

---

# ScalaFunctional Documentation

*Release 0.1.6*

**Pedro Rodriguez**

November 04, 2015



<b>1</b>	<b>Table of Contents</b>	<b>3</b>
1.1	API Documentation . . . . .	3
1.2	Developer Documentation . . . . .	21
<b>2</b>	<b>Documentation</b>	<b>49</b>
	<b>Python Module Index</b>	<b>51</b>



*ScalaFunctional* is a library for creating data pipelines and analysis in an easy and accessible way. It is primarily inspired by the APIs from [Apache Spark RDDs](#), [Scala Collections](#), and [Microsoft LINQ](#).



---

## Table of Contents

---

### 1.1 API Documentation

#### 1.1.1 Streams API

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

Additional entry point to Sequence which parses the input of a csv stream or file according to the defined options. `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

`functional.streams.json(json_file)`

Additional entry point to Sequence which parses the input of a json file handler or file from the given path. Json files are parsed in the following ways depending on if the root is a dictionary or 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', {'date_created': u'08/08', 'news_email': True, 'email': u'sarah@gmail.com'}]
```

**Parameters** `json_file` – path or file containing json content

**Returns** Sequence wrapping jsonl file

`functional.streams.jsonl(jsonl_file)`

Additional entry point to Sequence which parses the input of a jsonl file stream or file from the given path. Jsonl formatted files have a single valid json value on each line which is parsed by the python json module.

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

**Parameters** `jsonl_file` – path or file containing jsonl content

**Returns** Sequence wrapping jsonl file

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

Additional entry point to Sequence which parses input files as defined by options. Path specifies what file to parse. 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'
```

```
]
```

**param path** path to file

**param delimiter** delimiter to split joined text on. if None, defaults to `file.readlines()`

**param mode** file open mode

**param buffering** passed to `builtins.open`

**param encoding** passed to `builtins.open`

**param errors** passed to `builtins.open`

**param newline** passed to `builtins.open`

**return** output of file depending on options wrapped in a Sequence via `seq`

`functional.streams.range` (*\*args*)

Additional entry point to Sequence which wraps the builtin range generator. `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

`functional.streams.seq` (*\*args*)

Primary entrypoint for the functional package. Returns a `functional.pipeline.Sequence` wrapping the original sequence.

Additionally it parses various types of input to a Sequence as best it can.

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

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

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

```
>>> seq(range(4))
[0, 1, 2, 3]
```



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

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

**Parameters** **args** – Three types of arguments are valid. 1) Iterable which is then directly wrapped as a Sequence 2) A list of arguments is converted to a Sequence 3) A single non-iterable is converted to a single element Sequence

**Returns** wrapped sequence

## 1.1.2 Transformations and Actions API

The pipeline module contains the primary data structure Sequence and entry point seq

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

Bases: object

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

**count** (*func*)

Counts the number of elements in the sequence which satisfy the predicate func.

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

**Parameters** **func** – predicate to count elements on

**Returns** count of elements that satisfy predicate

**dict** (*default=None*)

Converts sequence of (Key, Value) pairs to a dictionary.

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

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

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

**Returns** dictionary from sequence of (Key, Value) elements

**difference** (*other*)

New sequence with unique elements present in sequence but not in other.

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

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

**Returns** difference of sequence and other

**distinct()**

Returns sequence of distinct elements. Elements must be hashable.

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

**Returns** sequence of distinct elements

**distinct\_by(func)**

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

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

**Returns** elements distinct by func

**drop(n)**

Drop the first n elements of the sequence.

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

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

**Returns** sequence without first n elements

**drop\_right(n)**

Drops the last n elements of the sequence.

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

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

**Returns** sequence with last n elements dropped

**drop\_while(func)**

Drops elements in the sequence while func evaluates to True, then returns the rest.

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

**Parameters** **func** – truth returning function

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

**empty()**

Returns True if the sequence has length zero.

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

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

**Returns** True if sequence length is zero

**enumerate** (*start=0*)

Uses python enumerate to to zip the sequence with indexes starting at start.

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

**Parameters** **start** – Beginning of zip

**Returns** enumerated sequence starting at start

**exists** (*func*)

Returns True if an element in the sequence makes func evaluate to True.

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

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

**Parameters** **func** – existence check function

**Returns** True if any element satisfies func

**filter** (*func*)

Filters sequence to include only elements where func is True.

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

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

**Returns** filtered sequence

**filter\_not** (*func*)

Filters sequence to include only elements where func is False.

