lz

Release 0.10.2

Azat Ibrakov

Dec 21, 2019

CONTENTS:

1 Submodules **1**

1.1 filtration module 1
1.2 functional module 2
1.3 iterating module 4
1.4 left module 7
1.5 logical module 7
1.6 replication module 8
1.7 reversal module 9
1.8 right module 9
1.9 sorting module 10
1.10 transposition module 10
1.11 typology module 10

2 Indices and tables **13**

Python Module Index **15**

Index **17**

CHAPTER

ONE

SUBMODULES

1.1 filtration module

lz.filtration.**grabber**(*predicate: Callable[[Domain], bool] = None*) → Callable[[Iterable[Domain]], Iterable[Domain]]

Returns function that selects elements from the beginning of iterable while given predicate is satisfied.

If predicate is not specified than true-like objects are selected.

```
>>> grab_while_true_like = grabber()
>>> list(grab_while_true_like(range(10)))
[]
```

```
>>> from operator import gt
>>> from functools import partial
>>> grab_while_less_than_five = grabber(partial(gt, 5))
>>> list(grab_while_less_than_five(range(10)))
[0, 1, 2, 3, 4]
```

lz.filtration.**kicker**(*predicate: Callable[[Domain], bool] = None*) → Callable[[Iterable[Domain]], Iterable[Domain]]

Returns function that skips elements from the beginning of iterable while given predicate is satisfied.

If predicate is not specified than true-like objects are skipped.

```
>>> kick_while_true_like = kicker()
>>> list(kick_while_true_like(range(10)))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> from operator import gt
>>> from functools import partial
>>> kick_while_less_than_five = kicker(partial(gt, 5))
>>> list(kick_while_less_than_five(range(10)))
[5, 6, 7, 8, 9]
```

lz.filtration.**scavenger**(*predicate: Callable[[Domain], bool] = None*) → Callable[[Iterable[Domain]], Iterable[Domain]]

Returns function that selects elements from iterable which dissatisfy given predicate.

If predicate is not specified than false-like objects are selected.

```
>>> to_false_like = scavenger()
>>> list(to_false_like(range(10)))
[0]
```

```
>>> def is_even(number: int) -> bool:
...     return number % 2 == 0
>>> to_odd = scavenger(is_even)
>>> list(to_odd(range(10)))
[1, 3, 5, 7, 9]
```

lz.filtration.**separator**(*predicate: Callable[[Domain], bool] = None*) → Callable[[Iterable[Domain]], Tuple[Iterable[Domain], Iterable[Domain]]]

Returns function that returns pair of iterables first of which consists of elements that dissatisfy given predicate and second one consists of elements that satisfy given predicate.

```
>>> split_by_truth = separator()
>>> tuple(map(list, split_by_truth(range(10))))
([0], [1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
>>> def is_even(number: int) -> bool:
...     return number % 2 == 0
>>> split_by_evenness = separator(is_even)
>>> tuple(map(list, split_by_evenness(range(10))))
([1, 3, 5, 7, 9], [0, 2, 4, 6, 8])
```

lz.filtration.**sifter**(*predicate: Callable[[Domain], bool] = None*) → Callable[[Iterable[Domain]], Iterable[Domain]]

Returns function that selects elements from iterable which satisfy given predicate.

If predicate is not specified than true-like objects are selected.

```
>>> to_true_like = sifter()
>>> list(to_true_like(range(10)))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> def is_even(number: int) -> bool:
...     return number % 2 == 0
>>> to_even = sifter(is_even)
>>> list(to_even(range(10)))
[0, 2, 4, 6, 8]
```

1.2 functional module

lz.functional.**apply**(*function: Callable[[...], Range], args: Iterable[Domain], kwargs: Dict[str, Any] = mappingproxy({})*) → Range

Calls given function with given positional and keyword arguments.

lz.functional.**cleave**(**functions: Callable[[...], Range]*) → Callable[[...], Iterable[Range]]

Returns function that separately applies given functions to the same arguments.

```
>>> to_min_and_max = cleave(min, max)
>>> list(to_min_and_max(range(10)))
[0, 9]
>>> list(to_min_and_max(range(0), default=None))
[None, None]
```

lz.functional.**combine**(**maps: Callable[[Domain], Range]*) → Callable[[Iterable[Domain]], Iterable[Range]]

