Definitions

toolz.itertoolz.remove(*predicate, seq*) [source]

Return those items of sequence for which predicate(item) is False

```
>>> def iseven(x):
...     return x % 2 == 0
>>> list(remove(iseven, [1, 2, 3, 4]))
[1, 3]
```

toolz.itertoolz.accumulate(*binop, seq, initial='__no__default__'*) [source]

Repeatedly apply binary function to a sequence, accumulating results

```
>>> from operator import add, mul
>>> list(accumulate(add, [1, 2, 3, 4, 5]))
[1, 3, 6, 10, 15]
>>> list(accumulate(mul, [1, 2, 3, 4, 5]))
[1, 2, 6, 24, 120]
```

Accumulate is similar to `reduce` and is good for making functions like cumulative sum:

```
>>> from functools import partial, reduce
>>> sum    = partial(reduce, add)
>>> cumsum = partial(accumulate, add)
```

Accumulate also takes an optional argument that will be used as the first value. This is similar to reduce.

```
>>> list(accumulate(add, [1, 2, 3], -1))
[-1, 0, 2, 5]
>>> list(accumulate(add, [], 1))
[1]
```

See Also:

itertools.accumulate : In standard itertools for Python 3.2+

toolz.itertoolz.groupby(*key, seq*) [source]

Group a collection by a key function

```
>>> names = ['Alice', 'Bob', 'Charlie', 'Dan', 'Edith', 'Frank']
>>> groupby(len, names)
{3: ['Bob', 'Dan'], 5: ['Alice', 'Edith', 'Frank'], 7: ['Charlie']}
```

```
>>> iseven = lambda x: x % 2 == 0
>>> groupby(iseven, [1, 2, 3, 4, 5, 6, 7, 8])
{False: [1, 3, 5, 7], True: [2, 4, 6, 8]}
```

Non-callable keys imply grouping on a member.

```
>>> groupby('gender', [{'name': 'Alice', 'gender': 'F'},
...                    {'name': 'Bob', 'gender': 'M'},
...                    {'name': 'Charlie', 'gender': 'M'}])
{'F': [{'gender': 'F', 'name': 'Alice'}],
 'M': [{'gender': 'M', 'name': 'Bob'},
       {'gender': 'M', 'name': 'Charlie'}]}
```

> **See Also:**
>
> countby

`toolz.itertoolz.merge_sorted(*seqs, **kwargs)` [source]

Merge and sort a collection of sorted collections

This works lazily and only keeps one value from each iterable in memory.

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

```
>>> ''.join(merge_sorted('abc', 'abc', 'abc'))
'aaabbbccc'
```

The “key” function used to sort the input may be passed as a keyword.

```
>>> list(merge_sorted([2, 3], [1, 3], key=lambda x: x // 3))
[2, 1, 3, 3]
```

toolz.itertoolz.interleave(*seqs, pass_exceptions=()*) [source]

Interleave a sequence of sequences

```
>>> list(interleave([[1, 2], [3, 4]]))
[1, 3, 2, 4]
```

```
>>> ''.join(interleave(('ABC', 'XY')))
'AXBYC'
```

Both the individual sequences and the sequence of sequences may be infinite

Returns a lazy iterator

toolz.itertoolz.unique(*seq, key=None*) [source]

Return only unique elements of a sequence

```
>>> tuple(unique((1, 2, 3)))
(1, 2, 3)
>>> tuple(unique((1, 2, 1, 3)))
(1, 2, 3)
```

Uniqueness can be defined by key keyword

```
>>> tuple(unique(['cat', 'mouse', 'dog', 'hen'], key=len))
('cat', 'mouse')
```

toolz.itertoolz.isiterable(*x*) [source]

Is x iterable?

```
>>> isiterable([1, 2, 3])
True
>>> isiterable('abc')
True
>>> isiterable(5)
False
```

toolz.itertoolz.isdistinct(*seq*) [source]

All values in sequence are distinct

```
>>> isdistinct([1, 2, 3])
True
>>> isdistinct([1, 2, 1])
False
```

```
>>> isdistinct("Hello")
False
>>> isdistinct("World")
True
```

toolz.itertoolz.take(*n*, *seq*) [source]

The first n elements of a sequence

```
>>> list(take(2, [10, 20, 30, 40, 50]))
[10, 20]
```

See Also:

drop tail

toolz.itertoolz.drop(*n*, *seq*) [source]

The sequence following the first n elements

```
>>> list(drop(2, [10, 20, 30, 40, 50]))
[30, 40, 50]
```

See Also:

take tail

toolz.itertoolz.take_nth(*n*, *seq*) [source]

