Toolz Documentation

Release 0.8.2

Matthew Rocklin, John Jacobsen

Jul 04, 2017

Contents

Toolz provides a set of utility functions for iterators, functions, and dictionaries. These functions interoperate well and form the building blocks of common data analytic operations. They extend the standard libraries *itertools* and *functools* and borrow heavily from the standard libraries of contemporary functional languages.

Toolz provides a suite of functions which have the following functional virtues:

- **Composable:** They interoperate due to their use of core data structures.
- **Pure:** They don't change their inputs or rely on external state.
- **Lazy:** They don't run until absolutely necessary, allowing them to support large streaming data sets.

Toolz functions are *pragmatic*. They understand that most programmers have deadlines.

- **Low Tech:** They're just functions, no syntax or magic tricks to learn
- **Tuned:** They're profiled and optimized
- **Serializable:** They support common solutions for parallel computing

This gives developers the power to write *powerful* programs to solve *complex problems* with relatively *simple code*. This code can be *easy to understand* without sacrificing *performance*. Toolz enables this approach, commonly associated with functional programming, within a natural Pythonic style suitable for most developers.

BSD licensed source code is available at http://github.com/pytoolz/toolz/ .

CHAPTER 1

Contents

Heritage

While Python was originally intended as an imperative language [*Guido*], it contains all elements necessary to support a rich set of features from the functional paradigm. In particular its core data structures, lazy iterators, and functions as first class objects can be combined to implement a common standard library of functions shared among many functional languages.

This was first recognized and supported through the standard libraries *itertools* and *functools* which contain functions like `permutations`, `chain` and `partial` to complement the standard `map`, `filter`, `reduce` already found in the core language. While these libraries contain substantial functionality they do not achieve the same level of adoption found in similar projects in other languages. This may be because they are incomplete and lack a number of commonly related functions like `compose` and `groupby` which often complement these core operations.

A completion of this set of functions was first attempted in the projects *itertoolz* and *functoolz* (note the z). These libraries contained several functions that were absent in the standard *itertools*/*functools* libraries. The `itertoolz`/`functoolz` libraries were eventually merged into the monolithic `toolz` project described here.

Most contemporary functional languages (Haskell, Scala, Clojure, ...) contain some variation of the functions found in `toolz`. The `toolz` project generally adheres closely to the API found in the Clojure standard library (see *cheatsheet*) and where disagreements occur that API usually dominates. The `toolz` API is also strongly affected by the principles of the Python language itself, and often makes deviations in order to be more approachable to that community.

The development of a functional standard library within a popular imperative language is not unique. Similar projects have arisen in other imperative-by-design languages that contain the necessary elements to support a functional standard library. *Underscore.js* in JavaScript has attained notable popularity in the web community. `LINQ` in C# follows a similar philosophy but mimics declarative database languages rather than functional ones. *Enumerable* is is the closest project in Ruby. Other excellent projects also exist within the Python ecosystem, most notably *Fn.py* and *Funcy*.

Installation and Dependencies

Toolz is pure Python and so is easily installable by the standard dependency manager `pip`:

```
pip install toolz
```

Toolz endeavors to be a very light dependency. It accomplishes this in three ways:

1. Toolz is pure Python
2. Toolz relies only on the standard library
3. Toolz simultaneously supports Python versions 2.6, 2.7, 3.2, 3.3

Composability

Toolz functions interoperate because they consume and produce only a small set of common, core data structures. Each `toolz` function consumes just iterables, dictionaries, and functions and each `toolz` function produces just iterables, dictionaries, and functions. This standardized interface enables us to compose several general purpose functions to solve custom problems.

Standard interfaces enable us to use many tools together, even if those tools were not designed with each other in mind. We call this "using together" composition.

Standard Interface

This is best explained by two examples; the automobile industry and LEGOs.

Autos

Automobile pieces are not widely composable because they do not adhere to a standard interface. You can't connect a Porsche engine to the body of a Volkswagen Beetle but include the safety features of your favorite luxury car. As a result when something breaks you need to find a specialist who understands exactly your collection of components and, depending on the popularity of your model, replacement parts may be difficult to find. While the customization provides a number of efficiencies important for automobiles, it limits the ability of downstream tinkerers. This ability for future developers to tinker is paramount in good software design.

