

---

# **PyFunctional Documentation**

*Release 0.6.0*

**Pedro Rodriguez**

February 02, 2017



<b>1</b>	<b>Table of Contents</b>	<b>3</b>
1.1	API Documentation . . . . .	3
1.2	Developer Documentation . . . . .	24
	<b>Python Module Index</b>	<b>55</b>



Welcome to the *PyFunctional* documentation. For a tutorial of how to use this package you should visit [pyfunctional.org](http://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, max\_repr\_items=100*)

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

**csv\_dict\_reader** (*csv\_file, fieldnames=None, restkey=None, restval=None, dialect='excel', \*\*kwds*)

**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`, `max_repr_items=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_right(['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, V 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(n=None)**

Converts sequence to list of elements.

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

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

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

**Parameters** *n* – Take *n* elements of sequence if not None

**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 function

**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*, \**initial*)

Reduce sequence of elements using func. API mirrors functools.reduce

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

**Parameters**

- **func** – two parameter, associative reduce function
- **initial** – single optional argument acting as initial value

**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)])
[('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 be present, 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

**show** (*n=10, headers=(), tablefmt='simple', floatfmt='g', numalign='decimal', stralign='left', missingval=''*)

Pretty print first *n* rows of sequence as a table. See <https://bitbucket.org/astanin/python-tabulate> for details on tabulate parameters

**Parameters**

- **n** – Number of rows to show
- **headers** – Passed to tabulate
- **tablefmt** – Passed to tabulate
- **floatfmt** – Passed to tabulate
- **numalign** – Passed to tabulate
- **stralign** – Passed to tabulate
- **missingval** – Passed to tabulate

**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

**smap** (*func*)

Alias to Sequence.starmap

starmaps f onto the sequence as itertools.starmap does.

```
>>> seq([(2, 3), (-2, 1), (0, 10)]).smap(lambda x, y: x + y)
[5, -1, 10]
```

**Parameters** **func** – function to starmap with

**Returns** sequence with func starmapped onto it

**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

**starmap** (*func*)

starmaps f onto the sequence as itertools.starmap does.

```
>>> seq([(2, 3), (-2, 1), (0, 10)]).starmap(lambda x, y: x + y)
[5, -1, 10]
```

**Parameters** **func** – function to starmap with

**Returns** sequence with func starmapped onto it

**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

**tabulate** (*n=None, headers=(), tablefmt='simple', floatfmt='g', numalign='decimal', stralign='left', missingval=''*)

Return pretty string table of first *n* rows of sequence or everything if *n* is None. See <https://bitbucket.org/astanin/python-tabulate> for details on tabulate parameters

**Parameters**

- **n** – Number of rows to show, if set to None return all rows
- **headers** – Passed to tabulate
- **tablefmt** – Passed to tabulate
- **floatfmt** – Passed to tabulate
- **numalign** – Passed to tabulate
- **stralign** – Passed to tabulate
- **missingval** – Passed to tabulate

**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='wt', 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='wt'*, *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** (*n=None*)

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]
```

**Parameters** **n** – Take `n` elements of sequence if not `None`

**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, max\_repr\_items=100*)

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, \*\*kwargs*)

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\_\_** = `mappingproxy({'__dict__': <attribute '__dict__' of 'Stream' objects>, '__init__': <function Stream.__init__`

**\_\_init\_\_** (*disable\_compression=False, max\_repr\_items=100*)

Default stream constructor. :param *disable\_compression*: Disable file compression detection

**\_\_module\_\_** = 'functional.streams'

**`__weakref__`**

list of weak references to the object (if defined)

**`_parse_args`** (*args, engine, error\_message*)**`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

**`csv_dict_reader`** (*csv\_file, fieldnames=None, restkey=None, restval=None, dialect='excel', \*\*kwds*)**`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,
                                     max_repr_items=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__` = `mappingproxy({'drop_right': <function Sequence.drop_right at 0x7f7f4a9d6ae8>, 'cartesian': <function Se`

`__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*, *max\_repr\_items=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
- **transform** – transformation to apply
- **engine** – execution engine
- **max\_repr\_items** – maximum number of items to print with repr

**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

`_repr_html_()`

Allows IPython render HTML tables :return: First 10 rows of data as an HTML table

`_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  $f(\text{current: B}, \text{next: A} \Rightarrow \text{result: B})$ . 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_right(['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, V 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** (*n=None*)

Converts sequence to list of elements.

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

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

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

**Parameters** *n* – Take *n* elements of sequence if not None

**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 function

**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, \*initial*)

Reduce sequence of elements using func. API mirrors functools.reduce

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

**Parameters**

- **func** – two parameter, associative reduce function
- **initial** – single optional argument acting as initial value

**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)])
[('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 be present, 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

**show** (*n=10, headers=(), tablefmt='simple', floatfmt='g', numalign='decimal', stralign='left', missingval=''*)

Pretty print first n rows of sequence as a table. See <https://bitbucket.org/astanin/python-tabulate> for details on tabulate parameters

**Parameters**

- **n** – Number of rows to show
- **headers** – Passed to tabulate
- **tablefmt** – Passed to tabulate
- **floatfmt** – Passed to tabulate
- **numalign** – Passed to tabulate
- **stralign** – Passed to tabulate
- **missingval** – Passed to tabulate

**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

**smap(func)**

Alias to Sequence.starmap

starmaps f onto the sequence as itertools.starmap does.

