streamsx Documentation

Release 1.14.14

IBMStreams

Apr 22, 2020

CONTENTS

Python APIs for use with IBM® Streaming Analytics service on IBM Cloud and on-premises IBM Streams.

CHAPTER

ONE

PYTHON APPLICATION API FOR STREAMS

Module that allows the definition and execution of streaming applications implemented in Python. Applications use Python code to process tuples and tuples are Python objects.

SPL operators may also be invoked from Python applications to allow use of existing IBM Streams toolkits.

See *topology*

streamsx.topology	Python application support for IBM Streams.
streamsx.topology.topology	Streaming application definition.
streamsx.topology.context	Context for submission and build of topologies.
streamsx.topology.schema	Schemas for streams.
streamsx.topology.state	Application state.
streamsx.topology.composite	Composite transformations.
streamsx.topology.tester	Testing support for streaming applications.
streamsx.topology.tester_runtime	Runtime tester functionality.
streamsx.ec	Access to the IBM Streams execution context.
streamsx.spl.op	Integration of SPL operators.
streamsx.spl.types	SPL type definitions.
streamsx.spl.toolkit	SPL toolkit integration.

1.1 streamsx.topology

Python application support for IBM Streams.

1.1.1 Overview

IBM® Streams is an advanced analytic platform that allows user-developed applications to quickly ingest, analyze and correlate information as it arrives from thousands of real-time sources. Streams can handle very high data throughput rates, millions of events or messages per second.

With this API Python developers can build streaming applications that can be executed using IBM Streams, including the processing being distributed across multiple computing resources (hosts or machines) for scalability.

IBM Streams is also available on IBM Cloud through *IBM Streaming Analytics service*

1.1.2 Creating Applications

Applications are created by declaring a flow graph contained in a `Topology` instance.

For details see *`streamsx.topology.topology`*.

1.1.3 Extensions

This package (*streamsx*) provides the core functionality to build streaming applications in Python for Streams.

Additional *streamsx.** packages are available that provide adapters to external systems, analytics and streaming primitives. This include:

- Apache Kafka integration - streamsx.kafka
- Database integration - streamsx.database
- Geospatial analytics- streamsx.geospatial
- IBM Event Streams integration - streamsx.eventstreams
- MQTT integration - streamsx.mqtt
- Cloud Object Storage integration - streamsx.objectstorage
- Streaming primitives - streamsx.standard

A full list of available packages is at : https://pypi.org/search?q=streamsx

1.1.4 Microservices

Publish-subscribe provides the abiltity to connect streams between independent IBM Streams applications regardless of their implementation language. This allows a microservice approach where a Streams application acting as a service publishes one or more streams. Subscriber services then subscribe to those streams without requiring any knowledge of how a stream is published.

Publish-subscribe overview

Applications can publish streams to a topic name which can then be subscribed to by other applications (or even the same application). Publish-subscribe works across applications written in SPL and those written using the Java/Scala and Python application APIs.

A subscriber matches a publisher if their topic filter matches a publisher's topic name and the stream type (schema) is an exact match to that of the publisher. It is recommended that a single stream type is used for a topic name.

A topic is a string value (encoded with UTF-8), based upon the MQTT topic style

Topic names for publishing a stream:

- Must be at least one character long.
- Use / as a level separator, zero length topic levels are valid.
- Must not include wild card characters + and #.
- Must not include the Unicode character NULL (U+0000).

Topic filters for subscribing to streams:

- Must be at least one character long.

- Use / as a level separator.
- Must not include the Unicode character NULL (U+0000).
- + is a single-level wildcard character that can be used at any level, but it must occupy the entire level. +, *a/b/+*, *+/b/+* and *+/b* are valid but *a/b/c+* is not valid.
- # is a wildcard character that matches any number of levels including the parent and any number of child levels. The multi-level wildcard character must be specified either on its own or following a topic level separator. In either case it must be the last character specified in the topic filter. # and 'a/b/#' are valid but *a/b/c#* and *a/#/c* are not valid.

Without a wildcard character a topic filter is an exact match for a topic name so that filter *a/b/c* only matches *a/b/c*.

Single-level filter (+) match examples are:

- filter + matches *a* and *b* but not *a/b*
- filter *a/+* matches *a/b*, *a/c* and *a/* but not *a*, *b/c* or *a/b/c*
- filter +/+ matches *a/b*, *b/c*, *d/* and */* but not *a* or *a/b/c*

Multi-level filter (#) match examples are:

- filter # matches every topic name such as *a*, *b/c*, *//*
- filter *a/b/#* matches *a/b* (parent), *a/b/c*, *a/b/d* and *a/b/c/d*

Note: A publish-subscribe match requires the stream type to match as well as the topic filter matching the topic name.

Publish-subscribe is a many to many relationship, any number of publishers can publish to the same topic and stream type, and there can be many subscribers to a topic.

For example a telco ingest microservice/application may process Call Detail Records from network switches and publish processed records on multiple topics, `cdr/voice/normal`, `cdr/voice/dropped`, `cdr/sms`, etc. by publishing each processed stream with its own topic. Then a dropped call analytic microservice would subscribe to the `cdr/voice/dropped` topic.

Publish-subscribe is dynamic, using IBM Streams dynamic connections, an application can be submitted that subscribes to topics published by other already running applications. Once the new application has initialized, it will start consuming tuples from published streams from existing applications. And any stream the new application publishes will be subscribed to by existing applications where the topic and stream type matches.

An application only receives tuples that are published while it is connected, thus tuples are lost during a connection failure.

A Python application publishes streams using `publish()` and subscribes using `subscribe()`.

A stream of `Python tuples` can only be subscribed to by Python Streams applications. All other types (schemas) can be subscribed to by any Streams application.

Module contents

1.2 streamsx.topology.topology

Streaming application definition.

1.2.1 Overview

IBM Streams is an advanced analytic platform that allows user-developed applications to quickly ingest, analyze and correlate information as it arrives from thousands of real-time sources. Streams can handle very high data throughput rates, millions of events or messages per second.

With this API Python developers can build streaming applications that can be executed using IBM Streams, including the processing being distributed across multiple computing resources (hosts or machines) for scalability.

1.2.2 Topology

A `Topology` declares a graph of *streams* and *operations* against tuples (data items) on those streams.

After being declared, a Topology is submitted to be compiled into a Streams application bundle (sab file) and then executed. The sab file is a self contained bundle that can be executed in a distributed Streams instance either using the Streaming Analytics service on IBM Cloud or an on-premise IBM Streams installation.

The compilation step invokes the Streams compiler to produce a bundle. This effectively, from a Python point of view, produces a runnable version of the Python topology that includes application specific Python C extensions to optimize performance.

The bundle also includes any required Python packages or modules that were used in the declaration of the application, excluding ones that are in a directory path containing `site-packages`.

The Python standard package tool `pip` uses a directory structure including `site-packages` when installing packages. Packages installed with `pip` can be included in the bundle with `add_pip_package()` when using a build service. This avoids the requirement to have packages be preinstalled in cloud environments.

Local Python packages and modules containing callables used in transformations such as `map()` are copied into the bundle from their local location. The addition of local packages to the bundle can be controlled with `Topology.include_packages` and `Topology.exclude_packages`.

The Streams runtime distributes the application's operations across the resources available in the instance.

Note: *Topology* represents a declaration of a streaming application that will be executed by a Streams instance as a *job*, either using the Streaming Analytics service on IBM Cloud or an on-premises distributed instance. *Topology* does not represent a running application, so an instance of *Stream* class does not contain the tuples, it is only a declaration of a stream.

1.2.3 Stream

A `Stream` can be an infinite sequence of tuples, such as a stream for a traffic flow sensor. Alternatively, a stream can be finite, such as a stream that is created from the contents of a file. When a streams processing application contains infinite streams, the application runs continuously without ending.

A stream has a schema that defines the type of each tuple on the stream. The schema for a stream is either:

- `Python` - A tuple may be any Python object. This is the default when the schema is not explictly or implicitly set.
- `String` - Each tuple is a Unicode string.
- `Binary` - Each tuple is a blob.
- `Json` - Each tuple is a Python dict that can be expressed as a JSON object.
- Structured - A stream that has a structured schema of a ordered list of attributes, with each attribute having a fixed type (e.g. float64 or int32) and a name. The schema of a structured stream is defined using typed named tuple or `StreamSchema`.

A stream's schema is implictly dervied from type hints declared for the callable of the transform that produces it. For example *readings* defined as follows would have a structured schema matching `SensorReading`

```
class SensorReading(typing.NamedTuple):
    sensor_id: str
    ts: int
    reading: float

def reading_from_json(value:dict) -> SensorReading:
    return SensorReading(value['id'], value['timestamp'], value['reading'])

topo = Topology()
json_readings = topo.source(HttpReadings()).as_json()
readings = json_readings.map(reading_from_json)
```

Deriving schemas from type hints can be disabled by setting the topology's `type_checking` attribute to false, for example this would change *readings* in the previous example to have generic Python object schema `Python`

```
topo = Topology()
topo.type_checking = False
```

1.2.4 Stream processing

Callables

A stream is processed to produce zero or more transformed streams, such as filtering a stream to drop unwanted tuples, producing a stream that only contains the required tuples.

Streaming processing is per tuple based, as each tuple is submitted to a stream consuming operators have their processing logic invoked for that tuple.

A functional operator is declared by methods on `Stream` such as `map()` which maps the tuples on its input stream to tuples on its output stream. *Stream* uses a functional model where each stream processing operator is defined in terms a Python callable that is invoked passing input tuples and whose return defines what output tuples are submitted for downstream processing.

The Python callable used for functional processing in this API may be:

- A Python lambda function.
- A Python function.
- An instance of a Python callable class.

For example a stream `words` containing only string objects can be processed by a `filter()` using a lambda function:

```
# Filter the stream so it only contains words starting with py
pywords = words.filter(lambda word : word.startswith('py'))
```

When a callable has type hints they are used to:

- define the schema of the resulting transformation, see *Stream*.
- type checking the correctness of the transformation at topology declaration time.

For example if the callable defining the source had type hints that indicated it was an iterator of `str` objects then the schema of the resultant stream would be `String`. If this source stream then underwent a `Stream.map()` transform with a callable that had a type hint for its argument, a check is made to ensure that the type of the argument is compatible with `str`.

Type hints are maintained through transforms regardless of resultant schema. For example a transform that has a return type hint of `int` defines the schema as `Python`, but the type hint is retained even though the schema is generic. Thus an error is raised at topology declaration time if a downstream transformation uses a callable with a type hint that is incompatible with being passed an `int`.

How type hints are used is specific to each transformation, such as `source()`, `map()`, `filter()` etc.

Type checking can be disabled by setting the topology's `type_checking` attribute to false.

When a callable is a lambda or defined inline (defined in the main Python script, a notebook or an interactive session) then a serialized copy of its definition becomes part of the topology. The supported types of captured globals for these callables is limited to avoid increasing the size of the application and serialization failures due non-serializable objects directly or indirectly referenced from captured globals. The supported types of captured globals are constants (`int`, `str`, `float`, `bool`, `bytes`, `complex`), modules, module attributes (e.g. classes, functions and variables defined in a module), inline classes and functions. If a lambda or inline callable causes an exception due to unsupported global capture then moving it to its own module is a solution.

Due to Python bug 36697 a lambda or inline callable can incorrect capture a global variable. For example an inline class using a attribute of `self.model` will incorrectly capture the global `model` even if the global variable `model` is never used within the class. To workaround this bug use attribute or variable names that do not shadow global variables (e.g. `self._model`).

