# 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 how frequently a function is executed, synchronizing clocks, time stamps, communication protocols, 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.

### Streams

A *stream* is a sequence of values. Values 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.

This chapter shows how to use functions on (i) elementary Python objects, (ii) lists and (iii) NumPy arrays to streams.

### From Lists to Streams

Here, we consider streams in which the value of a stream, at any point in time, is a list of Python objects (messages). Later, we describe streams with values that are NumPy arrays.

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].

Next, we describe a way of using a function on lists to define a function on streams.

#### Functions on Streams

Here we describe functions with the following signature:

Macintosh HD:Users:kmchandy:Desktop:Screen Shot 2015-03-02 at 1.01.16 PM.png

where *stream\_func* returns *None*, or a stream, or a list of two or more streams. The arguments of *stream\_fun*c are:

1. *list\_func*: a function on lists. *list\_func* is described below.
2. *inputs*: None, or a single stream, or a list of two or more streams. This specifies the input streams of the function.
3. *num\_outputs*: a nonnegative integer that specifies the number of output streams of *stream\_func*.
4. *state*: An object which records information about earlier computations of *stream\_func* that is used for the next computation of *stream\_func*. The type of *state* is arbitrary. *state* is *None* for stateless *stream\_func* and *state* is not *None* for stateful *stream\_func*. The value passed for *state* in the call to *stream\_func* is the value of the initial state.
5. *call\_streams*: A list of streams or *None*. If *None* is assigned to *call\_streams* in the call to *stream\_func*, then *inputs* is automatically assigned to *call\_streams*. stream\_func is executed when, and only when, any stream in *call\_streams* is modified.

If *inputs* is *None*, then this function has no input streams; in this case the function defines a stream *source*. A sensor that generates a stream of data is an example of a stream source.

If *num\_outputs* is 0, then this function has no output streams; in this case the function defines a stream *sink*. An actuator that receives streams of data, but that produces no streams itself, is an example of a sink. A printer is an example of an actuator that we use in many examples.

If *inputs* is not *None* and *num\_outputs* is positive, then *stream\_func* maps input streams to output streams. A function that detects a pattern in a single input stream and sends an alert on its single output stream when the pattern is detected is an example of *stream\_func* for which *inputs* is a single stream and *num\_outputs* is 1.

Next, we describe the signature of list function, *list\_func*, that is used for different arguments of *stream\_func*. The rule for the signature of *list\_func* is that a stream in *stream\_func* corresponds to a list in *list\_func*.

##### Stateless Functions: *state* is *None*

Next, we consider the case where the argument, *state*, has value *None* in the call to *stream\_func*.

###### Arguments of the list function, list\_func

1. If *inputs* is *None*, then *list\_func* has no arguments
2. If *inputs* is a stream, then *list\_func* has a single argument: a list.
3. If *inputs* is a list of streams, then *list\_func* has a single argument: a list of lists.

###### Returns: Values returned by the list function, list\_func

1. If *num\_outputs* is *0*, then *list\_func* returns no value.
2. If *num\_outputs* is 1, then *list\_func* returns a single value: a list.
3. If *num\_outputs* is more than 1, then *list\_func* returns a single value: a list of lists.

##### Stateful Functions

Next, we consider the case where the argument, *state*, has value other than *None* in the call to *stream\_func*.

###### Arguments of the list function, list\_func

* 1. If *inputs* is *None*, then *list\_func* has a single argument: *state*.
  2. If *inputs* is a stream, then *list\_func* has two arguments: a list, and *state*.
  3. If *inputs* is a list of streams, then *list\_func* has two arguments: a list of lists and *state*.

###### Returns: Values returned by the list function, list\_func

* 1. If *num\_outputs* is *0*, then *list\_func* returns a single value: *state*.
  2. If *num\_outputs* is 1, then *list\_func* returns two values: a list and *state*.
  3. If *num\_outputs* is more than 1, then *list\_func* returns two values: a list of lists and *state*.

##### Examples of Lists to Streams

Next, we present examples, from the Examples folder, that show how functions on streams are obtained from functions on lists. The goal of these examples is to show how to develop functions on streams by developing functions on lists. The examples are very simple because the logic is implemented in standard Python functions on lists, and the examples are not intended to illuminate standard Python programming.

###### Data Sources

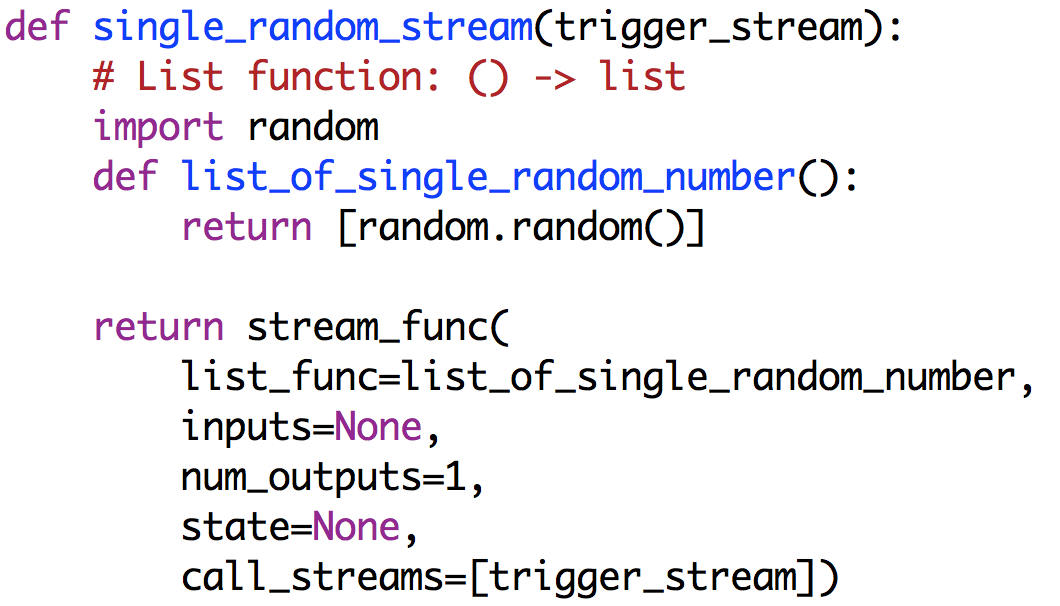
Sensors and RSS feeds are examples of data sources. A data source has no input streams and generates one or more output streams. A data source can be stateless or stateful. Next we give very simple examples to illustrate how functions on lists are used to create data sources.

##### Single-Output Stateless Source

**Problem**: Create a source that generates a single stream of random numbers uniformly distributed in the interval [0.0, 1.0). The source outputs an additional single random number whenever a stream called trigger\_stream is modified.

**List Function**: Random numbers can be generated using a state (which records the seed of the random number generator) or without state. We will use a stateless version. We write a function on lists that takes no arguments because *stream\_func* is stateless and is a source (and therefore has no input streams). The list function returns a single list because the source generates a single stream. So, we define the function: *list\_of\_single\_random\_number* that creates a list of a single random number.

**Stream Function**: We use *stream\_func* to create the stream source. The *inputs* parameter is *None* because the source has no inputs; *num\_outputs* is 1 because the source has a single output stream, *state* is *None* since the function is stateless, and *call\_streams* is a list containing a single element: *trigger\_stream*. The stream function returns a stream since *num\_outputs* is 1. Note that the value returned by *stream\_func* is a stream and not a list containing a stream.