Returns function that applies each map to corresponding argument.

```
>>> encoder_decoder = combine(str.encode, bytes.decode)
>>> list(encoder_decoder(['hello', b'world']))
[b'hello', 'world']
```

lz.functional.**compose**(*last_function: Callable[[Any], Range], *front_functions: Callable[[...], Any]*) → Callable[[...], Range]

Returns functions composition.

```
>>> sum_of_first_n_natural_numbers = compose(sum, range)
>>> sum_of_first_n_natural_numbers(10)
45
```

lz.functional.**curry**(*function: Callable[[...], Range], *, signature: Optional[paradigm.models.Base] = None*) → lz.functional.Curry

Returns curried version of given function.

```
>>> curried_pow = curry(pow)
>>> two_to_power = curried_pow(2)
>>> two_to_power(10)
1024
```

lz.functional.**flatmap**(*function: Callable[[Domain], Iterable[Range]], *iterables: Iterable[Domain]*) → Iterable[Range]

Applies given function to the arguments aggregated from given iterables and concatenates results into plain iterable.

```
>>> list(flatmap(range, range(5)))
[0, 0, 1, 0, 1, 2, 0, 1, 2, 3]
```

lz.functional.**flip**(*function: Callable[[...], Range]*) → Callable[[...], Range]

Returns function with positional arguments flipped.

```
>>> flipped_power = flip(pow)
>>> flipped_power(2, 4)
16
```

lz.functional.**identity**(*argument: Domain*) → Domain

Returns object itself.

```
>>> identity(0)
0
```

lz.functional.**pack**(*function: Callable[[...], Range]*) → Callable[[Iterable[Domain]], Range]

Returns function that works with single iterable parameter by unpacking elements to given function.

```
>>> packed_int = pack(int)
>>> packed_int(['10'])
10
>>> packed_int(['10'], {'base': 2})
2
```

lz.functional.**to_constant**(*object_: Domain*) → Callable[[...], Domain]

Returns function that always returns given object.

```
>>> always_zero = to_constant(0)
>>> always_zero()
0
```

(continues on next page)

(continued from previous page)

```
>>> always_zero(1)
0
>>> always_zero(how_about=2)
0
```

1.3 iterating module

`lz.iterating.`**`capacity`**(*iterable: Iterable[Any]*) → int
Returns number of elements in iterable.

```
>>> capacity(range(0))
0
>>> capacity(range(10))
10
```

`lz.iterating.`**`chop`**(*iterable: Iterable[Domain], *, size: int*) → Iterable[Sequence[Domain]]
Splits iterable into chunks of given size.

`lz.iterating.`**`chopper`**(*size: int*) → Callable[[Iterable[Domain]], Iterable[Sequence[Domain]]]
Returns function that splits iterable into chunks of given size.

```
>>> in_three = chopper(3)
>>> list(map(tuple, in_three(range(10))))
[(0, 1, 2), (3, 4, 5), (6, 7, 8), (9,)]
```

`lz.iterating.`**`cut`**(*iterable: Iterable[Domain], *, slice_: slice*) → Iterable[Domain]
Selects elements from iterable based on given slice.

Slice fields supposed to be unset or non-negative since it is hard to evaluate negative indices/step for arbitrary iterable which may be potentially infinite or change previous elements if iterating made backwards.

`lz.iterating.`**`cutter`**(*slice_: slice*) → Callable[[Iterable[Domain]], Iterable[Domain]]
Returns function that selects elements from iterable based on given slice.

```
>>> to_first_triplet = cutter(slice(3))
>>> list(to_first_triplet(range(10)))
[0, 1, 2]
```

```
>>> to_second_triplet = cutter(slice(3, 6))
>>> list(to_second_triplet(range(10)))
[3, 4, 5]
```

```
>>> cut_out_every_third = cutter(slice(0, None, 3))
>>> list(cut_out_every_third(range(10)))
[0, 3, 6, 9]
```

`lz.iterating.`**`expand`**(*object_: Domain*) → Iterable[Domain]
Wraps object into iterable.

```
>>> list(expand(0))
[0]
```

