datastreams Documentation

Release 0.1

Stuart Owen

July 22, 2015

Contents

Efficient, concise stream data processing.

```
>>> from datastreams import DataStream

>>> DataStream("Hello, gorgeous.")\
...     .filter(lambda char: char.isalpha())\
...     .map_method('lower')\
...     .count_frequency().to_list()
[('e', 2), ('g', 2), ('h', 1), ('l', 2), ('o', 3), ('s', 1), ('r', 1), ('u', 1)]
```

CHAPTER 1

Overview

1.1 Pipelining

DataStreams are perfect for pipelined feature calculation:

```
def calc_name(user):
    user.first_name = user.name.split(' ')[0] if user.name else ''
    return user

def calc_age(user):
   user.age = datetime.now() - user.birthday
   return user

DataStream(users)\
    .map(calc_name)\
    .map(calc_age)\
    .for_each(User.save)\
    .execute() # <- Nothing happens till here
```

Theses calculations are efficient because streams are generators by default, and no memory is wasted on unnecessary intermediary collections.

This is even better when certain calculation steps are long, and must be wrapped in their own functions. Of course, for brevity, you can use *set*:

```
DataStream(users)\
    .set('first_name', lambda user: user.name.split(' ')[0] if user.name else '')\
    .set('age', lambda user: datetime.now() - user.birthday)\
    .for_each(User.save)\
    .execute()
```

1.2 Joins

You can join `DataStream` - even streams of objects!

```
userstream = DataStream(users)
transactstream = DataStream(transactions)

user_spend = userstream.join('inner', 'user_id', transactstream)\
    .group_by('user_id')\
    .map(lambda usertrans: (usertrans[0], sum(tran.price for tran in usertrans[1])))\
```

```
    .to_dict()

# {'47328129': 240.95, '48190234': 40.73, ...}
```

1.3 `where` Clause

Chained `filter` s are a bit tiresome. `where` lets you perform simple filtering using more accessible language:

```
DataStream(users)\
    .where('age').gteq(18)\
    .where('age').lt(35)\
    .where('segment').is_in(target_segments)\
    .for_each(do_something).execute()
```

Instead of:

```
DataStream(users)\
    .filter(lambda user: user.age >= 18)\
    .filter(lambda user: user.age < 35)\
    .filter(lambda user: user.segment in target_segments)\
    .for_each(do_something).execute()
```

I bet you got tired just *reading* that many lambdas!

1.4 Apache Spark Integration

Integrating with Apache Spark is easy - just use `RddStream` instead of `DataStream` or `DataSet`, and pass it an RDD. The rest of the API is the same!

```
RddStream(myrdd)\
    .where('age').gteq(18)\
    .where('age').lt(35)\
    .where('segment').is_in(target_segments)\
    .for_each(do_something).execute()
```

CHAPTER 2

DataStream API

class datastreams.**DataStream**(*source*, *transform=<function <lambda>>*, *predicate=<function <lambda>>*)

Foundation for the package - *DataStream* allows you to chain map/filter/reduce/etc style operations together:

```
>>> stream = DataStream(range(10))
>>> stream.filter(lambda n: n % 2 == 0).map(lambda n: n*5).to_list()
... [0, 10, 20, 30, 40]
```

DataStreams are evaluated lazily (using generators), providing memory efficiency and speed. Using *collect()* produces a *DataSet*, which evalutes the whole stream and caches the result.

batch(*batch_size*)

Batches rows of a stream in a given chunk size

```
>>> DataStream(range(10)).batch(2).to_list()
... [[0, 1], [2, 3], [4, 5], [6, 7], [8, 9]]
```

Parameters **batch_size** (*int*) – size of each batch

Return type *DataStream*

chain()

Chains together iterables, flattening them

```
>>> DataStream(['this', 2, None]).map(dir).chain().to_list()
... ['__add__', '__class__', '__contains__', '__delattr__', ...]
```

Return type *DataStream*

collect()

Collects the stream into a *DataSet*

```
>>> DataStream(range(5)).map(lambda n: n * 5).collect()
... DataSet([0, 5, 10, 15, 20])
```

Return type *DataSet*

collect_as(*constructor*)

Collects using a constructor

```
>>> DataStream(range(5)).collect_as(str)
... DataSet(['0', '1', '2', '3', '4'])
```

Parameters **constructor** – class or constructor function

Return type *DataSet*

concat()
Alias for *chain()*

```
>>> DataStream(['this', 2, None]).map(dir).concat().to_list()
... ['__add__', '__class__', '__contains__', '__delattr__', ...]
```

Return type *DataStream*

concat_map(*function*)
map() a function over the stream, then concat it

```
>>> DataStream(['this', 2, None]).concat_map(dir).to_list()
... ['__add__', '__class__', '__contains__', '__delattr__', ...]
```