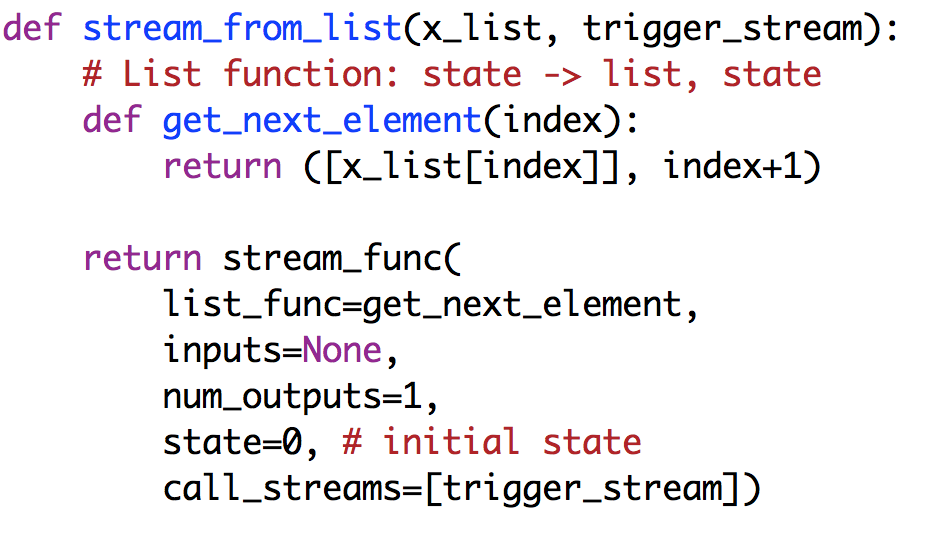


##### Single-Output Stateful Source

**Problem**: Create a source that generates a stream consisting of the sequence of values in a list, called *x\_list*. The source outputs the next value in the list when a stream, *trigger\_stream*, is modified. (An exception is raised after all the elements of *x\_list* have been output.)

**List Function**: We write a function, *get\_next\_element*, on lists that has a parameter (the state) called *index*, and that returns a list and a new state. The list returned is a list consisting of a single value – *x\_list*[*index*]– and the new state is *index+1*.

**Stream Function**: We use *stream\_func* to create the stream source. The *inputs* parameter is *None* because the source has no inputs; *num\_outputs* is 1 because the source has a single output stream, *state* is the value of the initial state, which is 0, and *call\_streams* is a list containing a single element: *trigger\_stream*. The stream function returns a stream since *num\_outputs* is 1. Note that the value returned by *stream\_func* is a stream and not a list containing a stream.

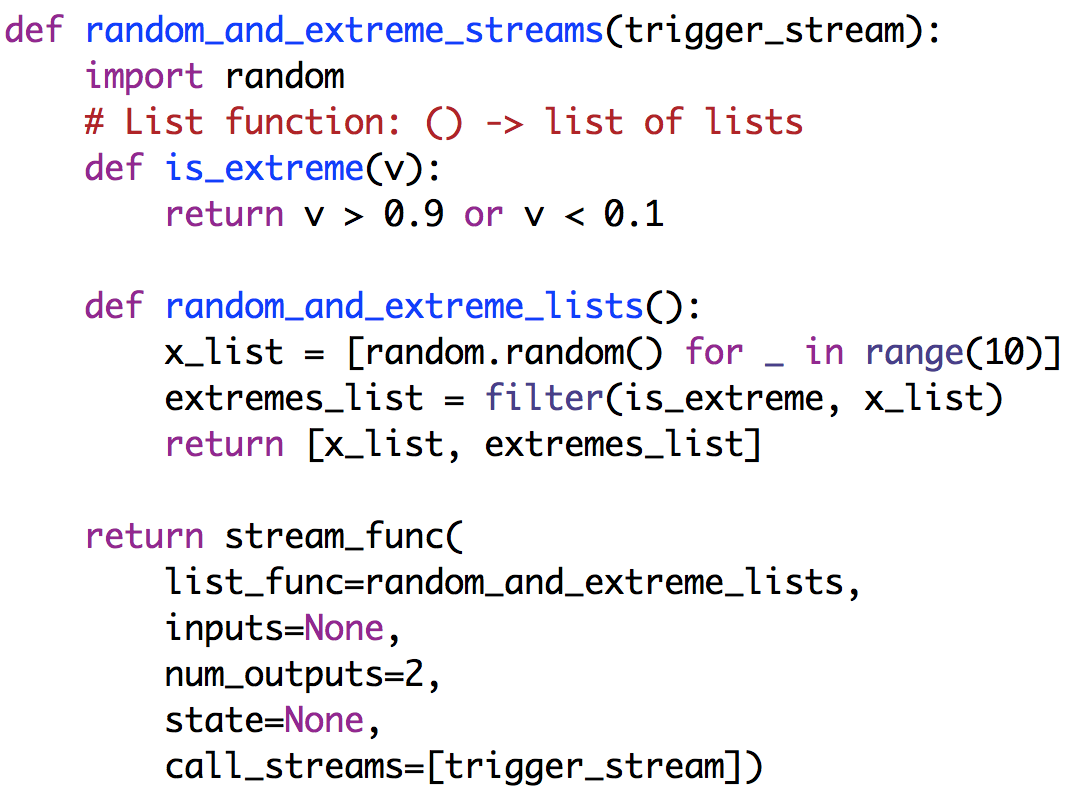


##### Many Outputs Stateless Source

**Problem**: Create a source that generates two streams where the first stream consists of random numbers uniformly distributed in the interval [0.0, 1.0) and the second stream consists of those values in the first stream that are greater than 0.9 or less than 0.1. The source outputs 10 values in the first stream when a stream, *trigger\_stream*, is modified.

**List Function**: We write a function, *random\_and\_extreme\_lists* that has no arguments and returns a list of two lists.

**Stream Function**: We use *stream\_func* to create the stream source. The *inputs* parameter is *None* because the source has no inputs; *num\_outputs* is 2 because the source has 2 output streams, *state* is *None* since the function is stateless, and *call\_streams* is a list containing a single element: *trigger\_stream*. The stream function returns a list of two streams since *num\_outputs* is 2.

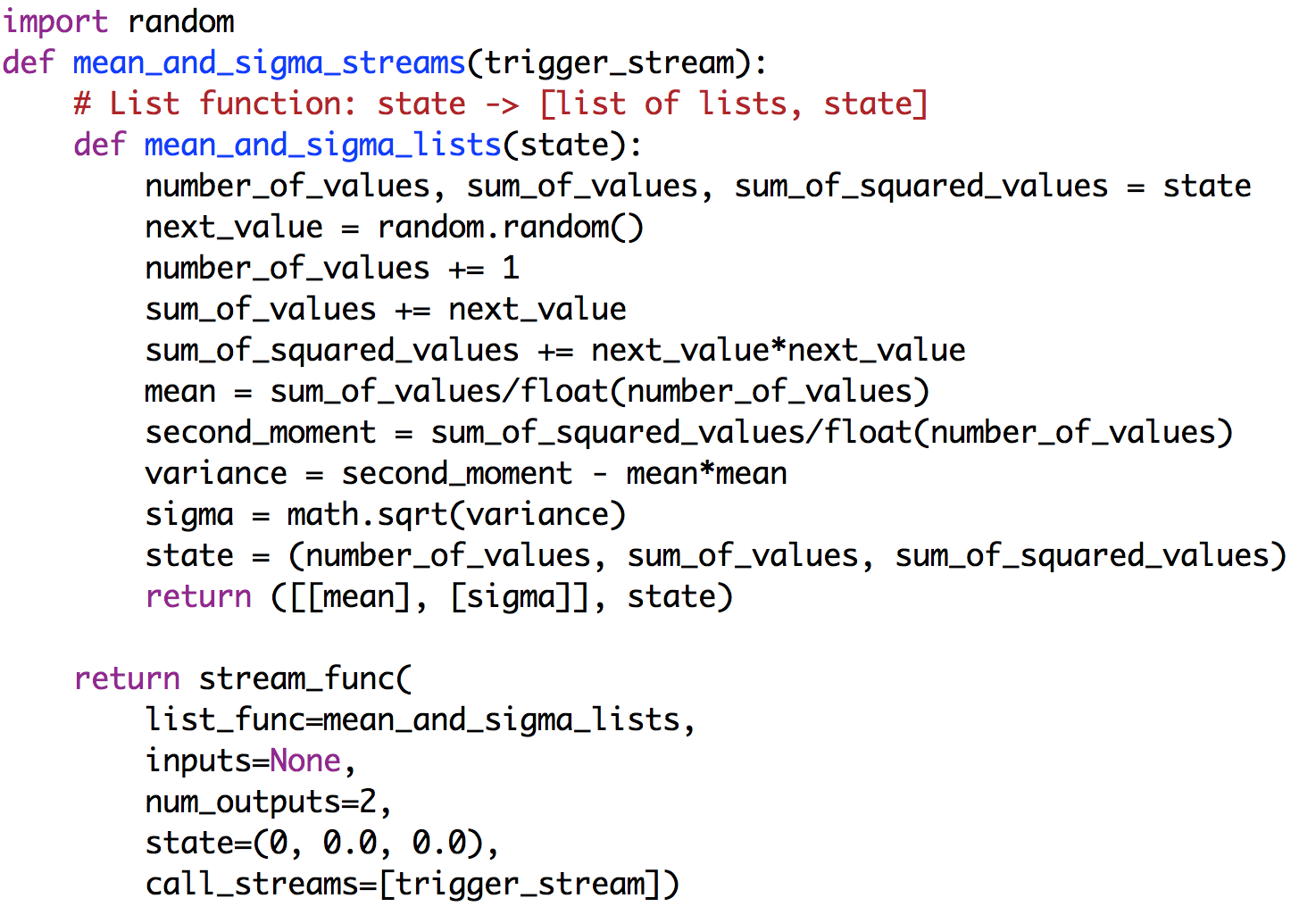


##### Many Outputs Stateful Source

**Problem**: Create a source with two output streams where one stream contains the average of a sequence of uniform random numbers in the interval [0.0, 1.0) and the other stream contains its standard deviation. The source outputs a list containing the average and standard deviation when a stream, called *trigger\_stream*, is extended.

