StreamPy Documentation

Release

Author

Jul 11, 2017

Contents

Contents:

CHAPTER 1

Agent module

This module contains the Agent class. The Agent and Stream classes are the building blocks of PythonStreams.

class Agent.**Agent**(*in_streams*, *out_streams*, *transition*, *state=None*, *call_streams=None*, *stream_manager=None*, *name=None*)

Bases: object

An agent is an automaton: a state-transition machine. An agent is initialized in __init__ and a state transition is executed by next().

An agent has lists of: (1) input streams, (2) output streams and (3) call streams. Streams are described in Stream.py.

During a state transition an agent: (1) May read values from its input streams. (Note that

reading values in a stream does not change the stream.)

2.Append values to the tails of its output streams.

3.Change the agent's own state.

When a call stream is modified the agent's next() method is called which causes the agent to execute a state transition.

The default is that every input stream is also a call stream, i.e., the agent executes a state transition when any of its input streams is modified. For performance reasons, we may not want the agent to execute state transitions each time any of its input streams is modified; we may want the agent to execute state transitions periodically — for example, every second. In this case, the call streams will be different from the input streams. A call stream that has a value appended to it every second will cause the agent to execute a state transition every second.

Parameters **in_streams** : list of streams

out_streams : list of streams

call_streams : list of streams

When a new value is added to a stream in this list a state transition is invoked. This the usual way (but not the only way) in which state transitions occur. A state transiton for an agent ag can also be executed by calling ag.next()

state: object

The state of the agent. The state is updated after a transition.

transition: function

This function is called by next() which is the state-transition operation for this agent. An agent's state transition is specified by its transition function.

stream_manager : function

Each stream has management variables, such as whether the stream is open or closed. After a state-transition the agent executes the stream_manager function to modify the management variables of the agent's output and call streams.

name : str, optional

name of this agent

Attributes

_in_lists: list of InList	InList defines the slice of a list. The j-th element of _in_lists is an InList that defines the slice of the j-th input stream that may be read by this agent in a state transition. For example, if listj = _in_lists[j].lists startj = _in_lists[j].start stopj = _in_lists[j].stop Then this agent can read the slice: listj[startj:stopj] of the jth input stream. This slice is a slice of the most recent values of the stream.
_out_lists: list	The j-th element of _out_lists is the list of values to be appended to the j-th output stream after the state transition.

Methods

next(stream_name=None)	Execute a state transition. The method has 3 parts: (i) set up the data structures to execute a state transition, (ii) call transition to: (a) get the values to be appended to output streams, (b) get the next state, and (c) update 'start' indices for each input stream. The 'start' pointers are indices where the agent asserts that it will no longer access elements of its input streams with indices earlier (i.e. smaller) that 'start'. (iii) update data structures after the transition.

next(*stream_name=None*)

Execute the next state transition.

This function does the following: Part 1: set up data structures for the state transition. Part 2: execute the state transition by calling self.transition Part 3: update data structures after the transition.

This method can be called by any agent and is called whenever a value is appended to any stream in call_streams

Parameters **stream_name** : str, optional

A new value was appended to the stream with name stream_name as a result of which this agent executes a state transition.

Agent.**EPSILON = 1e-12**

class Agent.**InList**

Bases: tuple

Attributes

list	Alias for field number 0
start	Alias for field number 1
stop	Alias for field number 2

Methods

count(...)	
index((value, [start, ...)	Raises ValueError if the value is not present.

list
Alias for field number 0

start
Alias for field number 1

stop
Alias for field number 2

CHAPTER 2

ML package

Subpackages

ML.Geomap package

Submodules

ML.Geomap.geomap module

Module contents

ML.KMeans package

Submodules

ML.KMeans.KMeansStream module

class ML.KMeans.KMeansStream.**KMeansStream**(*draw*, *output*, *k*, *incremental=True*, *figsize=(15, 8)*)

Helper class for kmeans clustering.

This class provides train and predict functions for using kmeans with *Stream_Learn*.

Parameters **draw** : boolean

Describes whether the data is to be plotted (data must have 2 or less dimensions).

output : boolean

Describes whether debug info is to be printed. Info includes average error, average number of iterations, current number of iterations, and number of changed points over time.

k : int

Describes the number of clusters to train.

incremental : boolean, optional

Describes whether the kmeans algorithm is run incrementally or not (the default is True). If incremental, then previous clusters are used to initialize new clusters. Otherwise, clusters are reinitialized randomly for each window.

figsize : tuple, optional

A tuple containing the width and height of the plot for the map (the default is (15, 8)).

Attributes

train	(function) The train function with signature as required by *Stream_Learn*.
predict	(function) The predict function with signature as required by 'Stream_Learn'.
avg_iterations	(float) The average number of iterations per window of data trained.
avg_error	(float) The average error per window of data trained.

Methods

reset()	Resets the KMeans functions and average values.

reset()
Resets the KMeans functions and average values.

Resets: train, predict, avg_iterations, avg_error

ML.KMeans.kmeans module

ML.KMeans.kmeans.**computeCentroids**(*X*, *index*, *k*)
Finds the centroids for the data given the index of the closest centroid for each data point.

Parameters **X** : numpy.ndarray

A numpy array with dimensions n * 2 for some integer n.

index : numpy.ndarray

A numpy array with dimensions n * 1 that describes the closest centroid to each point in *X*.

k : int

Describes the number of centroids. k - 1 is the maximum value that appears in *index*.

Returns **centroids** : numpy.ndarray

A numpy array with dimensions *k* * 2.

Notes

The centroids are computed by taking the mean of each group of points in *X* with the same index value. For i in [0, k), centroids[i] is the mean of all data points X[j] where index[j] is i.

ML.KMeans.kmeans.**evaluate_error**(*X*, *centroids*, *index*)
Returns the mean squared error.

Parameters **X** : numpy.ndarray

A numpy array with 2 columns.

centroids : numpy.ndarray

A numpy array with 2 columns.

index : numpy.ndarray

A numpy array with 1 column.

Returns float

The mean squared error.

Notes

The mean squared error is calculated as the average squared distance of each point from the closest centroid.

ML.KMeans.kmeans.**findClosestCentroids**(*X*, *centroids*)

Returns a numpy array containing the index of the closest centroid for each point in X.

Parameters **X** : numpy.ndarray

A numpy array with 2 columns.

centroids : numpy.ndarray

A numpy array with 2 columns.

Returns **index** : numpy.ndarray

A numpy array with dimensions n * 1, where n is the number of rows in *X*. For each row i in *index*, index[i] is in [0, k) where k is the number of rows in *centroids*.

ML.KMeans.kmeans.**init_plot**(*figsize=(15*, *8)*)

Initializes the plot.

Parameters **figsize** : tuple, optional

A tuple containing the width and height of the plot (the default is (15, 8)).

ML.KMeans.kmeans.**initialize**(*k*, *low*, *high*)

Returns k random points with x and y coordinates in [low, high).

Parameters **k** : int

The number of points to return.

low : int

The lower bound (inclusive) for a point.

high : int

The upper bound (exclusive) for a point.

Returns **centroids** : numpy.ndarray

Numpy array with dimensions *k* by 2.