Parameters **function** (*function*) – function to apply

Return type *DataStream*

count()
Counts the number of rows in this stream. This will exhaust a stream!

```
>>> DataStream(range(5)).count()
... 5
```

Return type int

count_frequency()
Counts frequency of each row in the stream

```
>>> DataStream(['a', 'a', 'b', 'c']).count_frequency()
... DataSet([('a', 2), ('b', 1), ('c', 1)])
```

Return type *DataSet*

dedupe(*key_fn=<function <lambda>>*)
Removes duplicates from a stream, returning only unique values.

```
>>> DataStream('aaaabcccddd').dedupe().to_list()
... ['a', 'b', 'c', 'd']
```

Parameters **key_fn** (*function*) – function returning a hashable value used to determine uniqueness

Returns DataStream

delete(*attr*)
Deletes the named attribute for each row in the stream

drop(*n*)
Drops n rows from the stream

```
>>> DataStream(range(10)).drop(5).to_list()
... [5, 6, 7, 8, 9]
```

Parameters **n** (*int*) – number of rows to be dropped

Return type *DataStream*

execute()
Evaluates the stream (nothing happens until a stream is evaluted)

```
>>> from pprint import pprint
>>> DataStream(range(3)).for_each(pprint).execute()
... 0
... 1
... 2
... <datastreams.DataStream at 0x7f6995ea4790>
```

filter(*filter_fn*)
Filters a stream using the passed in predicate function.

```
>>> DataStream(range(10)).filter(lambda n: n % 2 == 0).to_list()
... [0, 2, 4, 6, 8]
```

Parameters **filter_fn** (*function*) – only passes values for which filter_fn returns `True`

Return type *DataStream*

filter_method(*method*, **args*, ***kwargs*)
Filters using a method of the stream row using passed in args/kwargs

```
>>> DataStream(['hi', 'h1', 'ho']).filter_method('isalpha').to_list()
... ['hi', 'ho']
```

Parameters **method** (*str*) – name of method to be called

Return type *DataStream*

filters(*filter_fns*)
Apply a list of filter functions

```
>>> evens_less_than_six = [lambda n: n < 6, lambda n: n % 2 == 0]
>>> DataStream(range(10)).filters(evens_less_than_six).to_list()
... [0, 2, 4]
```

Parameters **filter_fns** (*list[function]*) – list of filter functions

Return type *DataStream*

for_each(*function*)
Calls a function for each row in the stream, but passes the row value through

```
>>> from pprint import pprint
>>> DataStream(range(3)).for_each(pprint).execute()
... 0
... 1
... 2
... <datastreams.DataStream at 0x7f6995ea4790>
```

Parameters **function** (*function*) – function to call on each row

Return type *DataStream*

classmethod from_csv(*path*, *headers=None*, *constructor=<class 'datastreams.datastreams.Datum'>*)

Stream rows from a csv file

```
>>> DataStream.from_csv('payments.csv').to_list()
... [Datum({'name': 'joe', 'charge': 174.93}), Datum({'name': 'sally', 'charge': 198.05}), .
```

Parameters

- **path** (*str*) – path to csv to be streamed
- **headers** (*list[str]*) – manual names for headers - if present, first row is pulled in as data, if `None`, first row is used as headers
- **constructor** – class or function to construct for each row

Return type *DataStream*

classmethod from_file(*path*)

Stream lines from a file

```
>>> DataStream.from_file('hamlet.txt').concat_map(str.split).take(7)
... ['The', 'Tragedy', 'of', 'Hamlet,', 'Prince', 'of', 'Denmark']
```

Parameters **path** (*str*) – path to file to be streamed

Return type *DataStream*

classmethod from_stdin()

Stream rows from stdin

Return type *DataStream*

get(*name*, *default=None*)

Gets the named attribute of each row in the stream

```
>>> Person = namedtuple('Person', ['name', 'year_born'])
>>> DataStream([Person('amy', 1987), Person('brad', 1980)]).get('year_born').to_list()
... [1987, 1980]
```

Parameters

- **attr** (*str*) – attribute name
- **default** – default value to use if attr name not found in row

Return type *DataStream*

group_by(*key*)

Groups a stream by key, returning a *DataSet* of `(K, tuple(V))`

```
>>> stream = DataStream(range(3) * 3)
>>> stream.group_by('real').to_dict()
... {0: (0, 0, 0), 1: (1, 1, 1), 2: (2, 2, 2)}
```

