aiostream

Feb 22, 2020

Contents

Generator-based operators for asynchronous iteration

CHAPTER 1

Presentation

aiostream provides a collection of stream operators that can be combined to create asynchronous pipelines of operations.

It can be seen as an asynchronous version of itertools, although some aspects are slightly different. Essentially, all the provided operators return a unified interface called a stream. A stream is an enhanced asynchronous iterable providing the following features:

- **Operator pipe-lining** - using pipe symbol `|`
- **Repeatability** - every iteration creates a different iterator
- **Safe iteration context** - using `async with` and the `stream` method
- **Simplified execution** - get the last element from a stream using `await`
- **Slicing and indexing** - using square brackets `[]`
- **Concatenation** - using addition symbol +

1.1 Requirements

The stream operators rely heavily on asynchronous generators (PEP 525):

- python >= 3.6

1.2 Stream operators

The stream operators are separated in 7 categories:

creation	*iterate*, *preserve*, *just*, *call*, *empty*, *throw*, *never*, *repeat*, *count*, *range*
transformation	*map*, *enumerate*, *starmap*, *cycle*, *chunks*
selection	*take*, *takelast*, *skip*, *skiplast*, *getitem*, *filter*, *until*, *takewhile*, *dropwhile*
combination	*map*, *zip*, *merge*, *chain*, *ziplatest*
aggregation	*accumulate*, *reduce*, *list*
advanced	*concat*, *flatten*, *switch*, *concatmap*, *flatmap*, *switchmap*
timing	*spaceout*, *timeout*, *delay*
miscellaneous	*action*, *print*

1.3 Demonstration

The following example demonstrates most of the streams capabilities:

```
import asyncio
from aiostream import stream, pipe


async def main():

    # Create a counting stream with a 0.2 seconds interval
    xs = stream.count(interval=0.2)

    # Operators can be piped using '|'
    ys = xs | pipe.map(lambda x: x**2)

    # Streams can be sliced
    zs = ys[1:10:2]

    # Use a stream context for proper resource management
    async with zs.stream() as streamer:

        # Asynchronous iteration
        async for z in streamer:

            # Print 1, 9, 25, 49 and 81
            print('->', z)

    # Streams can be awaited and return the last value
    print('9^2 = ', await zs)

    # Streams can run several times
    print('9^2 = ', await zs)

    # Streams can be concatenated
    one_two_three = stream.just(1) + stream.range(2, 4)

    # Print [1, 2, 3]
    print(await stream.list(one_two_three))


# Run main coroutine
loop = asyncio.get_event_loop()
```

(continues on next page)

(continued from previous page)

```
loop.run_until_complete(main())
loop.close()
```

More examples are available in the example section.

1.4 References

This library is inspired by:

- PEP 525: Asynchronous Generators
- Rx - Reactive Extensions
- aioreactive - Async/await reactive tools for Python 3.5+
- itertools - Functions creating iterators for efficient looping

CHAPTER 2

Stream operators

The stream operators produce objects of the Stream class.

They are separated in 7 categories:

creation	*iterate*, *preserve*, *just*, *call*, *empty*, *throw*, *never*, *repeat*, *count*, *range*
transformation	*map*, *enumerate*, *starmap*, *cycle*, *chunks*
selection	*take*, *takelast*, *skip*, *skiplast*, *getitem*, *filter*, *until*, *takewhile*, *dropwhile*
combination	*map*, *zip*, *merge*, *chain*, *ziplatest*
aggregation	*accumulate*, *reduce*, *list*
advanced	*concat*, *flatten*, *switch*, *concatmap*, *flatmap*, *switchmap*
timing	*spaceout*, *timeout*, *delay*
miscellaneous	*action*, *print*

They can be found in the *aiostream.stream* module.

Custom stream operators can be created using the @operator decorator.

2.1 Pipe-lining

Most of the operators have a `pipe()` method corresponding to their equivalent pipe operator. They are also gathered and accessible through the `aiostream.pipe` module. The pipe operators allow a 2-step instantiation.

