.: test_name_lowercase_with_underscores

Sub module functions

- Starts with _

  e.g.: _single_leading_underscore

Class functions

- Class function names should be lowercase, with words separated by underscores as necessary to improve readability

  e.g.: lowercase_with_underscores

Pytest configuration functions

- Pytest configuration function names should be lowercase, with words separated by underscores as necessary to improve readability.
- Present in taf/plugins.
- Must start with word pytest

  e.g.: pytest_name_lowercase_with_underscores

Constants and Variables

Constants

- Capital letters with underscores separating words.

  e.g.: MAX_OVERFLOW, TOTAL

Variables

- Should be lowercase, with words separated by underscores.
- Global variables, attributes of the class and instance variables come under this category.

Arguments

Class

- Lowercase and can use underscores for better readability.
- Always use object for the first argument if required.

User-defined Methods

- Lowercase and can use underscores for better readability.
- Always use self for the first argument to instance methods.
- Always use cls for the first argument to class methods.

User-defined Functions

- Lowercase and can use underscores for better readability.

Docstring

All files, classes, class methods and first level functions must have properly created docstrings. Note that type syntax in Python docstrings is Google style .

Google style tends to be easier to read for short and simple docstrings.

File

Each python file in TAF should contain a header where the main information about the file is stored:

- copyright
- licence information
- file name
- summary
- note with example of module usage in tests (optionally)

In accordance to Google style of docstrings should look as following example or Sphinx example:

Example:

```
# Copyright (c) 2011 - 2016, Intel Corporation.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

"""``dpdk.py``

Class for dpdk operations

Note:
    Examples of dpdk usage in tests::

        inst.ui.dpdk.modify_iface_status(bind_action='bind', ifaces=["0000:01:00.0", "01:00.0"],
                                         drv='igb_uio', force=False, show_status=True)

"""
```

Class

Create a class with appropriate docstring. Following keywords can be used:

Note	An optional section that provides additional information about the code, possibly including a discussion of the algorithm
Example	Sections support any reStructuredText formatting, including literal blocks:: or doctest format to mention any coding samples

Example:

```
class NoErrArgumentParser(argparse.ArgumentParser):
    """ArgumentParser class that handle only predefined for an instance options.

    Note:
        The original ArgumentParser class raises an error if handle unknown option.
        But py.test have it's own options and it's own custom parser and if ArgumentParser find them it raises an error.
        Using this class allows not to define all possible options in each module that uses ArgumentParser.

    Examples::

            def parse_args(self, *args, **kwargs):
                if len(args) > 0:
                    args_to_parse = args[0]
                else:
                    args_to_parse = sys.argv[1:]
                new_args_to_parse = []

    """
```

Function

Create a test case function with appropriate docstring. Sub-functions inside first level functions don‘t need to contain docstrings as far as they arent designed for any external calls. Ignore pylint messages.

In case you wish to create a docstring, following keywords can be used:

Args	description of the function arguments, keywords and their respective types
Returns	explanation of the returned values and their types
Raises	an optional section detailing which errors get raised and under what conditions
Yields	explanation of the yielded values and their types

Example function docstrings with Returns key:

```
def __get__(self, instance, owner):
    """This method is called from class.

    Args:
        owner (owner):  class instance.

    Returns:
        logging.LoggerAdapter:  logger adaptor.

    Raises:
```

(continues on next page)

(continued from previous page)

```
        KeyError: Cannot connect to logger adaptor.

    """
    if self.for_exception:
        caller_frame = inspect.stack()[2]
        module_name = inspect.getmodulename(caller_frame[1])
        func_name = caller_frame[3]
        try:
            class_name = caller_frame[0].f_locals["self"].__class__.__name__
        except KeyError:
            class_name = ""
        _logger_adaptor = self._get_logger(module_name, class_name, func_name)
    else:
        _logger_adaptor = self._get_logger(owner.__module__, owner.__name__)
    return _logger_adaptor
```

