# INTRODUCTION

This document describes an open-source Python 2.7 software package, PStreams, for processing streams of data.

### Goals of the package

1. Enable programmers to continue to use familiar data structures such as Python lists and NumPy arrays, and then transform their programs to operate on data streams. Programmers first focus on the logic of their code using fixed-size data structures, and later extend the logic to streaming systems in which new data may continue to arrive as time passes. Working with standard Python data structures such as lists and arrays simplifies the use of popular Python science and data-mining packages such as SciPy and SciKit-Learn for streams. The goal is to separate concerns of the underlying logic from that of working with the stream data structure: the first concern is to develop logic with fixed data structures and later extend the logic to deal with continuously changing streams.
2. Enable programmers to develop and test programs in a single process and later map software components of the program to hardware components in a distributed system. These hardware components include: single-board computers such as the Raspberry Pi, cloud services, and processing elements in the network connecting sensors to the cloud. The goal here too, is separation of concerns: the first concern is to develop the logic and second concern is to distribute the logic across multiple computing elements.
3. Separate concerns of processing logic from system control. The logic of a streaming application deals with the definitions of functions used in the application whereas control addresses issues such as synchronizing clocks, time stamps, communication protocols, fault tolerance and archival storage.

This document describes a Python streaming package. It is not an exposition of stream processing, distributed computing or Python. This introductory chapter does not discuss performance issues, incremental computation on streams, distributed computing, or data-mining packages. Of the three goals listed above, this chapter deals only with the first: converting functions that operate on Python data structures to functions that operate on streams.

(One way to navigate this document is to use *Bookmarks*. Click on *Insert* on the tool bar, then click on *Bookmark*, and finally click on the section that interests you in the Bookmark list. For example, if you want to look at the subsection on operations into moving windows of data streams then click on the *Bookmark* *WINDOWS\_SECTION*.)

### Streams

A *stream* is a sequence of values. The individual elements in a stream are called *messages*. The only way in which a stream can be modified is that messages can be appended to the end of a stream. A message in a stream cannot be modified, and messages within a stream cannot be reordered. An example of a stream is a sequence of measurements made by a sensor; as time progresses the sensor may make additional measurements and send messages containing measured values. The sensor cannot retract or modify messages that it has sent.

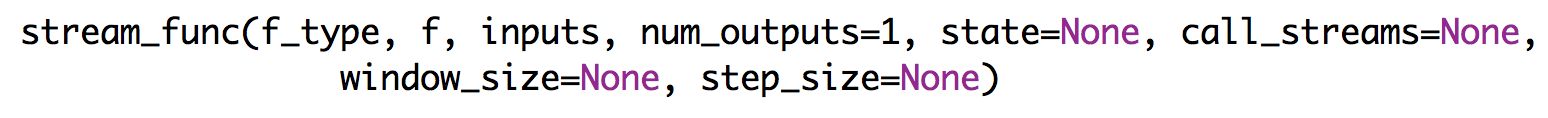
The messages in a stream are arbitrary objects; messages in a stream need not belong to a special class. Some messages in a stream may have timestamps and locations while other messages may not.

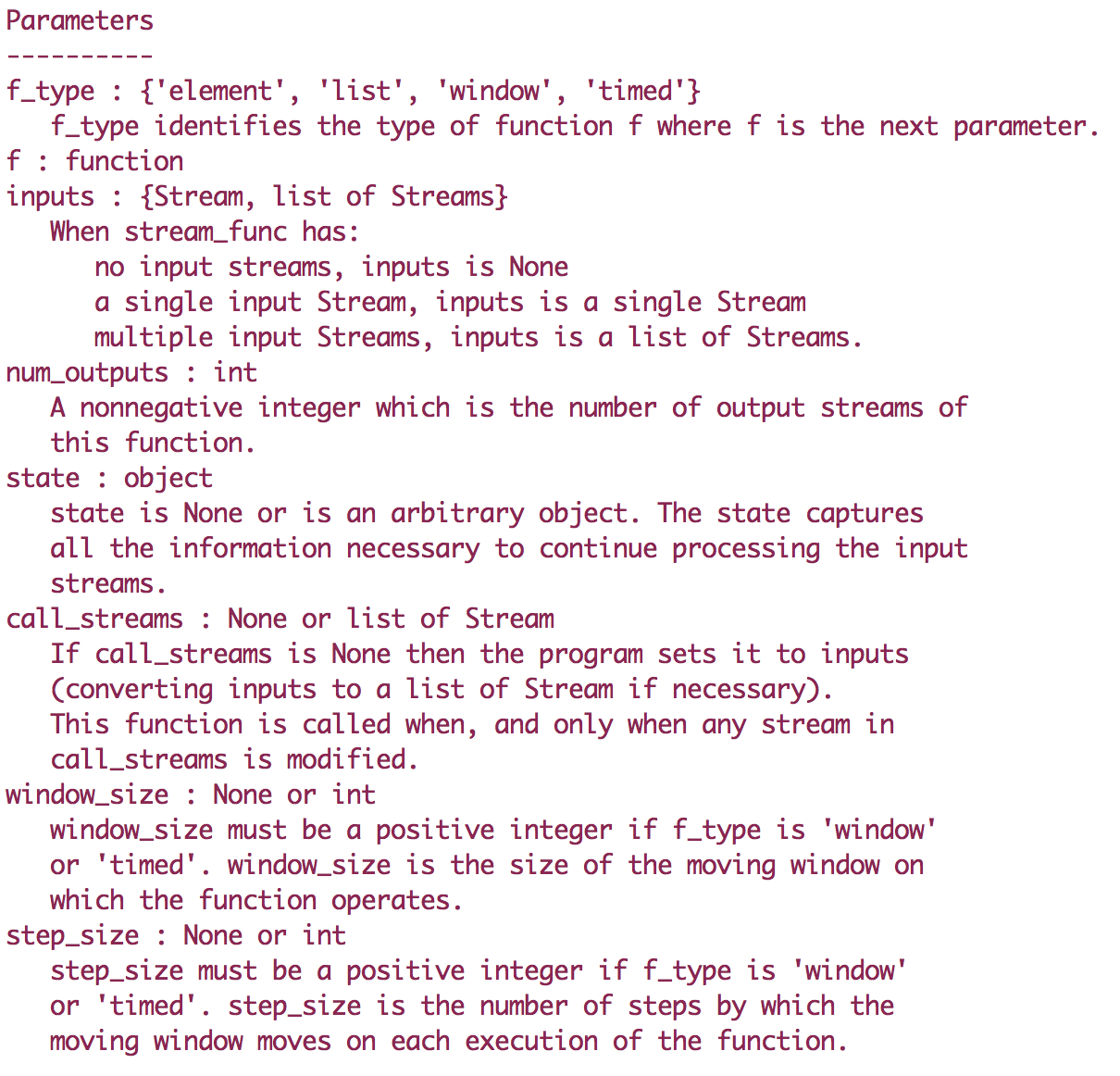
The value of a stream, at a point in time, is a Python list or a NumPy array. This list or array is extended when messages are appended to the stream. If at some point, the value of a stream is the list [3, 5], then from that point onwards, the value of that stream will be a list that begins with [3, 5]. For example, at a later point in time the stream can be [3, 5], or [3, 5, 2], or [3, 5, 2, 6], but not [3, 4] or [1, 5, 2].

This chapter shows how to convert functions on (i) elementary Python objects, (ii) lists (iii) NumPy arrays, (iv) moving windows, and (v) timed windows into functions on streams. We first discuss elementary objects, lists and NumPy arrays, and we consider windows later.

### From Objects, Lists, and Arrays to Streams

Next, we describe a way of using a function on lists to define a function on streams. We begin by developing functions on streams where the functions have a single parameter that is either a stream or a list of streams and the functions return a either stream or a list of streams. To develop a function g on streams we first develop a function *f* on objects that are elements of the streams. We use the following function to define *g*:





# ELEMENTS

We begin by looking at cases where *f\_type* is ‘*element’*. We start with examples in which all the arguments with default values, other than *num\_outputs*, are set to their defaults. So, we only need to specify the parameters *f*, *inputs* and *num\_outputs*.

We develop a function *g* on streams by first developing a function *f* on elements of streams and then use *f* with *stream\_func* to obtain *g*. The relationship between f and g is described next.

Let and be the lists of the n-th elements of the input and output streams, respectively. Let be the n-th value of the state of the computation. The initial state is passed as the parameter in stream\_func. The output and subsequent states of the computation are defined recursively as:

In some cases, f is stateless, i.e.,

in which case the state is not passed as a parameter. A function that is not stateless is called *stateful*.

For convenience, if *g* has a single input stream, then instead of passing *f* a list consisting of a single element, *f* is passed the element itself without enclosing it in a list. Likewise, if *g* has a single output stream, then *f* returns an element of the output stream without enclosing the element in a list.

##### Single Input and Single Output Stream

Let’s begin by considering a function g with a single input and a single output stream. Since g has a single output stream *num\_outputs* is set to 1.

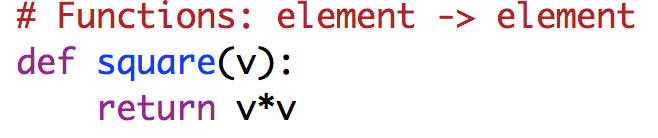
Since the stream function *g* that we are developing has a single input stream and a single output stream, we first develop a function *f* from a single object to a single object for stateless computations, and from (object, state) to (object, state) for stateful computations. For stateless computations,

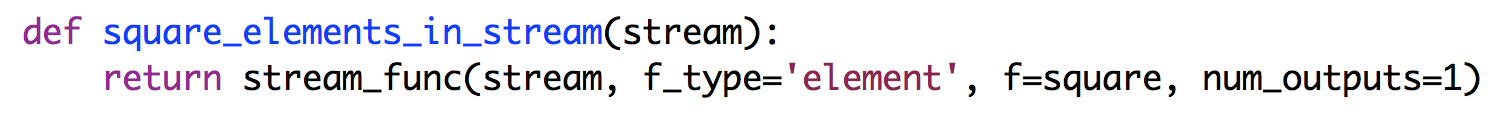
#### Example

We want to write a function where the n-th element of the output stream is the square of the n-th element of the input stream.

Note that x and y are scalars, not lists, in the equaation.

We first develop function *square* that squares numbers and then use it to create a function *square\_elements\_in\_stream* that has a single input stream *x* and generates a single output stream *y* that satisfies the above formula.





Then if x is a stream with values 0, 1, 2, 3, … and



then *r* will be a stream with values 0, 1, 4, 9, …

You can also use Python’s *partial* functions to define a function as in:





You can use *stream\_square* to create a stream *u* of squares of stream *x*:



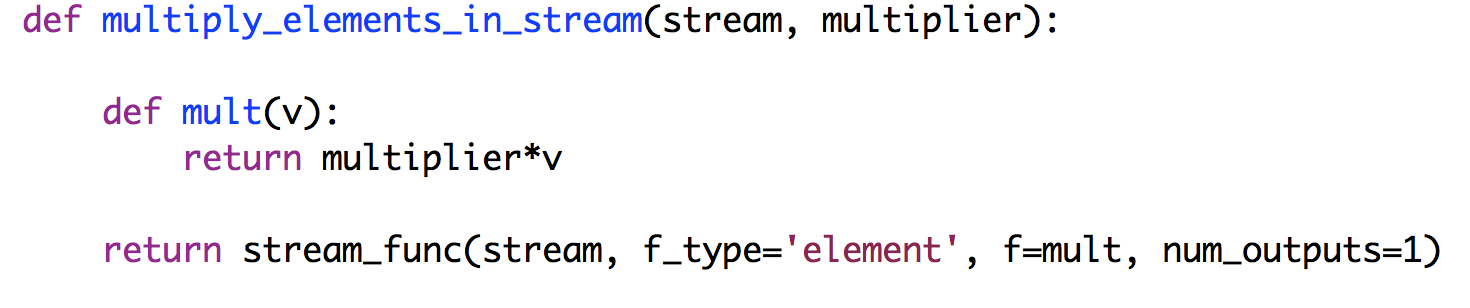
Functions on streams are ordinary Python functions and are used in the standard ways of Python. For example, you can compose functions, such as:



where v is a stream whose elements are the corresponding elements of x raised to the fourth power.

#### Example

Let’s look at another example. We want to write a function, *multiply\_elements\_in\_stream,* which has a single input stream and a single output stream, and which has a parameter, *multiplier*, where the elements of the output stream are *multiplier* times the corresponding elements of the input stream. We write a function *mult* that multiplies numbers and then use *mult* with *stream\_func* to define the desired function on streams.



You can now create a stream *b* whose elements are 3 times the elements of its input stream *x* as:



#### Example with State

Given a stream *x* we want to generate a stream *y* where the elements of *y* are the average of the elements of *x* up to that point, i.e., for n = 0, 1, 2,..:

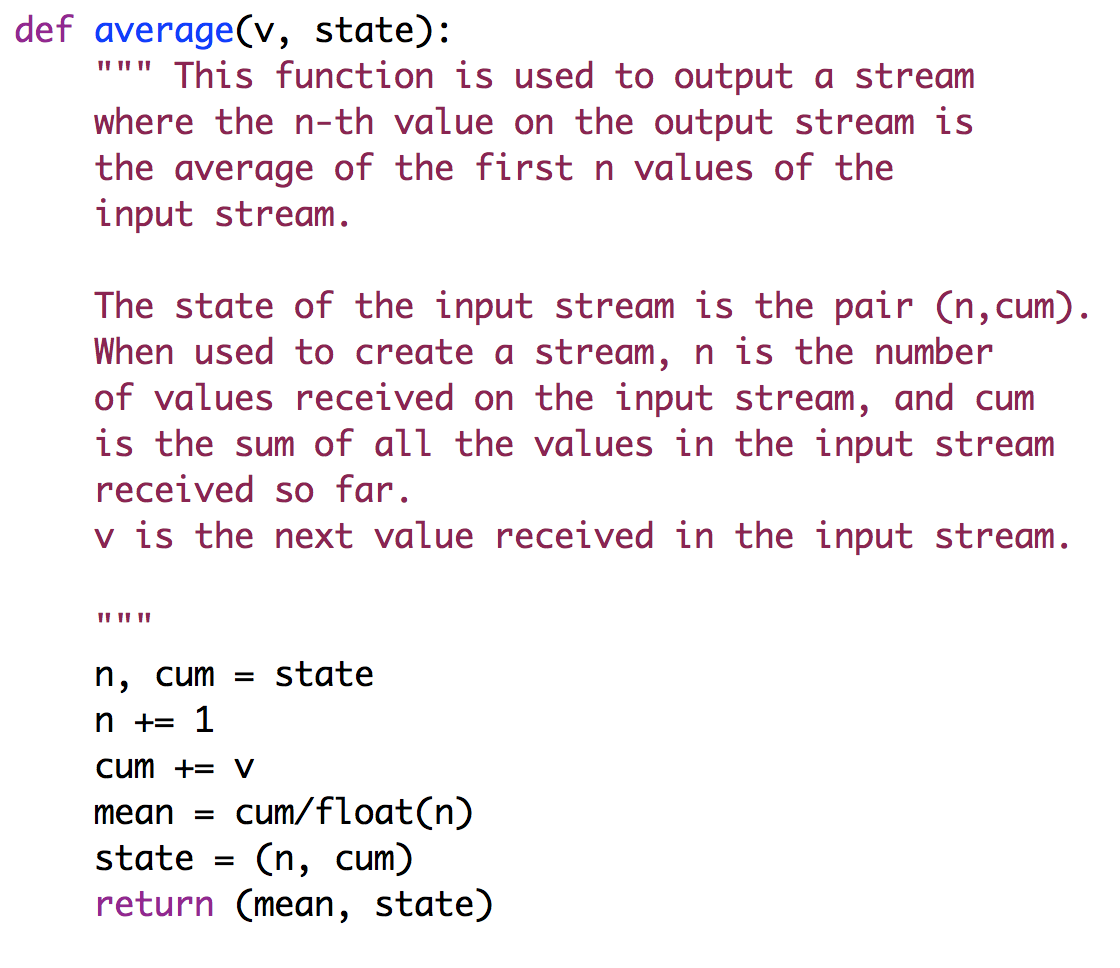
The computation of *y*[n] requires state[n], the state of the computation at step n:

because

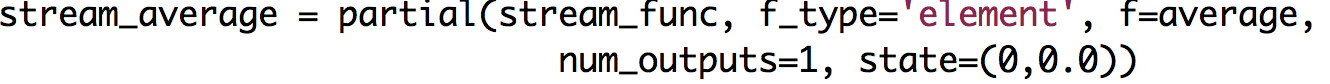
and so:

and:

We create a function *stream\_average* which takes an input stream *x* and produces the stream *y* of averages by first writing a function *average* with two parameters: an element of the input stream – in this case – a number *v*, and the state of the computation, where the state is the tuple (n+1, *cum*) where *cum* is the cumulative sum up to this point.