For instance, the following stream:

```
ys = stream.map(xs, lambda x: x**2)
```

is strictly equivalent to:

```
ys = pipe.map(lambda x: x**2)(xs)
```

and can be written as:

```
ys = xs | pipe.map(lambda x: x**2)
```

This syntax comes in handy when several operators are chained:

```
ys = (xs
    | pipe.operator1(*args1)
    | pipe.operator2(*args2)
    | pipe.operator3(*args3))
```

2.2 Creation operators

Note: Those operators do not have a pipe equivalent.

class aiostream.stream.**iterate**(*it*)
Generate values from a sychronous or asynchronous iterable.

class aiostream.stream.**preserve**(*ait*)
Generate values from an asynchronous iterable without explicitly closing the corresponding iterator.

class aiostream.stream.**just**(*value*)
Await if possible, and generate a single value.

class aiostream.stream.**call**(*func*, **args*, ***kwargs*)
Call the given function and generate a single value.

Await if the provided function is asynchronous.

class aiostream.stream.**empty**
Terminate without generating any value.

class aiostream.stream.**throw**(*exc*)
Throw an exception without generating any value.

class aiostream.stream.**never**
Hang forever without generating any value.

class aiostream.stream.**repeat**(*value*, *times=None*, ***, *interval=0*)
Generate the same value a given number of times.

If `times` is `None`, the value is repeated indefinitely. An optional interval can be given to space the values out.

class aiostream.stream.**range**(**args*, *interval=0*)
Generate a given range of numbers.

It supports the same arguments as the builtin function. An optional interval can be given to space the values out.

class aiostream.stream.**count**(*start=0*, *step=1*, ***, *interval=0*)
Generate consecutive numbers indefinitely.

Optional starting point and increment can be defined, respectively defaulting to `0` and `1`.

An optional interval can be given to space the values out.

2.3 Transformation operators

class aiostream.stream.**map**(*source*, *func*, **more_sources*, *ordered=True*, *task_limit=None*)

Apply a given function to the elements of one or several asynchronous sequences.

Each element is used as a positional argument, using the same order as their respective sources. The generation continues until the shortest sequence is exhausted. The function can either be synchronous or asynchronous (coroutine function).

The results can either be returned in or out of order, depending on the corresponding `ordered` argument. This argument is ignored if the provided function is synchronous.

The coroutines run concurrently but their amount can be limited using the `task_limit` argument. A value of `1` will cause the coroutines to run sequentially. This argument is ignored if the provided function is synchronous.

If more than one sequence is provided, they're also awaited concurrently, so that their waiting times don't add up.

It might happen that the provided function returns a coroutine but is not a coroutine function per se. In this case, one can wrap the function with `aiostream.async_` in order to force `map` to await the resulting coroutine. The following example illustrates the use `async_` with a lambda function:

```
from aiostream import stream, async_
...
ys = stream.map(xs, async_(lambda ms: asyncio.sleep(ms / 1000)))
```

Note: *map* is considered a combination operator if used with extra sources, and a transformation operator otherwise

---

class aiostream.stream.**enumerate**(*source*, *start=0*, *step=1*)

Generate `(index, value)` tuples from an asynchronous sequence.

This index is computed using a starting point and an increment, respectively defaulting to `0` and `1`.

class aiostream.stream.**starmap**(*source*, *func*, *ordered=True*, *task_limit=None*)

Apply a given function to the unpacked elements of an asynchronous sequence.

Each element is unpacked before applying the function. The given function can either be synchronous or asynchronous.

The results can either be returned in or out of order, depending on the corresponding `ordered` argument. This argument is ignored if the provided function is synchronous.

The coroutines run concurrently but their amount can be limited using the `task_limit` argument. A value of `1` will cause the coroutines to run sequentially. This argument is ignored if the provided function is synchronous.

class aiostream.stream.**cycle**(*source*)

Iterate indefinitely over an asynchronous sequence.

Note: it does not perform any buffering, but re-iterate over the same given sequence instead. If the sequence is not re-iterable, the generator might end up looping indefinitely without yielding any item.

class aiostream.stream.**chunks**(*source*, *n*)

Generate chunks of size `n` from an asynchronous sequence.

The chunks are lists, and the last chunk might contain less than `n` elements.

2.4 Selection operators

class aiostream.stream.**take**(*source*, *n*)
Forward the first `n` elements from an asynchronous sequence.

If `n` is negative, it simply terminates before iterating the source.

class aiostream.stream.**takelast**(*source*, *n*)
Forward the last `n` elements from an asynchronous sequence.

