# Pstreams

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

**Goals**:

1. Convert functions that operate on Python lists, NumPy arrays, or Pandas frames to functions that operate on real-time data streams. The intent is to enable programmers to continue using familiar data structures such as lists, then extend their programs to operate on streams. The goal is separation of concerns: first focus on the logic operating on fixed-size lists and arrays; then extend the logic to streaming systems in which new messages may arrive as time progresses. The package can also be used to program agents that operate on streams. The goal with writing applications that are networks of agents is to use the great deal of prior work on networks of processes.
2. Enable programmers to develop and test programs in a single process and later map functions in the program to components in a distributed system. These components include single-board computers such as the Raspberry Pi and Intel Galileo connected to sensors, cloud services such as the Google Compute and App Engines and Amazon EC2, and intermediate processing elements. The goal here again is separation of concerns: first develop and test the logic without being concerned about parallel execution and later distribute the logic.
3. Apply Python open-source science and data-mining packages such as NumPy, SciPy, Pandas and SciKit-Learn to streaming applications. Python has powerful packages that were not necessarily intended for streaming applications. The number and variety of these packages, their functionality, and the programming communities around them continue to grow.

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

**Streams**

A *stream* is a sequence of values; values may be appended to the tail of the sequence. 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. An example of a stream is a sequence of measurements made by a sensor; as time progresses the sensor may make more measurements and send messages containing measured values along the stream. However, the sensor is unable to retract or modify sent messages.

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.

Streams are processed by *agents*. Agents can read streams and append messages to the ends of streams. Agents extend operations on Python lists, NumPy arrays, and Pandas frames to streams. In this chapter we show, using simple examples, how to use agents to allow functions on lists to operate on streams. At the end of this chapter we give an overview of the entire document.

# 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 consider 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].

## Operations on lists: functions from a list to a list

Given a function f that has a list as input and returns a list, we define a function F that has a stream as input and returns a stream, and operates similarly to function f, as follows:

from Agent import \*

from ListOperators import op

def F(stream\_x): return op(f, x\_stream)

F(x\_stream) is a stream that is updated when messages are appended to x\_stream. If the value of x\_stream is a list, x\_list, at some point in time, then the value of F(x\_stream) will become f(x\_list) later. We do not specify precisely when F(x\_stream) will be updated because streams are updated asynchronously.

**Example of op: operations on lists**

We want to write a function that takes a stream as input and produces a stream as output. The elements of the output stream are squares of the elements of the input stream. First, we write (document and test) a function that takes a list as input and produces a list as output. Then, we use the function on lists to create a function on streams.

Function square\_elems has a single list as its input and returns a list containing the squares of the elements of the input list.

def square\_elems(x\_list):

return [w\*w for w in x\_list]

Then, assigning values to variables:

x\_list = [3, 5]

y\_list = square\_elems(x\_list)

# y\_list = [9, 25]

If x\_list is extended, for example, then y\_list remains unchanged.

x\_list.extend([2, 4])

# x\_list = [3, 5, 2, 4]

# y\_list = [9, 25]

We see next that extending function square\_elems from lists to streams has a different effect; we create an agent that automatically updates its output streams when its input streams change.

Now, let’s extend function square\_elems on lists to a function F on streams.

from Agent import \*

from ListOperators import op

def F(x\_stream):

return op(square\_elems, x\_stream)

Function F operates on its input stream in the same way that function square\_elems operates on its input list; however, F creates an agent that updates the output stream when the input stream is changed whereas the output of f is not automatically changed when its input is changed.

# Create a stream.

x\_stream = Stream()

# Value of x\_stream is now [ ], the empty list.

# Extend function f from lists to streams

y\_stream = F(x\_stream)

# Value of y\_stream is now [ ].

The statement x\_stream = Stream() creates an empty stream, i.e., a stream whose value is the empty list. The statement y\_stream = op(square\_elems, x\_stream) creates a stream called y\_stream, whose initial value is the empty list, and the statement creates an agent that updates y\_stream whenever the value of x\_stream changes.