Every nth item in seq

```
>>> list(take_nth(2, [10, 20, 30, 40, 50]))
[10, 30, 50]
```

toolz.itertoolz.first(*seq*) [source]

The first element in a sequence

```
>>> first('ABC')
'A'
```

toolz.itertoolz.second(*seq*) [source]

The second element in a sequence

```
>>> second('ABC')
'B'
```

toolz.itertoolz.nth(*n*, *seq*) [source]

The nth element in a sequence

```
>>> nth(1, 'ABC')
'B'
```

toolz.itertoolz.last(*seq*) [source]

The last element in a sequence

```
>>> last('ABC')
'C'
```

toolz.itertoolz.get(*ind*, *seq*, *default='__no__default__'*) [source]

Get element in a sequence or dict

Provides standard indexing

```
>>> get(1, 'ABC')       # Same as 'ABC'[1]
'B'
```

Pass a list to get multiple values

```
>>> get([1, 2], 'ABC')  # ('ABC'[1], 'ABC'[2])
('B', 'C')
```

Works on any value that supports indexing/getitem For example here we see that it works with dictionaries

```
>>> phonebook = {'Alice':  '555-1234',
...              'Bob':    '555-5678',
...              'Charlie':'555-9999'}
>>> get('Alice', phonebook)
'555-1234'
```

```
>>> get(['Alice', 'Bob'], phonebook)
('555-1234', '555-5678')
```

Provide a default for missing values

```
>>> get(['Alice', 'Dennis'], phonebook, None)
('555-1234', None)
```

See Also:

pluck

toolz.itertoolz.concat(*seqs*) [source]

Concatenate zero or more iterables, any of which may be infinite.

An infinite sequence will prevent the rest of the arguments from being included.

We use chain.from_iterable rather than `chain(*seqs)` so that seqs can be a generator.

```
>>> list(concat([[], [1], [2, 3]]))
[1, 2, 3]
```

See also:

itertools.chain.from_iterable equivalent

toolz.itertoolz.concatv(**seqs*) [source]

Variadic version of concat

```
>>> list(concatv([], ["a"], ["b", "c"]))
['a', 'b', 'c']
```

See also:

itertools.chain

toolz.itertoolz.mapcat(*func, seqs*) [source]

Apply func to each sequence in seqs, concatenating results.

```
>>> list(mapcat(lambda s: [c.upper() for c in s],
...             [["a", "b"], ["c", "d", "e"]]))
['A', 'B', 'C', 'D', 'E']
```

toolz.itertoolz.cons(*el, seq*) [source]

Add el to beginning of (possibly infinite) sequence seq.

```
>>> list(cons(1, [2, 3]))
[1, 2, 3]
```

toolz.itertoolz.interpose(*el, seq*) [source]

Introduce element between each pair of elements in seq

```
>>> list(interpose("a", [1, 2, 3]))
[1, 'a', 2, 'a', 3]
```

toolz.itertoolz.frequencies(*seq*) [source]

Find number of occurrences of each value in seq

```
>>> frequencies(['cat', 'cat', 'ox', 'pig', 'pig', 'cat'])
{'cat': 3, 'ox': 1, 'pig': 2}
```

See Also:

countby groupby

toolz.itertoolz.reduceby(*key, binop, seq, init='__no__default__'*) [source]

Perform a simultaneous groupby and reduction

The computation:

```
>>> result = reduceby(key, binop, seq, init)
```

is equivalent to the following:

```
>>> def reduction(group):
...     return reduce(binop, group, init)
```

```
>>> groups = groupby(key, seq)
>>> result = valmap(reduction, groups)
```

But the former does not build the intermediate groups, allowing it to operate in much less space. This makes it suitable for larger datasets that do not fit comfortably in memory

The `init` keyword argument is the default initialization of the reduction. This can be either a constant value like `0` or a callable like `lambda : 0` as might be used in `defaultdict`.

```
>>> from operator import add, mul
>>> iseven = lambda x: x % 2 == 0
```

```
>>> data = [1, 2, 3, 4, 5]
```

```
>>> reduceby(iseven, add, data)
{False: 9, True: 6}
```

```
>>> reduceby(iseven, mul, data)
{False: 15, True: 8}
```

```
>>> projects = [{'name': 'build roads', 'state': 'CA', 'cost': 1000000},
...             {'name': 'fight crime', 'state': 'IL', 'cost': 100000},
...             {'name': 'help farmers', 'state': 'IL', 'cost': 2000000},
...             {'name': 'help farmers', 'state': 'CA', 'cost': 200000}]
```

```
>>> reduceby('state',
...          lambda acc, x: acc + x['cost'],
...          projects, 0)
{'CA': 1200000, 'IL': 2100000}
```

```
>>> def set_add(s, i):
...     s.add(i)
...     return s
```

```
>>> reduceby(iseven, set_add, [1, 2, 3, 4, 1, 2, 3], set)
{True:  set([2, 4]),
 False: set([1, 3])}
```

toolz.itertoolz.iterate(*func, x*) [source]