ML.KMeans.kmeans.**initializeCentroids**(*X*, *k*)

Returns k random points from the data *X* without replacement.

Parameters **X** : numpy.ndarray

A numpy array with dimensions n * 2, where n >= *k*.

k : int

The number of points to return

Returns numpy.ndarray

Numpy array with dimensions *k* by 2.

ML.KMeans.kmeans.**initializeData**(*n*, *k*, *scale*, *low*, *high*)

Initialize n points around k random centroids each with a normal distribution and scale.

Parameters **n** : int

Describes the numbe of points to make around each centroid.

k : int

Describes the number of centroids.

scale : int

Describes the scale for the distribution.

low : int

The lower bound (inclusive) for a centroid.

high : int

The upper bound (exclusive) for a centroid.

Returns **X** : numpy.ndarray

A numpy array with dimensions (*n* * *k*) * 2.

ML.KMeans.kmeans.**initializeDataCenter**(*centroid*, *scale*, *n*)

Initialize n points with a normal distribution and scale around a centroid.

Parameters **centroid** : numpy.ndarray

Numpy array with dimensions 1 * 2.

scale : int

Describes the scale for the distribution.

n : int

Describes the number of points to make.

Returns **X** : numpy.ndarray

A numpy array with dimensions *n* * 2.

ML.KMeans.kmeans.**kmeans**(*X*, *k*, *initial_centroids=None*, *draw=False*, *output=False*)

Runs kmeans until clusters stop moving.

Parameters **X** : numpy.ndarray

A numpy array with 2 columns.

k : int

Describes the number of centroids.

initial_centroids : numpy.ndarray, optional

A numpy array with initial centroids to run the algorithm. This array has with dimensions *k* * 2. If not provided, algorithm is initialized with random centroids from the data *X*.

draw : boolean, optional

Describes whether the data is to be plotted (data must have 2 or less dimensions). The default is False.

output : boolean, optional

Describes whether debug info is to be printed (the default is False). Info includes current number of iterations and number of changed points over time.

Returns **centroids** : numpy.ndarray

Numpy array with learned centroids (dimensions are *k* * 2).

index : numpy.ndarray

Numpy array with dimensions n * 1, where n is the number of rows in *X*. Each value describes the closest centroid to each data point in *X*.

num_iters : int

Describes the number of iterations taken to run kmeans.

ML.KMeans.kmeans.**plotKMeans**(*X*, *centroids*, *previous*, *index*)

Plots the data and centroids.

This function plots the data with the current centroids and shows the movement of the centroids.

Parameters **X** : numpy.ndarray

A numpy array with 2 columns.

centroids : numpy.ndarray

A numpy array with 2 columns.

previous : numpy.ndarray

A numpy array with 2 columns and the same number of rows as *centroids*.

index : numpy.ndarray

A numpy array with 1 column.

Module contents

ML.LinearRegression package

Submodules

ML.LinearRegression.LinearRegressionStream module

class ML.LinearRegression.LinearRegressionStream.**LinearRegressionStream**(*draw*, *output*, *incremental=True*, *alpha=0.01*, *figsize=(15, 8)*)

Helper class for linear regression.

This class provides train and predict functions for using linear regression with *Stream_Learn*.

Parameters **draw** : boolean

Describes whether the data is to be plotted (data must have 1 dimension).

output : boolean

Describes whether debug info is to be printed. Info includes average error and current error.

incremental : boolean, optional

Describes whether the linear regression algorithm is run incrementally or not (the default is True). If incremental, then the algorithm uses incremental calculations for matrix inversion and matrix multiplication if the data has 1 feature, or stochastic gradient descent if the data has more than 1 feature. Otherwise, the algorithm uses linear algebra.

alpha : float, optional

Learning rate for stochastic gradient descent (the default is 0.01). Ignored if incremental is False or if incremental is True and data has 1 feature.

figsize : tuple, optional

A tuple containing the width and height of the plot for the map (the default is (15, 8)).

Attributes

train	(function) The train function with signature as required by *Stream_Learn*.
predict	(function) The predict function with signature as required by *Stream_Learn*.
w	(tuple) The learned weight vector.
avg_error	(float) The average error per window of data trained.

Methods

reset()	Resets the KMeans functions and average values.

reset()
Resets the KMeans functions and average values.

Resets: train, predict, avg_error

ML.LinearRegression.linear_regression module

ML.LinearRegression.linear_regression.**evaluate_error**(*X*, *y*, *w*)
Returns the mean squared error.

X [numpy.ndarray] Numpy array of data.

y [numpy.ndarray] Numpy array of outputs. Dimensions are n * 1, where n is the number of rows in *X*.

w [numpy.ndarray] Numpy array with dimensions (m + 1) * 1, where m is the number of columns in *X*.

Returns float

The mean squared error

ML.LinearRegression.linear_regression.**init_plot**(*figsize=(15, 8)*)
Initializes the plot.

Parameters **figsize** : tuple, optional

A tuple containing the width and height of the plot (the default is (15, 8)).

ML.LinearRegression.linear_regression.**plot**(*X*, *y*, *w*)
Plot X data, the actual y output, and the prediction line.

Parameters **X** : numpy.ndarray

Numpy array of data with 1 column.

y : numpy.ndarray

Numpy array of outputs. Dimensions are n * 1, where n is the number of rows in *X*.

w : numpy.ndarray

Numpy array with dimensions 2 * 1.

ML.LinearRegression.linear_regression.**predict**(*X*, *w*)
Returns the prediction for one data point.

Parameters **X** : numpy.ndarray

Numpy array of data

w : numpy.ndarray

Numpy array with dimensions (m + 1) * 1, where m is the number of columns in *X*.

Returns float

The mean squared error

ML.LinearRegression.linear_regression.**train**(*X*, *y*)
Trains a linear regression model using linear algebra.

Parameters **X** : numpy.ndarray

Numpy array of data

y : numpy.ndarray

Numpy array of outputs. Dimensions are n * 1, where n is the number of rows in *X*.

Returns **w** : numpy.ndarray

Trained vector with dimensions (m + 1) * 1, where m is the number of columns in *X*.

ML.LinearRegression.linear_regression.**train_sgd**(*X*, *y*, *alpha*, *w=None*)

Trains a linear regression model using stochastic gradient descent.

Parameters **X** : numpy.ndarray

Numpy array of data

y : numpy.ndarray

Numpy array of outputs. Dimensions are n * 1, where n is the number of rows in *X*.

alpha : float

Describes the learning rate.

w : numpy.ndarray, optional

The initial w vector (the default is zero).

Returns **w** : numpy.ndarray

Trained vector with dimensions (m + 1) * 1, where m is the number of columns in *X*.

Module contents

Submodules

ML.Stream_Learn module

class ML.Stream_Learn.**Stream_Learn**(*data_train*, *data_out*, *train_func*, *predict_func*, *min_window_size*, *max_window_size*, *step_size*, *num_features*, *filter_func=None*, *all_func=None*)

Stream framework for machine learning.

This class supports machine learning for streaming data using PSTREAMS. Given data for training and predicting along with functions to learn and predict, this class will output a stream of predictions. Both batch and continual learning is supported.

