ChanPy

Release 0.0.2

Jake Magers

May 12, 2020

CONTENTS

1 API **3**
- 1.1 Core 3
- 1.2 Transducers 13

2 Glossary **19**

Python Module Index **21**

Index **23**

ChanPy is a CSP Python library based on Clojure core.async. It provides equivalents for all the functions in core.async and provides channels that can be used with or without asyncio. ChanPy even provides support for applying transformations across channels in the same way Clojure does, via `transducers`.

Source code and issue tracking: https://github.com/JMagers/chanpy

License: Apache License 2.0

CHAPTER

ONE

API

Channels, the center around any CSP library. The `core` module provides all the essential functions for creating and managing them. For convenience, all of the public members of `core` exist at the top-level of the package.

Like Clojure's core.async, ChanPy channels have direct support for transformations via transducers. The `transducers` module provides many transducers as well as functions to help create and use them.

1.1 Core

Core functions for working with channels.

ChanPy's `channels` have full support for use with asyncio coroutines, callback based code, and multi-threaded designs. The functions in this module are designed to primarily accept and produce channels and by doing so, can be used almost identically with each of the aforementioned styles.

An extremely valuable feature from Clojure's core.async library is the ability to cheaply create asynchronous "processes" using go blocks. ChanPy, like aiochan, is able to do something similar by leveraging Python's own asyncio library. Channels can easily be used from within coroutines which can then be added as tasks to an event loop. ChanPy additionally offers ways for these tasks to be added from threads without a running event loop.

Note: Unless explicitly stated otherwise, any function involving asynchronous work should be assumed to require an asyncio event loop. Many of these functions leverage the use of an event loop for the efficiency reasons stated earlier. Threads with a running event loop will be able to directly call these functions but threads without one will be required to register one to themselves using `set_loop()` prior to doing so. Calling `set_loop()` will be unnecessary for threads that were created with `thread()` as those threads will have already been registered.

exception `chanpy.core.`**`QueueSizeError`**

Maximum pending channel operations exceeded.

Raised when too many operations have been enqueued on a channel. Consider using a windowing buffer to prevent enqueuing too many puts or altering your design to have less asynchronous "processes" access the channel at once.

Note: This exception is an indication of a design error. It should NOT be caught and discarded.

`chanpy.core.`**`alt`**`(`**ops*, *priority=False*, *default=Undefined*`)`

Returns an awaitable representing the first and only channel operation to finish.

Accepts a variable number of operations that either get from or put to a channel and commits only one of them. If no *default* is provided, then only the first op to finish will be committed. If *default* is provided and none of the *ops* finish immediately, then no operation will be committed and *default* will instead be used to complete the returned awaitable.

Parameters

- **`ops`** – Operations that either get from or put to a channel. A get operation is represented as simply a channel to get from. A put operation is represented as an iterable of the form `[channel, val]`, where *val* is an item to put onto *channel*.
- **`priority`** – An optional bool. If True, operations will be tried in order. If False, operations will be tried in random order.
- **`default`** – An optional value to use in case no operation finishes immediately.

Returns An awaitable that evaluates to a tuple of the form `(val, ch)`. If *default* is not provided, then *val* will be what the first successful operation returned and *ch* will be the channel used in that operation. If *default* is provided and none of the operations complete immediately, then the awaitable will evaluate to `(default, 'default')`.

Raises

- **`ValueError`** – If *ops* is empty or contains both a get and put operation to the same channel.
- **`RuntimeError`** – If the calling thread has no running event loop.

See also:

`b_alt()`

`chanpy.core.`**`b_alt`**`(`**ops*, *priority=False*, *default=Undefined*`)`

Same as *`alt()`* except it blocks instead of returning an awaitable.

Does not require an event loop.

`chanpy.core.`**`buffer`**`(`*n*`)`

Returns a fixed buffer with a capacity of *n*.

Puts to channels with this buffer will block if the capacity is reached.

Parameters **`n`** – A positive number.

`class` `chanpy.core.`**`chan`**`(`*buf_or_n=None*, *xform=None*, *ex_handler=None*`)`

A CSP channel with optional buffer, transducer, and exception handler.

Channels support multiple producers and consumers and may be buffered or unbuffered. Additionally, buffered channels can optionally have a transformation applied to the values put to them through the use of a *`transducer`*.

Channels may be used by threads with or without a running asyncio event loop. The *`get()`*, *`put()`*, and *`alt()`* functions provide direct support for asyncio by returning awaitables. Channels additionally can be used as asynchronous generators when used with `async for`. *`b_get()`*, *`b_put()`*, *`b_alt()`*, and *`to_iter()`* provide blocking alternatives for threads which do not wish to use asyncio. Channels can even be used with callback based code via *`f_put()`* and *`f_get()`*. A very valuable feature of channels is that producers and consumers of them need not be of the same type. For example, a value placed onto a channel with *`put()`* (asyncio) can be taken by a call to *`b_get()`* (blocking) from a separate thread.

A select/alt feature is also available using the *`alt()`* and *`b_alt()`* functions. This feature allows one to attempt many operations on a channel at once and only have the first operation to complete actually committed.