Next, let us append messages to x\_stream.

x\_stream.extend([3, 5])

# Value of x\_stream = [3, 5]

# Value of y\_stream will become f([3, 5]) = [9, 25]

When the value of x\_stream changes from [ ] to [3, 5], since square\_elems([3, 5]) = [9, 25], the agent that generates y\_stream will append messages with values 9 and 25 (in that order) to the tail of y\_stream. The time at which these messages will be sent is not specified. So, the value of y\_stream will become [9, 25].

Next, let us extend stream x further:

x\_stream.extend([2, 4])

# The value of x\_stream is [3, 5, 2, 4]

# The value of y\_stream will become f([3, 5, 2, 4]) = [9, 25, 4, 16]

The agent that generates y\_stream will put messages in f([2, 4]) = [4, 16] in y\_stream, so the value of y\_stream will become [9, 25, 4, 16]

**Restriction on list function f**

Function f must be prefix closed, i.e.

for all lists x, and for all n where 0 <= n < len(x):

f(x) = f(x[:n]).extend(f(x[n:])

The function must return the same value regardless of whether the list is processed element by element of in groups. An example of a function that we do *not* allow is:

def h(x):

if len(x) < 3: return x

else: return [0]

This is because if x = [1, 2, 3], then h(x[:2]) = [1, 2] and h(x[2:]) = [3], but h(x) = [0]. If h were prefix-closed, the value of h(x) would be [1, 2, 3].

**Functional composition of streams**

If f and g are functions from a list to a list, then we can define a function h that is a composition of g and f as follows:

def h(x\_list): return g(f(x\_list))

Likewise, we can compose functions on streams:

def F(x\_stream): return op(f, x\_stream)

def G(x\_stream): return op(g, x\_stream)

def H(x\_stream): return G(F(x\_stream)

## Merge: Functions from multiple lists to a single list

Given a function f that has a list of lists as input and returns a list, we define a function F that has a list of streams as input and returns a stream, as follows:

from Agent import \*

from ListOperators import merge

def F(list\_of\_streams): return merge(f, list\_of\_streams)

We use *merge* rather than *op* because op is an operator – a function from a list to a list – and here f is a function that merges multiple lists into a single list.

F(list\_of\_streams) is a stream that is updated when messages are appended to any stream in list\_of\_streams. If the value of list\_of\_streams is x at some point in time, then the value of F(list\_of\_streams) will become f(x) at that point or later.

**Example of *merge***

We want to write a function that takes multiple streams as input and produces a single stream as output. The output stream will wait for a value from each stream, sum them, and return the sum before continuing We first write a function f on lists and then use list-function f to create a function F on streams. So, we write a function f that takes multiple lists as input and produces a single list as output where the output list sums corresponding values of the input lists.

def f(list\_of\_lists):

return map(sum, zip(\*list\_of\_lists))

Next we use list-function f to create a function F on streams:

from Agent import \*

from ListOperators import merge

def F(list\_of\_streams): return merge(f, list\_of\_streams)

Next we create streams and use function F.

# Create streams x\_stream and y\_stream. Their initial values are empty lists.

x\_stream = Stream()

y\_stream = Stream()

# Create stream z\_stream and an agent that populates z\_stream.

z\_stream = F([x\_stream, y\_stream])

# Values of x\_stream, y\_stream, and z\_stream are empty lists.

# Since f([[ ], [ ]]) = [ ], the value of z\_stream is [ ]

Next let us append messages to x\_stream.

x\_stream.extend([3, 5])

# Values of x\_stream and y\_stream are [3, 5] and [ ] respectively.

# Since f([[3, 5], [5]]) = [ ], the value of z\_stream is [ ]

Next let us append messages to y\_stream:

y\_stream.extend([2, 4, 5])

# Values of x\_stream and y\_stream are [3, 5] and [2, 4, 5] respectively.

# z\_stream will become f([[3, 5], [2, 4, 5]]) = [5, 9]

Next let us append more messages to x\_stream:

x\_stream.extend([6, 7, 8])