Parameters **key** (*str*) – attribute name to group by

Return type *DataSet*

group_by_fn(*key_fn*)
Groups a stream by function, returning a *DataSet* of `(K, tuple(V))`

```
>>> stream = DataStream(['hi', 'hey', 'yo', 'sup'])
>>> stream.group_by_fn(lambda w: len(w)).to_dict()
... {2: ('hi', 'yo'), 3: ('hey', 'sup')}
```

Parameters **key_fn** (*function*) – key function returning hashable value to group by

Return type *DataSet*

inner_join(*key*, *right*)
Returns a dataset joined using keys in both dataset only

Parameters

- **right** (DataStream) – *DataStream* to be joined with
- **key** (*str*) – attribute name to join on

Return type *DataSet*

inner_join_by(*left_key_fn*, *right_key_fn*, *right*)
Returns a dataset joined using key functions to evaluate equality

Parameters

- **left_key_fn** (*function*) – key function that produces a hashable value from left stream
- **right_key_fn** (*function*) – key function that produces a hashable value from right stream
- **right** (DataStream) – *DataStream* to be joined with

Return type *DataSet*

join(*how*, *key*, *right*)
Returns a dataset joined using keys from right dataset only

Parameters

- **how** (*str*) – `left`, `right`, `outer`, or `inner`
- **right** (DataStream) – *DataStream* to be joined with
- **key** (*str*) – attribute name to join on

Return type *DataSet*

join_by(*how*, *left_key_fn*, *right_key_fn*, *right*)
Uses two key functions perform a join. Key functions should produce hashable types to be used to compare/index dicts.

Parameters

- **how** (*str*) – `left`, `right`, `outer`, or `inner`
- **right** (DataStream) – *DataStream* to be joined with
- **left_key_fn** (*function*) – key function that produces a hashable value from left stream
- **right_key_fn** (*function*) – key function that produces a hashable value from right stream

Return type *DataSet*

left_join (*key*, *right*)
Returns a dataset joined using keys from right dataset only

Parameters

- **right** (DataStream) – *DataStream* to be joined with
- **key** (*str*) – attribute name to join on

Return type *DataSet*

left_join_by (*left_key_fn*, *right_key_fn*, *right*)
Returns a dataset joined using key functions to evaluate equality

Parameters

- **how** (*str*) – `left`, `right`, `outer`, or `inner`
- **left_key_fn** (*function*) – key function that produces a hashable value from left stream
- **right_key_fn** (*function*) – key function that produces a hashable value from right stream
- **right** (DataStream) – *DataStream* to be joined with

Return type *DataSet*

map (*function*)
Apply a function to each row in this stream

```
>>> DataStream(range(5)).map(lambda n: n * 5).to_list()
... [0, 5, 10, 15, 20]
```

Parameters **function** (*function*) – function to apply

Return type *DataStream*

map_method (*method*, **args*, ***kwargs*)
Call named method of each row using supplied args/kwargs

```
>>> DataStream(['hi', 'hey', 'yo']).map_method('upper').to_list()
... ['HI', 'HEY', 'YO']
```

Parameters **method** (*str*) – name of method to be called

Return type *DataStream*

outer_join (*key*, *right*)
Returns a dataset joined using keys in either datasets

Parameters

- **right** (DataStream) – *DataStream* to be joined with
- **key** (*str*) – attribute name to join on

Return type *DataSet*

outer_join_by (*left_key_fn*, *right_key_fn*, *right*)
Returns a dataset joined using key functions to evaluate equality

Parameters

- **left_key_fn** (*function*) – key function that produces a hashable value from left stream

- **right_key_fn** (*function*) – key function that produces a hashable value from right stream
- **right** (DataStream) – *DataStream* to be joined with

Return type *DataSet*

pick_attrs(*attr_names*)
Picks attributes from each row in a stream. This is helpful for limiting row attrs to only those you want to save in a database, etc.

```
>>> Person = namedtuple('Person', ['name', 'year_born'])
>>> DataStream([Person('amy', 1987), Person('brad', 1980)]).pick_attrs(['year_born']).to_lis
... [Datum({'year_born': 1987}), Datum({'year_born': 1980})]
```

Parameters **attr_names** (*list[str]*) – list of attribute names to keep

Return type *DataStream*

pipe_to_stdout()
Pipes stream to stdout using `sys.stdout.write`

reduce(*function*, *initial=None*)
Applying a reducing function to rows in a stream

Parameters

- **function** (*function*) – reducing function, with parameters `last_iteration`, `next_value`
- **initial** – initial value for reduce, if None, takes the first element of this stream as initial

reduce_to_dataset(*function*, *initial=None*)
Applies a reducer over this stream, returning a *DataSet* of the results

Parameters

- **function** – reducing function, with parameters `last_iteration`, `next_value`
- **initial** – initial value for reduce, if None, takes the first element of this stream as initial