Parameters **data_train** : *Stream* or numpy.ndarray or other

A object containing data to be trained on. In the case of *Stream*, the object contains tuples of values where each tuple represents a row of data. Each tuple must have at least *num_features* values. The object can also contain non-tuples provided *filter_func* is used to extract the tuples in correct format. In the case of a *numpy* array, the array must have at least *num_features* columns. Any additional values / columns correspond to the output y data. If this is not a *Stream* or *numpy* array, the data will not be split into x and y.

data_out : *Stream*

A *Stream* object containing data to generate predictions on. The *Stream* contains tuples of values where each tuple represents a row of data and must have at least *num_features* values.

train_func : function

A function that trains a model. This function takes parameters x and y data, a model object, and a window_state tuple, and returns a trained model object. In the case of *data_train* as a *Stream*, this function has the signature (numpy.ndarray numpy.ndarray Object) -> (Object). The first parameter x will have dimensions i x *num_features*, where *min_window_size* <= i <= *max_window_size*. The second parameter y will have dimensions i x num_outputs, where num_outputs refers to the number of y outputs for an input. For example, num_outputs is 1 for 1 scalar output. For unsupervised learning, num_outputs is 0. In the case of *data_train* as a *numpy* array, this function has the signature (numpy.ndarray numpy.ndarray Object) -> (Object). The first parameter x will have dimensions N x *num_features*, where N refers to the total number of training examples. The second parameter y will have dimensions N x num_outputs where num_outputs is defined as before. If *data_train* is none of the above, the function has the signature (Object None Object) -> (Object). The first parameter is *data_train*. The third parameter is a model defined by this function. The fourth parameter is a window_state tuple with the values (current_window_size, steady_state, reset, *step_size*, *max_window_size*), where current_window_size describes the number of points in the window, steady_state is a boolean that describes whether the window has reached *max_window_size*, and reset is a boolean that can be set to True to reset the window.

predict_func : function

A function that takes as input 2 tuples corresponding to 1 row of data and a model and returns the prediction output. This function has the signature (tuple tuple Object) -> (Object). The first tuple x has *num_features* values and the second tuple y has num_outputs values, where num_outputs refers to the number of y outputs for an input. In the case of unsupervised learning, y is empty.

min_window_size : int

An int specifying the minimum size of the window to train on for continual learning. This will be ignored for batch learning.

max_window_size : int

An int specifying the maximum size of the window to train on for continual learning. This will be ignored for batch learning.

step_size : int

An int specifying the number of tuples to move the window by for continual learning. This will be ignored for batch learning.

num_features : int

An int that describes the number of features in the data.

filter_func : function, optional

A function that filters data for training. This function takes parameters x and y data and a model object, and returns a tuple with signature (boolean, tuple). The first value in the output describes if the data is to be trained on (True) or if it is an outlier (False). The second value is the tuple of data in correct format as described for *data_train*. If *data_train* is a *Stream* that contains tuples, this function has the signature (tuple tuple Object) -> (tuple). The first tuple x has *num_features* values and the second tuple y has num_outputs values, where num_outputs refers to the number of y outputs for an input.

The third parameter is a model defined by *train_func*. If *data_train* is a *Stream* that does not contain tuples, this function has the signature (Object None Object) -> (tuple), where the first parameter has the same type as the values in *data_train*.

all_func : function, optional

A function that processes the data for usage such as visualization. This function takes parameters x and y data, a model object, a state object, and a window_state tuple and returns an updated state object. This function has the signature (np.ndarray np.ndarray Object Object tuple) -> (Object). The first numpy array x has dimensions i x *num_features*, where *min_window_size* <= i <= *max_window_size*. The second numpy array y has dimensions i x num_outputs, where num_outputs refers to the number of y outputs for an input. The third parameter is the model object defined by *train_func*. The fourth parameter is a state object defined by this function. The fifth parameter is a window_state tuple with values as defined in description for *train_func*.

Methods

reset()	Resets the training window to *min_window_size*.
run()	Runs the framework and returns a *Stream* of outputs.

reset()

Resets the training window to *min_window_size*.

This function resets the training window to *min_window_size*. After resetting, the window has the last *min_window_size* points in the *Stream x_train*. For example, if *max_window_size* is 100, *min_window_size* is 2, and the window contains points [1, 100], after resetting the window contains points [98, 99].

Notes

If reset() is called before the window has reached *max_window_size*, the window will continue increasing in size until it reaches *max_window_size*. Then, the window will reset to *min_window_size*.

run()

Runs the framework and returns a *Stream* of outputs.

Returns **y_predict** : *Stream*

A *Stream* containing outputs as returned by *predict_func*.

Module contents

CHAPTER 3

MakeProcess module

CHAPTER 4

Operators module

This module has functions that convert operations on standard Python data structures to operations on streams.

The module has three collections of functions: (1) functions that convert operations on standard Python data structures to operations on streams. These functions operate on a list of input streams to generate a list of output streams. The functions deal with the following data structures:

1. lists,
2. individual elements of lists,
3. sliding windows, and
4. timed windows.

(2) functions that map the general case of multiple input streams and multiple output streams described above to the following special cases:

1. merge: an arbitrary number of input streams and a single output stream.
2. split: a single input stream and an arbitrary number of output streams.
3. op: a single input stream and a single output stream.
4. source: no input and an arbitrary number of output streams.

(e) sink: no ouput and an arbitrary number of input streams. These special cases simplify functions that need to be written for standard Python data structures. You can always use the multiple inputs and outputs case even if there is only one or no input or output; however, the functions for merge, split, op, source, and sink are simpler than the multiple input and output case.

(3) a function that provides a single common signature for converting operations on Python structures to operations on streams regardless of whether the function has no inputs, a single input stream, a list of input streams, or no outputs, a single output stream or a list of output streams. (12 October 2015. Mani. Changed initialization of output_lists.)

Operators.**adjustable_window_agent**(*f*, *inputs*, *outputs*, *state*, *call_streams=None*, *window_size=None*, *step_size=None*)

Operators.**adjustable_window_func**(*f*, *inputs*, *num_outputs*, *state*, *call_streams*, *window_size*, *step_size*)

Operators.**assert_is_list**(*x*)

Operators.**assert_is_list_of_lists**(*x*, *list_size=None*)

Operators.**assert_is_list_of_streams**(*x*)

Operators.**assert_is_list_of_streams_or_None**(*x*)

Operators.**assert_is_list_or_None**(*x*)

Operators.**asynch_element_agent**(*f*, *inputs*, *outputs*, *state*, *call_streams*, *window_size*, *step_size*)

Operators.**asynch_element_func**(*f*, *inputs*, *num_outputs*, *state*, *call_streams=None*, *window_size=None*, *step_size=None*)

Operators.**awf**(*inputs*, *outputs*, *func*, *window_size*, *step_size*, *state=None*, *call_streams=None*, ***kwargs*)

Operators.**dynamic_window_agent**(*f*, *input_stream*, *output_stream*, *state*, *min_window_size*, *max_window_size*, *step_size*)

Operators.**dynamic_window_func**(*f*, *inputs*, *state*, *min_window_size*, *max_window_size*, *step_size*)

Operators.**ef**(*inputs*, *outputs*, *func*, *state=None*, *call_streams=None*, ***kwargs*)

Operators.**element_agent**(*f*, *inputs*, *outputs*, *state*, *call_streams*, *window_size*, *step_size*)

Operators.**element_func**(*f*, *inputs*, *num_outputs*, *state*, *call_streams*, *window_size*, *step_size*)

Operators.**h**(*f_type*, **args*)
Calls the appropriate wrapper function given the name of the wrapper. The wrapper functions are list_func, element_func, window_func, ... for wrapper names 'list', 'element', 'window',..

