PyFunctional Documentation

Release 0.6.0

Pedro Rodriguez

June 06, 2016

Welcome to the *PyFunctional* documentation. For a tutorial of how to use this package you should visit pyfunctional.org.

The documentation on this site should primarily be used as an API reference. The Streams documentation covers ways to read data into *PyFunctional* while the Transformations and Actions documentations covers the available operators.

Table of Contents

1.1 API Documentation

1.1.1 Streams API

The streams API enables you to read data into *PyFunctional*. The *seq* function imported with *from functional import seq* is actually an instance of *functional.streams.Stream*. Therefore, all the methods available on *seq* such as *seq.csv* are documented in the *Streams* class.

class `functional.streams.`**`ParallelStream`**(*processes=None,* *partition_size=None,* *disable_compression=False*)

    Bases: *`functional.streams.Stream`*

    Parallelized version of functional.streams.Stream normally accessible as *pseq*

class `functional.streams.`**`Stream`**(*disable_compression=False*)

    Bases: `object`

    Represents and implements a stream which separates the responsibilities of Sequence and ExecutionEngine.

    An instance of Stream is normally accessed as *seq*

    `csv`(*csv_file, dialect='excel', **fmt_params*)

        Reads and parses the input of a csv stream or file.

        csv_file can be a filepath or an object that implements the iterator interface (defines next() or __next__() depending on python version).

```
>>> seq.csv('examples/camping_purchases.csv').take(2)
[['1', 'tent', '300'], ['2', 'food', '100']]
```

        Parameters

            • **`csv_file`** – path to file or iterator object

            • **`dialect`** – dialect of csv, passed to csv.reader

            • **`fmt_params`** – options passed to csv.reader

        Returns Sequence wrapping csv file

    `json`(*json_file*)

        Reads and parses the input of a json file handler or file.

        Json files are parsed differently depending on if the root is a dictionary or an array.

1) If the json's root is a dictionary, these are parsed into a sequence of (Key, Value) pairs

2) If the json's root is an array, these are parsed into a sequence of entries

```
>>> seq.json('examples/users.json').first()
[u'sarah', {u'date_created': u'08/08', u'news_email': True, u'email': u'sarah@gmail.com'}]
```

> **Parameters** `json_file` – path or file containing json content
>
> **Returns** Sequence wrapping jsonl file

jsonl (*jsonl_file*)
    Reads and parses the input of a jsonl file stream or file.

    Jsonl formatted files must have a single valid json value on each line which is parsed by the python json module.

```
>>> seq.jsonl('examples/chat_logs.jsonl').first()
{u'date': u'10/09', u'message': u'hello anyone there?', u'user': u'bob'}
```

> **Parameters** `jsonl_file` – path or file containing jsonl content
>
> **Returns** Sequence wrapping jsonl file

open (*path*, *delimiter=None*, *mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*)
    Reads and parses input files as defined.

    If delimiter is not None, then the file is read in bulk then split on it. If it is None (the default), then the file is parsed as sequence of lines. The rest of the options are passed directly to builtins.open with the exception that write/append file modes is not allowed.

```
>>> seq.open('examples/gear_list.txt').take(1)
[u'tent\n']
```

> **Parameters**
>
> - `path` – path to file
> - `delimiter` – delimiter to split joined text on. if None, defaults to per line split
> - `mode` – file open mode
> - `buffering` – passed to builtins.open
> - `encoding` – passed to builtins.open
> - `errors` – passed to builtins.open
> - `newline` – passed to builtins.open
>
> **Returns** output of file depending on options wrapped in a Sequence via seq

range (**args*)
    Alias to range function where seq.range(args) is equivalent to seq(range(args)).

```
>>> seq.range(1, 8, 2)
[1, 3, 5, 7]
```

> **Parameters** `args` – args to range function
>
> **Returns** range(args) wrapped by a sequence

sqlite3(*conn*, *sql*, *parameters=None*, **args*, ***kwargs*)
Reads input by querying from a sqlite database.

```
>>> seq.sqlite3('examples/users.db', 'select id, name from users where id = 1;').first()
[(1, 'Tom')]
```

> Parameters
>
> - **conn** – path or sqlite connection, cursor
> - **sql** – SQL query string
> - **parameters** – Parameters for sql query
>
> **Returns** Sequence wrapping SQL cursor

1.1.2 Transformations and Actions API

The pipeline module contains the transformations and actions API of PyFunctional

class functional.pipeline.**Sequence**(*sequence*, *transform=None*, *engine=None*)
Bases: object

Sequence is a wrapper around any type of sequence which provides access to common functional transformations and reductions in a data pipeline style

aggregate(**args*)
Aggregates the sequence by specified arguments. Its behavior varies depending on if one, two, or three arguments are passed. Assuming the type of the sequence is A:

One Argument: argument specifies a function of the type f(current: B, next: A => result: B. current represents results computed so far, and next is the next element to aggregate into current in order to return result.

Two Argument: the first argument is the seed value for the aggregation. The second argument is the same as for the one argument case.

Three Argument: the first two arguments are the same as for one and two argument calls. The additional third parameter is a function applied to the result of the aggregation before returning the value.

> Parameters **args** – options for how to execute the aggregation
>
> **Returns** aggregated value

all()
Returns True if the truth value of all items in the sequence true.

```
>>> seq([True, True]).all()
True
```

```
>>> seq([True, False]).all()
False
```

> **Returns** True if all items truth value evaluates to True

any()
Returns True if any element in the sequence has truth value True

```
>>> seq([True, False]).any()
True
```

```
>>> seq([False, False]).any()
False
```

> **Returns** True if any element is True

average(*projection=None*)
> Takes the average of elements in the sequence

```
>>> seq([1, 2]).average()
1.5
```

```
>>> seq([('a', 1), ('b', 2)]).average(lambda x: x[1])
```

> > **Parameters** **projection** – function to project on the sequence before taking the average
>
> > **Returns** average of elements in the sequence

cache(*delete_lineage=False*)
> Caches the result of the Sequence so far. This means that any functions applied on the pipeline before cache() are evaluated, and the result is stored in the Sequence. This is primarily used internally and is no more helpful than to_list() externally. delete_lineage allows for cache() to be used in internal initialization calls without the caller having knowledge of the internals via the lineage
>
> > **Parameters** **delete_lineage** – If set to True, it will cache then erase the lineage

cartesian(**iterables*, ***kwargs*)
> Returns the cartesian product of the passed iterables with the specified number of repetitions.
>
> The keyword argument *repeat* is read from kwargs to pass to itertools.cartesian.

```
>>> seq.range(2).cartesian(range(2))
[(0, 0), (0, 1), (1, 0), (1, 1)]
```

> > **Parameters**
> >
> > * **iterables** – elements for cartesian product
> >
> > * **kwargs** – the variable *repeat* is read from kwargs
> >
> > **Returns** cartesian product

count(*func*)
> Counts the number of elements in the sequence which satisfy the predicate func.

```
>>> seq([-1, -2, 1, 2]).count(lambda x: x > 0)
2
```

> > **Parameters** **func** – predicate to count elements on
>
> > **Returns** count of elements that satisfy predicate

dict(*default=None*)
> Converts sequence of (Key, Value) pairs to a dictionary.

```
>>> type(seq([('a', 1)]).dict())
dict
```

```
>>> seq([('a', 1), ('b', 2)]).dict()
{'a': 1, 'b': 2}
```

Parameters **default** – Can be a callable zero argument function. When not None, the returned dictionary is a collections.defaultdict with default as value for missing keys. If the value is not callable, then a zero argument lambda function is created returning the value and used for collections.defaultdict

Returns dictionary from sequence of (Key, Value) elements

difference(*other*)
New sequence with unique elements present in sequence but not in other.

```
>>> seq([1, 2, 3]).difference([2, 3, 4])
[1]
```

Parameters **other** – sequence to perform difference with

Returns difference of sequence and other

distinct()
Returns sequence of distinct elements. Elements must be hashable.

```
>>> seq([1, 1, 2, 3, 3, 3, 4]).distinct()
[1, 2, 3, 4]
```

Returns sequence of distinct elements

distinct_by(*func*)
Returns sequence of elements who are distinct by the passed function. The return value of func must be hashable. When two elements are distinct by func, the first is taken.

Parameters **func** – function to use for determining distinctness

Returns elements distinct by func

drop(*n*)
Drop the first n elements of the sequence.

```
>>> seq([1, 2, 3, 4, 5]).drop(2)
[3, 4, 5]
```

Parameters **n** – number of elements to drop

Returns sequence without first n elements

drop_right(*n*)
Drops the last n elements of the sequence.

```
>>> seq([1, 2, 3, 4, 5]).drop_right(2)
[1, 2, 3]
```

Parameters **n** – number of elements to drop

Returns sequence with last n elements dropped

drop_while(*func*)
Drops elements in the sequence while func evaluates to True, then returns the rest.

```
>>> seq([1, 2, 3, 4, 5, 1, 2]).drop_while(lambda x: x < 3)
[3, 4, 5, 1, 2]
```

Parameters **func** – truth returning function

Returns elements including and after func evaluates to False

empty()
Returns True if the sequence has length zero.

```
>>> seq([]).empty()
True
```

```
>>> seq([1]).empty()
False
```

Returns True if sequence length is zero

enumerate(*start=0*)
Uses python enumerate to to zip the sequence with indexes starting at start.

```
>>> seq(['a', 'b', 'c']).enumerate(start=1)
[(1, 'a'), (2, 'b'), (3, 'c')]
```

Parameters **start** – Beginning of zip

Returns enumerated sequence starting at start

exists(*func*)
Returns True if an element in the sequence makes func evaluate to True.

```
>>> seq([1, 2, 3, 4]).exists(lambda x: x == 2)
True
```

```
>>> seq([1, 2, 3, 4]).exists(lambda x: x < 0)
False
```

Parameters **func** – existence check function

Returns True if any element satisfies func

filter(*func*)
Filters sequence to include only elements where func is True.

```
>>> seq([-1, 1, -2, 2]).filter(lambda x: x > 0)
[1, 2]
```

Parameters **func** – function to filter on

Returns filtered sequence

filter_not(*func*)
Filters sequence to include only elements where func is False.

```
>>> seq([-1, 1, -2, 2]).filter_not(lambda x: x > 0)
[-1, -2]
```

Parameters **func** – function to filter_not on

Returns filtered sequence

find(*func*)

> Finds the first element of the sequence that satisfies func. If no such element exists, then return None.

```
>>> seq(["abc", "ab", "bc"]).find(lambda x: len(x) == 2)
'ab'
```

