
lz

Release 3.0.0

Azat Ibrakov

Feb 23, 2023

CONTENTS:

1	Submodules	1
1.1	filtration module	1
1.2	functional module	3
1.3	iterating module	5
1.4	left module	8
1.5	logical module	9
1.6	replication module	10
1.7	reversal module	10
1.8	right module	11
1.9	transposition module	12
1.10	typology module	12
2	Indices and tables	15
	Python Module Index	17
	Index	19

SUBMODULES

1.1 filtration module

`lz.filtration.grab(_predicate: Callable[[_T], bool], _value: Iterable[_T]) → Iterable[_T]`

Selects elements from the beginning of iterable while given predicate is satisfied.

```
>>> grab_while_true_like = grabber(bool)
>>> 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.grabber(_predicate: Callable[[_T], bool]) → Callable[[Iterable[_T]], Iterable[_T]]`

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

```
>>> grab_while_true_like = grabber(bool)
>>> 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.kick(_predicate: Callable[[_T], bool], _value: Iterable[_T]) → Iterable[_T]`

Skips elements from the beginning of iterable while given predicate is satisfied.

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

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

`lz.filtration.kicker(_predicate: Callable[[_T], bool]) → Callable[[Iterable[_T]], Iterable[_T]]`

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

```
>>> kick_while_true_like = kicker(bool)
>>> 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.scavenge(_predicate: Callable[[_T], bool], _value: Iterable[_T]) → Iterable[_T]`

Selects elements from iterable which dissatisfy given predicate.

```
>>> list(scavenge(bool, range(10)))
[0]
```

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

`lz.filtration.scavenger(_predicate: Callable[[_T], bool]) → Callable[[Iterable[_T]], Iterable[_T]]`

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

```
>>> to_false_like = scavenger(bool)
>>> 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.separate(_predicate: Callable[[_T], bool], _value: Iterable[_T]) → Tuple[Iterable[_T], Iterable[_T]]`

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

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

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

`lz.filtration.separator(_predicate: Callable[[_T], bool]) → Callable[[Iterable[_T]], Tuple[Iterable[_T], Iterable[_T]]]`

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(bool)
>>> 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.sift(_predicate: Callable[[_T], bool], _value: Iterable[_T]) → Iterable[_T]`

Selects elements from iterable which satisfy given predicate.

```
>>> list(sift(bool, range(10)))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

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

`lz.filtration.sifter(_predicate: Callable[[_T], bool]) → Callable[[Iterable[_T]], Iterable[_T]]`

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

```
>>> to_true_like = sifter(bool)
>>> 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.call(_function: Callable[[_Params], _T2], _args: _Params, _kwargs: _Params = mappingproxy({})) → _T2`

Calls given function with given positional and keyword arguments.

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

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]
```

(continues on next page)

(continued from previous page)

```
>>> list(to_min_and_max(range(0), default=None))
[None, None]
```

`lz.functional.combine(*maps: Callable[[_T], _T2]) → Callable[[...], Tuple[_T2, ...]]`

Returns function that applies each map to corresponding argument.

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

`lz.functional.compose(_last_function: Callable[[_T2], _T3], _penult_function: Callable[[...], _T2],
*_rest_functions: Callable[[...], Any]) → Callable[_Params], _T3]`

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[[...], _T2]) → Curry[_Arg, _KwArg, _T2]`

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[[_T], Iterable[_T2]], *iterables: Iterable[_T]) → Iterable[_T2]`

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[[...], _T2]) → Callable[[...], _T2]`

Returns function with positional arguments flipped.

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

`lz.functional.identity(_value: _T) → _T`

Returns object itself.

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

`lz.functional.pack(function: Callable[_Params], _T2]) → Callable[_T, _T2], _T2]`

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

