eventkit

Release 0.8.8

Ewald de Wit

Sep 29, 2022

CONTENTS

The primary use cases of eventkit are

- to send events between loosely coupled components;
- to compose all kinds of event-driven data pipelines.

The interface is kept as Pythonic as possible, with familiar names from Python and its libraries where possible. For scheduling asyncio is used and there is seamless integration with it.

See the examples and the introduction notebook to get a true feel for the possibilities.

CHAPTER

ONE

INSTALLATION

```
pip3 install eventkit
```

Python version 3.6 or higher is required.

CHAPTER

TWO

EXAMPLES

Create an event and connect two listeners

```
import eventkit as ev

def f(a, b):
    print(a * b)

def g(a, b):
    print(a / b)

event = ev.Event()
event += f
event += g
event.emit(10, 5)
```

Create a simple pipeline

```
import eventkit as ev

event = (
    ev.Sequence('abcde')
    .map(str.upper)
    .enumerate()
)

print(event.run())  # in Jupyter: await event.list()
```

Output:

```
[(0, 'A'), (1, 'B'), (2, 'C'), (3, 'D'), (4, 'E')]
```

Create a pipeline to get a running average and standard deviation

```
import random
import eventkit as ev

source = ev.Range(1000).map(lambda i: random.gauss(0, 1))

event = source.array(500)[ev.ArrayMean, ev.ArrayStd].zip()

print(event.last().run())  # in Jupyter: await event.last()
```

Output:

```
[(0.00790957852672618, 1.0345673260655333)]
```

Combine async iterators together

```
import asyncio
import eventkit as ev

async def ait(r):
    for i in r:
        await asyncio.sleep(0.1)
        yield i

async def main():
    async for t in ev.Zip(ait('XYZ'), ait('123')):
        print(t)

asyncio.get_event_loop().run_until_complete(main())  # in Jupyter: await main()
```

Output:

```
('X', '1')
('Y', '2')
('Z', '3')
```

Real-time video analysis pipeline

```
self.video = VideoStream(conf.CAM_ID)
scene = self.video | FaceTracker | SceneAnalyzer
lastScene = scene.aiter(skip_to_last=True)
async for frame, persons in lastScene:
    ...
```

Full source code

CHAPTER

THREE

DISTRIBUTED COMPUTING

The distex library provides a `poolmap` extension method to put multiple cores or machines to use:

```
from distex import Pool
import eventkit as ev
import bz2

pool = Pool()
# await pool  # un-comment in Jupyter
data = [b'A' * 1000000] * 1000

pipe = ev.Sequence(data).poolmap(pool, bz2.compress).map(len).mean().last()

print(pipe.run())  # in Jupyter: print(await pipe)
pool.shutdown()
```

CHAPTER

FOUR

INSPIRED BY:

- Qt Signals & Slots
- itertools
- aiostream
- Bacon
- aioreactive
- Reactive extensions
- underscore.js
- .NET Events

CHAPTER

FIVE

DOCUMENTATION

The complete API documentation.

5.1 eventkit

Release 0.8.8.

5.1.1 Event

class eventkit.event.**Event**(*name=''*, *_with_error_done_events=True*)

Enable event passing between loosely coupled components. The event emits values to connected listeners and has a selection of operators to create general data flow pipelines.

Parameters
name (str) – Name to use for this event.

__await__()

Asynchronously await the next emit of an event:

```
async def coro():
    args = await event
    ...
```

If the event does an empty `emit()`, then the value of `args` is set to `util.NO_VALUE`.

wait() and *__await__()* are each other's inverse.

async __aiter__(*skip_to_last=False*, *tuples=False*)

Synonym for *aiter()* with default arguments:

```
async def coro():
    async for args in event:
        ...
```

aiterate() and *__aiter__()* are each other's inverse.

error_event

Sub event that emits errors from this event as `emit(source, exception)`.

done_event

Sub event that emits when this event is done as `emit(source)`.

name()

This event's name.

Return type
str

done()

`True` if event has ended with no more emits coming, `False` otherwise.

Return type
bool

set_done()

Set this event to be ended. The event should not emit anything after that.

value()

This event's last emitted value.

connect(*listener*, *error=None*, *done=None*, *keep_ref=False*)

Connect a listener to this event. If the listener is added multiple times then it is invoked just as many times on emit.