Once closed, future puts will be unsuccessful but any pending puts will remain until consumed or until a *`reduced`* value is returned from the transformation. Once exhausted, all future gets will complete with the value None. Because of this, None cannot be put onto a channel either directly or indirectly through a transformation.

Parameters

- **`buf_or_n`** – An optional buffer that may be expressed as a positive number. If it's a number, a fixed buffer of that capacity will be used. If None, the channel will be unbuffered.
- **`xform`** – An optional `transducer` for transforming elements put onto the channel. *buf_or_n* must not be None if this is provided.
- **`ex_handler`** – An optional function to handle exceptions raised during transformation. Must accept the raised exception as a parameter. Any non-None return value will be put onto the buffer.

See also:

`buffer()` `dropping_buffer()` `sliding_buffer()`

`b_get`(**, wait=True*)

Same as `get()` except it blocks instead of returning an awaitable.

Does not require an event loop.

`b_put`(*val, *, wait=True*)

Same as `put()` except it blocks instead of returning an awaitable.

Does not require an event loop.

`close`()

Closes the channel.

`f_get`(*f*)

Asynchronously takes a value from the channel and calls *f* with it.

Does not require an event loop.

Parameters **`f`** – A non-blocking function accepting a single argument. Will be passed the value taken from the channel or None if the channel is exhausted.

Raises **`QueueSizeError`** – If the channel has too many pending get operations.

`f_put`(*val, f=None*)

Asynchronously puts *val* onto the channel and calls *f* when complete.

Does not require an event loop.

Parameters

- **`val`** – A non-None value to put onto the channel.
- **`f`** – An optional non-blocking function accepting the completion status of the put operation.

Returns False if the channel is already closed or True if it's not.

Raises **`QueueSizeError`** – If the channel has too many pending put operations.

`get`(**, wait=True*)

Attempts to take a value from the channel.

Gets will fail if the channel is exhausted or if `wait=False` and a value is not immediately available.

Parameters **`wait`** – An optional bool that if False, fails the get operation when a value is not immediately available.

Returns An awaitable that evaluates to a value taken from the channel or None if the operation fails.

Raises

- **RuntimeError** – If the calling thread has no running event loop.
- *QueueSizeError* – If the channel has too many pending get operations.

offer(*val*)
Same as `b_put(val, wait=False)`.

poll()
Same as `b_get(wait=False)`.

put(*val*, ***, *wait=True*)
Attempts to put *val* onto the channel.

Puts will fail in the following cases:

- the channel is already closed
- `wait=False` and *val* cannot be immediately put onto the channel
- a *reduced* value is returned during transformation

Parameters

- **val** – A non-None value to put onto the channel.
- **wait** – An optional bool that if False, fails the put operation when it cannot complete immediately.

Returns An awaitable that will evaluate to True if *val* is accepted onto the channel or False if it's not.

Raises

- **RuntimeError** – If the calling thread has no running event loop.
- *QueueSizeError* – If the channel has too many pending put operations.

to_iter()
Returns an iterator over the channel's values.

Calling `next()` on the returned iterator may block. Does not require an event loop.

chanpy.core.**dropping_buffer**(*n*)
Returns a windowing buffer that drops inputs when capacity is reached.

Puts to channels with this buffer will appear successful after the capacity is reached but nothing will be added to the buffer.

Parameters **n** – A positive number representing the buffer capacity.

chanpy.core.**get_loop**()
Returns the event loop for the current thread.

If `set_loop()` has been used to register an event loop to the current thread, then that loop will be returned. If no such event loop exists, then returns the running loop in the current thread.

Raises **RuntimeError** – If no event loop has been registered and no loop is running in the current thread.

See also:

`set_loop()`

chanpy.core.**go**(*coro*)
Adds a coroutine object as a task to the current event loop.

coro will be added as a task to the event loop returned from `get_loop()`.

Parameters **`coro`** – A coroutine object.

Returns A channel containing the return value of *coro*.

`chanpy.core.`**`is_chan`**`(`*ch*`)`

Returns True if *ch* is a channel.

`chanpy.core.`**`is_unblocking_buffer`**`(`*buf*`)`

Returns True if puts to the buffer will never block.

`chanpy.core.`**`map`**`(`*f*, *chs*, *buf_or_n=None*`)`

Repeatedly takes a value from each channel and applies *f*.

Asynchronously takes one value per source channel and passes the resulting list of values as positional arguments to *f*. Each return value of *f* will be put onto the returned channel. The returned channel closes if any one of the source channels closes.

Parameters

- **`f`** – A non-blocking function accepting `len(chs)` positional arguments.
- **`chs`** – An iterable of source channels.
- **`buf_or_n`** – An optional buffer to use with the returned channel. Can also be represented as a positive number. See `chan`.

Returns A channel containing the return values of *f*.

`chanpy.core.`**`merge`**`(`*chs*, *buf_or_n=None*`)`

Returns a channel that emits values from the provided source channels.

Transfers all values from *chs* onto the returned channel. The returned channel closes after the transfer finishes.

Parameters

- **`chs`** – An iterable of source channels.
- **`buf_or_n`** – An optional buffer to use with the returned channel. Can also be represented as a positive number. See `chan`.