Lego

Contrast this with Lego toys. With Lego you *can* connect a rocket engine and skis to a rowboat. This is a perfectly natural thing to do because every piece adheres to a simple interface - those simple and regular 5mm circular bumps. This freedom to connect pieces at will lets children unleash their imagination in such varied ways (like going arctic shark hunting with a rocket-ski-boat).

The abstractions in programming make it far more like Lego than like building cars. This breaks down a little when we start to be constrained by performance or memory issues but this affects only a very small fraction of applications. Most of the time we have the freedom to operate in the Lego model if we choose to give up customization and embrace simple core standards.

Other Standard Interfaces

The Toolz project builds off of a standard interface – this choice is not unique. Other standard interfaces exist and provide immeasurable benefit to their application areas.

The NumPy array serves as a foundational object for numeric and scientific computing within Python. The ability of any project to consume and produce NumPy arrays is largely responsible for the broad success of the various SciPy projects. We see similar development today with the Pandas DataFrame.

The UNIX toolset relies on files and streams of text.

JSON emerged as the standard interface for communication over the web. The virtues of standardization become glaringly apparent when we contrast JSON with its predecessor, XML. XML was designed to be extensible/customizable, allowing each application to design its own interface. This resulted in a sea of difficult to understand custom data languages that failed to develop a common analytic and data processing infrastructure. In contrast JSON is very restrictive and allows only a fixed set of data structures, namely lists, dictionaries, numbers, strings. Fortunately this set is common to most modern languages and so JSON is extremely widely supported, perhaps falling second only to CSV.

Standard interfaces permeate physical reality as well. Examples range from supra-national currencies to drill bits and electrical circuitry. In all cases the interoperation that results becomes a defining and invaluable feature of each solution.

Function Purity

We call a function *pure* if it meets the following criteria

1. It does not depend on hidden state, or equivalently it only depends on its inputs.
2. Evaluation of the function does not cause side effects

In short the internal work of a pure function is isolated from the rest of the program.

Examples

This is made clear by two examples:

```
# A pure function
def min(x, y):
    if x < y:
        return x
    else:
        return y


# An impure function
exponent = 2

def powers(L):
    for i in range(len(L)):
        L[i] = L[i]**exponent
    return L
```

The function `min` is pure. It always produces the same result given the same inputs and it doesn't affect any external variable.

The function `powers` is impure for two reasons. First, it depends on a global variable, `exponent`, which can change*[0]. Second, it changes the input `L` which may have external state. Consider the following execution:

[0] A function depending on a global value can be pure if the value never changes, i.e. is immutable.

```
>>> data = [1, 2, 3]
>>> result = powers(data)

>>> print result
[1, 4, 9]
>>> print data
[1, 4, 9]
```

We see that `powers` affected the variable `data`. Users of our function might be surprised by this. Usually we expect our inputs to be unchanged.

Another problem occurs when we run this code in a different context:

```
>>> data = [1, 2, 3]
>>> result = powers(data)
>>> print result
[1, 8, 27]
```

When we give `powers` the same inputs we receive different outputs; how could this be? Someone must have changed the value of `exponent` to be `3`, producing cubes rather than squares. At first this flexibility may seem like a feature and indeed in many cases it may be. The cost for this flexibility is that we need to keep track of the `exponent` variable separately whenever we use `powers`. As we use more functions these extra variables become a burden.

State

Impure functions are often more efficient but also require that the programmer "keep track" of the state of several variables. Keeping track of this state becomes increasingly difficult as programs grow in size. By eschewing state programmers are able to conceptually scale out to solve much larger problems. The loss of performance is often negligible compared to the freedom to trust that your functions work as expected on your inputs.

Maintaining state provides efficiency at the cost of surprises. Pure functions produce no surprises and so lighten the mental load of the programmer.

Testing

As an added bonus, testing pure functions is substantially simpler than testing impure ones. A programmer who has tried to test functions that include randomness will know this first-hand.

Laziness

Lazy iterators evaluate only when necessary. They allow us to semantically manipulate large amounts of data while keeping very little of it actually in memory. They act like lists but don't take up space.