`lz.iterating.`**`first`**(*iterable: Iterable[Domain]*) → Domain
Returns first element of iterable.

```
>>> first(range(10))
0
```

lz.iterating.**flatmapper**(*map_: Callable[[Domain], Iterable[Range]]*) → Callable[[Iterable[Domain]], Iterable[Range]]
Returns function that applies map to the each element of iterable and flattens results.

```
>>> relay = flatmapper(range)
>>> list(relay(range(5)))
[0, 0, 1, 0, 1, 2, 0, 1, 2, 3]
```

lz.iterating.**flatten**(*iterable: Iterable[Iterable[Domain]]*) → Iterable[Domain]
Returns plain iterable from iterable of iterables.

```
>>> list(flatten([range(5), range(10, 20)]))
[0, 1, 2, 3, 4, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

lz.iterating.**group_by**(*iterable: Iterable[Domain], *, key: Callable[[Domain], Hashable], mapping_cls: Type[MutableMapping]*) → Iterable[Tuple[Hashable, Iterable[Domain]]]
Groups iterable elements based on given key.

lz.iterating.**grouper**(*key: Callable[[Domain], Hashable], *, mapping_cls: Type[MutableMapping] = <class 'collections.OrderedDict'>*) → Callable[[Iterable[Domain]], Iterable[Tuple[Hashable, Iterable[Domain]]]]
Returns function that groups iterable elements based on given key.

```
>>> group_by_absolute_value = grouper(abs)
>>> list(group_by_absolute_value(range(-5, 5)))
[(5, [-5]), (4, [-4, 4]), (3, [-3, 3]), (2, [-2, 2]), (1, [-1, 1]), (0, [0])]
```

```
>>> def modulo_two(number: int) -> int:
...     return number % 2
>>> group_by_evenness = grouper(modulo_two)
>>> list(group_by_evenness(range(10)))
[(0, [0, 2, 4, 6, 8]), (1, [1, 3, 5, 7, 9])]
```

lz.iterating.**header**(*size: int*) → Callable[[Iterable[Domain]], Iterable[Domain]]
Returns function that selects elements from the beginning of iterable. Resulted iterable will have size not greater than given one.

```
>>> to_first_pair = header(2)
>>> list(to_first_pair(range(10)))
[0, 1]
```

lz.iterating.**in_four**(*iterable: Iterable[Domain], *, size: int = 4*) → Iterable[Sequence[Domain]]
Splits iterable into chunks of size 4.

lz.iterating.**in_three**(*iterable: Iterable[Domain], *, size: int = 3*) → Iterable[Sequence[Domain]]
Splits iterable into chunks of size 3.

lz.iterating.**in_two**(*iterable: Iterable[Domain], *, size: int = 2*) → Iterable[Sequence[Domain]]
Splits iterable into chunks of size 2.

lz.iterating.**interleave**(*iterable: Iterable[Iterable[Domain]]*) → Iterable[Domain]
Interleaves elements from given iterable of iterables.

```
>>> list(interleave([range(5), range(10, 20)]))
[0, 10, 1, 11, 2, 12, 3, 13, 4, 14, 15, 16, 17, 18, 19]
```

lz.iterating.**last**(*iterable: Iterable[Domain]*) → Domain
Returns last element of iterable.

```
>>> last(range(10))
9
```

lz.iterating.**mapper**(*map_: Callable[[Domain], Range]*) → Callable[[Iterable[Domain]], Iterable[Range]]
Returns function that applies given map to the each element of iterable.

```
>>> to_str = mapper(str)
>>> list(to_str(range(10)))
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

lz.iterating.**pairwise**(*iterable: Iterable[Domain], *, size: int = 2*) → Iterable[Tuple[Domain, ...]]
Slides over iterable with window of size 2.

lz.iterating.**quadruplewise**(*iterable: Iterable[Domain], *, size: int = 4*) → Iterable[Tuple[Domain, ...]]
Slides over iterable with window of size 4.

lz.iterating.**slide**(*iterable: Iterable[Domain], *, size: int*) → Iterable[Tuple[Domain, ...]]
Slides over iterable with window of given size.

lz.iterating.**slider**(*size: int*) → Callable[[Iterable[Domain]], Iterable[Tuple[Domain, ...]]]
Returns function that slides over iterable with window of given size.

```
>>> pairwise = slider(2)
>>> list(pairwise(range(10)))
[(0, 1), (1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 7), (7, 8), (8, 9)]
```

lz.iterating.**trail**(*iterable: Iterable[Domain], *, size: int*) → Iterable[Domain]
Selects elements from the end of iterable. Resulted iterable will have size not greater than given one.