```
>>> seq([(2, 3), (-2, 1), (0, 10)]).smap(lambda x, y: x + y)
[5, -1, 10]
```

**Parameters** **func** – function to starmap with

**Returns** sequence with func starmapped onto it

**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

**starmap(func)**

starmaps f onto the sequence as itertools.starmap does.

```
>>> seq([(2, 3), (-2, 1), (0, 10)]).starmap(lambda x, y: x + y)
[5, -1, 10]
```

**Parameters** **func** – function to starmap with

**Returns** sequence with func starmapped onto it

**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

**tabulate** (*n=None, headers=(), tablefmt='simple', floatfmt='g', numalign='decimal', stralign='left', missingval=''*)

Return pretty string table of first n rows of sequence or everything if n is None. See <https://bitbucket.org/astanin/python-tabulate> for details on tabulate parameters

**Parameters**

- **n** – Number of rows to show, if set to None return all rows
- **headers** – Passed to tabulate
- **tablefmt** – Passed to tabulate
- **floatfmt** – Passed to tabulate
- **numalign** – Passed to tabulate
- **stralign** – Passed to tabulate
- **missingval** – Passed to tabulate

**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='wt', 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='wt'*, *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** (*n=None*)

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]
```

**Parameters** *n* – Take *n* elements of sequence if not None

**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\_\_** = mappingproxy({'apply': <function Lineage.apply at 0x7f7f4a9d38c8>, '\_\_init\_\_': <function Lineage.\_\_init\_\_ at 0x7f7f4a9d38c8>})

**\_\_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__ = mappingproxy({'execution_strategies': <property object at 0x7f7f4a9c23b8>, '__slots__': (), '__getstate__':`

`__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 0x971940>, 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

**`_source`** = “from builtins import property as `_property`, tuple as `_tuple`\nfrom operator import itemgetter as `_itemgetter`”

**`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.starmap_t` (*func*)  
Transformation for `Sequence.starmap` and `Sequence.smap` :param func: starmap function :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.is_tabulatable` (*val*)

`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**

`functional.lineage`, 47  
`functional.pipeline`, 5  
`functional.streams`, 3  
`functional.transformations`, 48  
`functional.util`, 51