Operators.**h_agent**(*f_type*, **args*)
Calls the appropriate wrapper function given the name of the wrapper. The wrapper functions are list_agent, element_agent, window_agent, ... for wrapper names 'list', 'element', 'window',..

Operators.**lf**(*inputs*, *outputs*, *func*, *state=None*, *call_streams=None*, ***kwargs*)

Operators.**list_agent**(*f*, *inputs*, *outputs*, *state*, *call_streams*, *window_size*, *step_size*)

Operators.**list_func**(*f*, *inputs*, *num_outputs*, *state*, *call_streams*, *window_size*, *step_size*)

Operators.**list_index_for_timestamp**(*in_list*, *start_index*, *timestamp*)
A helper function for timed operators. The basic idea is to return the earliest index in in_list.list[start_index:in_list.stop] with a time field that is greater than or equal to timestamp. If no such index exists then return a negative number.

Parameters **in_list: InList**

InList = namedtuple('InList', ['list', 'start', 'stop']) A slice into a stream.

start_index: nonnegative integer

A pointer into in_list.list

timestamp: number

Returns Returns positive integer i where:

either: 'FOUND TIME WINDOW IN IN_LIST'

i >= start_index and i <= in_list.stop and (in_list[start_index] >= timestamp or in_list.list[i-2][0] < timestamp <= in_list.list[i-1][0])

)

or: 'NO TIME WINDOW IN IN_LIST'

i < 0 (negative i indicates no time window) and

(in_list.list[in_list.stop-1] <= timestamp or

the list is empty, i.e. (in_list.start = in_list.stop)

Operators.**main**()

Operators.**many_outputs_source**(*f_type*, *f*, *num_outputs*, *state*, *call_streams*, *window_size*, *step_size*)

Operators.**many_to_many**(*f_type*, *f*, *in_streams*, *num_outputs*, *state*, *call_streams*, *window_size*, *step_size*)

Operators.**many_to_many_agent**(*f_type*, *f*, *in_streams*, *out_streams*, *state*, *call_streams*, *window_size*, *step_size*)

Operators.**merge**(*f_type*, *f*, *in_streams*, *state*, *call_streams*, *window_size*, *step_size*)

Operators.**merge_agent**(*f_type*, *f*, *in_streams*, *out_stream*, *state*, *call_streams*, *window_size*, *step_size*)

Operators.**op**(*f_type*, *f*, *in_stream*, *state*, *call_streams*, *window_size*, *step_size*)

Operators.**op_agent**(*f_type*, *f*, *in_stream*, *out_stream*, *state*, *call_streams*, *window_size*, *step_size*)

Operators.**remove_novalue_and_open_multivalue**(*l*)

This function returns a list which is the same as the input parameter l except that (1) _no_value elements in l are deleted and (2) each _multivalue element in l is opened

i.e., for an object _multivalue(list_x) each element of list_x appears in the returned list.

Operators.**single_output_source**(*f_type*, *f*, *num_outputs*, *state*, *call_streams*, *window_size*, *step_size*)

Operators.**single_output_source_agent**(*f_type*, *f*, *out_stream*, *state*, *call_streams*, *window_size*, *step_size*)

Operators.**sink**(*f_type*, *f*, *in_stream*, *state*, *call_streams*, *window_size*, *step_size*)

Operators.**sink_merge**(*f_type*, *f*, *in_streams*, *state*, *call_streams*, *window_size*, *step_size*)

Operators.**split**(*f_type*, *f*, *in_stream*, *num_outputs*, *state*, *call_streams*, *window_size*, *step_size*)

Operators.**split_agent**(*f_type*, *f*, *in_stream*, *out_streams*, *state*, *call_streams*, *window_size*, *step_size*)

Operators.**stream_agent**(*inputs*, *outputs*, *f_type*, *f*, *state=None*, *call_streams=None*, *window_size=None*, *step_size=None*)

Provides a common signature for converting functions f on standard Python data structures to streams.

Parameters **f_type** : {'element', 'list', 'window', 'timed', 'asynch_element'}

f_type identifies the type of function f where f is the next parameter.

f : function

inputs : {Stream, list of Streams}

When stream_func has: no input streams, inputs is None a single input Stream, inputs is a single Stream multiple input Streams, inputs is a list of Streams.

outputs : list of Streams

state : object

state is None or is an arbitrary object. The state captures all the information necessary to continue processing the input streams.

call_streams : None or list of Stream

If call_streams is None then the program sets it to inputs (converting inputs to a list of Stream if necessary). This function is called when, and only when any stream in call_streams is modified.

window_size : None or int

window_size must be a positive integer if f_type is 'window' or 'timed'. window_size is the size of the moving window on which the function operates.

step_size : None or int

step_size must be a positive integer if f_type is 'window' or 'timed'. step_size is the number of steps by which the moving window moves on each execution of the function.

Returns None

Operators.**stream_func**(*inputs*, *f_type*, *f*, *num_outputs*, *state=None*, *call_streams=None*, *window_size=None*, *step_size=None*)

Provides a common signature for converting functions f on standard Python data structures to streams.

Parameters **f_type** : {'element', 'list', 'window', 'timed', 'asynch_element'}

f_type identifies the type of function f where f is the next parameter.

f : function

inputs : {Stream, list of Streams}

When stream_func has: no input streams, inputs is None a single input Stream, inputs is a single Stream multiple input Streams, inputs is a list of Streams.

num_outputs : int

A nonnegative integer which is the number of output streams of this function.

state : object

state is None or is an arbitrary object. The state captures all the information necessary to continue processing the input streams.

call_streams : None or list of Stream

If call_streams is None then the program sets it to inputs (converting inputs to a list of Stream if necessary). This function is called when, and only when any stream in call_streams is modified.

window_size : None or int

window_size must be a positive integer if f_type is 'window' or 'timed'. window_size is the size of the moving window on which the function operates.

step_size : None or int

step_size must be a positive integer if f_type is 'window' or 'timed'. step_size is the number of steps by which the moving window moves on each execution of the function.

Returns list of Streams

Function f is applied to the appropriate data structure in the input streams to put values in the output streams. stream_func returns the output streams.

