streamsx.testing Documentation

Release 0.3.1

IBMStreams

Jul 12, 2019

Contents

CHAPTER 1

Overview

Testing for IBM Streams SPL and Python applications.

CHAPTER 2

unittest integration

- Test streams by placing conditions on streams in an application, such as this stream must receive at least 100 tuples.
- Allow a test to be easily executed in different environments, such as standalone and against the public cloud service.

CHAPTER 3

nose integration

- Plugins to allow configuration changes when running tests using *nosetests* without modifying the test code.

`streamsx.testing`	IBM Streams application testing.
`streamsx.testing.nose`	Nose plugins for IBM Streams application testing.

3.1 streamsx.testing

IBM Streams application testing.

3.1.1 Overview

Testing of an application, sub-graph or operator is performed by building a Python topology that invokes the element under test in a standard Python unittest. The element under test can be a SPL application, sub-graph, operator or a Python application, sub-graph or single transformation. See *Testing overview*.

Testing of SPL functions is performed by declaring series of input data and expected output. See *SPL function testing overview*.

Python is a natural choice for testing of SPL applications as tests can be written simply and executed immediately without a compilation step. By use of the standard Python unittest existing tools such as `nosetets` can be used to run tests, produce reports and integrate with continuous integration tools such as Jenkins.

3.1.2 Testing overview

Allows testing of a streaming application by creation conditions on streams that are expected to become valid during the processing. *Tester* is designed to be used with Python's *unittest* module.

A complete application may be tested or fragments of it, for example a sub-graph can be tested in isolation that takes input data and scores it using a model.

Supports execution of the application on `STREAMING_ANALYTICS_SERVICE`, `DISTRIBUTED` or `STANDALONE`.

A *`Tester`* instance is created and associated with the `Topology` to be tested. Conditions are then created against streams, such as a stream must receive 10 tuples using *`tuple_count()`*.

Here is a simple example that tests a filter correctly passes tuples with values greater than 5:

```
import unittest
from streamsx.testing import Tester
from streamsx.topology.topology import Topology

class TestSimpleFilter(unittest.TestCase):

    def setUp(self):
        # Sets self.test_ctxtype and self.test_config
        Tester.setup_streaming_analytics(self)

    def test_filter(self):
        # Declare the application to be tested
        topology = Topology()
        s = topology.source([5, 7, 2, 4, 9, 3, 8])
        s = s.filter(lambda x : x > 5)

        # Create tester and assign conditions
        tester = Tester(topology)
        tester.contents(s, [7, 9, 8])

        # Submit the application for test
        # If it fails an AssertionError will be raised.
        tester.test(self.test_ctxtype, self.test_config)
```

A stream may have any number of conditions and any number of streams may be tested.

A *`local_check()`* is supported where a method of the unittest class is executed once the job becomes healthy. This performs checks from the context of the Python unittest class, such as checking external effects of the application or using the REST api to monitor the application.

A test fails-fast if any of the following occur:

- Any condition fails. E.g. a tuple failing a *`tuple_check()`*.
- The *`local_check()`* (if set) raises an error.
- **The job for the test:**
  - Fails to become healthy.
  - Becomes unhealthy during the test run.
  - Any processing element (PE) within the job restarts.

A test timeouts if it does not fail but its conditions do not become valid. The timeout is not fixed as an absolute test run time, but as a time since "progress" was made. This can allow tests to pass when healthy runs are run in a constrained environment that slows execution. For example with a tuple count condition of ten, progress is indicated by tuples arriving on a stream, so that as long as gaps between tuples are within the timeout period the test remains running until ten tuples appear.

Note: The test timeout value is not configurable.

Note: The submitted job (application under test) has additional elements (streams & operators) inserted to implement the conditions. These are visible through various APIs including the Streams console raw graph view. Such elements are put into the *Tester* category.

Note: `Tester` is an import of *streamsx.topology.tester.Tester*.

3.1.3 SPL function testing overview

SPL functions can tested using `FnTester` by providing series of input values and the expected function return values. Functions under test may be SPL or SPL native functions (implemented in Java or C++).

class streamsx.testing.**Tester**(*topology*)

Testing support for a Topology.

Allows testing of a Topology by creating conditions against the contents of its streams.

Conditions may be added to a topology at any time before submission.

If a topology is submitted directly to a context then the graph is not modified. This allows testing code to be inserted while the topology is being built, but not acted upon unless the topology is submitted in test mode.