Repeatedly apply a function func onto an original input

Yields x, then func(x), then func(func(x)), then func(func(func(x))), etc..

```
>>> def inc(x):  return x + 1
>>> counter = iterate(inc, 0)
>>> next(counter)
0
>>> next(counter)
1
>>> next(counter)
2
```

```
>>> double = lambda x: x * 2
>>> powers_of_two = iterate(double, 1)
>>> next(powers_of_two)
1
>>> next(powers_of_two)
2
>>> next(powers_of_two)
4
>>> next(powers_of_two)
8
```

toolz.itertoolz.sliding_window(*n, seq*) [source]

A sequence of overlapping subsequences

```
>>> list(sliding_window(2, [1, 2, 3, 4]))
[(1, 2), (2, 3), (3, 4)]
```

This function creates a sliding window suitable for transformations like sliding means / smoothing

```
>>> mean = lambda seq: float(sum(seq)) / len(seq)
>>> list(map(mean, sliding_window(2, [1, 2, 3, 4])))
[1.5, 2.5, 3.5]
```

toolz.itertoolz.partition(*n, seq, pad='__no__pad__'*) [source]

Partition sequence into tuples of length n

```
>>> list(partition(2, [1, 2, 3, 4]))
[(1, 2), (3, 4)]
```

If the length of `seq` is not evenly divisible by `n`, the final tuple is dropped if `pad` is not specified, or filled to length `n` by pad:

```
>>> list(partition(2, [1, 2, 3, 4, 5]))
[(1, 2), (3, 4)]
```

```
>>> list(partition(2, [1, 2, 3, 4, 5], pad=None))
[(1, 2), (3, 4), (5, None)]
```

See Also:

partition_all

toolz.itertoolz.partition_all(*n, seq*) [source]

Partition all elements of sequence into tuples of length at most n

The final tuple may be shorter to accommodate extra elements.

```
>>> list(partition_all(2, [1, 2, 3, 4]))
[(1, 2), (3, 4)]
```

```
>>> list(partition_all(2, [1, 2, 3, 4, 5]))
[(1, 2), (3, 4), (5,)]
```

See Also:

partition

toolz.itertoolz.count(*seq*) [source]

Count the number of items in seq

Like the builtin `len` but works on lazy sequencies.

Not to be confused with `itertools.count`

See also:

len

toolz.itertoolz.pluck(*ind, seqs, default='__no__default__'*) [source]

plucks an element or several elements from each item in a sequence.

`pluck` maps `itertoolz.get` over a sequence and returns one or more elements of each item in the sequence.

This is equivalent to running *map(curried.get(ind), seqs)*

`ind` can be either a single string/index or a sequence of strings/indices. `seqs` should be sequence containing sequences or dicts.

e.g.

```
>>> data = [{'id': 1, 'name': 'Cheese'}, {'id': 2, 'name': 'Pies'}]
>>> list(pluck('name', data))
['Cheese', 'Pies']
>>> list(pluck([0, 1], [[1, 2, 3], [4, 5, 7]]))
[(1, 2), (4, 5)]
```

See Also:

get map

toolz.itertoolz.join(*leftkey, leftseq, rightkey, rightseq, left_default='__no__default__', right_default='__no__default__'*) [source]

Join two sequences on common attributes

This is a semi-streaming operation. The LEFT sequence is fully evaluated and placed into memory. The RIGHT sequence is evaluated lazily and so can be arbitrarily large.

```
>>> friends = [('Alice', 'Edith'),
...            ('Alice', 'Zhao'),
...            ('Edith', 'Alice'),
...            ('Zhao', 'Alice'),
...            ('Zhao', 'Edith')]
```

```
>>> cities = [('Alice', 'NYC'),
...           ('Alice', 'Chicago'),
...           ('Dan', 'Syndey'),
...           ('Edith', 'Paris'),
...           ('Edith', 'Berlin'),
...           ('Zhao', 'Shanghai')]
```

```
>>> # Vacation opportunities
>>> # In what cities do people have friends?
>>> result = join(second, friends,
...               first, cities)
>>> for ((a, b), (c, d)) in sorted(unique(result)):
...     print((a, d))
('Alice', 'Berlin')
('Alice', 'Paris')
('Alice', 'Shanghai')
('Edith', 'Chicago')
('Edith', 'NYC')
('Zhao', 'Chicago')
('Zhao', 'NYC')
('Zhao', 'Berlin')
('Zhao', 'Paris')
```

