Streams Documentation

Release 0.6b

Sergey Arkhipov

September 30, 2016

Contents

Streams is an easy to use library to allow you to interpret your information as a data flow and process it in this way. It allows you parallel processing of a data flow and you can control it.

Actually Streams is dramatically inspired by Java 8 Stream API. Of course it is not a new beast in the zoo, I used the same approach in several projects before but this pattern goes to mainstream now and it is good to have it in Python too.

Just several examples to help you to feel what is it:

```
    from requests import get
    from operator import itemgetter

    average_price = Stream(urls)                                \ # make a stream from the list of urls
        .map(requests.get, parallel=4)                          \ # do url  fetching in parallel. 4 thr
        .map(lambda response: response.json()["model"])         \ # extract required field from JSON.
        .exclude(lambda model: model["deleted_at"] is None)     \ # we need only active accounts so filt
        .map(itemgetter("price"))                               \ # get a price from the model
        .decimals()                                             \ # convert prices into decimals
        .average()                                              \ # calulate average from the list
```

And not let's check the piece of code which does almost the same.

```
    from concurrent.futures import ThreadPoolExecutor
    from requests import get

    with ThreadPoolExecutor(4) as pool:
        average_price = Decimal("0.00")
        fetched_items = pool.map(requests.get, urls)
        for response in fetched_items:
            model = response.json()["model]
            if model["deleted_at"] is None: continue
            sum_of += Decimal(model["price"])
        average_price /= len(urls)
```

So this is Stream approach. Streams are lazy library and won't do anything if it is not needed. Let's say you have urls as iterator and it contains several billions of URLs you can't fit into the memory (ThreadPoolExecutor creates a list in the memory) or you want to build a pipeline of your data management and manipulate it according to some conditions, checkout Streams, maybe it will help you to create more accurate and maintainable code.

Just suppose Streams as a pipes from your *nix environment but migrated into Python. It also has some cool features you need to know about:

- Laziness,
- Small memory footprint even for massive data sets,
- Automatic and configurable parallelization,
- Smart concurrent pool management.

CHAPTER 1

Terms of content

1.1 Installation

Install with Pip or easy_install.

```
$ pip install streams
```

or

```
$ easy_install streams
```

If you want, you can always download the latest bleeding edge from GitHub:

```
$ git clone git://github.com/9seconds/streams.git
$ cd streams
$ ./setup.py install
```

Streams supports Python 2.6, 2.7, 3.2, 3.3, 3.4 and PyPy. Probably other implementations like Jython or IronPython will work, but I haven't tested them there.

1.2 User Guide

I supposed you've worked with Django and you've been using its ORM a lot. I will try to lead you to the idea of functional streams by example. Actually I did no Django for a while and syntax might be outdated a bit or I may confuse you so you are free to correct me through issue or pull request. Please do it, I appreciate your feedback.

If you didn't work with any ORM just try to follow the idea, I will try to explain what is going on and things that really matter.

1.2.1 What is Stream?

Let's go back to default Django example: libraries and books. Let's assume that we have app up and running and it does some data management from your beloved database. Let's say you want to fetch some recent books.

```
from library.models import Book

books = Book.objects.filter(pub_date__year=2014)
```

Good, isn't it? You have a collection of models called `Book` which possibly presents books in your app. And you want to have only those which were published in 2014. Good, figured out. Let's go further. Let's say you want to be more specific and you want to have only bestsellers. It is ok.

```
from library.models import Book

books = Book.objects.filter(pub_date__year=2014)
bestsellers = books.order_by("-sales_count")[:10]
```

You can do it like this. But why is it better than this approach?

```
from operator import attrgetter
from library.models import Book

books = Book.objects.all()
books = [book for book in books if book.pub_date.year == 2014]
bestsellers = sorted(books, key=attrgetter("sales_count"), reverse=True)
bestsellers = bestsellers[:10]
```

You will get the same result, right? Actually no. Look, on filtering step you fetch all objects from the database and process them all. It is ok if you have a dozen of models in your database but it can be big bottleneck if your data is growing. That's why everyone is trying to move as much filtering as possible into the database. Database knows how to manage your data accurately and what do to in the most efficient way. It will use indexes etc to speedup whole process and you do not need to do full scan everytime. It is best practice to fetch only that data you actually need from the database.