**List Function**: We write a function, *mean\_and\_sigma* that has no arguments and returns a list of two lists.

**Stream Function**: We use *stream\_func* to create the stream source. The *inputs* parameter is *None* because the source has no inputs; *num\_outputs* is 2 because the source has 2 output streams, state is (*number\_of\_values*, *sum\_of\_values*, *sum\_of\_squared\_values*), and *call\_streams* is a list containing a single element: *trigger\_stream*. The initial value of *state* is (0, 0.0, 0.0). The stream function returns a list of two streams since *num\_outputs* is 2.



###### Data Sinks

An actuator can be a data sink. A data sink has no output streams and receives one or more output streams. A data sink can be stateless or stateful. Next we give very simple examples to illustrate how functions on lists are used to create data sinks.

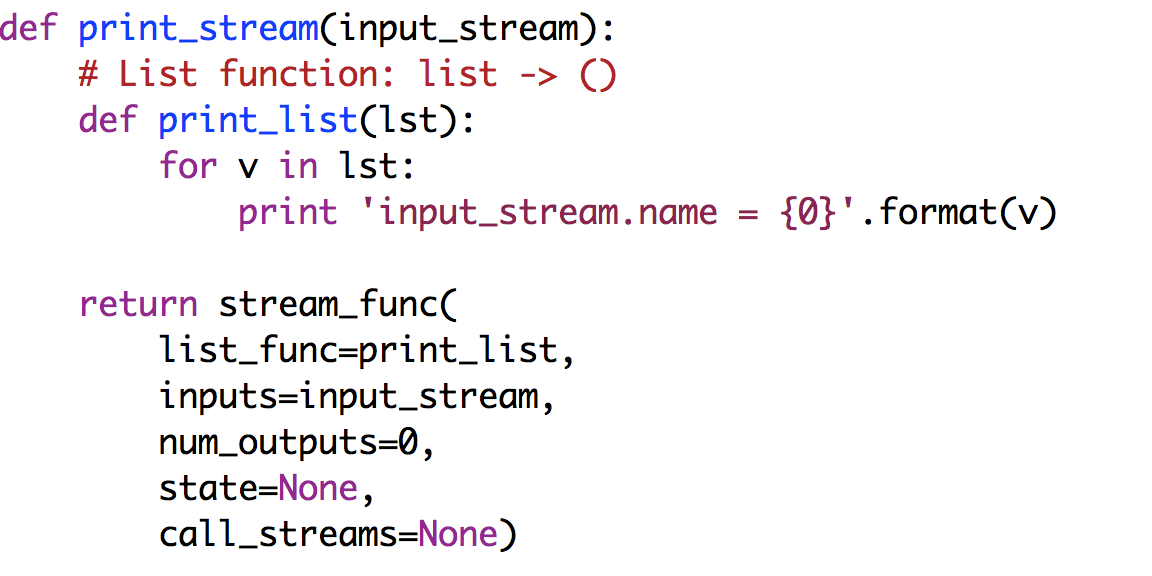
##### Single Input Stateless Sink

**Problem**: Create a sink that prints its input stream.

**List Function**: We write a function, *print\_list*, that has a single argument, a list, and that does not return a value. The function prints its argument.

**Stream Function**: We use *stream\_func* to create the stream source. The *inputs* parameter is a single input stream, *input\_stream*; *num\_outputs* is 0 because the sink has no outputs, *state* is *None* because this operation is stateless, and *call\_streams* is *None*. Since the argument for *call\_streams* is *None*, *call\_streams* is automatically set to [*input\_stream*], and hence *stream\_func* is invoked each time *input\_stream* is extended. The stream function returns no values since the sink has no output streams.

Since the function *print\_stream* may be called to print different streams on the same printer, printing the name of the stream helps to differentiate output from multiple streams. The name of a stream *s* is obtained by *s.name*, and its value is set by *s.set\_name*(*str*). Names of streams are discussed later.

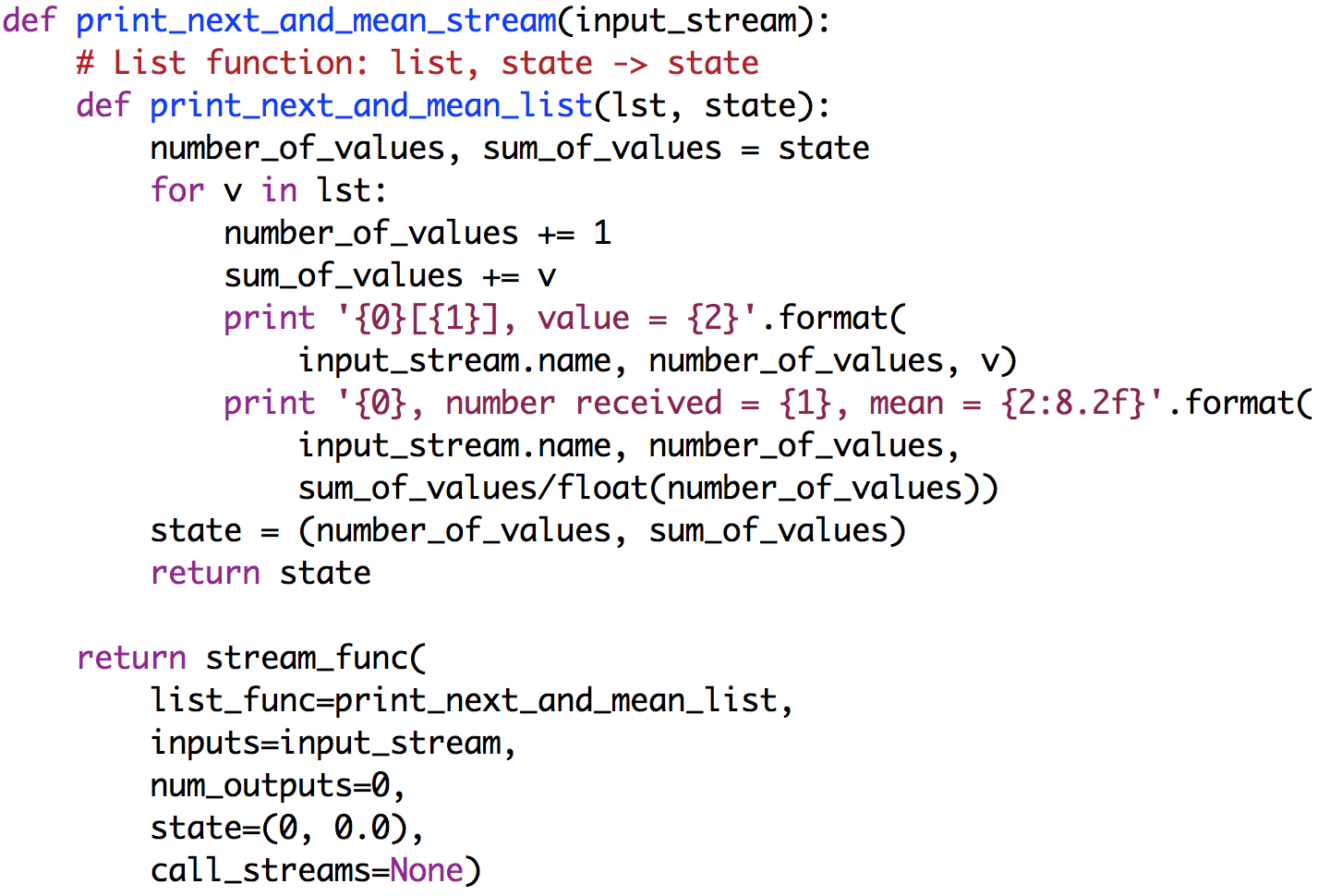


##### Single Input Stateful Sink

**Problem**: Create a sink that prints: (i) the next value and (ii) the mean value up to the current point, of its input stream.