See also:

`mix`

`class` `chanpy.core.`**`mix`**`(`*ch*`)`

Consumes values from each of its source channels and puts them onto *ch*.

A source channel can be added with `admix()` and removed with `unmix()` or `unmix_all()`.

A source channel can be given a set of attribute flags to modify how it is consumed with `toggle()`. If a channel has its `'pause'` attribute set to True, then the mix will stop consuming from it. Else if its `'mute'` attribute is set, then the channel will still be consumed but its values discarded.

A source channel may also be soloed by setting the `'solo'` attribute. If any source channel is soloed, then all of its other attributes will be ignored. Furthermore, non-soloed channels will have their attributes ignored and instead will take on whatever attribute has been set with `solo_mode()` (defaults to `'mute'` if `solo_mode()` hasn't been invoked).

Parameters **`ch`** – A channel to put values onto.

See also:

`merge()`

admix(*ch*)
Adds *ch* as a source channel of the mix.

solo_mode(*mode*)
Sets the *mode* for non-soloed source channels.

For as long as there is at least one soloed channel, non-soloed source channels will have their attributes ignored and will instead take on the provided *mode*.

Parameters **mode** – Either `'pause'` or `'mute'`. See *mix* for behaviors.

toggle(*state_map*)
Merges *state_map* with the current state of the mix.

state_map will be used to update the attributes of the mix's source channels by merging its contents with the current state of the mix. If *state_map* contains a channel that is not currently in the mix, then that channel will be added with the given attributes.

Parameters **state_map** – A dictionary of the form `{channel: attribute_map}` where *attribute_map* is a dictionary of the form `{attribute: bool}`. Supported attributes are `{'solo', 'pause', 'mute'}`. See *mix* for corresponding behaviors.

unmix(*ch*)
Removes *ch* from the set of source channels.

unmix_all()
Removes all source channels from the mix.

class chanpy.core.**mult**(*ch*)
A mult(iple) of the source channel that puts each of its values to its taps.

tap() can be used to subscribe a channel to the mult and therefore receive copies of the values from *ch*. Taps can later be unsubscribed using *untap()* or *untap_all()*.

No tap will receive the next value from *ch* until all taps have accepted the current value. If no tap exists, values will still be consumed from *ch* but will be discarded.

Parameters **ch** – A channel to get values from.

tap(*ch*, ***, *close=True*)
Subscribes a channel as a consumer of the mult.

Parameters

- **ch** – A channel to receive values from the mult's source channel.
- **close** – An optional bool. If True, *ch* will be closed after the source channel becomes exhausted.

untap(*ch*)
Unsubscribes a channel from the mult.

untap_all()
Unsubscribes all taps from the mult.

chanpy.core.**onto_chan**(*ch*, *coll*, ***, *close=True*)
Asynchronously transfers values from an iterable to a channel.

Parameters

- **ch** – A channel to put values onto.
- **coll** – An iterable to get values from.
- **close** – An optional bool. If True, *ch* will be closed after transfer finishes.

Returns A channel that closes after the transfer finishes.

See also:

`to_chan()`

`chanpy.core.`**`pipe`**`(`*from_ch*, *to_ch*, *, *close=True*`)`

Asynchronously transfers all values from *from_ch* to *to_ch*.

Parameters

- **`from_ch`** – A channel to get values from.
- **`to_ch`** – A channel to put values onto.
- **`close`** – An optional bool. If True, *to_ch* will be closed after transfer finishes.

Returns *to_ch*.

`chanpy.core.`**`pipeline`**`(`*n*, *to_ch*, *xform*, *from_ch*, *, *close=True*, *ex_handler=None*, *mode='thread'*, *chunksize=1*`)`

Transforms values from *from_ch* to *to_ch* in parallel.

Values from *from_ch* will be transformed in parallel using a pool of threads or processes. The transducer will be applied to values from *from_ch* independently (not across values) and may produce zero or more outputs per input. The transformed values will be put onto *to_ch* in order relative to the inputs. If *to_ch* closes, then *from_ch* will no longer be consumed from.

Parameters

- **`n`** – A positive int specifying the maximum number of workers to run in parallel.
- **`to_ch`** – A channel to put the transformed values onto.
- **`xform`** – A `transducer` that will be applied to each value independently (not across values).
- **`from_ch`** – A channel to get values from.
- **`close`** – An optional bool. If True, *to_ch* will be closed after transfer finishes.
- **`ex_handler`** – An optional exception handler. See `chan`.
- **`mode`** – Either `'thread'` or `'process'`. Specifies whether to use a thread or process pool to parallelize work.
- **`chunksize`** – An optional positive int that's only relevant when `mode='process'`. Specifies the approximate amount of values each worker will receive at once.

Returns A channel that closes after the transfer finishes.

Note: If CPython is being used with `mode='thread'`, then *xform* must release the GIL at some point in order to achieve any parallelism.

See also:

`pipeline_async()`

`chanpy.core.`**`pipeline_async`**`(`*n*, *to_ch*, *af*, *from_ch*, *, *close=True*`)`

Transforms values from *from_ch* to *to_ch* in parallel using an async function.