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

**Parameters** **func** – function to filter\_not on

**Returns** filtered sequence

**find** (*func*)

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

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

**Parameters** **func** – function to find with

**Returns** first element to satisfy func or None

**first** ()

Returns the first element of the sequence.

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

Raises `IndexError` when the sequence is empty.

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

**Returns** first element of sequence

### `flat_map` (*func*)

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

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

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

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

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

**Returns** application of `func` to elements followed by flattening

### `flatten` ()

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

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

**Returns** flattened sequence

### `fold_left` (*zero\_value*, *func*)

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

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

#### Parameters

- **zero\_value** – zero value to reduce into
- **func** – Two parameter function as described by function docs

**Returns** value from folding values with `func` into `zero_value` from left to right.

### `fold_right` (*zero\_value*, *func*)

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

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

**Parameters**

- **zero\_value** – zero value to reduce into
- **func** – Two parameter function as described by function docs

**Returns** value from folding values with func into zero\_value from right to left

**for\_all** (*func*)

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

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

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

**Parameters** **func** – function to check truth value of all elements with

**Returns** True if all elements make func evaluate to True

**for\_each** (*func*)

Executes func on each element of the sequence.

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

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

**group\_by** (*func*)

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

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

**Parameters** **func** – group by result of this function

**Returns** grouped sequence

**group\_by\_key** ()

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

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

**Returns** sequence grouped by key

**grouped** (*size*)

Partitions the elements into groups of length size.

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

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

The last partition will be at least of size 1 and no more than length size :param size: size of the partitions  
:return: 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** ()

Converts sequence to list of elements.

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

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

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

**Returns** list of elements in sequence

#### **make\_string** (*separator*)

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

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

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

**Returns** concatenated string separated by separator

#### **map** (*func*)

Maps f onto the elements of the sequence.

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

**Parameters** `func` – function to map with

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

**max()**

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

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

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

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

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

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

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

**Returns** Maximal value of sequence

**max\_by(func)**

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

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

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

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

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

**Parameters** `func` – function to compute max by

**Returns** Maximal element by `func(element)`

**min()**

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

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

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

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

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

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

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

**Returns** Minimal value of sequence

**min\_by** (*func*)

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

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

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

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

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

**Parameters** **func** – function to compute min by

**Returns** Maximal element by func(element)

**non\_empty** ()

Returns True if the sequence does not have length zero.

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

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

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

**order\_by** (*func*)

Orders the input according to func

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

**Parameters** **func** – order by 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*)

Reduce sequence of elements using func.

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

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

**Returns** reduced value using func

**reduce\_by\_key** (*func*)

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

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

**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*

**sorted** (*key=None, reverse=False*)

Uses python sort and its passed arguments to sort the input.

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

**Parameters**

- **key** – sort using key function
- **reverse** – return list reversed or not

**Returns** sorted sequence

**sum** (*projection=None*)

Takes sum of elements in sequence.

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

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

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

**Returns** sum of elements in sequence

**symmetric\_difference** (*other*)

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

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

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

**Returns** symmetric difference of sequence and *other*

**tail** ()

Returns the sequence, without its first element.

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

**Returns** sequence without first element

**tails()**

Returns consecutive tails of the sequence.

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

**Returns** consecutive tail(s) of the sequence

**take(n)**

Take the first n elements of the sequence.

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

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

**Returns** first n elements of sequence

**take\_while(func)**

Take elements in the sequence until func evaluates to False, then return them.

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

**Parameters** **func** – truth returning function

**Returns** elements taken until func evaluates to False

**to\_csv(path, mode='wb', dialect='excel', \*\*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
- **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, mode='w', buffering=-1, encoding=None, errors=None, newline=None*)

Saves the sequence to a file by executing `str(self)` which becomes `str(self.to_list())`

#### Parameters

- **path** – path to write file
- **delimiter** – delimiter to split joined text on. if None, defaults to `file.readlines()`
- **mode** – file open mode
- **buffering** – passed to `builtins.open`
- **encoding** – passed to `builtins.open`
- **errors** – passed to `builtins.open`
- **newline** – passed to `builtins.open`

**to\_json** (*path, root\_array=True, mode='wb'*)

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='w'*)

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

**to\_list** ()

Converts sequence to list of elements.