Specify outer joins with keyword arguments `left_default` and/or `right_default`. Here is a full outer join in which unmatched elements are paired with None.

```
>>> identity = lambda x: x
>>> list(join(identity, [1, 2, 3],
...           identity, [2, 3, 4],
...           left_default=None, right_default=None))
[(2, 2), (3, 3), (None, 4), (1, None)]
```

Usually the key arguments are callables to be applied to the sequences. If the keys are not obviously callable then it is assumed that indexing was intended, e.g. the following is a legal change

```
>>> # result = join(second, friends, first, cities)
>>> result = join(1, friends, 0, cities)
```

toolz.itertoolz.tail(*n, seq*) [source]

The last n elements of a sequence

```
>>> tail(2, [10, 20, 30, 40, 50])
[40, 50]
```

See Also:

drop take

toolz.itertoolz.diff(**seqs, **kwargs*) [source]

Return those items that differ between sequences

```
>>> list(diff([1, 2, 3], [1, 2, 10, 100]))
[(3, 10)]
```

Shorter sequences may be padded with a `default` value:

```
>>> list(diff([1, 2, 3], [1, 2, 10, 100], default=None))
[(3, 10), (None, 100)]
```

A `key` function may also be applied to each item to use during comparisons:

```
>>> list(diff(['apples', 'bananas'], ['Apples', 'Oranges'], key=str.lower))
[('bananas', 'Oranges')]
```

toolz.itertoolz.topk(*k, seq, key=None*) [source]

Find the k largest elements of a sequence

Operates lazily in `n*log(k)` time

```
>>> topk(2, [1, 100, 10, 1000])
(1000, 100)
```

Use a key function to change sorted order

```
>>> topk(2, ['Alice', 'Bob', 'Charlie', 'Dan'], key=len)
('Charlie', 'Alice')
```

See also:

heapq.nlargest

toolz.itertoolz.peek(*seq*) [source]

Retrieve the next element of a sequence

Returns the first element and an iterable equivalent to the original sequence, still having the element retrieved.

```
>>> seq = [0, 1, 2, 3, 4]
>>> first, seq = peek(seq)
>>> first
0
>>> list(seq)
[0, 1, 2, 3, 4]
```

toolz.itertoolz.random_sample(*prob, seq, random_state=None*) [source]

Return elements from a sequence with probability of prob

Returns a lazy iterator of random items from seq.

`random_sample` considers each item independently and without replacement. See below how the first time it returned 13 items and the next time it returned 6 items.

```
>>> seq = list(range(100))
>>> list(random_sample(0.1, seq))
[6, 9, 19, 35, 45, 50, 58, 62, 68, 72, 78, 86, 95]
>>> list(random_sample(0.1, seq))
[6, 44, 54, 61, 69, 94]
```

Providing an integer seed for `random_state` will result in deterministic sampling. Given the same seed it will return the same sample every time.

```
>>> list(random_sample(0.1, seq, random_state=2016))
[7, 9, 19, 25, 30, 32, 34, 48, 59, 60, 81, 98]
>>> list(random_sample(0.1, seq, random_state=2016))
[7, 9, 19, 25, 30, 32, 34, 48, 59, 60, 81, 98]
```

`random_state` can also be any object with a method `random` that returns floats between 0.0 and 1.0 (exclusive).

```
>>> from random import Random
>>> randobj = Random(2016)
>>> list(random_sample(0.1, seq, random_state=randobj))
[7, 9, 19, 25, 30, 32, 34, 48, 59, 60, 81, 98]
```

toolz.recipes.countby(*key, seq*) [source]

Count elements of a collection by a key function

```
>>> countby(len, ['cat', 'mouse', 'dog'])
{3: 2, 5: 1}
```

```
>>> def iseven(x): return x % 2 == 0
>>> countby(iseven, [1, 2, 3])
{True: 1, False: 2}
```

See Also:

groupby

toolz.recipes.partitionby(*func, seq*) [source]

Partition a sequence according to a function

Partition *s* into a sequence of lists such that, when traversing *s*, every time the output of *func* changes a new list is started and that and subsequent items are collected into that list.

```
>>> is_space = lambda c: c == " "
>>> list(partitionby(is_space, "I have space"))
[('I',), (' ',), ('h', 'a', 'v', 'e'), (' ',), ('s', 'p', 'a', 'c', 'e')]
```

```
>>> is_large = lambda x: x > 10
>>> list(partitionby(is_large, [1, 2, 1, 99, 88, 33, 99, -1, 5]))
[(1, 2, 1), (99, 88, 33, 99), (-1, 5)]
```