The += operator can be used as a synonym for this method:

```
import eventkit as ev

def f(a, b):
    print(a * b)

def g(a, b):
    print(a / b)

event = ev.Event()
event += f
event += g
event.emit(10, 5)
```

Parameters

- **listener** – The callback to invoke on emit of this event. It gets the `*args` from an emit as arguments. If the listener is a coroutine function, or a function that returns an awaitable, the awaitable is run in the asyncio event loop.
- **error** – The callback to invoke on error of this event. It gets (this event, exception) as two arguments.
- **done** – The callback to invoke on ending of this event. It gets this event as single argument.
- **keep_ref** (`bool`) –
  - `True`: A strong reference to the callable is kept
  - `False`: If the callable allows weak refs and it is garbage collected, then it is automatically disconnected from this event.

disconnect(*listener*, *error=None*, *done=None*)

Disconnect a listener from this event.

The -= operator can be used as a synonym for this method.

Parameters

- **listener** – The callback to disconnect. The callback is removed at most once. It is valid if the callback is already not connected.
- **error** – The error callback to disconnect.
- **done** – The done callback to disconnect.

disconnect_obj(*obj*)

Disconnect all listeners on the given object. (also the error and done listeners).

Parameters

obj – The target object that is to be completely removed from this event.

emit(**args*)

Emit a new value to all connected listeners.

Parameters

args – Argument values to emit to listeners.

emit_threadsafe(**args*)

Threadsafe version of *emit()* that doesn't invoke the listeners directly but via the event loop of the main thread.

clear()

Disconnect all listeners.

run()

Start the asyncio event loop, run this event to completion and return all values as a list:

```
import eventkit as ev

ev.Timer(0.25, count=10).run()
->
[0.25, 0.5, 0.75, 1.0, 1.25, 1.5, 1.75, 2.0, 2.25, 2.5]
```

Note: When running inside a Jupyter notebook this will give an error that the asyncio event loop is already running. This can be remedied by applying nest_asyncio or by using the top-level `await` statement of Jupyter:

```
await event.list()
```

Return type

List

pipe(**targets*)

Form several events into a pipe:

```
import eventkit as ev

e1 = ev.Sequence('abcde')
e2 = ev.Enumerate().map(lambda i, c: (i, i + ord(c)))
e3 = ev.Star().pluck(1).map(chr)
```

(continues on next page)

(continued from previous page)

```
e1.pipe(e2, e3)     # or: ev.Event.Pipe(e1, e2, e3)
->
['a', 'c', 'e', 'g', 'i']
```

Parameters
targets (*Event*) – One or more Events that have no source yet, or `Event` constructors that needs no arguments.

fork(**targets*)

Fork this event into one or more target events. Square brackets can be used as a synonym:

```
import eventkit as ev

ev.Range(2, 5)[ev.Min, ev.Max, ev.Sum].zip()
->
[(2, 2, 2), (2, 3, 5), (2, 4, 9)]
```

The events in the fork can be combined by one of the join methods of `Fork`.

Parameters
targets (*Event*) – One or more events that have no source yet, or `Event` constructors that need no arguments.

Return type
`Fork`

async aiter(*skip_to_last=False*, *tuples=False*)

Create an asynchronous iterator that yields the emitted values from this event:

```
async def coro():
    async for args in event.aiter():
        ...
```

__aiter__() is a synonym for *aiter()* with default arguments,

Parameters

- **skip_to_last** (`bool`) –
  - `True`: Backlogged source values are skipped over to yield only the latest value. Can be used as a slipper clutch between a source that produces too fast and the handling that can't keep up.
  - `False`: All events are yielded.
- **tuples** (`bool`) –
  - `True`: Always yield arguments as a tuple.
  - `False`: Unpack single argument tuples.

static init(*obj*, *event_names*)

Convenience function for initializing multiple events as members of the given object.

Parameters
event_names (`Iterable`) – Names to use for the created events.

static create(*obj*)

Create an event from a async iterator, awaitable, or event constructor without arguments.

Parameters
obj – The source object. If it's already an event then it is passed as-is.

static wait(*future*)

Create a new event that emits the value of the awaitable when it becomes available and then set this event done.

wait() and *__await__()* are each other's inverse.

Parameters
future (Awaitable) – Future to wait on.