Due to issue 2336 an inline class using `super()` will cause an `AttributeError` at runtime. Workaround is to call the super class's method directly, for example replace this code:

```
class A(X):
    def __init__(self):
        super().__init__()
```

with:

```
class A(X):
    def __init__(self):
        X.__init__(self)
```

or move the class to a module.

Stateful operations

Use of a class instance allows the operation to be stateful by maintaining state in instance attributes across invocations.

Note: For support with consistent region or checkpointing instances should ensure that the object's state can be pickled. See https://docs.python.org/3.5/library/pickle.html#handling-stateful-objects

---

Initialization and shutdown

Execution of a class instance effectively run in a context manager so that an instance's `__enter__` method is called when the processing element containing the instance is initialized and its `__exit__` method called when the processing element is stopped. To take advantage of this the class must define both `__enter__` and `__exit__` methods.

Note: Since an instance of a class is passed to methods such as `map()` `__init__` is only called when the topology is *declared*, not at runtime. Initialization at runtime, such as opening connections, occurs through the `__enter__` method.

Example of using `__enter__` to create custom metrics:

```
import streamsx.ec as ec

class Sentiment(object):
    def __init__(self):
        pass

    def __enter__(self):
        self.positive_metric = ec.CustomMetric(self, "positiveSentiment")
        self.negative_metric = ec.CustomMetric(self, "negativeSentiment")

    def __exit__(self, exc_type, exc_value, traceback):
        pass

    def __call__(self):
        pass
```

When an instance defines a valid `__exit__` method then it will be called with an exception when:

- the instance raises an exception during processing of a tuple
- a data conversion exception is raised converting a value to an structutured schema tuple or attribute

If `__exit__` returns a true value then the exception is suppressed and processing continues, otherwise the enclosing processing element will be terminated.

Tuple semantics

Python objects on a stream may be passed by reference between callables (e.g. the value returned by a map callable may be passed by reference to a following filter callable). This can only occur when the functions are executing in the same PE (process). If an object is not passed by reference a deep-copy is passed. Streams that cross PE (process) boundaries are always passed by deep-copy.

Thus if a stream is consumed by two map and one filter callables in the same PE they may receive the same object reference that was sent by the upstream callable. If one (or more) callable modifies the passed in reference those changes may be seen by the upstream callable or the other callables. The order of execution of the downstream callables is not defined. One can prevent such potential non-deterministic behavior by one or more of these techniques:

- Passing immutable objects
- Not retaining a reference to an object that will be submitted on a stream
- Not modifying input tuples in a callable
- Using copy/deepcopy when returning a value that will be submitted to a stream.

Applications cannot rely on pass-by reference, it is a performance optimization that can be made in some situations when stream connections are within a PE.

Application log and trace

IBM Streams provides application trace and log services which are accesible through standard Python loggers from the *logging* module.

See *Application log and trace*.

SPL operators

In addition an application declared by *Topology* can include stream processing defined by SPL primitive or composite operators. This allows reuse of adapters and analytics provided by IBM Streams, open source and third-party SPL toolkits.

See `streamsx.spl.op`

1.2.5 Module contents

1.2.6 Module contents

Classes

`PendingStream`	Pending stream connection.
`Routing`	Defines how tuples are routed to channels in a parallel region.
`Sink`	Termination of a *Stream*.
`Stream`	The Stream class is the primary abstraction within a streaming application.
`SubscribeConnection`	Connection mode between a subscriber and matching publishers.

continues on next page

Table 2 – continued from previous page

`Topology`	The Topology class is used to define data sources, and is passed as a parameter when submitting an application.
`View`	The View class provides access to a continuously updated sampling of data items on a `Stream` after submission.
`Window`	Declaration of a window of tuples on a *Stream*.

1.3 streamsx.topology.context

Context for submission and build of topologies.

1.3.1 Module contents

Functions

`build`	Build a topology to produce a Streams application bundle.
`run`	Run a topology in a distributed Streams instance.
`submit`	Submits a *Topology* (application) using the specified context type.

Classes

`ConfigParams`	Configuration options which may be used as keys in `submit()` *config* parameter.
`ContextTypes`	Submission context types.
`JobConfig`	Job configuration.
`SubmissionResult`	Passed back to the user after a call to submit.

1.4 streamsx.topology.schema

Schemas for streams.

1.4.1 Overview

A stream represents an unbounded flow of tuples with a declared schema so that each tuple on the stream complies with the schema. A stream's schema may be one of:

- `StreamsSchema` structured schema - a tuple is a sequence of attributes, and an attribute is a named value of a specific type.
- `Json` a tuple is a JSON object.
- `String` a tuple is a string.
- `Python` a tuple is any Python object, effectively an untyped stream.

1.4.2 Structured schemas

A structured schema is a sequence of attributes, and an attribute is a named value of a specific type. For example a stream of sensor readings can be represented as a schema with three attributes `sensor_id`, `ts` and `reading` with types of `int64`, `int64` and `float64` respectively.

This schema can be declared a number of ways:

Python 3.6:

```
class SensorReading(typing.NamedTuple):
    sensor_id: int
    ts: int
    reading: float

sensors = raw_readings.map(parse_sensor, schema=SensorReading)
```

Python 3:

```
SensorReading = typing.NamedTuple('SensorReading',
    [('sensor_id', int), ('ts', int), ('reading', float)]

sensors = raw_readings.map(parse_sensor, schema=SensorReading)
```

Python 3:

```
sensors = raw_readings.map(parse_sensor,
    schema='tuple<int64 sensor_id, int64 ts, float64 reading>')
```

The supported types are defined by IBM Streams and are listed in `StreamSchema`.

Structured schemas provide type-safety and efficient network serialization when compared to passing a `dict` using `Python` streams.

Streams with structured schemas can be interchanged with any IBM Streams application using `publish()` and `subscribe()` maintaining type safety.

1.4.3 Defining a stream's schema

Every stream within a `Topology` has defined schema. The schema may be defined explictly (for example `map()` or `subscribe()`) or implicity (for example `filter()` produces a stream with the same schema as its input stream).

Explictly defining a stream's schema is flexible and various types of values are accepted as the schema.

- Builtin types as aliases for common schema types:
  - `json` (module) - for `Json`
  - `str` - for `String`
  - `object` - for `Python`
- Values of the enumeration `CommonSchema`
- An instance of `typing.NamedTuple` (Python 3)
- An instance of `StreamSchema`
- A string of the format `tuple<...>` defining the attribute names and types. See `StreamSchema` for details on the format and types supported.

- A string containing a namespace qualified SPL stream type (e.g. `com.ibm.streams.geospatial::FlightPathEncounterTypes.Observation3D`)

1.4.4 Module contents

Functions

`is_common`	Is *schema* an common schema.

Classes

`CommonSchema`	Common stream schemas for interoperability within Streams applications.
`StreamSchema`	Defines a schema for a structured stream.

1.5 streamsx.topology.state

Application state.

1.5.1 Overview

Stateful applications are ones that include callables that are classes and thus can maintain state as instance variables.

By default any state is reset to its initial state after a processing element (PE) restart. A restart may occur due to:

- a failure in the PE or its resource,
- a explicit PE restart request,
- or a parallel region width change (IBM Streams 4.3 or later)

The application or a portion of it may be configured to maintain state after a PE restart by one of two mechanisms.

- Consistent region. A consistent region is a subgraph where the states of callables become consistent by processing all the tuples within defined points on a stream. After a PE restart all callables in the region are reset to the last consistent point, so that the state of all callables represents the processing of the same input tuples to the region.
  - `streamsx.topology.topology.Stream.set_consistent()`
  - `ConsistentRegionConfig`
  - Consistent region overview
- Checkpointing, each stateful callable is checkpointed periodically and after a PE restart its callables are reset to their most recent checkpointed state.
  - `streamsx.topology.topology.Topology.checkpoint_period`

1.5.2 Stateful callables

Use of a class instance allows a transformation (for example `map()`) to be stateful by maintaining state in instance attributes across invocations.

When the callable is in a consistent region or checkpointing then it is serialized using *dill*. The default serialization may be modified by using the standard Python pickle mechanism of `__getstate__` and `__setstate__`. This is required if the state includes objects that cannot be serialized, for example file descriptors. For details see See https://docs.python.org/3.5/library/pickle.html#handling-stateful-objects .

If the callable as `__enter__` and `__exit__` context manager methods then `__enter__` is called after the object has been deserialized by *dill*. Thus `__enter__` is used to recreate runtime objects that cannot be serialized such as open files or sockets.

1.5.3 Module contents

Classes

`ConsistentRegionConfig`	A `ConsistentRegionConfig` configures a consistent region.

1.6 streamsx.topology.composite

Composite transformations.

New in version 1.14.

1.6.1 Module contents

Classes

`Composite`	Composite transformations support a single logical transformation being a composite of one or more basic transformations.
`ForEach`	Abstract composite for each transformation.
`Map`	Abstract composite map transformation.
`Source`	Abstract composite source.

1.7 streamsx.topology.tester

Testing support for streaming applications.

1.7.1 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 only passes tuples with values greater than 5:

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

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: The package streamsx.testing provides nose plugins to provide control over tests without having to modify their source code.

Changed in version 1.9: - Python 2.7 supported (except with Streaming Analytics service).

1.7.2 Module contents

Classes

`Tester`	Testing support for a Topology.

1.8 streamsx.topology.tester_runtime

Runtime tester functionality.

1.8.1 Overview

Module containing runtime functionality for *`streamsx.topology.tester`*.

When test is executed any specified `Condition` instances are executed in the context of the application under test (and not the `unittest` class instance). This module separates out the runtime execution code from the test definition module *`tester`*.

1.8.2 Module contents

Classes

`Condition`	A condition for testing.

1.9 streamsx.ec

Access to the IBM Streams execution context.

1.9.1 Overview

This module (*streamsx.ec*) provides access to the execution context when Python code is running in a Streams application.

A Streams application runs distributed or standalone.

Distributed

Distributed is used when an application is submitted to the Streaming Analytics service on IBM Cloud or a IBM Streams distributed instance.

With distributed a running application is a *job* that contains one or more processing elements (PEs). A PE corresponds to a Linux operating system process. The PEs in a job may be distributed across the resources (hosts) in the Streams instance.

Standalone

Standalone is a mode where the complete application is run as a single PE (process) outside of a Streams instance.

Standalone is typically used for ad-hoc testing of an application.

1.9.2 Application log and trace

IBM Streams provides application trace and log services.

Application log

The *Streams application log* service is for application logging, where logging is defined as the recording of serviceability information pertaining to application or operator events. The purpose of logging is to provide an administrator with enough information to do problem determination for items they can potentially control. In general, very few events are logged in the normal running scenario of an application or operator. Events pertinent to the failure or partial failure of application runtime scenarios should be logged.

When running in distributed or standalone the *com.ibm.streams.log* logger has a handler that records messages to the *Streams application log* service. The level of the logger and its handler are set to the configured application log level at PE start up.

This logger and handler discard any message with level below *INFO* (20).

Python application code can log a message suitable for an administrator by using the *com.ibm.streams.log* logger or a child logger that has `logger.propagate` evaulating to `True`. Example of logging a file exception:

```
try:
    import numpy
except ImportError as e:
    logging.getLogger('com.ibm.streams.log').error(e)
    raise
```

Application code must not modify the *com.ibm.streams.log* logger, if additional handlers or different levels are required a child logger should be used.