## Symbols

- `__add__()` (functional.pipeline.Sequence method), 26
  - `__bool__()` (functional.pipeline.Sequence method), 27
  - `__call__()` (functional.streams.ParallelStream method), 24
  - `__call__()` (functional.streams.Stream method), 24
  - `__contains__()` (functional.pipeline.Sequence method), 27
  - `__dict__` (functional.lineage.Lineage attribute), 47
  - `__dict__` (functional.pipeline.Sequence attribute), 27
  - `__dict__` (functional.streams.Stream attribute), 24
  - `__dict__` (functional.transformations.Transformation attribute), 48
  - `__eq__()` (functional.pipeline.Sequence method), 27
  - `__getitem__()` (functional.lineage.Lineage method), 47
  - `__getitem__()` (functional.pipeline.Sequence method), 27
  - `__getnewargs__()` (functional.transformations.Transformation method), 48
  - `__getstate__()` (functional.transformations.Transformation method), 48
  - `__hash__()` (functional.pipeline.Sequence method), 27
  - `__init__()` (functional.lineage.Lineage method), 47
  - `__init__()` (functional.pipeline.Sequence method), 27
  - `__init__()` (functional.streams.ParallelStream method), 24
  - `__init__()` (functional.streams.Stream method), 24
  - `__iter__()` (functional.pipeline.Sequence method), 27
  - `__len__()` (functional.lineage.Lineage method), 48
  - `__module__` (functional.lineage.Lineage attribute), 48
  - `__module__` (functional.pipeline.Sequence attribute), 27
  - `__module__` (functional.streams.ParallelStream attribute), 24
  - `__module__` (functional.streams.Stream attribute), 24
  - `__module__` (functional.transformations.Transformation attribute), 48
  - `__ne__()` (functional.pipeline.Sequence method), 27
  - `__new__()` (functional.transformations.Transformation static method), 48
  - `__nonzero__()` (functional.pipeline.Sequence method), 27
  - `__repr__()` (functional.lineage.Lineage method), 48
  - `__repr__()` (functional.pipeline.Sequence method), 28
  - `__repr__()` (functional.transformations.Transformation method), 48
  - `__reversed__()` (functional.pipeline.Sequence method), 28
  - `__slots__` (functional.transformations.Transformation attribute), 48
  - `__str__()` (functional.pipeline.Sequence method), 28
  - `__weakref__` (functional.lineage.Lineage attribute), 48
  - `__weakref__` (functional.pipeline.Sequence attribute), 28
  - `__weakref__` (functional.streams.Stream attribute), 24
  - `__asdict__()` (functional.transformations.Transformation method), 48
  - `_evaluate__()` (functional.pipeline.Sequence method), 28
  - `_fields` (functional.transformations.Transformation attribute), 48
  - `_make__()` (functional.transformations.Transformation class method), 48
  - `_parse_args__()` (functional.streams.Stream method), 25
  - `_replace__()` (functional.transformations.Transformation method), 48
  - `_repr_html__()` (functional.pipeline.Sequence method), 28
  - `_source` (functional.transformations.Transformation attribute), 49
  - `_to_sqlite3_by_query__()` (functional.pipeline.Sequence method), 28
  - `_to_sqlite3_by_table__()` (functional.pipeline.Sequence method), 28
  - `_transform__()` (functional.pipeline.Sequence method), 28
  - `_wrap__()` (in module functional.pipeline), 47
- ## A
- `aggregate__()` (functional.pipeline.Sequence method), 5, 28
  - `all__()` (functional.pipeline.Sequence method), 5, 29
  - `any__()` (functional.pipeline.Sequence method), 5, 29
  - `apply__()` (functional.lineage.Lineage method), 48
  - `average__()` (functional.pipeline.Sequence method), 6, 29
- ## C
- `cache__()` (functional.pipeline.Sequence method), 6, 29

cache\_scan() (functional.lineage.Lineage method), 48  
 cartesian() (functional.pipeline.Sequence method), 6, 29  
 cartesian\_t() (in module functional.transformations), 49  
 compose() (in module functional.util), 51  
 compute\_partition\_size() (in module functional.util), 51  
 count() (functional.pipeline.Sequence method), 6, 30  
 csv() (functional.streams.Stream method), 3, 25  
 csv\_dict\_reader() (functional.streams.Stream method), 3, 25

## D

dict() (functional.pipeline.Sequence method), 6, 30  
 difference() (functional.pipeline.Sequence method), 7, 30  
 difference\_t() (in module functional.transformations), 49  
 distinct() (functional.pipeline.Sequence method), 7, 30  
 distinct\_by() (functional.pipeline.Sequence method), 7, 30  
 distinct\_by\_t() (in module functional.transformations), 49  
 distinct\_t() (in module functional.transformations), 49  
 drop() (functional.pipeline.Sequence method), 7, 30  
 drop\_right() (functional.pipeline.Sequence method), 7, 31  
 drop\_right\_t() (in module functional.transformations), 49  
 drop\_t() (in module functional.transformations), 49  
 drop\_while() (functional.pipeline.Sequence method), 8, 31  
 drop\_while\_t() (in module functional.transformations), 49