**List Function**: We write a function, *print\_next\_and\_mean*, that has two arguments: a list and a state. The function prints the next value, computes and prints the mean, and returns the next state. The state is (*number\_of\_values*, *sum\_of\_values*).

**Stream Function**: We use *stream\_func* to create the stream source. The *inputs* parameter is a single input stream, *input\_stream*; *num\_outputs* is 0 because the sink has no outputs, *state* is (*number\_of\_values*, *sum\_of\_values*) which is initially (0, 0.0), and *call\_streams* is *None*. Since the argument for *call\_streams* is *None*, *call\_streams* is automatically set to [*input\_stream*], and hence *stream\_func* is invoked each time *input\_stream* is extended. The stream function returns no values since it has no output streams.

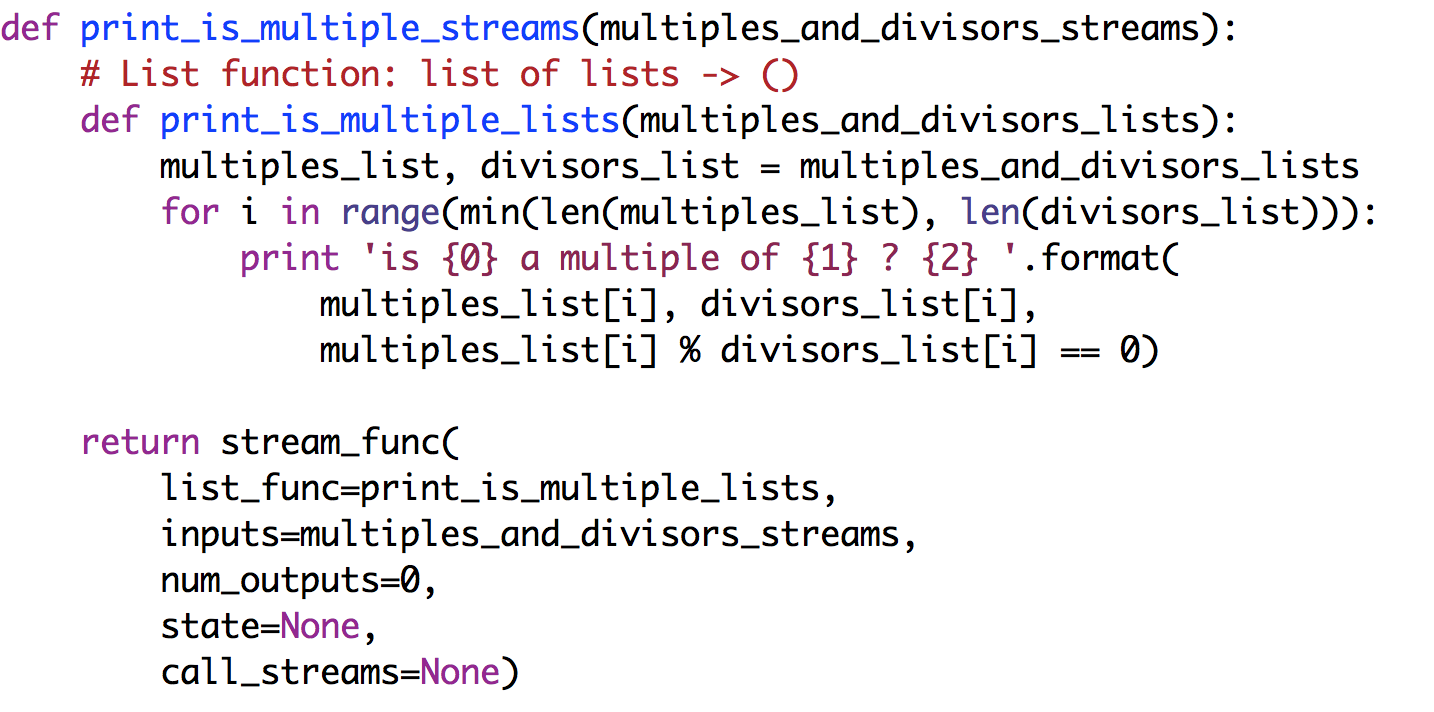


##### Many Inputs Stateless Sink

**Problem**: Write the program for a sink that receives two streams of positive integers, *multiples\_stream* and *divisors\_stream*. It prints the values received on each stream and a Boolean indicating whether the value in the *multiples\_stream* is a multiple of the value in the *divisors\_stream*.

**List Function**: We write a function, *print\_is\_multiple\_lists*, that has a single argument, a list of lists, and that does not return a value. A side effect of the function is to print the results.

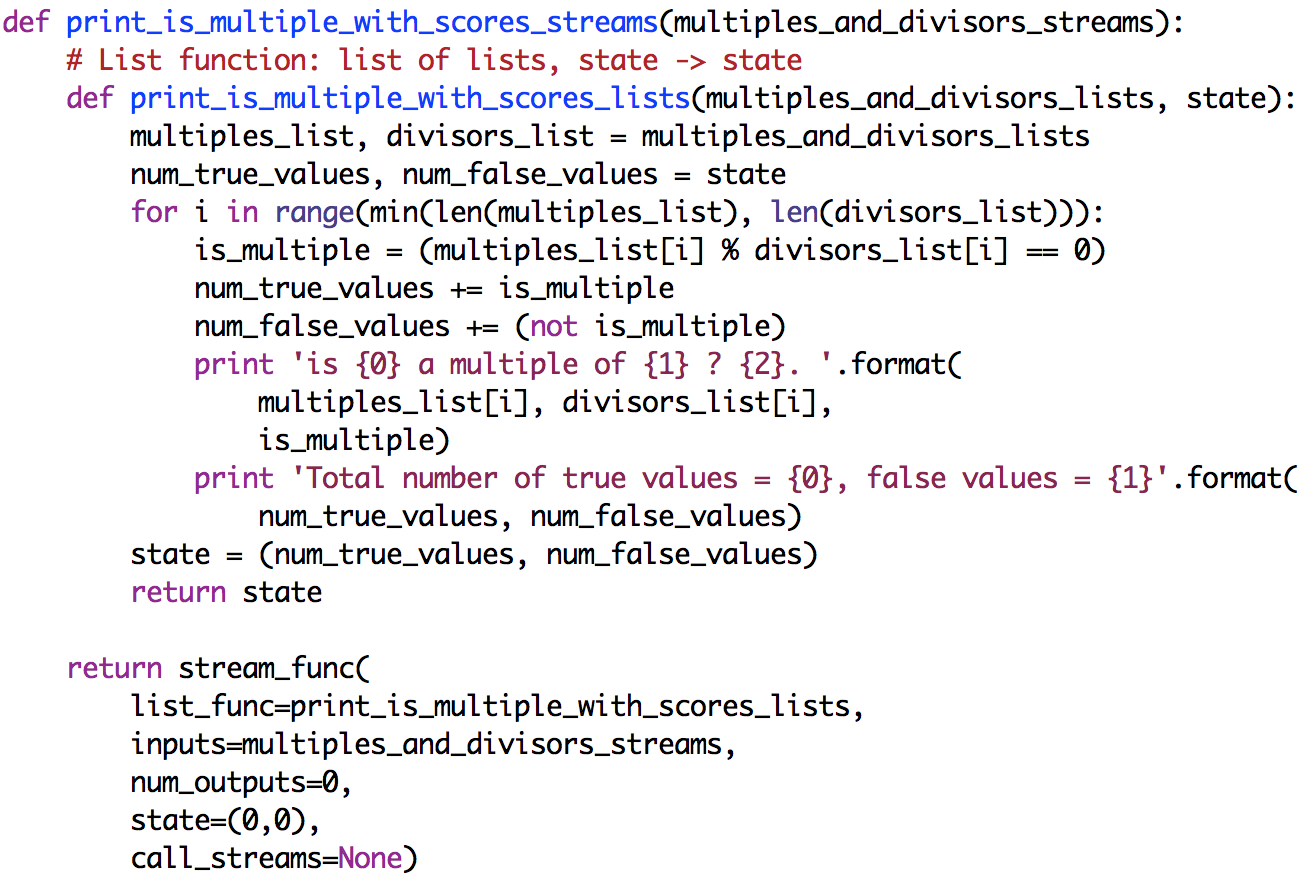
**Stream Function**: We use *stream\_func* to create the stream source. The *inputs* parameter is a list of two input streams, *multiples\_stream* and *divisors\_stream*; *num\_outputs* is 0 because the sink has no outputs, *state* is *None* because this operation is stateless, and *call\_streams* is *None*. Function *stream\_func* returns no values since the sink has no output streams. Since the argument for *call\_streams* is *None*, *call\_streams* is automatically set to [*multiples\_stream, divisors\_stream*], and hence *stream\_func* is invoked when either of the input streams is extended.