We now use this function with *stream\_func* to create a function, *stream\_average*, on streams. The initial state of the computation is passed as the parameter state in *stream\_func*. Initially, *n*=0 and *cum*=0.0, and so the initial state is (0, 0.0).



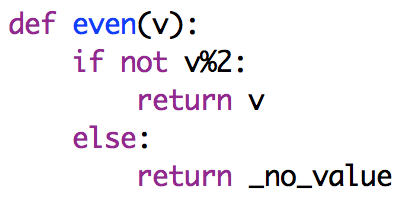
Given a stream *x* we can create a stream *avg* of the average values of the elements of *x* up to each point as follows:

Macintosh HD:Users:kmchandy:Desktop:Screen Shot 2015-05-21 at 4.58.13 PM.png

#### Example with \_no\_value

In some cases, we want to filter the input stream. For example, suppose we want to develop a function *g* that puts only even numbers on its input stream on to its output stream. The object *\_no\_value* is not placed in a stream; so we use the following function *f* on elements of the stream

Macintosh HD:Users:kmchandy:Desktop:Screen Shot 2015-05-22 at 12.38.42 PM.png



and the following function on streams:

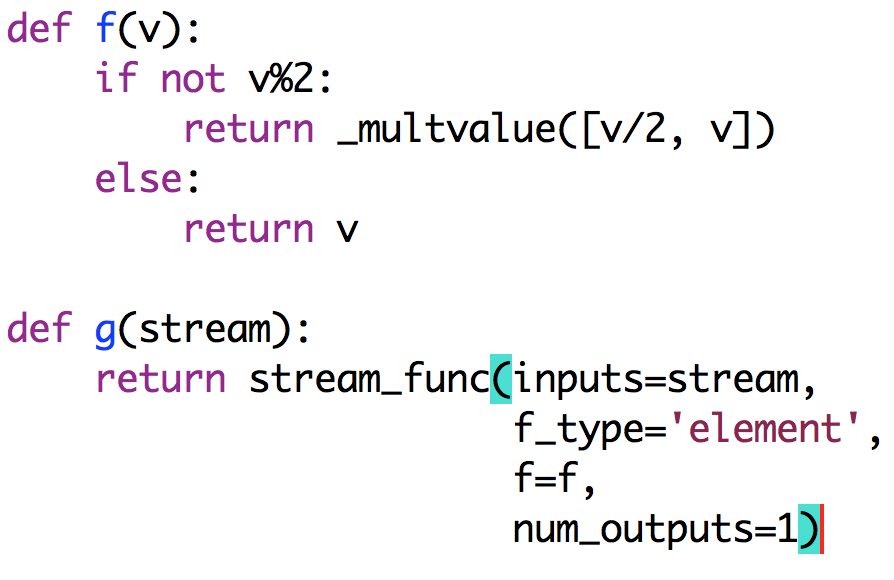
Macintosh HD:Users:kmchandy:Desktop:Screen Shot 2015-05-22 at 12.33.56 PM.png

We can now create a stream *a* that has the even numbers in stream *x* as follows:

Macintosh HD:Users:kmchandy:Desktop:Screen Shot 2015-05-22 at 12.35.26 PM.png

#### Example with \_multivalue

In some cases, we want a single element of the input stream to cause more than one value to be placed on the output stream. In this case, *f* returns an object of class *\_multiva*lue that is instantiated with a field that is a list. Each of the components in *\_multiva*lue are placed in the output stream as separate elements. For example, we want to create a function that has a single input stream of integers and that generates a single output stream of integers where for each integer *k* in the input stream the output stream has values *k*/2 and *k* if *k* is even and only *k* if *k* is odd.

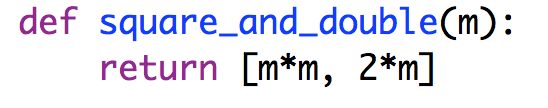


##### Single Input and Multiple Output Streams

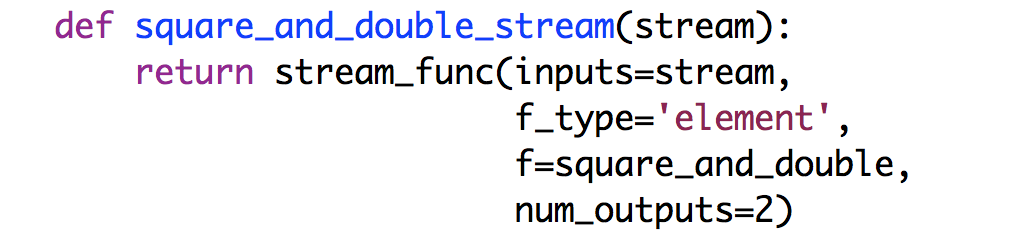
#### Example

Develop a function that generates two output streams from a single input stream where the first output stream is a square of the inputs and the second output stream doubles the inputs. If the output streams are and the input stream is then:

We first write a function, *square\_and\_double*, from a number to a list of two numbers, since this function has two outputs. Note that since the stream function has a single input stream, the parameter passed to *square\_and\_double* is a scalar rather than a list. And, since the stream function has two output streams, *square\_and\_double* returns a list of two numbers, rather than a scalar.



We now use this function on numbers to create a function on streams where the stream function has a single input stream and two output streams.



We can use this function on streams to create two streams, *squares* and *doubles*, where the elements of *squares* are squares of the corresponding elements of *x* and where the elements of *doubles* are twice the corresponding elements of *x*.



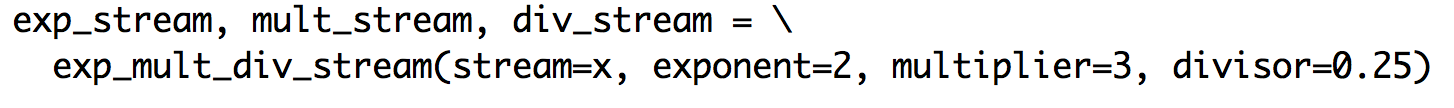
##### Example

Let’s look at another example. We want to develop a function, *exp\_mult\_div\_stream*(*stream, exponent, multiplier, divisor*) where the function has a single input stream ---- the first parameter, *stream* --- and where the function returns a list of three output streams. The elements of the three output streams are the values of the corresponding elements of the input stream (0) raised to *exponent*, (1) multiplied by *multiplier*, and (2) divided by *divisor*, respectively.

We first develop a function, *exp\_mult\_div\_number,* from a number to a list of 3 values where the elements of the list are: the number raised to *exponent*, the number multiplied by *multiplier*, and the number divided by *divisor*. Then, we use this function with *stream\_fu*nc to create a function on streams.



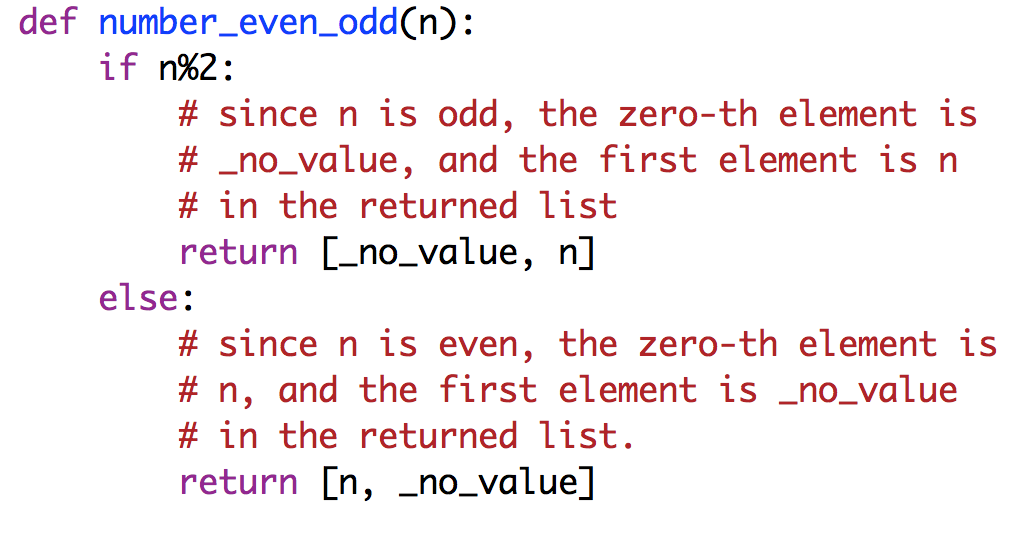
We can use this function to create streams, *exp\_stream*, *mult\_stream*, and *div\_stream* which correspond to exponentiation, multiplication and division of the elements in stream x as follows:



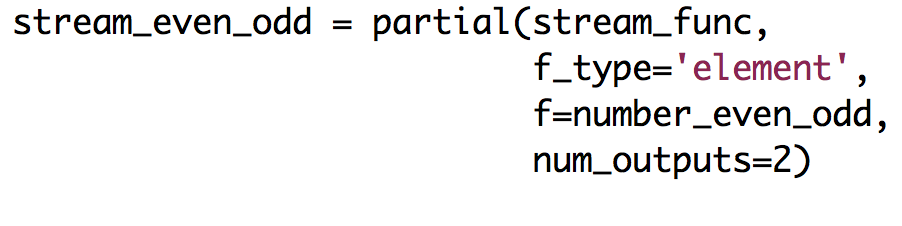
##### Example using the *\_*no\_value object

Write a function, *stream\_even\_odd,* which has a single input stream and two output streams: (0) an output stream containing the even numbers in the input stream and (1) another output stream containing the odd numbers. We first write a function, *number\_even\_odd* which has a single input parameter – a number, *n* – and returns a list of two values; it returns the list [*n*, *\_no\_value*] if n is even, and it returns the list *[\_no\_value*, *n*] if *n* is odd. Since *\_no\_value* does not appear in a stream, only even numbers appear in the first output stream and only odd number appear in the second output stream.

Macintosh HD:Users:kmchandy:Desktop:Screen Shot 2015-05-22 at 12.38.42 PM.png



We can now create the function, *stream\_even\_odd,* in one of the standard ways; for example:



You can now use the function *stream\_even\_odd* to create streams *evens* and *odds* containing the even and odd numbers of a stream *x*:



#### Example with State

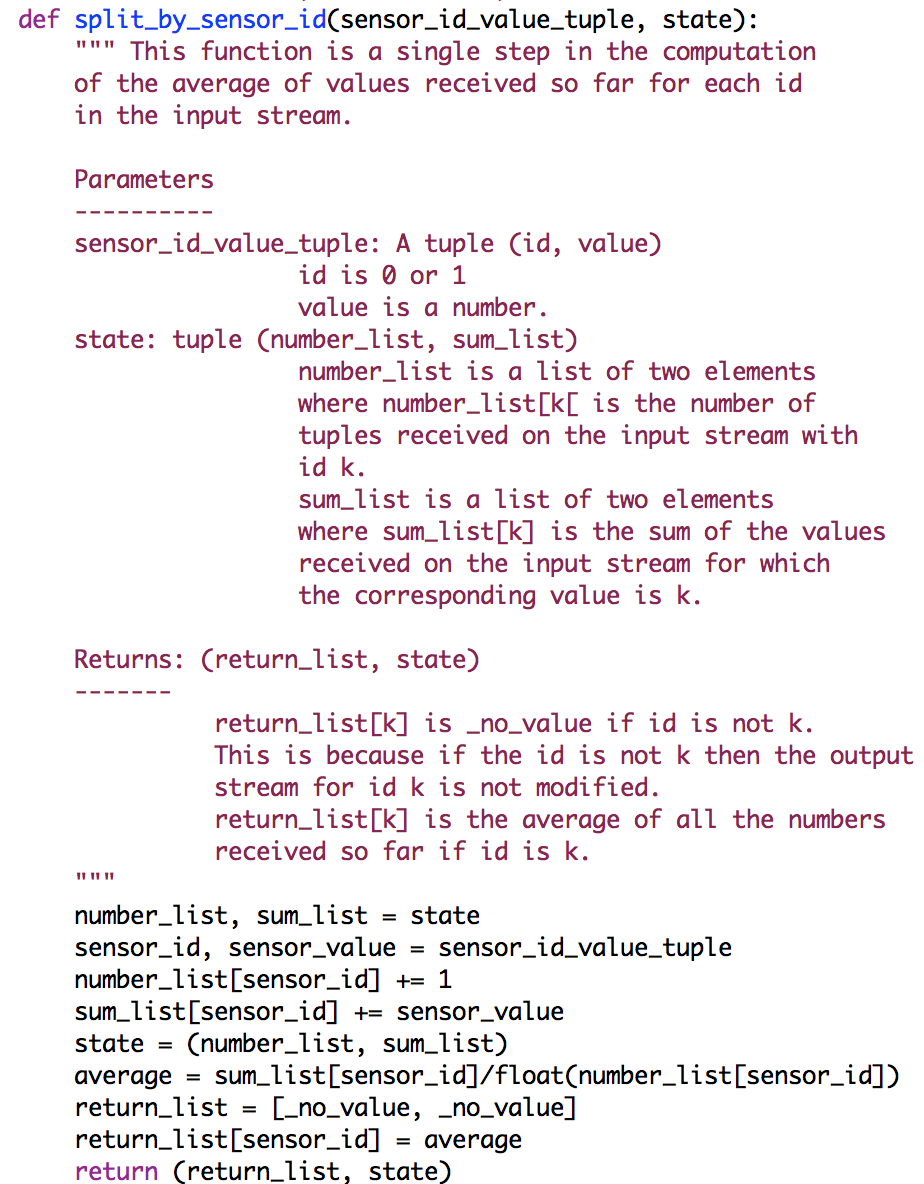
State is used in the same way for all examples; we give more examples here for completeness.

We are given a stream of tuples, (id, value), where id is the id of a sensor generating measurements and value is a measurement made by this sensor. We want to construct a function that takes this stream as input and outputs multiple streams, one for each sensor id. The elements of the output stream associated with an id k is a sequence of numbers where the n-th number in the sequence is the average of the first n values in the input stream for the sensor with id k.

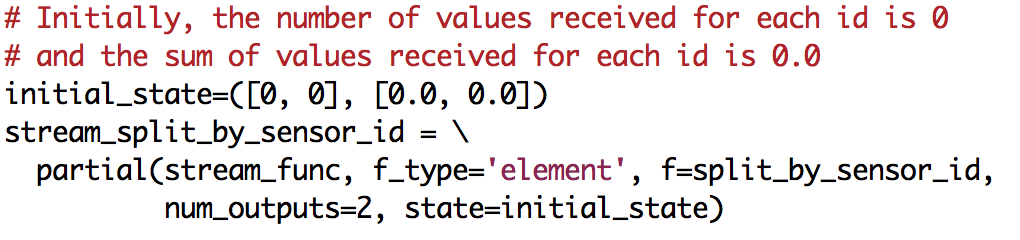
We assume, for simplicity that the ids have values 0 and 1; extending the program to arbitrary ids and arbitrary numbers of ids is straightforward.

As usual, we first write a function, *split\_by\_sensor\_id*, which has two parameters: an element of the input stream and the state of the computation. The function returns a list with two elements, one for each output list. The state of the computation is similar to the state of the average function given earlier for the single input single output case: the state has the number of values and sum of values received on the input stream for each id.

Then we use this function, with *stream\_func*, to create a function that takes an input stream and creates two output streams. Note that the state parameter passed to *stream\_func* is the initial state of the computation.



We now create a function with a single input stream and two output streams in the usual way:



Given a stream x, we can create two output streams – the averages for each id – as follows:

Macintosh HD:Users:kmchandy:Desktop:Screen Shot 2015-05-18 at 9.29.11 AM.png

##### Multiple Input and Single Output Streams

###### Example

We are given multiple input streams and we want to write a function that produces a single output stream where the n-th element of the output stream is the sum over all the input streams of their n-th elements.