## E

empty() (functional.pipeline.Sequence method), 8, 31  
 enumerate() (functional.pipeline.Sequence method), 8, 31  
 enumerate\_t() (in module functional.transformations), 49  
 evaluate() (functional.lineage.Lineage method), 48  
 execution\_strategies (functional.transformations.Transformation attribute), 49  
 exists() (functional.pipeline.Sequence method), 8, 31

## F

filter() (functional.pipeline.Sequence method), 8, 32  
 filter\_not() (functional.pipeline.Sequence method), 8, 32  
 filter\_not\_t() (in module functional.transformations), 49  
 filter\_t() (in module functional.transformations), 49  
 find() (functional.pipeline.Sequence method), 9, 32  
 first() (functional.pipeline.Sequence method), 9, 32  
 flat\_map() (functional.pipeline.Sequence method), 9, 32  
 flat\_map\_impl() (in module functional.transformations), 49  
 flat\_map\_t() (in module functional.transformations), 49  
 flatten() (functional.pipeline.Sequence method), 9, 33  
 flatten\_t() (in module functional.transformations), 49  
 fold\_left() (functional.pipeline.Sequence method), 9, 33  
 fold\_right() (functional.pipeline.Sequence method), 10, 33

for\_all() (functional.pipeline.Sequence method), 10, 33  
 for\_each() (functional.pipeline.Sequence method), 10, 34  
 function (functional.transformations.Transformation attribute), 49  
 functional.lineage (module), 47  
 functional.pipeline (module), 5, 26  
 functional.streams (module), 3, 24  
 functional.transformations (module), 48  
 functional.util (module), 51

## G

group\_by() (functional.pipeline.Sequence method), 10, 34  
 group\_by\_impl() (in module functional.transformations), 49  
 group\_by\_key() (functional.pipeline.Sequence method), 11, 34  
 group\_by\_key\_impl() (in module functional.transformations), 49  
 group\_by\_key\_t() (in module functional.transformations), 49  
 group\_by\_t() (in module functional.transformations), 49  
 grouped() (functional.pipeline.Sequence method), 11, 34  
 grouped\_impl() (in module functional.transformations), 50  
 grouped\_t() (in module functional.transformations), 50

## H

head() (functional.pipeline.Sequence method), 11, 34  
 head\_option() (functional.pipeline.Sequence method), 11, 34

## I

identity() (in module functional.util), 51  
 init() (functional.pipeline.Sequence method), 11, 35  
 init\_t() (in module functional.transformations), 50  
 inits() (functional.pipeline.Sequence method), 12, 35  
 inits\_t() (in module functional.transformations), 50  
 inner\_join() (functional.pipeline.Sequence method), 12, 35  
 inner\_join\_impl() (in module functional.transformations), 50  
 intersection() (functional.pipeline.Sequence method), 12, 35  
 intersection\_t() (in module functional.transformations), 50  
 is\_iterable() (in module functional.util), 52  
 is\_namedtuple() (in module functional.util), 52  
 is\_primitive() (in module functional.util), 52  
 is\_tabulatable() (in module functional.util), 52

## J

join() (functional.pipeline.Sequence method), 12, 35  
 join\_impl() (in module functional.transformations), 50

join\_t() (in module functional.transformations), 50  
 json() (functional.streams.Stream method), 3, 25  
 jsonl() (functional.streams.Stream method), 4, 25

## L

last() (functional.pipeline.Sequence method), 13, 36  
 last\_option() (functional.pipeline.Sequence method), 13, 36  
 lazy\_parallelize() (in module functional.util), 52  
 left\_join() (functional.pipeline.Sequence method), 13, 36  
 len() (functional.pipeline.Sequence method), 13, 37  
 Lineage (class in functional.lineage), 47  
 list() (functional.pipeline.Sequence method), 13, 37

## M

make\_string() (functional.pipeline.Sequence method), 14, 37  
 map() (functional.pipeline.Sequence method), 14, 37  
 map\_t() (in module functional.transformations), 50  
 max() (functional.pipeline.Sequence method), 14, 37  
 max\_by() (functional.pipeline.Sequence method), 15, 38  
 min() (functional.pipeline.Sequence method), 15, 38  
 min\_by() (functional.pipeline.Sequence method), 15, 39

## N

name (functional.transformations.Transformation attribute), 49  
 name() (in module functional.transformations), 50  
 non\_empty() (functional.pipeline.Sequence method), 16, 39