lz.iterating.**trailer**(*size: int*) → Callable[[Iterable[Domain]], Iterable[Domain]]
Returns function that selects elements from the end of iterable. Resulted iterable will have size not greater than given one.

```
>>> to_last_pair = trailer(2)
>>> list(to_last_pair(range(10)))
[8, 9]
```

lz.iterating.**triplewise**(*iterable: Iterable[Domain], *, size: int = 3*) → Iterable[Tuple[Domain, ...]]
Slides over iterable with window of size 3.

1.4 left module

lz.left.**accumulator**(*function: Callable[[Range, Domain], Range], initial: Range*) → Callable[[Iterable[Domain]], Iterable[Range]]

Returns function that yields cumulative results of given binary function starting from given initial object in direction from left to right.

```
>>> import math
>>> to_pi_approximations = accumulator(round, math.pi)
>>> list(to_pi_approximations(range(5, 0, -1)))
[3.141592653589793, 3.14159, 3.1416, 3.142, 3.14, 3.1]
```

lz.left.**applier**(*function: Callable[[...], Range], *args: Domain, **kwargs: Domain*) → Callable[[...], Range]

Returns function that behaves like given function with given arguments partially applied. Given positional arguments will be added to the left end.

```
>>> count_from_zero_to = applier(range, 0)
>>> list(count_from_zero_to(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

lz.left.**attach**(*iterable: Iterable[Domain], object_: Domain*) → Iterable[Domain]

Prepends given object to the iterable.

lz.left.**attacher**(*object_: Domain*) → Callable[[Iterable[Domain]], Iterable[Domain]]

Returns function that prepends given object to iterable.

```
>>> attach_hundred = attacher(100)
>>> list(attach_hundred(range(10)))
[100, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

lz.left.**folder**(*function: Callable[[Range, Domain], Range], initial: Range*) → Callable[[Iterable[Domain]], Range]

Returns function that cumulatively applies given binary function starting from given initial object in direction from left to right.

```
>>> to_sum_evaluation_order = folder('({} + {})'.format, 0)
>>> to_sum_evaluation_order(range(1, 10))
'(((((((((0 + 1) + 2) + 3) + 4) + 5) + 6) + 7) + 8) + 9)'
```

1.5 logical module

lz.logical.**conjoin**(**predicates: Callable[[Domain], bool]*) → Callable[[Domain], bool]

Returns conjunction of given predicates.

```
>>> is_valid_constant_identifier = conjoin(str.isupper, str.isidentifier)
>>> is_valid_constant_identifier('SECOND_SECTION')
True
>>> is_valid_constant_identifier('2ND_SECTION')
False
```

lz.logical.**disjoin**(**predicates: Callable[[Domain], bool]*) → Callable[[Domain], bool]

Returns disjunction of given predicates.

```
>>> alphabetic_or_numeric = disjoin(str.isalpha, str.isnumeric)
>>> alphabetic_or_numeric('Hello')
True
>>> alphabetic_or_numeric('42')
True
>>> alphabetic_or_numeric('Hello42')
False
```

lz.logical.**exclusive_disjoin**(**predicates: Callable[[Domain], bool]*) → Callable[[Domain], bool]

Returns exclusive disjunction of given predicates.

```
>>> from keyword import iskeyword
>>> valid_object_name = exclusive_disjoin(str.isidentifier, iskeyword)
>>> valid_object_name('valid_object_name')
True
>>> valid_object_name('_')
True
>>> valid_object_name('1')
False
>>> valid_object_name('lambda')
False
```

lz.logical.**negate**(*predicate: Callable[[Domain], bool]*) → Callable[[Domain], bool]

Returns negated version of given predicate.

```
>>> from lz.logical import negate
>>> false_like = negate(bool)
>>> false_like([])
True
>>> false_like([0])
False
```

1.6 replication module

lz.replication.**duplicate**(*object_: Domain*, *, *count: int = 2*) → Iterable[Domain]

Duplicates given object.

lz.replication.**replicate**(*object_: Domain*, *, *count: int*) → Iterable[Domain]

Returns given number of object replicas.