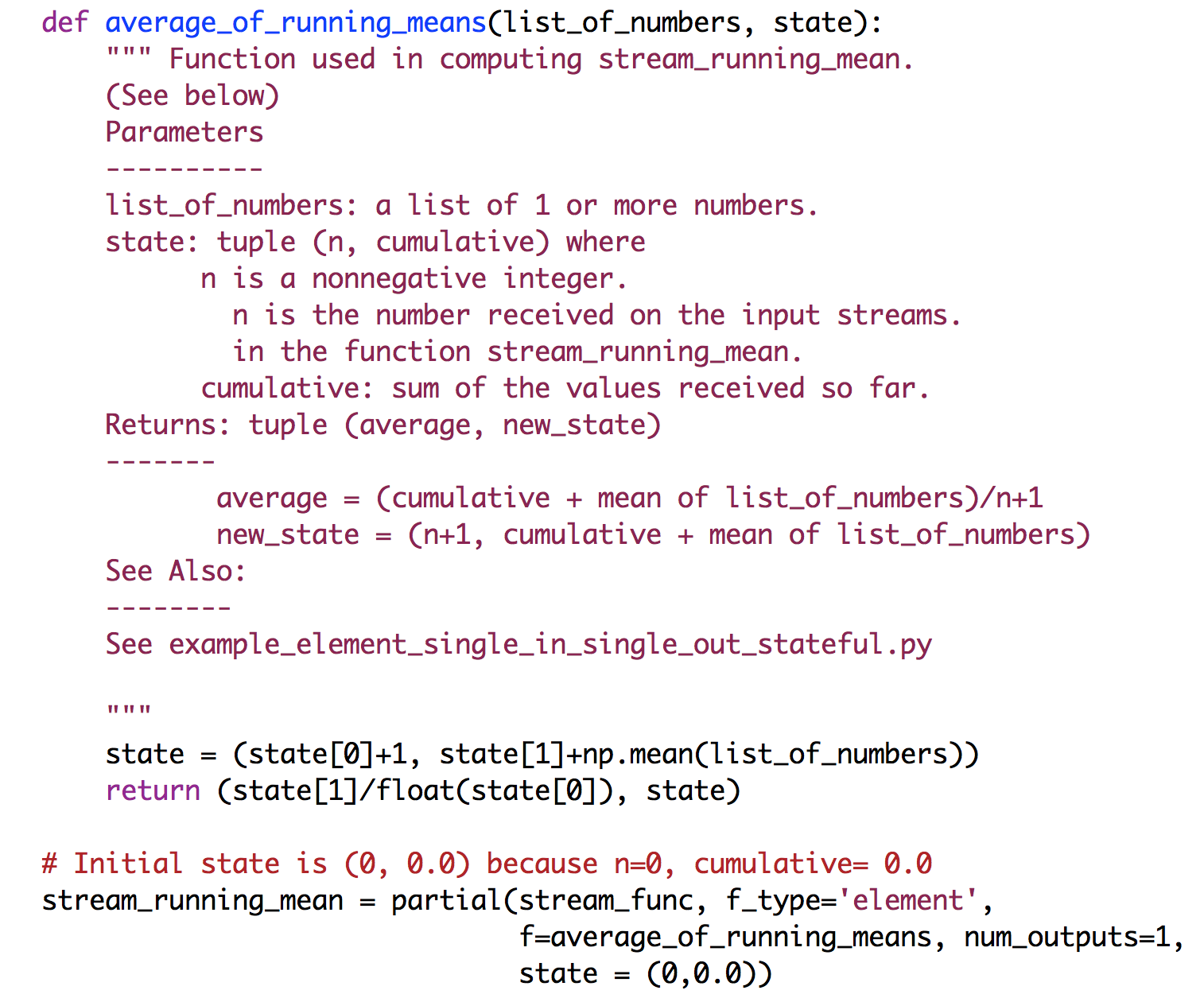
The next two examples are similar, where the n-th element of the output is the max and the mean of the n-th elements of the input streams. (Note that to use *np.mean* you have to import numpy as np.)



###### Example with State

Develop a function with input streams x, y, z and an output stream u where the elements of u are the running averages of the elements of x,yz:

where:



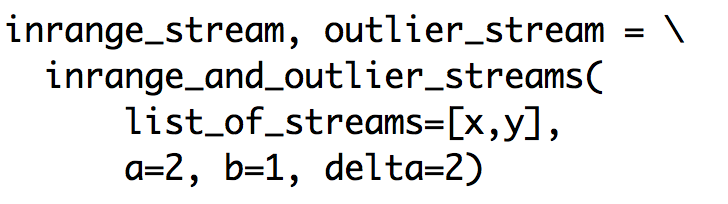
##### Multiple Input and Multiple Output Streams

#### Example

Develop a function that given two input streams *x* and *y*, and numeric parameters *a*, *b*, and *delta*, outputs two streams *inrange* and *outlier* where the pair (*x*[j],*y*[j]) appears in the *inrange* stream if: and in stream *outlier* otherwise.

#### Macintosh HD:Users:kmchandy:Desktop:Screen Shot 2015-05-18 at 5.01.41 PM.png

#### We can now use this function on streams to generate streams inrange\_stream and outlier\_stream from input streams x and y:



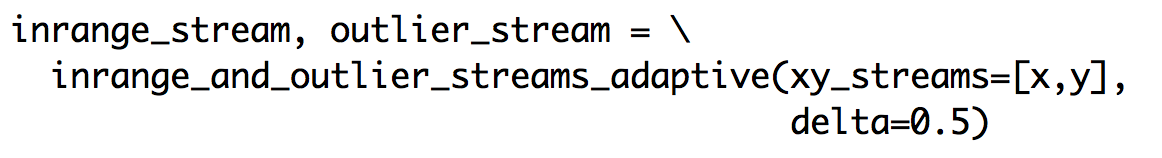
#### Example with State

This example is similar to the previous one: output in range and outlier streams. The difference is that in the previous example the parameters *a* and *b* were given and were fixed whereas in the next example the parameters are recomputed at each step using linear regression on all the values of *x* and *y* prior to that step.

The computation of linear regression for the n+1-the step uses *x\_sum* = , *y\_sum* = , *xx\_sum* =, *xy\_sum* = , and n. So, the state of the computation is: *state* = (*a, b, n, x\_sum, y\_sum, xx\_sum, xy\_sum).* In this example, the initial state corresponds to a straight line between the points (0.0, 0.0) and (1.0, 1.0). These two initial points are not shown here.

#### Macintosh HD:Users:kmchandy:Desktop:Screen Shot 2015-05-18 at 10.49.13 PM.png

#### Given streams x and y, we can use this function to generate streams inrange\_stream and outlier\_stream by continuously estimating the linear regression parameters on x and y, and using the latest estimates to determine values that are in range or outliers.



# WINDOWS

Next we look at cases where *f\_type* is ‘window’. A window is specified by the parameters *window\_size*, *step\_size* and a function *f*. The parameters *window\_size* and *step\_size* are positive integers. The function *f* operates on a single window if the stream function has a single input stream, and *f* operates on a list of windows, one per input stream, if the stream function has multiple input streams. Each window is a list or a NumPy array of size *window\_size*.

The stream returned by *stream\_func* when *f\_type* is ‘window’ is determined as follows. The moving window is [*n\*step\_size* : *n\*step\_size*+*window\_size*], where *n* is initially 0 and is incremented by 1 each time the window moves. Function *f* is applied to the moving windows of the input streams at each step, and the results of the functions are inserted into the output streams.

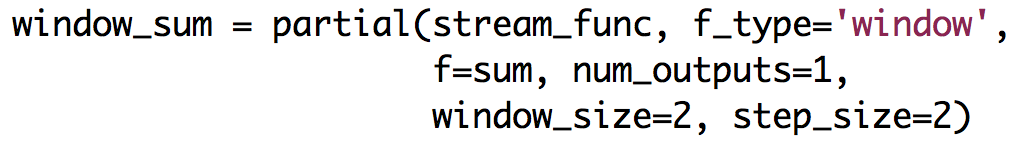
For example, if *step\_size*=2, *window\_size*=2, and *f* returns a single value for a single output stream, then the elements of the output stream will be [f([0:2]), f([2:4]), f([4:6]),.. and if *step\_size*=1, *window\_size*=2, the output stream will be [f([0:2]), f([1:3]), f([2:4]),..

##### Single Input and Single Output Streams

#### Example

Develop a function that has a single input stream and a single output stream where the elements of the output stream are the sums of the elements of each window in the input stream. Set the window and step sizes to 2.

Since the stream function has a single input stream and since the computation is stateless, function *f* has a single parameter that is either a list or a NumPy array of length *window\_size*. Since the stream function has a single output stream, *f* returns a scalar value. We use the Python function *sum* for *f* in this example.

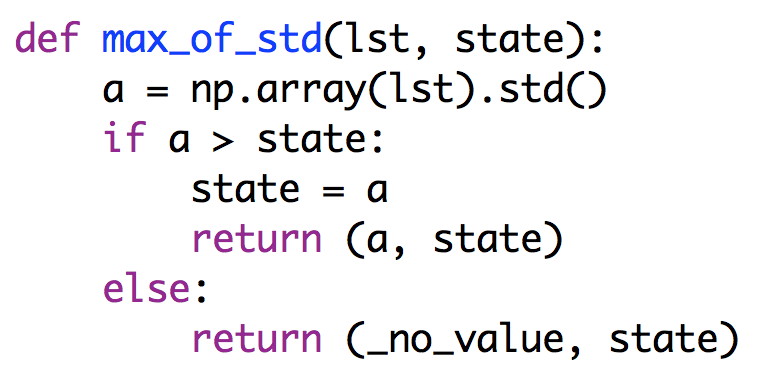


#### Example with State and \_no\_value

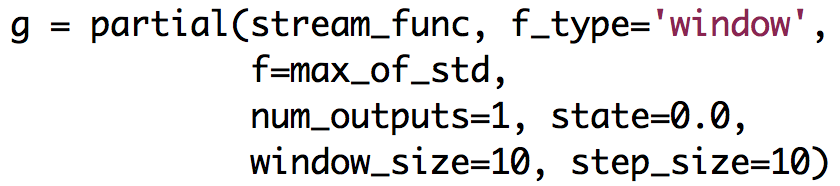
Develop a function that has a single input stream and a single output stream where the elements of the output stream are standard deviations of each window in the input stream, but only increasing standard deviations are output. The standard deviation of a window that is less than that of an earlier window is not output. Set the window and step sizes to 10.

Since the stream function has a single input stream and the computation has state, function *f* has two parameters: (i) a list or NumPy array of size *window\_size* and (2) a state. Since the stream function has a single output stream and the computation has state, f returns a tuple of two values: (i) a scalar to be placed on the output stream and (ii) the next state.

In this example, the computation keeps track of the maximum standard deviation seen so far. If the standard deviation of the current window does not exceed the state then no output should appear on the output stream and so, in this case, the function returns (\_*no\_value*, *state*).



The initial state is 0.0 and is specified as a parameter of *stream\_func* in our definition of stream function *g*.



Given a stream *x* we can now create stream *y* that outputs the maximum standard deviations as follows:

Macintosh HD:Users:kmchandy:Desktop:Screen Shot 2015-05-26 at 10.14.03 AM.png

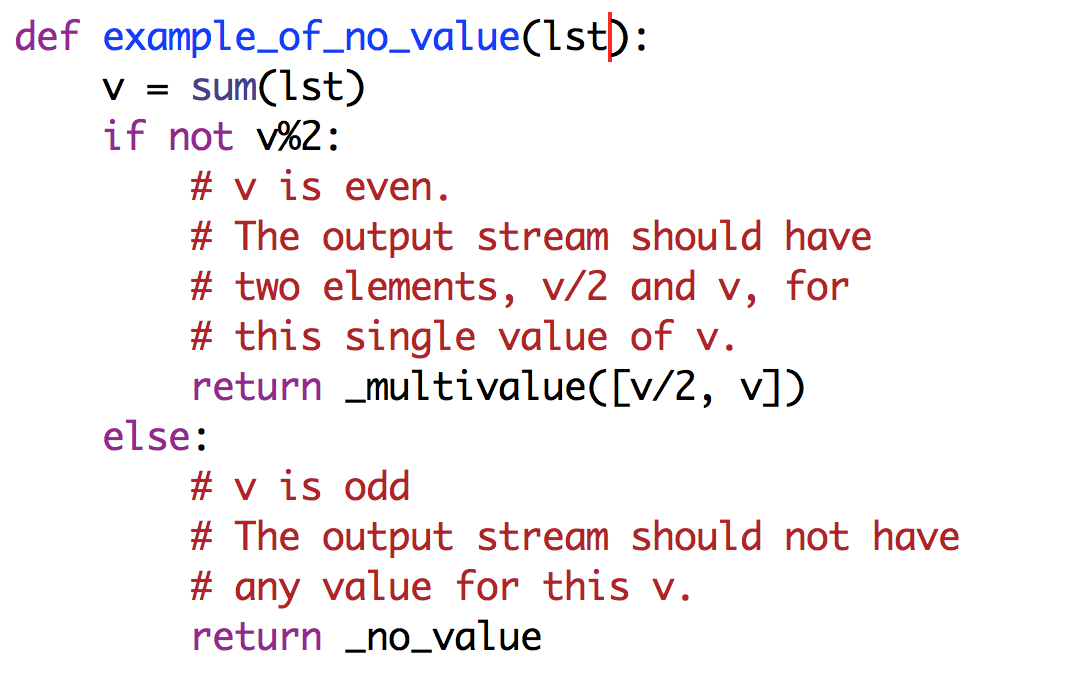
and, if we wish we can apply stream function *g* to stream *y* to create a new stream *z*:

Macintosh HD:Users:kmchandy:Desktop:Screen Shot 2015-05-26 at 10.15.27 AM.png

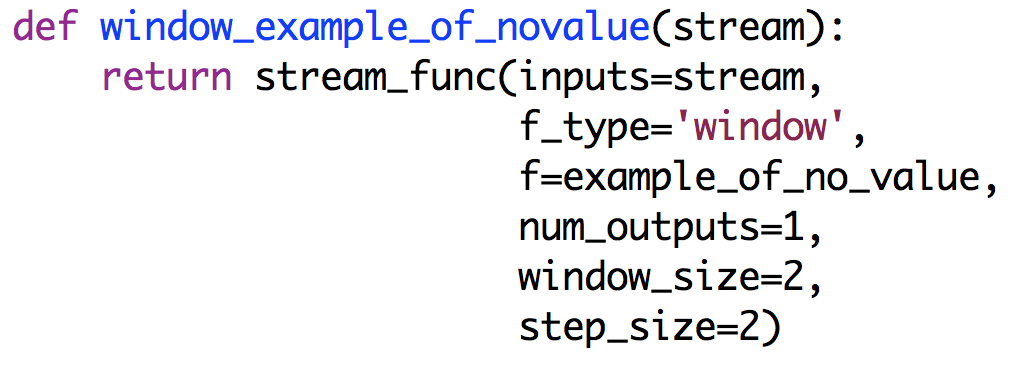
#### Example with \_no\_value and \_multivalue

Develop a window function where if the sum of the window is even then the output stream contains half of the sum and the sum. If the sum of the window is odd then the output stream contains no value.

We first develop a function that has an input parameter which is a list and that returns an element. The return value is a \_multivalue object if the sum of the list is even and is *\_no\_value* otherwise.



Next, we develop a window function with a single input stream, a single output stream, a window size of 2 and a step size of 2.



Given a stream *x*, we can use this stream function to create an output stream *a*:

Macintosh HD:Users:kmchandy:Desktop:Screen Shot 2015-05-24 at 5.44.01 PM.png

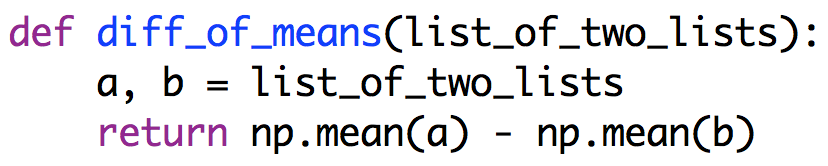
##### Multiple Input and Single Output Streams

We develop a window function *g* with multiple input streams and a single output stream by first developing a function *f* with a parameter that is a list of windows (i.e., a list of lists, each of size *window\_size*) and that returns an object that will be placed in the output stream. If the computation is stateful then *f* also has the state as a parameter.

#### Example

#### Develop a function with two input streams and a single output stream where the elements of the output stream are differences in the means of the windows in the input streams where the window and step sizes are both 2.

We first develop the function *diff\_of\_means* with a single input parameter which is a list of two lists and that returns a scalar.



Next, we develop the desired stream function using *diff\_of\_means* in *stream\_func*.