> > **Parameters func** – function to find with
> >
> > **Returns** first element to satisfy func or None

first()

> Returns the first element of the sequence.

```
>>> seq([1, 2, 3]).first()
1
```

> Raises IndexError when the sequence is empty.

```
>>> seq([]).first()
Traceback (most recent call last):
 ...
IndexError: list index out of range
```

> > **Returns** first element of sequence

flat_map(*func*)

> Applies func to each element of the sequence, which themselves should be sequences. Then appends each element of each sequence to a final result

```
>>> seq([[1, 2], [3, 4], [5, 6]]).flat_map(lambda x: x)
[1, 2, 3, 4, 5, 6]
```

```
>>> seq(["a", "bc", "def"]).flat_map(list)
['a', 'b', 'c', 'd', 'e', 'f']
```

```
>>> seq([[1], [2], [3]]).flat_map(lambda x: x * 2)
[1, 1, 2, 2, 3, 3]
```

> > **Parameters func** – function to apply to each sequence in the sequence
> >
> > **Returns** application of func to elements followed by flattening

flatten()

> Flattens a sequence of sequences to a single sequence of elements.

```
>>> seq([[1, 2], [3, 4], [5, 6]])
[1, 2, 3, 4, 5, 6]
```

> > **Returns** flattened sequence

fold_left(*zero_value, func*)

> Assuming that the sequence elements are of type A, folds from left to right starting with the seed value given by zero_value (of type A) using a function of type func(current: B, next: A) => B. current represents the folded value so far and next is the next element from the sequence to fold into current.

```
>>> seq('a', 'b', 'c').fold_left(['start'], lambda current, next: current + [next]))
['start', 'a', 'b', 'c']
```

> **Parameters**
>
> - **zero_value** – zero value to reduce into
>
> - **func** – Two parameter function as described by function docs
>
> **Returns** value from folding values with func into zero_value from left to right.

fold_right (*zero_value*, *func*)

Assuming that the sequence elements are of type A, folds from right to left starting with the seed value given by zero_value (of type A) using a function of type func(next: A, current: B) => B. current represents the folded value so far and next is the next element from the sequence to fold into current.

```
>>> seq('a', 'b', 'c').fold_left(['start'], lambda next, current: current + [next])
['start', 'c', 'b', a']
```

> **Parameters**
>
> - **zero_value** – zero value to reduce into
>
> - **func** – Two parameter function as described by function docs
>
> **Returns** value from folding values with func into zero_value from right to left

for_all (*func*)

Returns True if all elements in sequence make func evaluate to True.

```
>>> seq([1, 2, 3]).for_all(lambda x: x > 0)
True
```

```
>>> seq([1, 2, -1]).for_all(lambda x: x > 0)
False
```

> **Parameters** **func** – function to check truth value of all elements with
>
> **Returns** True if all elements make func evaluate to True

for_each (*func*)

Executes func on each element of the sequence.

```
>>> l = []
>>> seq([1, 2, 3, 4]).for_each(l.append)
>>> l
[1, 2, 3, 4]
```

> **Parameters** **func** – function to execute

group_by (*func*)

Group elements into a list of (Key, Value) tuples where func creates the key and maps to values matching that key.

```
>>> seq(["abc", "ab", "z", "f", "qw"]).group_by(len)
[(1, ['z', 'f']), (2, ['ab', 'qw']), (3, ['abc'])]
```

> **Parameters** **func** – group by result of this function
>
> **Returns** grouped sequence

group_by_key()

Group sequence of (Key, Value) elements by Key.

```
>>> seq([('a', 1), ('b', 2), ('b', 3), ('b', 4), ('c', 3), ('c', 0)]).group_by_key()
[('a', [1]), ('c', [3, 0]), ('b', [2, 3, 4])]
```

Returns sequence grouped by key

grouped(*size*)

Partitions the elements into groups of length size.

```
>>> seq([1, 2, 3, 4, 5, 6, 7, 8]).grouped(2)
[[1, 2], [3, 4], [5, 6], [7, 8]]
```

```
>>> seq([1, 2, 3, 4, 5, 6, 7, 8]).grouped(3)
[[1, 2, 3], [4, 5, 6], [7, 8]]
```

The last partition has at least one element but may have less than size elements.

Parameters **size** – size of the partitions

Returns sequence partitioned into groups of length size

head()

Returns the first element of the sequence.

```
>>> seq([1, 2, 3]).head()
1
```

Raises IndexError when the sequence is empty.

```
>>> seq([]).head()
Traceback (most recent call last):
 ...
IndexError: list index out of range
```

Returns first element of sequence

head_option()

Returns the first element of the sequence or None, if the sequence is empty.

```
>>> seq([1, 2, 3]).head_option()
1
```

```
>>> seq([]).head_option()
None
```

Returns first element of sequence or None if sequence is empty

init()

Returns the sequence, without its last element.

```
>>> seq([1, 2, 3]).init()
[1, 2]
```

Returns sequence without last element

inits()
> Returns consecutive inits of the sequence.

```
>>> seq([1, 2, 3]).inits()
[[1, 2, 3], [1, 2], [1], []]
```

> > **Returns** consecutive init()s on sequence

inner_join(*other*)
> Sequence and other must be composed of (Key, Value) pairs. If self.sequence contains (K, V) pairs and other contains (K, W) pairs, the return result is a sequence of (K, (V, W)) pairs. Will return only elements where the key exists in both sequences.

```
>>> seq([('a', 1), ('b', 2), ('c', 3)]).inner_join([('a', 2), ('c', 5)])
[('a', (1, 2)), ('c', (3, 5))]
```

> > **Parameters other** – sequence to join with
>
> > **Returns** joined sequence of (K, (V, W)) pairs

intersection(*other*)
> New sequence with unique elements present in sequence and other.

```
>>> seq([1, 1, 2, 3]).intersection([2, 3, 4])
[2, 3]
```

> > **Parameters other** – sequence to perform intersection with
>
> > **Returns** intersection of sequence and other

join(*other*, *join_type='inner'*)
> Sequence and other must be composed of (Key, Value) pairs. If self.sequence contains (K, V) pairs and other contains (K, W) pairs, the return result is a sequence of (K, (V, W)) pairs. If join_type is "left", V values will always be present, W values may be present or None. If join_type is "right", W values will always be present, W values may be present or None. If join_type is "outer", V or W may be present or None, but never at the same time.

```
>>> seq([('a', 1), ('b', 2), ('c', 3)]).join([('a', 2), ('c', 5)], "inner")
[('a', (1, 2)), ('c', (3, 5))]
```

```
>>> seq([('a', 1), ('b', 2), ('c', 3)]).join([('a', 2), ('c', 5)])
[('a', (1, 2)), ('c', (3, 5))]
```

```
>>> seq([('a', 1), ('b', 2)]).join([('a', 3), ('c', 4)], "left")
[('a', (1, 3)), ('b', (2, None)]
```

```
>>> seq([('a', 1), ('b', 2)]).join([('a', 3), ('c', 4)], "right")
[('a', (1, 3)), ('c', (None, 4)]
```

```
>>> seq([('a', 1), ('b', 2)]).join([('a', 3), ('c', 4)], "outer")
[('a', (1, 3)), ('b', (2, None)), ('c', (None, 4))]
```

> > **Parameters**
> >
> > - **other** – sequence to join with
> >
> > - **join_type** – specifies join_type, may be "left", "right", or "outer"

> > **Returns** side joined sequence of (K, (V, W)) pairs

last()
> Returns the last element of the sequence.

```
>>> seq([1, 2, 3]).last()
3
```

> Raises IndexError when the sequence is empty.

```
>>> seq([]).last()
Traceback (most recent call last):
 ...
IndexError: list index out of range
```

> > **Returns** last element of sequence

last_option()
> Returns the last element of the sequence or None, if the sequence is empty.

```
>>> seq([1, 2, 3]).last_option()
3
```

```
>>> seq([]).last_option()
None
```

> > **Returns** last element of sequence or None if sequence is empty

left_join(*other*)
> Sequence and other must be composed of (Key, Value) pairs. If self.sequence contains (K, V) pairs and other contains (K, W) pairs, the return result is a sequence of (K, (V, W)) pairs. V values will always be present, W values may be present or None.

```
>>> seq([('a', 1), ('b', 2)]).join([('a', 3), ('c', 4)])
[('a', (1, 3)), ('b', (2, None)]
```

> > **Parameters** **other** – sequence to join with

> > **Returns** left joined sequence of (K, (V, W)) pairs

len()
> Return length of sequence using its length function.

```
>>> seq([1, 2, 3]).len()
3
```

> > **Returns** length of sequence

list()
> Converts sequence to list of elements.

```
>>> type(seq([]).list())
list
```

```
>>> type(seq([]))
functional.pipeline.Sequence
```

```
>>> seq([1, 2, 3]).list()
[1, 2, 3]
```

Returns list of elements in sequence

make_string(*separator*)

Concatenate the elements of the sequence into a string separated by separator.

```
>>> seq([1, 2, 3]).make_string("@")
'1@2@3'
```

Parameters **separator** – string separating elements in string

Returns concatenated string separated by separator

map(*func*)

Maps f onto the elements of the sequence.

```
>>> seq([1, 2, 3, 4]).map(lambda x: x * -1)
[-1, -2, -3, -4]
```

Parameters **func** – function to map with

Returns sequence with func mapped onto it

max()

Returns the largest element in the sequence. If the sequence has multiple maximal elements, only the first one is returned.

The compared objects must have defined comparison methods. Raises TypeError when the objects are not comparable.

The sequence can not be empty. Raises ValueError when the sequence is empty.

```
>>> seq([2, 4, 5, 1, 3]).max()
5
```

```
>>> seq('aa', 'xyz', 'abcd', 'xyy').max()
'xyz'
```

```
>>> seq([1, "a"]).max()
Traceback (most recent call last):
 ...
TypeError: unorderable types: int() < str()
```

```
>>> seq([]).max()
Traceback (most recent call last):
 ...
ValueError: max() arg is an empty sequence
```

Returns Maximal value of sequence

max_by(*func*)

Returns the largest element in the sequence. Provided function is used to generate key used to compare the elements. If the sequence has multiple maximal elements, only the first one is returned.