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

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

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

**Returns** list of elements in sequence

**to\_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

**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** ()

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

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

**Returns** sequence zipped to its index

## 1.2 Developer Documentation

### 1.2.1 functional.streams

`functional.streams.csv` (*csv\_file*, *dialect='excel'*, *\*\*fmt\_params*)

Additional entry point to Sequence which parses the input of a csv stream or file according to the defined options. `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`functional.streams.json(json_file)`

Additional entry point to Sequence which parses the input of a json file handler or file from the given path. Json files are parsed in the following ways depending on if the root is a dictionary or 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`functional.streams.jsonl(jsonl_file)`

Additional entry point to Sequence which parses the input of a jsonl file stream or file from the given path. Jsonl formatted files 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`functional.streams.open(path, delimiter=None, mode='r', buffering=-1, encoding=None, errors=None, newline=None)`

Additional entry point to Sequence which parses input files as defined by options. Path specifies what file to parse. 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
```

]

**param path** path to file**param delimiter** delimiter to split joined text on. if None, defaults to `file.readlines()`**param mode** file open mode**param buffering** passed to `builtins.open`**param encoding** passed to `builtins.open`**param errors** passed to `builtins.open`**param newline** passed to `builtins.open`**return** output of file depending on options wrapped in a Sequence via `seq`

`functional.streams.range(*args)`

Additional entry point to Sequence which wraps the builtin range generator. `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

`functional.streams.seq(*args)`

Primary entrypoint for the functional package. Returns a `functional.pipeline.Sequence` wrapping the original sequence.

Additionally it parses various types of input to a Sequence as best it can.

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

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

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