# Values of x\_stream and y\_stream are [3, 5, 6, 7, 8] and [2, 4, 5] respectively.

# Value of z\_stream will become f([[3, 5, 6, 7, 8], [2, 4, 5]]) = [5, 9, 11]

**Restriction on merge function f**

The restriction is an extension to the one given earlier for operations on single lists.

for all lists x, y, z,… and for all n where 0 <= n < min(len(x), len(y), len(z), …)

f(x,y, z,…) = f(x[:n], y[:n], z[n:],.. ).extend(f(x[n:], y[n:], z[n:],..)

This restriction allows agents to process input lists incrementally, as data arrives, since data that arrives later cannot violate earlier output.

## Split: Functions from a single list to multiple lists

Given a function f that has a list as input and returns a list of *two* lists, we define a function F that has a list of streams as input and returns a list of streams, as follows:

from Agent import \*

from ListOperators import split

def F(x\_stream): return split(f, x\_stream)

To split a stream into n streams, where n is not 2, specify the number of outgoing streams as in the following example which splits x\_stream into 3 streams:

def F(x\_stream):

return split(f, x\_stream, num\_out\_streams=3)

We use *split* rather than *op* or *merge* because x\_stream is split into multiple output streams.

F(x\_stream) is a list of streams where streams in the list are updated when messages are appended to the stream x\_stream. If the value of x\_stream is x\_list at some point in time then the value of F(x\_stream) will become f(x\_list).

**Example of *split***

We want to write a function that takes a single stream as input and produces two streams as output. The values in the first output stream are the even numbers from the input stream, and the values in the second output stream are the odd numbers. We first write a function f that takes a single list as input and produces two lists as output. The first output list contains the even numbers from the input list. The second output list contains the odd numbers. We then use list-function f to create a function F that operates on streams.

def f(x\_list):

return (filter(lambda n: n % 2 == 0, x\_list), \

filter(lambda n: n % 2 != 0, x\_list))

Now, let’s extend function f from lists to streams

from Agent import \*

from ListOperators import split

def F(x\_stream): return split(f, x\_stream)

Next we create streams and use function F.

# Create a stream x\_stream. Its initial value is the empty list.

x\_stream = Stream()

# Create streams even\_stream and odd\_stream and

# create an agent that populates these streams.

even\_stream, odd\_stream = F(x\_stream)

Next, let us append messages to x\_stream.

x\_stream.extend([0, 3, 5, 4])

# Value of x\_stream = [0, 3, 5, 4]

# The value of even\_stream will become [0, 4]

# The value of odd\_stream will become [3, 5]

Next, let us append more messages to x\_stream.

x\_stream.extend([0, 4, 1, 6])

# Value of x\_stream = [0, 3, 5, 4, 0, 4, 1, 6]

# The value of even\_stream will become [0, 4, 0, 4, 6]

# The value of odd\_stream will become [3, 5, 1]

**Example: *split* when the number of output streams is not 2**

Consider the following function from a list to a tuple of three lists:

def f(x\_list):

return (filter(lambda n: n%2 == 0, x\_list), \

filter(lambda n: n%3 == 0, x\_list), \

filter(lambda n: n%5 == 0, x\_list))

Now, let’s extend function f from lists to streams

def F(x\_stream): return split(f, x\_stream, 3)

You can now use function F in the usual way, as for example:

x\_stream = Stream()

twos\_stream, threes\_stream, fives\_stream = F(x\_stream)

The restriction on list functions used for *split* is the same as that for *op*.

## many-to-many: functions from multiple lists to multiple lists

Given a function f that has a list of lists as input and returns a list of lists, we define a function F that has a list of streams as input and returns a list of *two* streams, as follows:

from Agent import \*

from ListOperators import many\_to\_many

def F(list\_of\_streams): return many\_to\_many(f, list\_of\_streams)

As with *split*, specify the number of output streams (num\_output\_streams) if f returns a list of n lists where n is not 2. For example if f returns a list of 3 lists:

many\_to\_many(f, list\_of\_streams, num\_out\_streams=3)

We use *many\_to\_many* rather than *op*, *merge*, or *split* because here f is a function from a list of lists to a list of lists.