So instead of

```
SELECT * FROM book
```

you do

```
SELECT * FROM book
    WHERE EXTRACT(year FROM "pub_date") == 2014
    ORDER BY sales_count DESC
    LIMIT 10
```

Sounds legit. But let's checkout how it looks like when do you work with ORM. Let's go back to our example:

```
books = Book.objects.filter(pub_date__year=2014)
bestsellers = books.order_by("-sales_count")[:10]
```

or in a short way

```
bestsellers = Book.objects \
    .filter(pub_date__year=2014) \
    .order_by("-sales_count")[:10]
```

You may assume it like a data stream you are processing on every step. First you set initial source of data, this is `Book.objects.all()`. Good. You may consider it as an iterable flow of data and you apply processing functions on that stream, first if filtering, second is sorting, third is slicing. You process the flow, not every objects, this is crucial concept. Everytime after execution of some flow (or `QuerySet`) method you get another instance of the same flow but with your modifications.

You may suppose that Streams library to provide you the same functionality but for any iterable. Of course this is not that efficient as Django ORM which knows the context of database and helps you to execute your queries in the most efficient way.

1.2.2 How to use Streams

Now you got an idea of Streams: to manage data flow itself, not every component. You can build your own toy map/reduce stuff with it if you really need to have it. Our you can just filter and process your data to exclude some Nones etc in parallel or to have some generic way to do it. It is up to you, I'll just show you some examples and if you want to have more information just go to the API documentation

So, for simplicity let's assume that you have giant gzipped CSV, in 10 GB. And you can use only 1GB of your memory so it is not possible to put everything in memory at once. This CSV has 5 columns, `author_id`, `book_name`.

Yeah, books again. Why not?

So your boss asked you to implement function which will read this csvfile and do some optional filtering on it. Also you must fetch the data from predefined external sources, search on prices in different shops (Amazon at least) and write some big XML file with an average price.

I some explanation on the go.

```
    from csv import reader
    from gzip import open as gzopen
    from collections import namedtuple
    try:
        from xml.etree import cElementTree as ET
    except ImportError:
        from xml.etree import ElementTree as ET
    from streams import Stream
    from other_module import shop_prices_fetch, author_fetch, publisher_fetch


    def extract_averages(csv_filename, xml_filename,
                         author_prefix=None, count=None, publisher=None, shops=None,
                         available=None):
        file_handler = gzopen(csv_filename, "r")
        try:
            csv_iterator = reader(file_handler)

            # great, we have CSV iterator right now which will read our
            # file line by line now let's convert it to stream
            stream = Stream(csv_iterator)

            # now let's fetch author names. Since every row looks like a
            # tuple of (key, value) where key is an author_id and value is
            # a book name we can do key_mapping here. And let's do it in
            # parallel it is I/O bound
            stream = stream.key_map(author_fetch, parallel=True)

            # okay, now let's keep only author name here
            stream = stream.key_map(lambda author: author["name"])

            # we have author prefix, right?
            if author_prefix is not None:
                stream = stream.filter(lambda (author, book): author.startswith(author_prefix))

            # let's fetch publisher now. Let's do it in 10 threads
            if publisher is not None:
                stream = stream.map(
                    lambda (author, book): (author, book, publisher_fetch(author, book)),
                    parallel=10
                )
```

```
                stream = stream.filter(lambda item: item[-1] == publisher)
                # we do not have to have publisher now, let's remove it
                stream = stream.map(lambda item: item[:2])

            # good. Let's compose the list of shops here
            stream.map(
                lambda (author, book): (author, book, shop_prices_fetch(author, book, shops))
            )

            # now let's make averages
            stream.map(lambda item: item[:2] + sum(item[3]) / len(item[3]))

            # let's remove unavailable books now.
            if available is not None:
                if available:
                    stream = stream.filter(lambda item: item[-1])
                else:
                    stream = stream.filter(lambda item: not item[-1])

            # ok, great. Now we have only those entries which we are requiring
            # let's compose xml now. Remember whole our data won't fit in memory.
            with open(xml_filename, "w") as xml:
                xml.write("<?xml version='1.0' encoding='UTF-8' standalone='yes'?>\n")
                xml.write("<books>\n")
                for author, book, average in stream:
                    book_element = ET.Element("book")
                    ET.SubElement(book_element, "name").text = unicode(book)
                    ET.SubElement(book_element, "author").text = unicode(author)
                    ET.SubElement(book_element, "average_price").text = unicode(average)
                    xml.write(ET.dumps(book_element) + "\n")
                xml.write("</books>\n")
        finally:
            file_handler.close()
```

