INTRODUCTION

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 streams. The intent is to enable programmers to continue using familiar data structures such as lists, and then have their programs extended to operate on data 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 too, 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. The goal is to exploit the knowledge and code base for non-streaming applications to streaming.

This document only describes a Python streaming package; the document is not an exposition on stream processing, distributed computing, or Python. This introductory chapter does not discuss performance issues, incremental computation on streams, distributed computing of streams, or science and data-mining packages. In terms of the three goals listed above, this chapter only deals with the first part of the first goal: how to convert functions that operate on Python lists to 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 the 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.

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 extend Python 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(x_stream): 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 where the elements of the output stream are squares of the elements of the input stream. We first write (document and test) a function takes a list as input and produces a list as output where the elements of the output list are squares of the elements of the input list. Then, we use the function on lists to create a function on streams.

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

```
def f(x_list): return [w*w for w in x_list]
```

Then, assigning values to variables:

```
x_list = [3, 5]

y_list = f(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 f from lists to streams has a different effect; we create an agent that always automatically updates its output streams when its input streams change.

Now, let's extend function f on lists to a function F on streams.

```
from Agent import *

from ListOperators import op

def F(x_stream): return op(f, x_stream)
```

Function F operates on its input stream in the same way that function f 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(f, 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 f([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] whereas 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 – whereas 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 where the output stream sums corresponding values of the input streams. 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 on the first output stream has the even numbers in the input stream and the second output stream has the odd numbers. We first write a function f that takes a single list as input and produces two lists as output. The values on the first output list has the even numbers in the input list and the second output list has 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_stream is a function from a list of streams to a list of streams.
def f_stream(list_of_streams):
    # 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)
# Extend the function f_list to apply to streams.
  return many_to_many(f_list, list_of_streams, state=0)
```

Control Signals