Operators.**tf**(*inputs*, *outputs*, *func*, *window_size*, *step_size*, *state=None*, *call_streams=None*, ***kwargs*)

Operators.**timed_agent**(*f*, *inputs*, *outputs*, *state*, *call_streams*, *window_size*, *step_size*)

Operators.**timed_func**(*f*, *inputs*, *num_outputs*, *state*, *call_streams*, *window_size*, *step_size*)

Operators.**wf**(*inputs*, *outputs*, *func*, *window_size*, *step_size*, *state=None*, *call_streams=None*, ***kwargs*)

Operators.**window_agent**(*f*, *inputs*, *outputs*, *state*, *call_streams*, *window_size*, *step_size*)

Operators.**window_func**(*f*, *inputs*, *num_outputs*, *state*, *call_streams*, *window_size*, *step_size*)

CHAPTER 5

RemoteQueue module

CHAPTER 6

Stream module

This module contains the Stream class. The Stream and Agent classes are the building blocks of PythonStreams. (Version 1.0 June 16, 2016. Created by: Mani Chandy)

class Stream.**Stream**(*name='NoName'*, *proc_name='UnknownProcess'*, *initial_value=[]*, *num_in_memory=16*, *min_history=4*)

Bases: object

A stream is a sequence of values. Agents can: (1) Append values to the tail of stream and close a stream. (2) Read a stream. (3) Subscribe to be notified when a value is added to a stream. (See Agent.py for details of agents.)

The ONLY way in which a stream can be modified is that values can be appended to its tail. The length of a stream (number of elements in its sequence) can stay the same or increase, but never decrease. If at some point, the length of a stream is k, then from that point onwards, the first k elements of the stream remain unchanged.

A stream is written by only one agent. Any number of agents can read a stream, and any number of agents can subscribe to a stream. An agent can be a reader and a subscriber and a writer of the same stream. An agent may subscribe to a stream without reading the stream's values; for example the agent may subscribe to a clock stream and the agent executes a state transition when the the clock stream has a new value, regardless of the value.

Parameters **name** : str, optional

name of the stream. Though the name is optional a named stream helps with debugging. default : 'NoName'

proc_name : str, optional

The name of the process in which this agent executes. default: 'UnknownProcess'

initial_value : list or array, optional

The list (or array) of initial values in the stream. If a stream starts in a known state, i.e., with a known sequence of messages, then set the initial_value to this sequence. default : []

num_in_memory: positive int, optional

It is the initial value of the number of elements in the stream that are stored in main memory. If the stream has 9000 elements and num_in_memory is 100 then the most recent 100 elements of the stream are stored in main memory and the earlier 8900 elements are stored in a file or discarded. num_in_memory may change. It increases if a reader is reading the i-th value of the stream and if j is the index of the most recent value in the stream and **|j - i|** exceeds num_in_memory. It may decrease if the gap between the indices of the most recent value in the stream and the earliest value being read by any agent is less than num_in_memory default : DEFAULT_NUM_IN_MEMORY

specified in SystemParameters.py

min_history: non-negative int, optional

The minimum number of elements of the stream that are stored in main memory. If min_history is 3 and the stream has 9000 elements then the elements 8997, 8998, 8999 will be stored in main memory even if no agent is reading the stream. min_history is used primarily for debugging. A debugger may need to read early values of the stream, and reading from main memory is faster than reading from a file. default : DEFAULT_MIN_HISTORY

specified in SystemParameters.py.

Notes

1.AGENTS SUBSCRIBING TO A STREAM

An agent is a state-transition automaton and the only action that an agent executes is a state transition. If agent x is a subscriber to a stream s then x.next() — a state transition of x — is invoked whenever messages are appended to s.

The only point at which an agent executes a state transition is when a stream to which the agent subscribes is modified.

An agent x subscribes to a stream s by executing s.call(x).

An agent x unsubscribes from a stream s by executing:

s.delete_caller(x)

2.AGENTS READING A STREAM

2.1 Agent registers for reading

An agent can read a stream only after it registers with the stream as a reader. An agents r registers with a stream s by executing:

s.reader(r)

An agent r deletes its registration for reading s by executing:

s.delete_reader(r)

An agent that reads a stream is also a subscriber to that stream unless the agent has a call-stream. If an agent has no call-stream and stream s is an input stream of that agent, then whenever s is modified, the agent is told to execute a state transition.

2.2 Slice of a stream that can be read by an agent

At any given point, an agent r that has registered to read a stream s can only read some of the most recent values in the stream. The number of values that an agent can read may vary from agent to agent. A reader r can only read a slice:

s[s.start[r]+s.offset: s.stop+s.offset]

of stream s where start[r], stop and offset are defined later.

3.WRITING A STREAM

3.1 Extending a stream

When an agent is created it is passed a list of streams that it can write.

An agent adds a single element v to a stream s by executing:

s.append(v)

An agent adds the sequence of values in a list l to a stream s by executing:

s.extend(l)

The operations append and extend of streams are analogous to operations with the same names on lists.

3.2 Closing a Stream

A stream is either closed or open. Initially a stream is open. An agent that writes a stream s can close s by executing:

s.close()

A closed stream cannot be modified.

4.MEMORY

4.1 The most recent values of a stream

The most recent elements of a stream are stored in main memory. In addition, the user can specify whether all or part of the stream is saved to a file.

Associated with each stream s is a list (or array) s.recent that includes the most recent elements of s. If the value of s is a sequence:

s[0], ..., s[n-1],

at a point in a computation then at that point, s.recent is a list

s[m], .., s[n-1]

for some m, followed by some padding (usually a sequence of zeroes, as described later).

The system ensures that all readers of stream s only read elements of s that are in s.recent.

4.2 Slice of a stream that can be read

Associated with a reader r of stream s is an integer s.start[r]. Reader r can only read the slice:

s.recent[s.start[r] :]

of s.recent.

For readers r1 and r2 of a stream s the values s.start[r1] and s.start[r2] may be different.

4.3 When a reader finishes reading part of a stream

Reader r informs stream s that it will only read values with indexes greater than or equal to j in the list, recent, by executing

s.set_start(r, j)

which causes s.start[r] to be set to j.

5.OPERATION

5.1 Memory structure

Associated with a stream is: (1) a list or NumPy darray, recent. (2) a nonnegative integer stop where:

1.recent[: stop] contains the most recent values of the stream,

2.the slice recent[stop:] is padded with padding values (either 0 or 0.0 or default value specified by the numpy data type).

3.**a nonnegative integer s.offset where**

recent[i] = stream[i + offset] for 0 <= i < s.stop

Example: if the sequence of values in a stream is:

0, 1, .., 949

and s.offset is 900, then s.recent[i] = s[900+i] for i in 0, 1, ..., 49.

and s.recent[i] is the default value for i > 49.

Invariant: len(s) = s.offset + s.stop

where len(s) is the number of values in stream s.

The size of s.recent increases and decreases so that the length of slice that any reader may read is less than the length of s.recent.

The entire stream, or the stream up to offset, can be saved in a file for later processing. You can also specify that no part of the stream is saved to a file.

In the current implementation old values of the stream are not saved.

5.2 Memory Management

We illustrate memory management with the following example with num_in_memory=4 and buffer_size=1

Assume that a point in time, for a stream s, the list of values in the stream is [1, 2, 3, 10, 20, 30]; num_in_memory=4; s.offset=3; s.stop=3; and s.recent = [10, 20, 30, 0]. The length of s.recent is num_in_memory (i.e. 4). The s.stop (i.e. 3) most recent values in the stream are 10, 20, 30. s[3] == 10 == s.recent[0] s[4] == 20 == s.recent[1] s[5] == 30 == s.recent[2] The values in s.recent[s.stop:] are padded values (zeroes).