That's it. On every step we've manipulated with given stream to direct it in the way we need. We've parallelized where neccessary and actually nothing was executed before we started to iterate the stream. Stream is lazy and it yields one record by one so we haven't swaped.

I guess it is a time to proceed to API documentation. Actually you need to check only Stream class methods documentation, the rest of are utility ones.

1.3 Streams API

This chapter contains documentation on Streams API. As a rule you have to use documentation on Stream class only but if you want you can check internals also.

1.3.1 streams

streams module contains just a Stream class. Basically you want to use only this class and nothing else from the module.

class `streams.`**`Stream`**`(`*iterator*`)`

Stream class provides you with the basic functionality of Streams. Please checkout member documentation to get an examples.

__init__(*iterator*)
Initializes the *Stream*.

Actually it does some smart handling of iterator. If you give it an instance of `dict` or its derivatives (such as `collections.OrderedDict`), it will iterate through it's items (key and values). Otherwise just normal iterator would be used.

Parameters **iterator** (*Iterable*) – Iterator which has to be converted into *Stream*.

__iter__()
To support iteration protocol.

__len__()
To support `len()` function if given iterator supports it.

__reversed__()
To support *reversed()* iterator.

all(*predicate=<type 'bool'>*, ***concurrency_kwargs*)
Check if all elements matching given `predicate` exist in the stream. If `predicate` is not defined, `bool()` is used.

Parameters

- **predicate** (*function*) – Predicate to apply to each element of the *Stream*.
- **concurrency_kwargs** (*dict*) – The same concurrency keywords as for *Stream.map()*.

Returns The result if we have matched elements or not.

```
>>> stream = Stream.range(5)
>>> stream.all(lambda item: item > 100)
... False
```

any(*predicate=<type 'bool'>*, ***concurrency_kwargs*)
Check if any element matching given `predicate` exists in the stream. If `predicate` is not defined, `bool()` is used.

Parameters

- **predicate** (*function*) – Predicate to apply to each element of the *Stream*.
- **concurrency_kwargs** (*dict*) – The same concurrency keywords as for *Stream.map()*.

Returns The result if we have matched elements or not.

```
>>> stream = Stream.range(5)
>>> stream.any(lambda item: item < 100)
... True
```

average()
Calculates the average of elements in the stream.

Returns The average of elements.

```
>>> stream = Stream.range(10000)
>>> stream.average()
... 4999.5
```

chain()
If elements of the stream are iterable, tries to flat that stream.

Returns new processed *Stream* instance.

```
>>> stream = Stream.range(3)
>>> stream = stream.tuplify()
>>> stream = stream.chain()
>>> list(stream)
>>> [0, 0, 1, 1, 2, 2]
```

classmethod concat(**streams*)

Lazily concatenates several stream into one. The same as Java 8 concat.

Parameters **streams** – The *Stream* instances you want to concatenate.

Returns new processed *Stream* instance.

```
>>> stream1 = Stream(range(2))
>>> stream2 = Stream(["2", "3", "4"])
>>> stream3 = Stream([list(), dict()])
>>> concatenated_stream = Stream.concat(stream1, stream2, stream3)
>>> list(concatenated_stream)
... [0, 1, "2", "3", "4", [], {}]
```

count(*element=<object object>*)

Returns the number of elements in the stream. If `element` is set, returns the count of particular element in the stream.