lz.replication.**replicator**(*count: int*) → Callable[[Domain], Iterable[Domain]]

Returns function that replicates passed object.

```
>>> triplicate = replicator(3)
>>> list(map(tuple, triplicate(range(5))))
[(0, 1, 2, 3, 4), (0, 1, 2, 3, 4), (0, 1, 2, 3, 4)]
```

1.7 reversal module

lz.reversal.**reverse**(*object_: Domain*, ***_: Any*) → Range
Returns reversed object.

```
>>> list(reverse(range(10)))
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> import io
>>> list(reverse(io.BytesIO(b'Hello\nWorld!')))
[b'World!', b'Hello\n']
```

1.8 right module

lz.right.**accumulator**(*function: Callable[[Domain, Range], Range], initial: Range*) → Callable[[Iterable[Domain]], Iterable[Iterable[Range]]]
Returns function that yields cumulative results of given binary function starting from given initial object in direction from right to left.

```
>>> def to_next_fraction(partial_denominator: int,
...                      reciprocal: float) -> float:
...     return partial_denominator + 1 / reciprocal
>>> to_simple_continued_fractions = accumulator(to_next_fraction, 1)
>>> from itertools import repeat
>>> [round(fraction, 4)
...  for fraction in to_simple_continued_fractions(list(repeat(1, 10)))]
[1, 2.0, 1.5, 1.6667, 1.6, 1.625, 1.6154, 1.619, 1.6176, 1.6182, 1.618]
```

lz.right.**applier**(*function: Callable[[...], Range], *args: Domain, **kwargs: Domain*) → Callable[[...], Range]
Returns function that behaves like given function with given arguments partially applied. Given positional arguments will be added to the right end.

```
>>> square = applier(pow, 2)
>>> square(10)
100
```

lz.right.**attach**(*iterable: Iterable[Domain], object_: Domain*) → Iterable[Domain]
Appends given object to the iterable.

lz.right.**attacher**(*object_: Domain*) → Callable[[Iterable[Domain]], Iterable[Domain]]
Returns function that appends given object to iterable.

```
>>> attach_hundred = attacher(100)
>>> list(attach_hundred(range(10)))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 100]
```

lz.right.**folder**(*function: Callable[[Domain, Range], Range], initial: Range*) → Callable[[Iterable[Domain]], Range]
Returns function that cumulatively applies given binary function starting from given initial object in direction from right to left.

```
>>> to_sum_evaluation_order = folder('({} + {})'.format, 0)
>>> to_sum_evaluation_order(range(1, 10))
'(1 + (2 + (3 + (4 + (5 + (6 + (7 + (8 + (9 + 0)))))))))'
```

1.9 sorting module

lz.sorting.**register_implementation**(*algorithm: str, implementation: Optional[Callable[[...], Iterable[Domain]]] = None, *, stable: bool = False, overwrite: bool = False*) → Union[Callable[[Callable[[...], Iterable[Domain]]], Callable[[...], Iterable[Domain]]], Callable[[...], Iterable[Domain]]]

Registers implementation of sorting algorithm.

```
>>> from typing import Any
>>> @register_implementation('CUSTOMSORTING')
... def custom_sorting(iterable: Iterable[Domain],
...                    *,
...                    key: Optional[Map[Domain, Any]] = None
...                    ) -> Iterable[Domain]:
...     ...
```

lz.sorting.**sorter**(*, *algorithm: str = 'TIMSORT', key: Optional[Callable[[Domain], lz.hints.Sortable]] = None*) → Callable[[Iterable[Domain]], Iterable[Domain]]

Returns function that generates sorted iterable by given key with specified algorithm.

```
>>> sort = sorter()
>>> sort('Hello World!')
[' ', '!', 'H', 'W', 'd', 'e', 'l', 'l', 'l', 'o', 'o', 'r']
```

1.10 transposition module

lz.transposition.**transpose**(*object_: Domain*) → Range

Transposes given object.

```
>>> list(map(tuple, transpose(zip(range(10), range(10, 20)))))
[(0, 1, 2, 3, 4, 5, 6, 7, 8, 9), (10, 11, 12, 13, 14, 15, 16, 17, 18, 19)]
```

1.11 typology module

lz.typology.**instance_of**(**types: type*) → Callable[[Domain], bool]