Return type
Wait

static aiterate(*ait*)

Create a new event that emits the yielded values from the asynchronous iterator.

The asynchronous iterator serves as a source for both the time and value of emits.

aiterate() and *__aiter__()* are each other's inverse.

Parameters
ait (AsyncIterable) – The asynchronous source iterator. It must `await` at least once; If necessary use:

```
await asyncio.sleep(0)
```

Return type
Aiterate

static sequence(*values*, *interval=0*, *times=None*)

Create a new event that emits the given values. Supply at most one `interval` or `times`.

Parameters

- **values** (Iterable) – The source values.
- **interval** (float) – Time interval in seconds between values.
- **times** (Optional[Iterable[float]]) – Relative times for individual values, in seconds since start of event. The sequence should match `values`.

Return type
Sequence

static repeat(*value=<NoValue>*, *count=1*, *interval=0*, *times=None*)

Create a new event that repeats `value` a number of `count` times.

Parameters

- **value** – The value to emit.
- **count** – Number of times to emit.
- **interval** (float) – Time interval in seconds between values.
- **times** (Optional[Iterable[float]]) – Relative times for individual values, in seconds since start of event. The sequence should match `values`.

Return type
`Repeat`

`static range`(**args*, *interval=0*, *times=None*)

Create a new event that emits the values from a range.

Parameters

- **args** – Same as for built-in `range`.
- **interval** (`float`) – Time interval in seconds between values.
- **times** (`Optional[Iterable[float]]`) – Relative times for individual values, in seconds since start of event. The sequence should match the range.

Return type
`Range`

`static timerange`(*start=0*, *end=None*, *step=1*)

Create a new event that emits the datetime value, at that datetime, from a range of datetimes.

Parameters

- **start** – Start time, can be specified as:
  - `datetime.datetime`.
  - `datetime.time`: Today is used as date.
  - `int` or `float`: Number of seconds relative to now. Values will be quantized to the given step.
- **end** – End time, can be specified as:
  - `datetime.datetime`.
  - `datetime.time`: Today is used as date.
  - `None`: No end limit.
- **step** – Number of seconds, or `datetime.timedelta`, to space between values.

Return type
`Timerange`

`static timer`(*interval*, *count=None*)

Create a new timer event that emits at regularly paced intervals the number of seconds since starting it.

Parameters

- **interval** (`float`) – Time interval in seconds between emits.
- **count** (`Optional[int]`) – Number of times to emit, or `None` for no limit.

Return type
`Timer`

`static marble`(*s*, *interval=0*, *times=None*)

Create a new event that emits the values from a Rx-type marble string.

Parameters

- **s** (`str`) – The string with characters that are emitted.
- **interval** (`float`) – Time interval in seconds between values.

- **times** (Optional[Iterable[float]]) – Relative times for individual values, in seconds since start of event. The sequence should match the marble string.

Return type
Marble

filter(*predicate=<class 'bool'>*)

For every source value, apply predicate and re-emit when True.

Parameters
predicate – The function to test every source value with. The default is to test the general truthiness with bool().

Return type
Filter

skip(*count=1*)

Drop the first count values from source and follow the source after that.

Parameters
count (int) – Number of source values to drop.

Return type
Skip

take(*count=1*)

Re-emit first count values from the source and then end.

Parameters
count (int) – Number of source values to re-emit.

Return type
Take

takewhile(*predicate=<class 'bool'>*)

Re-emit values from the source until the predicate becomes False and then end.

Parameters
predicate – The function to test every source value with. The default is to test the general truthiness with bool().

Return type
TakeWhile

dropwhile(*predicate=<function Event.<lambda>>*)

Drop source values until the predicate becomes False and after that re-emit everything from the source.

Parameters
predicate – The function to test every source value with. The default is to test the inverted general truthiness.

Return type
DropWhile

takeuntil(*notifier*)

Re-emit values from the source until the notifier emits and then end. If the notifier ends without any emit then keep passing source values.

Parameters
notifier (*Event*) – Event that signals to end this event.

Return type
TakeUntil

constant(*constant*)

On emit of the source emit a constant value:

```
emit(value) -> emit(constant)
```

Parameters
constant – The constant value to emit.