Example function docstrings with Yields key:

```
def parse_table_vlan(self, vlan_table):
    """Parses the vlan table.

    This needs to be a loop because previous the table
    is built based on previous entries.

    Args:
        vlan_table (list[str] | iter()):  List of vlan raw output

    Yields:
        iter(): A dictionary containing the portId, vlanId, and tagged state for each vlan

    """
    for row in vlan_table:
        match = re.search(
            r"(?P<portId>\S*\d+)?\s*(?P<vlanId>\d+)\s*(?P<pvid>PVID)?\s*(?:Egress)?\s*(?P<tagged>
↪\D+)?", row)
        if match:
            row = match.groupdict()
            row['vlanId'] = int(row['vlanId'])
            if row['tagged'] is None:
                row['tagged'] = 'Tagged'
            row['pvid'] = (row['pvid'] == 'PVID')
            if row['portId'] is not None:
                # Set portId on the first line and use that value for following lines
                row['portId'] = self.name_to_portid_map[row['portId']]
                port_id = row['portId']
            else:
                # This row doesn't have a portId because it implicitly uses the previous
                row['portId'] = port_id
            yield row
```

2.5.2 Test Case Structure

A group of test cases will be written in a python file which we call test suite. The name of the file should:

- be unique;

- start with *test_*;
- contain clear information about test suite (e.g. feature, setup, table name, etc.).

Test suite is divided into the following separate parts:

- header;
- imports block;
- additional functions (optional);
- test class;
- internal test class methods;
- test cases.

Header

Each test case python file in TAF3 (testcases directory) should contain a header with contains following information:

- copyright
- licence information
- file name
- summary
- note (contain information what following test case are tested)

```
# Copyright (c) 2011 - 2016, Intel Corporation.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

"""``test_vlan.py``

`Test Vlan implementation`

Note:
    Following test cases are tested:
     1. Verify that static VLAN can be created.
     2. Verify that static VLAN can be deleted and default VLAN cannot be deleted.

"""
```

Note: File header should NOT contain below:

1 #!/usr/bin/env python

Import

Import section has the following rules and sequence in TAF python code:

- import standard module (e.g., os, time);
- import 3rd-party libraries (e.g., pytest);
- import framework-specific libraries (e.g., from testlib import helpers);
- each section of above import group has to be separated by a blank line.

Example:

```
import time
import os

import pytest

from testlib import helpers
from testlib import loggers
```

Developing Suite Class

Create class with unique name per suite (with appropriate docstring).

Note: Do not use any of Unittest style methods for py.test test cases. All necessary fixtures/setup/teardowns have to be defined using py.test features

---

Class name should start with Test. Class decorators should contain the following information:

- full cross connection setup name;
- information about premium functionality (optional);
- information about features that are tested;
- list of platform in case test suite/case is platform dependent (optional);
- mark to skip pidchecker plugin (optional).

Example of test suite docstrings:

```
@pytest.mark.simplified
@helpers.run_on_platforms(["lxc", ])
@pytest.mark.skip_pidcheck("snmpd")
@pytest.mark.acl
@pytest.mark.lag
class TestRSTPSimplified(object):
"""Suite for testing custom feature.

"""
```

Example of test case functions docstrings:

Write a summary of the particular test case which should explain actual devices behavior.

Describe test steps of the particular test case.

```
def test_bpdu_packet_format(self, env):
    """Verify that BPDU packets sent by switch are correctly formatted.

    Steps:
        - # Capture BPDU frames from the DUT
        - # Verify BPDU frames are correctly formatted

    """
```

Its recommended to register all your markers in pytest.ini file.

```
# content of pytest.ini
[pytest]
markers =
    simplified: mark a tests which have to be execudted on "simplified" setup.
```

The following setups are allowed: simplified, golden, and diamond.

Class Methods and Variables

This section contains internal variables and help methods used in the particular test suite.

Section should start with following comment separated with a blank line:

```
# Attributes and Properties
```

Then, class attributes should contain short inline description:

```
tp_id = 0x9100
tagged = "Tagged"
untagged = "Untagged"
```

Class method should have a docstring with following parts:

- summary with method description;
- parameters with name and description (optional);
- return value description (optional);
- usage examples (optional).

2.6 Continuous integration

Continuous integration can help catch bugs by running your tests automatically. The main goal is to eliminate the long and tedious integration process, the work that you normally have to do between versions final development stage and its deployment in production. A **continuous integration (CI)** process is highly recommended and is extremely useful in ensuring that your application stays functional. TAF project uses open source Travis continuous integration service.