Example - A Tale of Two Cities

We open a file containing the text of the classic text "A Tale of Two Cities" by Charles Dickens[*1*].

```
>>> book = open('tale-of-two-cities.txt')
```

Much like a secondary school student, Python owns and opens the book without reading a single line of the text. The object `book` is a lazy iterator! Python will give us a line of the text only when we explicitly ask it to do so

```
>>> next(book)
"It was the best of times,"

>>> next(book)
"it was the worst of times,"
```

and so on. Each time we call `next` on `book` we burn through another line of the text and the `book` iterator marches slowly onwards through the text.

Computation

We can lazily operate on lazy iterators without doing any actual computation. For example lets read the book in upper case

```
>>> from toolz import map  # toolz' map is lazy by default

>>> loud_book = map(str.upper, book)

>>> next(loud_book)
"IT WAS THE AGE OF WISDOM,"
>>> next(loud_book)
"IT WAS THE AGE OF FOOLISHNESS,"
```

It is as if we applied the function `str.upper` onto every line of the book; yet the first line completes instantaneously. Instead Python does the uppercasing work only when it becomes necessary, i.e. when you call `next` to ask for another line.

Reductions

You can operate on lazy iterators just as you would with lists, tuples, or sets. You can use them in for loops as in

```
for line in loud_book:
    ...
```

You can instantiate them all into memory by calling them with the constructors `list`, or `tuple`.

```
loud_book = list(loud_book)
```

Of course if they are very large then this might be unwise. Often we use laziness to avoid loading large datasets into memory at once. Many computations on large datasets don't require access to all of the data at a single time. In particular *reductions* (like sum) often take large amounts of sequential data (like [1, 2, 3, 4]) and produce much more manageable results (like 10) and can do so just by viewing the data a little bit at a time. For example we can count all of the letters in the Tale of Two Cities trivially using functions from `toolz`

```
>>> from toolz import concat, frequencies
>>> letters = frequencies(concat(loud_book))
{ 'A': 48036,
  'B': 8402,
  'C': 13812,
  'D': 28000,
  'E': 74624,
  ...
```

In this case `frequencies` is a sort of reduction. At no time were more than a few hundred bytes of Tale of Two Cities necessarily in memory. We could just have easily done this computation on the entire Gutenberg collection

or on Wikipedia. In this case we are limited by the size and speed of our hard drive and not by the capacity of our memory.

Control Flow

Programming is hard when we think simultaneously about several concepts. Good programming breaks down big problems into small problems and builds up small solutions into big solutions. By this practice the need for simultaneous thought is restricted to only a few elements at a time.

All modern languages provide mechanisms to build data into data structures and to build functions out of other functions. The third element of programming, besides data and functions, is control flow. Building complex control flow out of simple control flow presents deeper challenges.

What?

Each element in a computer program is either

- A variable or value literal like `x`, `total`, or `5`
- A function or computation like the + in `x + 1`, the function `fib` in `fib(3)`, the method `split` in `line.split(',')`, or the = in `x = 0`
- Control flow like `if`, `for`, or `return`

Here is a piece of code; see if you can label each term as either variable/value, function/computation, or control flow

```
def fib(n):
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return b
```

Programming is hard when we have to juggle many code elements of each type at the same time. Good programming is about managing these three elements so that the developer is only required to think about a handful of them at a time. For example we might collect many integer variables into a list of integers or build a big function out of smaller ones. While we have natural ways to manage data and functions, control flow presents more of a challenge.

We organize our data into **data structures** like lists, dictionaries, or objects in order to group related data together – this allows us to manipulate large collections of related data as if we were only manipulating a single entity.

We **build large functions out of smaller ones**; enabling us to break up a complex task like doing laundry into a sequence of simpler tasks.

```
def do_laundry(clothes):
    wet_clothes = wash(clothes, coins)
    dry_clothes = dry(wet_clothes, coins)
    return fold(dry_clothes)
```

Control flow is more challenging; how do we break down complex control flow into simpler pieces that fit in our brain? How do we encapsulate commonly recurring patterns?