If a topology is submitted through the test method then the topology may be modified to include operations to ensure the conditions are met.

Warning: For future compatibility applications under test should not include intended failures that cause a processing element to stop or restart. Thus, currently testing is against expected application behavior.

Parameters **topology** – Topology to be tested.

add_condition(*stream*, *condition*)

Add a condition to a stream.

Conditions are normally added through `tuple_count()`, `contents()` or `tuple_check()`.

This allows an additional conditions that are implementations of `Condition`.

Parameters

- **stream** (`Stream`) – Stream to be tested.
- **condition** (`Condition`) – Arbitrary condition.

Returns stream

Return type Stream

contents(*stream*, *expected*, *ordered=True*)

Test that a stream contains the expected tuples.

Parameters

- **stream** (`Stream`) – Stream to be tested.
- **expected** (`list`) – Sequence of expected tuples.
- **ordered** (`bool`) – True if the ordering of received tuples must match expected.

Returns stream

Return type Stream

eventual_result(*stream*, *checker*)

Test a stream reaches a known result or state.

Creates a test condition that the tuples on a stream eventually reach a known result or state. Each tuple on *stream* results in a call to `checker(tuple_)`.

The return from *checker* is handled as:

- `None` - The condition requires more tuples to become valid.
- *true value* - The condition has become valid.
- *false value* - The condition has failed. Once a condition has failed it can never become valid.

Thus *checker* is typically stateful and allows ensuring that condition becomes valid from a set of input tuples. For example in a financial application the application under test may need to achieve a final known balance, but due to timings of windows the number of tuples required to set the final balance may be variable.

Once the condition becomes valid any false value, except `None`, returned by processing of subsequent tuples will cause the condition to fail.

Returning `None` effectively never changes the state of the condition.

Parameters

- **stream** (*Stream*) – Stream to be tested.
- **checker** (*callable*) – Callable that returns evaluates the state of the stream with result to the result.

New in version 1.11.

static get_streams_version(*test*)

Returns IBM Streams product version string for a test.

Returns the product version corresponding to the test's setup. For `STANDALONE` and `DISTRIBUTED` the product version corresponds to the version defined by the environment variable `STREAMS_INSTALL`.

Parameters **test** (*unittest.TestCase*) – Test case setup to run IBM Streams tests.

local_check(*callable*)

Perform local check while the application is being tested.

A call to *callable* is made after the application under test is submitted and becomes healthy. The check is in the context of the Python runtime executing the unittest case, typically the callable is a method of the test case.

The application remains running until all the conditions are met and *callable* returns. If *callable* raises an error, typically through an assertion method from *unittest* then the test will fail.

Used for testing side effects of the application, typically with *STREAMING_ANALYTICS_SERVICE* or *DISTRIBUTED*. The callable may also use the REST api for context types that support it to dynamically monitor the running application.

The callable can use *submission_result* and *streams_connection* attributes from *Tester* instance to interact with the job or the running Streams instance. These REST binding classes can be obtained as follows:

- `Job` - `tester.submission_result.job`
- `Instance` - `tester.submission_result.job.get_instance()`

- `StreamsConnection` - `tester.streams_connection`

Simple example of checking the job is healthy:

```
import unittest
from streamsx.topology.topology import Topology
from streamsx.topology.tester import Tester

class TestLocalCheckExample(unittest.TestCase):
    def setUp(self):
        Tester.setup_distributed(self)

    def test_job_is_healthy(self):
        topology = Topology()
        s = topology.source(['Hello', 'World'])

        self.tester = Tester(topology)
        self.tester.tuple_count(s, 2)

        # Add the local check
        self.tester.local_check = self.local_checks

        # Run the test
        self.tester.test(self.test_ctxtype, self.test_config)


    def local_checks(self):
        job = self.tester.submission_result.job
        self.assertEqual('healthy', job.health)
```

Warning: A local check must not cancel the job (application under test).

Warning: A local check is not supported in standalone mode.

Parameters **callable** – Callable object.

static minimum_streams_version(*test*, *required_version*)

Checks test setup matches a minimum required IBM Streams version.

Parameters

- **test** (*unittest.TestCase*) – Test case setup to run IBM Streams tests.
- **required_version** (*str*) – VRMF of the minimum version the test requires. Examples are `'4.3'`, `4.2.4`.

Returns True if the setup fulfills the minimum required version, false otherwise.

Return type bool

static require_streams_version(*test*, *required_version*)

Require a test has minimum IBM Streams version.