```
>>> seq(range(4))
[0, 1, 2, 3]
```

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

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

**Parameters** `args` – Three types of arguments are valid. 1) Iterable which is then directly wrapped as a Sequence 2) A list of arguments is converted to a Sequence 3) A single non-iterable is converted to a single element Sequence

**Returns** wrapped sequence

## 1.2.2 functional.pipeline

The pipeline module contains the primary data structure Sequence and entry point seq

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

Bases: object

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

`__add__` (*other*)

Concatenates sequence with other.

**Parameters** `other` – sequence to concatenate

**Returns** concatenated sequence with other

`__bool__` ()

Returns True if size is not zero.

**Returns** True if size is not zero

`__contains__` (*item*)

Checks if item is in sequence.

**Parameters** *item* – item to check

**Returns** True if item is in sequence

`__dict__` = `dict_proxy`({'all': <function all at 0x7efea977e8c0>, 'set': <function set at 0x7efea9780c08>, 'symmetric\_diff

`__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*)

Takes a sequence and wraps it around a Sequence object.

If the sequence is already an instance of Sequence, `__init__` will insure that it is at most wrapped exactly once.

If the sequence is a list or tuple, it is set as the sequence.

If it is an iterable, then it is expanded into a list then set to the sequence

If the object does not fit any of these classes, a `TypeError` is thrown

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

**Returns** sequence wrapped in a Sequence

`__iter__` ()

Return iterator of sequence.

**Returns** iterator of sequence

`__module__` = 'functional.pipeline'

`__ne__` (*other*)

Checks for inequality with the sequence's inequality operator.

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

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

`__nonzero__` ()

Returns True if size is not zero.

**Returns** True if size is not zero

**\_\_repr\_\_** ()

Return repr using sequence's repr function.

**Returns** sequence's repr**\_\_reversed\_\_** ()

Return reversed sequence using sequence's reverse function

**Returns** reversed sequence**\_\_str\_\_** ()

Return string using sequence's string function.

**Returns** sequence's string**\_\_weakref\_\_**

list of weak references to the object (if defined)

**\_evaluate** ()

Creates and returns an iterator which applies all the transformations in the lineage

**Returns** iterator over the transformed sequence**\_transform** (*transform*)

Copies the given Sequence and appends new transformation :param transform: transform to apply :return: transformed sequence

**\_unwrap\_sequence** ()

Retrieves the root sequence wrapped by one or more Sequence objects. Will not evaluate lineage, used internally in fetching lineage and the base sequence to use.

**Returns** root sequence**aggregate** (\*args)

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

One Argument: argument specifies a function of the type f(current: B, next: A =&gt; 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

**count** (*func*)

Counts the number of elements in the sequence which satisfy the predicate func.

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

**Parameters** **func** – predicate to count elements on

**Returns** count of elements that satisfy predicate

**dict** (*default=None*)

Converts sequence of (Key, Value) pairs to a dictionary.

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

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

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

**Returns** dictionary from sequence of (Key, Value) elements

**difference** (*other*)

New sequence with unique elements present in sequence but not in other.

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

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

**Returns** difference of sequence and other

**distinct** ()

Returns sequence of distinct elements. Elements must be hashable.

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

**Returns** sequence of distinct elements

**distinct\_by** (*func*)

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

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

**Returns** elements distinct by *func*

**drop** (*n*)

Drop the first *n* elements of the sequence.

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

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

**Returns** sequence without first *n* elements

**drop\_right** (*n*)

Drops the last *n* elements of the sequence.

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

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

**Returns** sequence with last *n* elements dropped

**drop\_while** (*func*)

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

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

**Parameters** *func* – truth returning function

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

**empty** ()

Returns True if the sequence has length zero.

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

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

**Returns** True if sequence length is zero

**enumerate** (*start=0*)

Uses python enumerate to to zip the sequence with indexes starting at start.

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

**Parameters** **start** – Beginning of zip

**Returns** enumerated sequence starting at start

**exists** (*func*)

Returns True if an element in the sequence makes func evaluate to True.

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

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

**Parameters** **func** – existence check function

**Returns** True if any element satisfies func

**filter** (*func*)

Filters sequence to include only elements where func is True.

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

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

**Returns** filtered sequence

**filter\_not** (*func*)

Filters sequence to include only elements where func is False.

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

**Parameters** **func** – function to filter\_not on

**Returns** filtered sequence

**find** (*func*)

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

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