If `n` is negative, it simply terminates after iterating the source.

Note: it is required to reach the end of the source before the first element is generated.

class aiostream.stream.**skip**(*source*, *n*)
Forward an asynchronous sequence, skipping the first `n` elements.

If `n` is negative, no elements are skipped.

class aiostream.stream.**skiplast**(*source*, *n*)
Forward an asynchronous sequence, skipping the last `n` elements.

If `n` is negative, no elements are skipped.

Note: it is required to reach the `n+1` th element of the source before the first element is generated.

class aiostream.stream.**getitem**(*source*, *index*)
Forward one or several items from an asynchronous sequence.

The argument can either be a slice or an integer. See the slice and item operators for more information.

class aiostream.stream.**filter**(*source*, *func*)
Filter an asynchronous sequence using an arbitrary function.

The function takes the item as an argument and returns `True` if it should be forwarded, `False` otherwise. The function can either be synchronous or asynchronous.

class aiostream.stream.**until**(*source*, *func*)
Forward an asynchronous sequence until a condition is met.

Contrary to the `takewhile` operator, the last tested element is included in the sequence.

The given function takes the item as an argument and returns a boolean corresponding to the condition to meet. The function can either be synchronous or asynchronous.

class aiostream.stream.**takewhile**(*source*, *func*)
Forward an asynchronous sequence while a condition is met.

Contrary to the `until` operator, the last tested element is not included in the sequence.

The given function takes the item as an argument and returns a boolean corresponding to the condition to meet. The function can either be synchronous or asynchronous.

class aiostream.stream.**dropwhile**(*source*, *func*)
Discard the elements from an asynchronous sequence while a condition is met.

The given function takes the item as an argument and returns a boolean corresponding to the condition to meet. The function can either be synchronous or asynchronous.

2.5 Combination operators

class aiostream.stream.**map**(*source*, *func*, **more_sources*, *ordered=True*, *task_limit=None*)

Apply a given function to the elements of one or several asynchronous sequences.

Each element is used as a positional argument, using the same order as their respective sources. The generation continues until the shortest sequence is exhausted. The function can either be synchronous or asynchronous (coroutine function).

The results can either be returned in or out of order, depending on the corresponding `ordered` argument. This argument is ignored if the provided function is synchronous.

The coroutines run concurrently but their amount can be limited using the `task_limit` argument. A value of `1` will cause the coroutines to run sequentially. This argument is ignored if the provided function is synchronous.

If more than one sequence is provided, they're also awaited concurrently, so that their waiting times don't add up.

It might happen that the provided function returns a coroutine but is not a coroutine function per se. In this case, one can wrap the function with `aiostream.async_` in order to force `map` to await the resulting coroutine. The following example illustrates the use `async_` with a lambda function:

```
from aiostream import stream, async_
...
ys = stream.map(xs, async_(lambda ms: asyncio.sleep(ms / 1000)))
```

Note: *map* is considered a combination operator if used with extra sources, and a transformation operator otherwise

---

class aiostream.stream.**zip**(**sources*)

Combine and forward the elements of several asynchronous sequences.

Each generated value is a tuple of elements, using the same order as their respective sources. The generation continues until the shortest sequence is exhausted.

Note: the different sequences are awaited in parrallel, so that their waiting times don't add up.

class aiostream.stream.**merge**(**sources*)

Merge several asynchronous sequences together.

All the sequences are iterated simultaneously and their elements are forwarded as soon as they're available. The generation continues until all the sequences are exhausted.

class aiostream.stream.**chain**(**sources*)

Chain asynchronous sequences together, in the order they are given.

Note: the sequences are not iterated until it is required, so if the operation is interrupted, the remaining sequences will be left untouched.

class aiostream.stream.**ziplatest**(**sources*, *partial=True*, *default=None*)

Combine several asynchronous sequences together, producing a tuple with the lastest element of each sequence whenever a new element is received.

The value to use when a sequence has not procuded any element yet is given by the `default` argument (defaulting to `None`).

The producing of partial results can be disabled by setting the optional argument `partial` to `False`.

All the sequences are iterated simultaneously and their elements are forwarded as soon as they're available. The generation continues until all the sequences are exhausted.