Return type
Constant

iterate(*it*)

On emit of the source, emit the next value from an iterator:

```
emit(a, b, ...) -> emit(next(it))
```

The time of events follows the source and the values follow the iterator.

Parameters
it – The source iterator to use for generating values. When the iterator is exhausted the event is set to be done.

Return type
Iterate

count(*start=0*, *step=1*)

Count and emit the number of source emits:

```
emit(a, b, ...) -> emit(count)
```

Parameters

- **start** – Start count.
- **step** – Add count by this amount for every new source value.

Return type
Count

enumerate(*start=0*, *step=1*)

Add a count to every source value:

```
emit(a, b, ...) -> emit(count, a, b, ...)
```

Parameters

- **start** – Start count.
- **step** – Increase by this amount for every new source value.

Return type
Enumerate

timestamp()

Add a timestamp (from time.time()) to every source value:

```
emit(a, b, ...) -> emit(timestamp, a, b, ...)
```

The timestamp is the float number in seconds since the midnight Jan 1, 1970 epoch.

Return type
Timestamp

partial(**left_args*)

Pad source values with extra arguments on the left:

```
emit(a, b, ...) -> emit(*left_args, a, b, ...)
```

Parameters
left_args – Arguments to inject.

Return type
Partial

partial_right(**right_args*)

Pad source values with extra arguments on the right:

```
emit(a, b, ...) -> emit(a, b, ..., *right_args)
```

Parameters
right_args – Arguments to inject.

Return type
PartialRight

star()

Unpack a source tuple into positional arguments, similar to the star operator:

```
emit((a, b, ...)) -> emit(a, b, ...)
```

star() and *pack()* are each other's inverse.

Return type
Star

pack()

Pack positional arguments into a tuple:

```
emit(a, b, ...) -> emit((a, b, ...))
```

star() and *pack()* are each other's inverse.

Return type
Pack

pluck(**selections*)

Extract arguments or nested properties from the source values.

Select which argument positions to keep:

```
emit(a, b, c, d).pluck(1, 2) -> emit(b, c)
```

Re-order arguments:

```
emit(a, b, c).pluck(2, 1, 0) -> emit(c, b, a)
```

To do an empty emit leave `selections` empty:

```
emit(a, b).pluck() -> emit()
```

Select nested properties from positional arguments:

```
emit(person, account).pluck(
    '1.number', '0.address.street') ->

emit(account.number, person.address.street)
```

If no value can be extracted then `NO_VALUE` is emitted in its place.

Parameters
selections (Union[int, str]) – The values to extract.

Return type
Pluck

map(*func*, *timeout=None*, *ordered=True*, *task_limit=None*)

Apply a sync or async function to source values using positional arguments:

```
emit(a, b, ...) -> emit(func(a, b, ...))
```

or if `func` returns an awaitable then it will be awaited:

```
emit(a, b, ...) -> emit(await func(a, b, ...))
```

In case of timeout or other failure, `NO_VALUE` is emitted.

Parameters

- **func** – The function or coroutine constructor to apply.
- **timeout** – Timeout in seconds since coroutine is started
- **ordered** –
  - `True`: The order of emitted results preserves the order of the source values.
  - `False`: Results are in order of completion.
- **task_limit** – Max number of concurrent tasks, or None for no limit.

`timeout`, `ordered` and `task_limit` apply to async functions only.

Return type
Map

emap(*constr*, *joiner*)

Higher-order event map that creates a new `Event` instance for every source value:

```
emit(a, b, ...) -> new Event constr(a, b, ...)
```

Parameters

- **constr** – Constructor function for creating a new event. Apart from returning an `Event`, the constructor may also return an awaitable or an asynchronous iterator, in which case an `Event` will be created.
- **joiner** (`AddableJoinOp`) – Join operator to combine the emits of nested events.