F(list\_of\_streams) is a list of streams where streams in this list are updated when messages are appended to any stream in list\_of\_streams. If the value of list\_of\_streams is x at some point in time then the value of F(list\_of\_streams) will become f(x).

**Example of many\_to\_many**

We want to write a function that takes multiple streams as input and produces two output streams. The values in the first output stream are the even values in the sum of the values of the input stream, and the second output stream contains the odd values. We first write a function f that takes multiple lists as input and produces two output lists where the values in the first output list are the even values in the sum of the values of the input list, and the second output list contains the odd values. Then we use list-function f to create a function F on streams.

def f(list\_of\_lists):

r = map(sum, zip(\*list\_of\_lists))

return (filter(lambda v: v % 2 == 0, r),\

filter(lambda v: v % 2 != 0, r))

Next, we use list-function f to create a function F on streams

from Agent import \*

from ListOperators import many\_to\_many

def F(list\_of\_streams): return many\_to\_many(f, list\_of\_streams)

Then defining the following variables

x\_list = [5, 11]

y\_list = [2, 4, 5]

evens\_list, odds\_list = f([x\_list, y\_list])

# evens\_list = [ ]

# odds\_list = [7, 15]

Next, let us extend the function from lists to streams.

x\_stream = Stream()

y\_stream = Stream()

even\_stream, odd\_stream = F([x\_stream, y\_stream])

Next, let us append messages to x\_stream and y\_stream

x\_stream.extend([5, 11])

y\_stream.extend([2, 4, 5])

# x\_stream = [5, 11]

# y\_stream = [2, 4, 5]

# even\_stream = [ ]

# odd\_stream = [7, 15]

Next, let us append more messages to the input streams:

x\_stream.extend([7, 12, 6])

y\_stream.extend([4, 5])

# x\_stream = [5, 11, 7, 12 6]

# y\_stream = [2, 4, 5, 4, 5]

# even\_stream = [ 12, 16]

# odd\_stream = [7, 15, 11]

**Using *many\_to\_many* or *split* or *merge* or *op***

You can use *many\_to\_many* instead of the other functions as in the following example which uses many\_to\_many instead of *op*.

# f is a function from a list of lists to a list of lists.

def f(list\_of\_lists):

return [[v\*v for v in list\_of\_lists[0]]]

x\_stream = Stream()

# many\_to\_many returns a list of streams (in this case a single stream).

# Extract y\_stream from the list of streams.

y\_stream = many\_to\_many(f, [x], 1)[0]

For convenience, use op, split and merge for one-to-one, one-to-many, and many-to-one functions, respectively

The restriction on many\_to\_many is the same as for merge.

**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.

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. The name ‘output\_stream’ is assigned to the stream in variable y\_stream:

y\_stream.set\_name(‘output\_stream’)

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:

x\_stream = Stream(‘input\_stream’)

Programs for actuators generate commands to the actuators; for the case of the printer, the command is ‘print.’ The actuator doesn’t generate new streams. As usual, we first write a function on lists and then extend the function to operate on streams. The function on lists executes a command to the actuator - print a value - and returns an empty list.

Next we extend the function from lists to streams. We make the list function f local to the stream function F unless f is used in elsewhere, as in:

from Agent import \*

from ListOperators import op

def print\_stream(x\_stream):

# Function on lists that sends commands to printer

def print\_list(x\_list):

for v in x\_list: print s.name, ' = ', v

return [ ]

# Create agent that sends commands when messages are

# appended to x\_stream

return op(print\_list, x\_stream)

Execution of the statement print\_stream(y) now causes messages in stream y to be printed.