2.6 Aggregatation operators

class aiostream.stream.**accumulate**(*source*, *func=<built-in function add>*, *initializer=None*)
Generate a series of accumulated sums (or other binary function) from an asynchronous sequence.

If `initializer` is present, it is placed before the items of the sequence in the calculation, and serves as a default when the sequence is empty.

class aiostream.stream.**reduce**(*source*, *func*, *initializer=None*)
Apply a function of two arguments cumulatively to the items of an asynchronous sequence, reducing the sequence to a single value.

If `initializer` is present, it is placed before the items of the sequence in the calculation, and serves as a default when the sequence is empty.

class aiostream.stream.**list**(*source*)
Build a list from an asynchronous sequence.

All the intermediate steps are generated, starting from the empty list.

This operator can be used to easily convert a stream into a list:

```
lst = await stream.list(x)
```

..note:: The same list object is produced at each step in order to avoid memory copies.

2.7 Advanced operators

Note: The *concat*, *flatten* and *switch* operators all take a **stream of streams** as an argument (also called **stream of higher order**) and return a flattened stream using their own merging strategy.

---

class aiostream.stream.**concat**(*source*, *task_limit=None*)
Given an asynchronous sequence of sequences, generate the elements of the sequences in order.

The sequences are awaited concurrently, although it's possible to limit the amount of running sequences using the *task_limit* argument.

Errors raised in the source or an element sequence are propagated.

class aiostream.stream.**flatten**(*source*, *task_limit=None*)
Given an asynchronous sequence of sequences, generate the elements of the sequences as soon as they're received.

The sequences are awaited concurrently, although it's possible to limit the amount of running sequences using the *task_limit* argument.

Errors raised in the source or an element sequence are propagated.

class aiostream.stream.**switch**(*source*)
Given an asynchronous sequence of sequences, generate the elements of the most recent sequence.

Element sequences are generated eagerly, and closed once they are superseded by a more recent sequence. Once the main sequence is finished, the last subsequence will be exhausted completely.

Errors raised in the source or an element sequence (that was not already closed) are propagated.

Note: The *concatmap*, *flatmap* and *switchmap* operators provide a simpler access to the three merging strategy listed above.

class aiostream.stream.**concatmap**(*source*, *func*, **more_sources*, *task_limit=None*)
Apply a given function that creates a sequence from the elements of one or several asynchronous sequences, and generate the elements of the created sequences in order.

The function is applied as described in *map*, and must return an asynchronous sequence. The returned sequences are awaited concurrently, although it's possible to limit the amount of running sequences using the *task_limit* argument.

class aiostream.stream.**flatmap**(*source*, *func*, **more_sources*, *task_limit=None*)
Apply a given function that creates a sequence from the elements of one or several asynchronous sequences, and generate the elements of the created sequences as soon as they arrive.

The function is applied as described in *map*, and must return an asynchronous sequence. The returned sequences are awaited concurrently, although it's possible to limit the amount of running sequences using the *task_limit* argument.

Errors raised in a source or output sequence are propagated.

class aiostream.stream.**switchmap**(*source*, *func*, **more_sources*)
Apply a given function that creates a sequence from the elements of one or several asynchronous sequences and generate the elements of the most recently created sequence.

The function is applied as described in *map*, and must return an asynchronous sequence. Errors raised in a source or output sequence (that was not already closed) are propagated.

2.8 Timing operators

class aiostream.stream.**spaceout**(*source*, *interval*)
Make sure the elements of an asynchronous sequence are separated in time by the given interval.

class aiostream.stream.**timeout**(*source*, *timeout*)
Raise a time-out if an element of the asynchronous sequence takes too long to arrive.

Note: the timeout is not global but specific to each step of the iteration.

class aiostream.stream.**delay**(*source*, *delay*)
Delay the iteration of an asynchronous sequence.

2.9 Miscellaneous operators

class aiostream.stream.**action**(*source*, *func*)
Perform an action for each element of an asynchronous sequence without modifying it.

The given function can be synchronous or asynchronous.

class aiostream.stream.**print**(*source*, *template=None*, ***kwargs*)
Print each element of an asynchronous sequence without modifying it.

An optional template can be provided to be formatted with the elements. All the keyword arguments are forwarded to the builtin function print.