The sequence can not be empty. Raises ValueError when the sequence is empty.

```
>>> seq([2, 4, 5, 1, 3]).max_by(lambda num: num % 4)
3
```

```
>>> seq('aa', 'xyz', 'abcd', 'xyy').max_by(len)
'abcd'
```

```
>>> seq([]).max_by(lambda x: x)
Traceback (most recent call last):
 ...
ValueError: max() arg is an empty sequence
```

> **Parameters func** – function to compute max by
>
> **Returns** Maximal element by func(element)

min()

> Returns the smallest element in the sequence. If the sequence has multiple minimal elements, only the first one is returned.
>
> The compared objects must have defined comparison methods. Raises TypeError when the objects are not comparable.
>
> The sequence can not be empty. Raises ValueError when the sequence is empty.

```
>>> seq([2, 4, 5, 1, 3]).min()
1
```

```
>>> seq('aa', 'xyz', 'abcd', 'xyy').min()
'aa'
```

```
>>> seq([1, "a"]).min()
Traceback (most recent call last):
 ...
TypeError: unorderable types: int() < str()
```

```
>>> seq([]).min()
Traceback (most recent call last):
 ...
ValueError: min() arg is an empty sequence
```

> **Returns** Minimal value of sequence

min_by(*func*)

> Returns the smallest element in the sequence. Provided function is used to generate key used to compare the elements. If the sequence has multiple minimal elements, only the first one is returned.
>
> The sequence can not be empty. Raises ValueError when the sequence is empty.

```
>>> seq([2, 4, 5, 1, 3]).min_by(lambda num: num % 6)
5
```

```
>>> seq('aa', 'xyz', 'abcd', 'xyy').min_by(len)
'aa'
```

```
>>> seq([]).min_by(lambda x: x)
Traceback (most recent call last):
 ...
ValueError: min() arg is an empty sequence
```

> **Parameters func** – function to compute min by
>
> **Returns** Maximal element by func(element)

non_empty()
> Returns True if the sequence does not have length zero.

```
>>> seq([]).non_empty()
False
```

```
>>> seq([1]).non_empty()
True
```

> **Returns** True if sequence length is not zero

order_by(*func*)
> Orders the input according to func

```
>>> seq([(2, 'a'), (1, 'b'), (4, 'c'), (3, 'd')]).order_by(lambda x: x[0])
[1, 2, 3, 4]
```

> **Parameters func** – order by funciton
>
> **Returns** ordered sequence

outer_join(*other*)
> Sequence and other must be composed of (Key, Value) pairs. If self.sequence contains (K, V) pairs and other contains (K, W) pairs, the return result is a sequence of (K, (V, W)) pairs. One of V or W will always be not None, but the other may be None

```
>>> seq([('a', 1), ('b', 2)]).outer_join([('a', 3), ('c', 4)], "outer")
[('a', (1, 3)), ('b', (2, None)), ('c', (None, 4))]
```

> **Parameters other** – sequence to join with
>
> **Returns** outer joined sequence of (K, (V, W)) pairs

partition(*func*)
> Partition the sequence based on satisfying the predicate func.

```
>>> seq([-1, 1, -2, 2]).partition(lambda x: x < 0)
([-1, -2], [1, 2])
```

> **Parameters func** – predicate to partition on
>
> **Returns** tuple of partitioned sequences

product(*projection=None*)
> Takes product of elements in sequence.

```
>>> seq([1, 2, 3, 4]).product()
24
```

```
>>> seq([]).product()
1
```

```
>>> seq([(1, 2), (1, 3), (1, 4)]).product(lambda x: x[0])
1
```

> **Parameters** **projection** – function to project on the sequence before taking the product

> **Returns** product of elements in sequence

reduce(*func*)
> Reduce sequence of elements using func.

```
>>> seq([1, 2, 3]).reduce(lambda x, y: x + y)
6
```

> **Parameters** **func** – two parameter, associative reduce function

> **Returns** reduced value using func

reduce_by_key(*func*)
> Reduces a sequence of (Key, Value) using func on each sequence of values.

```
>>> seq([('a', 1), ('b', 2), ('b', 3), ('b', 4), ('c', 3), ('c', 0)])                    .reduc
[('a', 1), ('c', 3), ('b', 9)]
```

> **Parameters** **func** – reduce each list of values using two parameter, associative func

> **Returns** Sequence of tuples where the value is reduced with func

reverse()
> Returns the reversed sequence.

```
>>> seq([1, 2, 3]).reverse()
[3, 2, 1]
```

> **Returns** reversed sequence

right_join(*other*)
> Sequence and other must be composed of (Key, Value) pairs. If self.sequence contains (K, V) pairs and other contains (K, W) pairs, the return result is a sequence of (K, (V, W)) pairs. W values will always bepresent, V values may be present or None.

```
>>> seq([('a', 1), ('b', 2)]).join([('a', 3), ('c', 4)])
[('a', (1, 3)), ('b', (2, None)]
```

> **Parameters** **other** – sequence to join with

> **Returns** right joined sequence of (K, (V, W)) pairs

select(*func*)
> Selects f from the elements of the sequence.

```
>>> seq([1, 2, 3, 4]).select(lambda x: x * -1)
[-1, -2, -3, -4]
```

> **Parameters** **func** – function to select with

> **Returns** sequence with func mapped onto it

sequence
> Alias for to_list used internally for brevity

> **Returns** result of to_list() on sequence

---

set()
> Converts sequence to a set of elements.

```
>>> type(seq([])).to_set()
set
```

```
>>> type(seq([]))
functional.pipeline.Sequence
```

```
>>> seq([1, 1, 2, 2]).set()
{1, 2}
```

> :return:set of elements in sequence

size()
> Return size of sequence using its length function.

>> **Returns** size of sequence

slice(*start*, *until*)
> Takes a slice of the sequence starting at start and until but not including until.

```
>>> seq([1, 2, 3, 4]).slice(1, 2)
[2]
>>> seq([1, 2, 3, 4]).slice(1, 3)
[2, 3]
```

>> **Parameters**
>>
>> * **start** – starting index
>>
>> * **until** – ending index
>>
>> **Returns** slice including start until but not including until

sliding(*size*, *step=1*)
> Groups elements in fixed size blocks by passing a sliding window over them.
>
> The last window has at least one element but may have less than size elements

>> **Parameters**
>>
>> * **size** – size of sliding window
>>
>> * **step** – step size between windows
>>
>> **Returns** sequence of sliding windows

sorted(*key=None*, *reverse=False*)
> Uses python sort and its passed arguments to sort the input.

```
>>> seq([2, 1, 4, 3]).sorted()
[1, 2, 3, 4]
```

>> **Parameters**
>>
>> * **key** – sort using key function
>>
>> * **reverse** – return list reversed or not
>>
>> **Returns** sorted sequence

sum (*projection=None*)
    Takes sum of elements in sequence.

```
>>> seq([1, 2, 3, 4]).sum()
10
```

```
>>> seq([(1, 2), (1, 3), (1, 4)]).sum(lambda x: x[0])
3
```

> Parameters **projection** – function to project on the sequence before taking the sum
>
> Returns sum of elements in sequence

symmetric_difference (*other*)
    New sequence with elements in either sequence or other, but not both.

```
>>> seq([1, 2, 3, 3]).symmetric_difference([2, 4, 5])
[1, 3, 4, 5]
```

> Parameters **other** – sequence to perform symmetric difference with
>
> Returns symmetric difference of sequence and other

tail ()
    Returns the sequence, without its first element.

```
>>> seq([1, 2, 3]).init()
[2, 3]
```

> Returns sequence without first element

tails ()
    Returns consecutive tails of the sequence.

```
>>> seq([1, 2, 3]).tails()
[[1, 2, 3], [2, 3], [3], []]
```

> Returns consecutive tail()s of the sequence

take (*n*)
    Take the first n elements of the sequence.

```
>>> seq([1, 2, 3, 4]).take(2)
[1, 2]
```

> Parameters **n** – number of elements to take
>
> Returns first n elements of sequence

take_while (*func*)
    Take elements in the sequence until func evaluates to False, then return them.

```
>>> seq([1, 2, 3, 4, 5, 1, 2]).take_while(lambda x: x < 3)
[1, 2]
```

> Parameters **func** – truth returning function
>
> Returns elements taken until func evaluates to False

---

to_csv(*path*, *mode='wb'*, *dialect='excel'*, *compression=None*, ***fmtparams*)
Saves the sequence to a csv file. Each element should be an iterable which will be expanded to the elements of each row.

> **Parameters**
>> * **path** – path to write file
>> * **mode** – file open mode
>> * **dialect** – passed to csv.writer
>> * **fmtparams** – passed to csv.writer

to_dict(*default=None*)
Converts sequence of (Key, Value) pairs to a dictionary.

```
>>> type(seq([('a', 1)]).to_dict())
dict
```

```
>>> seq([('a', 1), ('b', 2)]).to_dict()
{'a': 1, 'b': 2}
```

> **Parameters default** – Can be a callable zero argument function. When not None, the returned dictionary is a collections.defaultdict with default as value for missing keys. If the value is not callable, then a zero argument lambda function is created returning the value and used for collections.defaultdict
>
> **Returns** dictionary from sequence of (Key, Value) elements

to_file(*path*, *delimiter=None*, *mode='wt'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*, *compresslevel=9*, *format=None*, *check=-1*, *preset=None*, *filters=None*, *compression=None*)
Saves the sequence to a file by executing str(self) which becomes str(self.to_list()). If delimiter is defined will instead execute self.make_string(delimiter)

> **Parameters**
>> * **path** – path to write file
>> * **delimiter** – if defined, will call make_string(delimiter) and save that to file.
>> * **mode** – file open mode
>> * **buffering** – passed to builtins.open
>> * **encoding** – passed to builtins.open
>> * **errors** – passed to builtins.open
>> * **newline** – passed to builtins.open
>> * **compression** – compression format
>> * **compresslevel** – passed to gzip.open
>> * **format** – passed to lzma.open
>> * **check** – passed to lzma.open
>> * **preset** – passed to lzma.open
>> * **filters** – passed to lzma.open

to_json(*path*, *root_array=True*, *mode='wb'*, *compression=None*)

    Saves the sequence to a json file. If root_array is True, then the sequence will be written to json with an array at the root. If it is False, then the sequence will be converted from a sequence of (Key, Value) pairs to a dictionary so that the json root is a dictionary.

        Parameters

- **path** – path to write file

- **root_array** – write json root as an array or dictionary

- **mode** – file open mode

to_jsonl(*path*, *mode='wb'*, *compression=None*)

    Saves the sequence to a jsonl file. Each element is mapped using json.dumps then written with a newline separating each element.