Return type
`Emap`

mergemap(*constr*)

emap() that uses *merge()* to combine the nested events:

```
marbles = [
    'A   B    C    D',
    '_1   2  3    4',
    '__K   L     M   N']

ev.Range(3).mergemap(lambda v: ev.Marble(marbles[v]))
->
['A', '1', 'K', 'B', '2', 'L', '3', 'C', 'M', '4', 'D', 'N']
```

Return type
`Mergemap`

concatmap(*constr*)

emap() that uses *concat()* to combine the nested events:

```
marbles = [
    'A    B    C    D',
    '_       1    2    3    4',
    '__                  K    L      M   N']

ev.Range(3).concatmap(lambda v: ev.Marble(marbles[v]))
->
['A', 'B', '1', '2', '3', 'K', 'L', 'M', 'N']
```

Return type
`Concatmap`

chainmap(*constr*)

emap() that uses *chain()* to combine the nested events:

```
marbles = [
    'A    B    C    D           ',
    '_       1    2    3    4',
    '__                  K    L      M   N']

ev.Range(3).chainmap(lambda v: ev.Marble(marbles[v]))
->
['A', 'B', 'C', 'D', '1', '2', '3', '4', 'K', 'L', 'M', 'N']
```

Return type
Chainmap

switchmap(*constr*)

emap() that uses *switch()* to combine the nested events:

```
marbles = [
    'A    B    C    D           ',
    '_                K    L      M   N',
    '__       1    2      3    4'
]
ev.Range(3).switchmap(lambda v: Event.marble(marbles[v]))
->
['A', 'B', '1', '2', 'K', 'L', 'M', 'N'])
```

Return type
Switchmap

reduce(*func*, *initializer=<NoValue>*)

Apply a two-argument reduction function to the previous reduction result and the current value and emit the new reduction result.

Parameters

- **func** – Reduction function:

```
emit(args) -> emit(func(prev_args, args))
```

- **initializer** – First argument of first reduction:

```
first_result = func(initializer, first_value)
```

If no initializer is given, then the first result is emitted on the second source emit.

Return type
Reduce

min()

Minimum value.

Return type
Min

max()

Maximum value.

Return type
Max

sum(*start=0*)

Total sum.

Parameters
start – Value added to total sum.

Return type
Sum

product(*start=1*)

Total product.

Parameters
start – Initial start value.

Return type
Product

mean()

Total average.

Return type
Mean

any()

Test if predicate holds for at least one source value.

Return type
Any

all()

Test if predicate holds for all source values.

Return type
All

ema(*n=None*, *weight=None*)

Exponential moving average.

Parameters

- **n** (Optional[int]) – Number of periods.
- **weight** (Optional[float]) – Weight of new value.

Give either n or weight. The relation is weight = 2 / (n + 1).

Return type
Ema

previous(*count=1*)

For every source value, emit the count-th previous value:

```
source:  -ab---c--d-e-
output:  --a---b--c-d-
```

Starts emitting on the count + 1-th source emit.

Parameters
count (int) – Number of periods to go back.

Return type
Previous

pairwise()

Emit (previous_source_value, current_source_value) tuples. Starts emitting on the second source emit:

```
source:  -a----b------c--------d-----
output:  ------(a,b)--(b,c)----(c,d)-
```

Return type
Pairwise

changes()

Emit only source values that have changed from the previous value.

Return type
Changes

unique(*key=None*)

Emit only unique values, dropping values that have already been emitted.

Parameters
key – *The callable ``key(value)`* is used to group values. The default of None groups values by equality. The resulting group must be hashable.

Return type
Unique

last()

Wait until source has ended and re-emit its last value.

Return type
Last

list()

Collect all source values and emit as list when the source ends.

Return type
List

deque(*count=0*)

Emit a deque with the last count values from the source (or less in the lead-in phase).

Parameters
count – Number of last periods to use, or 0 to use all.

Return type
Deque

array(*count=0*)

Emit a numpy array with the last count values from the source (or less in the lead-in phase).

Parameters
count – Number of last periods to use, or 0 to use all.

Return type
Array

chunk(*size*)

Chunk values up in lists of equal size. The last chunk can be shorter.

Parameters
size (int) – Chunk size.

Return type
Chunk

chunkwith(*timer*, *emit_empty=True*)

Emit a chunked list of values when the timer emits.

Parameters

- **timer** (*Event*) – Event to use for timing the chunks.
- **emit_empty** (bool) – Emit empty list if no values present since last emit.

Return type
ChunkWith

chain(**sources*)

Re-emit from a source until it ends, then move to the next source, Repeat until all sources have ended, ending the chain. Emits from pending sources are queued up:

```
source 1:  -a----b---c|
source 2:        --2-----3--4|
source 3:  ------------x---------y--|
output:    -a----b---c2--3--4x---y--|
```

Parameters
sources (*Event*) – Source events.