CHAPTER 3

Core objects

3.1 Stream base class

class aiostream.core.**Stream**(*factory*)

Enhanced asynchronous iterable.

It provides the following features:

- **Operator pipe-lining** - using pipe symbol `|`
- **Repeatability** - every iteration creates a different iterator
- **Safe iteration context** - using `async with` and the `stream` method
- **Simplified execution** - get the last element from a stream using `await`
- **Slicing and indexing** - using square brackets `[]`
- **Concatenation** - using addition symbol +

It is not meant to be instanciated directly. Use the stream operators instead.

Example:

```
xs = stream.count()     # xs is a stream object
ys = xs | pipe.skip(5) # pipe xs and skip the first 5 elements
zs = ys[5:10:2]        # slice ys using start, stop and step

async with zs.stream() as streamer:  # stream zs in a safe context
    async for z in streamer:         # iterate the zs streamer
        print(z)                     # Prints 10, 12, 14

result = await zs  # await zs and return its last element
print(result)      # Prints 14
result = await zs  # zs can be used several times
print(result)      # Prints 14
```

stream()
Return a streamer context for safe iteration.

Example:

```
xs = stream.count()
async with xs.stream() as streamer:
    async for item in streamer:
        <block>
```

3.2 Stream context manager

aiostream.core.**streamcontext**(*aiterable*)
Return a stream context manager from an asynchronous iterable.

The context management makes sure the aclose asynchronous method of the corresponding iterator has run before it exits. It also issues warnings and RuntimeError if it is used incorrectly.

It is safe to use with any asynchronous iterable and prevent asynchronous iterator context to be wrapped twice.

Correct usage:

```
ait = some_asynchronous_iterable()
async with streamcontext(ait) as streamer:
    async for item in streamer:
        <block>
```

For streams objects, it is possible to use the stream method instead:

```
xs = stream.count()
async with xs.stream() as streamer:
    async for item in streamer:
        <block>
```

3.3 Operator decorator

aiostream.core.**operator**(*func=None*, ***, *pipable=False*)
Create a stream operator from an asynchronous generator (or any function returning an asynchronous iterable).

Decorator usage:

```
@operator
async def random(offset=0., width=1.):
    while True:
        yield offset + width * random.random()
```

Decorator usage for pipable operators:

```
@operator(pipable=True)
async def multiply(source, factor):
    async with streamcontext(source) as streamer:
        async for item in streamer:
            yield factor * item
```

In the case of pipable operators, the first argument is expected to be the asynchronous iteratable used for piping.

The return value is a dynamically created class. It has the same name, module and doc as the original function.

A new stream is created by simply instanciating the operator:

```
xs = random()
ys = multiply(xs, 2)
```

The original function is called at instanciation to check that signature match. In the case of pipable operators, the source is also checked for asynchronous iteration.

The operator also have a pipe class method that can be used along with the piping synthax:

```
xs = random()
ys = xs | multiply.pipe(2)
```

This is strictly equivalent to the previous example.

Other methods are available:

- *original*: the original function as a static method
- *raw*: same as original but add extra checking

The raw method is useful to create new operators from existing ones:

```
@operator(pipable=True)
def double(source):
    return multiply.raw(source, 2)
```

CHAPTER 4

Examples

4.1 Demonstration

The following example demonstrates most of the streams capabilities:

```
import asyncio
from aiostream import stream, pipe


async def main():

    # Create a counting stream with a 0.2 seconds interval
    xs = stream.count(interval=0.2)

    # Operators can be piped using '|'
    ys = xs | pipe.map(lambda x: x**2)

    # Streams can be sliced
    zs = ys[1:10:2]

    # Use a stream context for proper resource management
    async with zs.stream() as streamer:

        # Asynchronous iteration
        async for z in streamer:

            # Print 1, 9, 25, 49 and 81
            print('->', z)

    # Streams can be awaited and return the last value
    print('9^2 = ', await zs)

    # Streams can run several times
    print('9^2 = ', await zs)
```

(continues on next page)

(continued from previous page)

```
    # Streams can be concatenated
    one_two_three = stream.just(1) + stream.range(2, 4)

    # Print [1, 2, 3]
    print(await stream.list(one_two_three))


# Run main coroutine
loop = asyncio.get_event_loop()
loop.run_until_complete(main())
loop.close()
```