Lets motivate this with an example of a common control structure, applying a function to each element in a list. Imagine we want to download the HTML source for a number of webpages.

```
from urllib import urlopen

urls = ['http://www.google.com', 'http://www.wikipedia.com', 'http://www.apple.com']
```

```
html_texts = []
for item in urls:
    html_texts.append(urlopen(item))
return html_texts
```

Or maybe we want to compute the Fibonacci numbers on a particular set of integers

```
integers = [1, 2, 3, 4, 5]
fib_integers = []
for item in integers:
    fib_integers.append(fib(item))
return fib_integers
```

These two unrelated applications share an identical control flow pattern. They apply a function (`urlopen` or `fib`) onto each element of an input list (`urls`, or `integers`), appending the result onto an output list. Because this control flow pattern is so common we give it a name, `map`, and say that we map a function (like `urlopen`) onto a list (like `urls`).

Because Python can treat functions like variables we can encode this control pattern into a higher-order-function as follows:

```
def map(function, sequence):
    output = []
    for item in sequence:
        output.append(function(item))
    return output
```

This allows us to simplify our code above to the following, pithy solutions

```
html_texts = map(urlopen, urls)
fib_integers = map(fib, integers)
```

Experienced Python programmers know that this control pattern is so popular that it has been elevated to the status of **syntax** with the popular list comprehension

```
html_texts = [urlopen(url) for url in urls]
```

Why?

So maybe you already knew about `map` and don't use it or maybe you just prefer list comprehensions. Why should you keep reading?

Managing Complexity

The higher order function `map` gives us a name to call a particular control pattern. Regardless of whether or not you use a for loop, a list comprehension, or `map` itself, it is useful to recognize the operation and to give it a name. Naming control patterns lets us tackle complex problems a larger scale without burdening our mind with rote details. It is just as important as bundling data into data structures or building complex functions out of simple ones.

Naming control flow patterns enables programmers to manipulate increasingly complex operations.

Other Patterns

The function `map` has friends. Advanced programmers may know about `map`'s siblings, `filter` and `reduce`. The `filter` control pattern is also handled by list comprehension syntax and `reduce` is often replaced by straight for loops, so if you don't want to use them there is no immediately practical reason why you would care.

Most programmers however don't know about the many cousins of `map`/`filter`/`reduce`. Consider for example the unsung heroine, `groupby`. A brief example grouping names by their length follows:

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

Groupby collects each element of a list into sublists determined by the value of a function. Lets see `groupby` in action again, grouping numbers by evenness.

```
>>> def iseven(n):
...     return n % 2 == 0

>>> groupby(iseven, [1, 2, 3, 4, 5, 6, 7])
{True: [2, 4, 6], False: [1, 3, 5, 7]}
```

If we were to write this second operation out by hand it might look something like the following:

```
evens = []
odds = []
for item in numbers:
    if iseven(item):
        evens.append(item)
    else:
        odds.append(item)
```

Most programmers have written code exactly like this over and over again, just like they may have repeated the `map` control pattern. When we identify code as a `groupby` operation we mentally collapse the detailed manipulation into a single concept.

The Toolz library contains dozens of patterns like `map` and `groupby`. Learning a core set (maybe a dozen) covers the vast majority of common programming tasks often done by hand.

A rich vocabulary of core control functions conveys the following benefits:

- You identify new patterns
- You make fewer errors in rote coding
- You can depend on well tested and benchmarked implementations

But this does not come for free. As in spoken language the use of a rich vocabulary can alienate new practitioners. Most functional languages have fallen into this trap and are seen as unapproachable and smug. Python maintains a low-brow reputation and benefits from it. Just as with spoken language the value of using just-the-right-word must be moderated with the comprehension of the intended audience.

Curry

Traditionally partial evaluation of functions is handled with the `partial` higher order function from `functools`. Currying provides syntactic sugar.

```
>>> double = partial(mul, 2)     # Partial evaluation
>>> doubled = double(2)          # Currying
```

This syntactic sugar is valuable when developers chain several higher order functions together.