2.6.1 TAF Travis CI job

- TAF project notifies Travis whenever pull request is submitted or updated.
- TAF Travis job is configured via .travis.yml and run script project_checker.py.

Note: .travis.yml file is located in the TAF root directory

project_checker.py file is located in TAF ci branch

TAF Travis job performs the following verification steps:

1. Run flake8 tool
2. Run pylint tool
3. Run taf/unittests

TAF Travis build status and logging messages you can find by the following link - https://travis-ci.org/taf3/taf .

2.6.2 Code Errors that trigger -1 verified

```
FLAKE8_FATAL_ERRORS = {
 "E101", # indentation contains mixed spaces and tabs
 "E111", # indentation is not a multiple of four
 "E112", # expected an indented block
 "E113", # unexpected indentation
 "E114", # indentation is not a multiple of four (comment)
 "E115", # expected an indented block (comment)
 "E116", # unexpected indentation (comment)
 "E711", # (^) comparison to None should be 'if cond is None:'
 "E712", # (^) comparison to True should be 'if cond is True:' or 'if cond:'
 "E713", # test for membership should be 'not in'
 "E714", # test for object identity should be 'is not'
 "E721", # do not compare types, use 'isinstance()'
 "E731", # do not assign a lambda expression, use a def
 "W191", # indentation contains tabs
 "W601", # .has_key() is deprecated, use 'in'
 "W602", # deprecated form of raising exception
 "W603", # '<>' is deprecated, use '!='
 "W604", # backticks are deprecated, use 'repr()'
 "F403", # 'from module import *' used; unable to detect undefined names
 "F821", # undefined name name
 "F822", # undefined name name in __all__
 "F831", # duplicate argument name in function definition
 "N804", # first argument of a classmethod should be named 'cls'
 "N805", # first argument of a method should be named 'self'
 "N811", # constant imported as non constant
 "N812", # lowercase imported as non lowercase
 "N813", # camelcase imported as lowercase
 "N814", # camelcase imported as constant
}
```

```
PYLINT_FATAL_ERRORS = {
 "C0121",  # Missing required attribute "%s"
 "C0202",  # Class method %s should have cls as first argument
 "C0203",  # Metaclass method %s should have mcs as first argument
 "C0204",  # Metaclass class method %s should have %s as first argument
 "C1001",  # Old-style class defined.

 "E0001",  # (syntax error raised for a module; message varies)
```

(continues on next page)

(continued from previous page)

```
    "E0011",  # Unrecognized file option %r
    "E0012",  # Bad option value %r
    "E0100",  # __init__ method is a generator
    "E0101",  # Explicit return in __init__
    "E0102",  # %s already defined line %s
    "E0103",  # %r not properly in loop
    "E0104",  # Return outside function
    "E0105",  # Yield outside function
    "E0106",  # Return with argument inside generator
    "E0107",  # Use of the non-existent %s operator
    "E0108",  # Duplicate argument name %s in function definition
    "E0202",  # An attribute affected in %s line %s hide this method
    "E0203",  # Access to member %r before its definition line %s
    "E0211",  # Method has no argument
    "E0213",  # Method should have "self" as first argument
    "E0221",  # Interface resolved to %s is not a class
    "E0222",  # Missing method %r from %s interface
    "E0235",  # __exit__ must accept 3 arguments: type, value, traceback
    "E0501",  # Old: Non ascii characters found but no encoding specified (PEP 263)
    "E0502",  # Old: Wrong encoding specified (%s)
    "E0503",  # Old: Unknown encoding specified (%s)
    "E0601",  # Using variable %r before assignment
    "E0602",  # Undefined variable %r
    "E0603",  # Undefined variable name %r in __all__
    "E0604",  # Invalid object %r in __all__, must contain only strings
    "E0611",  # No name %r in module %r
    "E0701",  # Bad except clauses order (%s)
    "E0702",  # Raising %s while only classes, instances or string are allowed
    "E0710",  # Raising a new style class which doesn't inherit from BaseException
    "E0711",  # NotImplemented raised - should raise NotImplementedError
    "E0712",  # Catching an exception which doesn\'t inherit from BaseException: %s
    "E1001",  # Use of __slots__ on an old style class
    "E1002",  # Use of super on an old style class
    "E1003",  # Bad first argument %r given to super()
    "E1004",  # Missing argument to super()
    "E1101",  # %s %r has no %r member
    "E1102",  # %s is not callable
    "E1103",  # %s %r has no %r member (but some types could not be inferred)
    "E1111",  # Assigning to function call which doesn't return
    "E1120",  # No value passed for parameter %s in function call
    "E1121",  # Too many positional arguments for function call
    "E1122",  # Old: Duplicate keyword argument %r in function call
    "E1123",  # Passing unexpected keyword argument %r in function call
    "E1124",  # Parameter %r passed as both positional and keyword argument
    "E1125",  # Old: Missing mandatory keyword argument %r
    "E1200",  # Unsupported logging format character %r (%#02x) at index %d
    "E1201",  # Logging format string ends in middle of conversion specifier
    "E1205",  # Too many arguments for logging format string
    "E1206",  # Not enough arguments for logging format string
    "E1300",  # Unsupported format character %r (%#02x) at index %d
    "E1301",  # Format string ends in middle of conversion specifier
    "E1302",  # Mixing named and unnamed conversion specifiers in format string
    "E1303",  # Expected mapping for format string, not %s
    "E1304",  # Missing key %r in format string dictionary
    "E1305",  # Too many arguments for format string
    "E1306",  # Not enough arguments for format string
```