Application trace

The *Streams application trace* service is for application tracing, where tracing is defined as the recording of application or operator internal events and data. The purpose of tracing is to allow application or operator developers to debug their applications or operators.

When running in distributed or standalone the root logger has a handler that records messages to the *Streams application trace* service. The level of the logger and its handler are set to the configured application trace level at PE start up.

Python application code can trace a message using the root logger or a child logger that has `logger.propagate` evaulating to `True`. Example of logging a trace message:

```
trace = logging.getLogger(__name__)

...

    trace.info("Threshold set to %f", val)
```

Any existing logging performed by modules will automatically become Streams trace messages if the application is using the *logging* package.

Application code must not modify the root logger, if additional handlers or different levels are required a child logger should be used.

1.9.3 Execution Context

This module (*streamsx.ec*) provides access to the execution context when Python code is running in a Streams application.

Access is only supported when running:

- Streams 4.2 or later

This module may be used by Python functions or classes used in a *Topology* or decorated SPL operators.

Most functionality is only available when a Python class is being invoked in a Streams application.

Changed in version 1.9: Support for Python 2.7

1.9.4 Module contents

Functions

`channel`	Return the parallel region global channel number *obj* is executing in.
`domain_id`	Return the instance identifier.
`get_application_configuration`	Get a named application configuration.
`get_application_directory`	Get the application directory.
`instance_id`	Return the instance identifier.
`is_active`	Tests is code is active within a IBM Streams execution context.
`is_standalone`	Is the execution context standalone.
`job_id`	Return the job identifier.
`local_channel`	Return the parallel region local channel number *obj* is executing in.
`local_max_channels`	Return the local maximum number of channels for the parallel region *obj* is executing in.
`max_channels`	Return the global maximum number of channels for the parallel region *obj* is executing in.
`pe_id`	Return the PE identifier.
`shutdown`	Return the processing element (PE) shutdown event.

Classes

`CustomMetric`	Create a custom metric.
`MetricKind`	Enumeration for the kind of a metric.

1.10 streamsx.spl.op

Integration of SPL operators.

1.10.1 Invoking SPL Operators

IBM Streams supports *Stream Processing Language* (SPL), a domain specific language for streaming analytics. SPL creates an application by building a graph of operator invocations. These operators are declared in an SPL toolkit.

SPL streams have a structured schema, such as `tuple<rstring id, timestamp ts, float64 value>` for a sensor reading with a sensor identifier, timestamp and value. A schema is defined using `StreamSchema`.

A Python topology application can take advantage of SPL operators by using streams with structured schemas. A stream of Python objects can be converted to a structured stream using `map()` with the *schema* parameter set:

```
# s is stream of Python objects representing a sensor
s = ...

# map s to a structured stream using a lambda function
# for each sensor reading r a Python tuple is created
# with the required values matching the order of the
```

(continues on next page)

(continued from previous page)

```
# structured schema.
s2 = s.map(lambda r : (r.sensor_id, r.reading_time, r.reading),
     schema='tuple<rstring id, timestamp ts, float64 value>'
```

An SPL operator is invoked in an application by creating an instance of:

- `Invoke` - Invocation of an arbitrary SPL operator.
- `Source` - Invocation of an SPL source operator with one input port.
- `Map` - Invocation of an SPL map operator with one input port and one output port.
- `Sink` - Invocation of an SPL sink operator with one output port.

In SPL, operator invocation supports a number of clauses that are supported in Python.

Values for operator clauses

When an operator clause requires a value, the value may be passed as a constant, an input attribute (passed using the *attribute* method of the invocation), or an arbitrary SPL expression (passed as a string or an `Expression`). Because a string is interpreted as an SPL expression, a string constant should be passed by enclosing the quoted string in outer quotes (for example, '"a string constant"').

SPL is strictly typed so when passing a constant as a value the value may need to be strongly typed.

- `bool`, `int`, `float` and `str` values map automatically to SPL *boolean*, *int32*, *float64* and *rstring* respectively.
- `Enum` values map to an operator custom literal using the symbolic name of the value. For custom literals only the symbolic name needs to match a value expected by the operator, the class name and other values are arbitrary.
- The module *`streamsx.spl.types`* provides functions to create typed SPL expressions from values.

An optional type may be set to SPL *null* by passing either Python *None* or the value returned from `null()`.

Param clause

Operator parameterization is through operator parameters that configure and modify the operator for the specific application.

Parameters are passed as a *dict* containing the parameter names and their values (see *Values for operator clauses*).

Examples

To invoke a *Beacon* operator from the SPL standard toolkit producing 100 tuples at the rate of two per second:

```
schema = StreamSchema('tuple<uint64 seq>')
beacon = op.Source(topology, 'spl.utility::Beacon', schema,
    params = {'iterations':100, 'period':0.5})
```

To use an `IntEnum` to pass a custom literal to the `Parse` operator:

```
from enum import IntEnum

class DataFormats(IntEnum):
    csv = 0
    txt = 1

...
```

(continues on next page)

(continued from previous page)

```
params['format'] = DataFormats.csv
```

To create a *count* parameter of type *uint64* for the SPL *DeDuplicate* operator:

```
params['count'] = streamsx.spl.types.uint64(20)
```

After the instance representing the operator invocation has been created, additional parameters may be added through the *params* attribute. If the value is an expression that is only valid in the context of the operator invocation then the parameter must be added after the operator invocation has been created.

For example, the *Filter* operator uses an expression that is usually dependent on the context, filtering tuples based upon their attribute values:

```
fs = op.Map('spl.relational::Filter', beacon)
fs.params['filter'] = fs.expression('seq % 2ul == 0ul')
```

Output clause

The operator output clause defines the values of attributes on outgoing tuples on the operator invocation's output ports.

When a tuple is submitted by an operator invocation each of its attributes is set in one of three ways:

- By the operator based upon its state and input tuples. For example, a US ZIP code operator might set the *zipcode* attribute based upon its lookup of the ZIP code from the address details in the input tuple.
- By the operator implicitly setting output attributes from matching input attributes when those attributes have not been explicitly set elsewhere. Many streaming operators implicitly set output attributes to allow attributes to flow through the operator without any explicit coding. This only occurs when an output attribute is not explicitly set by the operator, or the output clause, and the input tuple has an attribute that matches the output attribute (same name and type, or same name and same type as the underlying type of an output attribute with an optional type). For example, in the US ZIP code operator, if the output tuple included attributes of `rstring city, rstring state` that matched input attributes, then they would be implicitly copied from the input tuple to the output tuple.
- By an output clause in the operator invocation. In this case the application invoking the operator is explicitly setting attributes using SPL expressions. An operator may provide output functions that return values based upon the operator's state and input tuples. For example, the US ZIP code operator might provide a `ZIPCode()` output function rather than explicitly setting an output attribute. Then the application is free to use any attribute name to represent the ZIP code in its output tuple.

In Python an output tuple attribute is set by creating an attribute in the operator invocation instance that is set to a return from the *output* method. The attribute value passed to the *output* method is passed as described in *Values for operator clauses*.

For example, invoking an SPL *Beacon* operator using an output function to set the sequence number of a tuple and an SPL expression to set the timestamp:

```
schema = StreamSchema('tuple<uint64 seq, timestamp ts>')
beacon = op.Source(topology, 'spl.utility::Beacon', schema, params = {'period':0.1})

# Set the seq attribute using an output function provided by Beacon
beacon.seq = beacon.output('IterationCount()')

# Set the ts attribute using an SPL function that returns the current time
beacon.ts = beacon.output('getTimestamp()')
```

See also:

Streams Processing Language (SPL) Reference Reference documentation.

Developing Streams applications Developing Streams applications.

Operator invocations Operator invocations from the SPL reference documentation.

1.10.2 Module contents

Functions

`main_composite`	Wrap a main composite invocation as a *Topology*.

Classes

`Expression`	An SPL expression.
`Invoke`	Declaration of an invocation of an SPL operator in a Topology.
`Map`	Declaration of an invocation of an SPL *map* operator.
`Sink`	Declaration of an invocation of an SPL sink operator.
`Source`	Declaration of an invocation of an SPL *source* operator.

1.11 streamsx.spl.types

SPL type definitions.

1.11.1 Overview

SPL is strictly typed, thus when invoking SPL operators using classes from `streamsx.spl.op` then any parameters must use the SPL type required by the operator.

1.11.2 Module contents

Functions

`float32`	Create an SPL `float32` value.
`float64`	Create an SPL `float64` value.
`int16`	Create an SPL `int16` value.
`int32`	Create an SPL `int32` value.
`int64`	Create an SPL `int64` value.
`int8`	Create an SPL `int8` value.
`null`	Return an SPL `null`.
`rstring`	Create an SPL `rstring` value.
`uint16`	Create an SPL `uint16` value.
`uint32`	Create an SPL `uint32` value.

continues on next page

Table 15 – continued from previous page

`uint64`	Create an SPL `uint64` value.
`uint8`	Create an SPL `uint8` value.

Classes

`Timestamp`	SPL native timestamp type with nanosecond resolution.

1.12 streamsx.spl.toolkit

SPL toolkit integration.

1.12.1 Overview

SPL operators are defined by an SPL toolkit. When a `Topology` contains invocations of SPL operators, their defining toolkit must be made known using `add_toolkit()`.

Toolkits shipped with the IBM Streams product under `$STREAMS_INSTALL/toolkits` are implictly known and must not be added through `add_toolkit`.

1.12.2 Module contents

Functions

`add_toolkit`	Add an SPL toolkit to a topology.
`add_toolkit_dependency`	Add a version dependency on an SPL toolkit to a topology.

CHAPTER

TWO

SPL PRIMITIVE PYTHON OPERATORS

SPL primitive Python operators provide the ability to perform tuple processing using Python in an SPL application.

A Python function or class is simply turned into an SPL primitive operator through provided decorators.

SPL (Streams Processing Language) is a domain specific language for streaming analytics supported by Streams.

streamsx.spl.spl	SPL Python primitive operators.

2.1 streamsx.spl.spl

SPL Python primitive operators.

2.1.1 Overview

SPL primitive operators that call a Python function or class methods are created by decorators provided by this module.

The name of the function or callable class becomes the name of the operator.

A decorated function is a stateless operator while a decorated class is an optionally stateful operator.

These are the supported decorators that create an SPL operator:

- `@spl.source` - Creates a source operator that produces tuples.
- `@spl.filter` - Creates a operator that filters tuples.
- `@spl.map` - Creates a operator that maps input tuples to output tuples.
- `@spl.for_each` - Creates a operator that terminates a stream processing each tuple.
- `@spl.primitive_operator` - Creates an SPL primitive operator that has an arbitrary number of input and output ports.

Decorated functions and classes must be located in the directory `opt/python/streams` in the SPL toolkit. Each module in that directory will be inspected for operators during extraction. Each module defines the SPL namespace for its operators by the function `spl_namespace`, for example:

```
from streamsx.spl import spl

def spl_namespace():
    return 'com.example.ops'

@spl.map()
```

(continues on next page)

(continued from previous page)

```
def Pass(*tuple_):
    return tuple_
```

creates a pass-through operator `com.example.ops::Pass`.

SPL primitive operators are created by executing the extraction script *spl-python-extract* against the SPL toolkit. Once created the operators become part of the toolkit and may be used like any other SPL operator.

2.1.2 Python classes as SPL operators

Overview

Decorating a Python class creates a stateful SPL operator where the instance fields of the class are the operator's state. An instance of the class is created when the SPL operator invocation is initialized at SPL runtime. The instance of the Python class is private to the SPL operator and is maintained for the lifetime of the operator.