See also:

partition groupby itertools.groupby

`toolz.functoolz.identity(x)` [source]

Identity function. Return x

```
>>> identity(3)
3
```

`toolz.functoolz.thread_first(val, *forms)` [source]

Thread value through a sequence of functions/forms

```
>>> def double(x): return 2*x
>>> def inc(x):    return x + 1
>>> thread_first(1, inc, double)
4
```

If the function expects more than one input you can specify those inputs in a tuple. The value is used as the first input.

```
>>> def add(x, y): return x + y
>>> def pow(x, y): return x**y
>>> thread_first(1, (add, 4), (pow, 2))  # pow(add(1, 4), 2)
25
```

So in general

thread_first(x, f, (g, y, z))

expands to

g(f(x), y, z)

See Also:

thread_last

`toolz.functoolz.thread_last(val, *forms)` [source]

Thread value through a sequence of functions/forms

```
>>> def double(x): return 2*x
>>> def inc(x):    return x + 1
>>> thread_last(1, inc, double)
4
```

If the function expects more than one input you can specify those inputs in a tuple. The value is used as the last input.

```
>>> def add(x, y): return x + y
>>> def pow(x, y): return x**y
>>> thread_last(1, (add, 4), (pow, 2))  # pow(2, add(4, 1))
32
```

So in general

thread_last(x, f, (g, y, z))

expands to

g(y, z, f(x))

```
>>> def iseven(x):
...     return x % 2 == 0
>>> list(thread_last([1, 2, 3], (map, inc), (filter, iseven)))
[2, 4]
```

See Also:

thread_first

`toolz.functoolz.memoize` [source]

Cache a function's result for speedy future evaluation

Considerations:

Trades memory for speed. Only use on pure functions.

```
>>> def add(x, y):  return x + y
>>> add = memoize(add)
```

Or use as a decorator

```
>>> @memoize
... def add(x, y):
...     return x + y
```

Use the `cache` keyword to provide a dict-like object as an initial cache

```
>>> @memoize(cache={(1, 2): 3})
... def add(x, y):
...     return x + y
```

Note that the above works as a decorator because `memoize` is curried.

It is also possible to provide a `key(args, kwargs)` function that calculates keys used for the cache, which receives an `args` tuple and `kwargs` dict as input, and must return a hashable value. However, the default key function should be sufficient most of the time.

```
>>> # Use key function that ignores extraneous keyword arguments
>>> @memoize(key=lambda args, kwargs: args)
... def add(x, y, verbose=False):
...     if verbose:
...         print('Calculating %s + %s' % (x, y))
...     return x + y
```

toolz.functoolz.compose(*funcs) [source]

Compose functions to operate in series.

Returns a function that applies other functions in sequence.

Functions are applied from right to left so that `compose(f, g, h)(x, y)` is the same as `f(g(h(x, y)))`.

If no arguments are provided, the identity function (f(x) = x) is returned.

```
>>> inc = lambda i: i + 1
>>> compose(str, inc)(3)
'4'
```

See Also:

pipe

toolz.functoolz.pipe(data, *funcs) [source]

Pipe a value through a sequence of functions

I.e. `pipe(data, f, g, h)` is equivalent to `h(g(f(data)))`

We think of the value as progressing through a pipe of several transformations, much like pipes in UNIX

`$ cat data | f | g | h`

```
>>> double = lambda i: 2 * i
>>> pipe(3, double, str)
'6'
```

See Also:

compose thread_first thread_last

toolz.functoolz.complement(*func*) [source]

Convert a predicate function to its logical complement.

In other words, return a function that, for inputs that normally yield True, yields False, and vice-versa.

```
>>> def iseven(n): return n % 2 == 0
>>> isodd = complement(iseven)
>>> iseven(2)
True
>>> isodd(2)
False
```

class toolz.functoolz.juxt(**funcs*) [source]

Creates a function that calls several functions with the same arguments

Takes several functions and returns a function that applies its arguments to each of those functions then returns a tuple of the results.

Name comes from juxtaposition: the fact of two things being seen or placed close together with contrasting effect.

```
>>> inc = lambda x: x + 1
>>> double = lambda x: x * 2
>>> juxt(inc, double)(10)
(11, 20)
>>> juxt([inc, double])(10)
(11, 20)
```

toolz.functoolz.do(*func, x*) [source]

Runs `func` on `x`, returns `x`

Because the results of `func` are not returned, only the side effects of `func` are relevant.