Creates predicate that checks if object is instance of given types.

```
>>> is_any_string = instance_of(str, bytes, bytearray)
>>> is_any_string(b'')
True
>>> is_any_string('')
True
>>> is_any_string(1)
False
```

lz.typology.**subclass_of**(**types: type*) → Callable[[Domain], bool]

Creates predicate that checks if type is subclass of given types.

```
>>> is_metaclass = subclass_of(type)
>>> is_metaclass(type)
```

(continues on next page)

(continued from previous page)

```
True
>>> is_metaclass(object)
False
```

Note: If member is not listed in documentation it should be considered as implementation detail that can change and should not be relied upon.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

l

lz.filtration, 1
lz.functional, 2
lz.iterating, 4
lz.left, 7
lz.logical, 7
lz.replication, 8
lz.reversal, 9
lz.right, 9
lz.sorting, 10
lz.transposition, 10
lz.typology, 10

INDEX

A

accumulator() (*in module lz.left*), 7
accumulator() (*in module lz.right*), 9
applier() (*in module lz.left*), 7
applier() (*in module lz.right*), 9
apply() (*in module lz.functional*), 2
attach() (*in module lz.left*), 7
attach() (*in module lz.right*), 9
attacher() (*in module lz.left*), 7
attacher() (*in module lz.right*), 9

C

capacity() (*in module lz.iterating*), 4
chop() (*in module lz.iterating*), 4
chopper() (*in module lz.iterating*), 4
cleave() (*in module lz.functional*), 2
combine() (*in module lz.functional*), 2
compose() (*in module lz.functional*), 3
conjoin() (*in module lz.logical*), 7
curry() (*in module lz.functional*), 3
cut() (*in module lz.iterating*), 4
cutter() (*in module lz.iterating*), 4

D

disjoin() (*in module lz.logical*), 7
duplicate() (*in module lz.replication*), 8

E

exclusive_disjoin() (*in module lz.logical*), 8
expand() (*in module lz.iterating*), 4

F

first() (*in module lz.iterating*), 4
flatmap() (*in module lz.functional*), 3
flatmapper() (*in module lz.iterating*), 5
flatten() (*in module lz.iterating*), 5
flip() (*in module lz.functional*), 3
folder() (*in module lz.left*), 7
folder() (*in module lz.right*), 9

G

grabber() (*in module lz.filtration*), 1
group_by() (*in module lz.iterating*), 5
grouper() (*in module lz.iterating*), 5

H

header() (*in module lz.iterating*), 5

I

identity() (*in module lz.functional*), 3
in_four() (*in module lz.iterating*), 5
in_three() (*in module lz.iterating*), 5
in_two() (*in module lz.iterating*), 5
instance_of() (*in module lz.typology*), 10
interleave() (*in module lz.iterating*), 5

K

kicker() (*in module lz.filtration*), 1

L

last() (*in module lz.iterating*), 6
lz.filtration (*module*), 1
lz.functional (*module*), 2
lz.iterating (*module*), 4
lz.left (*module*), 7
lz.logical (*module*), 7
lz.replication (*module*), 8
lz.reversal (*module*), 9
lz.right (*module*), 9
lz.sorting (*module*), 10
lz.transposition (*module*), 10
lz.typology (*module*), 10

M

mapper() (*in module lz.iterating*), 6

N

negate() (*in module lz.logical*), 8

P

pack() (*in module lz.functional*), 3
pairwise() (*in module lz.iterating*), 6

Q

quadruplewise() (*in module lz.iterating*), 6

R

register_implementation() (*in module lz.sorting*), 10
replicate() (*in module lz.replication*), 8
replicator() (*in module lz.replication*), 8
reverse() (*in module lz.reversal*), 9

S

scavenger() (*in module lz.filtration*), 1
separator() (*in module lz.filtration*), 2
sifter() (*in module lz.filtration*), 2
slide() (*in module lz.iterating*), 6
slider() (*in module lz.iterating*), 6
sorter() (*in module lz.sorting*), 10
subclass_of() (*in module lz.typology*), 10

T

to_constant() (*in module lz.functional*), 3
trail() (*in module lz.iterating*), 6
trailer() (*in module lz.iterating*), 6
transpose() (*in module lz.transposition*), 10
triplewise() (*in module lz.iterating*), 6