Skips the test if the test's setup is not at the required minimum IBM Streams version.

Parameters

- **test** (*unittest.TestCase*) – Test case setup to run IBM Streams tests.
- **required_version** (*str*) – VRMF of the minimum version the test requires. Examples are `'4.3'`, `4.2.4`.

resets(*minimum_resets=10*)

Create a condition that randomly resets consistent regions. The condition becomes valid when each consistent region in the application under test has been reset *minimum_resets* times by the tester.

The resets are performed at arbitrary intervals scaled to the period of the region (if it is periodically triggered).

Note: A region is reset by initiating a request though the Job Control Plane. The reset is not driven by any injected failure, such as a PE restart.

Parameters **minimum_resets** (*int*) – Minimum number of resets for each region.

New in version 1.11.

run_for(*duration*)

Run the test for a minimum number of seconds.

Creates a test wide condition that becomes *valid* when the application under test has been running for *duration* seconds. Maybe be called multiple times, the test will run as long as the maximum value provided.

Can be used to test applications without any externally visible streams, or streams that do not have testable conditions. For example a complete application may be tested by runnning it for for ten minutes and use *local_check()* to test any external impacts, such as messages published to a message queue system.

Parameters **duration** (*float*) – Minimum number of seconds the test will run for.

static setup_distributed(*test*, *verbose=None*)

Set up a unittest.TestCase to run tests using IBM Streams distributed mode.

Requires a local IBM Streams install define by the `STREAMS_INSTALL` environment variable. If `STREAMS_INSTALL` is not set then the test is skipped.

The Streams instance to use is defined by the environment variables:

- `STREAMS_ZKCONNECT` - Zookeeper connection string (optional)
- `STREAMS_DOMAIN_ID` - Domain identifier
- `STREAMS_INSTANCE_ID` - Instance identifier

The user used to submit and monitor the job is set by the optional environment variables:

- `STREAMS_USERNAME` - User name defaulting to *streamsadmin*.
- `STREAMS_PASSWORD` - User password defaulting to *passw0rd*.

The defaults match the setup for testing on a IBM Streams Quick Start Edition (QSE) virtual machine.

Warning: `streamtool` is used to submit the job and requires that `streamtool` does not prompt for authentication. This is achieved by using `streamtool genkey`.

See also:

Generating authentication keys for IBM Streams

Two attributes are set in the test case:

- test_ctxtype - Context type the test will be run in.
- test_config - Test configuration.

Parameters

- **test** (*unittest.TestCase*) – Test case to be set up to run tests using Tester
- **verbose** (*bool*) – If *true* then the `streamsx.topology.test` logger is configured at `DEBUG` level with output sent to standard error.

Returns: None

static setup_standalone(*test*, *verbose=None*)

Set up a unittest.TestCase to run tests using IBM Streams standalone mode.

Requires a local IBM Streams install define by the `STREAMS_INSTALL` environment variable. If `STREAMS_INSTALL` is not set, then the test is skipped.

A standalone application under test will run until a condition fails or all the streams are finalized or when the *run_for()* time (if set) elapses. Applications that include infinite streams must include set a run for time using *run_for()* to ensure the test completes

Two attributes are set in the test case:

- test_ctxtype - Context type the test will be run in.
- test_config- Test configuration.

Parameters

- **test** (*unittest.TestCase*) – Test case to be set up to run tests using Tester
- **verbose** (*bool*) – If *true* then the `streamsx.topology.test` logger is configured at `DEBUG` level with output sent to standard error.

Returns: None

static setup_streaming_analytics(*test*, *service_name=None*, *force_remote_build=False*, *verbose=None*)

Set up a unittest.TestCase to run tests using Streaming Analytics service on IBM Cloud.

The service to use is defined by:

- VCAP_SERVICES environment variable containing *streaming_analytics* entries.
- service_name which defaults to the value of STREAMING_ANALYTICS_SERVICE_NAME environment variable.

If VCAP_SERVICES is not set or a service name is not defined, then the test is skipped.

Two attributes are set in the test case:

- test_ctxtype - Context type the test will be run in.
- test_config - Test configuration.

Parameters

- **test** (*unittest.TestCase*) – Test case to be set up to run tests using Tester
- **service_name** (*str*) – Name of Streaming Analytics service to use. Must exist as an entry in the VCAP services. Defaults to value of STREAMING_ANALYTICS_SERVICE_NAME environment variable.

