aiter Documentation

Richard Kiss

Feb 25, 2019

Contents:

1 Indices and tables 3

Python Module Index 5

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 push_q.

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.

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.push_aiter* object that will yield the same objects in the same order. This object supports *fork*, which will let you create a duplicate stream.

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 asynchronous 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.

Obviously this iterator advances as slow its slowest component.

example:

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

aiter.**flatten_aiter**(*aiter*)
Take an async iterator that returns lists and return the individual elements.

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.

aiter.**iter_to_aiter**(*iter*)
This converts a regular iterator to an async iterator

aiter.**join_aiters**(*aiter_of_aiters*)
Takes an iterator of async iterators and pipe them into a single async iterator.

This creates a task to monitor the main iterator, plus a task for each active iterator that has come out of the main iterator.

aiter.**map_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.

aiter.**map_filter_aiter**(*map_f*, *aiter*)
In this case, the map_f must return a list, which will be flattened. You can filter items by excluding them from the list. Empty lists are okay.

aiter.**preload_aiter**(*preload_size*, *aiter*)
This aiter wraps around another aiter, and forces a preloaded buffer of the given size.

class aiter.**push_aiter**(*tail=None*, *next_preflight=None*)
An asynchronous iterator based on a linked-list. Data goes in the head via "push". Allows peeking to determine how many elements are ready. Can be copied very cheaply by copying the tail. Has a "preflight" that is called whenever __anext__ is called. The __anext__ method is wrapped with a semaphore so multiple tasks can use the same iterator, and each output will go to only one task.

This is functionally very 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.

Parameters

- **next_preflight** (*a function that takes no arguments*) – called every time *__anext__* is invoked. Intended to give the creator a chance to add more elements to the queue, if necessary.
- **tail** (*push_aiter_head*) – the tail where new objects are pushed into the queue. Used by *fork*

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

fork(*is_active=True*)
Create and return a clone of this aiter at its current state. This copy can be used by a separate task, and it will get the same output.

You might use this is you have an event stream that has different types of events, and you want to write several modules, each of which handles a subset of event types. Each module can get a view of the entire stream and simply ignore events it's not interested in.

Parameters **is_active** (*True or False*) – If *False*, this clone is a passive one, that will only return elements that have already been requested by at least one active clone. You might use this feature for a task that keeps stats on events, but it doesn't want to force the events to be removed from because a backlog of events might cause flow-control to be invoked.

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

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

is_stopped()
Return a boolean indicating whether or not *stop* has been called.

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.

CHAPTER 1

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

aiter, 1

Index

A

active_aiter() (in module aiter), 1
aiter (module), 1
aiter_forker() (in module aiter), 1
aiter_to_iter() (in module aiter), 1
available_iter() (aiter.push_aiter method), 2
azip() (in module aiter), 1

F

flatten_aiter() (in module aiter), 1
fork() (aiter.push_aiter method), 2

G

gated_aiter (class in aiter), 1

I

is_item_available() (aiter.push_aiter method), 2
is_len_at_least() (aiter.push_aiter method), 2
is_stopped() (aiter.push_aiter method), 2
iter_to_aiter() (in module aiter), 1

J

join_aiters() (in module aiter), 1

M

map_aiter() (in module aiter), 1
map_filter_aiter() (in module aiter), 1

P

preload_aiter() (in module aiter), 1
push() (aiter.push_aiter method), 2
push_aiter (class in aiter), 2

S

sharable_aiter (class in aiter), 2
stop() (aiter.push_aiter method), 2