##### Many Inputs Stateful Sink

**Problem**: This problem is the same as that of the previous example except that the number of True results and the number of False results are also printed.

**Process of creating stream functions from list functions:** The process of creating the list and stream functions is obvious from the previous examples, and so, we won’t repeat it here. *stream\_func* takes a list of streams as its argument and returns no value.

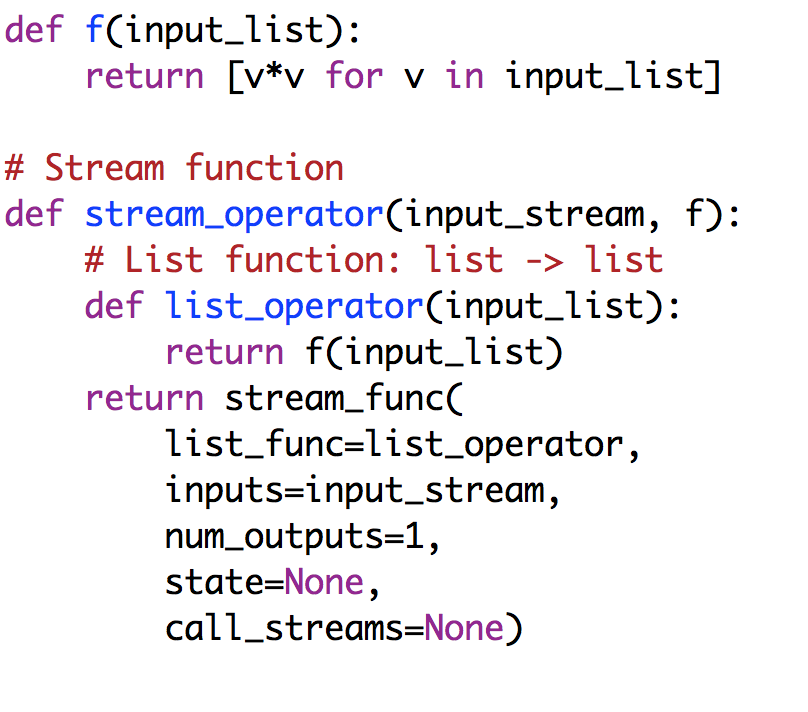


###### Data Stream Operators

A stream operator has a single input stream and a single output stream. Stream operators are convenient constructs for many applications.

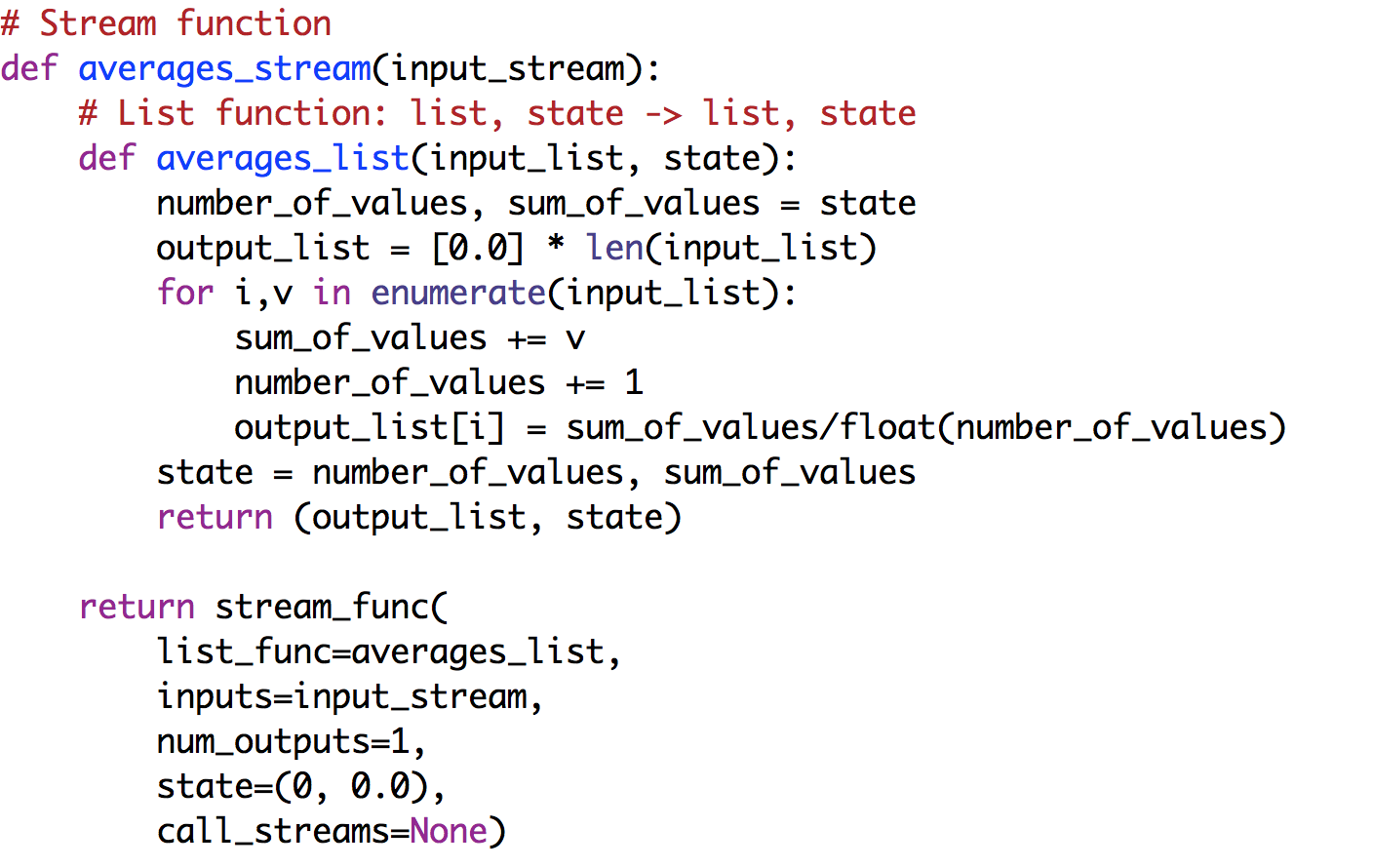
##### Stateless Stream Operator

**Problem**: Write a function that takes (i) *input\_stream*, a stream of numbers and (ii) a function *f* on lists as input. The function returns a single output stream that consists of function applied to the stream, treating the stream as a list.



##### Stateful Stream Operator

**Problem**: Write a function that takes a stream of numbers as input and produces a single output stream that consists of the average of all the inputs up to that point.