- **force_remote_build** (*bool*) – Force use of the Streaming Analytics build service. If *false* and `STREAMS_INSTALL` is set then a local build will be used if the local environment is suitable for the service, otherwise the Streams application bundle is built using the build service.
- **verbose** (*bool*) – If *true* then the `streamsx.topology.test` logger is configured at `DEBUG` level with output sent to standard error.

If run with Python 2 the test is skipped, only Python 3.5 is supported with Streaming Analytics service.

Returns: None

test(*ctxtype*, *config=None*, *assert_on_fail=True*, *username=None*, *password=None*, *always_collect_logs=False*)

Test the topology.

Submits the topology for testing and verifies the test conditions are met and the job remained healthy through its execution.

The submitted application (job) is monitored for the test conditions and will be canceled when all the conditions are valid or at least one failed. In addition if a local check was specified using *local_check()* then that callable must complete before the job is cancelled.

The test passes if all conditions became valid and the local check callable (if present) completed without raising an error.

The test fails if the job is unhealthy, any condition fails or the local check callable (if present) raised an exception. In the event that the test fails when submitting to the *STREAMING_ANALYTICS_SERVICE* context, the application logs are retrieved as a tar file and are saved to the current working directory. The filesystem path to the application logs is saved in the tester's result object under the *application_logs* key, i.e. *tester.result['application_logs']*

Parameters

- **ctxtype** (*str*) – Context type for submission.
- **config** – Configuration for submission.
- **assert_on_fail** (*bool*) – True to raise an assertion if the test fails, False to return the passed status.
- **username** (*str*) – **Deprecated**
- **password** (*str*) – **Deprecated**
- **always_collect_logs** (*bool*) – True to always collect the console log and PE trace files of the test.

result
The result of the test. This can contain exit codes, application log paths, or other relevant test information.

submission_result
Result of the application submission from `submit()`.

streams_connection
Connection object that can be used to interact with the REST API of the Streaming Analytics service or instance.

Type StreamsConnection

Returns *True* if test passed, *False* if test failed if *assert_on_fail* is *False*.

Return type bool

Deprecated since version 1.8.3: `username` and `password` parameters. When required for a distributed test use the environment variables `STREAMS_USERNAME` and `STREAMS_PASSWORD` to define the Streams user.

tuple_check(*stream*, *checker*)

Check each tuple on a stream.

For each tuple `t` on *stream* `checker(t)` is called.

If the return evaluates to *False* then the condition fails. Once the condition fails it can never become valid. Otherwise the condition becomes or remains valid. The first tuple on the stream makes the condition valid if the checker callable evaluates to *True*.

The condition can be combined with *tuple_count()* with `exact=False` to test a stream map or filter with random input data.

An example of combining *tuple_count* and *tuple_check* to test a filter followed by a map is working correctly across a random set of values:

```
def rands():
    r = random.Random()
    while True:
        yield r.random()

class TestFilterMap(unittest.testCase):
# Set up omitted

    def test_filter(self):
        # Declare the application to be tested
        topology = Topology()
        r = topology.source(rands())
        r = r.filter(lambda x : x > 0.7)
        r = r.map(lambda x : x + 0.2)

        # Create tester and assign conditions
        tester = Tester(topology)
        # Ensure at least 1000 tuples pass through the filter.
        tester.tuple_count(r, 1000, exact=False)
        tester.tuple_check(r, lambda x : x > 0.9)


        # Submit the application for test
        # If it fails an AssertionError will be raised.
        tester.test(self.test_ctxtype, self.test_config)
```

Parameters

- **stream** (*Stream*) – Stream to be tested.
- **checker** (*callable*) – Callable that must evaluate to True for each tuple.

tuple_count(*stream*, *count*, *exact=True*)

Test that a stream contains a number of tuples.

If *exact* is *True*, then condition becomes valid when *count* tuples are seen on *stream* during the test. Subsequently if additional tuples are seen on *stream* then the condition fails and can never become valid.

If *exact* is *False*, then the condition becomes valid once *count* tuples are seen on *stream* and remains valid regardless of any additional tuples.

Parameters

- **stream** (*Stream*) – Stream to be tested.
- **count** (*int*) – Number of tuples expected.
- **exact** (*bool*) – *True* if the stream must contain exactly *count* tuples, *False* if the stream must contain at least *count* tuples.

Returns stream

Return type Stream

class streamsx.testing.**FnTester**(*name*)

SPL function tester.

Creates a holder for an SPL function under test.

Parameters **name** – SPL namespace qualified name of the function.