Try the programs in the **Examples/ExamplesElementaryOperations** folder. These examples deal with list functions without state, such as the ones discussed above, and functions with state, which are discussed next.

List Operations with State

A *state* is a record of some aspect of earlier computations. For example, suppose we want a function from a stream to a stream, where each message of the output stream is the cumulative sum of the messages in the input stream. If the value of the input stream is [2, 4, -3, - 3, 5] then the output stream would become [2, 6, 3, 0, 5]. In this example, the state at every point in the computation is the sum of the messages received so far.

We first write a function f which has a list and a state as input and returns a list and state as output. The state is the cumulative sum of the values in the list. We then use the list-function f to create a function F on streams.

def f(x\_list, cumulative):

result\_list = []

for v in x\_list:

cumulative += v

result\_list.append(cumulative)

return (result\_list, cumulative)

We extend the function from lists to streams, and we specify the initial value, 0, of the state.

def F(x\_stream): return op(f, x\_stream, state=0)

We can then use function F as in:

x\_stream = Stream()

y\_stream = F(x\_stream)

# Appending messages to x\_stream, we get:

x\_stream.extend([2, 4, -3])

# Value of x\_stream is [2, 4, -3]

# y\_stream becomes [2, 6, 3]

# Appending more messages to x\_stream, we get:

x\_stream.extend([-3, 5])

# Value of x\_stream is [2, 4, -3, -3, 5]

# y\_stream becomes [2, 6, 3, 0, 5]

The functions *merge*, *split* and *many-to-many* with state are handled in exactly the same way; examples are provided here for completeness.

**An example of *merge* with state**

We want to merge two streams, x\_stream and y\_stream, into a single stream, z\_stream, where the n-th value of z\_stream is the sum of the first n values of x\_stream and the n-th value of y\_stream. We first write a function on lists. This function has two lists, x\_list and y\_list, and a state as input, and returns a list and a new state as output. The state is the cumulative of the values in x\_list.

def f(two\_lists, state):

result\_list = []

for j in range(min(len(two\_lists[0]), len(two\_lists[1])):

result\_list.append(state+two\_lists[0][j]+two\_lists[1][j])

state += two\_lists[0][j]

return (result\_list, state)

We extend the function from lists to streams, and we specify the initial value, 0, of the state.

from Agent import \*

from ListOperators import merge

def F(list\_of\_streams):

return merge(f, list\_of\_streams, state=0)

Next let’s create and append messages to streams.

x\_stream= Stream()

y\_stream= Stream()

z\_stream = F([x\_stream, y\_stream])

# At this point, the values of x\_stream, y\_stream, z\_stream are empty lists

# Append values to x\_stream and y\_stream

x\_stream.extend([5, 11])

y\_stream.extend([2, 4, 5])

# Value of x\_stream is [5, 11]

# Value of y\_stream is [2, 4, 5]

# z\_stream will become [7, 20]

# Append more values to x\_stream and y\_stream

x\_stream.extend([15, 20])

y\_stream.extend([6, 7])

# Value of x\_stream is [5, 11, 15, 20]

# Value of y\_stream is [2, 4, 5, 6, 7]

# z\_stream will become [7, 20, 36, 57]

**An example of *split* with state**

We want to write a function that takes a single stream as input and produces two streams as output. The values of the input stream are fed into one of the output streams until the next even value appears in the input stream. The appearance of an even number in the input stream is a signal to switch the input stream to the other output stream. We first write a function that takes a single list as input and produces two lists as output. Then, we apply the function to streams.

def f(x\_list, state):

y\_list = []

z\_list = []

for j in range(len(x\_list)):

if (x\_list[j]+state) % 2 == 0:

state = 1

y\_list.append(x\_list[j])

else:

state = 0

z\_list.append(x\_list[j])

return ([y\_list, z\_list], state)

Now, let’s extend function f from lists to streams. The initial state can be 0 or 1 depending on whether the input stream is initially directed to the first or the second output stream. Here we set the initial state to 0.

from Agent import \*

from ListOperators import split

def F(x\_stream): return split(f, x\_stream, state=0)