**Parameters** `func` – function to find with

**Returns** first element to satisfy `func` or `None`

**first()**

Returns the first element of the sequence.

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

Raises `IndexError` when the sequence is empty.

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

**Returns** first element of sequence

**flat\_map(func)**

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

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

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

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

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

**Returns** application of `func` to elements followed by flattening

**flatten()**

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

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

**Returns** flattened sequence

**fold\_left(zero\_value, func)**

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

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

**Parameters**

- **zero\_value** – zero value to reduce into
- **func** – Two parameter function as described by function docs

**Returns** value from folding values with `func` into `zero_value` from left to right.

**fold\_right** (*zero\_value, func*)

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

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

**Parameters**

- **zero\_value** – zero value to reduce into
- **func** – Two parameter function as described by function docs

**Returns** value from folding values with `func` into `zero_value` from right to left

**for\_all** (*func*)

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

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

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

**Parameters** **func** – function to check truth value of all elements with

**Returns** True if all elements make `func` evaluate to True

**for\_each** (*func*)

Executes `func` on each element of the sequence.

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

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

**group\_by** (*func*)

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

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

**Parameters** **func** – group by result of this function

**Returns** grouped sequence

**group\_by\_key** ()

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

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

**Returns** sequence grouped by key

**grouped** (*size*)

Partitions the elements into groups of length *size*.

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

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

The last partition will be at least of size 1 and no more than length *size* :param *size*: size of the partitions  
:return: 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()**

Converts sequence to list of elements.

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

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

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

**Returns** list of elements in sequence

**make\_string(separator)**

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

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

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

**Returns** concatenated string separated by separator

**map** (*func*)

Maps *f* onto the elements of the sequence.

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

**Parameters** *func* – function to map with

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

**max** ()

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

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

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

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

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

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

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

**Returns** Maximal value of sequence

**max\_by** (*func*)

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

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

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

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

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

**Parameters** *func* – function to compute max by

**Returns** Maximal element by *func*(element)

**min()**

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

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

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

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

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

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

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

**Returns** Minimal value of sequence

**min\_by(func)**

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

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

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

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

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

**Parameters** **func** – function to compute min by

**Returns** Maximal element by func(element)

**non\_empty()**

Returns True if the sequence does not have length zero.

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

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

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

**order\_by** (*func*)

Orders the input according to func

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

**Parameters** **func** – order by 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*)

Reduce sequence of elements using func.

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

**Parameters** **func** – two parameter, associative reduce function**Returns** reduced value using func



**reduce\_by\_key** (*func*)

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

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

**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**sorted** (*key=None, reverse=False*)

Uses python sort and its passed arguments to sort the input.

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

**Parameters**

- **key** – sort using key function
- **reverse** – return list reversed or not

**Returns** sorted sequence**sum** (*projection=None*)

Takes sum of elements in sequence.