Simple examples

Example testing a function with a single parameter (`spl.math::abs`) with `int32` and `float64` values:

```
import unittest
from streamsx.testing import Tester, FnTester

class TestStandardFunctions(unittest.TestCase):

    def setUp(self):
        # Sets self.test_ctxtype and self.test_config
        Tester.setup_standalone(self)

    def test_abs(self):
        # Declare the tester
        tester = FnTester('spl.math::abs')

        # Setup a series of int64 values for testing
        args = [1,2,-3,0,-5]
        tester.series(args, [abs(i) for i in args], name='abs_int64')

        # Setup a series of float64 values for testing
        args = [0.5, 0.0, -4.5]
        tester.series(args, [abs(i) for i in args], name='abs_float64')

        # Execute the test
        tester.test(self)
```

Note: The function under test and its series are tested using a generated application run with *Tester*.

series(*args*, *expected*, *name=None*)

Declare a function test with a series of values.

Each value in *args* is passed into the function under test and the result expected to be the corresponding value in *expected*.

Each value in *args* is a simple value for functions that accept a single parameter. Otherwise each value is a `tuple` with the number of required parameters.

Each value in *expected* is a simple value for functions that return an SPL type that is not an SPL tuple. Otherwise each value is a `tuple` with the correct number of values for the returned tuple schema.

Multiple series may be created for a single instance of `FnTester`, typically using different data types accepted by the function.

The series tests are not executed until FnTester.test is called.

The series *name* can aid with diagnostics when debugging tests or functions to clearly indicate which series is failing.

Parameters

- **args** (*list*) – List of values to be passed into the function under test.
- **expected** (*list*) – List of expected results.
- **name** (*str*) – Name of series. Defaults to a generated name.

test(*test*, *assert_on_fail=True*, *always_collect_logs=False*)

Test the function.

Submits this function for testing and verifies all the series have the expected results.

The submitted job containing the series tests is monitored and will be canceled when all the series are valid or at least one failed.

The test passes if all series became valid.

The test fails if the job is unhealthy or any series fails.

In the event that the test fails the application logs are retrieved (when supported by the Streams instance) as a tar file and are saved to the current working directory. The filesystem path to the application logs is saved in the tester's result object under the *application_logs* key, i.e. *tester.result['application_logs']*

The test case *test* must have been setup with on of *Tester.setup_standalone()*, *Tester.setup_distributed()* or *Tester.setup_streaming_analytics()*.

Parameters

- **test** – Instance of `unittest.TestCase` running the function series.
- **assert_on_fail** (*bool*) – True to raise an assertion if the test fails, False to return the passed status.
- **always_collect_logs** (*bool*) – True to always collect the console log and PE trace files of the test.

Returns *True* if test passed, *False* if test failed if *assert_on_fail* is *False*.

Return type bool

3.2 streamsx.testing.nose

Nose plugins for IBM Streams application testing.

Classes

AddConfigurationPlugin()	Add arbitrary configuration to a test
DisableSSLVerifyPlugin()	Disable SSL certification verification.
JobConfigPlugin()	Job configuration plugin.
SkipStandalonePlugin()	Skip standalone tests.

class streamsx.testing.nose.**AddConfigurationPlugin**
Add arbitrary configuration to a test

Enabled with `--with-streamsx-add-config`.

This plugin adds arbitrary configuration items to a test's `test_config` dictionary by updating with the dictionary value supplied by the options.

These options must be set when using this plugin.

- `--streamsx-test-context CONTEXT` - Context that will have its configuration added to. Any test that runs with a different context will not have its
- `--streamsx-test-config CODE` - Code that is executed using built-in method *exec*. The execution must set the local variable `cfg` to a dictionary that will then be used as `test.test_config.update(cfg)` before the test is run.

Example:

```
nosetests --with-streamsx-add-config --streamsx-test-context STANDALONE --
↪streamsx-test-config "cfg = {'topology.keepArtifacts':True}"
```

configure(*options*, *conf*)
Configure the plugin and system, based on selected options.