4.2 Simple computation

This simple example computes $11^2 + 13^2$ in 1.5 second:

```
import asyncio
from aiostream import stream, pipe


async def main():
    # This stream computes 11^2 + 13^2 in 1.5 second
    xs = (
        stream.count(interval=0.1)      # Count from zero every 0.1 s
        | pipe.skip(10)                 # Skip the first 10 numbers
        | pipe.take(5)                  # Take the following 5
        | pipe.filter(lambda x: x % 2)  # Keep odd numbers
        | pipe.map(lambda x: x ** 2)    # Square the results
        | pipe.accumulate()             # Add the numbers together
    )
    print('11^2 + 13^2 = ', await xs)


# Run main coroutine
loop = asyncio.get_event_loop()
loop.run_until_complete(main())
loop.close()
```

4.3 Preserve a generator

This example shows how to preserve an async generator from being closed by the iteration context:

```
import asyncio
from aiostream import stream, operator


async def main():
    async def agen():
        yield 1
        yield 2
```

(continues on next page)

(continued from previous page)

```
        yield 3

    # The xs stream does not preserve the generator
    xs = stream.iterate(agen())
    print(await xs[0])             # Print 1
    print(await stream.list(xs))   # Print [] (2 and 3 have never yielded)

    # The xs stream does preserve the generator
    xs = stream.preserve(agen())
    print(await xs[0])             # Print 1
    print(await stream.list(xs))   # Print [2, 3]

    # Transform agen into a stream operator
    agen_stream = operator(agen)
    xs = agen_stream()             # agen is now reusable
    print(await stream.list(xs))   # Print [1, 2, 3]
    print(await stream.list(xs))   # Print [1, 2, 3]


# Run main coroutine
loop = asyncio.get_event_loop()
loop.run_until_complete(main())
loop.close()
```

4.4 Norm server

The next example runs a TCP server that computes the euclidean norm of vectors for its clients.

Run the server:

```
$ python3.6 norm_server.py
Serving on ('127.0.0.1', 8888)
```

Test using a netcat client:

```
$ nc localhost 8888
--------------------------------------
Compute the Euclidean norm of a vector
--------------------------------------
[...]
```

Check the logs on the server side, and see how the computation is performed on the fly.

```
import asyncio
from aiostream import stream, pipe

# Constants

INSTRUCTIONS = """\
--------------------------------------
Compute the Euclidean norm of a vector
--------------------------------------
Enter each coordinate of the vector on a separate line, and add an empty
line at the end to get the result. Anything else will result in an error.
```

(continues on next page)

(continued from previous page)

```
> """

ERROR = """\
-> Error ! Try again...
"""

RESULT = """\
-> Euclidean norm: {}
"""


# Client handler

async def euclidean_norm_handler(reader, writer):

    # Define lambdas
    strip =        lambda x: x.decode().strip()
    nonempty =     lambda x: x != ''
    square =       lambda x: x ** 2
    write_cursor = lambda x: writer.write(b'> ')
    square_root =  lambda x: x ** 0.5

    # Create awaitable
    handle_request = (
        stream.iterate(reader)
        | pipe.print('string: {}')
        | pipe.map(strip)
        | pipe.takewhile(nonempty)
        | pipe.map(float)
        | pipe.map(square)
        | pipe.print('square: {:.2f}')
        | pipe.action(write_cursor)
        | pipe.accumulate(initializer=0)
        | pipe.map(square_root)
        | pipe.print('norm -> {:.2f}')
    )

    # Loop over norm computations
    while not reader.at_eof():
        writer.write(INSTRUCTIONS.encode())
        try:
            result = await handle_request
        except ValueError:
            writer.write(ERROR.encode())
        else:
            writer.write(RESULT.format(result).encode())


# Main function

def run_server(bind='127.0.0.1', port=8888):

    # Start the server
    loop = asyncio.get_event_loop()
    coro = asyncio.start_server(euclidean_norm_handler, bind, port)
    server = loop.run_until_complete(coro)
```

(continues on next page)