        Parameters

- **path** – path to write file

- **mode** – mode to write in, defaults to 'w' to overwrite contents

- **compression** – compression format

to_list()

    Converts sequence to list of elements.

```
>>> type(seq([]).to_list())
list
```

```
>>> type(seq([]))
functional.pipeline.Sequence
```

```
>>> seq([1, 2, 3]).to_list()
[1, 2, 3]
```

        Returns list of elements in sequence

to_pandas(*columns=None*)

    Converts sequence to a pandas DataFrame using pandas.DataFrame.from_records

        Parameters columns – columns for pandas to use

        Returns DataFrame of sequence

to_set()

    Converts sequence to a set of elements.

```
>>> type(seq([])).to_set()
set
```

```
>>> type(seq([]))
functional.pipeline.Sequence
```

```
>>> seq([1, 1, 2, 2]).to_set()
{1, 2}
```

    :return:set of elements in sequence

to_sqlite3(*conn*, *target*, **args*, ***kwargs*)

    Saves the sequence to sqlite3 database. Target table must be created in advance. The table schema is inferred from the elements in the sequence if only target table name is supplied.

```
>>> seq([(1, 'Tom'), (2, 'Jack')])                    .to_sqlite3('users.db', 'INSERT INTO user
```

```
>>> seq([{'id': 1, 'name': 'Tom'}, {'id': 2, 'name': 'Jack'}]).to_sqlite3(conn, 'user')
```

> Parameters
>
> - **conn** – path or sqlite connection, cursor
> - **target** – SQL query string or table name
> - **args** – passed to sqlite3.connect
> - **kwargs** – passed to sqlite3.connect

union(*other*)
> New sequence with unique elements from self and other.

```
>>> seq([1, 1, 2, 3, 3]).union([1, 4, 5])
[1, 2, 3, 4, 5]
```

> Parameters **other** – sequence to union with
>
> Returns union of sequence and other

where(*func*)
> Selects elements where func evaluates to True.

```
>>> seq([-1, 1, -2, 2]).where(lambda x: x > 0)
[1, 2]
```

> Parameters **func** – function to filter on
>
> Returns filtered sequence

zip(*sequence*)
> Zips the stored sequence with the given sequence.

```
>>> seq([1, 2, 3]).zip([4, 5, 6])
[(1, 4), (2, 5), (3, 6)]
```

> Parameters **sequence** – second sequence to zip
>
> Returns stored sequence zipped with given sequence

zip_with_index(*start=0*)
> Zips the sequence to its index, with the index being the second element of each tuple.

```
>>> seq(['a', 'b', 'c']).zip_with_index()
[('a', 0), ('b', 1), ('c', 2)]
```

> Returns sequence zipped to its index

1.2 Developer Documentation

1.2.1 functional.streams

class `functional.streams.`**`ParallelStream`**(*processes=None*, *partition_size=None*, *disable_compression=False*)

 Bases: *functional.streams.Stream*

 Parallelized version of functional.streams.Stream normally accessible as *pseq*

 `__call__`(**args*, ***kwargs*)
 Create a Sequence using a parallel ExecutionEngine.

 If args has more than one argument then the argument list becomes the sequence.

 If args[0] is primitive, a sequence wrapping it is created.

 If args[0] is a list, tuple, iterable, or Sequence it is wrapped as a Sequence.

 Parameters **args** – Sequence to wrap

 Returns Wrapped sequence

 `__init__`(*processes=None*, *partition_size=None*, *disable_compression=False*)
 Configure Stream for parallel processing and file compression detection :param processes: Number of parallel processes :param disable_compression: Disable file compression detection

 `__module__` = 'functional.streams'

class `functional.streams.`**`Stream`**(*disable_compression=False*)
 Bases: `object`

 Represents and implements a stream which separates the responsibilities of Sequence and ExecutionEngine.

 An instance of Stream is normally accessed as *seq*

 `__call__`(**args*)
 Create a Sequence using a sequential ExecutionEngine.

 If args has more than one argument then the argument list becomes the sequence.

 If args[0] is primitive, a sequence wrapping it is created.

 If args[0] is a list, tuple, iterable, or Sequence it is wrapped as a Sequence.

 Parameters **args** – Sequence to wrap

 Returns Wrapped sequence

 `__dict__` = dict_proxy({'__module__': 'functional.streams', '__dict__': <attribute '__dict__' of 'Stream' objects>, '__w

 `__init__`(*disable_compression=False*)
 Default stream constructor. :param disable_compression: Disable file compression detection

 `__module__` = 'functional.streams'

 `__weakref__`
 list of weak references to the object (if defined)

 `csv`(*csv_file*, *dialect='excel'*, ***fmt_params*)
 Reads and parses the input of a csv stream or file.

 csv_file can be a filepath or an object that implements the iterator interface (defines next() or __next__() depending on python version).

```
>>> seq.csv('examples/camping_purchases.csv').take(2)
[['1', 'tent', '300'], ['2', 'food', '100']]
```

> **Parameters**
>
> > • **csv_file** – path to file or iterator object
> >
> > • **dialect** – dialect of csv, passed to csv.reader
> >
> > • **fmt_params** – options passed to csv.reader
>
> **Returns** Sequence wrapping csv file

json(*json_file*)

> Reads and parses the input of a json file handler or file.
>
> Json files are parsed differently depending on if the root is a dictionary or an array.
>
> 1) If the json's root is a dictionary, these are parsed into a sequence of (Key, Value) pairs
>
> 2) If the json's root is an array, these are parsed into a sequence of entries

```
>>> seq.json('examples/users.json').first()
[u'sarah', {u'date_created': u'08/08', u'news_email': True, u'email': u'sarah@gmail.com'}]
```

> **Parameters** **json_file** – path or file containing json content
>
> **Returns** Sequence wrapping jsonl file

jsonl(*jsonl_file*)

> Reads and parses the input of a jsonl file stream or file.
>
> Jsonl formatted files must have a single valid json value on each line which is parsed by the python json module.

```
>>> seq.jsonl('examples/chat_logs.jsonl').first()
{u'date': u'10/09', u'message': u'hello anyone there?', u'user': u'bob'}
```

> **Parameters** **jsonl_file** – path or file containing jsonl content
>
> **Returns** Sequence wrapping jsonl file

open(*path*, *delimiter=None*, *mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*)

> Reads and parses input files as defined.
>
> If delimiter is not None, then the file is read in bulk then split on it. If it is None (the default), then the file is parsed as sequence of lines. The rest of the options are passed directly to builtins.open with the exception that write/append file modes is not allowed.

```
>>> seq.open('examples/gear_list.txt').take(1)
[u'tent\n']
```

> **Parameters**
>
> > • **path** – path to file
> >
> > • **delimiter** – delimiter to split joined text on. if None, defaults to per line split
> >
> > • **mode** – file open mode
> >
> > • **buffering** – passed to builtins.open

- **encoding** – passed to builtins.open

- **errors** – passed to builtins.open

- **newline** – passed to builtins.open

Returns output of file depending on options wrapped in a Sequence via seq

range(**args*)

Alias to range function where seq.range(args) is equivalent to seq(range(args)).

```
>>> seq.range(1, 8, 2)
[1, 3, 5, 7]
```

Parameters **args** – args to range function

Returns range(args) wrapped by a sequence

sqlite3(*conn*, *sql*, *parameters=None*, **args*, ***kwargs*)

Reads input by querying from a sqlite database.

```
>>> seq.sqlite3('examples/users.db', 'select id, name from users where id = 1;').first()
[(1, 'Tom')]
```

Parameters

- **conn** – path or sqlite connection, cursor

- **sql** – SQL query string

- **parameters** – Parameters for sql query

Returns Sequence wrapping SQL cursor

1.2.2 functional.pipeline

The pipeline module contains the transformations and actions API of PyFunctional

class functional.pipeline.**Sequence**(*sequence*, *transform=None*, *engine=None*)

Bases: object

Sequence is a wrapper around any type of sequence which provides access to common functional transformations and reductions in a data pipeline style

__add__(*other*)

Concatenates sequence with other.

Parameters **other** – sequence to concatenate

Returns concatenated sequence with other

__bool__()

Returns True if size is not zero.

Returns True if size is not zero

__contains__(*item*)

Checks if item is in sequence.

Parameters **item** – item to check

Returns True if item is in sequence

__dict__ = dict_proxy({'all': <function all at 0x7f1e41e12230>, '_to_sqlite3_by_table': <function _to_sqlite3_by_table a

__eq__(*other*)
> Checks for equality with the sequence's equality operator.

>> **Parameters** **other** – object to compare to

>> **Returns** true if the underlying sequence is equal to other

__getitem__(*item*)
> Gets item at given index.

>> **Parameters** **item** – key to use for getitem

>> **Returns** item at index key

__hash__()
> Return the hash of the sequence.

>> **Returns** hash of sequence

__init__(*sequence*, *transform=None*, *engine=None*)
> Takes a Sequence, list, tuple. or iterable sequence and wraps it around a Sequence object. If the sequence is already an instance of Sequence, it will in total be wrapped exactly once. A TypeError is raised if sequence is none of these.

>> **Parameters** **sequence** – sequence of items to wrap in a Sequence

>> **Returns** sequence wrapped in a Sequence

__iter__()
> Return iterator of sequence.

>> **Returns** iterator of sequence

__module__ = 'functional.pipeline'

__ne__(*other*)
> Checks for inequality with the sequence's inequality operator.

>> **Parameters** **other** – object to compare to

>> **Returns** true if the underlying sequence is not equal to other

__nonzero__()
> Returns True if size is not zero.

>> **Returns** True if size is not zero

__repr__()
> Return repr using sequence's repr function.

>> **Returns** sequence's repr

__reversed__()
> Return reversed sequence using sequence's reverse function

>> **Returns** reversed sequence

__str__()
> Return string using sequence's string function.

>> **Returns** sequence's string

__weakref__
> list of weak references to the object (if defined)

_evaluate()
> Creates and returns an iterator which applies all the transformations in the lineage

>> **Returns** iterator over the transformed sequence

_to_sqlite3_by_query(*conn*, *sql*)
> Saves the sequence to sqlite3 database by supplied query. Each element should be an iterable which will be expanded to the elements of each row. Target table must be created in advance.

>> **Parameters**

>>> • **conn** – path or sqlite connection, cursor

>>> • **sql** – SQL query string

_to_sqlite3_by_table(*conn*, *table_name*)
> Saves the sequence to the specified table of sqlite3 database. Each element can be a dictionary, namedtuple, tuple or list. Target table must be created in advance.

>> **Parameters**

>>> • **conn** – path or sqlite connection, cursor

>>> • **table_name** – table name string

_transform(**transforms*)
> Copies the given Sequence and appends new transformation :param transform: transform to apply or list of transforms to apply :return: transformed sequence

aggregate(**args*)
> Aggregates the sequence by specified arguments. Its behavior varies depending on if one, two, or three arguments are passed. Assuming the type of the sequence is A:

> One Argument: argument specifies a function of the type f(current: B, next: A => result: B. current represents results computed so far, and next is the next element to aggregate into current in order to return result.