The base plugin class sets the plugin to enabled if the enable option for the plugin (self.enableOpt) is true.

options(*parser*, *env=environ({'HOSTNAME': 'build-9365248-project-251772-streamsxtesting', 'PYPY_VERSION_35': 'pypy3.5-7.0.0', 'APPDIR': '/app', 'HOME': '/home/docs', 'OLDPWD': '/', 'CONDA_VERSION': '4.6.14', 'READTHEDOCS': 'True', 'READTHEDOCS_PROJECT': 'streamsxtesting', 'PATH': '/home/docs/checkouts/readthedocs.org/user_builds/streamsxtesting/envs/pypackage/bin:/home/docs/.pyenv/shims:/us 'LANG': 'C.UTF-8', 'DEBIAN_FRONTEND': 'noninteractive', 'BIN_PATH': '/home/docs/checkouts/readthedocs.org/user_builds/streamsxtesting/envs/pypackage/bin', 'PYTHON_VERSION_35': '3.5.7', 'READTHEDOCS_VERSION': 'pypackage', 'PYTHON_VERSION_27': '2.7.16', 'PYTHON_VERSION_36': '3.6.8', 'PWD': '/home/docs/checkouts/readthedocs.org/user_builds/streamsxtesting/checkouts/pypackage/docs/source', 'PYTHON_VERSION_37': '3.7.3', 'PYENV_ROOT': '/home/docs/.pyenv', 'DOCUTILSCONFIG': '/home/docs/checkouts/readthedocs.org/user_builds/streamsxtesting/checkouts/pypackage/docs/sourc*)
Register commandline options.

Implement this method for normal options behavior with protection from OptionConflictErrors. If you override this method and want the default –with-$name option to be registered, be sure to call super().

class streamsx.testing.nose.**DisableSSLVerifyPlugin**
Disable SSL certification verification.

Disables SSL certification when running distributed tests. This is useful when a test instance with a self-signed certificate, such as the IBM Streams Quick Start edition.

Enabled with `--with-streamsx-disable-ssl-verify`.

configure(*options*, *conf*)
Configure the plugin and system, based on selected options.

The base plugin class sets the plugin to enabled if the enable option for the plugin (self.enableOpt) is true.

class streamsx.testing.nose.**JobConfigPlugin**
Job configuration plugin.

Plugin that modifies the job configuration object for the application under test.

Enabled with `--with-streamsx-jco`.

These options are supported:

- `--streamsx-jco-default-tag=tag` - Sets the resource tag for the default host pool. The default host pool is where transformations/operators with explicit resource tags are assigned to and by default maps to the resource tag `application`.

configure(*options*, *conf*)
Configure the plugin and system, based on selected options.

The base plugin class sets the plugin to enabled if the enable option for the plugin (self.enableOpt) is true.

options(*parser*, *env=environ({'HOSTNAME': 'build-9365248-project-251772-streamsxtesting'*, *'PYPY_VERSION_35': 'pypy3.5-7.0.0'*, *'APPDIR': '/app'*, *'HOME': '/home/docs'*, *'OLDPWD': '/'*, *'CONDA_VERSION': '4.6.14'*, *'READTHEDOCS': 'True'*, *'READTHEDOCS_PROJECT': 'streamsxtesting'*, *'PATH': '/home/docs/checkouts/readthedocs.org/user_builds/streamsxtesting/envs/pypackage/bin:/home/docs/.pyenv/shims:/us*
'LANG': 'C.UTF-8', *'DEBIAN_FRONTEND': 'noninteractive'*, *'BIN_PATH': '/home/docs/checkouts/readthedocs.org/user_builds/streamsxtesting/envs/pypackage/bin'*, *'PYTHON_VERSION_35': '3.5.7'*, *'READTHEDOCS_VERSION': 'pypackage'*, *'PYTHON_VERSION_27': '2.7.16'*, *'PYTHON_VERSION_36': '3.6.8'*, *'PWD': '/home/docs/checkouts/readthedocs.org/user_builds/streamsxtesting/checkouts/pypackage/docs/source'*, *'PYTHON_VERSION_37': '3.7.3'*, *'PYENV_ROOT': '/home/docs/.pyenv'*, *'DOCUTILSCONFIG': '/home/docs/checkouts/readthedocs.org/user_builds/streamsxtesting/checkouts/pypackage/docs/sourc*
Register commandline options.

Implement this method for normal options behavior with protection from OptionConflictErrors. If you override this method and want the default –with-$name option to be registered, be sure to call super().

class streamsx.testing.nose.**SkipStandalonePlugin**
Skip standalone tests.

Automatically skips any tests that have been configured for standalone using `Tester.setup_standalone()`.

Enabled with `--with-streamsx-skip-standalone`.

configure(*options*, *conf*)
Configure the plugin and system, based on selected options.

The base plugin class sets the plugin to enabled if the enable option for the plugin (self.enableOpt) is true.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

n

s

Index