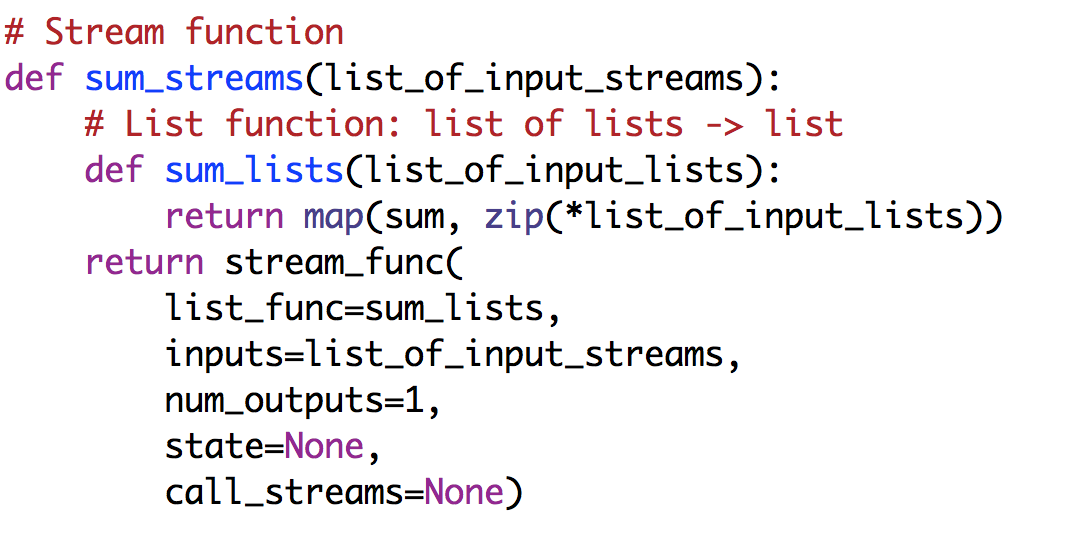


###### Merge Streams

A merge has two or more input streams and a single output stream. A merge can be stateless – in which case the list function has no state – or stateful in which case the list function has state as an input and returns a new state.

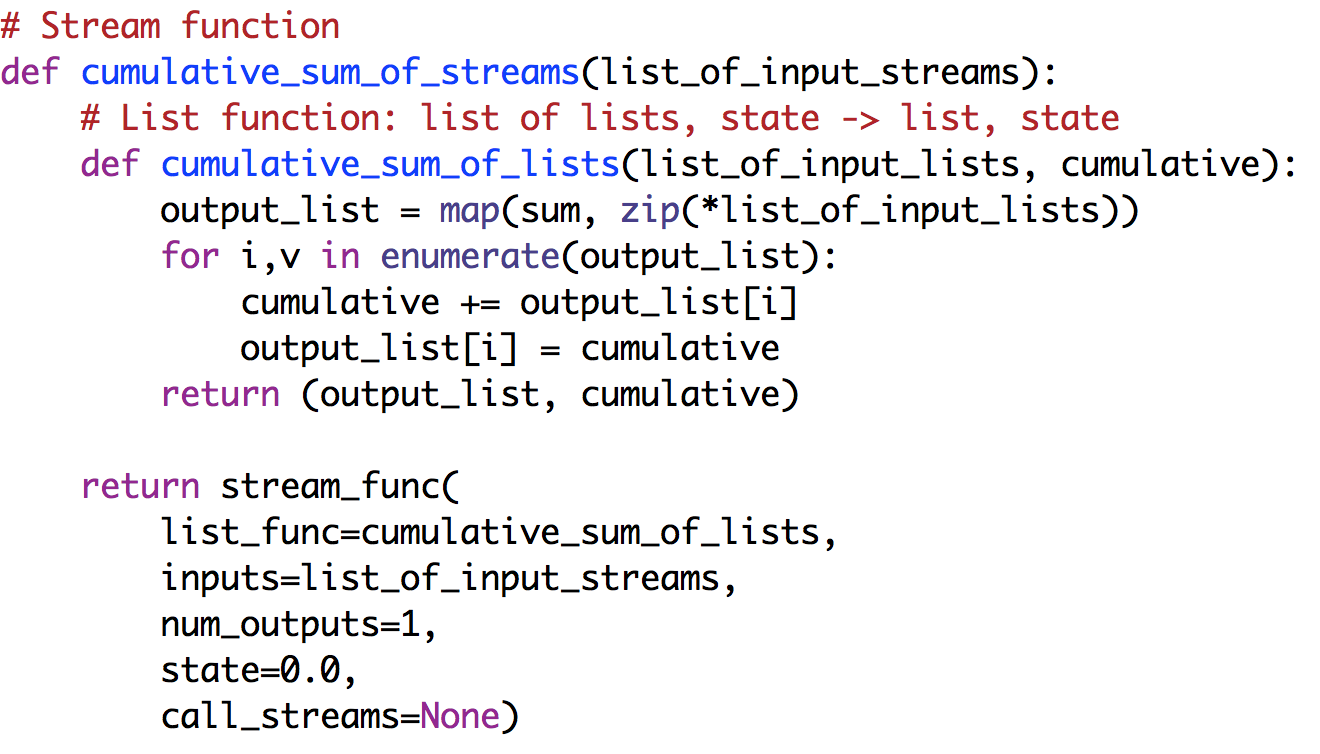
##### Stateless Merge

**Problem**: Write a function that takes two or more streams of numbers as input and produces a single output stream that consists of the sums of corresponding elements of the input streams: the n-th element on the output stream is the sum of the n-th elements over all the input streams.



##### Stateful Merge

**Problem**: Write a function that takes two or more streams of numbers as input and produces a single output stream that consists of the cumulative sum of the all the values received on the input streams.

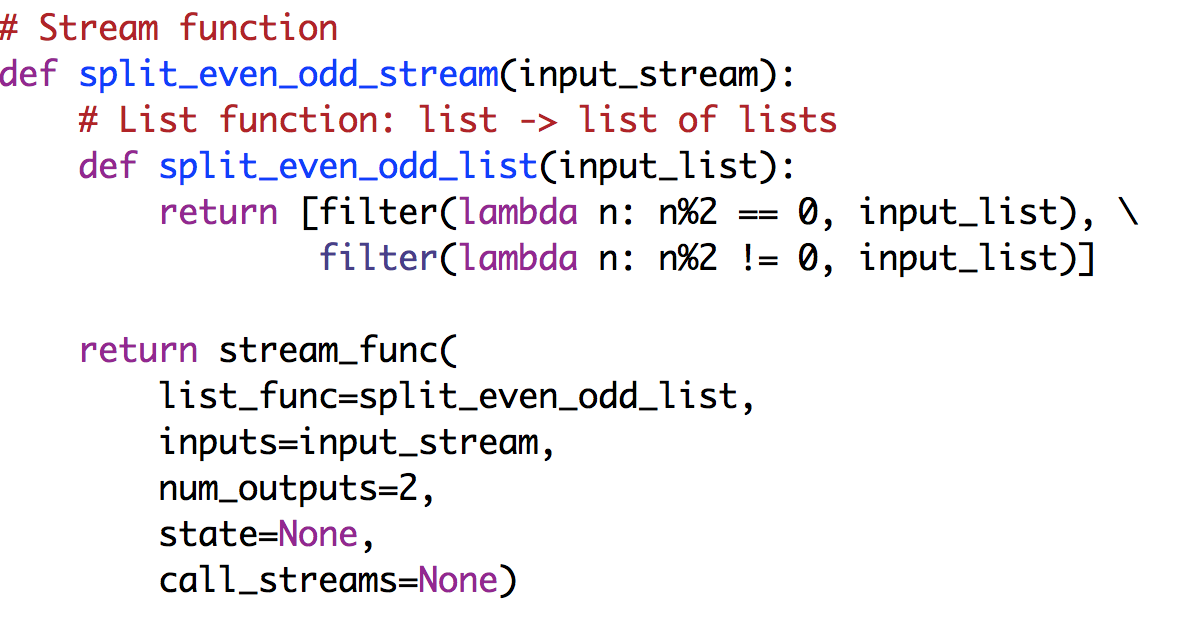


###### Split Streams

A split has one input stream and more than one output stream. The stream function has a single argument – a single stream – and returns a list of streams. The corresponding list function has a single argument – a single stream – and returns a list of lists.