Logging functions can be made by composing `do` with a storage function like `list.append` or `file.write`

```
>>> from toolz import compose
>>> from toolz.curried import do
```

```
>>> log = []
>>> inc = lambda x: x + 1
>>> inc = compose(inc, do(log.append))
>>> inc(1)
2
>>> inc(11)
12
>>> log
[1, 11]
```

class toolz.functoolz.curry(**args, **kwargs*) [source]

Curry a callable function

Enables partial application of arguments through calling a function with an incomplete set of arguments.

```
>>> def mul(x, y):
...     return x * y
>>> mul = curry(mul)
```

```
>>> double = mul(2)
>>> double(10)
20
```

Also supports keyword arguments

```
>>> @curry                  # Can use curry as a decorator
... def f(x, y, a=10):
...     return a * (x + y)
```

```
>>> add = f(a=1)
>>> add(2, 3)
5
```

See Also:

toolz.curried - namespace of curried functions

http://toolz.readthedocs.org/en/latest/curry.html

toolz.functoolz.flip [source]

Call the function call with the arguments flipped

This function is curried.

```
>>> def div(a, b):
...     return a / b
...
>>> flip(div, 2, 1)
0.5
>>> div_by_two = flip(div, 2)
>>> div_by_two(4)
2.0
```

This is particularly useful for built in functions and functions defined in C extensions that accept positional only arguments. For example: isinstance, issubclass.

```
>>> data = [1, 'a', 'b', 2, 1.5, object(), 3]
>>> only_ints = list(filter(flip(isinstance, int), data))
>>> only_ints
[1, 2, 3]
```

class toolz.functoolz.excepts(*exc, func, handler=<function return_none>*) [source]

A wrapper around a function to catch exceptions and dispatch to a handler.

This is like a functional try/except block, in the same way that ifexprs are functional if/else blocks.

```
>>> excepting = excepts(
...     ValueError,
...     lambda a: [1, 2].index(a),
...     lambda _: -1,
... )
>>> excepting(1)
0
>>> excepting(3)
-1
```

Multiple exceptions and default except clause. >>> excepting = excepts((IndexError, KeyError), lambda a: a[0]) >>> excepting([]) >>> excepting([1]) 1 >>> excepting({}) >>> excepting({0: 1}) 1

toolz.dicttoolz.merge(*dicts, **kwargs) [source]

Merge a collection of dictionaries

```
>>> merge({1: 'one'}, {2: 'two'})
{1: 'one', 2: 'two'}
```

Later dictionaries have precedence

```
>>> merge({1: 2, 3: 4}, {3: 3, 4: 4})
{1: 2, 3: 3, 4: 4}
```

See Also:

merge_with

toolz.dicttoolz.merge_with(func, *dicts, **kwargs) [source]

Merge dictionaries and apply function to combined values

A key may occur in more than one dict, and all values mapped from the key will be passed to the function as a list, such as func([val1, val2, ...]).

```
>>> merge_with(sum, {1: 1, 2: 2}, {1: 10, 2: 20})
{1: 11, 2: 22}
```

```
>>> merge_with(first, {1: 1, 2: 2}, {2: 20, 3: 30})
{1: 1, 2: 2, 3: 30}
```

See Also:

merge

toolz.dicttoolz.valmap(*func, d, factory=<type 'dict'>*) [source]

Apply function to values of dictionary

```
>>> bills = {"Alice": [20, 15, 30], "Bob": [10, 35]}
>>> valmap(sum, bills)
{'Alice': 65, 'Bob': 45}
```

See Also:

keymap itemmap

toolz.dicttoolz.keymap(*func, d, factory=<type 'dict'>*) [source]

Apply function to keys of dictionary

```
>>> bills = {"Alice": [20, 15, 30], "Bob": [10, 35]}
>>> keymap(str.lower, bills)
{'alice': [20, 15, 30], 'bob': [10, 35]}
```

See Also:

valmap itemmap

toolz.dicttoolz.itemmap(*func, d, factory=<type 'dict'>*) [source]

Apply function to items of dictionary

```
>>> accountids = {"Alice": 10, "Bob": 20}
>>> itemmap(reversed, accountids)
{10: "Alice", 20: "Bob"}
```

See Also:

keymap valmap

toolz.dicttoolz.valfilter(*predicate, d, factory=<type 'dict'>*)

[source]

Filter items in dictionary by value

```
>>> iseven = lambda x: x % 2 == 0
>>> d = {1: 2, 2: 3, 3: 4, 4: 5}
>>> valfilter(iseven, d)
{1: 2, 3: 4}
```