A reader r of stream s has access to the list: s.recent[s.start[r] : s.stop]

Assume that s has two readers r and q where: s.start[r] = 1 and s.start[q] = 2. So agent r can read the slice [1:3] of recent which is the list [20, 30], and agent q can read the slice [2:3] of recent which is the list [30]. An invariant is:

0 <= s.start[r] <= s.stop

for any reader r.

When a value v is appended to stream s, v is inserted in s.recent[s.stop], replacing a default value, and s.stop is incremented. If s.stop >= len(s.recent) then a new s.recent is created and the values that may be read by any reader are copied into the new s.recent, and s.start, s.stop, and s._begin are modified.

Example: Start with the previous example. (Assume min_history is 0. This parameter is discussed in the next paragraph.) When a new value, 40 is appended to the stream, the list of values in s becomes. [1, 2, 3, 10, 20, 30, 40]. s.recent becomes [10, 20, 30, 40], and s.stop becomes 4. Since s.stop >= len(recent), a new copy of recent

is made and the elements that are being read in s are copied into the new copy. So, recent becomes [20, 30, 40, 0] because no agent is reading s[3] = 10. Then s.stop becomes 3 and s.offset becomes 4. s.start is modified with s.start[r] becoming 0 and s.start[q] becoming 1, so that r continues to have access to values of the stream after 20; thus r can now read the list [20, 30, 40] and q can read the list [30, 40].

At a later point, agent r informs the stream that it no longer needs to access elements 20, 30, 40 and so s.start[r] becomes 3. Later agent q informs the stream that it no longer needs to access element 30 and s.start[q] becomes 2. At this point r has access to the list [] and q to the list [40].

Now suppose the agent writing stream s extends the stream by the list [50, 60, 70, 80]. At this point, agent q needs to access the list [40, 50, 60, 70, 80] which is longer than len(recent). In this case the size of recent is doubled, and the new recent becomes: [40, 50, 60, 70, 80, 0, 0, 0], with s.start[r] = 1 and s.start[q] = 0. s.stop becomes 5.

Example of min_history = 4. Now suppose the stream is extended by 90, 100 so that s.recent becomes [40, 50, 60, 70, 80, 90, 100, 0] with s.stop = 7. Suppose r and q inform the stream that they no longer need to access the elements currently in the stream, and so s.start[r] and s.start[q] become 7. (In this case the size of recent may be made smaller (halved); but, this is not done in the current implementation and will be done later.) Next suppose the stream is extended by [110]. Since r and q only need to read this value, all the earlier values could be deleted from recent; however, min_history elements must remain in recent and so recent becomes: [80, 90, 100, 110, 0, 0, 0, 0]

Attributes

recent	(list or NumPy array.) A list or array that includes the most recent values of the stream. This list or array is padded with default values (see stop).
stop	(int) index into the list recent. s.recent[:s.stop] contains the s.stop most recent values of stream s. s.recent[s.stop:] contains padded values.
offset: int	index into the stream used to map the location of an element in the entire stream with the location of the same element in s.recent, which only contains the most recent elements of the stream. For a stream s: s.recent[i] = s[i + s.offset] for i in range(s.stop) Note: In later versions, offset will be implemented as a list of ints.
start	(dict of readers.) key = reader value = start index of the reader Reader r can read the slice: s.recent[s.start[r] : s.stop] in s.recent which is equivalent to the following slice in the entire stream: s[s.start[r]+s.offset: s.stop+s.offset] Invariant: For all readers r: stop - start[r] <= len(recent) This invariant is maintained by increasing the size of recent when necessary.
subscribers_set: set	the set of subscribers for this stream. Subscribers are agents to be notified when an element is added to the stream.
closed: boolean	True if and only if the stream is closed. An exception is thrown if a value is appended to a closed stream.
close_message: _close or np.NaN	This message is appended to a stream to indicate that when this message is received the stream should be closed. If the stream is implemented as a list then close_message is _close, and for StreamArray the close_message is np.NaN (not a number).
_begin	(int) index into the list, recent recent[:_begin] is not being accessed by any reader; therefore recent[:_begin] can be deleted from main memory. Invariant: for all readers r: _begin <= min(start[r])

Methods

append(value)	Append a single value to the end of the stream.

Continued on next page

Table 6.1 – continued from previous page

call(agent)	Register a subscriber for this stream.
close()	Close this stream."
delete_caller(agent)	Delete a subscriber for this stream.
delete_reader(reader)	Delete this reader from this stream.
extend(value_list)	Extend the stream by value_list.
get_contents_after_column_value(...)	Assumes that the stream consists of rows where the number of elements in each row exceeds column_number.
get_index_for_column_value(column_number, value)	Similar to get_contents_after_column_value except that the value returned is an index into recent rather than the sequence of rows.
get_last_n(n)	**Parameters**
get_latest()	Returns the latest element in the stream.
get_latest_n(n)	Same as get_last_n()
is_empty()	Returns: boolean ——- True if and only if this stream is empty.
print_recent()	
reader(r[, start_index])	A newly registered reader starts reading recent from index start, i.e., reads recent[start_index:s.stop] If reader has already been registered with this stream its start value is updated to start_index.
set_name(name)	
set_start(reader, starting_value)	The reader tells the stream that it is only accessing elements of the list recent with index start or higher.

append(*value*)
Append a single value to the end of the stream.

call(*agent*)
Register a subscriber for this stream.

close()
Close this stream."

delete_caller(*agent*)
Delete a subscriber for this stream.

delete_reader(*reader*)
Delete this reader from this stream.

extend(*value_list*)
Extend the stream by value_list.

Parameters value_list: list

get_contents_after_column_value(*column_number*, *value*)
Assumes that the stream consists of rows where the number of elements in each row exceeds column_number. Also assumes that values in the column with index column_number are in increasing order.

Returns the rows in the stream for which: row[column_number] >= value

get_index_for_column_value(*column_number*, *value*)
Similar to get_contents_after_column_value except that the value returned is an index into recent rather than the sequence of rows.

get_last_n(*n*)

Parameters **n** : positive integer

Returns The list of the last n elements of the stream. If the

number of elements in the stream is less than n, then it returns all the elements in the stream.

Note

Requirement: n >= self.min_history

get_latest()
Returns the latest element in the stream. If the stream is empty then it returns the empty list

get_latest_n(*n*)
Same as get_last_n()

is_empty()

True if and only if this stream is empty.

print_recent()

reader(*r*, *start_index=0*)
A newly registered reader starts reading recent from index start, i.e., reads recent[start_index:s.stop] If reader has already been registered with this stream its start value is updated to start_index.

set_name(*name*)

set_start(*reader*, *starting_value*)
The reader tells the stream that it is only accessing elements of the list recent with index start or higher.

class Stream.**StreamArray**(*name='NoName'*, *proc_name='UnknownProcess'*, *dimension=0*, *dtype=<type 'float'>*, *initial_value=None*, *num_in_memory=16*, *min_history=4*)
Bases: *Stream.Stream*

Methods

append(value)	**Parameters**
call(agent)	Register a subscriber for this stream.
close()	Close this stream."
delete_caller(agent)	Delete a subscriber for this stream.
delete_reader(reader)	Delete this reader from this stream.
extend(lst)	See extend() for the class Stream.
get_contents_after_column_value(...)	Assumes that the stream consists of rows where the number of elements in each row exceeds column_number.
get_contents_after_time(start_time)	
get_index_for_column_value(column_number, value)	Similar to get_contents_after_column_value except that the value returned is an index into recent rather than the sequence of rows.