> Two Argument: the first argument is the seed value for the aggregation. The second argument is the same as for the one argument case.

> Three Argument: the first two arguments are the same as for one and two argument calls. The additional third parameter is a function applied to the result of the aggregation before returning the value.

>> **Parameters** **args** – options for how to execute the aggregation

>> **Returns** aggregated value

all()
> Returns True if the truth value of all items in the sequence true.

```
>>> seq([True, True]).all()
True
```

```
>>> seq([True, False]).all()
False
```

>> **Returns** True if all items truth value evaluates to True

any()
> Returns True if any element in the sequence has truth value True

```
>>> seq([True, False]).any()
True
```

```
>>> seq([False, False]).any()
False
```

> **Returns** True if any element is True

average(*projection=None*)
> Takes the average of elements in the sequence

```
>>> seq([1, 2]).average()
1.5
```

```
>>> seq([('a', 1), ('b', 2)]).average(lambda x: x[1])
```

> > **Parameters projection** – function to project on the sequence before taking the average
>
> > **Returns** average of elements in the sequence

cache(*delete_lineage=False*)
> Caches the result of the Sequence so far. This means that any functions applied on the pipeline before cache() are evaluated, and the result is stored in the Sequence. This is primarily used internally and is no more helpful than to_list() externally. delete_lineage allows for cache() to be used in internal initialization calls without the caller having knowledge of the internals via the lineage
>
> > **Parameters delete_lineage** – If set to True, it will cache then erase the lineage

cartesian(**iterables*, ***kwargs*)
> Returns the cartesian product of the passed iterables with the specified number of repetitions.
>
> The keyword argument *repeat* is read from kwargs to pass to itertools.cartesian.

```
>>> seq.range(2).cartesian(range(2))
[(0, 0), (0, 1), (1, 0), (1, 1)]
```

> > **Parameters**
> >
> > > • **iterables** – elements for cartesian product
> > >
> > > • **kwargs** – the variable *repeat* is read from kwargs
> >
> > **Returns** cartesian product

count(*func*)
> Counts the number of elements in the sequence which satisfy the predicate func.

```
>>> seq([-1, -2, 1, 2]).count(lambda x: x > 0)
2
```

> > **Parameters func** – predicate to count elements on
>
> > **Returns** count of elements that satisfy predicate

dict(*default=None*)
> Converts sequence of (Key, Value) pairs to a dictionary.

```
>>> type(seq([('a', 1)]).dict())
dict
```

```
>>> seq([('a', 1), ('b', 2)]).dict()
{'a': 1, 'b': 2}
```

Parameters default – Can be a callable zero argument function. When not None, the returned dictionary is a collections.defaultdict with default as value for missing keys. If the value is not callable, then a zero argument lambda function is created returning the value and used for collections.defaultdict

Returns dictionary from sequence of (Key, Value) elements

difference(*other*)
New sequence with unique elements present in sequence but not in other.

```
>>> seq([1, 2, 3]).difference([2, 3, 4])
[1]
```

Parameters other – sequence to perform difference with

Returns difference of sequence and other

distinct()
Returns sequence of distinct elements. Elements must be hashable.

```
>>> seq([1, 1, 2, 3, 3, 3, 4]).distinct()
[1, 2, 3, 4]
```

Returns sequence of distinct elements

distinct_by(*func*)
Returns sequence of elements who are distinct by the passed function. The return value of func must be hashable. When two elements are distinct by func, the first is taken.

Parameters func – function to use for determining distinctness

Returns elements distinct by func

drop(*n*)
Drop the first n elements of the sequence.

```
>>> seq([1, 2, 3, 4, 5]).drop(2)
[3, 4, 5]
```

Parameters n – number of elements to drop

Returns sequence without first n elements

drop_right(*n*)
Drops the last n elements of the sequence.

```
>>> seq([1, 2, 3, 4, 5]).drop_right(2)
[1, 2, 3]
```

Parameters n – number of elements to drop

Returns sequence with last n elements dropped

drop_while(*func*)
Drops elements in the sequence while func evaluates to True, then returns the rest.

```
>>> seq([1, 2, 3, 4, 5, 1, 2]).drop_while(lambda x: x < 3)
[3, 4, 5, 1, 2]
```

> **Parameters** `func` – truth returning function

> **Returns** elements including and after func evaluates to False

empty()
    Returns True if the sequence has length zero.

```
>>> seq([]).empty()
True
```

```
>>> seq([1]).empty()
False
```

> **Returns** True if sequence length is zero

enumerate(*start=0*)
    Uses python enumerate to to zip the sequence with indexes starting at start.

```
>>> seq(['a', 'b', 'c']).enumerate(start=1)
[(1, 'a'), (2, 'b'), (3, 'c')]
```

> **Parameters** `start` – Beginning of zip

> **Returns** enumerated sequence starting at start

exists(*func*)
    Returns True if an element in the sequence makes func evaluate to True.

```
>>> seq([1, 2, 3, 4]).exists(lambda x: x == 2)
True
```

```
>>> seq([1, 2, 3, 4]).exists(lambda x: x < 0)
False
```

> **Parameters** `func` – existence check function

> **Returns** True if any element satisfies func

filter(*func*)
    Filters sequence to include only elements where func is True.

```
>>> seq([-1, 1, -2, 2]).filter(lambda x: x > 0)
[1, 2]
```

> **Parameters** `func` – function to filter on

> **Returns** filtered sequence

filter_not(*func*)
    Filters sequence to include only elements where func is False.

```
>>> seq([-1, 1, -2, 2]).filter_not(lambda x: x > 0)
[-1, -2]
```

> **Parameters** `func` – function to filter_not on

> **Returns** filtered sequence

find (*func*)

Finds the first element of the sequence that satisfies func. If no such element exists, then return None.

```
>>> seq(["abc", "ab", "bc"]).find(lambda x: len(x) == 2)
'ab'
```

Parameters **func** – function to find with

Returns first element to satisfy func or None

first ()

Returns the first element of the sequence.

```
>>> seq([1, 2, 3]).first()
1
```

Raises IndexError when the sequence is empty.

```
>>> seq([]).first()
Traceback (most recent call last):
 ...
IndexError: list index out of range
```

Returns first element of sequence

flat_map (*func*)

Applies func to each element of the sequence, which themselves should be sequences. Then appends each element of each sequence to a final result

```
>>> seq([[1, 2], [3, 4], [5, 6]]).flat_map(lambda x: x)
[1, 2, 3, 4, 5, 6]
```

```
>>> seq(["a", "bc", "def"]).flat_map(list)
['a', 'b', 'c', 'd', 'e', 'f']
```

```
>>> seq([[1], [2], [3]]).flat_map(lambda x: x * 2)
[1, 1, 2, 2, 3, 3]
```

Parameters **func** – function to apply to each sequence in the sequence

Returns application of func to elements followed by flattening

flatten ()

Flattens a sequence of sequences to a single sequence of elements.

```
>>> seq([[1, 2], [3, 4], [5, 6]])
[1, 2, 3, 4, 5, 6]
```

Returns flattened sequence

fold_left (*zero_value*, *func*)

Assuming that the sequence elements are of type A, folds from left to right starting with the seed value given by zero_value (of type A) using a function of type func(current: B, next: A) => B. current represents the folded value so far and next is the next element from the sequence to fold into current.

```
>>> seq('a', 'b', 'c').fold_left(['start'], lambda current, next: current + [next]))
['start', 'a', 'b', 'c']
```

> **Parameters**
>
> - **zero_value** – zero value to reduce into
> - **func** – Two parameter function as described by function docs
>
> **Returns** value from folding values with func into zero_value from left to right.

fold_right(*zero_value*, *func*)

Assuming that the sequence elements are of type A, folds from right to left starting with the seed value given by zero_value (of type A) using a function of type func(next: A, current: B) => B. current represents the folded value so far and next is the next element from the sequence to fold into current.

```
>>> seq('a', 'b', 'c').fold_left(['start'], lambda next, current: current + [next])
['start', 'c', 'b', a']
```

> **Parameters**
>
> - **zero_value** – zero value to reduce into
> - **func** – Two parameter function as described by function docs
>
> **Returns** value from folding values with func into zero_value from right to left

for_all(*func*)

Returns True if all elements in sequence make func evaluate to True.

```
>>> seq([1, 2, 3]).for_all(lambda x: x > 0)
True
```

```
>>> seq([1, 2, -1]).for_all(lambda x: x > 0)
False
```

> **Parameters** **func** – function to check truth value of all elements with
>
> **Returns** True if all elements make func evaluate to True

for_each(*func*)

Executes func on each element of the sequence.

```
>>> l = []
>>> seq([1, 2, 3, 4]).for_each(l.append)
>>> l
[1, 2, 3, 4]
```

> **Parameters** **func** – function to execute

group_by(*func*)

Group elements into a list of (Key, Value) tuples where func creates the key and maps to values matching that key.

```
>>> seq(["abc", "ab", "z", "f", "qw"]).group_by(len)
[(1, ['z', 'f']), (2, ['ab', 'qw']), (3, ['abc'])]
```

> **Parameters** **func** – group by result of this function
>
> **Returns** grouped sequence

group_by_key()
> Group sequence of (Key, Value) elements by Key.

```
>>> seq([('a', 1), ('b', 2), ('b', 3), ('b', 4), ('c', 3), ('c', 0)]).group_by_key()
[('a', [1]), ('c', [3, 0]), ('b', [2, 3, 4])]
```

> > **Returns** sequence grouped by key

grouped(*size*)
> Partitions the elements into groups of length size.

```
>>> seq([1, 2, 3, 4, 5, 6, 7, 8]).grouped(2)
[[1, 2], [3, 4], [5, 6], [7, 8]]
```

```
>>> seq([1, 2, 3, 4, 5, 6, 7, 8]).grouped(3)
[[1, 2, 3], [4, 5, 6], [7, 8]]
```

> The last partition has at least one element but may have less than size elements.