Partial Evaluation

Often when composing smaller functions to form big ones we need partial evaluation. We do this in the word counting example:

```
>>> def stem(word):
...     """ Stem word to primitive form """
...     return word.lower().rstrip(",.!:;'-\"").lstrip("'\"")

>>> wordcount = compose(frequencies, partial(map, stem), str.split)
```

Here we want to map the `stem` function onto each of the words produced by `str.split`. We want a `stem_many` function that takes a list of words, stems them, and returns a list back. In full form this would look like the following:

```
>>> def stem_many(words):
...     return map(stem, words)
```

The `partial` function lets us create this function more naturally.

```
>>> stem_many = partial(map, stem)
```

In general

```
>>> def f(x, y, z):
...     # Do stuff with x, y, and z

>>> # partially evaluate f with known values a and b
>>> def g(z):
...     return f(a, b, z)

>>> # partially evaluate f with known values a and b
>>> g = partial(f, a, b)
```

Curry

In this context currying is just syntactic sugar for partial evaluation. A curried function partially evaluates if it does not receive enough arguments to compute a result.

```
>>> from toolz import curry

>>> @curry                 # We can use curry as a decorator
... def mul(x, y):
...     return x * y

>>> double = mul(2)        # mul didn't receive enough arguments to evaluate
...                        # so it holds onto the 2 and waits, returning a
...                        # partially evaluated function, double

>>> double(5)
10
```

So if `map` was curried...

```
>>> map = curry(map)
```

Then we could replace the `partial` with a function evaluation

```
>>> # wordcount = compose(frequencies, partial(map, stem), str.split)
>>> wordcount = compose(frequencies, map(stem), str.split)
```

In this particular example it's probably simpler to stick with `partial`. Once `partial` starts occurring several times in your code it may be time to switch to the `curried` namespace.

The Curried Namespace

All functions present in the `toolz` namespace are curried in the `toolz.curried` namespace.

So you can exchange an import line like the following

```
>>> from toolz import *
```

For the following

```
>>> from toolz.curried import *
```

And all of your favorite `toolz` functions will curry automatically. We've also included curried versions of the standard Python higher order functions like `map`, `filter`, `reduce` so you'll get them too (whether you like it or not.)

Streaming Analytics

The toolz functions can be composed to analyze large streaming datasets. Toolz supports common analytics patterns like the selection, grouping, reduction, and joining of data through pure composable functions. These functions often have analogs to familiar operations in other data analytics platforms like SQL or Pandas.

Throughout this document we'll use this simple dataset of accounts

```
>>> accounts = [(1, 'Alice', 100, 'F'),  # id, name, balance, gender
...             (2, 'Bob', 200, 'M'),
...             (3, 'Charlie', 150, 'M'),
...             (4, 'Dennis', 50, 'M'),
...             (5, 'Edith', 300, 'F')]
```

Selecting with `map` and `filter`

Simple projection and linear selection from a sequence is achieved through the standard functions `map` and `filter`.

```
SELECT name, balance
FROM accounts
WHERE balance > 150;
```

These functions correspond to the SQL commands `SELECT` and `WHERE`.

```
>>> from toolz.curried import pipe, map, filter, get
>>> pipe(accounts, filter(lambda acc: acc[2] > 150),
...               map(get([1, 2])),
...               list)
```

note: this uses the curried_ versions of ‘‘map‘‘ and ‘‘filter‘‘.

Of course, these operations are also well supported with standard list/generator comprehension syntax. This syntax is more often used and generally considered to be more Pythonic.

```
>>> [(name, balance) for (id, name, balance, gender) in accounts
...                  if balance > 150]
```

Split-apply-combine with `groupby` and `reduceby`

We separate split-apply-combine operations into the following two concepts

1. Split the dataset into groups by some property
2. Reduce each of the groups with some synopsis function

Toolz supports this common workflow with

1. a simple in-memory solution
2. a more sophisticated streaming solution.

In Memory Split-Apply-Combine

The in-memory solution depends on the functions groupby to split, and valmap to apply/combine.

```
SELECT gender, SUM(balance)
FROM accounts
GROUP BY gender;
```

We first show these two functions piece by piece to show the intermediate groups.

```
>>> from toolz import groupby, valmap, compose
>>> from toolz.curried import get, pluck

>>> groupby(get(3), accounts)
{'F': [(1, 'Alice', 100, 'F'), (5, 'Edith', 300, 'F')],
 'M': [(2, 'Bob', 200, 'M'), (3, 'Charlie', 150, 'M'), (4, 'Dennis', 50, 'M')]}

>>> valmap(compose(sum, pluck(2)),
...        _)
{'F': 400, 'M': 400}
```

Then we chain them together into a single computation

```
>>> pipe(accounts, groupby(get(3)),
...               valmap(compose(sum, pluck(2))))
{'F': 400, 'M': 400}
```

Streaming Split-Apply-Combine

The `groupby` function collects the entire dataset in memory into a dictionary. While convenient, the `groupby` operation is *not streaming* and so this approach is limited to datasets that can fit comfortably into memory.