If the class has instance fields then they are the state of the operator and are private to each invocation of the operator.

If the *__init__* method has parameters beyond the first *self* parameter then they are mapped to operator parameters. Any parameter that has a default value becomes an optional parameter to the SPL operator. Parameters of the form **args* and ***kwargs* are not supported.

Warning: Parameter names must be valid SPL identifers, SPL identifiers start with an ASCII letter or underscore, followed by ASCII letters, digits, or underscores. The name also must not be an SPL keyword.

Parameter names `suppress` and `include` are reserved.

The value of the operator parameters at SPL operator invocation are passed to the *__init__* method. This is equivalent to creating an instance of the class passing the operator parameters into the constructor.

For example, with this decorated class producing an SPL source operator:

```
@spl.source()
class Range(object):
  def __init__(self, stop, start=0):
    self.start = start
    self.stop = stop

  def __iter__(self):
      return zip(range(self.start, self.stop))
```

The SPL operator *Range* has two parameters, *stop* is mandatory and *start* is optional, defaulting to zero. Thus the SPL operator may be invoked as:

```
// Produces the sequence of values from 0 to 99
//
// Creates an instance of the Python class
// Range using Range(100)
//
stream<int32 seq> R = Range() {
  param
    stop: 100;
}
```

or both operator parameters can be set:

```
// Produces the sequence of values from 50 to 74
//
// Creates an instance of the Python class
// Range using Range(75, 50)
//
stream<int32 seq> R = Range() {
  param
    start: 50;
    stop: 75;
}
```

Operator state

Use of a class allows the operator to be stateful by maintaining state in instance attributes across invocations (tuple processing).

When the operator is in a consistent region or checkpointing then it is serialized using *dill*. The default serialization may be modified by using the standard Python pickle mechanism of `__getstate__` and `__setstate__`. This is required if the state includes objects that cannot be serialized, for example file descriptors. For details see See https://docs.python.org/3.5/library/pickle.html#handling-stateful-objects .

If the class has `__enter__` and `__exit__` context manager methods then `__enter__` is called after the instance has been deserialized by *dill*. Thus `__enter__` is used to recreate runtime objects that cannot be serialized such as open files or sockets.

Operator initialization & shutdown

Execution of an instance for an operator effectively run in a context manager so that an instance's `__enter__` method is called when the processing element containing the operator is initialized and its `__exit__` method called when the processing element is stopped. To take advantage of this the class must define both `__enter__` and `__exit__` methods.

Note: Initialization such as opening files should be in `__enter__` in order to support stateful operator restart & checkpointing.

Example of using `__enter__` and `__exit__` to open and close a file:

```
import streamsx.ec as ec

@spl.map()
class Sentiment(object):
    def __init__(self, name):
        self.name = name
        self.file = None

    def __enter__(self):
        self.file = open(self.name, 'r')

    def __exit__(self, exc_type, exc_value, traceback):
        if self.file is not None:
            self.file.close()

    def __call__(self):
        pass
```

When an instance defines a valid `__exit__` method then it will be called with an exception when:

- the instance raises an exception during processing of a tuple
- a data conversion exception is raised converting a Python value to an SPL tuple or attribute

If `__exit__` returns a true value then the exception is suppressed and processing continues, otherwise the enclosing processing element will be terminated.

Application log and trace

IBM Streams provides application trace and log services which are accesible through standard Python loggers from the *logging* module.

See *Application log and trace*.

2.1.3 Python functions as SPL operators

Decorating a Python function creates a stateless SPL operator. In SPL terms this is similar to an SPL Custom operator, where the code in the Python function is the custom code. For operators with input ports the function is called for each input tuple, passing a Python representation of the SPL input tuple. For an SPL source operator the function is called to obtain an iterable whose contents will be submitted to the output stream as SPL tuples.

Operator parameters are not supported.

An example SPL sink operator that prints each input SPL tuple after its conversion to a Python tuple:

```
@spl.for_each()
def PrintTuple(*tuple_):
    "Print each tuple to standard out."
     print(tuple_, flush=True)
```

2.1.4 Processing SPL tuples in Python

SPL tuples are converted to Python objects and passed to a decorated callable.

Overview

For each SPL tuple arriving at an input port a Python function is called with the SPL tuple converted to Python values suitable for the function call. How the tuple is passed is defined by the tuple passing style.

Tuple Passing Styles

An input tuple can be passed to Python function using a number of different styles:

- *dictionary*
- *tuple*
- *attributes by name* **not yet implemented**
- *attributes by position*

Dictionary

Passing the SPL tuple as a Python dictionary is flexible and makes the operator independent of any schema. A disadvantage is the reduction in code readability for Python function by not having formal parameters, though getters such as `tuple['id']` mitigate that to some extent. If the function is general purpose and can derive meaning from the keys that are the attribute names then `**kwargs` can be useful.

When the only function parameter is `**kwargs` (e.g. `def myfunc(**tuple_):`) then the passing style is *dictionary*.

All of the attributes are passed in the dictionary using the SPL schema attribute name as the key.

Tuple

Passing the SPL tuple as a Python tuple is flexible and makes the operator independent of any schema but is brittle to changes in the SPL schema. Another disadvantage is the reduction in code readability for Python function by not having formal parameters. However if the function is general purpose and independent of the tuple contents `*args` can be useful.

When the only function parameter is `*args` (e.g. `def myfunc(*tuple_):`) then the passing style is *tuple*.

All of the attributes are passed as a Python tuple with the order of values matching the order of the SPL schema.

Attributes by name

(**not yet implemented**)

Passing attributes by name can be robust against changes in the SPL scheme, e.g. additional attributes being added in the middle of the schema, but does require that the SPL schema has matching attribute names.

When *attributes by name* is used then SPL tuple attributes are passed to the function by name for formal parameters. Order of the attributes and parameters need not match. This is supported for function parameters of kind `POSITIONAL_OR_KEYWORD` and `KEYWORD_ONLY`.

If the function signature also contains a parameter of the form `**kwargs` (`VAR_KEYWORD`) then any attributes not bound to formal parameters are passed in its dictionary using the SPL schema attribute name as the key.

If the function signature also contains an arbitrary argument list `*args` then any attributes not bound to formal parameters or to `**kwargs` are passed in order of the SPL schema.

If there are only formal parameters any non-bound attributes are not passed into the function.

Attributes by position

Passing attributes by position allows the SPL operator to be independent of the SPL schema but is brittle to changes in the SPL schema. For example a function expecting an identifier and a sensor reading as the first two attributes would break if an attribute representing region was added as the first SPL attribute.

When *attributes by position* is used then SPL tuple attributes are passed to the function by position for formal parameters. The first SPL attribute in the tuple is passed as the first parameter. This is supported for function parameters of kind *POSITIONAL_OR_KEYWORD*.

If the function signature also contains an arbitrary argument list **args* (*VAR_POSITIONAL*) then any attributes not bound to formal parameters are passed in order of the SPL schema.

The function signature must not contain a parameter of the form `**kwargs` (*VAR_KEYWORD*).

If there are only formal parameters any non-bound attributes are not passed into the function.

The SPL schema must have at least the number of positional arguments the function requires.

Selecting the style

For signatures only containing a parameter of the form `*args` or `**kwargs` the style is implicitly defined:

- `def f(**tuple_)` - *dictionary* - `tuple_` will contain a dictionary of all of the SPL tuple attribute's values with the keys being the attribute names.
- `def f(*tuple_)` - *tuple* - `tuple_` will contain all of the SPL tuple attribute's values in order of the SPL schema definition.

Otherwise the style is set by the `style` parameter to the decorator, defaulting to *attributes by name*. The style value can be set to:

- `'name'` - *attributes by name* (the default)
- `'position'` - *attributes by position*

Examples

These examples show how an SPL tuple with the schema and value:

```
tuple<rstring id, float64 temp, boolean increase>
{id='battery', temp=23.7, increase=true}
```

is passed into a variety of functions by showing the effective Python call and the resulting values of the function's parameters.

Dictionary consuming all attributes by `**kwargs`:

```
@spl.map()
def f(**tuple_)
    pass
# f({'id':'battery', 'temp':23.7, 'increase': True})
#     tuple_={'id':'battery', 'temp':23.7, 'increase':True}
```

Tuple consuming all attributes by `*args`:

```
@spl.map()
def f(*tuple_)
    pass
# f('battery', 23.7, True)
#     tuple_=('battery',23.7, True)
```

Attributes by name consuming all attributes:

```
@spl.map()
def f(id, temp, increase)
    pass
# f(id='battery', temp=23.7, increase=True)
#     id='battery'
#     temp=23.7
#     increase=True
```

Attributes by name consuming a subset of attributes:

```
@spl.map()
def f(id, temp)
    pass
# f(id='battery', temp=23.7)
#    id='battery'
#    temp=23.7
```

Attributes by name consuming a subset of attributes in a different order:

```
@spl.map()
def f(increase, temp)
    pass
# f(temp=23.7, increase=True)
#    increase=True
#    temp=23.7
```

Attributes by name consuming *id* by name and remaining attributes by `**kwargs`:

```
@spl.map()
def f(id, **tuple_)
    pass
# f(id='battery', {'temp':23.7, 'increase':True})
#    id='battery'
#    tuple_={'temp':23.7, 'increase':True}
```

Attributes by name consuming *id* by name and remaining attributes by `*args`:

```
@spl.map()
def f(id, *tuple_)
    pass
# f(id='battery', 23.7, True)
#    id='battery'
#    tuple_=(23.7, True)
```

Attributes by position consuming all attributes:

```
@spl.map(style='position')
def f(key, value, up)
     pass
# f('battery', 23.7, True)
#    key='battery'
#    value=23.7
#    up=True
```

Attributes by position consuming a subset of attributes:

```
@spl.map(style='position')
def f(a, b)
   pass
# f('battery', 23.7)
#    a='battery'
#    b=23.7
```

Attributes by position consuming *id* by position and remaining attributes by `*args`:

```
@spl.map(style='position')
def f(key, *tuple_)
```

(continues on next page)

(continued from previous page)

```
    pass
# f('battery', 23.7, True)
#    key='battery'
#    tuple_=(23.7, True)
```

In all cases the SPL tuple must be able to provide all parameters required by the function. If the SPL schema is insufficient then an error will result, typically an SPL compile time error.

The SPL schema can provide a subset of the formal parameters if the remaining attributes are optional (having a default).

Attributes by name consuming a subset of attributes with an optional parameter not matched by the schema:

```
@spl.map()
def f(id, temp, pressure=None)
   pass
# f(id='battery', temp=23.7)
#     id='battery'
#     temp=23.7
#     pressure=None
```

2.1.5 Submission of SPL tuples from Python

The return from a decorated callable results in submission of SPL tuples on the associated outut port.

A Python function must return:

- `None`
- a Python tuple
- a Python dictionary
- a list containing any of the above.

None

When `None` is return then no tuple will be submitted to the operator output port.

Python tuple

When a Python tuple is returned it is converted to an SPL tuple and submitted to the output port.