> > **Parameters** **size** – size of the partitions

> > **Returns** sequence partitioned into groups of length size

head()
> Returns the first element of the sequence.

```
>>> seq([1, 2, 3]).head()
1
```

> Raises IndexError when the sequence is empty.

```
>>> seq([]).head()
Traceback (most recent call last):
 ...
IndexError: list index out of range
```

> > **Returns** first element of sequence

head_option()
> Returns the first element of the sequence or None, if the sequence is empty.

```
>>> seq([1, 2, 3]).head_option()
1
```

```
>>> seq([]).head_option()
None
```

> > **Returns** first element of sequence or None if sequence is empty

init()
> Returns the sequence, without its last element.

```
>>> seq([1, 2, 3]).init()
[1, 2]
```

> > **Returns** sequence without last element

inits()
>   Returns consecutive inits of the sequence.

```
>>> seq([1, 2, 3]).inits()
[[1, 2, 3], [1, 2], [1], []]
```

>   **Returns** consecutive init()s on sequence

inner_join(*other*)
>   Sequence and other must be composed of (Key, Value) pairs. If self.sequence contains (K, V) pairs and other contains (K, W) pairs, the return result is a sequence of (K, (V, W)) pairs. Will return only elements where the key exists in both sequences.

```
>>> seq([('a', 1), ('b', 2), ('c', 3)]).inner_join([('a', 2), ('c', 5)])
[('a', (1, 2)), ('c', (3, 5))]
```

>   **Parameters other** – sequence to join with
>
>   **Returns** joined sequence of (K, (V, W)) pairs

intersection(*other*)
>   New sequence with unique elements present in sequence and other.

```
>>> seq([1, 1, 2, 3]).intersection([2, 3, 4])
[2, 3]
```

>   **Parameters other** – sequence to perform intersection with
>
>   **Returns** intersection of sequence and other

join(*other*, *join_type='inner'*)
>   Sequence and other must be composed of (Key, Value) pairs. If self.sequence contains (K, V) pairs and other contains (K, W) pairs, the return result is a sequence of (K, (V, W)) pairs. If join_type is "left", V values will always be present, W values may be present or None. If join_type is "right", W values will always be present, W values may be present or None. If join_type is "outer", V or W may be present or None, but never at the same time.

```
>>> seq([('a', 1), ('b', 2), ('c', 3)]).join([('a', 2), ('c', 5)], "inner")
[('a', (1, 2)), ('c', (3, 5))]
```

```
>>> seq([('a', 1), ('b', 2), ('c', 3)]).join([('a', 2), ('c', 5)])
[('a', (1, 2)), ('c', (3, 5))]
```

```
>>> seq([('a', 1), ('b', 2)]).join([('a', 3), ('c', 4)], "left")
[('a', (1, 3)), ('b', (2, None)]
```

```
>>> seq([('a', 1), ('b', 2)]).join([('a', 3), ('c', 4)], "right")
[('a', (1, 3)), ('c', (None, 4)]
```

```
>>> seq([('a', 1), ('b', 2)]).join([('a', 3), ('c', 4)], "outer")
[('a', (1, 3)), ('b', (2, None)), ('c', (None, 4))]
```

>   **Parameters**
>
>   • **other** – sequence to join with
>
>   • **join_type** – specifies join_type, may be "left", "right", or "outer"

Returns side joined sequence of (K, (V, W)) pairs

last()
Returns the last element of the sequence.

```
>>> seq([1, 2, 3]).last()
3
```

Raises IndexError when the sequence is empty.

```
>>> seq([]).last()
Traceback (most recent call last):
 ...
IndexError: list index out of range
```

Returns last element of sequence

last_option()
Returns the last element of the sequence or None, if the sequence is empty.

```
>>> seq([1, 2, 3]).last_option()
3
```

```
>>> seq([]).last_option()
None
```

Returns last element of sequence or None if sequence is empty

left_join(*other*)
Sequence and other must be composed of (Key, Value) pairs. If self.sequence contains (K, V) pairs and other contains (K, W) pairs, the return result is a sequence of (K, (V, W)) pairs. V values will always be present, W values may be present or None.

```
>>> seq([('a', 1), ('b', 2)]).join([('a', 3), ('c', 4)])
[('a', (1, 3)), ('b', (2, None)]
```

Parameters `other` – sequence to join with

Returns left joined sequence of (K, (V, W)) pairs

len()
Return length of sequence using its length function.

```
>>> seq([1, 2, 3]).len()
3
```

Returns length of sequence

list()
Converts sequence to list of elements.

```
>>> type(seq([]).list())
list
```

```
>>> type(seq([]))
functional.pipeline.Sequence
```

```
>>> seq([1, 2, 3]).list()
[1, 2, 3]
```

> **Returns** list of elements in sequence

make_string(*separator*)

    Concatenate the elements of the sequence into a string separated by separator.

```
>>> seq([1, 2, 3]).make_string("@")
'1@2@3'
```

> **Parameters** **separator** – string separating elements in string
>
> **Returns** concatenated string separated by separator

map(*func*)

    Maps f onto the elements of the sequence.

```
>>> seq([1, 2, 3, 4]).map(lambda x: x * -1)
[-1, -2, -3, -4]
```

> **Parameters** **func** – function to map with
>
> **Returns** sequence with func mapped onto it

max()

    Returns the largest element in the sequence. If the sequence has multiple maximal elements, only the first one is returned.

    The compared objects must have defined comparison methods. Raises TypeError when the objects are not comparable.

    The sequence can not be empty. Raises ValueError when the sequence is empty.

```
>>> seq([2, 4, 5, 1, 3]).max()
5
```

```
>>> seq('aa', 'xyz', 'abcd', 'xyy').max()
'xyz'
```

```
>>> seq([1, "a"]).max()
Traceback (most recent call last):
 ...
TypeError: unorderable types: int() < str()
```

```
>>> seq([]).max()
Traceback (most recent call last):
 ...
ValueError: max() arg is an empty sequence
```

> **Returns** Maximal value of sequence

max_by(*func*)

    Returns the largest element in the sequence. Provided function is used to generate key used to compare the elements. If the sequence has multiple maximal elements, only the first one is returned.

    The sequence can not be empty. Raises ValueError when the sequence is empty.

```
>>> seq([2, 4, 5, 1, 3]).max_by(lambda num: num % 4)
3
```

```
>>> seq('aa', 'xyz', 'abcd', 'xyy').max_by(len)
'abcd'
```

```
>>> seq([]).max_by(lambda x: x)
Traceback (most recent call last):
 ...
ValueError: max() arg is an empty sequence
```

>   **Parameters func** – function to compute max by
>
>   **Returns** Maximal element by func(element)

min()

Returns the smallest element in the sequence. If the sequence has multiple minimal elements, only the first one is returned.

The compared objects must have defined comparison methods. Raises TypeError when the objects are not comparable.

The sequence can not be empty. Raises ValueError when the sequence is empty.

```
>>> seq([2, 4, 5, 1, 3]).min()
1
```

```
>>> seq('aa', 'xyz', 'abcd', 'xyy').min()
'aa'
```

```
>>> seq([1, "a"]).min()
Traceback (most recent call last):
 ...
TypeError: unorderable types: int() < str()
```

```
>>> seq([]).min()
Traceback (most recent call last):
 ...
ValueError: min() arg is an empty sequence
```

>   **Returns** Minimal value of sequence

min_by(*func*)

Returns the smallest element in the sequence. Provided function is used to generate key used to compare the elements. If the sequence has multiple minimal elements, only the first one is returned.

The sequence can not be empty. Raises ValueError when the sequence is empty.

```
>>> seq([2, 4, 5, 1, 3]).min_by(lambda num: num % 6)
5
```

```
>>> seq('aa', 'xyz', 'abcd', 'xyy').min_by(len)
'aa'
```

```
>>> seq([]).min_by(lambda x: x)
Traceback (most recent call last):
 ...
ValueError: min() arg is an empty sequence
```

> **Parameters** `func` – function to compute min by
>
> **Returns** Maximal element by func(element)

non_empty()
    Returns True if the sequence does not have length zero.

```
>>> seq([]).non_empty()
False
```

```
>>> seq([1]).non_empty()
True
```

> **Returns** True if sequence length is not zero

order_by(*func*)
    Orders the input according to func

```
>>> seq([(2, 'a'), (1, 'b'), (4, 'c'), (3, 'd')]).order_by(lambda x: x[0])
[1, 2, 3, 4]
```

> **Parameters** `func` – order by funciton
>
> **Returns** ordered sequence

outer_join(*other*)
    Sequence and other must be composed of (Key, Value) pairs. If self.sequence contains (K, V) pairs and
    other contains (K, W) pairs, the return result is a sequence of (K, (V, W)) pairs. One of V or W will always
    be not None, but the other may be None

```
>>> seq([('a', 1), ('b', 2)]).outer_join([('a', 3), ('c', 4)], "outer")
[('a', (1, 3)), ('b', (2, None)), ('c', (None, 4))]
```

> **Parameters** `other` – sequence to join with
>
> **Returns** outer joined sequence of (K, (V, W)) pairs

partition(*func*)
    Partition the sequence based on satisfying the predicate func.

```
>>> seq([-1, 1, -2, 2]).partition(lambda x: x < 0)
([-1, -2], [1, 2])
```

> **Parameters** `func` – predicate to partition on
>
> **Returns** tuple of partitioned sequences

product(*projection=None*)
    Takes product of elements in sequence.

```
>>> seq([1, 2, 3, 4]).product()
24
```

```
>>> seq([]).product()
1
```

```
>>> seq([(1, 2), (1, 3), (1, 4)]).product(lambda x: x[0])
1
```

> Parameters **projection** – function to project on the sequence before taking the product

> Returns product of elements in sequence

reduce(*func*)
> Reduce sequence of elements using func.

```
>>> seq([1, 2, 3]).reduce(lambda x, y: x + y)
6
```

> Parameters **func** – two parameter, associative reduce function

> Returns reduced value using func

reduce_by_key(*func*)
> Reduces a sequence of (Key, Value) using func on each sequence of values.

```
>>> seq([('a', 1), ('b', 2), ('b', 3), ('b', 4), ('c', 3), ('c', 0)])        .reduc
[('a', 1), ('c', 3), ('b', 9)]
```

> Parameters **func** – reduce each list of values using two parameter, associative func

> Returns Sequence of tuples where the value is reduced with func

reverse()
> Returns the reversed sequence.

```
>>> seq([1, 2, 3]).reverse()
[3, 2, 1]
```

> Returns reversed sequence

right_join(*other*)
> Sequence and other must be composed of (Key, Value) pairs. If self.sequence contains (K, V) pairs and other contains (K, W) pairs, the return result is a sequence of (K, (V, W)) pairs. W values will always bepresent, V values may be present or None.

```
>>> seq([('a', 1), ('b', 2)]).join([('a', 3), ('c', 4)])
[('a', (1, 3)), ('b', (2, None)]
```

> Parameters **other** – sequence to join with

> Returns right joined sequence of (K, (V, W)) pairs

select(*func*)
> Selects f from the elements of the sequence.

```
>>> seq([1, 2, 3, 4]).select(lambda x: x * -1)
[-1, -2, -3, -4]
```

> Parameters **func** – function to select with

> Returns sequence with func mapped onto it

sequence
> Alias for to_list used internally for brevity

> Returns result of to_list() on sequence

---

set()
> Converts sequence to a set of elements.

```
>>> type(seq([])).to_set()
set
```

```
>>> type(seq([]))
functional.pipeline.Sequence
```

```
>>> seq([1, 1, 2, 2]).set()
{1, 2}
```

> :return:set of elements in sequence

size()
> Return size of sequence using its length function.