Values will be gathered from *from_ch* and passed to *af* along with a channel for its outputs to be placed onto. *af* will be called as `af(val, result_ch)` and should return immediately, having spawned some asynchronous operation that will place zero or more outputs onto *result_ch*. Up to *n* of these asynchronous "processes" will be run at once, each of which will be required to close their corresponding *result_ch* when finished. Values from these result channels will be placed onto *to_ch* in order relative to the inputs from *from_ch*. If *to_ch* closes, then *from_ch* will no longer be consumed from and any unclosed result channels will be closed.

Parameters

- **`n`** – A positive int representing the maximum number of asynchronous "processes" to run at once.
- **`to_ch`** – A channel to place the results onto.
- **`af`** – A non-blocking function that will be called as `af(val, result_ch)`. This function will presumably spawn some kind of asynchronous operation that will place outputs onto *result_ch*. *result_ch* must be closed before the asynchronous operation finishes.
- **`from_ch`** – A channel to get values from.
- **`close`** – An optional bool. If True, *to_ch* will be closed after transfer finishes.

Returns A channel that closes after the transfer finishes.

See also:

`pipeline()`

`chanpy.core.`**`promise_chan`**(*xform=None*, *ex_handler=None*)

Returns a channel that emits the same value forever.

Creates a channel with an optional *`transducer`* and exception handler that always returns the same value to consumers. The value emitted will be the first item put onto the channel or None if the channel was closed before the first put.

Parameters

- **`xform`** – An optional transducer. See *`chan`*.
- **`ex_handler`** – An optional exception handler. See *`chan`*.

`class` `chanpy.core.`**`pub`**(*ch*, *topic_fn*, *buf_fn=None*)

A pub(lication) of the source channel divided into topics.

The values of *ch* will be categorized into topics defined by *topic_fn*. Each topic will be given its own *`mult`* for channels to subscribe to. Channels can be subscribed to a given topic with *`sub()`* and unsubscribed with *`unsub()`* or *`unsub_all()`*.

Parameters

- **`ch`** – A channel to get values from.
- **`topic_fn`** – A function that given a value from *ch* returns a topic identifier.
- **`buf_fn`** – An optional function that given a topic returns a buffer to be used with that topic's *`mult`* channel. If not provided, channels will be unbuffered.

See also:

`mult`

`sub`(*topic*, *ch*, ***, *close=True*)

Subscribes a channel to the given *topic*.

Parameters

- **`topic`** – A topic identifier.
- **`ch`** – A channel to subscribe.

- **close** – An optional bool. If True, *ch* will be closed when the source channel is exhausted.

unsub(*topic*, *ch*)
Unsubscribes a channel from the given *topic*.

unsub_all(*topic=Undefined*)
Unsubscribes all subs from a *topic* or all topics if not provided.

`chanpy.core.`**reduce**(*rf*, *init*, *ch*) → result_ch
reduce(rf, ch) -> result_ch

Asynchronously reduces a channel.

Asynchronously collects values from *ch* and reduces them using *rf*, placing the final result in the returned channel. If *ch* is exhausted, then *init* will be used as the result. If *ch* is not exhausted, then the first call to *rf* will be `rf(init, val)` where *val* is taken from *ch*. *rf* will continue to get called as `rf(prev_rf_return, next_val)` until either *ch* is exhausted or *rf* returns a *`reduced`* value.

Parameters

- **rf** – A *`reducing function`* accepting 2 args. If *init* is not provided, then *rf* must return a value to be used as *init* when called with 0 args.
- **init** – An optional initial value.
- **ch** – A channel to get values from.

Returns A channel containing the result of the reduction.

See also:

`transduce()`

`chanpy.core.`**set_loop**(*loop*)
Registers an event loop to the current thread.

Any thread not running an asyncio event loop will be required to run this function before any asynchronous functions are used. This is because most of the functions in this library that involve asynchronous work are designed to do so through an event loop.

A single event loop may be registered to any number of threads at once.

Parameters **loop** – An asyncio event loop.

Returns A context manager that on exit will unregister loop and reregister the event loop that was originally set.

See also:

`get_loop()` *`thread()`*

`chanpy.core.`**sliding_buffer**(*n*)
Returns a windowing buffer that evicts the oldest element when capacity is reached.

Puts to channels with this buffer will complete successfully after the capacity is reached but will evict the oldest element in the buffer.

Parameters **n** – A positive number representing the buffer capacity.

`chanpy.core.`**split**(*pred*, *ch*, *true_buf=None*, *false_buf=None*)
Splits the values of a channel into two channels based on a predicate.

Returns a tuple of the form `(true_ch, false_ch)` where *true_ch* contains all the values from *ch* where the predicate returns True and *false_ch* contains all the values that return False.

Parameters

- **`pred`** – A predicate function, `pred(value) -> bool`.
- **`ch`** – A channel to get values from.
- **`true_buf`** – An optional buffer to use with *true_ch*. See *`chan`*.
- **`false_buf`** – An optional buffer to use with *false_ch*. See *`chan`*.