Toolz achieves streaming split-apply-combine with reduceby, a function that performs a simultaneous reduction on each group as the elements stream in. To understand this section you should first be familiar with the builtin function `reduce`.

The `reduceby` operation takes a key function, like `get(3)` or `lambda x: x[3]`, and a binary operator like `add` or `lesser = lambda acc, x: acc if acc < x else x`. It successively applies the key function to each item in succession, accumulating running totals for each key by combining each new value with the previous using the binary operator. It can't accept full reduction operations like `sum` or `min` as these require access to the entire group at once. Here is a simple example:

```
>>> from toolz import reduceby

>>> def iseven(n):
...     return n % 2 == 0

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

>>> reduceby(iseven, add, [1, 2, 3, 4])
{True: 6, False: 4}
```

The even numbers are added together `(2 + 4 = 6)` into group `True`, and the odd numbers are added together `(1 + 3 = 4)` into group `False`.

Note that we have to replace the reduction `sum` with the binary operator `add`. The incremental nature of `add` allows us to do the summation work as new data comes in. The use of binary operators like `add` over full reductions like `sum` enables computation on very large streaming datasets.

The challenge to using `reduceby` often lies in the construction of a suitable binary operator. Here is the solution for our accounts example that adds up the balances for each group:

```
>>> binop = lambda total, account: total + account[2]

>>> reduceby(get(3), binop, accounts, 0)
{'F': 400, 'M': 400}
```

This construction supports datasets that are much larger than available memory. Only the output must be able to fit comfortably in memory and this is rarely an issue, even for very large split-apply-combine computations.

Semi-Streaming `join`

We register multiple datasets together with join. Consider a second dataset storing addresses by ID

```
>>> addresses = [(1, '123 Main Street'),  # id, address
...              (2, '5 Adams Way'),
...              (5, '34 Rue St Michel')]
```

We can join this dataset against our accounts dataset by specifying attributes which register different elements with each other; in this case they share a common first column, id.

```
SELECT accounts.name, addresses.address
FROM accounts, addresses
WHERE accounts.id = addresses.id;
```

```
>>> from toolz import join, first

>>> result = join(first, accounts,
...               first, addresses)

>>> for ((id, name, bal, gender), (id, address)) in result:
...     print((name, address))
('Alice', '123 Main Street')
('Bob', '5 Adams Way')
('Edith', '34 Rue St Michel')
```

Join takes four main arguments, a left and right key function and a left and right sequence. It returns a sequence of pairs of matching items. In our case the return value of `join` is a sequence of pairs of tuples such that the first element of each tuple (the ID) is the same. In the example above we unpack this pair of tuples to get the fields that we want (`name` and `address`) from the result.

Join on arbitrary functions / data

Those familiar with SQL are accustomed to this kind of join on columns. However a functional join is more general than this; it doesn't need to operate on tuples, and key functions do not need to get particular columns. In the example below we match numbers from two collections so that exactly one is even and one is odd.

```
>>> def iseven(x):
...     return x % 2 == 0
>>> def isodd(x):
...     return x % 2 == 1

>>> list(join(iseven, [1, 2, 3, 4],
...           isodd, [7, 8, 9]))
[(2, 7), (4, 7), (1, 8), (3, 8), (2, 9), (4, 9)]
```

Semi-Streaming Join

The Toolz Join operation fully evaluates the *left* sequence and streams the *right* sequence through memory. Thus, if streaming support is desired the larger of the two sequences should always occupy the right side of the join.

Algorithmic Details

The semi-streaming join operation in `toolz` is asymptotically optimal. Computationally it is linear in the size of the input + output. In terms of storage the left sequence must fit in memory but the right sequence is free to stream.