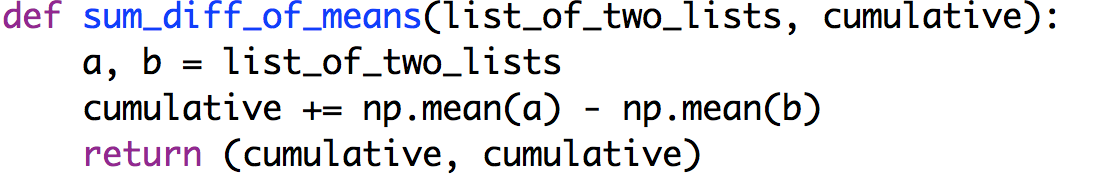
Given streams *x* and *y*, we can generate a stream *z* which has the differences in the means of windows in x and y as follows:

Macintosh HD:Users:kmchandy:Desktop:Screen Shot 2015-05-26 at 11.48.37 AM.png

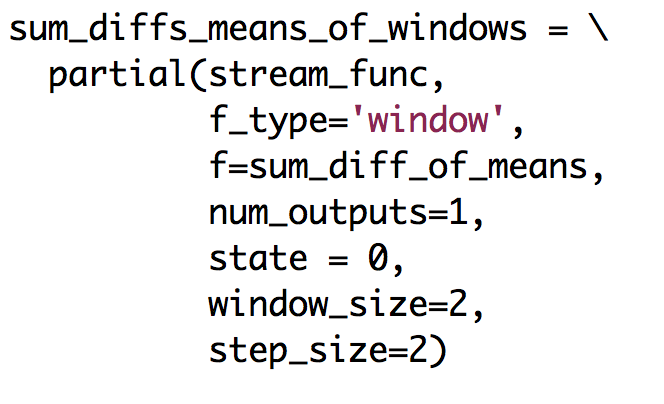
#### Example with State

Develop a function with two input streams and a single output stream where the output stream is the sum of the differences of the means of windows into the two input streams.

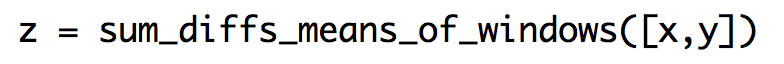
This computation uses a state, *cumulative*, which is the sum of the differences in the means of the windows seen so far in the input streams. Since the stream function has multiple input streams the parameter of *f* are the list and the state, and since the stream function has a single output stream, function *f* returns the object to be placed on the output stream and the new state.



We now develop the stream function. The initial state (*cumulative* = 0.0) is passed as a parameter to *stream\_func*.



Given streams *x* and *y* we can now use this stream function to generate a stream *z* whose values are the sums of the differences of the means.



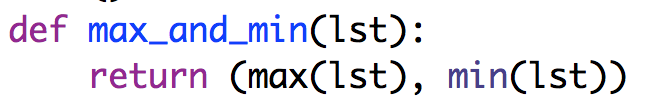
##### Single Input and Multiple Output Streams

Next we develop a window function *g* with a single input stream and multiple output streams. We first develop a function *f* on a single window (i.e., a list) of the input stream for stateless computations, and on a single window of the input stream and the state for stateful computations. Since *g* has multiple output streams, *f* returns a list with one element for each output stream, and also returns the next state for stateful computations.

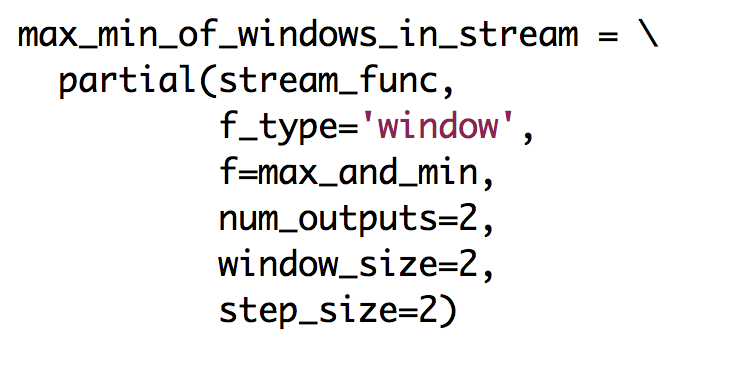
#### Example

Develop a function that has a single input stream and two output streams. One output stream has the maximum values of windows into the input stream and the other output stream has the minimum values. The window size is 2 and the step size is 2.

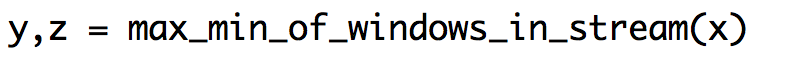
Since this computation is stateless, we first develop a function with a single parameter – a window (i.e., list) – and that returns a list with 2 values, one for each output stream.



Next, we develop a function on streams using *max\_and\_mi*n:



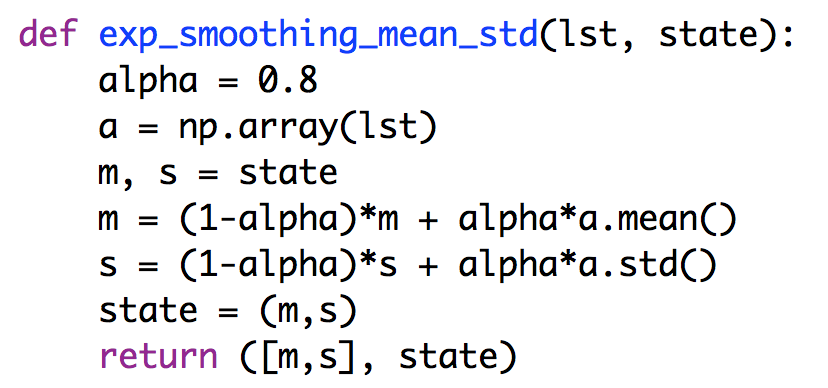
Given a stream *x*, we can now use this function to create streams *y* and *z* with the maximum and minimum values for each window in *x*.



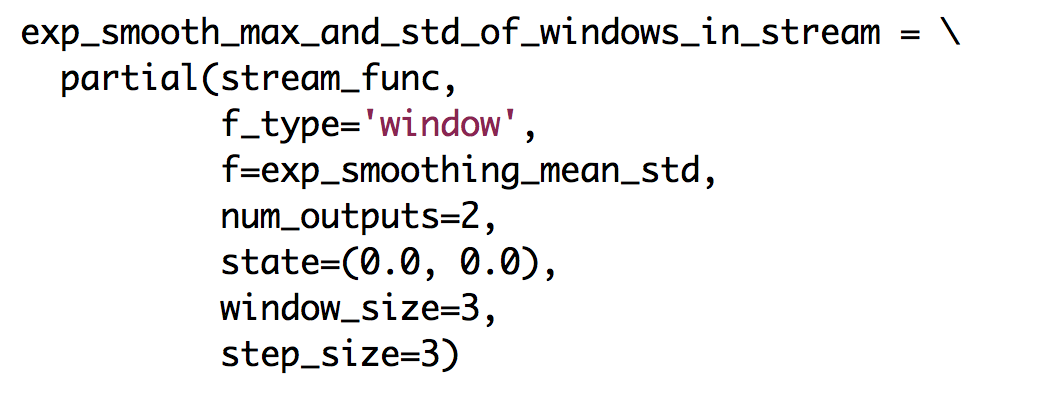
#### Example with State

Develop a function with a single input stream and two output streams where one output stream has the exponentially smoothed mean and the other the standard deviation of windows in the input streams.

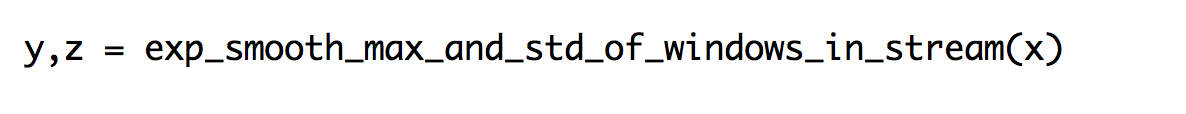
The state of the computation is the mean and standard deviation at the last point; these values are used in the exponentially smoothing operation. We first write a function that has two parameters: a window (i.e., a list) and the state. The function returns the object to be placed on the single output stream and the next state.



Next we write a function on streams. The initial state is set, arbitrarily, to 0.0 and 0.0, as parameters for *state* in *stream\_func*.



Given a stream *x*, we can use the stream function to generate streams *y* and *z* of the exponentially smooth values of the mean and standard deviation of the windows in *x*.



##### Multiple Input and Multiple Output Streams

Next we develop a window function *g* with multiple input streams and multiple output streams. We first develop a function *f* on a single window (i.e., a list) of the input stream for stateless computations, and on a single window of the input stream and the state for stateful computations. Since *g* has multiple output streams, *f* returns a list with one element for each output stream, and also returns the next state for stateful computations.

#### Example

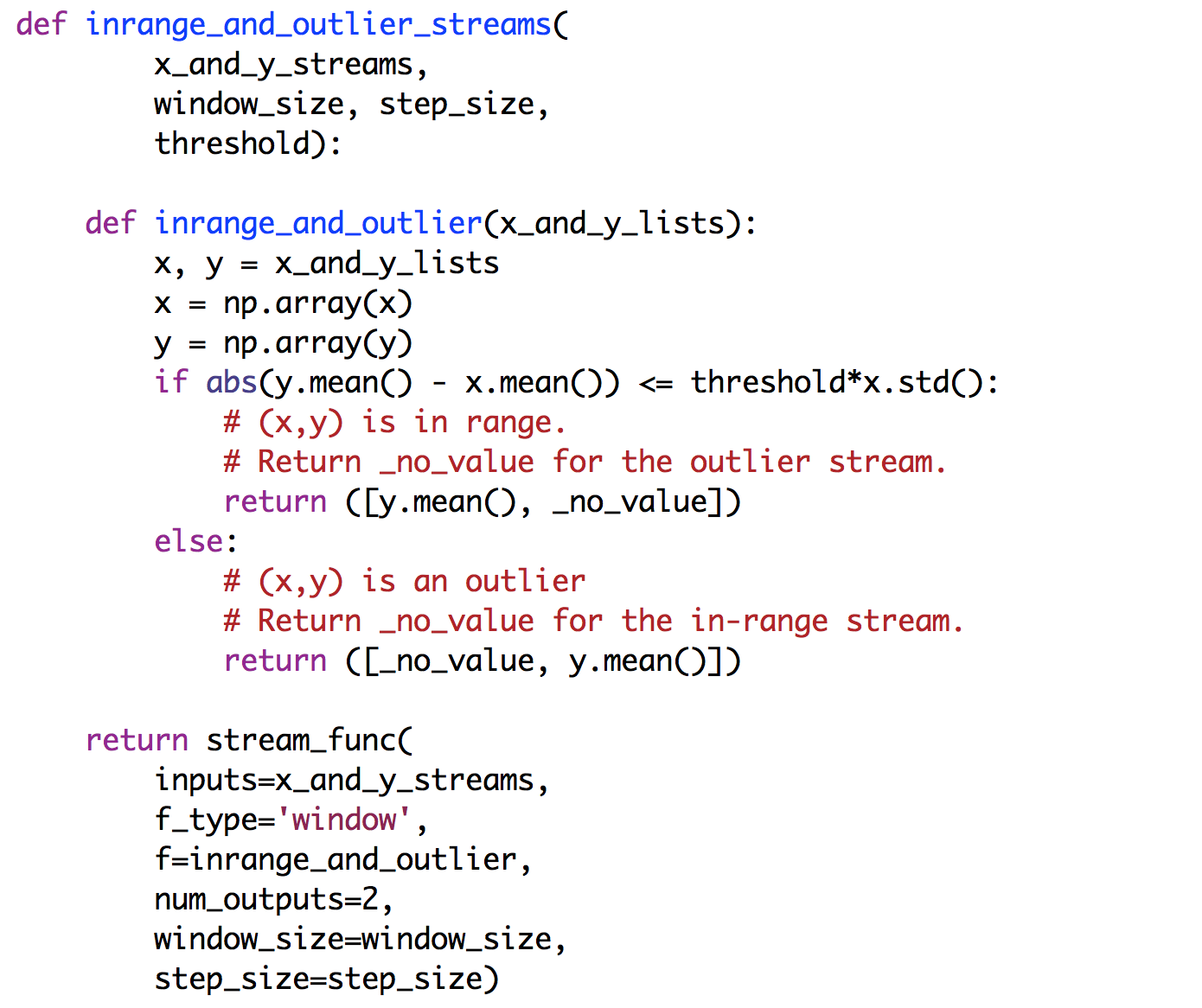
Develop a function, inrange\_and\_outlier\_streams with the following parameters:

* *x\_and\_y\_streams*: a list of two streams (usually) called x and y.
* *window\_size*, *step\_size*: the window and step sizes (which must be positive integers) for a moving window.
* *threshold*: a number

The function computes the mean and standard deviation of windows in stream *x* and it computes the mean of windows in stream *y.* The window size and step size are parameters of the function. The function generates two output streams that we call the *inrange* stream and the *outlier* stream. If the mean of the window in y differs from the mean of the window in *x* by more than *threshold* times the standard deviation of the *x* window, then the *x* and *y* windows are “outliers” and the means of the *x* and *y* windows appear in the *outlier* stream; otherwise they appear in the *inrange* stream.

We first write a function *inrange\_and\_outlier* that operates on two windows (i.e., lists of the specified window size) and that returns a list of two values, one for each output stream. If the object \_*no\_value* is returned by the function then nothing is placed on the corresponding output stream.

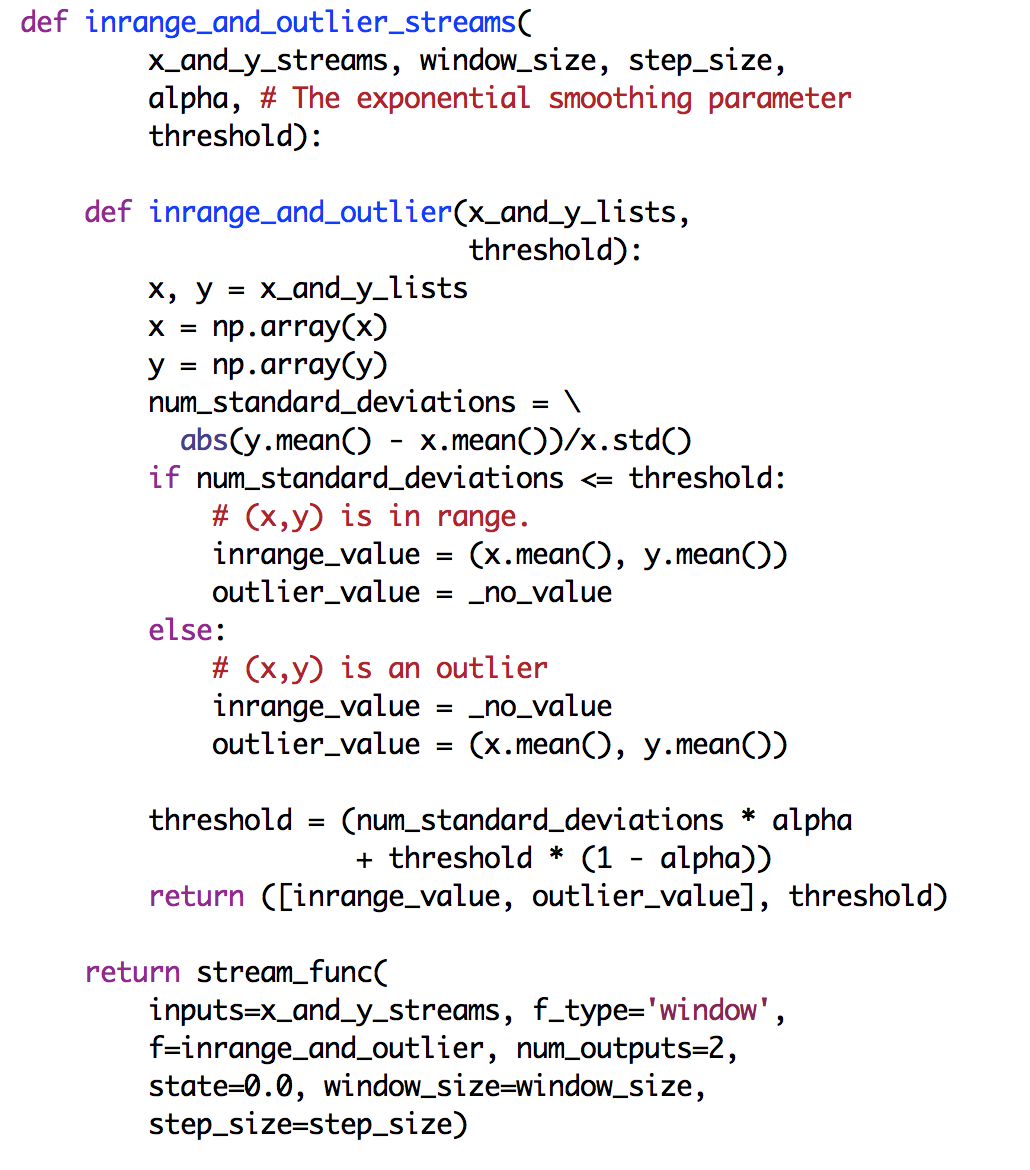
We use *inrange\_and\_outlier* with stream\_func to obtain the desired stream function.



#### Example with State

#### Develop a function identical to the one given in the previous example, except that the threshold is adaptive. The threshold is exponentially smoothed at each step using a parameter alpha.

The *state* is the threshold, and an initial state of 0.0 is passed as a parameter of *stream\_func*.