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

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

**Parameters** **projection** – function to project on the sequence before taking the sum**Returns** sum of elements in sequence**symmetric\_difference** (*other*)

New sequence with elements in either sequence or other, but not both.

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

**Parameters** **other** – sequence to perform symmetric difference with**Returns** symmetric difference of sequence and other**tail()**

Returns the sequence, without its first element.

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

**Returns** sequence without first element

**tails()**

Returns consecutive tails of the sequence.

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

**Returns** consecutive tail(s) of the sequence

**take(n)**

Take the first n elements of the sequence.

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

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

**Returns** first n elements of sequence

**take\_while(func)**

Take elements in the sequence until func evaluates to False, then return them.

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

**Parameters** **func** – truth returning function

**Returns** elements taken until func evaluates to False

**to\_csv(path, mode='wb', dialect='excel', \*\*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
- **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*, *mode*='w', *buffering*=-1, *encoding*=None, *errors*=None, *newline*=None)  
Saves the sequence to a file by executing str(self) which becomes str(self.to\_list())

**Parameters**

- **path** – path to write file
- **delimiter** – delimiter to split joined text on. if None, defaults to file.readlines()
- **mode** – file open mode
- **buffering** – passed to builtins.open
- **encoding** – passed to builtins.open
- **errors** – passed to builtins.open
- **newline** – passed to builtins.open

**to\_json** (*path*, *root\_array*=True, *mode*='wb')

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*='w')

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

**to\_list** ()

Converts sequence to list of elements.

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

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

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

**Returns** list of elements in sequence

**to\_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

#### **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** ()

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

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

**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*)

Bases: `object`

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

`__dict__` = `dict_proxy({'__module__': 'functional.lineage', '__getitem__': <function __getitem__ at 0x7efea977d050>,`

`__getitem__` (*item*)

`__init__` (*prior\_lineage=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*)

`cache_scan` ()

`evaluate` (*sequence*)

### 1.2.4 functional.transformations

**class** `functional.transformations.EXECUTION_STRATEGIES`

Bases: `enum.Enum`

`__format__` (*format\_spec*)

`__module__` = 'functional.transformations'

`static __new__` (*value*)

`__reduce_ex__` (*proto*)

`__repr__` ()

`__str__` ()

`__member_map__` = `OrderedDict([('PRE_COMPUTE', <EXECUTION_STRATEGIES.PRE_COMPUTE: 1>))]`

`__member_names__` = ['PRE\_COMPUTE']

**`__member_type__`**  
alias of object

**`__value2member_map__`** = {1: <EXECUTION\_STRATEGIES.PRE\_COMPUTE: 1>}

**class** `functional.transformations.Transformation` (*name, function, execution\_strategies*)  
Bases: tuple

**`__dict__`** = dict\_proxy({'function': <property object at 0x7efea99fbaf8>, '\_\_module\_\_': 'functional.transformations', ...})

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

**`__getstate__`** ()  
Exclude the OrderedDict from pickling

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

**static** **`__new__`** (*\_cls, name, function, execution\_strategies*)  
Create new instance of Transformation(name, function, execution\_strategies)

**`__repr__`** ()  
Return a nicely formatted representation string

**`__slots__`** = ()

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

**`__fields`** = ('name', 'function', 'execution\_strategies')

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

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

**`execution_strategies`**  
Alias for field number 2

**`function`**  
Alias for field number 1

**`name`**  
Alias for field number 0

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

`functional.transformations.symmetric_difference_t` (*other*)  
Transformation for Sequence.symmetric\_difference :param other: sequence to symmetric\_difference with :return: transformation

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

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

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

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

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

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

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

`functional.transformations.zip_with_index_t` ()  
Transformation for Sequence.zip\_with\_index :return: transformation

## 1.2.5 functional.util

**class** `functional.util.ReusableFile` (*path*, *delimiter=None*, *mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*)

Bases: `object`

Class which emulates the builtin file except that calling `iter()` on it will return separate iterators on different file handlers (which are automatically closed when iteration stops). This is useful for allowing a file object to be iterated over multiple times while keep evaluation lazy.

`__dict__` = `dict_proxy({'__module__': 'functional.util', '__iter__': <function __iter__ at 0x7efea99ff5f0>, '__dict__': <`

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

Constructor arguments are passed directly to `builtins.open` :param *path*: passed to `open` :param *delimiter*: passed to `open` :param *mode*: passed to `open` :param *buffering*: passed to `open` :param *encoding*: passed to `open` :param *errors*: passed to `open` :param *newline*: passed to `open` :return: `ReusableFile` from the arguments

`__iter__` ()

Returns a new iterator over the file using the arguments from the constructor. Each call to `__iter__` returns a new iterator independent of all others :return: iterator over file