See Also:

keyfilter itemfilter valmap

toolz.dicttoolz.keyfilter(*predicate, d, factory=<type 'dict'>*)

[source]

Filter items in dictionary by key

```
>>> iseven = lambda x: x % 2 == 0
>>> d = {1: 2, 2: 3, 3: 4, 4: 5}
>>> keyfilter(iseven, d)
{2: 3, 4: 5}
```

See Also:

valfilter itemfilter keymap

toolz.dicttoolz.itemfilter(*predicate, d, factory=<type 'dict'>*)

[source]

Filter items in dictionary by item

```
>>> def isvalid(item):
...     k, v = item
...     return k % 2 == 0 and v < 4
```

```
>>> d = {1: 2, 2: 3, 3: 4, 4: 5}
>>> itemfilter(isvalid, d)
{2: 3}
```

See Also:

keyfilter valfilter itemmap

toolz.dicttoolz.assoc(*d, key, value, factory=<type 'dict'>*) [source]

Return a new dict with new key value pair

New dict has d[key] set to value. Does not modify the initial dictionary.

```
>>> assoc({'x': 1}, 'x', 2)
{'x': 2}
>>> assoc({'x': 1}, 'y', 3)
{'x': 1, 'y': 3}
```

toolz.dicttoolz.dissoc(*d, *keys*) [source]

Return a new dict with the given key(s) removed.

New dict has d[key] deleted for each supplied key. Does not modify the initial dictionary.

```
>>> dissoc({'x': 1, 'y': 2}, 'y')
{'x': 1}
>>> dissoc({'x': 1, 'y': 2}, 'y', 'x')
{}
>>> dissoc({'x': 1}, 'y') # Ignores missing keys
{'x': 1}
```

toolz.dicttoolz.assoc_in(*d, keys, value, factory=<type 'dict'>*) [source]

Return a new dict with new, potentially nested, key value pair

```
>>> purchase = {'name': 'Alice',
...             'order': {'items': ['Apple', 'Orange'],
...                       'costs': [0.50, 1.25]},
...             'credit card': '5555-1234-1234-1234'}
>>> assoc_in(purchase, ['order', 'costs'], [0.25, 1.00])
{'credit card': '5555-1234-1234-1234',
 'name': 'Alice',
 'purchase': {'costs': [0.25, 1.00], 'items': ['Apple', 'Orange']}}
```

toolz.dicttoolz.update_in(*d, keys, func, default=None, factory=<type 'dict'>*) [source]

Update value in a (potentially) nested dictionary

inputs: d - dictionary on which to operate keys - list or tuple giving the location of the value to be changed in d func - function to operate on that value

If keys == [k0,..,kX] and d[k0]..[kX] == v, update_in returns a copy of the original dictionary with v replaced by func(v), but does not mutate the original dictionary.

If k0 is not a key in d, update_in creates nested dictionaries to the depth specified by the keys, with the innermost value set to func(default).

```
>>> inc = lambda x: x + 1
>>> update_in({'a': 0}, ['a'], inc)
{'a': 1}
```

```
>>> transaction = {'name': 'Alice',
...                'purchase': {'items': ['Apple', 'Orange'],
...                             'costs': [0.50, 1.25]},
...                'credit card': '5555-1234-1234-1234'}
>>> update_in(transaction, ['purchase', 'costs'], sum)
{'credit card': '5555-1234-1234-1234',
 'name': 'Alice',
 'purchase': {'costs': 1.75, 'items': ['Apple', 'Orange']}}
```

```
>>> # updating a value when k0 is not in d
>>> update_in({}, [1, 2, 3], str, default="bar")
{1: {2: {3: 'bar'}}}
>>> update_in({1: 'foo'}, [2, 3, 4], inc, 0)
{1: 'foo', 2: {3: {4: 1}}}
```

toolz.dicttoolz.get_in(*keys, coll, default=None, no_default=False*)

[source]

Returns coll[i0][i1]...[iX] where [i0, i1, ..., iX]==keys.

If coll[i0][i1]...[iX] cannot be found, returns `default`, unless `no_default` is specified, then it raises KeyError or IndexError.