##### Stateless Split

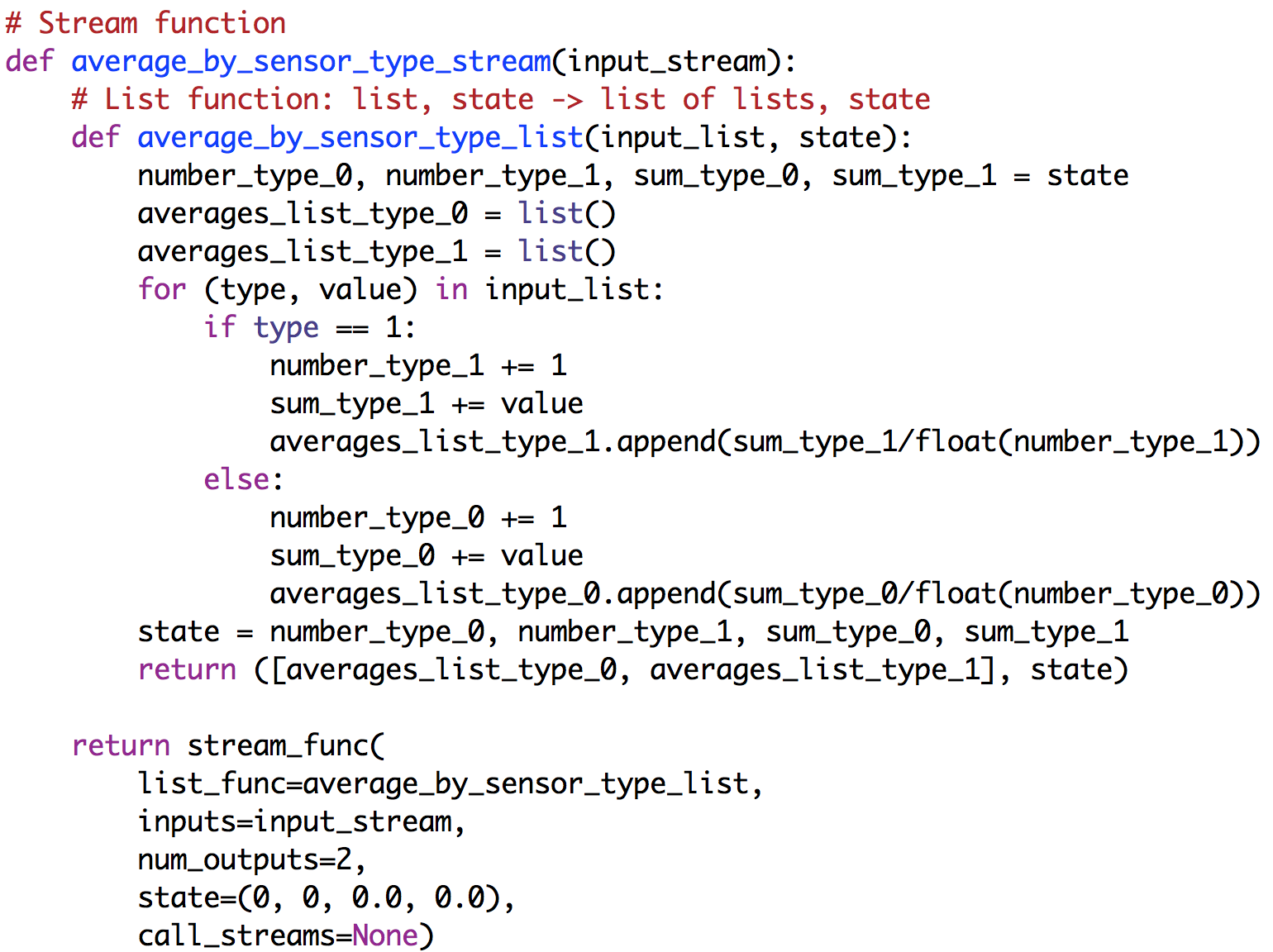
**Problem**: Write a function, *split\_even\_odd\_stream,* that has a single input stream, which is a stream of integers. It returns a list of two output streams; values in the input stream that are even-numbered are output on one stream and odd-numbered values are output on the other stream.



##### Stateful Split

**Problem**: Write a function that has a single argument: a stream of pairs (type, value) where type is either 0 or 1, and value is an integer. The function returns a list of two output streams; one output stream contains the running averages of values of type 0 input, and the other output stream has the averages for type 1 input.

The state for each type records the number of instances (*number\_type\_0*, *number\_type\_1* ) of types 0 and 1, received on the input stream, and the running sums (*sum\_type\_0*, *sum\_type\_1*) of the values of types 0 and 1.

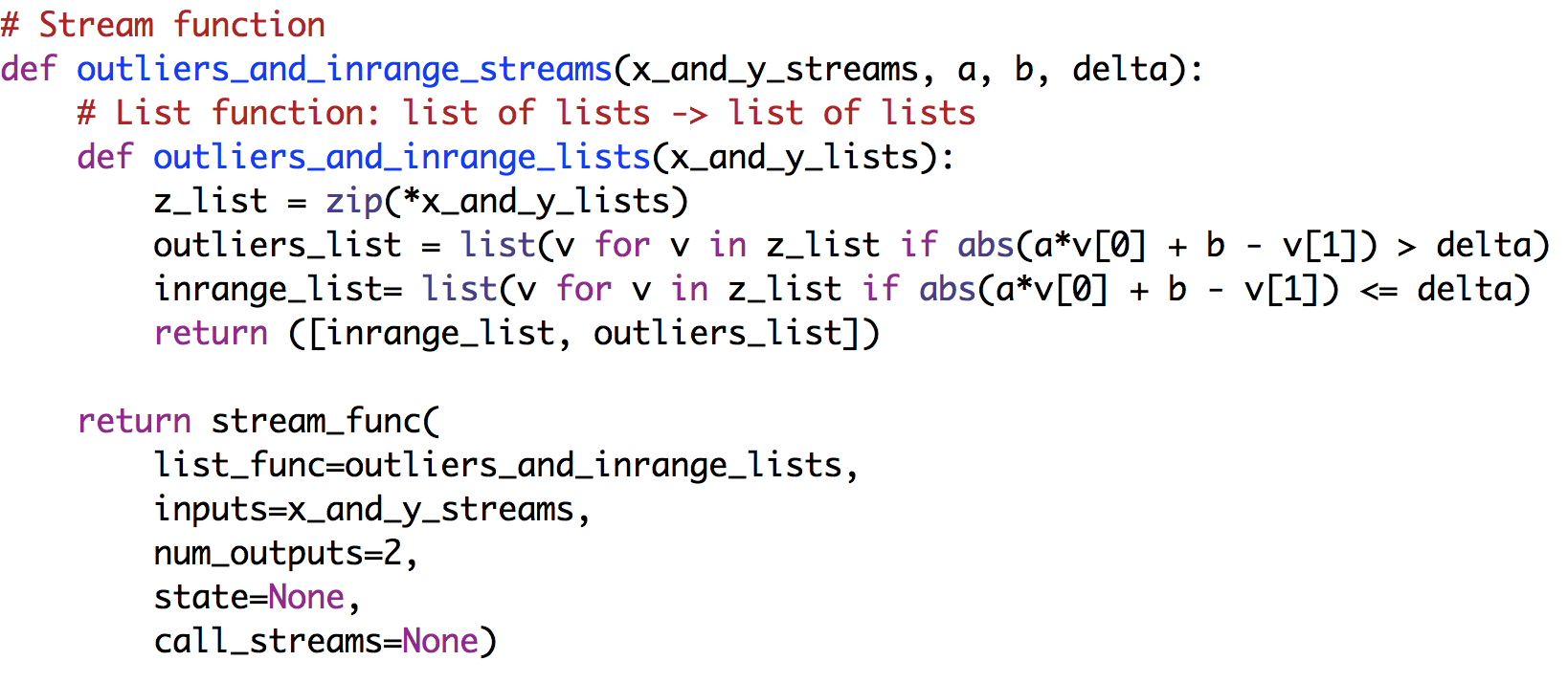


###### Many Input Streams and Many Output Streams

A many-to-many function has more than one input stream and more than one output stream.