Parameters **element** (*object*) – The element we need to count in the stream

Returns The number of elements of the count of particular element.

decimals()

Tries to convert everything to `decimal.Decimal` and keeps only successful attempts.

Returns new processed *Stream* instance.

```
>>> stream = Stream([1, 2.0, "3", "4.0", None, {}])
>>> stream = stream.longs()
>>> list(stream)
... [Decimal('1'), Decimal('2'), Decimal('3'), Decimal('4.0')]
```

Note: It is not the same as `stream.map(Decimal)` because it removes failed attempts.

Note: It tries to use `cdecimal` module if possible.

distinct()

Removes duplicates from the stream.

Returns new processed *Stream* instance.

Note: All objects in the stream have to be hashable (support `__hash__()`).

Note: Please use it carefully. It returns new *Stream* but will keep every element in your memory.

divisible_by(*number*)

Filters stream for the numbers divisible by the given one.

Parameters **number** (*int*) – Number which every element should be divisible by.

Returns new processed *Stream* instance.

```
>>> stream = Stream.range(6)
>>> stream = stream.divisible_by(2)
>>> list(stream)
... [0, 2, 4]
```

evens()
Filters and keeps only even numbers from the stream.

Returns new processed *Stream* instance.

```
>>> stream = Stream.range(6)
>>> stream = stream.evens()
>>> list(stream)
... [0, 2, 4]
```

exclude(*predicate*, ***concurrency_kwargs*)
Excludes items from *Stream* according to the predicate. You can consider behaviour as the same as for `itertools.ifilterfalse()`.

As *Stream.filter()* it also supports parallelization. Please checkout *Stream.map()* keyword arguments.

Parameters

- **predicate** (*function*) – Predicate for filtering elements of the *Stream*.
- **concurrency_kwargs** (*dict*) – The same concurrency keywords as for *Stream.map()*.

Returns new processed *Stream* instance.

```
>>> stream = Stream.range(6)
>>> stream = stream.exclude(lambda item: item % 2 == 0)
>>> list(stream)
... [1, 3, 5]
```

exclude_nones()
Excludes `None` from the stream.

Returns new processed *Stream* instance.

```
>>> stream = Stream([1, 2, None, 3, None, 4])
>>> stream = stream.exclude_nones()
>>> list(stream)
... [1, 2, 3, 4]
```

filter(*predicate*, ***concurrency_kwargs*)
Does filtering according to the given `predicate` function. Also it supports parallelization (if predicate is pretty heavy function).

You may consider it as equivalent of `itertools.ifilter()` but for stream with a possibility to parallelize this process.

Parameters

- **predicate** (*function*) – Predicate for filtering elements of the *Stream*.
- **concurrency_kwargs** (*dict*) – The same concurrency keywords as for *Stream.map()*.

Returns new processed *Stream* instance.

```
>>> stream = Stream.range(5)
>>> stream = stream.filter(lambda item: item % 2 == 0)
>>> list(stream)
... [0, 2, 4]
```

first
Returns a first element from iterator and does not changes internals.

```
>>> stream = Stream.range(10)
>>> stream.first
... 0
>>> stream.first
... 0
>>> list(stream)
... [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

floats()
Tries to convert everything to `float()` and keeps only successful attempts.

Returns new processed *Stream* instance.

```
>>> stream = Stream([1, 2, "3", "4", None, {}, 5])
>>> stream = stream.floats()
>>> list(stream)
... [1.0, 2.0, 3.0, 4.0, 5.0]
```

Note: It is not the same as `stream.map(float)` because it removes failed attempts.

instances_of(*cls*)
Filters and keeps only instances of the given class.

Parameters **cls** (*class*) – Class for filtering.

Returns new processed *Stream* instance.

```
>>> int_stream = Stream.range(4)
>>> str_stream = Stream.range(4).strings()
>>> result_stream = Stream.concat(int_stream, str_stream)
>>> result_stream = result_stream.instances_of(str)
>>> list(result_stream)
... ['0', '1',  '2', '3']
```

ints()
Tries to convert everything to `int()` and keeps only successful attempts.

Returns new processed *Stream* instance.