`__module__` = 'functional.util'

`__weakref__`

list of weak references to the object (if defined)

`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_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



---

## Documentation

---

The best place to see examples of *ScalaFunctional* usage is on the project's github readme page at [github.com/EntilZha/ScalaFunctional](https://github.com/EntilZha/ScalaFunctional). The docs on this site are primarily meant to give comprehensive documentation of every public function and API in *ScalaFunctional*. Its secondary purpose is to document internal methods to make development easier for maintainers and contributors.



**f**

functional.lineage, 42  
functional.pipeline, 5  
functional.streams, 3  
functional.transformations, 42  
functional.util, 46





## Symbols

- `__add__()` (functional.pipeline.Sequence method), 23
- `__bool__()` (functional.pipeline.Sequence method), 23
- `__contains__()` (functional.pipeline.Sequence method), 24
- `__dict__` (functional.lineage.Lineage attribute), 42
- `__dict__` (functional.pipeline.Sequence attribute), 24
- `__dict__` (functional.transformations.Transformation attribute), 43
- `__dict__` (functional.util.ReusableFile attribute), 46
- `__eq__()` (functional.pipeline.Sequence method), 24
- `__format__()` (functional.transformations.EXECUTION\_STRATEGIES method), 42
- `__getitem__()` (functional.lineage.Lineage method), 42
- `__getitem__()` (functional.pipeline.Sequence method), 24
- `__getnewargs__()` (functional.transformations.Transformation method), 43
- `__getstate__()` (functional.transformations.Transformation method), 43
- `__hash__()` (functional.pipeline.Sequence method), 24
- `__init__()` (functional.lineage.Lineage method), 42
- `__init__()` (functional.pipeline.Sequence method), 24
- `__init__()` (functional.util.ReusableFile method), 46
- `__iter__()` (functional.pipeline.Sequence method), 24
- `__iter__()` (functional.util.ReusableFile method), 46
- `__len__()` (functional.lineage.Lineage method), 42
- `__module__` (functional.lineage.Lineage attribute), 42
- `__module__` (functional.pipeline.Sequence attribute), 24
- `__module__` (functional.transformations.EXECUTION\_STRATEGIES attribute), 42
- `__module__` (functional.transformations.Transformation attribute), 43
- `__module__` (functional.util.ReusableFile attribute), 46
- `__ne__()` (functional.pipeline.Sequence method), 24
- `__new__()` (functional.transformations.EXECUTION\_STRATEGIES static method), 42
- `__new__()` (functional.transformations.Transformation static method), 43
- `__nonzero__()` (functional.pipeline.Sequence method), 24
- `__reduce_ex__()` (functional.transformations.EXECUTION\_STRATEGIES method), 42
- `__repr__()` (functional.lineage.Lineage method), 42
- `__repr__()` (functional.pipeline.Sequence method), 24
- `__repr__()` (functional.transformations.EXECUTION\_STRATEGIES method), 42
- `__repr__()` (functional.transformations.Transformation method), 43
- `__reversed__()` (functional.pipeline.Sequence method), 25
- `__slots__` (functional.transformations.Transformation attribute), 43
- `__str__()` (functional.pipeline.Sequence method), 25
- `__str__()` (functional.transformations.EXECUTION\_STRATEGIES method), 42
- `__weakref__` (functional.lineage.Lineage attribute), 42
- `__weakref__` (functional.pipeline.Sequence attribute), 25
- `__weakref__` (functional.util.ReusableFile attribute), 46
- `_asdict()` (functional.transformations.Transformation method), 43
- `_evaluate()` (functional.pipeline.Sequence method), 25
- `_fields` (functional.transformations.Transformation attribute), 43
- `_make()` (functional.transformations.Transformation class method), 43
- `_member_map_` (functional.transformations.EXECUTION\_STRATEGIES attribute), 42
- `_member_names_` (functional.transformations.EXECUTION\_STRATEGIES attribute), 42
- `_member_type_` (functional.transformations.EXECUTION\_STRATEGIES attribute), 42
- `_replace()` (functional.transformations.Transformation method), 43
- `_transform()` (functional.pipeline.Sequence method), 25
- `_unwrap_sequence()` (functional.pipeline.Sequence method), 25
- `_value2member_map_` (functional.transformations.EXECUTION\_STRATEGIES attribute), 42

attribute), 43  
 \_wrap() (in module functional.pipeline), 41

## A

aggregate() (functional.pipeline.Sequence method), 5, 25  
 all() (functional.pipeline.Sequence method), 5, 25  
 any() (functional.pipeline.Sequence method), 5, 25  
 apply() (functional.lineage.Lineage method), 42  
 average() (functional.pipeline.Sequence method), 6, 26

## C

cache() (functional.pipeline.Sequence method), 6, 26  
 cache\_scan() (functional.lineage.Lineage method), 42  
 count() (functional.pipeline.Sequence method), 6, 26  
 csv() (in module functional.streams), 3, 21

## D

dict() (functional.pipeline.Sequence method), 6, 26  
 difference() (functional.pipeline.Sequence method), 6, 26  
 difference\_t() (in module functional.transformations), 43  
 distinct() (functional.pipeline.Sequence method), 6, 27  
 distinct\_by() (functional.pipeline.Sequence method), 7, 27  
 distinct\_by\_t() (in module functional.transformations), 43  
 distinct\_t() (in module functional.transformations), 43  
 drop() (functional.pipeline.Sequence method), 7, 27  
 drop\_right() (functional.pipeline.Sequence method), 7, 27  
 drop\_right\_t() (in module functional.transformations), 43  
 drop\_t() (in module functional.transformations), 43  
 drop\_while() (functional.pipeline.Sequence method), 7, 27  
 drop\_while\_t() (in module functional.transformations), 43

## E

empty() (functional.pipeline.Sequence method), 7, 27  
 enumerate() (functional.pipeline.Sequence method), 7, 28  
 enumerate\_t() (in module functional.transformations), 43  
 evaluate() (functional.lineage.Lineage method), 42  
 EXECUTION\_STRATEGIES (class in functional.transformations), 42  
 execution\_strategies (functional.transformations.Transformation attribute), 43  
 exists() (functional.pipeline.Sequence method), 8, 28

## F

filter() (functional.pipeline.Sequence method), 8, 28  
 filter\_not() (functional.pipeline.Sequence method), 8, 28  
 filter\_not\_t() (in module functional.transformations), 44  
 filter\_t() (in module functional.transformations), 44  
 find() (functional.pipeline.Sequence method), 8, 28  
 first() (functional.pipeline.Sequence method), 8, 29

