aiter Documentation

Richard Kiss

Dec 04, 2019

Contents:

This documentation is a work-in-progress, and your contributions are welcome at <https://github.com/richardkiss/aiter>. Useful patterns building upon asynchronous iterators

`aiter.`**`active_aiter`**(*aiter*)

Wrap an aiter with a task that actively yanks out the items and puts them into a *`aiter.push_aiter`*.

This might be useful if you have an iterator that needs its elements pulled out as soon as they are created and cached in memory, even if the consumer is not yet ready. Be careful though, since getting too far behind can mean lots of memory is consumed, especially if each element uses a lot of memory, and can interfere with the flow control of TCP (for example) that depends on a data backlog.

Parameters **`aiter`** (*`aiter`*) – an async iterator

Returns a *`aiter.push_aiter`* yielding the same elements as aiter

Return type *`aiter.push_aiter`*

`aiter.`**`aiter_forker`**(*aiter*)

If you have an aiter that you would like to fork (split into multiple iterators, each of which produces the same elements), wrap it with this function.

Returns a `aiter._aiter_fork` object that will yield the same objects in the same order. This object supports `fork`, which will let you create a duplicate stream.

Parameters **`aiter`** (*`aiter`*) – an async iterator

Returns a `aiter._aiter_fork`

Return type `aiter._aiter_fork`

`aiter.`**`aiter_to_iter`**(*aiter*, *loop=None*)

Convert an async iterator to a regular iterator by invoking run_until_complete repeatedly.

Parameters

- **`aiter`** (*`aiter`*) – an async iterator
- **`loop`** (*`asyncio event loop`*) – the loop which will run *aiter*

Returns a *synchronous* iterator returning the same elements as aiter

Return type a *synchronous* iterator

`aiter.`**`azip`**(**aiters*)

async version of zip This function takes a list of async iterators and returns a single async iterator that yields tuples of elements.

This iterator advances as slow its slowest component (obviously).

example:

async for a, b, c in azip(aiter1, aiter2, aiter3): print(a, b, c)

Parameters **`aiters`** (*`aiters`*) – one or more async iterators

Returns an aiter returning N-tuples similar to zip

Return type an aiter

aiter.**flatten_aiter**(*aiter*)

Take an async iterator that returns lists and return the individual elements.

Parameters **aiter** (*aiter*) – an async iterator yielding lists

Returns an async iterator where the elements are the flattened inputs

Return type an async iterator

class aiter.**gated_aiter**(*aiter*)

Returns an aiter that you can "push" integer values into. When a number is pushed, that many items are allowed out through the gate.

This is kind of like a discrete version of an electronic transistor.

Parameters **aiter** (*aiter*) – an async iterator

Returns an async iterator yielding the same values as the original aiter

Return type *aiter.gated_aiter*

push(*count*)

Note that several additional items are allowed through the gated_aiter.

Parameters **count** (*int*) – the number of items that can be allowed out the aiter. These are cumulative.

stop()

After the previously authorized items (from *push*) are pulled out the aiter, the aiter will exit.

aiter.**iter_to_aiter**(*iter*)

Parameters **iter** (*synchronous iterator*) – a synchronous iterator

This converts a regular iterator to an async iterator.

aiter.**join_aiters**(*aiter_of_aiters*)

This wrapper takes an aiter of aiters and pipe the items coming out of all of them into a single aiter.

Parameters **aiter_of_aiters** (*async iterator*) – an aiter that yields aiters

Returns an aiter returning elements that come from any of the underlying aiters

Return type async iterator

aiter.**map_aiter**(*map_f*, *aiter*, *worker_count=1*)

Take an async iterator and a map function, and apply the function to everything coming out of the iterator before passing it on. In this case, the map_f must return a list, which will be flattened. Empty lists are okay, so you can filter items by excluding them from the list.

Note that since there are multiple workers, the order or processed elements might not match the input order.

Parameters

- **aiter** (*async iterator*) – an aiter
- **map_f** (*a function, regular or async, that accepts a single parameter and returns a list (or other iterable)*) – the mapping function
- **worker_count** (*int*) – the number of worker tasks that pull items out of aiter

Returns an aiter returning transformed items that have been processed through map_f

Return type an async iterator

`aiter.`**`map_filter_aiter`**`(`*map_f*, *aiter*`)`

Take an async iterator and a map function, and apply the function to everything coming out of the iterator before passing it on. In this case, the map_f must return a list, which will be flattened. Empty lists are okay, so you can filter items by excluding them from the list.

Parameters

- **`aiter`** (*`async iterator`*) – an aiter
- **`map_f`** (*`a function, regular or async, that accepts a single parameter and returns a list (or other iterable)`*) – the mapping function

Returns an aiter returning transformed items that have been processed through map_f

Return type an async iterator

`aiter.`**`preload_aiter`**`(`*preload_size*, *aiter*`)`

This aiter wraps around another aiter, and forces a preloaded buffer of the given size. When an element is removed, the loader is given a kick to try to refill the preload buffer.

Parameters

- **`preload_size`** (*`int`*) – the maximum number of items to attempt to preload
- **`aiter`** (*`async iterator`*) – an aiter

Returns an async iterator yielding the same values as the original aiter

Return type async iterator

`class` `aiter.`**`push_aiter`**

An asynchronous iterator based on a linked-list. Data goes in the head via "push". Allows peeking to determine how many elements are ready.

This is functionally similar to an `async.Queue` object. It creates an aiter that you can *push* items into. Unlike a *Queue* object, you can also invoke *`stop`*, which will raise a *StopAsyncIteration* on the listener's side, allowing for a clean exit.

You'd use this when you want to "turn around" execution, ie. have a task that is occasionally invoked (like a hardware interrupt) to produce a new event for an aiter.

`available_iter``()`

Return a *synchronous* iterator of elements that are immediately available to be consumed without waiting for a task switch.

`is_item_available``()`

Return a boolean indicating whether or not an element is available without blocking for a task switch.

Returns whether or not the aiter has been stopped

Return type bool

`is_len_at_least``(`*n*`)`

Return a boolean indicating whether or not *n* elements are available without blocking for a task switch.

Parameters **`n`** (*`int`*) – count of items

Returns True iff n items are available

Return type bool

`is_stopped``()`

Return a boolean indicating whether or not *`stop`* has been called. Additional elements may still be available.

Returns whether or not the aiter has been stopped

Return type bool

push(**items*)

Accept one or more item and push them to the end of the aiter's queue.

stop()

Raise a *StopAsyncIteration* exception on the listener side once no more already-queued elements are pending.

class aiter.**sharable_aiter**(*aiter*)

Not all iterators can have multiple consumers. For example, asynchronous generators don't allow it. But if you wrap it with one of these, you'll be okay.

class aiter.**stoppable_aiter**(*aiter*)

A wrapper around an iterator that supports a manual shut-off.

CHAPTER 1

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index