#### Given input streams x and y, we can use the stream function to generate streams inrange\_stream and outlier\_stream with specified parameters:



# LISTS

Next we look at cases where *f\_type* is ‘list’. Functions on streams are developed in exactly the same way as in the case where *f\_type* is ‘element’ except that operations on elements are replaced by operations on lists. For example, to develop a function g with a single input stream and a single output stream, when *f\_type* is ‘list’, we first develop a function f with a parameter that is a list and that returns a list. The input parameter of f is a list of elements of the input stream, and the list returned by f contains elements of the output stream.

When *f\_type* is ‘list’, the object \_*no*\_*value* and objects of class \_*multivalue* are not used. This is because a function can return a list with no value by returning an empty list, and a function can return a list with arbitrary many values.

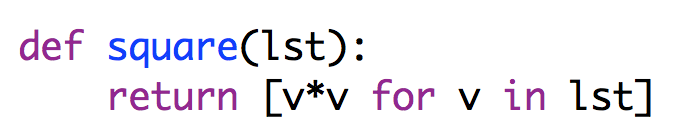
An operation on a large list can be more efficient than repeatedly operating on each element of the list. In such cases setting *f\_type* to ‘list’ is preferable.

We don’t give detailed explanations for the examples that are presented next because the explanations are so similar to those given for the case where *f\_type* is ‘element’.

##### Single Input and Single Output Streams

#### Example

#### Develop a function with a single input stream and a single output stream where the elements of the output stream are squares of elements of the input stream. We first develop a function that squares elements of a list.



Next, we use the function as a parameter of *stream\_func* to obtain the desired function on streams.

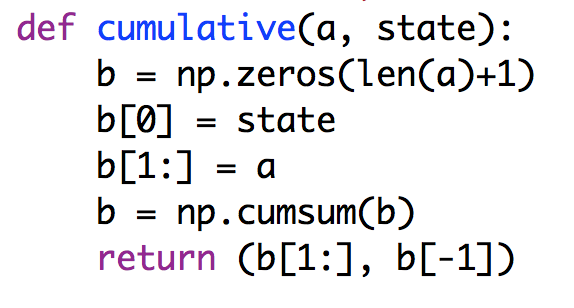
Macintosh HD:Users:kmchandy:Desktop:Screen Shot 2015-05-26 at 5.49.49 PM.png

Given a stream *x*, we can use this function to generate a stream *u* of squares of *x*:

Macintosh HD:Users:kmchandy:Desktop:Screen Shot 2015-05-26 at 5.51.01 PM.png

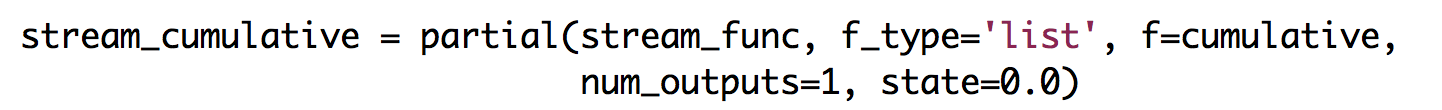
#### Example with State

#### Develop a function with a single input stream and a single output stream where the elements of the output stream are sums of the elements of the input stream. The state of the computation is the sum of the values received so far in the input stream. We first develop a function, cumulative, that returns a list consisting of the cumulative sum of the values of its input list, starting at an initial value that is the state of the computation.



For example if *a* is the list [5, 7] and *state* is 10 then the function *cumulative* returns the list [15, 22].

Next, we use function *cumulative* to develop a function on streams. The initial state, 0.0, is passed as a parameter of *stream\_func*.



Given a stream *x* we can create a stream *v* that has the cumulative values of *x*:

Macintosh HD:Users:kmchandy:Desktop:Screen Shot 2015-05-26 at 6.06.33 PM.png

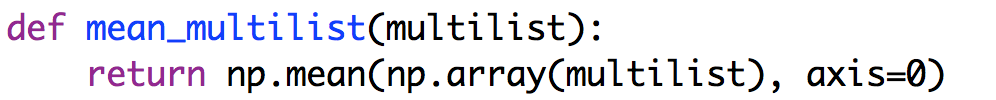
##### Multiple Input and Single Output Streams

We develop a function *g* with a list of multiple input streams and a single output stream by first developing a function *f* with an input parameter that is a list of lists and that returns a single list; then we use *f* with *stream\_func* to obtain *g*.

#### Example

Develop a function with an arbitrary number (at least 1) of input streams and a single output stream. The k-th element of the output stream is the average of the k-th elements of all the input streams. If the input streams are and the output stream is then:

We first develop a function with an arbitrary number (at least 1) of input lists and a single output list. The k-th element of the output list is the average of the k-th elements of all the input lists, and so if the input lists are and the output list is then the above equation holds.



Next, we use this function as a parameter of *stream\_func* to obtain the desired function, *stream\_mean*, on streams:

Macintosh HD:Users:kmchandy:Desktop:Screen Shot 2015-05-26 at 6.17.10 PM.png

Given streams *x, y, z*, we can create a stream *w* whose elements are the means of the corresponding elements of *x, y, z*:

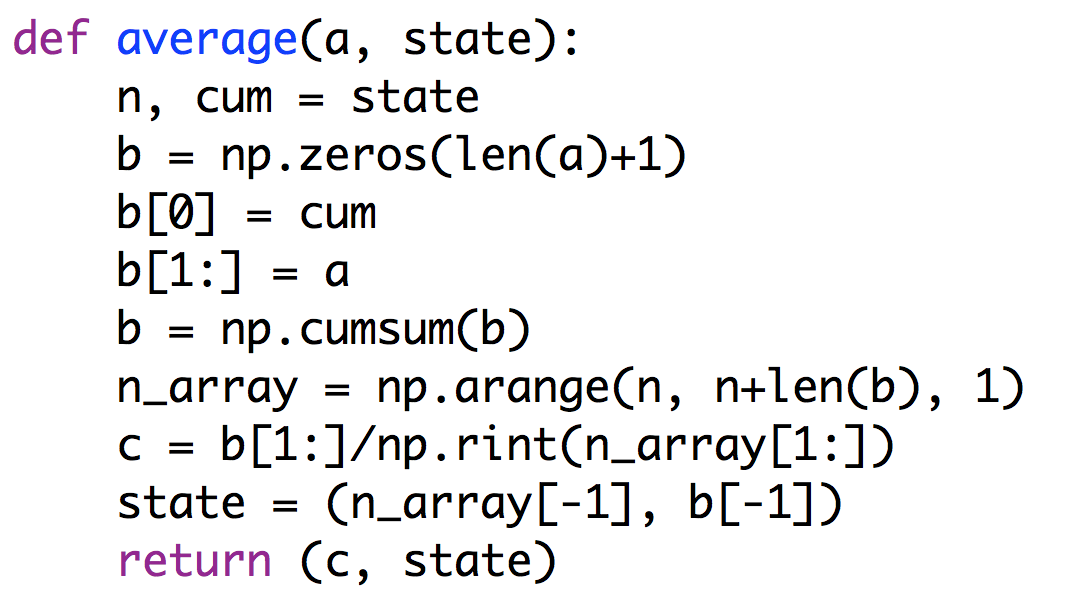
Macintosh HD:Users:kmchandy:Desktop:Screen Shot 2015-05-26 at 6.19.44 PM.png

#### Example with State

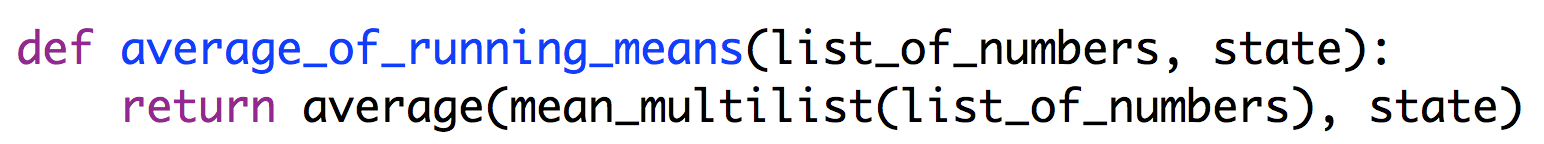
Develop a function with an arbitrary number (at least 1) of input streams and a single output stream. The k-th element of the output stream is the running average of the k-th elements of all the input streams, i.e., if the input streams are and the output stream is then the nth value on the output stream is:

where is defined as:

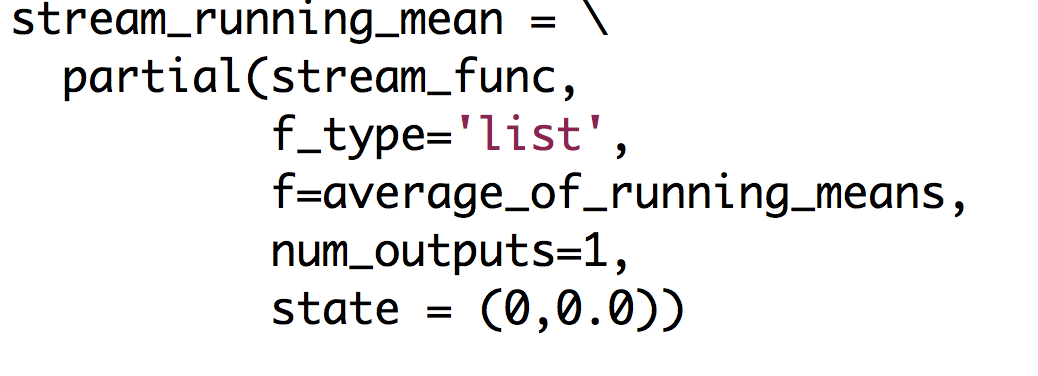
The state of the computation is a tuple consisting of n and where n values have been placed on the output stream so far and *cum* stands for cumulative. We develop a function *average* that computes given and given the state.



and we compute y using the mean\_multilist function given earlier to obtain the function that computes the running average of the means:



Next we develop the function on streams:



Given streams x, y, z we can use this function on streams to create a stream u whose elements are the running means of x, y, z:

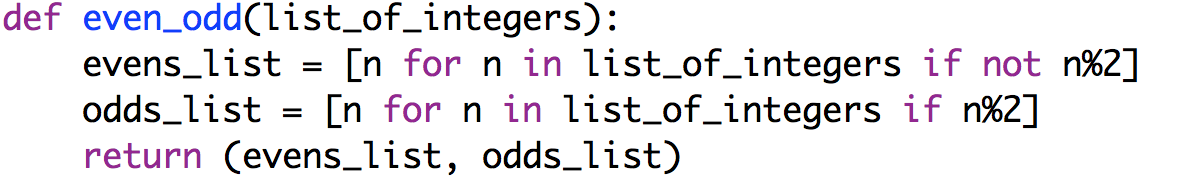
Macintosh HD:Users:kmchandy:Desktop:Screen Shot 2015-05-27 at 4.33.09 PM.png

##### Single Input and Multiple Output Streams

We develop a function *g* with a single input stream and a list of multiple output streams by first developing a function *f* on an element of the input stream and that returns a list of elements, one for each output stream. If the computation is stateful, *f* also has state as an input parameter and it returns the new state. Then, we use *f* with stream\_func to obtain *g*.

#### Example

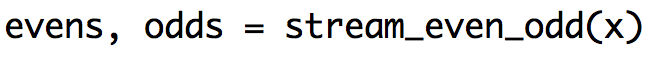
Develop a function that has a single input stream of integers and two output streams where one output stream has the even numbers in the input stream and the other output stream has the odd numbers. We first develop a function with a single input parameter – a list of integers – and which returns a list of two lists: (i) a list of even numbers in the input list and (ii) a list of odd numbers in the input list.



We use this function with *stream\_func* to obtain the desired function on streams:

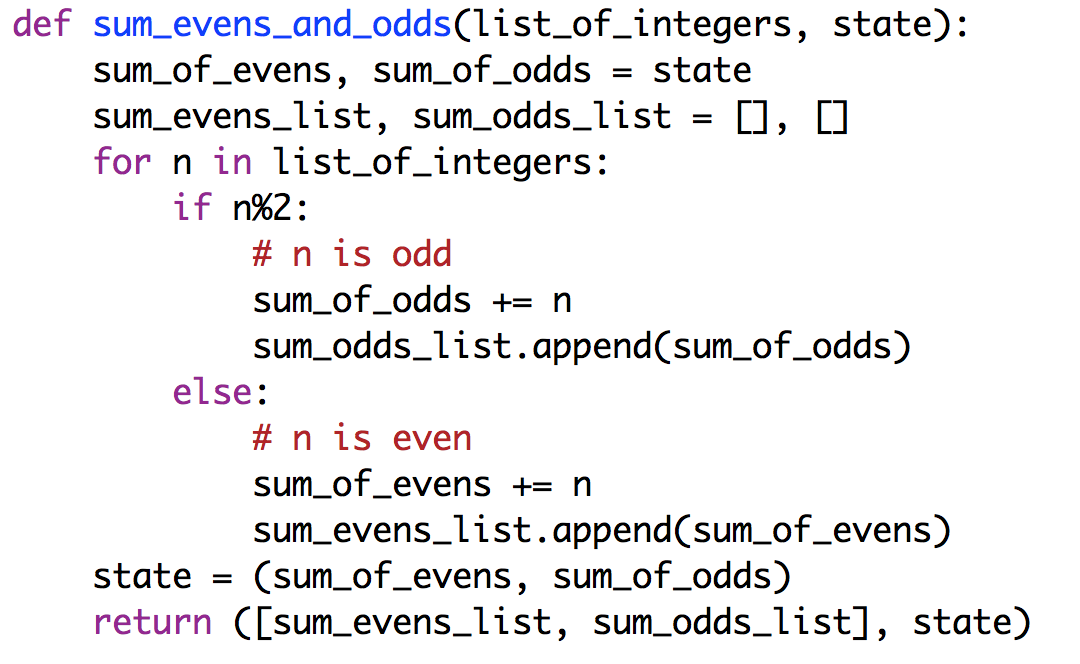


Given a stream *x*, we can use this function on streams to create streams *evens* and *odds*, containing the even and odd values in x, as follows:

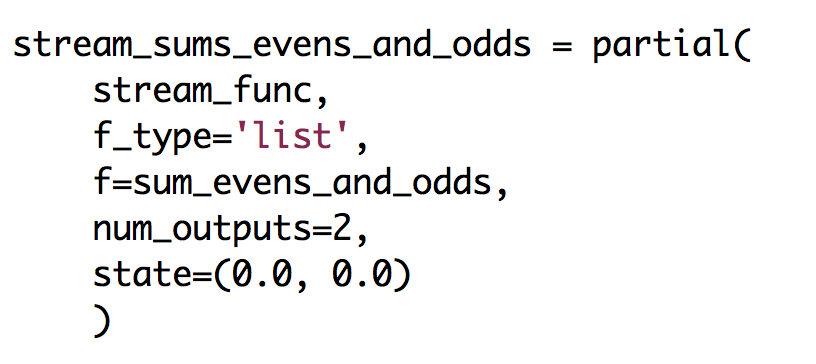


#### Example with State

Develop a function that has a single input stream of integers and two output streams where one output stream has the sum of the even numbers in the input stream and the other output stream has the sum of the odd numbers. We first develop a function with two parameters – a list of integers and the state of the computation – and which returns the next state and a list of two lists: (i) a list of even numbers in the input list and (ii) a list of odd numbers in the input list. The state is a tuple: the sum of the even numbers and the sum of the odd numbers received.



We use this function with *stream\_func* to obtain the desired function on streams:



Given a stream *u*, we can use this function on streams to create streams *sums\_of\_evens* and *sums\_of\_odds*, containing the even and odd values in *u*, as follows:

Macintosh HD:Users:kmchandy:Desktop:Screen Shot 2015-05-27 at 5.56.57 PM.png

##### Multiple Input and Multiple Output Streams

We develop a function *g* with a list of multiple input streams and a list of multiple output streams by first developing a function *f* on a list of elements of each of the input streams and that returns a list of elements, one for each output stream. If the computation is stateful, *f* also has state as an input parameter and it returns the new state. Then, we use *f* with *stream\_func* to obtain *g*.