Continued on next page

Table 6.2 – continued from previous page

get_last_n(n)	**Parameters**
get_latest()	Returns the latest element in the stream.
get_latest_n(n)	Same as get_last_n()
is_empty()	Returns: boolean ——- True if and only if this stream is empty.
print_recent()	
reader(r[, start_index])	A newly registered reader starts reading recent from index start, i.e., reads recent[start_index:s.stop] If reader has already been registered with this stream its start value is updated to start_index.
set_name(name)	
set_start(reader, starting_value)	The reader tells the stream that it is only accessing elements of the list recent with index start or higher.

append(*value*)

Parameters value: 1-D numpy array

The value appended to the StreamArray

Notes

See self._create_recent() for a description of the elements of the stream.

extend(*lst*)

See extend() for the class Stream. Extend the stream by an numpy ndarray.

Parameters lst: np.ndarray

Notes

See self._create_recent() for a description of the elements of the stream.

get_contents_after_time(*start_time*)

class Stream.**StreamSeries**(*name=None*)

Bases: *Stream.Stream*

Methods

append(value)	Append a single value to the end of the stream.
call(agent)	Register a subscriber for this stream.
close()	Close this stream.”
delete_caller(agent)	Delete a subscriber for this stream.
delete_reader(reader)	Delete this reader from this stream.
extend(value_list)	Extend the stream by value_list.

Continued on next page

Table 6.3 – continued from previous page

get_contents_after_column_value(...)	Assumes that the stream consists of rows where the number of elements in each row exceeds column_number.
get_index_for_column_value(column_number, value)	Similar to get_contents_after_column_value except that the value returned is an index into recent rather than the sequence of rows.
get_last_n(n)	**Parameters**
get_latest()	Returns the latest element in the stream.
get_latest_n(n)	Same as get_last_n()
is_empty()	Returns: boolean ——- True if and only if this stream is empty.
print_recent()	
reader(r[, start_index])	A newly registered reader starts reading recent from index start, i.e., reads recent[start_index:s.stop] If reader has already been registered with this stream its start value is updated to start_index.
set_name(name)	
set_start(reader, starting_value)	The reader tells the stream that it is only accessing elements of the list recent with index start or higher.

class Stream.**StreamTimed**

Bases: *Stream.StreamArray*

Methods

append(value)	**Parameters**
call(agent)	Register a subscriber for this stream.
close()	Close this stream.”
delete_caller(agent)	Delete a subscriber for this stream.
delete_reader(reader)	Delete this reader from this stream.
extend(lst)	See extend() for the class Stream.
get_contents_after_column_value(...)	Assumes that the stream consists of rows where the number of elements in each row exceeds column_number.
get_contents_after_time(start_time)	
get_index_for_column_value(column_number, value)	Similar to get_contents_after_column_value except that the value returned is an index into recent rather than the sequence of rows.
get_last_n(n)	**Parameters**
get_latest()	Returns the latest element in the stream.
get_latest_n(n)	Same as get_last_n()
is_empty()	Returns: boolean ——- True if and only if this stream is empty.

Continued on next page

Table 6.4 – continued from previous page

print_recent()	
reader(r[, start_index])	A newly registered reader starts reading recent from index start, i.e., reads recent[start_index:s.stop] If reader has already been registered with this stream its start value is updated to start_index.
set_name(name)	
set_start(reader, starting_value)	The reader tells the stream that it is only accessing elements of the list recent with index start or higher.

class Stream.**TimeAndValue**

Bases: tuple

Attributes

time	Alias for field number 0
value	Alias for field number 1

Methods

count(...)	
index((value, [start, ...)	Raises ValueError if the value is not present.

time

Alias for field number 0

value

Alias for field number 1

Stream.**main**()

Stream.**remove_novalue_and_open_multivalue**(*l*)

This function returns a list which is the same as the input parameter l except that (1) _no_value elements in l are deleted and (2) each _multivalue element in l is opened

i.e., for an object _multivalue(list_x) each element of list_x appears in the returned list.

CHAPTER 7

SystemParameters module

SYSTEM_PARAMETERS

CHAPTER 8

examples_element_wrapper module

This module contains examples of the 'element' wrapper. A function that takes a single element of an input stream and generates a single element of an output stream is an example of a function that is wrapped by the 'element' wrapper to create a function on streams.

The module has the following parts: (1) op or simple operator:

single input, single output

2. **sink:** single input, no outputs
3. **source:** no input, multiple outputs
4. **split:** single input, multiple output
5. **merge:** multiple input, single output
6. **general case:** multiple input, multiple output

All of the first 5 cases are specializations of the general case; however, the syntactic sugar they provide can be helpful.

For each of the above cases we first consider agents that are stateless and then consider agents with state.

`examples_element_wrapper.`**`average`**(*v*, *state*)

Parameters **v** : number

The next element of the input stream of the agent.

state: (n, cumulative)

The state of the agent where n : number

The value of the next element in the agent's input stream.

cumulative [number] The sum of the values that the agent has received on its input stream.

Returns (mean, state)

mean : floating point number

The average of the values received so far by the agent

state : (n, cumulative)

The new state of the agent.

examples_element_wrapper.**average_stream**(*stream*)

examples_element_wrapper.**boolean_of_values_greater_than_threshold**(*stream*, *threshold*)

examples_element_wrapper.**cumulative_stream**(*stream*)

examples_element_wrapper.**cumulative_sum**(*v*, *cumulative*)

Parameters **v** : number

The next element of the input stream of the agent.

cumulative: number

The state of the agent. The state is the sum of all the values received on the agent's input stream.

Returns (cumulative, cumulative)

cumulative : number

The state after the transition, i.e., the sum of values received on the agent's input stream including the value received in this transition.

examples_element_wrapper.**discard_odds**(*stream*)

examples_element_wrapper.**discard_odds_1**(*stream*)

examples_element_wrapper.**double**(*v*)

examples_element_wrapper.**double_stream**(*stream*)

examples_element_wrapper.**even**(*v*)

examples_element_wrapper.**even2**(*v*)

examples_element_wrapper.**even3**(*v*)

examples_element_wrapper.**even_1**(*v*)

examples_element_wrapper.**even_odd**(*m*)

examples_element_wrapper.**even_odd_stream**(*stream*)

examples_element_wrapper.**evens_and_halves**(*stream*)

examples_element_wrapper.**evens_and_halves_3**(*stream*)

examples_element_wrapper.**exp_mult_div_stream**(*stream*, *exponent*, *multiplier*, *divisor*)

examples_element_wrapper.**file_to_stream**(*filename*, *output_stream*, *time_period=0*)