`get_in` is a generalization of `operator.getitem` for nested data structures such as dictionaries and lists.

```
>>> transaction = {'name': 'Alice',
...                'purchase': {'items': ['Apple', 'Orange'],
...                             'costs': [0.50, 1.25]},
...                'credit card': '5555-1234-1234-1234'}
>>> get_in(['purchase', 'items', 0], transaction)
'Apple'
>>> get_in(['name'], transaction)
'Alice'
>>> get_in(['purchase', 'total'], transaction)
>>> get_in(['purchase', 'items', 'apple'], transaction)
>>> get_in(['purchase', 'items', 10], transaction)
>>> get_in(['purchase', 'total'], transaction, 0)
0
>>> get_in(['y'], {}, no_default=True)
Traceback (most recent call last):
    ...
KeyError: 'y'
```

See Also:

itertoolz.get operator.getitem

class toolz.sandbox.core.EqualityHashKey(*key, item*) [source]

Create a hash key that uses equality comparisons between items.

This may be used to create hash keys for otherwise unhashable types:

```
>>> from toolz import curry
>>> EqualityHashDefault = curry(EqualityHashKey, None)
>>> set(map(EqualityHashDefault, [[], (), [1], [1]]))
{=[]=, =()=, =[1]=}
```

Caution: adding N `EqualityHashKey` items to a hash container may require O(N**2) operations, not O(N) as for typical hashable types. Therefore, a suitable key function such as `tuple` or `frozenset` is usually preferred over using `EqualityHashKey` if possible.

The `key` argument to `EqualityHashKey` should be a function or index that returns a hashable object that effectively distinguishes unequal items. This helps avoid the poor scaling that occurs when using the default key. For example, the above example can be improved by using a key function that distinguishes items by length or type:

```
>>> EqualityHashLen = curry(EqualityHashKey, len)
>>> EqualityHashType = curry(EqualityHashKey, type)  # this works too
>>> set(map(EqualityHashLen, [[], (), [1], [1]]))
{=[]=, =()=, =[1]=}
```

`EqualityHashKey` is convenient to use when a suitable key function is complicated or unavailable. For example, the following returns all unique values based on equality:

```
>>> from toolz import unique
>>> vals = [[], [], (), [1], [1], [2], {}, {}, {}]
>>> list(unique(vals, key=EqualityHashDefault))
[[], (), [1], [2], {}]
```

Warning: don't change the equality value of an item already in a hash containter. Unhashable types are unhashable for a reason. For example:

```
>>> L1 = [1] ; L2 = [2]
>>> s = set(map(EqualityHashDefault, [L1, L2]))
>>> s
{=[1]=, =[2]=}
```

```
>>> L1[0] = 2  # Don't do this!  ``s`` now has duplicate items!
>>> s
{=[2]=, =[2]=}
```

Although this may appear problematic, immutable data types is a common idiom in functional programming, and``EqualityHashKey`` easily allows the same idiom to be used by convention rather than strict requirement.

> **See Also:**
>
> identity

`toolz.sandbox.core.unzip(seq)` [source]

Inverse of `zip`

```
>>> a, b = unzip([('a', 1), ('b', 2)])
>>> list(a)
['a', 'b']
>>> list(b)
[1, 2]
```

Unlike the naive implementation `def unzip(seq): zip(*seq)` this implementation can handle a finite sequence of infinite sequences.

Caveats:

- The implementation uses `tee`, and so can use a significant amount of auxiliary storage if the resulting iterators are consumed at different times.
- The top level sequence cannot be infinite.

`toolz.sandbox.parallel.fold`(*binop, seq, default='__no__default__', map=<type 'itertools.imap'>, chunksize=128, combine=None*) [source]

Reduce without guarantee of ordered reduction.

inputs:

> **`binop` - associative operator. The associative property allows us to**
>
> leverage a parallel map to perform reductions in parallel.

`seq` - a sequence to be aggregated `default` - an identity element like 0 for `add` or 1 for mul

> **`map` - an implementation of `map`. This may be parallel and**
>
> determines how work is distributed.

> **`chunksize` - Number of elements of `seq` that should be handled**
>
> within a single function call

> **`combine` - Binary operator to combine two intermediate results.**
>
> If `binop` is of type (total, item) -> total then `combine` is of type (total, total) -> total Defaults to `binop` for common case of operators like add

Fold chunks up the collection into blocks of size `chunksize` and then feeds each of these to calls to `reduce`. This work is distributed with a call to `map`, gathered back and then refolded to finish the computation. In this way `fold` specifies only how to chunk up data but leaves the distribution of this work to an externally provided `map` function. This function can be sequential or rely on multithreading, multiprocessing, or even distributed solutions.

If `map` intends to serialize functions it should be prepared to accept and serialize lambdas. Note that the standard `pickle` module fails here.

```
>>> # Provide a parallel map to accomplish a parallel sum
>>> from operator import add
>>> fold(add, [1, 2, 3, 4], chunksize=2, map=map)
10
```