Returns A tuple of the form `(true_ch, false_ch)`.

`chanpy.core.`**`thread`**(*f*, *executor=None*)

Registers current loop to a separate thread and then calls *f* from it.

Calls *f* in another thread, returning immediately to the calling thread. The separate thread will have the loop from the calling thread registered to it while *f* runs.

Parameters

- **`f`** – A function accepting no arguments.
- **`executor`** – An optional `ThreadPoolExecutor` to submit *f* to.

Returns A channel containing the return value of *f*.

`chanpy.core.`**`timeout`**(*msecs*)

Returns a channel that closes after given milliseconds.

`chanpy.core.`**`to_chan`**(*coll*)

Returns a channel that emits all values from an iterable and then closes.

Parameters **`coll`** – An iterable to get values from.

See also:

`onto_chan()`

`chanpy.core.`**`to_list`**(*ch*)

Asynchronously reduces the values from a channel to a list.

Returns A channel containing a list of values from *ch*.

`chanpy.core.`**`transduce`**(*xform*, *rf*, *init*, *ch*) → result_ch

transduce(xform, rf, ch) -> result_ch

Asynchronously reduces a channel with a transformation.

Asynchronously collects values from *ch* and reduces them using a transformed reducing function equal to `xform(rf)`. See *`reduce()`* for more information on reduction. After the transformed reducing function has received all input it will be called once more with a single argument, the accumulated result.

Parameters

- **`xform`** – A *`transducer`*.
- **`rf`** – A *`reducing function`* accepting both 1 and 2 arguments. If *init* is not provided, then *rf* must return a value to be used as *init* when called with 0 arguments.
- **`init`** – An optional initial value.
- **`ch`** – A channel to get values from.

Returns A channel containing the result of the reduction.

See also:

`reduce()`

1.2 Transducers

Transducers, composable algorithmic transformations.

Notable features of transducers:

- Are decoupled from the context in which they are used. This means they can be reused with any transducible process, including iterables and channels.
- Are composable with simple function composition. See `comp()`.
- Support early termination via `reduced` values.

Creating transducers: Transducers are also known as `reducing function` transformers. They are simply functions that accept a reducing function as input and return a new reducing function as output.

See clojure.org for more information about transducers.

`chanpy.transducers.`**`append`**`(`*appendable*, *val*`)` → appended result

append(appendable) -> appendable

append() -> []

A `reducing function` that appends *val* to *appendable*.

`chanpy.transducers.`**`cat`**`(`*rf*`)`

A `transducer` that concatenates the contents of its inputs.

Expects each input to be an iterable, the contents of which will be outputted one at a time.

See also:

`mapcat()`

`chanpy.transducers.`**`comp`**`(`**xforms*`)`

Returns a new `transducer` equal to the composition of *xforms*.

The returned transducer passes values through the given transformations from left to right.

Parameters **`xforms`** – Transducers.

`chanpy.transducers.`**`completing`**`(`*rf*, *cf=<function identity>*`)`

Returns a wrapper around *rf* that calls *cf* when invoked with one argument.

Parameters

- **`rf`** – A `reducing function`.
- **`cf`** – An optional function that accepts a single argument. Used as the completion arity for the returned `reducing function`.

Returns A `reducing function` that dispatches to *cf* when called with a single argument or *rf* when called with any other number of arguments.

`chanpy.transducers.`**`dedupe`**`(`*rf*`)`

A `transducer` that drops consecutive duplicate values.

`chanpy.transducers.`**`distinct`**`(`*rf*`)`

A `transducer` that drops duplicate values.

`chanpy.transducers.`**`drop`**`(`*n*`)`

Returns a `transducer` that drops the first *n* inputs.

The returned transducer drops the first *n* inputs if *n* < the number of inputs. If *n* >= the number of inputs, then drops all of them.

Parameters **`n`** – A number.

`chanpy.transducers.`**`drop_last`**`(`*n*`)`

Returns a *`transducer`* that drops the last *n* values.

The returned transducer drops the last *n* inputs if *n* < the number of inputs. If *n* >= the number of inputs, then drops all of them.

Parameters **`n`** – A number.

Note: No values will be outputted until *n* inputs have been received.

`chanpy.transducers.`**`drop_while`**`(`*pred*`)`

Returns a *`transducer`* that drops inputs until the predicate returns False.

Parameters **`pred`** – A predicate function, `pred(input) -> bool`.

`chanpy.transducers.`**`ensure_reduced`**`(`*x*`)`

Returns `reduced(x)` if *x* is not already a *`reduced`* value.

`chanpy.transducers.`**`filter`**`(`*pred*`)`

Returns a *`transducer`* that outputs values for which predicate returns True.

Parameters **`pred`** – A predicate function, `pred(value) -> bool`.

See also:

`filter_indexed()` *`remove()`*

`chanpy.transducers.`**`filter_indexed`**`(`*f*`)`

Returns a *`transducer`* which filters values based on `f(index, value)`.

The returned transducer outputs values that return True when passed into *f* with the corresponding index. *f* will be called as `f(index, value)` where *index* represents the nth *value* to be passed into the transformation starting at 0.