```
>>> packed_int = pack(int)
>>> packed_int(['10'])
10
```

(continues on next page)

(continued from previous page)

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

`lz.functional.to_constant(_value: _T) → Callable[[...], _T]`

Returns function that always returns given value.

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

1.3 iterating module

`lz.iterating.capacity(_value: Any) → int`

`lz.iterating.capacity(_iterable: Iterable[Any]) → int`

`lz.iterating.capacity(_iterable: Sized) → int`

Returns number of elements in value.

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

`lz.iterating.chop(_iterable: Iterable[_T], *, size: int) → Iterable[Sequence[_T]]`

`lz.iterating.chop(_iterable: Sequence[_T], *, size: int) → Iterable[Sequence[_T]]`

`lz.iterating.chop(_iterable: Iterable[_T], *, size: int) → Iterable[Sequence[_T]]`

Splits iterable into chunks of given size.

`lz.iterating.chopper(_size: int) → Callable[[Iterable[_T]], Iterable[Sequence[_T]]]`

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[_T], *, slice_: slice) → Iterable[_T]`

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[_T]], Iterable[_T]]`

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(_value: _T) → Iterable[_T]`

Wraps value into iterable.

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

`lz.iterating.first(_iterable: Iterable[_T]) → _T`

Returns first element of iterable.

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

`lz.iterating.flatmapper(_map: Callable[[_T], Iterable[_T2]]) → Callable[[Iterable[_T]], Iterable[_T2]]`

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[_T]]) → Iterable[_T]`

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[_T], *, key: Callable[[_T], Hashable]) → Iterable[Tuple[Hashable, Iterable[_T]]]`

Groups iterable elements based on given key.

`lz.iterating.grouper(_key: Callable[[_T], Hashable]) → Callable[[Iterable[_T]], Iterable[Tuple[Hashable, Iterable[_T]]]]`

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[_T]], Iterable[_T]]`

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[_T], *, size: int = 4) → Iterable[Sequence[_T]]`

Splits iterable into chunks of given size.

`lz.iterating.in_three(_iterable: Iterable[_T], *, size: int = 3) → Iterable[Sequence[_T]]`

Splits iterable into chunks of given size.

`lz.iterating.in_two(_iterable: Iterable[_T], *, size: int = 2) → Iterable[Sequence[_T]]`

Splits iterable into chunks of given size.

`lz.iterating.interleave(_iterable: Iterable[Iterable[_T]]) → Iterable[_T]`

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[_T]) → _T`

Returns last element of iterable.

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

`lz.iterating.mapper(_map: Callable[[_T], _T2]) → Callable[[Iterable[_T]], Iterable[_T2]]`

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[_T], *, size: int = 2) → Iterable[Tuple[_T, ...]]`

Slides over iterable with window of given size.

`lz.iterating.quadruplewise(_iterable: Iterable[_T], *, size: int = 4) → Iterable[Tuple[_T, ...]]`

Slides over iterable with window of given size.

`lz.iterating.slide(_iterable: Iterable[_T], *, size: int) → Iterable[Tuple[_T, ...]]`

Slides over iterable with window of given size.

`lz.iterating.slider(_size: int) → Callable[[Iterable[_T]], Iterable[Tuple[_T, ...]]]`

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[_T], *, size: int) → Iterable[_T]`

`lz.iterating.trail(iterable: Sequence[_T], *, size: int) → Sequence[_T]`

`lz.iterating.trail(_iterable: Iterable[_T], *, size: int) → Iterable[_T]`

Selects elements from the end of iterable. Resulted iterable will have size not greater than given one.

`lz.iterating.trailer(_size: int) → Callable[[Iterable[_T]], Iterable[_T]]`

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[_T], *, size: int = 3) → Iterable[Tuple[_T, ...]]`

Slides over iterable with window of given size.

1.4 left module

`lz.left.accumulate(_function: Callable[[_T2, _T1], _T2], _initial: _T2, _iterable: Iterable[_T1]) → Iterable[_T2]`

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

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

`lz.left.accumulator(_function: Callable[[_T2, _T1], _T2], _initial: _T2) → Callable[[Iterable[_T1]], Iterable[_T2]]`

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[[...], _T2], *args: _T1, **kwargs: _T1) → Callable[[...], _T2]`

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(_target: Iterable[_T1], _value: _T1) → Iterable[_T1]`

`lz.left.attach(_target: List[_T1], _value: _T1) → List[_T1]`

`lz.left.attach(_target: Tuple[_T1, ...], _value: _T1) → Tuple[_T1, ...]`

Prepends given value to the target.

`lz.left.attacher(_value: _T1) → Callable[[Iterable[_T1]], Iterable[_T1]]`

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.fold(_function: Callable[[_T2, _T1], _T2], _initial: _T2, _iterable: Iterable[_T1]) → _T2`

Cumulatively applies given binary function starting from given initial object in direction from left to right.

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

`lz.left.folder(_function: Callable[[_T2, _T1], _T2], _initial: _T2) → Callable[[Iterable[_T1]], _T2]`

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[[_Params], bool]) → Callable[[_Params], bool]`

Returns conjunction of given predicates.

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

`lz.logical.disjoin(*predicates: Callable[[_Params], bool]) → Callable[[_Params], 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[[_Params], bool]) → Callable[[_Params], 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')
```

(continues on next page)

(continued from previous page)

```
False
>>> valid_object_name('lambda')
False
```

`lz.logical.negate(predicate: Callable[[_Params], bool]) → Callable[[_Params], bool]`

Returns negated version of given predicate.

```
>>> false_like = negate(bool)
>>> false_like([])
True
>>> false_like([0])
False
```

1.6 replication module

`lz.replication.duplicate(_value: Any, *, count: int = 2) → Iterable[Any]`

Duplicates given object.