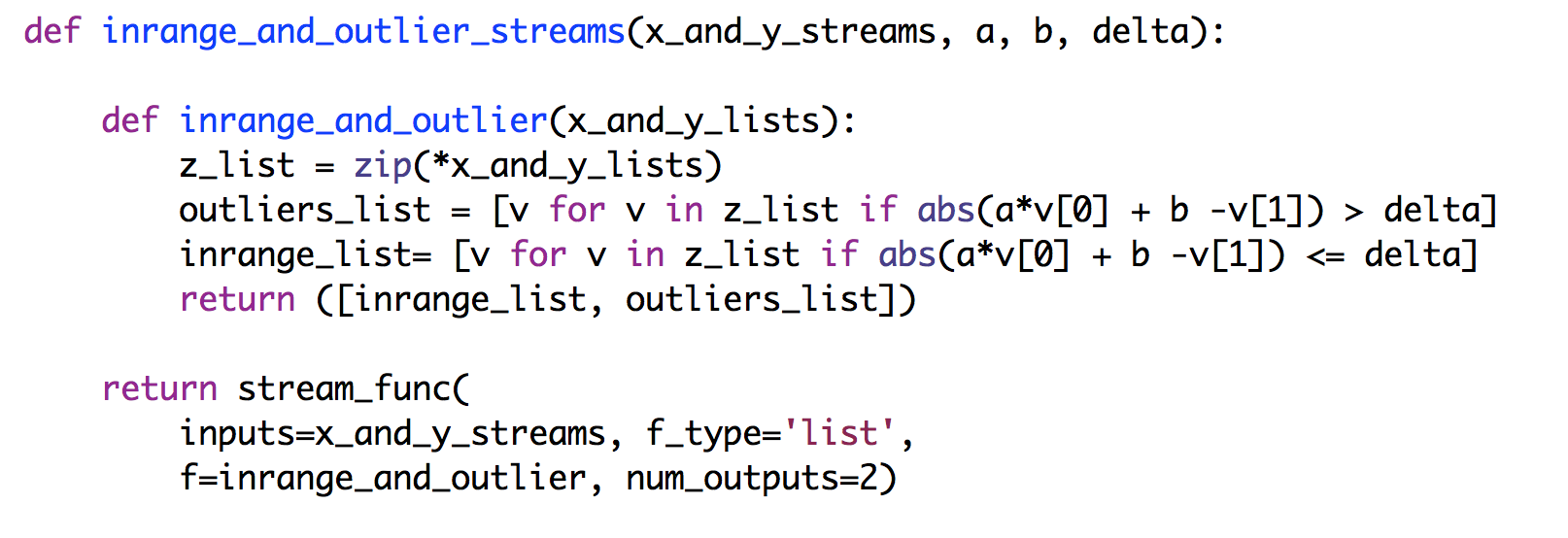
#### Example

Develop a function, inrange\_and\_outlier\_streams, with the following parameters:

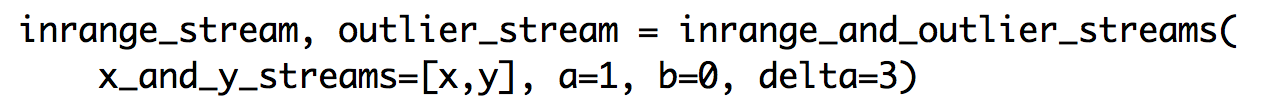
1. x\_and\_y\_streams, a list of two streams, stream x and stream y.
2. a, b, delta where (x[k], y[k]) is defined to be an outlier if:

and is defined to be in range otherwise.

The function has two output streams, and *x\_and\_y\_streams, a, b, delta* stream containing the in-range and outlier values.



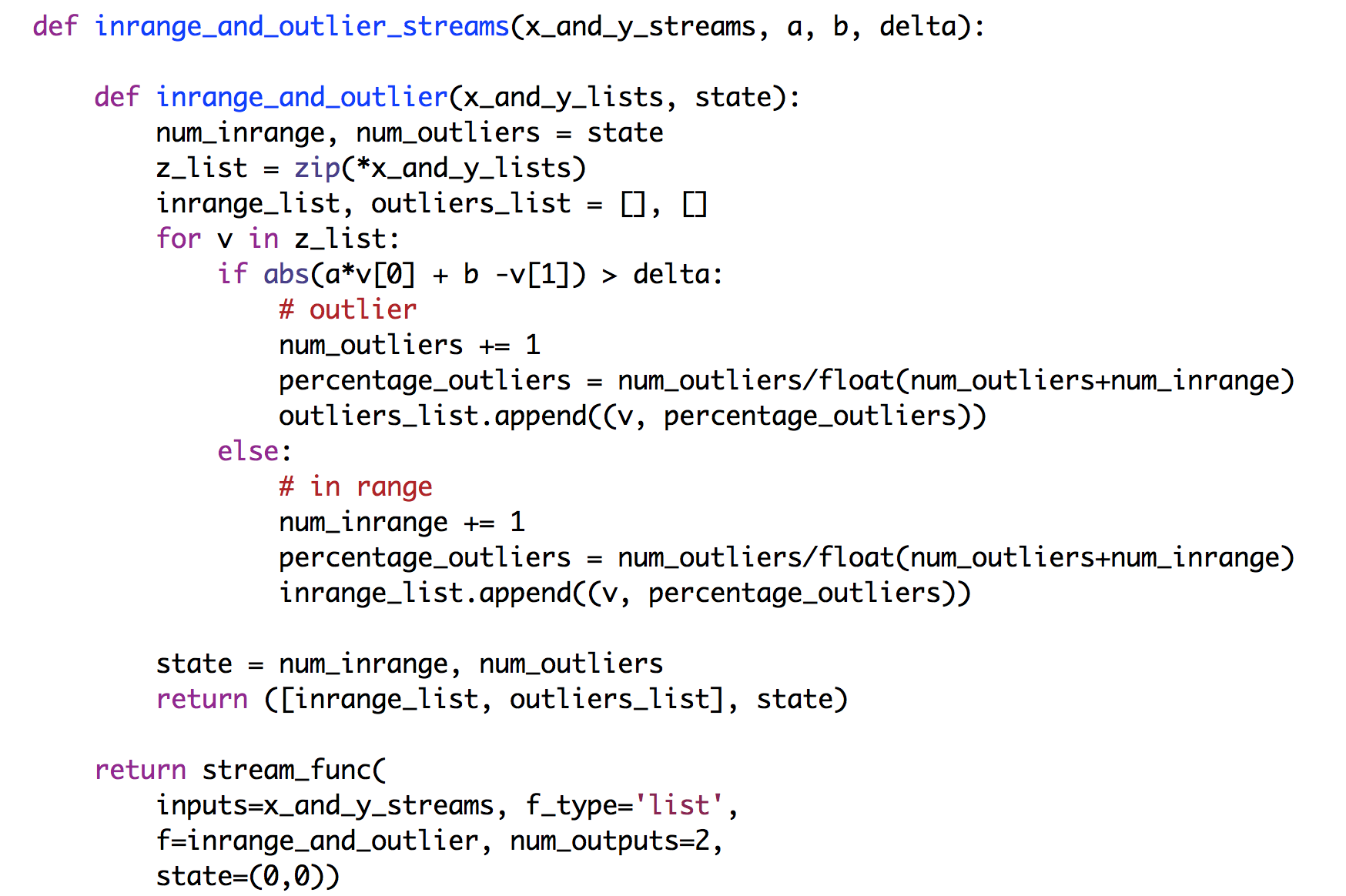
Given streams x and y, we can create streams *inrange\_stream* and *outlier\_stream*:



#### Example with State

Consider the same problem as in the previous example except that the output streams also have the percentage of outliers.

The state of the computation is the number of in-range and the number of outliers seen so far.



# TIMED WINDOWS

Next we look at the case where *f\_type* is ‘timed’; in this case, functions operate on timed windows. A timed window is a list where the elements of the list are named tuples of type *TimeAndValue*. A *TimeAndValue* tuple has two fields, a *time* and a *value*. The time field is a nonnegative number – either an integer or a real number. The value field is arbitrary. Later elements in the stream of *TimeAndValue* tuples should have higher values of the time field; however, out-of-order elements can be handled, and we discuss this situation later.

A timed window is specified by the parameters *window\_size*, *step\_size* and a function *f*. The parameters *window\_size* and *step\_size* are positive numbers – either integers or real numbers. If the stream function has a single input stream then the function *f* operates on a single window, and if the stream function has multiple input streams then *f* operates on a list of windows where the list has one window for each input stream.

The stream returned by *stream\_func* when *f\_type* is ‘timed’ is computed as follows. A timed moving window is specified by the start and end times of the window. The n-th timed window is [*n\*step\_size* : *n\*step\_size*+*window\_size*], where *n* is initially 0 and is incremented by 1 each time the window moves; here *n*\**step\_size* is the start time of the window and *n\*step\_size*+*window\_size* is the end time of that window.

A window specified by start time s and end\_time e is a list of *TimeAndValue* tuples where for each tuple in the list, its time field t is at or after the start time and before the end time:

Successive elements in the list should have increasing order of time values.

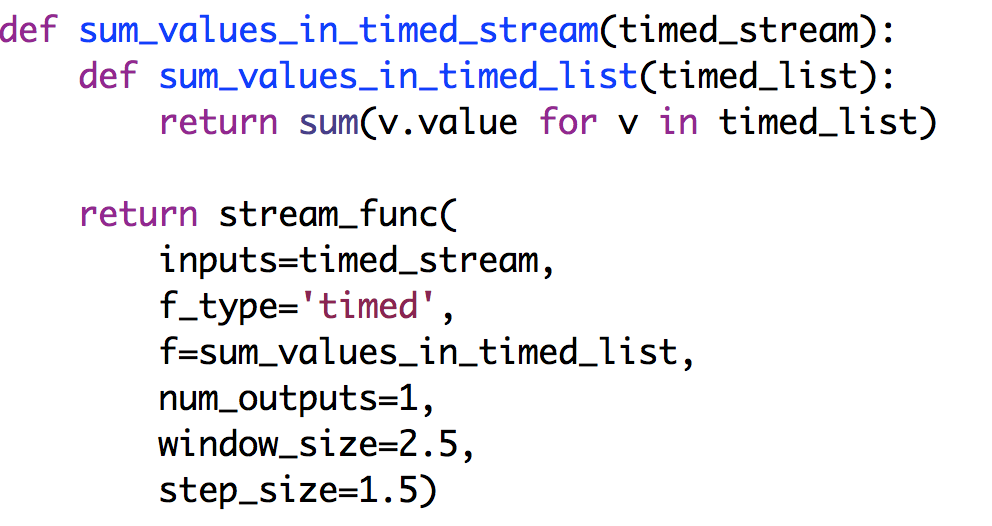
Function *f* is applied to the moving windows of the input streams at each step, and the results of the functions are inserted into the output streams. The number of elements in a timed window may vary; some windows may be empty lists because there is no *TimeAndValue* tuple in the list with a time field at or after the start time of the window and before the end time of the window.

If the stream function has a single input stream then f is applied to a single timed window, i.e., to a list of *TimeAndValue* tuples. If the stream function has multiple input streams then f is applied to a list of timed windows, with one timed window for each input stream, where all the windows in the list have the same start and end times.

##### Single Input and Single Output Streams

#### Example

Develop a function with a single input stream and a single output stream where the elements of the output stream are sums of the values in timed windows where the window size is 2.5 and the step size is 1.5. Note that successive windows overlap in this example: the first window has start time 0.0 and end time 2.5; the next window has start time 1.5 and end time 4.0.

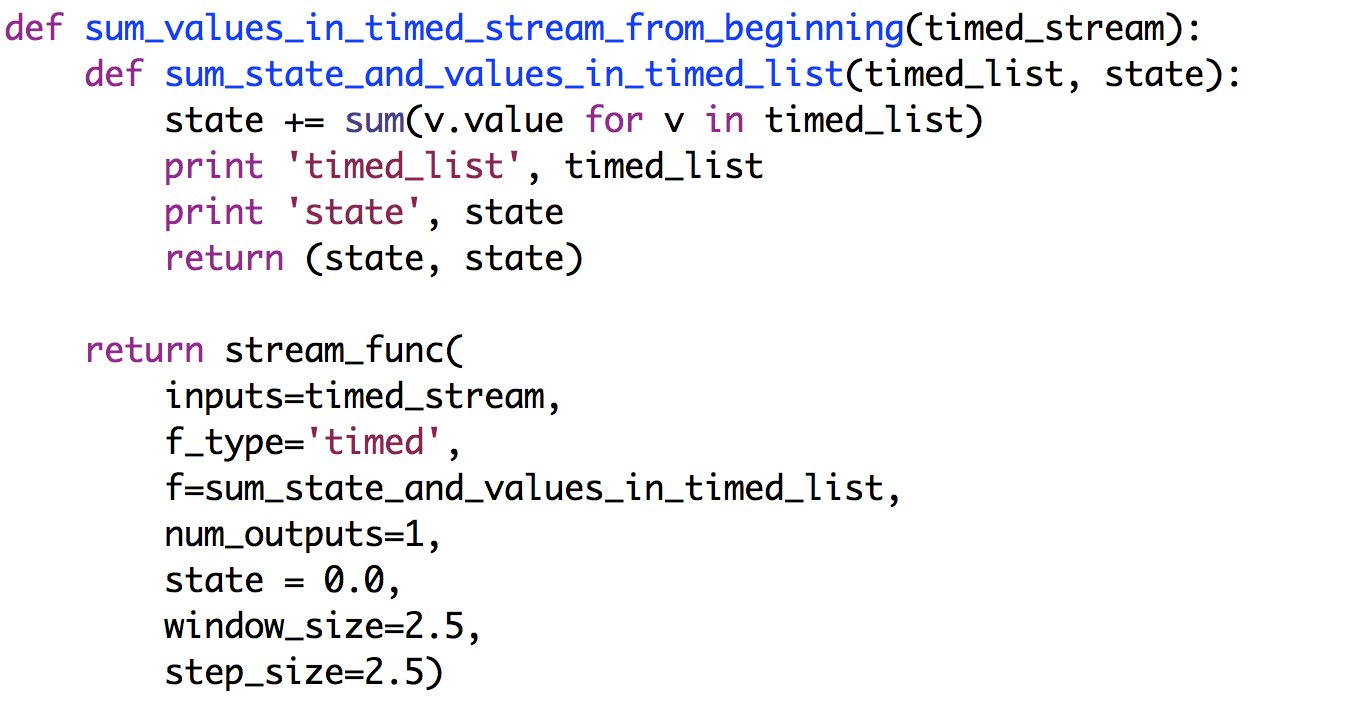


Given a stream *x* we can generate a stream *y* that has the sum of the values in timed windows of *x*:

Macintosh HD:Users:kmchandy:Desktop:Screen Shot 2015-06-14 at 6.43.22 PM.png

#### Example with State

Develop a function with a single input stream and a single output stream where the elements of the output stream are cumulative sums of the values in timed windows from the beginning of the input stream.



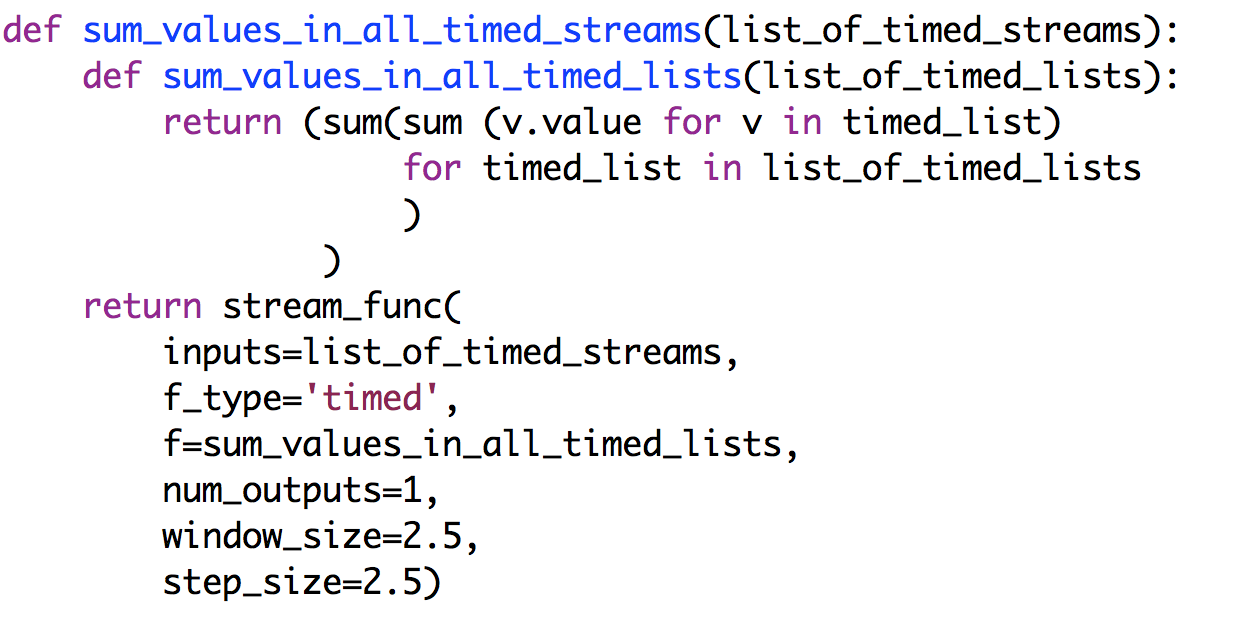
Given a stream *x* we can generate a stream *b* with sums from the beginning of the timed windows of *x*:

Macintosh HD:Users:kmchandy:Desktop:Screen Shot 2015-06-14 at 6.47.15 PM.png

##### Multiple Input and Single Output Streams

#### Example

Develop a stream function with multiple input streams and a single output stream where the output stream contains sums of all the values in the timed windows of all the input streams. The window size is 2.5 and the step size is 2.5.