Parameters **filename: str**

output_stream: Stream

time_period: int or float, nonnegative

examples_element_wrapper.**inrange_and_outlier_streams**(*x_and_y_streams*, *A*, *B*, *DELTA*)

examples_element_wrapper.**join_streams**(*list_of_streams*)

examples_element_wrapper.**main**()

examples_element_wrapper.**max_and_min_seen_across_all_streams**(*list_of_streams*)

examples_element_wrapper.**max_and_min_stream**(*list_of_streams*)

examples_element_wrapper.**max_and_min_with_names**(*list_of_numbers*, *state*)

examples_element_wrapper.**max_of_all_inputs**(*list_of_elements*, *previous_max*)

examples_element_wrapper.**max_seen_across_all_streams**(*list_of_streams*)

examples_element_wrapper.**max_stream**(*list_of_streams*)

examples_element_wrapper.**max_with_index**(*list_of_numbers*, *state*)

examples_element_wrapper.**mean_stream**(*list_of_streams*)

examples_element_wrapper.**multiply_elements_in_stream**(*stream*, *multiplier*)

examples_element_wrapper.**print0**(*stream*)

examples_element_wrapper.**print_stream**(*stream*)

examples_element_wrapper.**print_streams**(*list_of_streams*)

examples_element_wrapper.**print_sums**(*list_of_streams*)

examples_element_wrapper.**rand**(*output_stream*, *num_outputs*, *time_period=0*)

Parameters **output_stream: Stream**

num_outputs: int, positive

time_period: int or float, positive

examples_element_wrapper.**single_stream_of_random_numbers**(*timer_stream*)

examples_element_wrapper.**square**(*v*)

examples_element_wrapper.**square_and_double**(*m*)

examples_element_wrapper.**square_and_double_stream**(*stream*)

examples_element_wrapper.**square_stream**(*stream*)

examples_element_wrapper.**stream_to_file**(*stream*, *filename*)

examples_element_wrapper.**temperature_and_humidity_streams**(*stream*, *DELTA*)

examples_element_wrapper.**timer**(*output_stream*, *num_outputs*, *time_period*)

Parameters **stream: Stream**

num_outputs: int, positive

time_period: int or float, positive

CHAPTER 9

examples_timed_window_wrapper module

examples_timed_window_wrapper.**avg_of_max_and_min_in_timed_list**(*timed_list*, *state*)

examples_timed_window_wrapper.**avg_of_max_and_min_values_in_all_timed_lists**(*list_of_timed_lists*, *state*)

examples_timed_window_wrapper.**diff_of_counts_in_lists**(*list_of_two_lists*)

examples_timed_window_wrapper.**exponential_smoothed_timed_windows**(*input_stream*, *func*, *alpha*, *window_size*, *step_size*, *initial_state*)

Parameters

input_stream: Stream A previously defined stream This is the only input stream of the agent.

func: function func operates on a list of TimeAndValue objects and returns an object that can be smoothed exponentially.

alpha: number The exponential smoothing parameter.

window_size, step_size, initial_state: Already defined.

examples_timed_window_wrapper.**max_and_min_of_values_in_timed_list**(*timed_list*)

examples_timed_window_wrapper.**max_and_min_values_in_all_timed_lists**(*list_of_timed_lists*)

examples_timed_window_wrapper.**max_sums_timed_windows**(*list_of_timed_lists*, *state*)

examples_timed_window_wrapper.**sum_values_in_all_timed_lists**(*list_of_timed_lists*)

examples_timed_window_wrapper.**sum_values_in_timed_list**(*timed_list*)

CHAPTER 10

examples_window_wrapper module

This module contains examples of the 'window' wrapper. A window wrapper wraps a function that has a parameter which is a list or a list of lists and that returns a value or a list of values. The wrapped function operates on a sliding window of a stream or a list of sliding windows of a list of streams, and returns a stream or a list of streams.

A sliding window is defined by a window size and a step size. An operation on a sliding window carries out its first operation only when the size of the stream is at least the window size. A window is a list of the specified size. An operation is carried out on the window; then the window is moved forward in the stream by the step size.

For example, if the operation on the window is sum, and the window size is 3 and the step size is 2, and the stream is [5, 7] at a point in time t0, then no window operation is carried at t0. If at a later time, t1, the stream is [5, 7, 8] then the sum operation is carried out on this window of size 3 to return 20 at t1. Then the window operation waits until the window steps forward by 2. If at a later time, t2, the stream is [5, 7, 8, 2], no operation is carried out at t2. At a later time t3, if the stream is [5, 7, 8, 2, 5] then an operation is carried out on the window [8, 2, 5] to give 15.

A window operation on multiple input streams waits until sliding windows are available on all the input streams. The window sizes and step sizes for all windows are identical.

The examples below deal with stateless and stateful cases of single and multiple input streams and single and multiple output streams. We don't use windows for sources or for sinks because simple elements are adequate for this purpose. Likewise, we don't use windows for asynchronous merges.

examples_window_wrapper.**avg_of_difference_of_two_windows**(*list_of_two_windows*, *state*)

examples_window_wrapper.**combine_windows**(*input_stream*, *weights*)

examples_window_wrapper.**diff_of_means_of_two_windows**(*list_of_two_windows*)

examples_window_wrapper.**exp_smoothing_mean_and_std_of_stream**(*stream*, *alpha*, *window_size*, *step_size*)

examples_window_wrapper.**extent**(*lst*)

examples_window_wrapper.**inrange_and_outlier_streams**(*list_of_two_streams*, *window_size*, *step_size*, *alpha*, *threshold*)

examples_window_wrapper.**max_and_min**(*lst*)

examples_window_wrapper.**mean_inc**(*lst*, *state*)

examples_window_wrapper.**mean_of_window**(*stream*)

CHAPTER 11

source_stream module

source_stream.**main**()

source_stream.**source_stream**(*output_stream*, *number_of_values*, *time_period*, *func*, ***kwargs*)

Periodically appends a value to output_stream. The values appended are obtained by calling the function func and passing it keyword arguments.

If number_of_values is non-negative, then it is the maximum number of values inserted into output_stream. If number_of_values is negative then values are appended to output_stream forever.

If time_period is 0 then number_of_values must be non-negative; in this case all the values are appended to output_stream when source_stream is called.

Parameters **output_stream: Stream**

(Could also be a list.) The stream to which values are appended. (Note: Appending messages to a list forever will cause memory

overflow.)

number_of_values: int

time_period: int or float, nonnegative

The time between successive appends to output_stream.

func: function

The return value of this function is appended to output_stream.

CHAPTER 12

tutorial_average_of_stream module

tutorial_average_of_stream.**update_avg**(*v*, *state*)

CHAPTER 13

tutorial_element_wrapper_stream_operator module

CHAPTER 14

tutorial_exponential_smoothing module

tutorial_exponential_smoothing.**update_smooth**(*lst*, *state*, *h*, *alpha*)

CHAPTER 15

tutorial_filter_1 module

tutorial_filter_1.**ft**(*v*, *threshold*)

CHAPTER 16

tutorial_many_to_many module

CHAPTER 17

tutorial_merge module

tutorial_merge.**dif**(*lst*)

CHAPTER 18

tutorial_sine module

CHAPTER 19

tutorial_split module

tutorial_split.**execute**(v, g, h)

CHAPTER 20

weave_streams module

CHAPTER 21

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W