## O

open() (functional.streams.Stream method), 4, 25  
 order\_by() (functional.pipeline.Sequence method), 16, 39  
 order\_by\_t() (in module functional.transformations), 50  
 outer\_join() (functional.pipeline.Sequence method), 16, 39

## P

pack() (in module functional.util), 52  
 parallelize() (in module functional.util), 53  
 ParallelStream (class in functional.streams), 3, 24  
 partition() (functional.pipeline.Sequence method), 16, 40  
 partition\_t() (in module functional.transformations), 50  
 product() (functional.pipeline.Sequence method), 16, 40

## R

range() (functional.streams.Stream method), 4, 26  
 reduce() (functional.pipeline.Sequence method), 17, 40  
 reduce\_by\_key() (functional.pipeline.Sequence method), 17, 40  
 reduce\_by\_key\_t() (in module functional.transformations), 50

reverse() (functional.pipeline.Sequence method), 17, 40  
 reversed\_t() (in module functional.transformations), 50  
 right\_join() (functional.pipeline.Sequence method), 17, 40

## S

select() (functional.pipeline.Sequence method), 17, 41  
 select\_t() (in module functional.transformations), 50  
 Sequence (class in functional.pipeline), 5, 26  
 sequence (functional.pipeline.Sequence attribute), 18, 41  
 set() (functional.pipeline.Sequence method), 18, 41  
 show() (functional.pipeline.Sequence method), 18, 41  
 size() (functional.pipeline.Sequence method), 18, 41  
 slice() (functional.pipeline.Sequence method), 18, 42  
 slice\_t() (in module functional.transformations), 50  
 sliding() (functional.pipeline.Sequence method), 19, 42  
 sliding\_impl() (in module functional.transformations), 50  
 sliding\_t() (in module functional.transformations), 50  
 smap() (functional.pipeline.Sequence method), 19, 42  
 sorted() (functional.pipeline.Sequence method), 19, 42  
 sorted\_t() (in module functional.transformations), 51  
 split\_every() (in module functional.util), 53  
 sqlite3() (functional.streams.Stream method), 5, 26  
 starmap() (functional.pipeline.Sequence method), 19, 42  
 starmap\_t() (in module functional.transformations), 51  
 Stream (class in functional.streams), 3, 24  
 sum() (functional.pipeline.Sequence method), 19, 43  
 symmetric\_difference() (functional.pipeline.Sequence method), 20, 43  
 symmetric\_difference\_t() (in module functional.transformations), 51

## T

tabulate() (functional.pipeline.Sequence method), 20, 43  
 tail() (functional.pipeline.Sequence method), 20, 43  
 tail\_t() (in module functional.transformations), 51  
 tails() (functional.pipeline.Sequence method), 20, 43  
 tails\_t() (in module functional.transformations), 51  
 take() (functional.pipeline.Sequence method), 20, 44  
 take\_t() (in module functional.transformations), 51  
 take\_while() (functional.pipeline.Sequence method), 21, 44  
 take\_while\_t() (in module functional.transformations), 51  
 to\_csv() (functional.pipeline.Sequence method), 21, 44  
 to\_dict() (functional.pipeline.Sequence method), 21, 44  
 to\_file() (functional.pipeline.Sequence method), 21, 44  
 to\_json() (functional.pipeline.Sequence method), 22, 45  
 to\_jsonl() (functional.pipeline.Sequence method), 22, 45  
 to\_list() (functional.pipeline.Sequence method), 22, 45  
 to\_pandas() (functional.pipeline.Sequence method), 22, 46  
 to\_set() (functional.pipeline.Sequence method), 22, 46  
 to\_sqlite3() (functional.pipeline.Sequence method), 23, 46

Transformation (class in functional.transformations), 48

## U

union() (functional.pipeline.Sequence method), 23, 46

union\_t() (in module functional.transformations), 51

unpack() (in module functional.util), 53

## W

where() (functional.pipeline.Sequence method), 23, 46

where\_t() (in module functional.transformations), 51

## Z

zip() (functional.pipeline.Sequence method), 23, 47

zip\_t() (in module functional.transformations), 51

zip\_with\_index() (functional.pipeline.Sequence method),  
23, 47

zip\_with\_index\_t() (in module functional.transformations), 51