> > **Returns** size of sequence

slice(*start*, *until*)
> Takes a slice of the sequence starting at start and until but not including until.

```
>>> seq([1, 2, 3, 4]).slice(1, 2)
[2]
>>> seq([1, 2, 3, 4]).slice(1, 3)
[2, 3]
```

> > **Parameters**
> >
> > - **start** – starting index
> >
> > - **until** – ending index
> >
> > **Returns** slice including start until but not including until

sliding(*size*, *step=1*)
> Groups elements in fixed size blocks by passing a sliding window over them.

> The last window has at least one element but may have less than size elements

> > **Parameters**
> >
> > - **size** – size of sliding window
> >
> > - **step** – step size between windows
> >
> > **Returns** sequence of sliding windows

sorted(*key=None*, *reverse=False*)
> Uses python sort and its passed arguments to sort the input.

```
>>> seq([2, 1, 4, 3]).sorted()
[1, 2, 3, 4]
```

> > **Parameters**
> >
> > - **key** – sort using key function
> >
> > - **reverse** – return list reversed or not
> >
> > **Returns** sorted sequence

sum(*projection=None*)
    Takes sum of elements in sequence.

```
>>> seq([1, 2, 3, 4]).sum()
10
```

```
>>> seq([(1, 2), (1, 3), (1, 4)]).sum(lambda x: x[0])
3
```

        Parameters projection – function to project on the sequence before taking the sum

        Returns sum of elements in sequence

symmetric_difference(*other*)
    New sequence with elements in either sequence or other, but not both.

```
>>> seq([1, 2, 3, 3]).symmetric_difference([2, 4, 5])
[1, 3, 4, 5]
```

        Parameters other – sequence to perform symmetric difference with

        Returns symmetric difference of sequence and other

tail()
    Returns the sequence, without its first element.

```
>>> seq([1, 2, 3]).init()
[2, 3]
```

        Returns sequence without first element

tails()
    Returns consecutive tails of the sequence.

```
>>> seq([1, 2, 3]).tails()
[[1, 2, 3], [2, 3], [3], []]
```

        Returns consecutive tail()s of the sequence

take(*n*)
    Take the first n elements of the sequence.

```
>>> seq([1, 2, 3, 4]).take(2)
[1, 2]
```

        Parameters n – number of elements to take

        Returns first n elements of sequence

take_while(*func*)
    Take elements in the sequence until func evaluates to False, then return them.

```
>>> seq([1, 2, 3, 4, 5, 1, 2]).take_while(lambda x: x < 3)
[1, 2]
```

        Parameters func – truth returning function

        Returns elements taken until func evaluates to False

to_csv (*path*, *mode='wb'*, *dialect='excel'*, *compression=None*, ***fmtparams*)
    Saves the sequence to a csv file. Each element should be an iterable which will be expanded to the elements of each row.

> **Parameters**
>> - **path** – path to write file
>> - **mode** – file open mode
>> - **dialect** – passed to csv.writer
>> - **fmtparams** – passed to csv.writer

to_dict (*default=None*)
    Converts sequence of (Key, Value) pairs to a dictionary.

```
>>> type(seq([('a', 1)]).to_dict())
dict
```

```
>>> seq([('a', 1), ('b', 2)]).to_dict()
{'a': 1, 'b': 2}
```

> **Parameters default** – Can be a callable zero argument function. When not None, the returned dictionary is a collections.defaultdict with default as value for missing keys. If the value is not callable, then a zero argument lambda function is created returning the value and used for collections.defaultdict
>
> **Returns** dictionary from sequence of (Key, Value) elements

to_file (*path*, *delimiter=None*, *mode='wt'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*, *compresslevel=9*, *format=None*, *check=-1*, *preset=None*, *filters=None*, *compression=None*)
    Saves the sequence to a file by executing str(self) which becomes str(self.to_list()). If delimiter is defined will instead execute self.make_string(delimiter)

> **Parameters**
>> - **path** – path to write file
>> - **delimiter** – if defined, will call make_string(delimiter) and save that to file.
>> - **mode** – file open mode
>> - **buffering** – passed to builtins.open
>> - **encoding** – passed to builtins.open
>> - **errors** – passed to builtins.open
>> - **newline** – passed to builtins.open
>> - **compression** – compression format
>> - **compresslevel** – passed to gzip.open
>> - **format** – passed to lzma.open
>> - **check** – passed to lzma.open
>> - **preset** – passed to lzma.open
>> - **filters** – passed to lzma.open

to_json(*path*, *root_array=True*, *mode='wb'*, *compression=None*)

Saves the sequence to a json file. If root_array is True, then the sequence will be written to json with an array at the root. If it is False, then the sequence will be converted from a sequence of (Key, Value) pairs to a dictionary so that the json root is a dictionary.

> **Parameters**
>
> - **path** – path to write file
> - **root_array** – write json root as an array or dictionary
> - **mode** – file open mode

to_jsonl(*path*, *mode='wb'*, *compression=None*)

Saves the sequence to a jsonl file. Each element is mapped using json.dumps then written with a newline separating each element.

> **Parameters**
>
> - **path** – path to write file
> - **mode** – mode to write in, defaults to 'w' to overwrite contents
> - **compression** – compression format

to_list()

Converts sequence to list of elements.

```
>>> type(seq([]).to_list())
list
```

```
>>> type(seq([]))
functional.pipeline.Sequence
```

```
>>> seq([1, 2, 3]).to_list()
[1, 2, 3]
```

> **Returns** list of elements in sequence

to_pandas(*columns=None*)

Converts sequence to a pandas DataFrame using pandas.DataFrame.from_records

> **Parameters** **columns** – columns for pandas to use
>
> **Returns** DataFrame of sequence

to_set()

Converts sequence to a set of elements.

```
>>> type(seq([])).to_set()
set
```

```
>>> type(seq([]))
functional.pipeline.Sequence
```

```
>>> seq([1, 1, 2, 2]).to_set()
{1, 2}
```

:return:set of elements in sequence

to_sqlite3(*conn*, *target*, **args*, ***kwargs*)

Saves the sequence to sqlite3 database. Target table must be created in advance. The table schema is inferred from the elements in the sequence if only target table name is supplied.

```
>>> seq([(1, 'Tom'), (2, 'Jack')])                    .to_sqlite3('users.db', 'INSERT INTO user
```

```
>>> seq([{'id': 1, 'name': 'Tom'}, {'id': 2, 'name': 'Jack'}]).to_sqlite3(conn, 'user')
```

> **Parameters**
>
> - **conn** – path or sqlite connection, cursor
>
> - **target** – SQL query string or table name
>
> - **args** – passed to sqlite3.connect
>
> - **kwargs** – passed to sqlite3.connect

union(*other*)

> New sequence with unique elements from self and other.

```
>>> seq([1, 1, 2, 3, 3]).union([1, 4, 5])
[1, 2, 3, 4, 5]
```

> **Parameters other** – sequence to union with
>
> **Returns** union of sequence and other

where(*func*)

> Selects elements where func evaluates to True.

```
>>> seq([-1, 1, -2, 2]).where(lambda x: x > 0)
[1, 2]
```

> **Parameters func** – function to filter on
>
> **Returns** filtered sequence

zip(*sequence*)

> Zips the stored sequence with the given sequence.

```
>>> seq([1, 2, 3]).zip([4, 5, 6])
[(1, 4), (2, 5), (3, 6)]
```

> **Parameters sequence** – second sequence to zip
>
> **Returns** stored sequence zipped with given sequence

zip_with_index(*start=0*)

> Zips the sequence to its index, with the index being the second element of each tuple.

```
>>> seq(['a', 'b', 'c']).zip_with_index()
[('a', 0), ('b', 1), ('c', 2)]
```

> **Returns** sequence zipped to its index

functional.pipeline.**_wrap**(*value*)

> Wraps the passed value in a Sequence if it is not a primitive. If it is a string argument it is expanded to a list of characters.

```
>>> _wrap(1)
1
```

```
>>> _wrap("abc")
['a', 'b', 'c']
```

```
>>> type(_wrap([1, 2]))
functional.pipeline.Sequence
```

> **Parameters** **value** – value to wrap

> **Returns** wrapped or not wrapped value

1.2.3 functional.lineage

class functional.lineage.**Lineage**(*prior_lineage=None*, *engine=None*)

> Bases: object

> Class for tracking the lineage of transformations, and applying them to a given sequence.

> **__dict__** = dict_proxy({'__module__': 'functional.lineage', '__getitem__': <function __getitem__ at 0x7f1e41dfacf8>, '

> **__getitem__**(*item*)
>> Return specific transformation in lineage. :param item: Transformation to retrieve :return: Requested transformation

> **__init__**(*prior_lineage=None*, *engine=None*)
>> Construct an empty lineage if prior_lineage is None or if its not use it as the list of current transformations

>> **Parameters** **prior_lineage** – Lineage object to inherit

>> **Returns** new Lineage object

> **__len__**()
>> Number of transformations in lineage

>> **Returns** number of transformations

> **__module__** = 'functional.lineage'

> **__repr__**()
>> Returns readable representation of Lineage

>> **Returns** readable Lineage

> **__weakref__**
>> list of weak references to the object (if defined)

> **apply**(*transform*)
>> Add the transformation to the lineage :param transform: Transformation to apply

> **cache_scan**()
>> Scan the lineage for the index of the most recent cache. :return: Index of most recent cache

> **evaluate**(*sequence*)
>> Compute the lineage on the sequence.

>> **Parameters** **sequence** – Sequence to compute

>> **Returns** Evaluated sequence

1.2.4 functional.transformations

class `functional.transformations.`**`Transformation`**(*name*, *function*, *execution_strategies*)
    Bases: `tuple`

    `__dict__` **= dict_proxy({'function': <property object at 0x7f1e41e6af18>, '__module__': 'functional.transformations', '**

    `__getnewargs__`()
        Return self as a plain tuple. Used by copy and pickle.