Parameters **`f`** – A function, `f(index, value) -> bool`.

See also:

`filter()` *`remove_indexed()`*

`chanpy.transducers.`**`identity`**`(`*x*`)`

A NOP *`transducer`* that simply returns its argument.

`chanpy.transducers.`**`interpose`**`(`*sep*`)`

Returns a *`transducer`* that outputs each input separated by *sep*.

`chanpy.transducers.`**`into`**`(`*appendable*, *xform*, *coll*`)`

Transfers all values from *coll* into *appendable* with a transformation.

Same as *`itransduce(xform, append, appendable, coll)`*.

`chanpy.transducers.`**`ireduce`**`(`*rf*, *init*, *coll*`)` → reduction result

ireduce(rf, coll) -> reduction result

Returns the result of reducing an iterable.

Reduces *coll* by repeatedly calling *rf* with 2 arguments. If *coll* is empty, then *init* will be returned. If *coll* is not empty, then the first call to *rf* will be `rf(init, first_coll_value)`. *rf* will continue to get called as `rf(prev_rf_return, next_coll_value)` until either *coll* is exhausted or *rf* returns a *`reduced`* value.

Parameters

- **rf** – A *reducing function* accepting 2 arguments. If *init* is not provided, then *rf* must return a value to be used as *init* when called with 0 arguments.
- **init** – An optional initial value.
- **coll** – An iterable.

See also:

reduced *itransduce()*

`chanpy.transducers.`**`is_reduced`**(*x*)

Returns True if *x* is the result from a call to *reduced*.

`chanpy.transducers.`**`itransduce`**(*xform*, *rf*, *init*, *coll*) → reduction result

itransduce(xform, rf, coll) -> reduction result

Returns the result of reducing an iterable with a transformation.

Reduces *coll* using a transformed reducing function equal to `xform(rf)`. See *ireduce()* for more information on reduction. After the transformed reducing function has received all input it will be called once more with a single argument, the result thus far.

Parameters

- **xform** – A *transducer*.
- **rf** – A *reducing function* accepting both 1 and 2 arguments. If *init* is not provided, then *rf* must return a value to be used as *init* when called with 0 arguments.
- **init** – An optional initial value.
- **coll** – An iterable.

See also:

ireduce()

`chanpy.transducers.`**`keep`**(*f*)

Returns a *transducer* that outputs the non-None return values of `f(value)`.

See also:

keep_indexed()

`chanpy.transducers.`**`keep_indexed`**(*f*)

Returns a *transducer* that outputs the non-None return values of `f(index, value)`.

The returned transducer outputs the non-None return values of `f(index, value)` where *index* represents the nth *value* to be passed into the transformation starting at 0.

Parameters **f** – A function, `f(index, value) -> any`.

See also:

keep()

`chanpy.transducers.`**`map`**(*f*)

Returns a *transducer* that applies *f* to each input.

Parameters **f** – A function, `f(input) -> any`.

See also:

map_indexed()

`chanpy.transducers.`**`map_indexed`**`(`*f*`)`

Returns a `transducer` that transforms using `f(index, value)`.

The returned transducer applies *f* to each value with the corresponding index. *f* will be called as `f(index, value)` where *index* represents the nth *value* to be passed into the transformation starting at 0.

Parameters **`f`** – A function, `f(index, value) -> any`.

See also:

`chanpy.transducers.map()`

`chanpy.transducers.`**`mapcat`**`(`*f*`)`

Returns a `transducer` that applies *f* to each input and concatenates the result.

`chanpy.transducers.`**`multi_arity`**`(`**funcs*`)`

Returns a new multi-arity function which dispatches to *funcs*.

The returned function will dispatch to the provided functions based on the number of positional arguments it was called with. If called with zero arguments it will dispatch to the first function in *funcs*, if called with one argument it will dispatch to the second function in *funcs*, etc.

Parameters **`funcs`** – Functions to dispatch to. Each function represents a different arity for the returned function. None values may be used to represent arities that don't exist.

`chanpy.transducers.`**`partition`**`(`*n*, *step=None*, *pad=None*`)`

Returns a `transducer` that partitions values into tuples of size *n*.

The returned transducer partitions the values into tuples of size *n* that are *step* items apart.

- If *step* < *n*, partitions will overlap *n* - *step* elements.
- If *step* == *n*, the default, no overlapping or dropping will occur.
- If *step* > *n*, *step* - *n* values will be dropped between partitions.

If the last partition size is greater than 0 but less than *n*:

- If *pad* is None, the last partition is discarded.
- If *pad* exists, its values will be used to fill the partition to a desired size of *n*. The padded partition will be outputted even if its size is < *n*.

Parameters

- **`n`** – A positive int representing the length of each partition. The last partition may be < *n* if *pad* is provided.
- **`step`** – An optional positive int used as the offset between partitions.
- **`pad`** – An optional iterable of any size. If the last partition size is greater than 0 and less than *n*, then *pad* will be applied to it.

See also:

`partition_all()`

`chanpy.transducers.`**`partition_all`**`(`*n*, *step=None*`)`

Returns a `transducer` that partitions all values.