The values of a Python tuple are assigned to an output SPL tuple by position, so the first value in the Python tuple is assigned to the first attribute in the SPL tuple:

```
# SPL input schema: tuple<int32 x, float64 y>
# SPL output schema: tuple<int32 x, float64 y, float32 z>
@spl.map(style='position')
def myfunc(a,b):
   return (a,b,a+b)

# The SPL output will be:
# All values explictly set by returned Python tuple
# based on the x,y values from the input tuple
```

(continues on next page)

(continued from previous page)

```
# x is set to: x
# y is set to: y
# z is set to: x+y
```

The returned tuple may be *sparse*, any attribute value in the tuple that is `None` will be set to their SPL default or copied from a matching attribute in the input tuple (same name and type, or same name and same type as the underlying type of an output attribute with an optional type), depending on the operator kind:

```
# SPL input schema: tuple<int32 x, float64 y>
# SPL output schema: tuple<int32 x, float64 y, float32 z>
@spl.map(style='position')
def myfunc(a,b):
   return (a,None,a+b)

# The SPL output will be:
# x is set to: x (explictly set by returned Python tuple)
# y is set to: y (set by matching input SPL attribute)
# z is set to: x+y
```

When a returned tuple has fewer values than attributes in the SPL output schema the attributes not set by the Python function will be set to their SPL default or copied from a matching attribute in the input tuple (same name and type, or same name and same type as the underlying type of an output attribute with an optional type), depending on the operator kind:

```
# SPL input schema: tuple<int32 x, float64 y>
# SPL output schema: tuple<int32 x, float64 y, float32 z>
@spl.map(style='position')
def myfunc(a,b):
   return a,

# The SPL output will be:
# x is set to: x (explictly set by returned Python tuple)
# y is set to: y (set by matching input SPL attribute)
# z is set to: 0 (default int32 value)
```

When a returned tuple has more values than attributes in the SPL output schema then the additional values are ignored:

```
# SPL input schema: tuple<int32 x, float64 y>
# SPL output schema: tuple<int32 x, float64 y, float32 z>
@spl.map(style='position')
def myfunc(a,b):
   return (a,b,a+b,a/b)

# The SPL output will be:
# All values explictly set by returned Python tuple
# based on the x,y values from the input tuple
# x is set to: x
# y is set to: y
# z is set to: x+y
#
# The fourth value in the tuple a/b = x/y is ignored.
```

Python dictionary

A Python dictionary is converted to an SPL tuple for submission to the associated output port. An SPL attribute is set from the dictionary if the dictionary contains a key equal to the attribute name. The value is used to set the attribute, unless the value is `None`.

If the value in the dictionary is `None`, or no matching key exists, then the attribute value is set to its SPL default or copied from a matching attribute in the input tuple (same name and type, or same name and same type as the underlying type of an output attribute with an optional type), depending on the operator kind.

Any keys in the dictionary that do not map to SPL attribute names are ignored.

Python list

When a list is returned, each value is converted to an SPL tuple and submitted to the output port, in order of the list starting with the first element (position 0). If the list contains *None* at an index then no SPL tuple is submitted for that index.

The list must only contain Python tuples, dictionaries or *None*. The list can contain a mix of valid values.

The list may be empty resulting in no tuples being submitted.

2.1.6 Module contents

Functions

`extracting`	Is a module being loaded by `spl-python-extract`.
`ignore`	Decorator to ignore a Python function.

Classes

`PrimitiveOperator`	Primitive operator super class.
`filter`	Decorator that creates a filter SPL operator from a callable class or function.
`for_each`	Creates an SPL operator with a single input port.
`input_port`	Declare an input port and its processor method.
`map`	Decorator to create a map SPL operator from a callable class or function.
`primitive_operator`	Creates an SPL primitive operator with an arbitrary number of input ports and output ports.
`source`	Create a source SPL operator from an iterable.

CHAPTER

THREE

STREAMS PYTHON REST API

Module that allows interaction with an running Streams instance or service through HTTPS REST APIs.

streamsx.build	REST API bindings for IBM® Streams Cloud Pak for Data build service.
streamsx.rest	REST API bindings for IBM® Streams & Streaming Analytics service.
streamsx.rest_primitives	Primitive objects for REST bindings.

3.1 streamsx.build

REST API bindings for IBM® Streams Cloud Pak for Data build service.

3.1.1 Streams Build REST API

The REST Build API provides programmatic support for creating, submitting and managing Streams builds. You can use the REST Build API from any application that can establish an HTTPS connection to the server that is running the build service. The current support includes only methods for managing toolkits in the build service.

Cloud Pak for Data

`of_endpoint()` is the entry point to using the Streams Build REST API bindings, returning an `BuildService`.

See also:

IBM Streaming Analytics service

3.1.2 Module contents

Classes

`BuildService`	IBM Streams build service.

3.2 streamsx.rest

REST API bindings for IBM® Streams & Streaming Analytics service.

3.2.1 Streams REST API

The Streams REST API provides programmatic access to configuration and status information for IBM Streams objects such as domains, instances, and jobs.

IBM Cloud Pak for Data (Streams 5)

Integrated configuration within project

`of_service()` is the entry point to using the Streams REST API bindings, returning an `Instance`. The configuration required to connect is obtained from `ipcd_util.get_service_details` passing in the IBM Streams service instance name.

Integrated & standalone configurations

`of_endpoint()` is the entry point to using the Streams REST API bindings, returning an `Instance`.

IBM Streams On-premises (4.2, 4.3)

`StreamsConnection` is the entry point to using the Streams REST API bindings. Through its functions and the returned objects status information can be obtained for items such as `instances` and `jobs`.

3.2.2 Streaming Analytics REST API

You can use the Streaming Analytics REST API to manage your service instance and the IBM Streams jobs that are running on the instance. The Streaming Analytics REST API is accessible from IBM Cloud applications that are bound to your service instance or from an application outside of IBM Cloud that is configured with the service instance VCAP information.

`StreamingAnalyticsConnection` is the entry point to using the Streaming Analytics REST API. The function `get_streaming_analytics()` returns a `StreamingAnalyticsService` instance which is the wrapper around the Streaming Analytics REST API. This API allows functions such as `start` and `stop` the service instance.

In addition *StreamingAnalyticsConnection* extends from `StreamsConnection` and thus provides access to the Streams REST API for the service instance.

See also:

IBM Streams REST API overview Reference documentation for the Streams REST API.

Streaming Analytics REST API Reference documentation for the Streaming Analytics service REST API.

See also:

IBM Streaming Analytics service

3.2.3 Module contents

Classes

`StreamingAnalyticsConnection`	Creates a connection to a running Streaming Analytics service and exposes methods to retrieve the state of the service and its instance.
`StreamsConnection`	Creates a connection to a running distributed IBM Streams instance and exposes methods to retrieve the state of that instance.

3.3 streamsx.rest_primitives

Primitive objects for REST bindings.

3.3.1 Overview

Contains classes representing primitive Streams objects, such as `Instance`, `Job`, `PE`, etc.

3.3.2 Module contents

Classes

`ActiveService`	Domain or instance service.
`ActiveVersion`	Contains IBM Streams installation information
`ApplicationBundle`	Application bundle tied to an instance.
`ApplicationConfiguration`	An application configuration.
`Domain`	IBM Streams domain.
`ExportedStream`	Stream exported stream by a job.
`Host`	Resource in a Streams domain or instance.
`ImportedStream`	Stream imported by a job.
`Installation`	IBM Streams installation.
`Instance`	IBM Streams instance.
`Job`	A running streams application.
`JobGroup`	A job group definition.
`Metric`	Streams custom or system metric.
`Operator`	An operator invocation within a job.
`OperatorConnection`	Connection between operators.
`OperatorInputPort`	Operator input port.
`OperatorOutputPort`	Operator output port.
`PE`	Processing element (PE) within a job.
`PEConnection`	Stream connection between two PEs.
`PublishedTopic`	Metadata for a published topic.
`Resource`	A resource available to a IBM Streams domain.
`ResourceAllocation`	A resource that is allocated to an IBM Streams instance.
`ResourceTag`	Resource tag defined in a Streams domain

continues on next page

Table 4 – continued from previous page

`RestResource`	HTTP REST resource identifier.
`StreamingAnalyticsService`	Streaming Analytics service running on IBM Cloud.
`Toolkit`	IBM Streams toolkit.
`View`	View on a stream.
`ViewItem`	A stream tuple in view.

CHAPTER

FOUR

SCRIPTS

The *streamsx* package provides a number of command line scripts.

4.1 spl-python-extract

4.1.1 Overview

Extracts SPL Python primitive operators from decorated Python classes and functions.

Executing this script against an SPL toolkit creates the SPL primitive operator meta-data required by the SPL compiler (*sc*).

4.1.2 Usage

```
spl-python-extract [-h] -i DIRECTORY [--make-toolkit] [-v]

Extract SPL operators from decorated Python classes and functions.

optional arguments:
  -h, --help            show this help message and exit
  -i DIRECTORY, --directory DIRECTORY
                        Toolkit directory
  --make-toolkit        Index toolkit using spl-make-toolkit
  -v, --verbose         Print more diagnostics
```

4.1.3 SPL Python primitive operators

SPL operators that call a Python function or callable class are created by decorators provided by the *streamsx* package.

To create SPL operators from Python functions or classes one or more Python modules are created in the `opt/python/streams` directory of an SPL toolkit.

`spl-python-extract` is a Python script that creates SPL operators from Python functions and classes contained in modules under `opt/python/streams`.

The resulting operators embed the Python runtime to allow stream processing using Python.

Details on how to implement SPL Python primitive operators see *streamsx.spl.spl*.

4.2 streamsx-info

4.2.1 Overview

Information about streamsx package and environment.

Prints to standard out information about the *streamsx* package and environment variables used to support Python in IBM Streams and Streaming Analytics service.

A Python warning is issued if a mismatch is detected between the installed *streamsx* package and its modules. This is typically due to having a different version of the modules accessible through the environment variable `PYTHONPATH`.

Warning: When using the *streamsx* package ensure that the environment variable `PYTHONPATH` does **not** include a path ending with `com.ibm.streamsx.topology/opt/python/packages`. The IBM Streams environment configuration script `streamsprofile.sh` modifies or sets `PYTHONPATH` to include the Python support from the SPL topology toolkit shipped with the product. This was to support Python before the *streamsx* package was available. The recommendation is to unset `PYTHONPATH` or modify it not to include the path to the topology toolkit.

Output is subject to change in the order and information displayed. Intended as an ad-hoc tool to help diagnose issues with *streamsx*.

Script may also be run as Python module:

```
python -m streamsx.scripts.info
```

4.2.2 Usage

```
usage: streamsx-info [-h]

    Prints support information about streamsx package and environment.

optional arguments:
    -h, --help  show this help message and exit
```

4.3 streamsx-runner

4.3.1 Overview

Submits or builds a Streams application to the Streaming Analytics service.

The application to be submitted can be:

- A Python application defined through `Topology` using the `--topology` flag.
- An SPL application (main composite) using the `--main-composite` flag.
- A Streams application bundle (`sab` file) using the `--bundle` flag.

4.3.2 Streaming Analytics service

The Streaming Analytics service is defined by:

- Service name - `--service-name` defaulting to environment variable `STREAMING_ANALYTICS_SERVICE_NAME`. The service name must exist in the vcap services.
- Vcap services - Environment variable `VCAP_SERVICES` containing JSON representation of the service definitions or a file name containing the service definitions.

4.3.3 Job submission

Job submission occurs unless `--create-bundle` is set.

4.3.4 Bundle creation

When `-create-bundle` is specified with `-main-composite` or `--topology` then a Streams application bundle (sab file) is created.

If environment variable *STREAMS_INSTALL* is set the the build is local otherwise the build occurs in the IBM Cloud using the Streaming Analytics service.

When `STREAMS_INSTALL` is not set then *streamsx-runner* can be executed with no local Streams install.