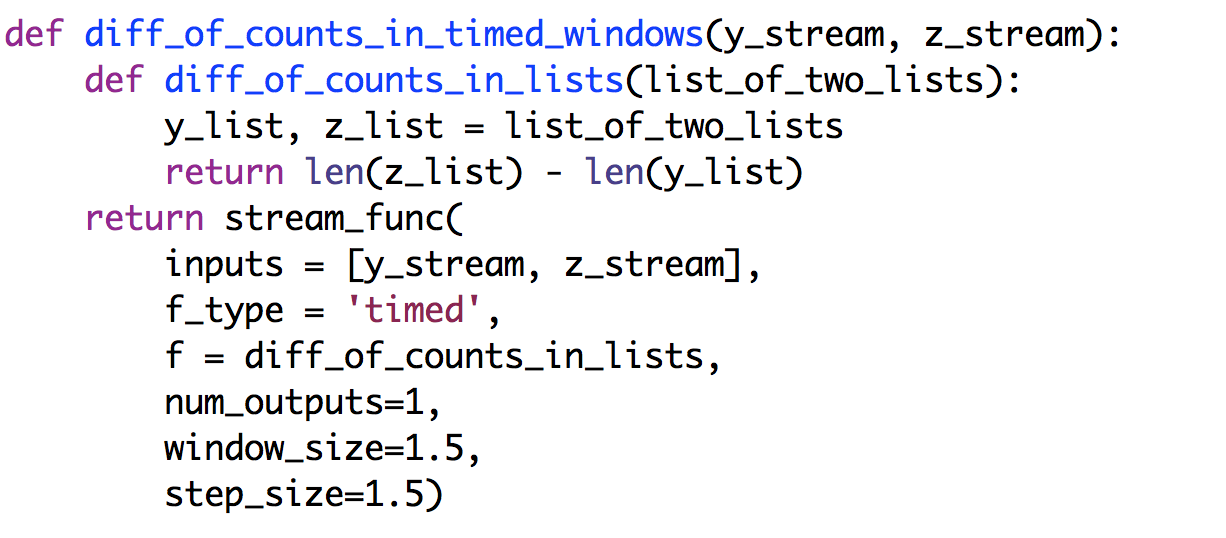


Given a list of streams, for example given streams *x* and *z*, and list [*x,z*], we can generate a stream *a* that sums values in corresponding timed windows of *x* and *z*:

Macintosh HD:Users:kmchandy:Desktop:Screen Shot 2015-06-14 at 6.51.10 PM.png

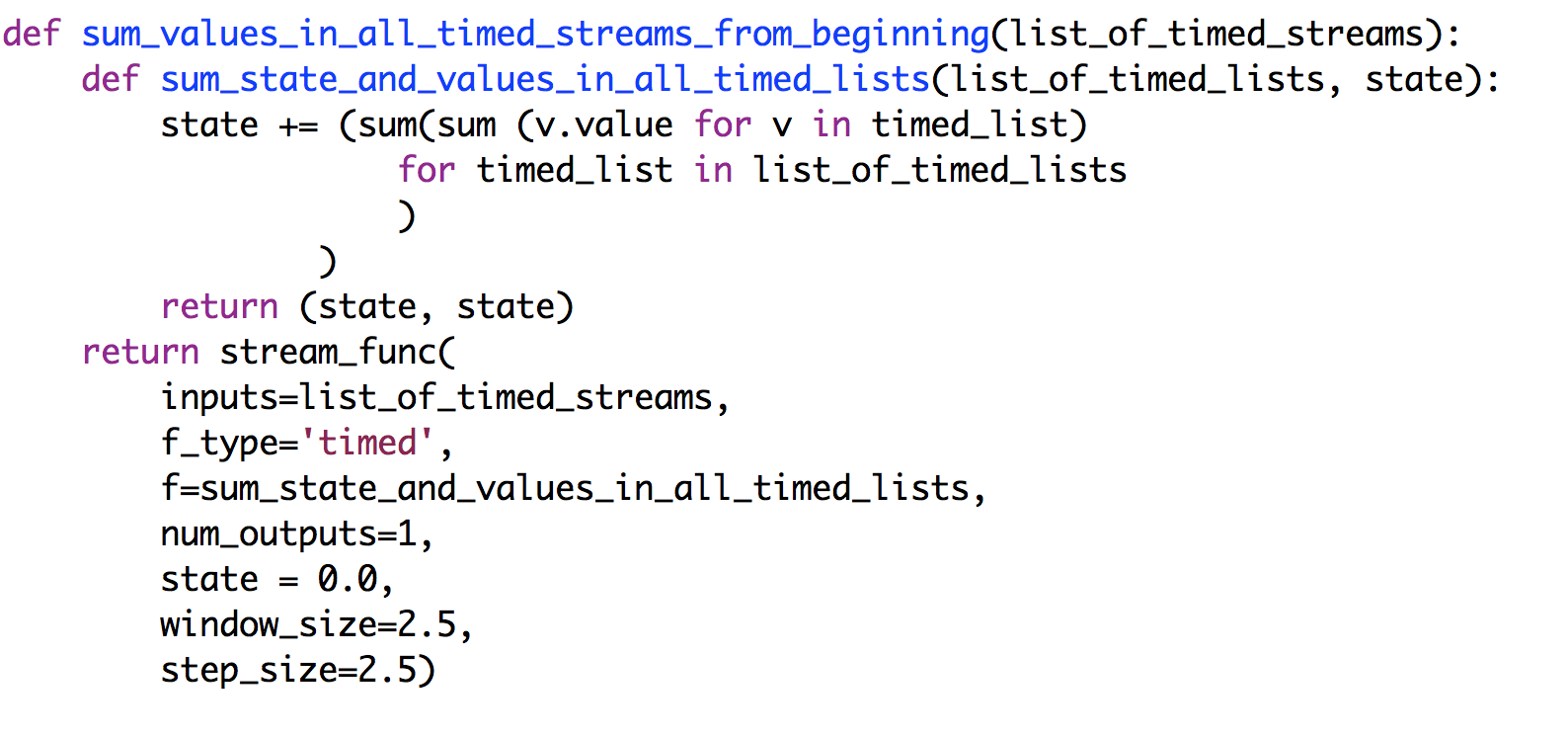
#### Example

Develop a stream function with two input streams and a single output stream where the output stream contains the differences in the number of elements in the timed windows of the two input streams. The window size and step size are both 1.5.



#### Example with State

Develop a stream function with multiple input streams and a single output stream where the output stream contains the cumulative sums of all the values in the timed windows of all the input streams from the beginning of all the input streams. The window size is 2.5 and the step size is 2.5.



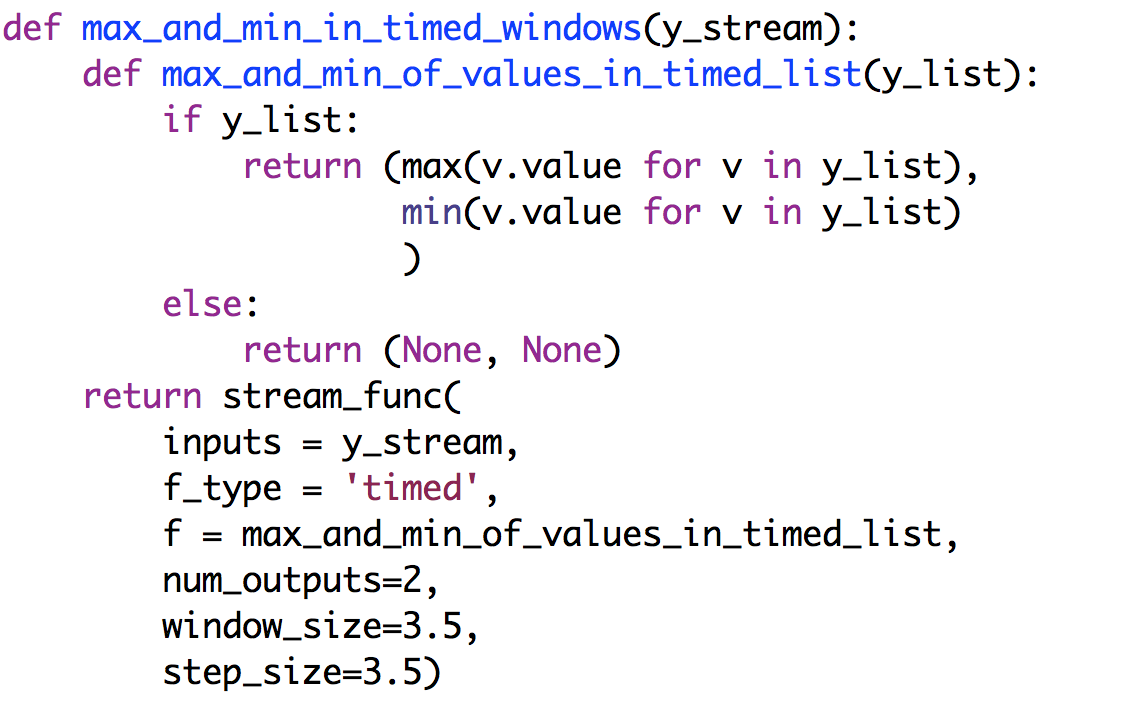
Given a list of streams, for example given streams *x* and *z*, and list [*x,z*], we can generate a stream *c* that outputs the cumulative sum of values in corresponding timed windows of *x* and *z*:

Macintosh HD:Users:kmchandy:Desktop:Screen Shot 2015-06-14 at 6.53.44 PM.png

##### Single Input and Multiple Output Streams

#### Example

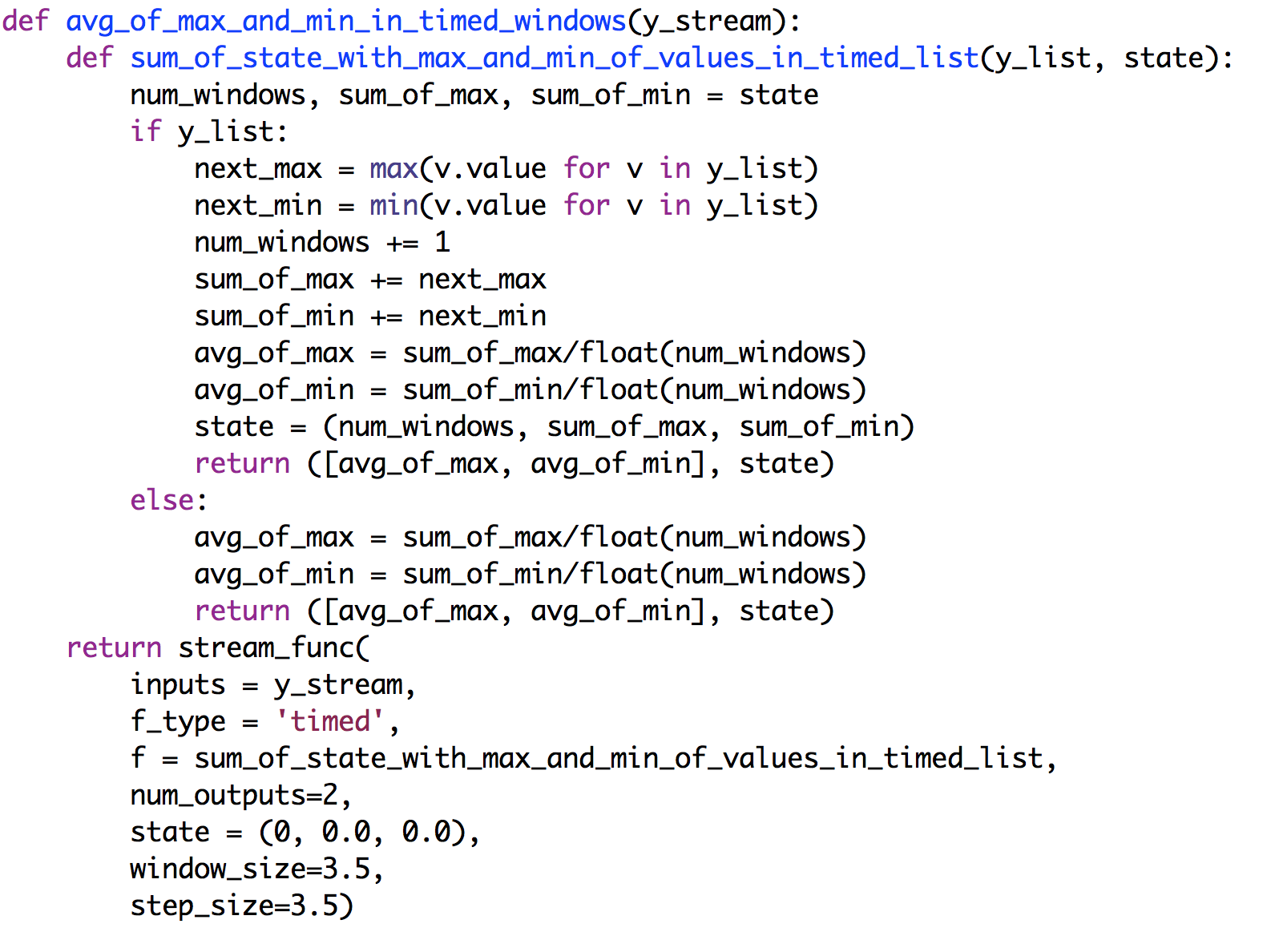
Develop a stream function with a single input streams and two output streams where one output stream contains the maximum value of each time window and the other output stream contains the minimum values.

  
Given a stream *x* we can generate streams *e* and *f* containing the maxima and minima of timed windows:

Macintosh HD:Users:kmchandy:Desktop:Screen Shot 2015-06-14 at 6.56.02 PM.png

#### Example with State

Develop a stream function with a single input streams and two output streams where one output stream contains the average over all nonempty time windows of the maximum values of each time window and the other output stream contains the average over all nonempty time windows of the minimum values.



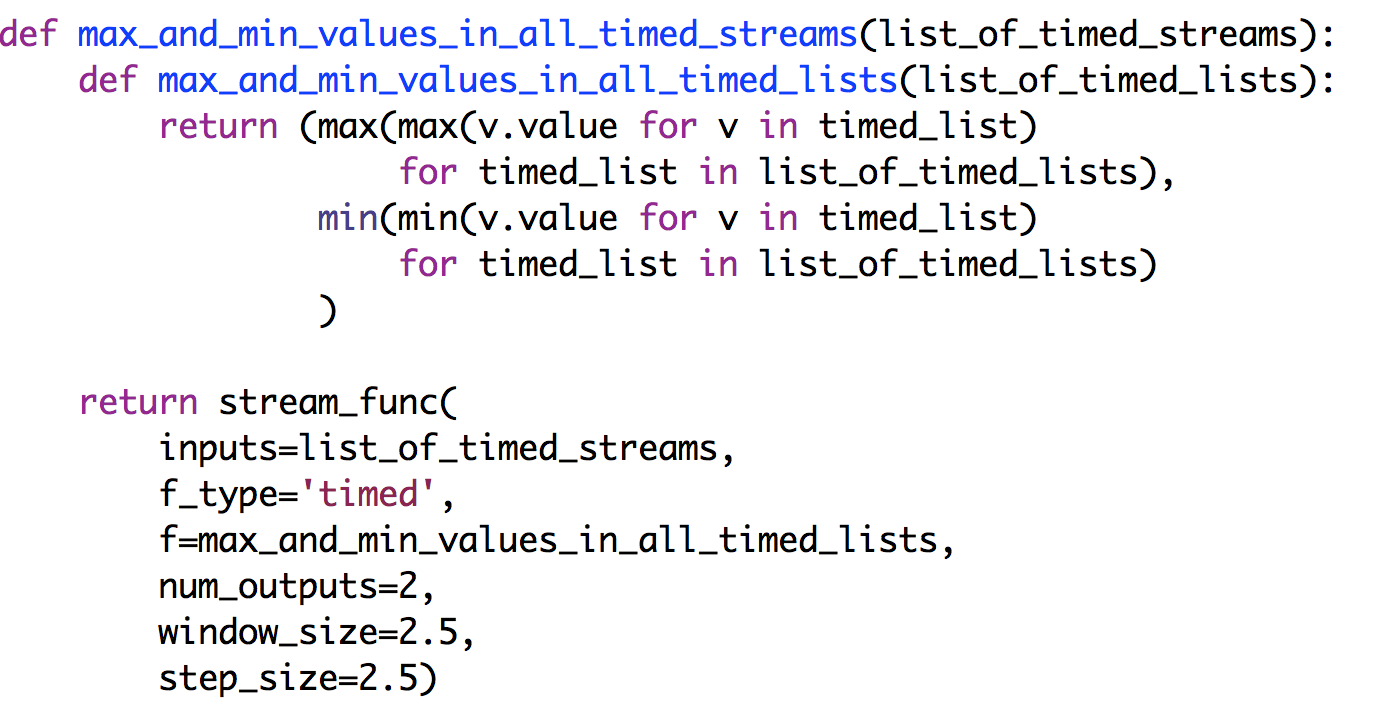
Given a stream *x* we can generate streams *g* and *h* containing the averages of the maxima and minima of timed windows:

Macintosh HD:Users:kmchandy:Desktop:Screen Shot 2015-06-15 at 10.26.56 AM.png

##### Multiple Input and Multiple Output Streams

#### Example

Develop a stream function with a list of one or more input streams and two output streams where one output stream contains the maximum value of each time window across all the input streams and the other output stream contains the minimum values.