flat\_map() (functional.pipeline.Sequence method), 9, 29  
 flat\_map\_impl() (in module functional.transformations), 44  
 flat\_map\_t() (in module functional.transformations), 44  
 flatten() (functional.pipeline.Sequence method), 9, 29  
 flatten\_t() (in module functional.transformations), 44  
 fold\_left() (functional.pipeline.Sequence method), 9, 29  
 fold\_right() (functional.pipeline.Sequence method), 9, 30  
 for\_all() (functional.pipeline.Sequence method), 10, 30  
 for\_each() (functional.pipeline.Sequence method), 10, 30  
 function (functional.transformations.Transformation attribute), 43  
 functional.lineage (module), 42  
 functional.pipeline (module), 5, 23  
 functional.streams (module), 3, 21  
 functional.transformations (module), 42  
 functional.util (module), 46

## G

group\_by() (functional.pipeline.Sequence method), 10, 30  
 group\_by\_impl() (in module functional.transformations), 44  
 group\_by\_key() (functional.pipeline.Sequence method), 10, 30  
 group\_by\_key\_impl() (in module functional.transformations), 44  
 group\_by\_key\_t() (in module functional.transformations), 44  
 group\_by\_t() (in module functional.transformations), 44  
 grouped() (functional.pipeline.Sequence method), 10, 30  
 grouped\_impl() (in module functional.transformations), 44  
 grouped\_t() (in module functional.transformations), 44

## H

head() (functional.pipeline.Sequence method), 11, 31  
 head\_option() (functional.pipeline.Sequence method), 11, 31

## I

identity() (in module functional.util), 46  
 init() (functional.pipeline.Sequence method), 11, 31  
 init\_t() (in module functional.transformations), 44  
 inits() (functional.pipeline.Sequence method), 11, 31  
 inits\_t() (in module functional.transformations), 44  
 inner\_join() (functional.pipeline.Sequence method), 11, 31  
 inner\_join\_impl() (in module functional.transformations), 44  
 intersection() (functional.pipeline.Sequence method), 11, 32  
 intersection\_t() (in module functional.transformations), 44

is\_iterable() (in module functional.util), 46  
 is\_primitive() (in module functional.util), 46

## J

join() (functional.pipeline.Sequence method), 12, 32  
 join\_impl() (in module functional.transformations), 44  
 join\_t() (in module functional.transformations), 44  
 json() (in module functional.streams), 3, 22  
 jsonl() (in module functional.streams), 3, 22

## L

last() (functional.pipeline.Sequence method), 12, 32  
 last\_option() (functional.pipeline.Sequence method), 12, 33  
 left\_join() (functional.pipeline.Sequence method), 13, 33  
 len() (functional.pipeline.Sequence method), 13, 33  
 Lineage (class in functional.lineage), 42  
 list() (functional.pipeline.Sequence method), 13, 33

## M

make\_string() (functional.pipeline.Sequence method), 13, 33  
 map() (functional.pipeline.Sequence method), 13, 33  
 map\_t() (in module functional.transformations), 44  
 max() (functional.pipeline.Sequence method), 14, 34  
 max\_by() (functional.pipeline.Sequence method), 14, 34  
 min() (functional.pipeline.Sequence method), 14, 34  
 min\_by() (functional.pipeline.Sequence method), 15, 35

## N

name (functional.transformations.Transformation attribute), 43  
 name() (in module functional.transformations), 45  
 non\_empty() (functional.pipeline.Sequence method), 15, 35

## O

open() (in module functional.streams), 4, 22  
 order\_by() (functional.pipeline.Sequence method), 15, 35  
 order\_by\_t() (in module functional.transformations), 45  
 outer\_join() (functional.pipeline.Sequence method), 16, 36

## P

partition() (functional.pipeline.Sequence method), 16, 36  
 partition\_t() (in module functional.transformations), 45  
 product() (functional.pipeline.Sequence method), 16, 36

## R

range() (in module functional.streams), 4, 22  
 reduce() (functional.pipeline.Sequence method), 16, 36  
 reduce\_by\_key() (functional.pipeline.Sequence method), 16, 36

reduce\_by\_key\_t() (in module functional.transformations), 45

ReusableFile (class in functional.util), 46

reverse() (functional.pipeline.Sequence method), 17, 37  
 reversed\_t() (in module functional.transformations), 45  
 right\_join() (functional.pipeline.Sequence method), 17, 37

## S

select() (functional.pipeline.Sequence method), 17, 37  
 select\_t() (in module functional.transformations), 45  
 seq() (in module functional.streams), 4, 23  
 Sequence (class in functional.pipeline), 5, 23  
 sequence (functional.pipeline.Sequence attribute), 17, 37  
 set() (functional.pipeline.Sequence method), 17, 37  
 size() (functional.pipeline.Sequence method), 17, 37  
 slice() (functional.pipeline.Sequence method), 18, 38  
 slice\_t() (in module functional.transformations), 45  
 sorted() (functional.pipeline.Sequence method), 18, 38  
 sorted\_t() (in module functional.transformations), 45  
 sum() (functional.pipeline.Sequence method), 18, 38  
 symmetric\_difference() (functional.pipeline.Sequence method), 18, 38  
 symmetric\_difference\_t() (in module functional.transformations), 45

## T

tail() (functional.pipeline.Sequence method), 18, 38  
 tail\_t() (in module functional.transformations), 45  
 tails() (functional.pipeline.Sequence method), 19, 39  
 tails\_t() (in module functional.transformations), 45  
 take() (functional.pipeline.Sequence method), 19, 39  
 take\_t() (in module functional.transformations), 45  
 take\_while() (functional.pipeline.Sequence method), 19, 39  
 take\_while\_t() (in module functional.transformations), 45  
 to\_csv() (functional.pipeline.Sequence method), 19, 39  
 to\_dict() (functional.pipeline.Sequence method), 19, 39  
 to\_file() (functional.pipeline.Sequence method), 19, 40  
 to\_json() (functional.pipeline.Sequence method), 20, 40  
 to\_jsonl() (functional.pipeline.Sequence method), 20, 40  
 to\_list() (functional.pipeline.Sequence method), 20, 40  
 to\_set() (functional.pipeline.Sequence method), 20, 40  
 Transformation (class in functional.transformations), 43

## U

union() (functional.pipeline.Sequence method), 21, 41  
 union\_t() (in module functional.transformations), 45

## W

where() (functional.pipeline.Sequence method), 21, 41  
 where\_t() (in module functional.transformations), 45

## Z

zip() (functional.pipeline.Sequence method), 21, 41

zip\_t() (in module functional.transformations), 45

zip\_with\_index() (functional.pipeline.Sequence method),  
21, 41

zip\_with\_index\_t() (in module functional.transformations), 45