Return type
Chain

merge(**sources*)

Re-emit everything from the source events:

```
source 1:  -a----b-------------c------d-|
source 2:     ------1-----2------3--4-|
source 3:      --------x----y--|
output:    -a----b--1--x--2-y--c-3--4-d-|
```

Parameters
sources – Source events.

Return type
Merge

concat(**sources*)

Re-emit everything from one source until it ends and then move to the next source:

```
source 1:  -a----b-----|
source 2:    --1-----2-----3----4--|
source 3:                  -----------x--y--|
output:    -a----b---------3----4----x--y--|
```

Parameters
sources – Source events.

Return type
Concat

switch(**sources*)

Re-emit everything from one source and move to another source as soon as that other source starts to emit:

```
source 1:  -a----b---c-----d---|
source 2:        -----------x---y-|
source 3:  ---------1----2----3-----|
output:    -a----b--1----2--x---y---|
```

Parameters
sources – Source events.

Return type
Switch

zip(**sources*)

Zip sources together: The i-th emit has the i-th value from each source as positional arguments. Only emits when each source has emtted its i-th value and ends when any source ends:

```
source 1:    -a----b------------------c------d---e--f---|
source 2:    --------1-------2-------3---------4-----|
output emit: --------(a,1)---(b,2)----(c,3)----(d,4)-|
```

Parameters
sources – Source events.

Return type
Zip

ziplatest(**sources*, *partial=True*)

Emit zipped values with the latest value from each of the source events. Emits every time when a source emits:

```
source 1:   -a-------------------b-------c---|
source 2:   ---------------1---------------------2------|
output emit: (a,NoValue)---(a,1)-(b,1)---(c,1)--(c,2)--|
```

Parameters

- **sources** – Source events.
- **partial** (bool) –
  - True: Use NoValue for sources that have not emitted yet.
  - False: Wait until all sources have emitted.

Return type
Ziplatest

delay(*delay*)

Time-shift all source events by a delay:

```
source:  -abc-d-e---f---|
output:  ---abc-d-e---f---|
```

This applies to the source errors and the source done event as well.

Parameters
delay – Time delay of all events (in seconds).

Return type
Delay

timeout(*timeout*)

When the source doesn't emit for longer than the timeout period, do an empty emit and set this event as done.

Parameters
timeout – Timeout value.

Return type
Timeout

throttle(*maximum*, *interval*, *cost_func=None*)

Limit number of emits per time without dropping values. Values that come in too fast are queued and re-emitted as soon as allowed by the limits.

A nested `status_event` emits `True` when throttling starts and `False` when throttling ends.

The limit can be dynamically changed with `set_limit`.

Parameters

- **maximum** – Maximum payload per interval.
- **interval** – Time interval (in seconds).
- **cost_func** – The sum of `cost_func(value)` for every source value inside the `interval` that is to remain under the `maximum`. The default is to count every source value as 1.

Return type
Throttle

debounce(*delay*, *on_first=False*)

Filter out values from the source that happen in rapid succession.

Parameters

- **delay** – Maximal time difference (in seconds) between successive values before debouncing kicks in.
- **on_first** (bool) –
  - True: First value is send immediately and following values in the rapid succession are dropped:

```
source: -abcd----efg-
output: -a-------e---
```

  - False: Last value of a rapid succession is send after the delay and the values before that are dropped:

```
source:  -abcd----efg--
output:   ----d------g-
```

Return type
Debounce

copy()

Create a shallow copy of the source values.

Return type
Copy

deepcopy()

Create a deep copy of the source values.

Return type
Deepcopy

sample(*timer*)

At the times that the timer emits, sample the value from this event and emit the sample.

Parameters
timer (*Event*) – Event used to time the samples.

Return type
Sample

errors()

Emit errors from the source.

Return type
Errors

end_on_error()

End on any error from the source.

Return type
EndOnError

5.1.2 Op

5.1.3 Create

5.1.4 Select

5.1.5 Transform

5.1.6 Aggregate

5.1.7 Combine

5.1.8 Timing

5.1.9 Array

5.1.10 Misc

5.1.11 Util

PYTHON MODULE INDEX

e

INDEX

Symbols

A

C

D

E

F

I

L

M

N

P

R

S

T

U

V

W

Z