(continued from previous page)

```
    # Serve requests until Ctrl+C is pressed
    print('Serving on {}'.format(server.sockets[0].getsockname()))
    try:
        loop.run_forever()
    except KeyboardInterrupt:
        pass

    # Close the server
    server.close()
    loop.run_until_complete(server.wait_closed())
    loop.close()


# Main execution

if __name__ == '__main__':
    run_server()
```

4.5 Extra operators

This example shows how extra operators can be created and combined with others:

```
import asyncio
import random as random_module

from aiostream import operator, pipe, streamcontext


@operator
async def random(offset=0., width=1., interval=0.1):
    """Generate a stream of random numbers."""
    while True:
        await asyncio.sleep(interval)
        yield offset + width * random_module.random()


@operator(pipable=True)
async def power(source, exponent):
    """Raise the elements of an asynchronous sequence to the given power."""
    async with streamcontext(source) as streamer:
        async for item in streamer:
            yield item ** exponent


@operator(pipable=True)
def square(source):
    """Square the elements of an asynchronous sequence."""
    return power.raw(source, 2)


async def main():
    xs = (
        random()              # Stream random numbers
        | square.pipe()       # Square the values
```

(continues on next page)

(continued from previous page)

```
        | pipe.take(5)         # Take the first five
        | pipe.accumulate())   # Sum the values
    print(await xs)


# Run main coroutine
loop = asyncio.get_event_loop()
loop.run_until_complete(main())
loop.close()
```

CHAPTER 5

Utilities

aiostream also provides utilites for general asynchronous iteration and asynchronous context management.

5.1 Asynchronous iteration

Utilities for asynchronous iteration.

aiostream.aiter_utils.**aiter**(*obj*)
Access aiter magic method.

aiostream.aiter_utils.**anext**(*obj*)
Access anext magic method.

aiostream.aiter_utils.**await_**(*obj*)
Identity coroutine function.

aiostream.aiter_utils.**async_**(*fn*)
Wrap the given function into a coroutine function.

aiostream.aiter_utils.**is_async_iterable**(*obj*)
Check if the given object is an asynchronous iterable.

aiostream.aiter_utils.**assert_async_iterable**(*obj*)
Raise a TypeError if the given object is not an asynchronous iterable.

aiostream.aiter_utils.**is_async_iterator**(*obj*)
Check if the given object is an asynchronous iterator.

aiostream.aiter_utils.**assert_async_iterator**(*obj*)
Raise a TypeError if the given object is not an asynchronous iterator.

class aiostream.aiter_utils.**AsyncIteratorContext**(*aiterator*)
Asynchronous iterator with context management.

The context management makes sure the aclose asynchronous method of the corresponding iterator has run before it exits. It also issues warnings and RuntimeError if it is used incorrectly.

Correct usage:

```
ait = some_asynchronous_iterable()
async with AsyncIteratorContext(ait) as safe_ait:
    async for item in safe_ait:
        <block>
```

It is nonetheless not meant to use directly. Prefer aitercontext helper instead.

aiostream.aiter_utils.**aitercontext**(*aiterable*, *, *cls=<class 'aiostream.aiter_utils.AsyncIteratorContext'>*)

Return an asynchronous context manager from an asynchronous iterable.

The context management makes sure the aclose asynchronous method has run before it exits. It also issues warnings and RuntimeError if it is used incorrectly.

It is safe to use with any asynchronous iterable and prevent asynchronous iterator context to be wrapped twice.

Correct usage:

```
ait = some_asynchronous_iterable()
async with aitercontext(ait) as safe_ait:
    async for item in safe_ait:
        <block>
```

An optional subclass of AsyncIteratorContext can be provided. This class will be used to wrap the given iterable.

Reference table:

creation	*iterate*, *preserve*, *just*, *call*, *empty*, *throw*, *never*, *repeat*, *count*, *range*
transformation	*map*, *enumerate*, *starmap*, *cycle*, *chunks*
selection	*take*, *takelast*, *skip*, *skiplast*, *getitem*, *filter*, *until*, *takewhile*, *dropwhile*
combination	*map*, *zip*, *merge*, *chain*, *ziplatest*
aggregation	*accumulate*, *reduce*, *list*
advanced	*concat*, *flatten*, *switch*, *concatmap*, *flatmap*, *switchmap*
timing	*spaceout*, *timeout*, *delay*
miscellaneous	*action*, *print*

Python Module Index

a

Index

T

U

Z