When compiling an SPL application (`--main-composite`) then the path to the application toolkit containing the main composite must be listed with `--toolkits`.

Any other required local toolkits must be listed with with `--toolkits`.

4.3.5 Usage

```
streamsx-runner [-h] [--service-name SERVICE_NAME] | [--create-bundle]
              (--topology TOPOLOGY | --main-composite MAIN_COMPOSITE | --bundle BUNDLE)
              [--toolkits TOOLKITS [TOOLKITS ...]] [--job-name JOB_NAME]
              [--preload] [--trace {error,warn,info,debug,trace}]
              [--submission-parameters SUBMISSION_PARAMETERS [SUBMISSION_PARAMETERS ...
↪]]
              [--job-config-overlays file]

Execute a Streams application using a Streaming Analytics service.

optional arguments:
  -h, --help            show this help message and exit
  --service-name SERVICE_NAME
                        Submit to Streaming Analytics service
  --create-bundle       Create a bundle (sab file). No job submission occurs.
  --topology TOPOLOGY   Topology to call
  --main-composite MAIN_COMPOSITE
                        SPL main composite (namespace::composite_name)
  --bundle BUNDLE       Streams application bundle (sab file) to submit to
                        service

Build options:
  Application build options
```

(continues on next page)

(continued from previous page)

```
  --toolkits TOOLKITS [TOOLKITS ...]
                        SPL toolkit path containing the main composite and any
                        other required SPL toolkit paths.

Job options:
  Job configuration options

  --job-name JOB_NAME   Job name
  --preload             Preload job onto all resources in the instance
  --trace {error,warn,info,debug,trace}
                        Application trace level
  --submission-parameters SUBMISSION_PARAMETERS [SUBMISSION_PARAMETERS ...], -p␣
↪SUBMISSION_PARAMETERS [SUBMISSION_PARAMETERS ...]
                        Submission parameters as name=value pairs
  --job-config-overlays file
                        Path to file containing job configuration overlays
                        JSON. Overrides any job configuration set by the
                        application.
```

4.3.6 Submitting to Streaming Analytics service

An application is submitted to a Streaming Analytics service using `--service-name SERVICE_NAME`. The named service must exist in the VCAP services definition pointed to by the `VCAP_SERVICES` environment variable.

The application is submitted as source (except `--bundle`) and compiled into a Streams application bundle (`sab` file) using the build service before being submitted as a running job to the service instance.

See also:

Accessing a service

Python applications

To submit a Python application a Python function must be defined that returns the application (and optionally its configuration) to be submitted. The fully qualified name of this function is specified using the `--topology` flag.

For example, an application can be submitted as:

```
streamsx-runner --service-name Streaming-Analytics-xd \
    --topology com.example.apps.sensor_ingester
```

The function returns one of:

- a `Topology` instance defining the application
- **a `tuple` containing two values, in order:**
  - a `Topology` instance defining the application
  - **job configuration, one of:**
    - `JobConfig` instance
    - `dict` corresponding to the configuration object passed into `submit()`

For example the above function might be defined as:

```
def _create_sensor_ingester_app():
   topo = Topology('SensorIngesterApp')

   # Application declaration omitted
   ...

   return topo

def sensor_ingester():
    return (_create_sensor_ingester_app(), JobConfig(job_name='SensorIngester'))
```

Thus when this application is submitted using the *sensor_ingester* function it is always submitted with the same job name *SensorIngester*.

The function must be accessible from the current Python path (typically through environment variable `PYTHONPATH`).

SPL applications

The main composite that defines the application is specified using the `-main-composite` flag specifing the fully namespace qualified name.

Any required local SPL toolkits, *including the one containing the main composite*, must be indivdually specified by location to the `--toolkits` flag. Any SPL toolkit that is present on the IBM Cloud service need not be included.

For example, an application that uses the Slack toolkit might be submitted as:

```
streamsx-runner --service-name Streaming-Analytics-xd \
    --main-composite com.example.alert::SlackAlerter \
    --toolkits $HOME/app/alerters $HOME/toolkits/com.ibm.streamsx.slack
```

where `$HOME/app/alerters` is the location of the SPL application toolkit containing the `com.example.alert::SlackAlerter` main composite.

Warning: The main composite name must be namespace qualified. Use of the default namespace for a main composite is not recommended as it increases the chance of a name clash with another SPL toolkit.

Streams application bundles

A Streams application bundle is submitted to a service instance using `--bundle`. The argument to `--bundle` is a locally accessible file that will be uploaded to the service.

The bundle must have been created on using an IBM Streams install whose architecture and OS version matches the service instance. Currently this is `x86_64` and RedHat/CentOS 6 or 7 depending on the service instance.

The `--toolkits` flag must not be specified when submitting a bundle.

Job options

Job options, such as `--job-name`, configure the running job.

For `--topology` job options set as arguments to `streamsx-runner` override any configuration returned from the function defining the application.

4.3.7 Creating Streams application bundles

`--create-bundle` uses a local IBM Streams install to attempt to mimic the build that would occur with `-topology` or `--main-composite`. Differences between the local environment and the IBM Cloud Streaming Analytics build environment may cause build failures in one and not the other.

This can be used as a mechanism to perform a local test build before using the service, or as a valid mechanism to create bundles for later upload with `--bundle`.

For example simply changing the `--service-name name` to `--create-bundle` perfoms a local build of the same application:

```
# Submit to an Streaming Analytics service
streamsx-runner --service-name Streaming-Analytics-xd \
    --main-composite com.example.alert::SlackAlerter \
    --toolkits $HOME/app/alerters $HOME/toolkits/com.ibm.streamsx.slack

# Build the same application locally
streamsx-runner --create-bundle \
    --main-composite com.example.alert::SlackAlerter \
    --toolkits $HOME/app/alerters $HOME/toolkits/com.ibm.streamsx.slack
```

4.4 streamsx-sc

4.4.1 Overview

SPL compiler for IBM Streams running on IBM Cloud Pak for Data.

`streamsx-sc` replicates a sub-set of Streams 4.3 `sc` options.

`streamsx-sc` is supported for Streams 5.x (Cloud Pak for Data). A local install of Streams is **not** required, simply the installation of the *streamsx* package. All functionality is implemented through the Cloud Pak for Data and Streams build service REST apis.

Cloud Pak for Data configuration

Integrated configuration

The Streams instance (and its build service) and authentication are defined through environment variables:

- **CP4D_URL** - Cloud Pak for Data deployment URL, e.g. *https://cp4d_server:31843*.
- **STREAMS_INSTANCE_ID** - Streams service instance name.
- **STREAMS_USERNAME** - (optional) User name to submit the job as, defaulting to the current operating system user name.
- **STREAMS_PASSWORD** - Password for authentication.

Standalone configuration

The Streams build service and authentication are defined through environment variables:

- **STREAMS_BUILD_URL** - Streams build service URL, e.g. when the service is exposed as node port: *https://<NODE-IP>:<NODE-PORT>*
- **STREAMS_USERNAME** - (optional) User name to submit the job as, defaulting to the current operating system user name.
- **STREAMS_PASSWORD** - Password for authentication.

4.4.2 Usage

```
streamsx-sc [-h] --main-composite name [--spl-path SPL_PATH]
        [--optimized-code-generation] [--no-optimized-code-generation]
        [--prefer-facade-tuples] [--ld-flags LD_FLAGS]
        [--cxx-flags CXX_FLAGS] [--c++std C++STD]
        [--data-directory DATA_DIRECTORY]
        [--output-directory OUTPUT_DIRECTORY] [--disable-ssl-verify]
        [--static-link] [--standalone-application]
        [--set-relax-fusion-relocatability-restartability]
        [--checkpoint-directory path] [--profiling-sampling rate]
        [compile-time-args [compile-time-args ...]]
```

Options and arguments

compile-time-args: Pass named arguments each in the format *name=value* to the compiler. The name cannot contain the character = but otherwise is a free form string. It matches the name parameter that is specified in calls that are made to the compile-time argument access functions from within SPL code. The value can be any string. See Compile-time arguments .

-M,–main-composite: SPL Main composite

-t,–spl-path: Set the toolkit lookup paths. Separate multiple paths with `:`. Each path is a toolkit directory or a directory of toolkit directories. This path overrides the `STREAMS_SPLPATH` environment variable.

-a,–optimized-code-generation: Generate optimized code with less runtime error checking

—no-optimized-code-generation: Generate non-optimized code with more runtime error checking. Do not use with the –optimized-code- generation option.

-k,–prefer-facade-tuples: Generate the facade tuples when it is possible.

-w,–ld-flags: Pass the specified flags to ld while linking occurs.

-x,–cxx-flags: Pass the specified flags to the C++ compiler during the build.

–c++std: Specify the language level for the underlying C++ compiles.

–data-directory: Specifies the location of the data directory to use.

–output-directory: Specifies a directory where the application artifacts are placed.

–disable-ssl-verify: Disable SSL verification against the build service

Deprecated arguments Arguments supported by *sc* but deprecated. They have no affect on compilation.

-s,–static-link

-T,–standalone-application

-O,–set-relax-fusion-relocatability-restartability

-K,–checkpoint-directory

-S,–profiling-sampling

4.4.3 Toolkits

The application toolkit is defined as the working directory of *streamsx-sc*.

Local toolkits are found through the toolkit path set by *–spl-path* or environment variable `STREAMS_SPLPATH`. Local toolkits are included in the build code archive sent to the build service if:

- the toolkit is defined as a dependent of the application toolkit including recursive dependencies of required local toolkits.
- and a toolkit of a higher version within the required dependency range does not exist locally or remotely on the build service.

The toolkit path for the compilation on the build service includes:

- the application toolkit
- local tookits included in the build code archive
- all toolkits uploaded on the Streams build service
- all product toolkits on the Streams build service

The application toolkit and local toolkits included in the build archive are processed prior to the actual compilation by:

- having any Python SPL primitive operators extracted using `spl-python-extract`
- indexed using `spl-make-toolkit`

New in version 1.13.

4.5 streamsx-service

4.5.1 Overview

Control commands for a Streaming Analytics service.

4.5.2 Usage

```
streamsx-service [-h] [--service-name SERVICE_NAME] [--full-response]
              {start,status,stop} ...

Control commands for a Streaming Analytics service.

positional arguments:
  {start,status,stop}   Supported commands
    start               Start the service instance
    status              Get the service status.
    stop                Stop the instance for the service.

optional arguments:
```

(continues on next page)

(continued from previous page)

```
  -h, --help            show this help message and exit
  --service-name SERVICE_NAME
                        Streaming Analytics service name
  --full-response       Print the full JSON response.

service.py stop [-h] [--force]

optional arguments:
  -h, --help  show this help message and exit
  --force     Stop the service even if jobs are running.
```

4.5.3 Controlling a Streaming Analytics service

The Streaming Analytics service to control is defined using `--service-name SERVICE_NAME`. If not provided then the service name is defined by the environment variable `STREAMING_ANALYTICS_SERVICE_NAME`.

The named service must exist in the VCAP services definition pointed to by the `VCAP_SERVICES` environment variable.

The response from making the control request is printed to standard out in JSON format. By default a minimal response is printed including the status of the service and the job count. The complete response from the service REST API is printed if the option `--full-response` is given.

4.6 streamsx-streamtool

4.6.1 Overview

Command line interface for IBM Streams running on IBM Cloud Pak for Data.

`streamsx-streamtool` replicates a sub-set of Streams `streamtool` commands focusing on supporting DevOps for streaming applications.