```
>>> stream = Stream([1, 2, "3", "4", None, {}, 5])
>>> stream = stream.ints()
>>> list(stream)
... [1, 2, 3, 4, 5]
```

Note: It is not the same as `stream.map(int)` because it removes failed attempts.

classmethod iterate(*function*, *seed_value*)
Returns seed stream. The same as for Java 8 iterate.

Returns an infinite sequential ordered Stream produced by iterative application of a function `f` to an initial element seed, producing a Stream consisting of `seed`, `f(seed)`, `f(f(seed))`, etc.

The first element (position 0) in the Stream will be the provided seed. For `n > 0`, the element at position n, will be the result of applying the function f to the element at position `n - 1`.

Parameters

- **function** (*function*) – The function to apply to the seed.
- **seed_value** (*object*) – The seed value of the function.

Returns new processed *Stream* instance.

```
>>> stream = Stream.iterate(lambda value: value ** 2, 2)
>>> iterator = iter(stream)
>>> next(iterator)
... 2
>>> next(iterator)
... 4
>>> next(iterator)
... 8
```

key_map(*predicate*, ***concurrency_kwargs*)

Maps only key in (key, value) pair. If element is single one, then it would be *Stream.tuplify()* first.

Parameters

- **predicate** (*function*) – Predicate to apply to the key of element in the *Stream*.
- **concurrency_kwargs** (*dict*) – The same concurrency keywords as for *Stream.map()*.

Returns new processed *Stream* instance.

```
>>> stream = Stream.range(4)
>>> stream = stream.tuplify()
>>> stream = stream.key_map(lambda item: item ** 3)
>>> list(stream)
... [(0, 0), (1, 1), (8, 2), (27, 3)]
>>> stream = Stream.range(4)
>>> stream = stream.key_map(lambda item: item ** 3)
>>> list(stream)
... [(0, 0), (1, 1), (8, 2), (27, 3)]
```

keys()

Iterates only keys from the stream (first element from the `tuple`). If element is single then it will be used.

Returns new processed *Stream* instance.

```
>>> stream = Stream.range(5)
>>> stream = stream.key_map(lambda item: item ** 3)
>>> stream = stream.keys()
>>> list(stream)
... [0, 1, 8, 27, 64]
```

largest(*size*)

Returns `size` largest elements from the stream.

Returns new processed *Stream* instance.

```
>>> stream = Stream.range(3000)
>>> stream.largest(5)
```

```
>>> list(stream)
>>> [2999, 2998, 2997, 2996, 2995]
```

limit(*size*)

Limits stream to given `size`.

Parameters **size** (*int*) – The size of new *Stream*.

Returns new processed *Stream* instance.

```
>>> stream = Stream.range(1000)
>>> stream = stream.limit(5)
>>> list(stream)
... [0, 1, 2, 3, 4]
```

longs()

Tries to convert everything to `long()` and keeps only successful attempts.

Returns new processed *Stream* instance.

```
>>> stream = Stream([1, 2, "3", "4", None, {}, 5])
>>> stream = stream.longs()
>>> list(stream)
... [1L, 2L, 3L, 4L, 5L]
```

Note: It is not the same as `stream.map(long)` because it removes failed attempts.

map(*predicate*, ***concurrency_kwargs*)

The corner method of the *Stream* and others are basing on it. It supports parallelization out of box. Actually it works just like `itertools.imap()`.

Parameters

- **predicate** (*function*) – Predicate to map each element of the *Stream*.
- **concurrency_kwargs** (*dict*) – The same concurrency keywords.

Returns new processed *Stream* instance.

Parallelization is configurable by keywords. There is 2 keywords supported: `parallel` and `process`. If you set one keyword to `True` then *Stream* would try to map everything concurrently. If you want more intelligent tuning just set the number of workers you want.

For example, you have a list of URLs to fetch

```
>>> stream = Stream(urls)
```

You can fetch them in parallel

```
>>> stream.map(requests.get, parallel=True)
```

By default, the number of workers is the number of cores on your computer. But if you want to have 64 workers, you are free to do it

```
>>> stream.map(requests.get, parallel=64)
```

The same for `process` which will try to use processes.