(continues on next page)

(continued from previous page)

```
    "E1310",  # Suspicious argument in %s.%s call

    "F0001",  # (error prevented analysis; message varies)
    "F0002",  # %s: %s (message varies)
    "F0010",  # error while code parsing: %s

    "R0401",  # Cyclic import (%s)
    "W0102",  # Dangerous default value %s as argument
    "W0109",  # Duplicate key %r in dictionary
    "W0121",  # Use raise ErrorClass(args) instead of raise ErrorClass, args.
    "W0122",  # Use of exec
    "W0150",  # %s statement in finally block may swallow exception
    "W0199",  # Assert called on a 2-uple. Did you mean \'assert x,y\'?
    "W0211",  # Static method with %r as first argument
    "W0221",  # Arguments number differs from %s method
    "W0233",  # __init__ method from a non direct base class %r is called
    "W0234",  # iter returns non-iterator
    "W0311",  # Bad indentation. Found %s %s, expected %s
    "W0331",  # Use of the <> operator
    "W0332",  # Use of "l" as long integer identifier
    "W0333",  # Use of the `` operator
    "W0401",  # Wildcard import %s
    "W0402",  # Uses of a deprecated module %r
    "W0404",  # Reimport %r (imported line %s)
    "W0410",  # __future__ import is not the first non docstring statement
    "W0406",  # Module import itself
    "W0512",  # Cannot decode using encoding "%s", unexpected byte at position %d
    "W0601",  # Global variable %r undefined at the module level
    "W0602",  # Using global for %r but no assigment is done
    "W0604",  # Using the global statement at the module level
    "W0614",  # Unused import %s from wildcard import
    "W0622",  # Redefining built-in %r
    "W0623",  # Redefining name %r from %s in exception handler
    "W0631",  # Using possibly undefined loop variable %r
    "W0632",  # Possible unbalanced tuple unpacking with sequence%s:
    "W0633",  # Attempting to unpack a non-sequence%s
    "W0701",  # Raising a string exception
    "W0702",  # No exception type(s) specified
    "W0711",  # Exception to catch is the result of a binary "%s" operation
    "W0712",  # Implicit unpacking of exceptions is not supported in Python 3
    "W1001",  # Use of "property" on an old style class
    "W1111",  # Assigning to function call which only returns None
    "W1201",  # Specify string format arguments as logging function parameters
    "W1300",  # Format string dictionary key should be a string, not %s
    "W1301",  # Unused key %r in format string dictionary
    "W1501",  # "%s" is not a valid mode for open.
```

}

Chapter 3

Indices and tables

- genindex
- modindex