Return type *DataSet*

right_join(*key*, *right*)
Returns a dataset joined using keys in right dataset only

Parameters

- **right** (DataStream) – *DataStream* to be joined with
- **key** (*str*) – attribute name to join on

Return type *DataSet*

right_join_by(*left_key_fn*, *right_key_fn*, *right*)
Returns a dataset joined using key functions to evaluate equality

Parameters

- **left_key_fn** (*function*) – key function that produces a hashable value from left stream
- **right_key_fn** (*function*) – key function that produces a hashable value from right stream
- **right** (DataStream) – *DataStream* to be joined with

Return type *DataSet*

sample(*probability*, *n*)
Sample N rows with a given probability of choosing a given row

```
>>> DataStream(range(100)).sample(0.1, 5)
...
```

Parameters

- **probability** (*float*) – probability that a sample is chosen
- **n** (*int*) – population size to sample

Return type *DataStream*

set(*name*, *transfer_func=None*, *value=None*)
Sets the named attribute of each row in the stream using the supplied function

Parameters

- **name** – attribute name
- **transfer_func** – function that takes the row and returns the value to be stored at the named attribute

Return type *DataStream*

take(*n*)
Takes n rows from the stream

```
>>> DataStream(range(100000)).take(3).to_list()
... [0, 1, 2]
```

Parameters **n** (*int*) – number of rows to be taken

Return type *DataStream*

take_now(*n*)
Like take, but evaluates immediately and returns a *DataSet*

```
>>> DataStream(range(100000)).take_now(3)
... DataSet([0, 1, 2])
```

Parameters **n** (*int*) – number of rows to be taken

Return type *DataSet*

to_dict()
Converts a stream to a dict

```
>>> stream = DataStream(['hi', 'hey', 'yo', 'sup'])
>>> stream.group_by_fn(lambda w: len(w)).to_dict()
... {2: ('hi', 'yo'), 3: ('hey', 'sup')}
```

Return type dict

to_list()
Converts a stream to a list

```
>>> DataStream(range(5)).map(lambda n: n * 5).to_list()
... [0, 5, 10, 15, 20]
```

Return type list

to_set()
Converts a stream to a *set*

```
>>> DataStream([1, 2, 3, 4, 2, 3]).to_set()
... {1, 2, 3, 4}
```

Return type *set*

where(*name=<class 'datastreams.datastreams.Nothing'>*)
Short hand for common filter functions - `where` selects an attribute to be filtered on, with a condition like `gt` or `contains` following it.

```
>>> Person = namedtuple('Person', ['name', 'year_born'])
>>> DataStream([Person('amy', 1987), Person('brad', 1980)]).where('year_born').gt(1983).to_l
... [Person(name='amy', year_born=1987)]
```

Parameters **name** (*str*) – attribute name to filter on

Return type FilterRadix

window(*length*, *interval*)
Windows the rows of a stream in a given length and interval

```
>>> DataStream(range(5)).window(3, 2).to_list()
... [DataSet([0, 1, 2]), DataSet([2, 3, 4])]
```

Parameters

- **length** (*int*) – length of window
- **interval** (*int*) – distance between windows

Return type *DataStream*

CHAPTER 3

DataSet API

class datastreams.**DataSet**(*source*)

Like a *DataStream*, but with the source cached as a list. Able to perform tasks that require the whole source, like sorting and reversing.

apply(*function*)

Apply a function to the whole dataset

Parameters **function** (*function*) – function to be called on the whole dataset

Return type *DataSet*

call(*function*)

Call a function with the whole dataset, returning the original

```
>>> from pprint import pprint
>>> DataSet([1, 2, 3]).apply(pprint)
... DataSet([1, 2, 3])
... DataSet([1, 2, 3])
```

Parameters **function** (*function*) – function to be called on the whole dataset

Return type *DataSet*

reverse()

Reverses a *DataSet*

```
>>> DataSet(range(5)).reverse()
... DataSet([4, 3, 2, 1, 0])
```

Return type *DataSet*

sort_by(*key_fn*, *descending=True*)

Sort the *DataSet* using the given key function

```
>>> Person = namedtuple('Person', ['name', 'year_born'])
>>> DataSet([Person('amy', 1987), Person('brad', 1980)]).sort_by(lambda p: p.year_born)
... DataSet([Datum({'name': 'amy', 'year_born': 1980}), Datum({'name': 'brad', 'year_born':
```

Parameters

- **key_fn** (*function*) – function used select the key used to sort the dataset
- **descending** (*bool*) – sorts descending if `True`

Return type *DataSet*

to_stream()
Streams from this dataset

Return type *DataStream*

CHAPTER 4

Etc

Check it out on github!

This project is licensed under the MIT license.

d

W