`streamsx-streamtool` is supported for Streams Cloud Pak for Data (5.x) instances A local install of Streams is **not** required, simply the installation of the *streamsx* package. All functionality is implemented through Cloud Pak for Data and Streams REST apis.

Cloud Pak for Data configuration

The Streams instance and authentication are defined through environment variables, the details depend on if the Streams instance is running in integrated or standalone configuration.

Integrated configuration

- **CP4D_URL** - Cloud Pak for Data deployment URL, e.g. *https://cp4d_server:31843*.
- **STREAMS_INSTANCE_ID** - Streams service instance name.
- **STREAMS_USERNAME** - (optional) User name to submit the job as, defaulting to the current operating system user name. Overridden by the `--User` option.
- **STREAMS_PASSWORD** - Password for authentication.

Standalone configuration

- **STREAMS_REST_URL** - Streams SWS service (REST API) URL, e.g. when the service is exposed as node port: *https://<NODE-IP>:<NODE-PORT>*
- **STREAMS_BUILD_URL** - Streams build service (REST API) URL, e.g. when the service is exposed as node port: *https://<NODE-IP>:<NODE-PORT>*. Required for *lstoolkit* and *rmtoolkit*.
- **STREAMS_USERNAME** - (optional) User name to submit the job as, defaulting to the current operating system user name.
- **STREAMS_PASSWORD** - Password for authentication.

4.6.2 Usage

```
streamsx-streamtool submitjob [-h] [--jobConfig file-name]
        [--jobname job-name] [--jobgroup jobgroup-name]
        [--outfile file-name] [--P parameter-name]
        [--User user]
        sab-pathname

streamsx-streamtool canceljob [-h] [--force] [--collectlogs]
        [--jobs job-id | --jobnames job-names | --file file-name]
        [--User user]
        [jobid [jobid ...]]

streamsx-streamtool lsjobs [-h] [--jobs job-id] [--users user]
        [--jobnames job-names] [--fmt format-spec]
        [--xheaders] [--long] [--showtimestamp]
        [--User user]

streamsx-streamtool lsappconfig [-h] [--fmt format-spec] [--User user]

streamsx-streamtool mkappconfig [-h] [--property name=value]
        [--propfile property-file]
        [--description description] [--User user]
        config-name

streamsx-streamtool rmappconfig [-h] [--noprompt] [--User user] config-name

streamsx-streamtool chappconfig [-h] [--property name=value]
        [--description description] [--User user]
        config-name

streamsx-streamtool getappconfig [-h] [--User user] config-name

streamsx-streamtool lstoolkit [-h]
        (--all | --id toolkit-id | --name toolkit-name | --regex toolkit-regex)
        [--User user]

streamsx-streamtool rmtoolkit [-h]
        (--toolkitid toolkit-id | --toolkitname toolkit-name | --toolkitregex toolkit-
↪regex)
        [--User user]

streamsx-streamtool uploadtoolkit [-h] --path toolkit-path [--User user]
```

(continues on next page)

(continued from previous page)

```
streamsx-streamtool updateoperators [-h] [--jobname job-name]
        [--jobConfig file-name]
        [--parallelRegionWidth parallelRegionName=width]
        [--force] [--User user]
        [jobid]
```

4.6.3 submitjob

The streamtool submitjob command previews or submits one job.

Description:

A submitted job runs an application that is defined by an application bundle. Application bundles are created by the Stream Processing Language (SPL) compiler. A job consists of one or more processing elements (PEs). The PEs are placed on one or more of the application resources for the instance. The submission fails if the PE placement constraints can't be met.

Jobs remain in the system until they are canceled or the instance is stopped.

```
streamsx-streamtool submitjob [-h] [--jobConfig file-name]
        [--jobname job-name] [--jobgroup jobgroup-name]
        [--outfile file-name] [--P parameter-name]
        [--User user]
        sab-pathname
```

Options and arguments

sab-pathname Specifies the path name for the application bundle file. If you do not specify an absolute path, the command seeks the file in the directory where you ran the command. Alternatively, you can specify the path name for the application description language (ADL) file if the application bundle file exists in the same directory.

-g,–jobConfig: Specifies the name of an external file that defines a job configuration overlay. You can use a job configuration overlay to set the job configuration when the job is submitted or to change the configuration of a running job.

-P,–P: Specifies a submission-time parameter and value for the job. You can specify this option multiple times in the command.

-J,–jobgroup: Specifies the job group. If you do not specify this option, the command uses the following job group: default.

—jobname: Specifies the name of the job.

—outfile: Specifies the path and file name of the output file in which the command writes the list of submitted job IDs. The path can be an absolute or relative path. If you do not specify a path, the file is created in the directory where you run the command.

-U,–User: Specifies an IBM Streams user ID that has authority to run the command.

4.6.4 canceljob

The streamtool canceljob command cancels one or more jobs.

This command stops the processing elements (PEs) for the job and removes knowledge of the jobs and their PEs from the instance. The log files for the processing elements are scheduled for removal.

If you specify to collect the PE logs before they are removed, the operation can time out waiting for the termination of PEs. If such a timeout occurs, the operation fails and the jobs or PEs are still in the system. The canceljob command can be run again later to cancel them.

You can use the –force option to ignore a PE termination timeout and force the job to cancel.

```
streamsx-streamtool canceljob [-h] [--force] [--collectlogs]
        [--jobs job-id | --jobnames job-names | --file file-name]
        [--User user]
        [jobid [jobid ...]]
```

Options and arguments

jobid Specifies a list of job IDs.

-f,–file: Specifies the file that contains a list of job IDs, one per line.

-j,–jobs: Specifies a list of job IDs, which are delimited by commas.

—jobnames: Specifies a list of job names, which are delimited by commas.

—collectlogs: Specifies to collect the log and trace files for each processing element that is associated with the job.

—force: Specifies to quickly cancel a job and remove the job from the Streams data table.

-U,–User: Specifies an IBM Streams user ID that has authority to run the command.

4.6.5 lsjobs

The streamtool lsjobs command lists the jobs in the instance.

The streamtool lsjobs command provides a health summary for each job. The health summary is an aggregation of the PE health summaries for the job. If all of the PEs for a job are reported as healthy, the job is reported as healthy. Otherwise, the job is reported as not healthy. Use the streamtool lspes command to determine the health of PEs.

The command also reports the status of each job. For more information about job states, see the IBM Streams product documentation.

The date and time that the job was submitted are presented in local time with the iso8601 format: yyyy-mm-ddThh:mm:ss+/-hhmm, where the final hhmm values are the local offset from UTC. For example: 2010-03-16T13:41:53-0500.

When job selection options are specified, selected jobs must meet all of the selection criteria. After a cancel request for a job is processed, this command no longer reports the job or its processing elements (PEs).

```
streamsx-streamtool lsjobs [-h] [--jobs job-id] [--users user]
        [--jobnames job-names] [--fmt format-spec]
        [--xheaders] [--long] [--showtimestamp]
        [--User user]
```

Options and arguments

-j,–jobs: Specifies a list of job IDs, which are delimited by commas.

—jobnames: Specifies a list of job names, which are delimited by commas.

-u,–users: Specifies to select from this list of user IDs, which are delimited by commas.

—xheaders: Specifies to exclude headings from the report.

-l,–long: Reports launch count, full host names, and all of the operator instance names for the PEs.

—fmt: Specifies the presentation format. The command supports the following values:

- %Mf: Multiline record format. One line per field.
- %Nf: Name prefixed field table format. One line per job.
- %Tf: Standard table format, which is the default. One line per job.

—showtimestamp: Specifies to show a time stamp in the output to indicate when the command was run.

-U,–User: Specifies an IBM Streams user ID that has authority to run the command.

4.6.6 lsappconfig

The streamtool lsappconfig command lists the available configurations that enable connections to an external application.

Retrieve a list of configurations for making a connection to an external application.

```
streamsx-streamtool lsappconfig [-h] [--fmt format-spec] [--User user]
```

Options and arguments

—fmt: Specifies the presentation format. The command supports the following values:

- %Mf: Multiline record format. One line per field.
- %Nf: Name prefixed field table format. One line per cfgname.
- %Tf: Standard table format, which is the default. One line per cfgname.

-U,–User: Specifies an IBM Streams user ID that has authority to run the command.

4.6.7 mkappconfig

The streamtool mkappconfig command creates a configuration that enables connection to an external application.

Operators can retrieve the configuration information to make a connection to an external application, such as an Internet Of Things application. The properties include items that the application needs at runtime, like connection information and credentials.

Use this command to register properties or a properties file. Create the property file using a name=value syntax.

```
streamsx-streamtool mkappconfig [-h] [--property name=value]
        [--propfile property-file]
        [--description description] [--User user]
        config-name
```

Options and arguments

config-name: Name of the app config

—description: Specifies a description for the application configuration. The description can be 1024 characters in length. If the description contains blank characters, it must be enclosed in single or double quotation marks. Quotation marks within the description must be preceded by a backslash ().

—property: Specifies a property name and value pair to add to or change in the configuration. This option can be specified multiple times and has an additive effect.

—propfile: Specifies the path to a file that contains a list of application configuration properties for connecting to an external application. The properties are listed as name=value pairs, each on a separate line. Use this option as a way to include multiple configuration properties when you create an application configuration. Options that you specify at the command line override values that are specified in this property file.

-U,–User: Specifies an IBM Streams user ID that has authority to run the command.

4.6.8 rmappconfig

The streamtool rmappconfig command removes a configuration that enables connection to an external application.

This command removes a configuration that is used for making a connection to an external application.

```
streamsx-streamtool rmappconfig [-h] [--noprompt] [--User user] config-name
```

Options and arguments

config-name: Name of the app config

—noprompt: Specifies to suppress confirmation prompts.

-U,–User: Specifies an IBM Streams user ID that has authority to run the command.

4.6.9 chappconfig

The streamtool chappconfig command updates a configuration that enables connection to an external application.

Use this command to change the configuration properties that are used to make a connection to an external application, such as an Internet Of Things application. You can change the values of properties or add new properties.

```
streamsx-streamtool chappconfig [-h] [--property name=value]
        [--description description] [--User user]
        config-name
```

Options and arguments

config-name: Name of the app config

—description: Specifies a description for the application configuration. The description can be 1024 characters in length. If the description contains blank characters, it must be enclosed in single or double quotation marks. Quotation marks within the description must be preceded by a backslash ().

—property: Specifies a property name and value pair to add to or change in the configuration. This option can be specified multiple times and has an additive effect.

-U,–User: Specifies an IBM Streams user ID that has authority to run the command.

4.6.10 getappconfig

The streamtool getappconfig command displays the properties of a configuration that enables connection to an external application.

This command retrieves the properties and values of a specific configuration for connecting to an external application.

```
streamsx-streamtool getappconfig [-h] [--User user] config-name
```

Options and arguments

config-name: Name of the app config

-U,–User: Specifies an IBM Streams user ID that has authority to run the command.

4.6.11 lstoolkit

List toolkits from a build service.

```
streamsx-streamtool lstoolkit [-h]
        (--all | --id toolkit-id | --name toolkit-name | --regex toolkit-regex)
        [--User user]
```

Options and arguments

-a,–all: List all toolkits

-i,–id: List a specific toolkit given its toolkit id

-n,–name: List all toolkits with this name

-r,–regex: List all toolkits where the name matches the given regex pattern

4.6.12 rmtoolkit

Remove toolkits from a build service.

```
streamsx-streamtool rmtoolkit [-h]
        (--id toolkit-id | --name toolkit-name | --regex toolkit-regex)
        [--User user]
```

Options and arguments

-i,–id: Specifies the id of the toolkit to delete

-n,–name: Remove all toolkits with this name

-r,–regex: Remove all toolkits where the name matches the given regex pattern

4.6.13 uploadtoolkit

Upload a toolkit to a build service.

```
streamsx-streamtool uploadtoolkit [-h] --path toolkit-path [--User user]
```

Options and arguments

-p,–path: Specifies the path of the indexed toolkit to upload

New in version 1.13.