The returned transducer partitions values into tuples of size *n* that are *step* items apart. Partitions at the end may have a size < *n*.

- If *step* < *n*, partitions will overlap *n* - *step* elements.
- If *step* == *n*, the default, no overlapping or dropping will occur.

- If *step* > *n*, *step* - *n* values will be dropped between partitions.

  Parameters

  - **n** – An optional positive int representing the size of each partition (may be less for partitions at the end).
  - **step** – An optional positive int used as the offset between partitions. Defaults to *n*.

See also:

partition()

`chanpy.transducers.`**`partition_by`**`(`*f*`)`

Returns a *transducer* that partitions inputs by *f*.

In this context, a partition is defined as a tuple containing consecutive items for which `f(item)` returns the same value. That is to say, a new partition will be started each time `f(item)` returns a different value than the previous call.

Parameters **f** – A function, `f(item) -> any`.

`chanpy.transducers.`**`random_sample`**`(`*prob*`)`

Returns a *transducer* that selects inputs with the given probability.

Parameters **prob** – A number between 0 and 1.

`class` `chanpy.transducers.`**`reduced`**`(`*x*`)`

Wraps *x* in such a way that a reduce will terminate with *x*.

A *reducing function* can return `reduced(x)` to terminate a reduction early with the value *x*.

If used with a transduce function such as *itransduce()*, the reduction will terminate with the result of invoking the completion arity with *x*.

`chanpy.transducers.`**`reductions`**`(`*rf*, *init=Undefined*`)`

Returns a *transducer* that outputs each intermediate result from a reduction.

The transformation first outputs *init*. From then on, all outputs will be derived from `rf(prev_output, val)` where *val* is an input to the transformation. *rf* will continue to get called until all input has been exhausted or *rf* returns a *reduced* value.

Parameters

- **rf** – A *reducing function* accepting 2 arguments. If *init* is not provided, then *rf* must return a value to be used as *init* when called with 0 arguments.
- **init** – An optional initial value.

See also:

ireduce()

`chanpy.transducers.`**`remove`**`(`*pred*`)`

Returns a *transducer* that drops values for which predicate returns True.

Parameters **pred** – A predicate function, `pred(value) -> bool`.

See also:

filter() *remove_indexed()*

`chanpy.transducers.`**`remove_indexed`**`(`*f*`)`

Returns a *transducer* which drops values based on `f(index, value)`.

The returned transducer drops values that return True when passed into *f* with the corresponding index. *f* will be called as `f(index, value)` where *index* represents the nth *value* to be passed into the transformation starting at 0.

Parameters **`f`** – A function, `f(index, value) -> bool`.

See also:

`filter_indexed()` *`remove()`*

`chanpy.transducers.`**`replace`**`(`*smap*`)`

Returns a *`transducer`* that replaces values based on the given dictionary.

The returned transducer replaces any input that's a key in *smap* with the key's corresponding value. Inputs that aren't a key in *smap* will be outputted without any transformation.

Parameters **`smap`** – A dictionary that maps values to their replacements.

`chanpy.transducers.`**`take`**`(`*n*`)`

Returns a *`transducer`* that outputs the first *n* inputs.

The returned transducer outputs the first *n* inputs if *n* < the number of inputs. If *n* >= the number of inputs, then outputs all of them.

Parameters **`n`** – A number.

`chanpy.transducers.`**`take_last`**`(`*n*`)`

Returns a *`transducer`* that outputs the last *n* inputs.

The returned transducer outputs the last *n* inputs if *n* < the number of inputs. If *n* >= the number of inputs, then outputs all of them.

Note: No values will be outputted until the completion arity is called.

Parameters **`n`** – A number.

`chanpy.transducers.`**`take_nth`**`(`*n*`)`

Returns a *`transducer`* that outputs every nth input starting with the first.

Parameters **`n`** – A positive int.

`chanpy.transducers.`**`take_while`**`(`*pred*`)`

Returns a *`transducer`* that outputs values until the predicate returns False.

Parameters **`pred`** – A predicate function, `f(value) -> bool`.

`chanpy.transducers.`**`unreduced`**`(`*x*`)`

Returns *x* if it's not a *`reduced`* value else returns the unwrapped value.

`chanpy.transducers.`**`xiter`**`(`*xform*, *coll*`)`

Returns an iterator over the transformed elements in *coll*.

Useful for when you want to transform an iterable into another iterable in a lazy fashion.

Parameters

- **`xform`** – A *`transducer`*.
- **`coll`** – A potentially infinite iterable.

CHAPTER TWO

GLOSSARY

reducing function A type of function used for reduction.

It may have up to three different arities:

- The *step* arity accepts 2 arguments, the accumulated result and an input. It returns the new accumulated result of the reduction.
- The *init* arity is optional and accepts 0 arguments. It returns an initial value for the accumulated result if one is not explicitly provided.
- The *completion* arity is required only when used with a `transducer` and accepts 1 argument, the accumulated result of the reduction. See clojure.org for more information about use with transducers.

`multi_arity()` can be used to help create these multi-arity functions.