Next let’s create and append messages to streams.

x\_stream= Stream()

y\_stream, z\_stream = F(x\_stream)

# Append messages to x\_stream

x\_stream.extend([4, 5, 10, 11])

# Value of x\_stream is [4, 5, 10, 11]

# Value of y\_stream will become [4, 5]

# Value of z\_stream will become [10]

# Append more messages to x\_stream

x\_stream.extend([13, 16, 9])

# Value of x\_stream is [4, 5, 10, 11, 13, 16, 9]

# Value of y\_stream will become [4, 5, 16, 9]

# Value of z\_stream will become [10, 11, 13]

**An example of many\_to\_many with state**

We want to write a function that has a list of multiple streams as input and a list of two streams as output. The first output stream contains the sums of corresponding values across all the input streams when the sums are even; the second output stream contains sums that are odd.

As usual, we first write a function on lists and extend it to streams. The function on lists has a list of lists and a state as input, and a list of two lists and a state as output. In the program below the list function is local to the stream function.

# f\_list is a function from list of lists and state to a list of lists and state.

def f\_list(list\_of\_lists, state):

sum\_list = map(sum, zip(\*list\_of\_lists))

if not sum\_list:

return ([[], []], state)

cumulative\_list = [0] \* len(sum\_list)

cumulative\_list[0] = sum\_list[0] + state

for i in range(1, len(sum\_list)):

cumulative\_list[i] = cumulative\_list[i-1] + sum\_list[i]

state = cumulative\_list[-1] if cumulative\_list else 0

first\_list = filter(lambda v: v % 2 == 0, cumulative\_list)

second\_list = filter(lambda v: v % 2 != 0, cumulative\_list)

return ([first\_list, second\_list], state)

# f\_stream is a function from a list of streams to a list of streams.

def f\_stream(list\_of\_streams):

# Extend the function f\_list to apply to streams.

return many\_to\_many(f\_list, list\_of\_streams, state=0)

Control Signals

You may want a function to update its output streams periodically, say every second, rather than updating its output whenever its input changes. Or, you may want a function to update its output only when its input has changed by a sufficient amount. In this section we show how to control the times at which a function updates its output; additional control signals are described later.

*The default is that the agent updates its output whenever its input changes.* The programs given above use the default*.*

The idea in separating control from the function code is to enable you to first develop a program, focusing on the logic, without being concerned about control or too concerned about performance or concurrency. After you have developed a satisfactory program without controls, you can then introduce control signals, and also distribute your program across multiple devices.

Recall that given a function f from a list to a list, you can create a function F from a stream to a stream as follow:

def F(x\_stream): return op(f, x\_stream)

If you wanted the function to execute only when a message appears in a stream in list\_of\_trigger\_streams, you define F as:

def F(x\_stream, list\_of\_trigger\_streams):

return op(f, x\_stream, call\_streams=list\_of\_trigger\_streams)

**Example**

def f(x\_list): return([v\*v for v in x\_list])

# F is a function from a stream to a stream, where F is executed only

# when a message appears in a stream in list\_of\_trigger\_streams

def F(x\_stream, list\_of\_trigger\_streams):

return op(f, x\_stream, call\_streams=list\_of\_trigger\_streams)

x\_stream = Stream('x')

trigger\_stream = Stream('trigger')

list\_of\_trigger\_streams = [trigger\_stream]

y\_stream = F(x\_stream, list\_of\_trigger\_streams)

x\_stream.extend(range(4))

The value of x\_stream is [0, 1, 2, 3], but function F has not been triggered yet, and so y\_stream is still empty. Moreover, y\_stream will remain empty until a message is appended to trigger\_stream. Next, append any message to trigger\_stream; the content of the message is immaterial.

trigger\_stream.extend([1])

Now F has been triggered, and so y\_stream will become [0, 1, 4, 9].

Triggering function operation based on time is helpful in many applications. We will discuss programming patterns for trigger streams in later sections.