`lz.replication.replicate(_value: Any, *, count: int) → Iterable[Any]`

`lz.replication.replicate(_value: _Immutable, *, count: int) → Iterable[_Immutable]`

`lz.replication.replicate(_value: _Immutable, *, count: int) → Iterable[_Immutable]`

`lz.replication.replicate(_value: Iterable[_T], *, count: int) → Iterable[Iterable[_T]]`

`lz.replication.replicate(_value: bytearray, *, count: int) → Iterable[bytearray]`

`lz.replication.replicate(_value: FrozenSet[_T], *, count: int) → Iterable[FrozenSet[_T]]`

`lz.replication.replicate(_value: List[_T], *, count: int) → Iterable[List[_T]]`

`lz.replication.replicate(_value: Set[_T], *, count: int) → Iterable[Set[_T]]`

`lz.replication.replicate(_value: Tuple[_T, ...], *, count: int) → Iterable[Tuple[_T, ...]]`

`lz.replication.replicate(_value: Dict[_Key, _Value], *, count: int) → Iterable[Dict[_Key, _Value]]`

Returns given number of object replicas.

`lz.replication.replicator(count: int) → Callable[[_T], Iterable[_T]]`

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(_value: Any) → Any`

`lz.reversal.reverse(_value: Sequence[_T]) → Sequence[_T]`

`lz.reversal.reverse(_value: Reversible[_T]) → Iterable[_T]`

`lz.reversal.reverse(_value: TextIO, *, batch_size: int = 8192, lines_separator: Optional[str] = None, keep_lines_separator: bool = True) → Iterable[str]`

`lz.reversal.reverse(_value: BinaryIO, *, batch_size: int = 8192, lines_separator: Optional[bytes] = None, keep_lines_separator: bool = True, code_unit_size: int = 1) → Iterable[bytes]`

`lz.reversal.reverse(_value: BinaryIO, *, batch_size: int = 8192, lines_separator: Optional[bytes] = None, keep_lines_separator: bool = True, code_unit_size: int = 1) → Iterable[bytes]`

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']
```

`lz.reversal.reverse_bytes_stream(_value: BinaryIO, *, batch_size: int = 8192, lines_separator: Optional[bytes] = None, keep_lines_separator: bool = True, code_unit_size: int = 1) → Iterable[bytes]`

Returns reversed byte stream.

`lz.reversal.reverse_file_object(_value: TextIO, *, batch_size: int = 8192, lines_separator: Optional[str] = None, keep_lines_separator: bool = True) → Iterable[str]`

Returns reversed file object.

1.8 right module

`lz.right.accumulate(_function: Callable[[_T1, _T2], _T2], _initial: _T2, _iterable: Iterable[_T1]) → Iterable[_T2]`

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
>>> from itertools import repeat
>>> [round(fraction, 4)
...  for fraction in accumulate(to_next_fraction, 1, 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.accumulator(_function: Callable[[_T1, _T2], _T2], _initial: _T2) → Callable[[Iterable[_T1]], Iterable[Iterable[_T2]]]`

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[[...], _T2], *args: Any, **kwargs: Any) → Callable[[...], _T2]`

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[_T1], _value: _T1) → Iterable[_T1]`
`lz.right.attach(_iterable: List[_T1], _value: _T1) → List[_T1]`
`lz.right.attach(_iterable: Tuple[_T1, ...], _value: _T1) → Tuple[_T1, ...]`
 Appends given object to the iterable.
`lz.right.attacher(_value: _T1) → Callable[[Iterable[_T1]], Iterable[_T1]]`
 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.fold(_function: Callable[[_T2, _T1], _T2], _initial: _T2, _iterable: Iterable[_T1]) → _T2`
 Cumulatively applies given binary function starting from given initial object in direction from left to right.

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

`lz.right.folder(_function: Callable[[_T1, _T2], _T2], _initial: _T2) → Callable[[Iterable[_T1]], _T2]`
 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 transposition module

`lz.transposition.transpose(_value: Collection[Iterable[_T]]) → Iterable[Collection[_T]]`
`lz.transposition.transpose(_value: Iterator[Collection[_T]]) → Collection[Iterable[_T]]`
 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)]
```