##### Stateless Many to Many

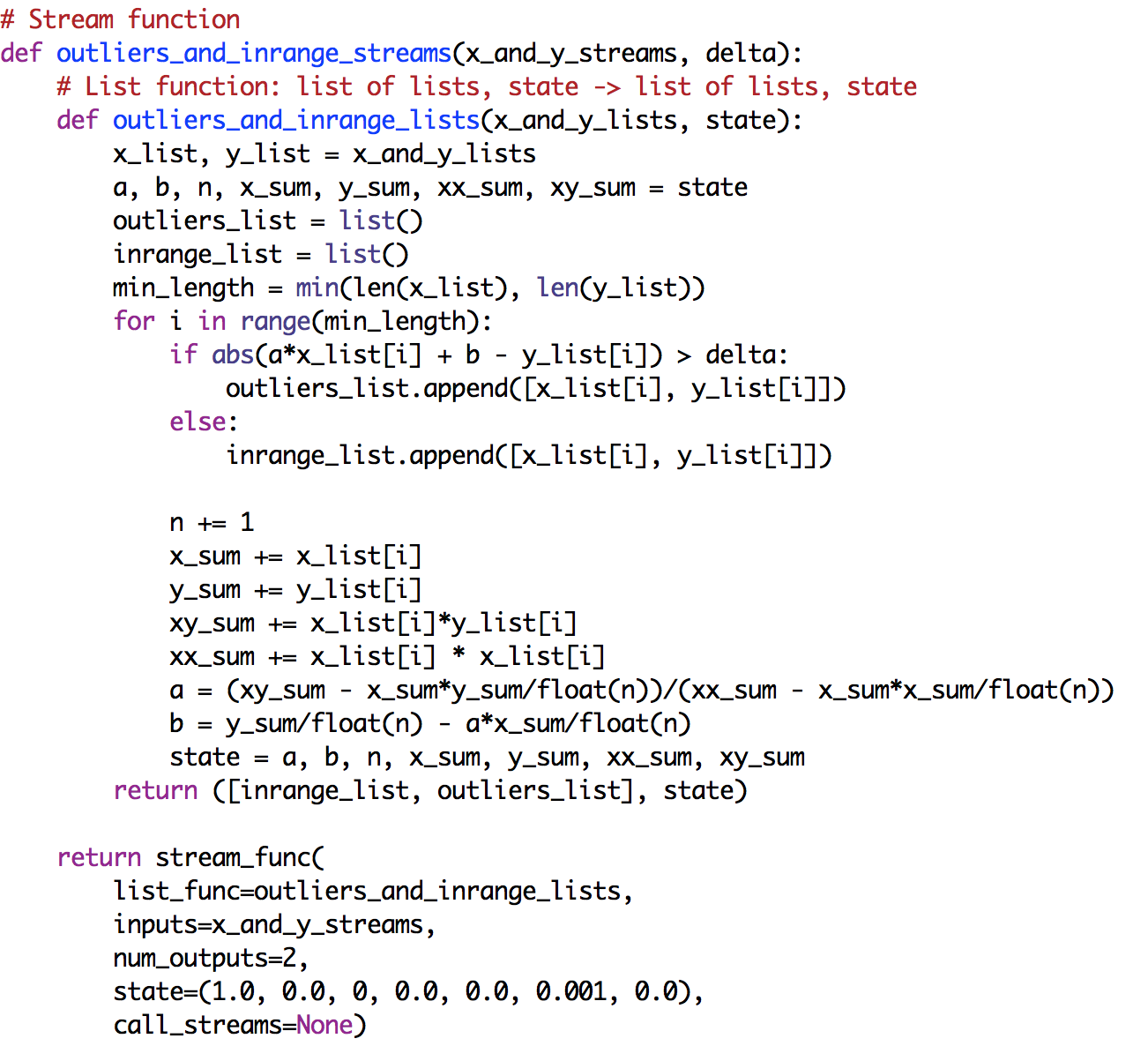
**Problem**: Write a function *separate\_outliers\_and\_inrange\_streams* that takes two streams of numbers as input and produces two output streams. The function has three constant parameters, *a,* *b* and *delta*. One of the input streams is called *x\_stream* and the other *y\_stream*. One of the output streams is called *outlier\_stream* and the other is called *inrange\_stream*. Let *x*[*n*] and *y*[*n*] be the *n*-th elements of *x\_stream* and *y\_stream*, respectively. If *abs*(*a.x*[*n*] + *b*[*n*] – *y*[*n*]) > *delta* then the tuple (*x*[*n*], *y*[*n*]) is placed in *outlier\_strea*m, otherwise the tuple is placed in the *inrange\_stream*.

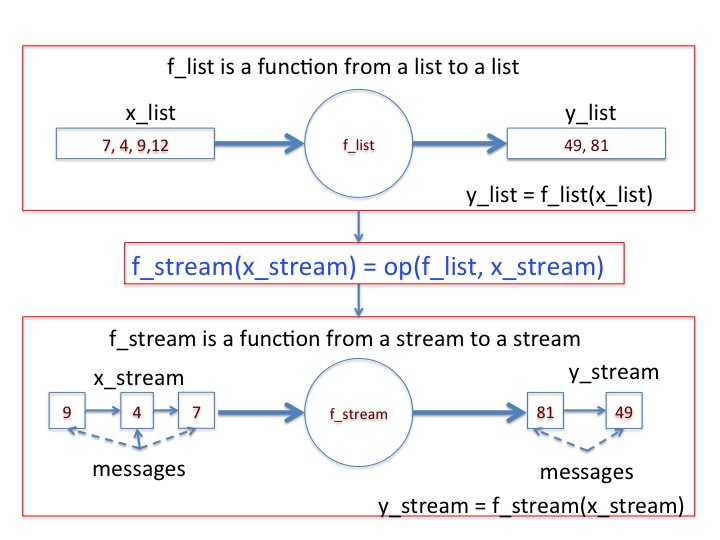


##### Stateful Many to Many

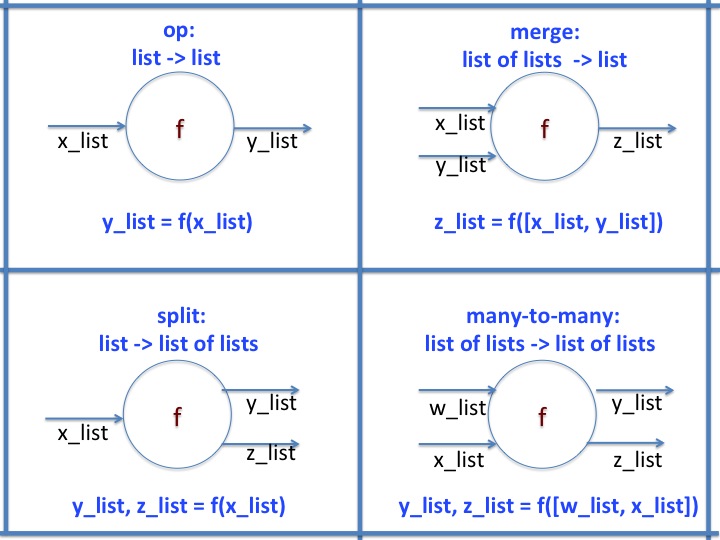
**Problem**: The problem is the same as before except that the regression line, y = a\*x + b is recomputed each time a value appears in both the x and y streams. In this problem, a and b are computed at each step, and therefore they are not input parameters. We compute a using:

and





**Figure: Relationship between functions on lists to functions on streams**



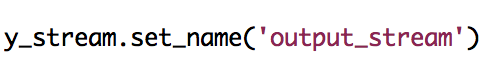
**Actuators and Printing Streams**

A printer is an example of an actuator: an actuator gets input streams and it changes the environment - for example, an actuator may adjust a valve position, change a thermostat setting or print an object. We discuss printing next, and describe other types of actuators later in the book

When a printer receives multiple streams it prints messages in the order in which they are received, and this order may not be the order in which they were generated. Giving streams names is helpful when looking at a printer output, because the output associates a stream name with a message.

##### Giving names to streams

For a stream *s* and a string *r*, *s.set\_name*(*r*) makes string *r* the name of stream *s*. For example, the name ‘output\_stream’ is assigned to the stream in variable *y\_stream* as follows:

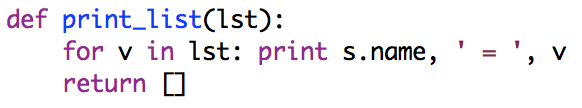


You can also give a name to a stream when it is created. For example, give the name ‘input\_stream’ to *x\_stream* when *x\_stream* is created as follows:

Macintosh HD:Users:kmchandy:Desktop:Screen Shot 2015-02-20 at 2.34.39 PM.png

Programs for actuators generate commands to the actuators; for the case of the printer, the command is ‘print.’ The actuator reads streams but doesn’t (typically) append values to streams. Some actuators may have output streams that indicate the operations that the actuator has completed.

As usual, to specify an actuator we first write a function on lists and then extend the function to operate on streams. The function on lists executes a command on the actuator and returns an empty list. A list function for printing is:



We extend the function to streams in the usual way:

Macintosh HD:Users:kmchandy:Desktop:Screen Shot 2015-02-20 at 2.48.37 PM.png

Several functions for printing streams are found in the *PrintFunctions.py* folder.

Examples of functions on streams, obtained from functions on lists, using *op*, *merge*, *split* and *many\_to\_many*, are in the *ExamplesElementaryOperations* folder. Next, we discuss how to convert operations on lists with state to operations on streams with state.