4.6.14 updateoperators

Adjust a job configuration while the job is running in order to improve the job performance

```
streamsx-streamtool updateoperators [-h] [--jobname job-name]
        [--jobConfig file-name]
        [--parallelRegionWidth parallelRegionName=width]
        [--force] [--User user]
        [jobid]
```

Options and arguments

jobid: Specifies a job ID

—jobname: Specifies the name of the job

-g,–jobConfig: Specifies the name of an external file that defines a job configuration overlay. You can use a job configuration overlay to set the job configuration when the job is submitted or to change the configuration of a running job.

—parallelRegionWidth: Specifies a parallel region name and its width.

—force: Specifies whether to automatically stop the PEs that need to be stopped.

-U,–User: Specifies an IBM Streams user ID that has authority to run the command.

CHAPTER

FIVE

ENVIRONMENTS

5.1 IBM Streaming Analytics service

5.1.1 Overview

IBM® Streaming Analytics for IBM Cloud is powered by IBM® Streams, an advanced analytic platform that you can use to ingest, analyze, and correlate information as it arrives from different types of data sources in real time. When you create an instance of the Streaming Analytics service, you get your own instance of IBM® Streams running in IBM® Cloud, ready to run your IBM® Streams applications.

See also:

Overview at ibm.com

IBM Cloud catalog

Streaming Analytics service documentation

5.1.2 Package support

This *streamsx* package supports :

- Developing streaming applications in Python that can be submitted to a Streaming Analytics service. See *streamsx.topology.topology*, `STREAMING_ANALYTICS_SERVICE`.
- Submitting streaming applications written in Python or SPL to a Streaming Anlaytics service. See *Python applications*, *SPL applications*.
- Submitting a pre-compiled Streams application bundle (`sab` file) Python or SPL to a Streaming Anlaytics service. See *Streams application bundles*.
- Python bindings to the IBM Streams REST API and the Streaming Analytics REST API. See *streamsx.rest*

5.1.3 Accessing a service

In order to use a Streaming Analytics service you must have access to credentials for the service. There are two mechanisms used by this package, VCAP services and direct use of Streaming Analytics credentials.

VCAP services

This is the format used by Cloud Foundry for bindable services. The service key for Streaming Analytics service is `streaming-analytics`, the value of that key in the VCAP services is a list of accessible services, each service represented by a separate object.

Each streaming analytics object must have these keys:

- `name` identifying the name of the service.
- `credentials` identifying the connection credentials for the service.

Example VCAP services containing two Streaming Analytics services *sa-test* and *sa-prod* (with the specific connection details elided):

```
{
"streaming-analytics": [
{
  "name": "sa-test",
  "credentials":
  {
     "apikey": "...",
     "iam_apikey_description": "Auto generated apikey during resource-key operation␣
↪for Instance - ...",
     "iam_apikey_name": "auto-generated-apikey-...",
     "iam_role_crn": "crn:v1:bluemix:public:iam::::serviceRole:Writer",
     "iam_serviceid_crn": "crn:v1:bluemix:public:iam-identity ...",
      "v2_rest_url": "https://streams-app-service.ng.bluemix.net/v2/streaming_
↪analytics/..."
  }
},
{
  "name": "sa-prod",
  "credentials":
  {
     "apikey": "...",
     "iam_apikey_description": "Auto generated apikey during resource-key operation␣
↪for Instance - ...",
     "iam_apikey_name": "auto-generated-apikey-...",
     "iam_role_crn": "crn:v1:bluemix:public:iam::::serviceRole:Writer",
     "iam_serviceid_crn": "crn:v1:bluemix:public:iam-identity ...",
      "v2_rest_url": "https://streams-app-service.ng.bluemix.net/v2/streaming_
↪analytics/..."
  }
}
]
}
```

Note: The specific keys in the credentials may differ depending on the service plan.

See also:

https://docs.run.pivotal.io/devguide/deploy-apps/environment-variable.html#VCAP-SERVICES

Cloud Foundry applications

When a Streaming Analytics service is bound to a Cloud Foundry Python application the environment variable `VCAP_SERVICES` is automatically defined and contains a string representation of the JSON VCAP services information.

Client applications

Client applications are ones that run outside of the IBM Cloud, for example on a local laptop, or applications that are not bound to a service.

Client applications running must define a valid VCAP services in its JSON format as either:

- In the environment variable `VCAP_SERVICES` containing a string representation of the JSON VCAP services information.
- In a file containing a string representation of the JSON VCAP services information and have the file's absolute path in either:
  - the environment variable `VCAP_SERVICES`
  - the configuration property `VCAP_SERVICES` when submitting an application using `submit()` with context type `STREAMING_ANALYTICS_SERVICE`. This overrides the environment variable *VCAP_SERVICES*.

The contents of the file must be manually created, the credentials for the `credentials` key are obtained from the Streaming Analytics manage console. Select the *Service Credentials* page and then copy the required credentials. You may need to first create credentials. You can an copy the credentials by taking the *View credentials* action and then clicking the *copy to clipboard* icon on the right hand side.

Warning: The credential information in VCAP services is in plain text. Ensure that the any file containing the information or setting the environment variable has suitable permissions set. For example only readable by the intended user.

Selecting the service

The Streaming Analyitcs service to use is specifed by its name, the required service much exist in the VCAP service information using the `name` key.

The name of the service to use is set by:

- the environment variable `STREAMING_ANALYTICS_SERVICE_NAME`.
- the configuration property `SERVICE_NAME` when submitting an application using `submit()` with context type `STREAMING_ANALYTICS_SERVICE`. This overrides the environment variable *STREAMING_ANALYTICS_SERVICE_NAME*.
- the `--service-name` option to `streamsx-runner`.

Service definition

The Streaming Analytics service to use may be specified solely using its credentials. The credentials are specified:

- with the configuration property `SERVICE_DEFINITION` when submitting an application using `submit()` with context type `STREAMING_ANALYTICS_SERVICE`.
- when using `streamsx.rest.StreamingAnalyticsConnection.of_definition()` to create a REST connection.

Credentials obtained from the Streaming Analytics manage console. Select the *Service Credentials* page and then copy the required credentials. You may need to first create credentials. You can an copy the credentials by taking the *View credentials* action and then clicking the *copy to clipboard* icon on the right hand side.

5.2 IBM Streams Python setup

5.2.1 Developer setup

Developers install the *streamsx* package Python Package Index (PyPI) to use this functionality:

```
pip install streamsx
```

If already installed upgrade to the latest version is recommended:

```
pip install --upgrade streamsx
```

A local install of IBM Streams is **not** required when:

- Using the Streams and Streaming Analytics REST bindings *streamsx.rest*.
- Devloping and submitting streaming applications using *streamsx.topology.topology* to Cloud Pak for Data or Streaming Analytics service on IBM Cloud.
  - The environment variable `JAVA_HOME` must reference a Java 1.8 JRE or JDK/SDK.

A local install of IBM Streams is required when:

- Developing and submitting streaming applications using *streamsx.topology.topology* to IBM Streams 4.2, 4.3 distributed or standalone contexts.
  - If set the environment variable `JAVA_HOME` must reference a Java 1.8 JRE or JDK/SDK, otherwise the Java install from `$STREAMS_INSTALL/java` is used.
- Creating SPL toolkits with Python primitive operators using *streamsx.spl.spl* decorators for use with 4.2, 4.3 distributed or standalone applications.

Warning: When using the *streamsx* package ensure that the environment variable `PYTHONPATH` does **not** include a path ending with `com.ibm.streamsx.topology/opt/python/packages`. The IBM Streams environment configuration script `streamsprofile.sh` modifies or sets `PYTHONPATH` to include the Python support from the SPL topology toolkit shipped with the product. This was to support Python before the *streamsx* package was available. The recommendation is to unset `PYTHONPATH` or modify it not to include the path to the topology toolkit.

Note: The *streamsx* package is self-contained and does not depend on any SPL topology toolkit (`com.ibm.streamsx.topology`) installed under `$STREAMS_INSTALL/toolkits` or on the SPL compiler's (`sc`) toolkit

path. This is true at SPL compilation time and runtime.

5.2.2 Streaming Analytics service

The service instance has Anaconda installed with Python 3.6 as the runtime environment and has `PYTHONHOME` Streams application environment variable pre-configured.

Any streaming applications using Python must use Python 3.6 when submitted to the service instance. The *streamsx* package must be installed locally and applications are submitted to the `STREAMING_ANALYTICS_SERVICE` context.

5.2.3 IBM Cloud Pak for Data

An IBM Streams service instance within Cloud Pak for Data has Anaconda installed with Python 3.6 as the runtime environment and has `PYTHONHOME` Streams application environment variable pre-configured.

Any streaming applications using Python must use Python 3.6 when submitted to the service instance.

Streaming applications can be submitted through Jupyter notebooks running in Cloud Pak for Data projects. The *streamsx* package is preinstalled and applications are submitted to the `DISTRIBUTED` context.

Streaming applications can be submitted externally to the OpenShift cluster containing Cloud Pak for Data. The *streamsx* package must be installed locally and applications are submitted to the `DISTRIBUTED` context. The specific environment variables depend on if the Streams instance is in a integrated or standalone configuration. See `DISTRIBUTED` for details.

5.2.4 IBM Streams 4.2, 4.3

For a distributed cluster running Streams Python 3.7, 3.6 or 3.5 may be used.

Anaconda or Miniconda distributions may be used as the Python runtime, these have the advantage of being pre-built and including a number of standard packages. Ananconda installs may be downloaded at: https://www.continuum.io/downloads .

If building Python from source then it must be built to support embedding of the runtime with shared libraries (`--enable-shared` option to *configure*).

Distributed

For distributed the Streams application environment variable `PYTHONHOME` must be set to the Python install path.

This is set using *streamtool* as:

```
streamtool setproperty --application-ev PYTHONHOME=path_to_python_install
```

The application environment variable may also be set using the Streams console. The *Instance Management* view has an *Application Environment Variables* section. Expanding the details for that section allows modification of the set of environment variables available to Streams applications.

The Python install path must be accessible on every application resource that will execute Python code within a Streams application.

Note: The Python version used to declare and submit the application must compatible with the setting of `PYTHONHOME` in the instance. For example, if `PYTHONHOME` Streams application instance variable points to a Python 3.6 install, then Python 3.5 or 3.6 can be used to declare and submit the application.

Standalone

The environment `PYTHONHOME` must be set to the Python install path.

5.2.5 Bundle Python version compatibility

As of 1.13 Streams application bundles (sab files) invoking Python are binary compatible with a range of Python releases when using Python 3.

The minimum verson supported is the version of Python used during bundle creation.

The maximum version supported is the highest version of Python with a proposed release schedule.

For example if a sab is built with Python 3.6 then it can be submitted to a Streams instance using 3.6 or higher, up to & including 3.9 which is the highest Python release with a proposed release schedule as of 1.13.

Note: Compatability across Python releases is dependent on Python's Stable Application Binary Inteface.

5.3 Indices and tables

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

M

S