The results are not normalized, as in SQL, in that they permit repeated values. If normalization is desired, consider composing with the function `unique` (note that `unique` is not fully streaming.)

More Complex Example

The accounts example above connects two one-to-one relationships, `accounts` and `addresses`; there was exactly one name per ID and one address per ID. This need not be the case. The join abstraction is sufficiently flexible to join

one-to-many or even many-to-many relationships. The following example finds city/person pairs where that person has a friend who has a residence in that city. This is an example of joining two many-to-many relationships, because a person may have many friends and because a friend may have many residences.

```
>>> 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 ((name, friend), (friend, city)) in sorted(unique(result)):
...     print((name, city))
('Alice', 'Berlin')
('Alice', 'Paris')
('Alice', 'Shanghai')
('Edith', 'Chicago')
('Edith', 'NYC')
('Zhao', 'Chicago')
('Zhao', 'NYC')
('Zhao', 'Berlin')
('Zhao', 'Paris')
```

Join is computationally powerful:

- It is expressive enough to cover a wide set of analytics operations
- It runs in linear time relative to the size of the input and output
- Only the left sequence must fit in memory

Disclaimer

Toolz is a general purpose functional standard library, not a library specifically for data analytics. While there are obvious benefits (streaming, composition, ...) users interested in data analytics might be better served by using projects specific to data analytics like Pandas or SQLAlchemy.

Parallelism

PyToolz tries to support other parallel processing libraries. It does this by ensuring easy serialization of `toolz` functions and providing architecture-agnostic parallel algorithms.

In practice `toolz` is developed against `multiprocessing` and `ipyparallel`.

Serialization

Multiprocessing or distributed computing requires the transmission of functions between different processes or computers. This is done through serializing the function into text, sending that text over a wire, and deserializing the text back into a function. To the extent possible PyToolz functions are compatible with the standard serialization library `pickle`.

The `pickle` library often fails for complex functions including lambdas, closures, and class methods. When this occurs we recommend the alternative serialization library `dill`.

Example with parallel map

Most parallel processing tasks may be significantly accelerated using only a parallel map operation. A number of high quality parallel map operations exist in other libraries, notably `multiprocessing`, `ipyparallel`, and `threading` (if your operation is not processor bound).

In the example below we extend our wordcounting solution with a parallel map. We show how one can progress in development from sequential, to multiprocessing, to distributed computation all with the same domain code.

```
from toolz.curried import map
from toolz import frequencies, compose, concat, merge_with

def stem(word):
    """ Stem word to primitive form

    >>> stem("Hello!")
    'hello'
    """
    return
    word.lower().rstrip(",.!)-*_?:;$'-\"").lstrip("-*'\"(_$'")


wordcount = compose(frequencies, map(stem), concat, map(str.split), open)

if __name__ == '__main__':
    # Filenames for thousands of books from which we'd like to count words
    filenames = ['Book_%d.txt'%i for i in range(10000)]

    # Start with sequential map for development
    # pmap = map

    # Advance to Multiprocessing map for heavy computation on single machine
    # from multiprocessing import Pool
    # p = Pool(8)
    # pmap = p.map

    # Finish with distributed parallel map for big data
    from ipyparallel import Client
    p = Client()[:]
    pmap = p.map_sync

    total = merge_with(sum, pmap(wordcount, filenames))
```

This smooth transition is possible because

1. The `map` abstraction is a simple function call and so can be replaced. This transformation would be difficult if we had written our code with a for loop or list comprehension

2. The operation `wordcount` is separate from the parallel solution.
3. The task is embarrassingly parallel, needing only a very simple parallel strategy. Fortunately this is the common case.

Parallel Algorithms

PyToolz does not implement parallel processing systems. It does however provide parallel algorithms that can extend existing parallel systems. Our general solution is to build algorithms that operate around a user-supplied parallel map function.

In particular we provide a parallel `fold` in `toolz.sandbox.parallel.fold`. This fold can work equally well with `multiprocessing.Pool.map` `threading.Pool.map` or `ipyparallel`'s `map_async`.

API

This page contains a comprehensive list of all functions within `toolz`. Docstrings should provide sufficient understanding for any individual function.