    `__getstate__`()
        Exclude the OrderedDict from pickling

    `__module__` **= 'functional.transformations'**

    static **`__new__`**(*_cls*, *name*, *function*, *execution_strategies*)
        Create new instance of Transformation(name, function, execution_strategies)

    `__repr__`()
        Return a nicely formatted representation string

    `__slots__` **= ()**

    `_asdict`()
        Return a new OrderedDict which maps field names to their values

    `_fields` = ('name', 'function', 'execution_strategies')

    classmethod **`_make`**(*iterable*, *new=<built-in method __new__ of type object at 0x9192c0>*, *len=<built-in function len>*)
        Make a new Transformation object from a sequence or iterable

    `_replace`(*_self*, ***kwds*)
        Return a new Transformation object replacing specified fields with new values

    `execution_strategies`
        Alias for field number 2

    `function`
        Alias for field number 1

    `name`
        Alias for field number 0

`functional.transformations.`**`cartesian_t`**(*iterables*, *repeat*)
    Transformation for Sequence.cartesian :param iterables: elements for cartesian product :param repeat: how many times to repeat iterables :return: transformation

`functional.transformations.`**`difference_t`**(*other*)
    Transformation for Sequence.difference :param other: sequence to different with :return: transformation

`functional.transformations.`**`distinct_by_t`**(*func*)
    Transformation for Sequence.distinct_by :param func: distinct_by function :return: transformation

`functional.transformations.`**`distinct_t`**()
    Transformation for Sequence.distinct :return: transformation

`functional.transformations.`**`drop_right_t`**(*n*)
    Transformation for Sequence.drop_right :param n: number to drop from right :return: transformation

`functional.transformations.`**`drop_t`**(*n*)
    Transformation for Sequence.drop :param n: number to drop from left :return: transformation

`functional.transformations.`**`drop_while_t`**(*func*)
    Transformation for Sequence.drop_while :param func: drops while func is true :return: transformation

functional.transformations.**enumerate_t**(*start*)
> Transformation for Sequence.enumerate :param start: start index for enumerate :return: transformation

functional.transformations.**filter_not_t**(*func*)
> Transformation for Sequence.filter_not :param func: filter_not function :return: transformation

functional.transformations.**filter_t**(*func*)
> Transformation for Sequence.filter :param func: filter function :return: transformation

functional.transformations.**flat_map_impl**(*func*, *sequence*)
> Implementation for flat_map_t :param func: function to map :param sequence: sequence to flat_map over :return: flat_map generator

functional.transformations.**flat_map_t**(*func*)
> Transformation for Sequence.flat_map :param func: function to flat_map :return: transformation

functional.transformations.**flatten_t**()
> Transformation for Sequence.flatten :return: transformation

functional.transformations.**group_by_impl**(*func*, *sequence*)
> Implementation for group_by_t :param func: grouping function :param sequence: sequence to group :return: grouped sequence

functional.transformations.**group_by_key_impl**(*sequence*)
> Implementation for group_by_key_t :param sequence: sequence to group :return: grouped sequence

functional.transformations.**group_by_key_t**()
> Transformation for Sequence.group_by_key :return: transformation

functional.transformations.**group_by_t**(*func*)
> Transformation for Sequence.group_by :param func: grouping function :return: transformation

functional.transformations.**grouped_impl**(*wrap*, *size*, *sequence*)
> Implementation for grouped_t :param wrap: wrap children values with this :param size: size of groups :param sequence: sequence to group :return: grouped sequence

functional.transformations.**grouped_t**(*wrap*, *size*)
> Transformation for Sequence.grouped :param wrap: wrap children values with this :param size: size of groups :return: transformation

functional.transformations.**init_t**()
> Transformation for Sequence.init :return: transformation

functional.transformations.**inits_t**(*wrap*)
> Transformation for Sequence.inits :param wrap: wrap children values with this :return: transformation

functional.transformations.**inner_join_impl**(*other*, *sequence*)
> Implementation for part of join_impl :param other: other sequence to join with :param sequence: first sequence to join with :return: joined sequence

functional.transformations.**intersection_t**(*other*)
> Transformation for Sequence.intersection :param other: sequence to intersect with :return: transformation

functional.transformations.**join_impl**(*other*, *join_type*, *sequence*)
> Implementation for join_t :param other: other sequence to join with :param join_type: join type (inner, outer, left, right) :param sequence: first sequence to join with :return: joined sequence

functional.transformations.**join_t**(*other*, *join_type*)
> Transformation for Sequence.join, Sequence.inner_join, Sequence.outer_join, Sequence.right_join, and Sequence.left_join :param other: other sequence to join with :param join_type: join type from left, right, inner, and outer :return: transformation

`functional.transformations.`**`map_t`**(*func*)
>   Transformation for Sequence.map :param func: map function :return: transformation

`functional.transformations.`**`name`**(*function*)
>   Retrieve a pretty name for the function :param function: function to get name from :return: pretty name

`functional.transformations.`**`order_by_t`**(*func*)
>   Transformation for Sequence.order_by :param func: order_by function :return: transformation

`functional.transformations.`**`partition_t`**(*wrap*, *func*)
>   Transformation for Sequence.partition :param wrap: wrap children values with this :param func: partition function :return: transformation

`functional.transformations.`**`reduce_by_key_t`**(*func*)
>   Transformation for Sequence.reduce_by_key :param func: reduce function :return: transformation

`functional.transformations.`**`reversed_t`**()
>   Transformation for Sequence.reverse :return: transformation

`functional.transformations.`**`select_t`**(*func*)
>   Transformation for Sequence.select :param func: select function :return: transformation

`functional.transformations.`**`slice_t`**(*start*, *until*)
>   Transformation for Sequence.slice :param start: start index :param until: until index (does not include element at until) :return: transformation

`functional.transformations.`**`sliding_impl`**(*wrap*, *size*, *step*, *sequence*)
>   Implementation for sliding_t :param wrap: wrap children values with this :param size: size of window :param step: step size :param sequence: sequence to create sliding windows from :return: sequence of sliding windows

`functional.transformations.`**`sliding_t`**(*wrap*, *size*, *step*)
>   Transformation for Sequence.sliding :param wrap: wrap children values with this :param size: size of window :param step: step size :return: transformation

`functional.transformations.`**`sorted_t`**(*key=None*, *reverse=False*)
>   Transformation for Sequence.sorted :param key: key to sort by :param reverse: reverse or not :return: transformation

`functional.transformations.`**`symmetric_difference_t`**(*other*)
>   Transformation for Sequence.symmetric_difference :param other: sequence to symmetric_difference with :return: transformation

`functional.transformations.`**`tail_t`**()
>   Transformation for Sequence.tail :return: transformation

`functional.transformations.`**`tails_t`**(*wrap*)
>   Transformation for Sequence.tails :param wrap: wrap children values with this :return: transformation

`functional.transformations.`**`take_t`**(*n*)
>   Transformation for Sequence.take :param n: number to take :return: transformation

`functional.transformations.`**`take_while_t`**(*func*)
>   Transformation for Sequence.take_while :param func: takes while func is True :return: transformation

`functional.transformations.`**`union_t`**(*other*)
>   Transformation for Sequence.union :param other: sequence to union with :return: transformation

`functional.transformations.`**`where_t`**(*func*)
>   Transformation for Sequence.where :param func: where function :return: transformation

`functional.transformations.`**`zip_t`**(*zip_sequence*)
>   Transformation for Sequence.zip :param zip_sequence: sequence to zip with :return: transformation

`functional.transformations.`**`zip_with_index_t`**(*start*)
> Transformation for Sequence.zip_with_index :return: transformation

1.2.5 functional.util

`functional.util.`**`compose`**(**functions*)
> Compose all the function arguments together :param functions: Functions to compose :return: Single composed function

`functional.util.`**`compute_partition_size`**(*result*, *processes*)
> Attempts to compute the partition size to evenly distribute work across processes. Defaults to 1 if the length of result cannot be determined.
>
> > **Parameters**
> >
> > - **result** – Result to compute on
> > - **processes** – Number of processes to use
> >
> > **Returns** Best partition size

`functional.util.`**`identity`**(*arg*)
> Function which returns the argument. Used as a default lambda function.

```
>>> obj = object()
>>> obj is identity(obj)
True
```

> > **Parameters arg** – object to take identity of
> >
> > **Returns** return arg

`functional.util.`**`is_iterable`**(*val*)
> Check if val is not a list, but is a collections.Iterable type. This is used to determine when list() should be called on val

```
>>> l = [1, 2]
>>> is_iterable(l)
False
>>> is_iterable(iter(l))
True
```

> > **Parameters val** – value to check
> >
> > **Returns** True if it is not a list, but is a collections.Iterable

`functional.util.`**`is_namedtuple`**(*val*)
> Use Duck Typing to check if val is a named tuple. Checks that val is of type tuple and contains the attribute _fields which is defined for named tuples. :param val: value to check type of :return: True if val is a namedtuple

`functional.util.`**`is_primitive`**(*val*)
> Checks if the passed value is a primitive type.

```
>>> is_primitive(1)
True
```

```
>>> is_primitive("abc")
True
```

```
>>> is_primitive(True)
True
```

```
>>> is_primitive({})
False
```

```
>>> is_primitive([])
False
```

```
>>> is_primitive(set([]))
```

> **Parameters** **val** – value to check
>
> **Returns** True if value is a primitive, else False

functional.util.**lazy_parallelize**(*func*, *result*, *processes=None*, *partition_size=None*)
> Lazily computes an iterable in parallel, and returns them in pool chunks :param func: Function to apply :param result: Data to apply to :param processes: Number of processes to use in parallel :param partition_size: Size of partitions for each parallel process :return: Iterable of chunks where each chunk as func applied to it

functional.util.**pack**(*func*, *args*)
> Pack a function and the args it should be applied to :param func: Function to apply :param args: Args to evaluate with :return: Packed (func, args) tuple

functional.util.**parallelize**(*func*, *result*, *processes=None*, *partition_size=None*)
> Creates an iterable which is lazily computed in parallel from applying func on result :param func: Function to apply :param result: Data to apply to :param processes: Number of processes to use in parallel :param partition_size: Size of partitions for each parallel process :return: Iterable of applying func on result

functional.util.**split_every**(*parts*, *iterable*)
> Split an iterable into parts of length parts

```
>>> l = iter([1, 2, 3, 4])
>>> split_every(2, l)
[[1, 2], [3, 4]]
```

> **Parameters**
>
> > - **iterable** – iterable to split
> > - **parts** – number of chunks
>
> **Returns** return the iterable split in parts

functional.util.**unpack**(*packed*)
> Unpack the function and args then apply the function to the arguments and return result :param packed: input packed tuple of (func, args) :return: result of applying packed function on packed args

f

Symbols

A

C

Z