```
>>> stream.map(requests.get, process=True)
```

and

```
>>> stream.map(requests.get, process=64)
```

Note: Python multiprocessing has its caveats and pitfalls, please use it carefully (especially `predicate`). Read the documentation on `multiprocessing` and try to google best practices.

Note: If you set both `parallel` and `process` keywords only `parallel` would be used. If you want to disable some type of concurrency just set it to `None`.

```
>>> stream.map(requests.get, parallel=None, process=64)
```

is equal to

```
>>> stream.map(requests.get, process=64)
```

The same for `parallel`

```
>>> stream.map(requests.get, parallel=True, process=None)
```

is equal to

```
>>> stream.map(requests.get, parallel=True)
```

Note: By default no concurrency is used.

median()

Returns median value from the stream.

Returns The median of the stream.

```
>>> stream = Stream.range(10000)
>>> stream.median()
... 5000
```

Note: Please be noticed that all elements from the stream would be fetched in the memory.

nth(*nth_element*)

Returns Nth element from the stream.

Parameters **nth_element** (*int*) – Number of element to return.

Returns Nth element.

```
>>> stream = Stream.range(10000)
>>> stream.average()
... 4999.5
```

Note: Please be noticed that all elements from the stream would be fetched in the memory (except of the case where `nth_element == 1`).

odds()

Filters and keeps only odd numbers from the stream.

Returns new processed *Stream* instance.

```
>>> stream = Stream.range(6)
>>> stream = stream.odds()
>>> list(stream)
... [1, 3, 5]
```

only_falses()
Keeps only those elements where `bool(item) == False`.

Returns new processed *Stream* instance.

```
>>> stream = Stream([1, 2, None, 0, {}, [], 3])
>>> stream = stream.only_trues()
>>> list(stream)
... [None, 0, {}, []]
```

Opposite to *Stream.only_trues()*.

only_nones()
Keeps only `None` in the stream (for example, for counting).

Returns new processed *Stream* instance.

```
>>> stream = Stream([1, 2, None, 3, None, 4])
>>> stream = stream.only_nones()
>>> list(stream)
... [None, None]
```

only_trues()
Keeps only those elements where `bool(element) == True`.

Returns new processed *Stream* instance.

```
>>> stream = Stream([1, 2, None, 0, {}, [], 3])
>>> stream = stream.only_trues()
>>> list(stream)
... [1, 2, 3]
```

partly_distinct()
Excludes some duplicates from the memory.

Returns new processed *Stream* instance.

Note: All objects in the stream have to be hashable (support `__hash__()`).

Note: It won't guarantee you that all duplicates will be removed especially if your stream is pretty big and cardinallity is huge.

---

peek(*predicate*)
Does the same as Java 8 peek.

Parameters **predicate** (*function*) – Predicate to apply on each element.

Returns new processed *Stream* instance.

Returns a stream consisting of the elements of this stream, additionally performing the provided action on each element as elements are consumed from the resulting stream.

classmethod range (**args*, ***kwargs*)
Creates numerial iterator. Absoultely the same as `Stream.range(10)` and `Stream(range(10))` (in Python 2: `Stream(xrange(10))`). All arguments go to *range()* (`xrange()`) directly.

Returns new processed *Stream* instance.

```
>>> stream = Stream.range(6)
>>> list(stream)
... [0, 1, 2, 3 ,4, 5]
>>> stream = Stream.range(1, 6)
>>> list(stream)
... [1, 2, 3, 4, 5]
>>> stream = Stream.range(1, 6, 2)
>>> list(stream)
... [1, 3, 5]
```

reduce (*function*, *initial=<object object>*)
Applies *reduce()* for the iterator

Parameters

- **function** (*function*) – Reduce function
- **initial** (*object*) – Initial value (if nothing set, first element) would be used.