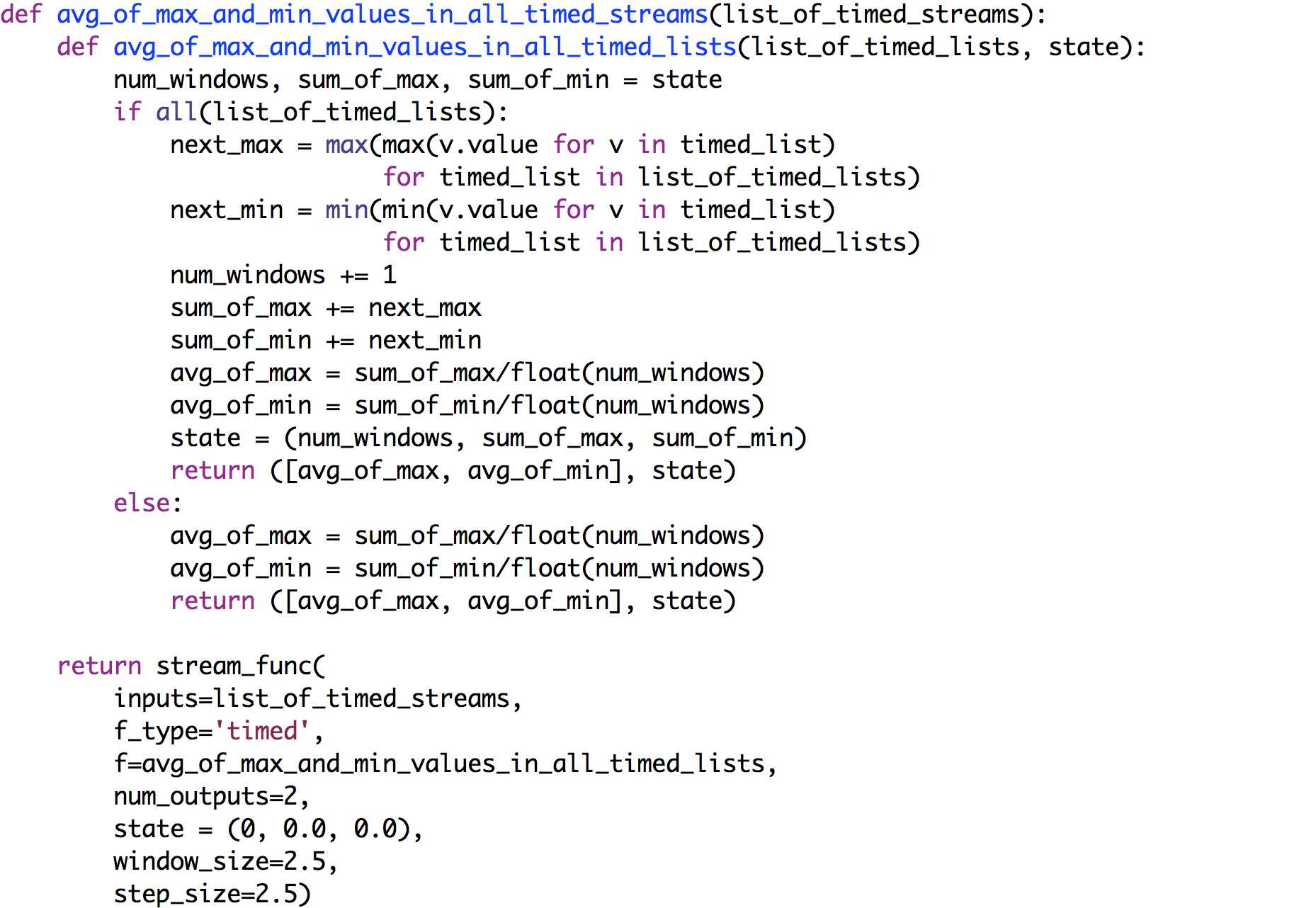


Given streams *x* and *z* we can generate streams *p* and *f* containing the maxima and minima of timed windows:

Macintosh HD:Users:kmchandy:Desktop:Screen Shot 2015-06-15 at 10.38.08 AM.png

#### Example with State

Develop a stream function with a list of one or more input streams and two output streams where one output stream contains the average over time of the maximum value of each time window across all the input streams and the other output stream contains the minimum values.



Given streams *x* and *z* we can generate streams *r* and *s* that contain the averages of the maximum and minimum values of the time windows into streams *x* and *z*:

Macintosh HD:Users:kmchandy:Desktop:Screen Shot 2015-06-15 at 11.23.39 AM.png

##### DATA SOURCES

A data source function is like any other stream function except that the data source has inputs set to *None* and the function is executed when values are appended to a stream in *call\_streams*.

#### Example

Write a data-source function (i.e., a function without input streams and) with a single output stream where the function outputs a single random number on its output stream when it receives a message on a trigger stream.

We first write a function, *ran*, which returns a uniform random number; then we use *ran* with *stream\_func* to obtain the stream function *single\_stream\_of\_random\_numbers*. We sets *inputs* to None, and *call\_streams* to the list [*trigger\_stream*].

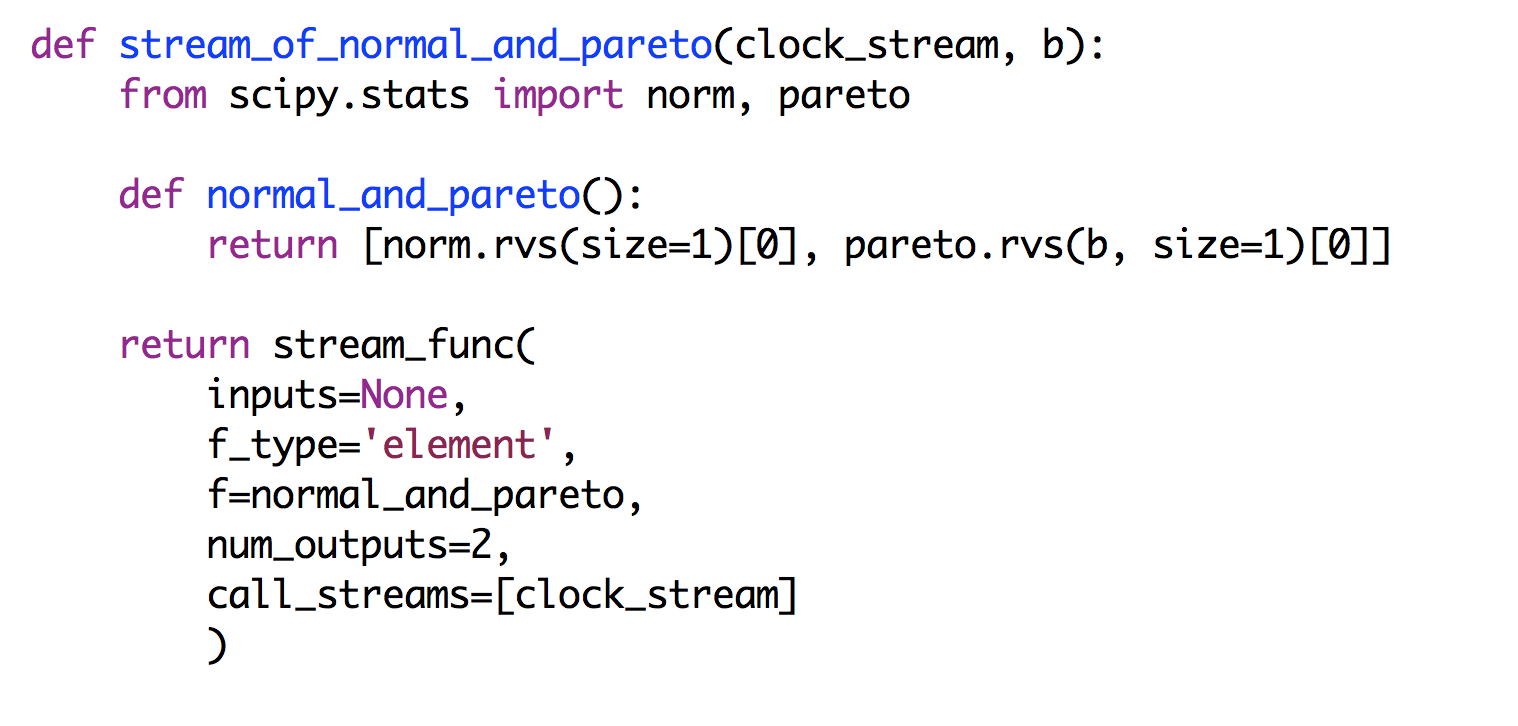
##### Macintosh HD:Users:kmchandy:Desktop:Screen Shot 2015-06-02 at 3.22.38 PM.png

The trigger stream in the above example is usually a stream of clock ‘ticks’ or some other timing signal. Note that in this example, *random* must be imported.

#### Example

Write a data-source function that has two output streams where one output contains random numbers with the standard normal distribution and the other stream contains random numbers with a Pareto distribution with a parameter *b*. The function executes when a message is appended to a stream parameter called *clock\_stream*.

We first develop a function, *norm\_and\_pareto*, which returns a list of two values: a random variable with the normal distribution and another random variable with the Pareto distribution. Then we use *norm\_and\_pareto* with stream\_func to obtain the desired stream function *stream\_of\_normal\_and\_pareto*.



###### DATA SINKS

A data sink has no output streams and receives a *single* input streams. A data sink can be stateless or stateful. Examples of data streams are actuators and printers. We restrict a data sink to have a single input stream because otherwise actuators would have to be programmed to be thread-safe. For example, a braking system with multiple actuator inputs must have logic to manage concurrent commands.

##### Stateless Sink

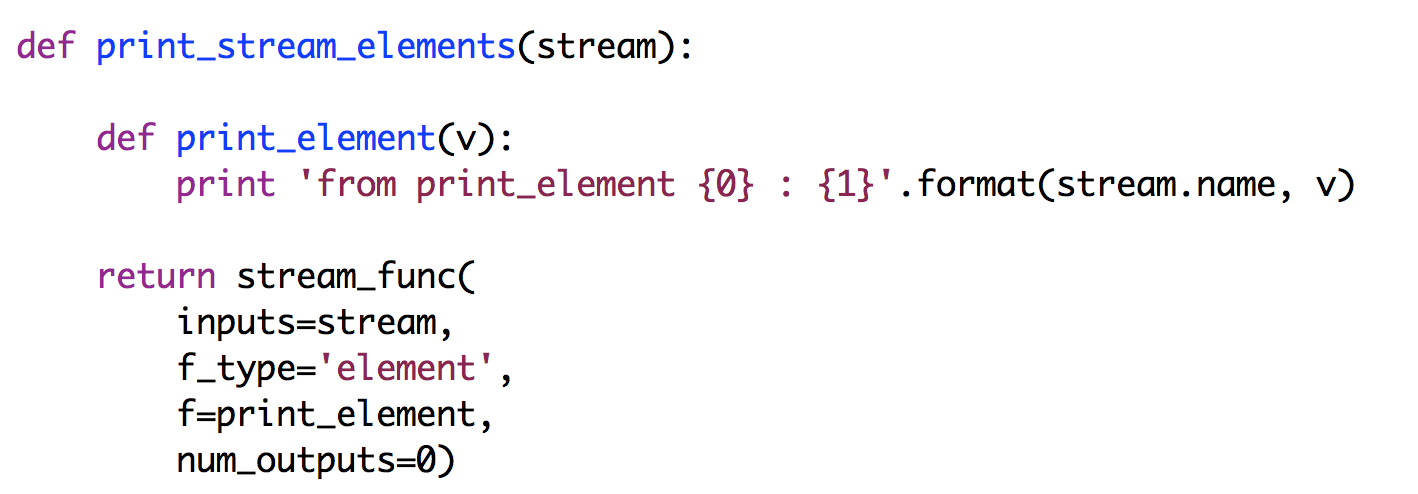
#### Example

Create a sink that prints its input stream.

We can create the sink using ‘element’ or ‘list’ for f\_type. Let’s first use ‘list’. We first develop a function *print\_list* that prints elements of a list. Then we use *print\_list* with *stream\_func* to get the function *print\_stream* that prints elements of a stream.



Next, let’s use *f\_type* as ‘element’. We first develop a function *print\_element* that prints a single element, and then we use *print\_element* with *stream\_func* to get a function *print\_stream\_elements*.

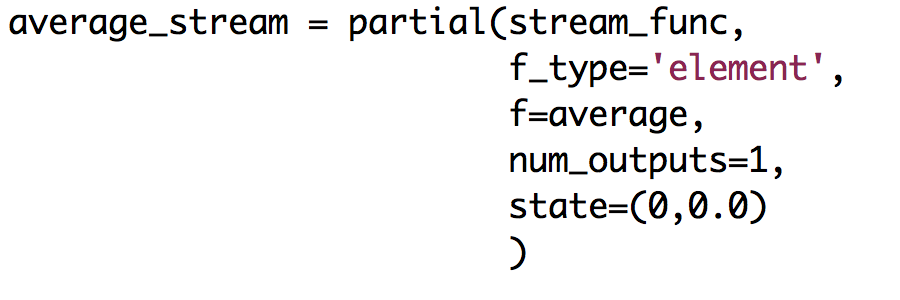


##### Stateful Sink

**Problem**: Create a sink that prints the mean value up to the current point, of its input stream. We change the average function given earlier to print the average and it returns None instead of returning the mean.

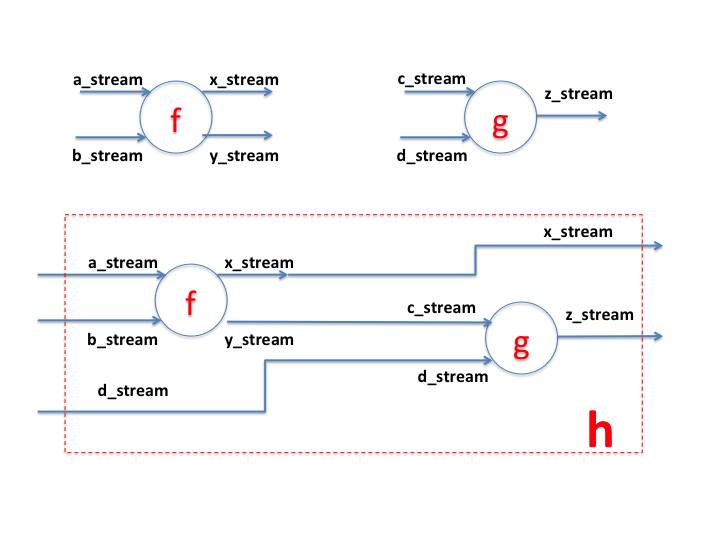


Next, we use the average function with *stream\_func* to get a function *average\_stream* that outputs a stream of average values.



###### FUNCTION COMPOSITION

Stream functions can be combined in the same way as other Python functions. Suppose we had a function f that had two input streams and two output streams, and a function g with two input streams and one output stream, as shown in the diagram. For convenience, we call f’s input streams a\_stream and b\_stream, and we call its output streams x\_stream and y\_stream. We call g’s input streams c\_stream and d\_stream, and we call its output stream z\_stream.

We want to combine the functions *f* and *g* to obtain function h as shown in the diagram. Function *h* has three input streams and two output streams. The input streams of *h* are *a\_stream*, *b\_stream* and *d\_stream*, and its output streams are *x\_stream* and *z\_stream*. Within function *h*, the output *y\_stream* of *f* becomes the input stream, *c\_stream*, of *g*.

In the code below, the elements of *x\_stream* and *y\_stream* are the sums and differences of the corresponding elements of *a\_stream* and *b\_stream*, and *z\_stream* is the sum of corresponding elements of *c\_stream* and d*\_stream*.