Reducing functions additionally support a form of early termination via `reduced` values.

transducer Also known as a `reducing function` transformer, it's simply a function that accepts a reducing function as input and returns a new reducing function as output. It's commonly referred to as a transformation or xform throughout the documentation.

The `transducers` module provides many transducers as well as functions to help create and use them.

See clojure.org for information about transducers.

PYTHON MODULE INDEX

C

chanpy.core, 3
chanpy.transducers, 13

INDEX

A

admix() (*chanpy.core.mix method*), 7
alt() (*in module chanpy.core*), 3
append() (*in module chanpy.transducers*), 13

B

b_alt() (*in module chanpy.core*), 4
b_get() (*chanpy.core.chan method*), 5
b_put() (*chanpy.core.chan method*), 5
buffer() (*in module chanpy.core*), 4

C

cat() (*in module chanpy.transducers*), 13
chan (*class in chanpy.core*), 4
chanpy.core (*module*), 3
chanpy.transducers (*module*), 13
close() (*chanpy.core.chan method*), 5
comp() (*in module chanpy.transducers*), 13
completing() (*in module chanpy.transducers*), 13

D

dedupe() (*in module chanpy.transducers*), 13
distinct() (*in module chanpy.transducers*), 13
drop() (*in module chanpy.transducers*), 13
drop_last() (*in module chanpy.transducers*), 14
drop_while() (*in module chanpy.transducers*), 14
dropping_buffer() (*in module chanpy.core*), 6

E

ensure_reduced() (*in module chanpy.transducers*), 14

F

f_get() (*chanpy.core.chan method*), 5
f_put() (*chanpy.core.chan method*), 5
filter() (*in module chanpy.transducers*), 14
filter_indexed() (*in module chanpy.transducers*), 14

G

get() (*chanpy.core.chan method*), 5
get_loop() (*in module chanpy.core*), 6
go() (*in module chanpy.core*), 6

I

identity() (*in module chanpy.transducers*), 14
interpose() (*in module chanpy.transducers*), 14
into() (*in module chanpy.transducers*), 14
ireduce() (*in module chanpy.transducers*), 14
is_chan() (*in module chanpy.core*), 7
is_reduced() (*in module chanpy.transducers*), 15
is_unblocking_buffer() (*in module chanpy.core*), 7
itransduce() (*in module chanpy.transducers*), 15

K

keep() (*in module chanpy.transducers*), 15
keep_indexed() (*in module chanpy.transducers*), 15

M

map() (*in module chanpy.core*), 7
map() (*in module chanpy.transducers*), 15
map_indexed() (*in module chanpy.transducers*), 15
mapcat() (*in module chanpy.transducers*), 16
merge() (*in module chanpy.core*), 7
mix (*class in chanpy.core*), 7
mult (*class in chanpy.core*), 8
multi_arity() (*in module chanpy.transducers*), 16

O

offer() (*chanpy.core.chan method*), 6
onto_chan() (*in module chanpy.core*), 8

P

partition() (*in module chanpy.transducers*), 16
partition_all() (*in module chanpy.transducers*), 16
partition_by() (*in module chanpy.transducers*), 17
pipe() (*in module chanpy.core*), 9
pipeline() (*in module chanpy.core*), 9
pipeline_async() (*in module chanpy.core*), 9
poll() (*chanpy.core.chan method*), 6
promise_chan() (*in module chanpy.core*), 10

pub (*class in chanpy.core*), 10
put() (*chanpy.core.chan method*), 6

Q

QueueSizeError, 3

R

random_sample() (*in module chanpy.transducers*), 17
reduce() (*in module chanpy.core*), 11
reduced (*class in chanpy.transducers*), 17
reducing function, **19**
reductions() (*in module chanpy.transducers*), 17
remove() (*in module chanpy.transducers*), 17
remove_indexed() (*in module chanpy.transducers*), 17
replace() (*in module chanpy.transducers*), 18

S

set_loop() (*in module chanpy.core*), 11
sliding_buffer() (*in module chanpy.core*), 11
solo_mode() (*chanpy.core.mix method*), 8
split() (*in module chanpy.core*), 11
sub() (*chanpy.core.pub method*), 10

T

take() (*in module chanpy.transducers*), 18
take_last() (*in module chanpy.transducers*), 18
take_nth() (*in module chanpy.transducers*), 18
take_while() (*in module chanpy.transducers*), 18
tap() (*chanpy.core.mult method*), 8
thread() (*in module chanpy.core*), 12
timeout() (*in module chanpy.core*), 12
to_chan() (*in module chanpy.core*), 12
to_iter() (*chanpy.core.chan method*), 6
to_list() (*in module chanpy.core*), 12
toggle() (*chanpy.core.mix method*), 8
transduce() (*in module chanpy.core*), 12
transducer, **19**

U

unmix() (*chanpy.core.mix method*), 8
unmix_all() (*chanpy.core.mix method*), 8
unreduced() (*in module chanpy.transducers*), 18
unsub() (*chanpy.core.pub method*), 11
unsub_all() (*chanpy.core.pub method*), 11
untap() (*chanpy.core.mult method*), 8
untap_all() (*chanpy.core.mult method*), 8

X

xiter() (*in module chanpy.transducers*), 18