```
>>> Stream = stream.range(5)
>>> stream.reduce(operator.add)
... 10
```

regexp (*regexp*, *flags=0*)
Filters stream according to the regular expression using `re.match()`. It also supports the same flags as `re.match()`.

Parameters

- **regexp** (*str*) – Regular expression for filtering.
- **flags** (*int*) – Flags from `re`.

Returns new processed *Stream* instance.

```
>>> stream = Stream.range(100)
>>> stream = stream.strings()
>>> stream = stream.regexp(r"^1")
>>> list(stream)
... ['1', '10', '11', '12', '13', '14', '15', '16', '17', '18', '19']
```

reversed()
Reverses the stream.

Returns new processed *Stream* instance.

... note:: If underlying iterator won't support reversing, we are in trouble and need to fetch everything into the memory.

skip (*size*)
Skips first `size` elements.

Parameters **size** (*int*) – The amount of elements to skip.

Returns new processed *Stream* instance.

```
>>> stream = Stream.range(10)
>>> stream = stream.skip(5)
>>> list(stream)
... [5, 6, 7, 8, 9]
```

smallest(*size*)
Returns `size` largest elements from the stream.

Returns new processed *Stream* instance.

```
>>> stream = Stream.range(3000)
>>> stream.smallest(5)
>>> list(stream)
>>> [0, 1, 2, 3, 4]
```

sorted(*key=None*, *reverse=False*)
Sorts the stream elements.

Parameters

- **key** (*function*) – Key function for sorting
- **reverse** (*bool*) – Do we need to sort in descending order?

Returns new processed *Stream* instance.

... note:: Of course no magic here, we need to fetch all elements for sorting into the memory.

strings()
Tries to convert everything to `unicode()` (`str` for Python 3) and keeps only successful attempts.

Returns new processed *Stream* instance.

```
>>> stream = Stream([1, 2.0, "3", "4.0", None, {}])
>>> stream = stream.strings()
>>> list(stream)
... ['1', '2.0', '3', '4.0', 'None', '{}']
```

Note: It is not the same as `stream.map(str)` because it removes failed attempts.

Note: It tries to convert to `unicode` if possible, not `bytes`.

sum()
Returns the sum of elements in the stream.

```
>>> Stream = stream.range(10)
>>> stream = stream.decimals()
>>> stream = stream.sum()
... Decimal('45')
```

Note: Do not use *sum()* here. It does sum regarding to defined `__add__()` of the classes. So it can sum `decimal.Decimal` with `int` for example.

tuplify(*clones=2*)
Tuplifies iterator. Creates a tuple from iterable with `clones` elements.

Parameters **clones** (*int*) – The count of elements in result tuple.

Returns new processed *Stream* instance.

```
>>> stream = Stream.range(2)
>>> stream = stream.tuplify(3)
>>> list(stream)
... [(0, 0, 0), (1, 1, 1)]
```

value_map(*predicate*, ***concurrency_kwargs*)

Maps only value in (key, value) pair. If element is single one, then it would be *Stream.tuplify()* first.

Parameters

- **predicate** (*function*) – Predicate to apply to the value of element in the *Stream*.
- **concurrency_kwargs** (*dict*) – The same concurrency keywords as for *Stream.map()*.

Returns new processed *Stream* instance.

```
>>> stream = Stream.range(4)
>>> stream = stream.tuplify()
>>> stream = stream.value_map(lambda item: item ** 3)
>>> list(stream)
... [(0, 0), (1, 1), (2, 8), (3, 27)]
>>> stream = Stream.range(4)
>>> stream = stream.value_map(lambda item: item ** 3)
>>> list(stream)
... [(0, 0), (1, 1), (2, 8), (3, 27)]
```

values()

Iterates only values from the stream (last element from the `tuple`). If element is single then it will be used.

Returns new processed *Stream* instance.

```
>>> stream = Stream.range(5)
>>> stream = stream.key_map(lambda item: item ** 3)
>>> stream = stream.values()
>>> list(stream)
... [0, 1, 2, 3, 4]
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

S

Symbols

A

C

D

E

F

I

K

L

M

N

O

P

R

S

T

V