1.10 typology module

`lz.typology.instance_of(*types: type) → Callable[[Any], 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
```

(continues on next page)

(continued from previous page)

```
>>> is_any_string('')
True
>>> is_any_string(1)
False
```

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

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

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

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

PYTHON MODULE INDEX

|
lz.filtration, 1
lz.functional, 3
lz.iterating, 5
lz.left, 8
lz.logical, 9
lz.replication, 10
lz.reversal, 10
lz.right, 11
lz.transposition, 12
lz.typology, 12

A

accumulate() (in module *lz.left*), 8
 accumulate() (in module *lz.right*), 11
 accumulator() (in module *lz.left*), 8
 accumulator() (in module *lz.right*), 11
 applier() (in module *lz.left*), 8
 applier() (in module *lz.right*), 11
 attach() (in module *lz.left*), 8
 attach() (in module *lz.right*), 12
 attacher() (in module *lz.left*), 8
 attacher() (in module *lz.right*), 12

C

call() (in module *lz.functional*), 3
 capacity() (in module *lz.iterating*), 5
 chop() (in module *lz.iterating*), 5
 chopper() (in module *lz.iterating*), 5
 cleave() (in module *lz.functional*), 3
 combine() (in module *lz.functional*), 4
 compose() (in module *lz.functional*), 4
 conjoin() (in module *lz.logical*), 9
 curry() (in module *lz.functional*), 4
 cut() (in module *lz.iterating*), 5
 cutter() (in module *lz.iterating*), 5

D

disjoin() (in module *lz.logical*), 9
 duplicate() (in module *lz.replication*), 10

E

exclusive_disjoin() (in module *lz.logical*), 9
 expand() (in module *lz.iterating*), 6

F

first() (in module *lz.iterating*), 6
 flatmap() (in module *lz.functional*), 4
 flatmapper() (in module *lz.iterating*), 6
 flatten() (in module *lz.iterating*), 6
 flip() (in module *lz.functional*), 4
 fold() (in module *lz.left*), 9
 fold() (in module *lz.right*), 12

folder() (in module *lz.left*), 9
 folder() (in module *lz.right*), 12

G

grab() (in module *lz.filtration*), 1
 grabber() (in module *lz.filtration*), 1
 group_by() (in module *lz.iterating*), 6
 grouper() (in module *lz.iterating*), 6

H

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

I

identity() (in module *lz.functional*), 4
 in_four() (in module *lz.iterating*), 7
 in_three() (in module *lz.iterating*), 7
 in_two() (in module *lz.iterating*), 7
 instance_of() (in module *lz.typology*), 12
 interleave() (in module *lz.iterating*), 7

K

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

L

last() (in module *lz.iterating*), 7
 lz.filtration
 module, 1
 lz.functional
 module, 3
 lz.iterating
 module, 5
 lz.left
 module, 8
 lz.logical
 module, 9
 lz.replication
 module, 10
 lz.reversal
 module, 10
 lz.right

- module, 11
- lz.transposition
 - module, 12
- lz.typology
 - module, 12

M

mapper() (*in module lz.iterating*), 7

module

- lz.filtration, 1
- lz.functional, 3
- lz.iterating, 5
- lz.left, 8
- lz.logical, 9
- lz.replication, 10
- lz.reversal, 10
- lz.right, 11
- lz.transposition, 12
- lz.typology, 12

N

negate() (*in module lz.logical*), 10

P

pack() (*in module lz.functional*), 4

pairwise() (*in module lz.iterating*), 7

Q

quadruplewise() (*in module lz.iterating*), 7

R

replicate() (*in module lz.replication*), 10

replicator() (*in module lz.replication*), 10

reverse() (*in module lz.reversal*), 10

reverse_bytes_stream() (*in module lz.reversal*), 11

reverse_file_object() (*in module lz.reversal*), 11

S

scavenge() (*in module lz.filtration*), 2

scavenger() (*in module lz.filtration*), 2

separate() (*in module lz.filtration*), 2

separator() (*in module lz.filtration*), 2

sift() (*in module lz.filtration*), 3

sifter() (*in module lz.filtration*), 3

slide() (*in module lz.iterating*), 7

slider() (*in module lz.iterating*), 7

subclass_of() (*in module lz.typology*), 13

T

to_constant() (*in module lz.functional*), 5

trail() (*in module lz.iterating*), 7

trailer() (*in module lz.iterating*), 8

transpose() (*in module lz.transposition*), 12

triplewise() (*in module lz.iterating*), 8