Itertoolz

accumulate(binop, seq[, initial])	Repeatedly apply binary function to a sequence, accumulating results
concat(seqs)	Concatenate zero or more iterables, any of which may be infinite.
concatv(*seqs)	Variadic version of concat
cons(el, seq)	Add el to beginning of (possibly infinite) sequence seq.
count(seq)	Count the number of items in seq
diff(*seqs, **kwargs)	Return those items that differ between sequences
drop(n, seq)	The sequence following the first n elements
first(seq)	The first element in a sequence
frequencies(seq)	Find number of occurrences of each value in seq
get(ind, seq[, default])	Get element in a sequence or dict
groupby(key, seq)	Group a collection by a key function
interleave(seqs)	Interleave a sequence of sequences
interpose(el, seq)	Introduce element between each pair of elements in seq
isdistinct(seq)	All values in sequence are distinct
isiterable(x)	Is x iterable?
iterate(func, x)	Repeatedly apply a function func onto an original input
join(leftkey, leftseq, rightkey, rightseq[, ...])	Join two sequences on common attributes
last(seq)	The last element in a sequence
mapcat(func, seqs)	Apply func to each sequence in seqs, concatenating results.
merge_sorted(*seqs, **kwargs)	Merge and sort a collection of sorted collections
nth(n, seq)	The nth element in a sequence
partition(n, seq[, pad])	Partition sequence into tuples of length n
partition_all(n, seq)	Partition all elements of sequence into tuples of length at most n
peek(seq)	Retrieve the next element of a sequence

Continued on next page

Table 1.1 – continued from previous page

pluck(ind, seqs[, default])	plucks an element or several elements from each item in a sequence.
random_sample(prob, seq[, random_state])	Return elements from a sequence with probability of prob
reduceby(key, binop, seq[, init])	Perform a simultaneous groupby and reduction
remove(predicate, seq)	Return those items of sequence for which predicate(item) is False
second(seq)	The second element in a sequence
sliding_window(n, seq)	A sequence of overlapping subsequences
tail(n, seq)	The last n elements of a sequence
take(n, seq)	The first n elements of a sequence
take_nth(n, seq)	Every nth item in seq
topk(k, seq[, key])	Find the k largest elements of a sequence
unique(seq[, key])	Return only unique elements of a sequence

countby(key, seq)	Count elements of a collection by a key function
partitionby(func, seq)	Partition a sequence according to a function

Functoolz

complement(func)	Convert a predicate function to its logical complement.
compose(*funcs)	Compose functions to operate in series.
curry(*args, **kwargs)	Curry a callable function
do(func, x)	Runs `func` on `x`, returns `x`
excepts(exc, func[, handler])	A wrapper around a function to catch exceptions and dispatch to a handler.
flip	Call the function call with the arguments flipped
identity(x)	Identity function.
juxt(*funcs)	Creates a function that calls several functions with the same arguments
memoize	Cache a function's result for speedy future evaluation
pipe(data, *funcs)	Pipe a value through a sequence of functions
thread_first(val, *forms)	Thread value through a sequence of functions/forms
thread_last(val, *forms)	Thread value through a sequence of functions/forms

Dicttoolz

assoc(d, key, value[, factory])	Return a new dict with new key value pair
dissoc(d, *keys)	Return a new dict with the given key(s) removed.
assoc_in(d, keys, value[, factory])	Return a new dict with new, potentially nested, key value pair
get_in(keys, coll[, default, no_default])	Returns coll[i0][i1]...[iX] where [i0, i1, ..., iX]==keys.
keyfilter(predicate, d[, factory])	Filter items in dictionary by key
keymap(func, d[, factory])	Apply function to keys of dictionary
itemfilter(predicate, d[, factory])	Filter items in dictionary by item
itemmap(func, d[, factory])	Apply function to items of dictionary
merge(*dicts, **kwargs)	Merge a collection of dictionaries
merge_with(func, *dicts, **kwargs)	Merge dictionaries and apply function to combined values

Continued on next page

Table 1.4 – continued from previous page

update_in(d, keys, func[, default, factory])	Update value in a (potentially) nested dictionary
valfilter(predicate, d[, factory])	Filter items in dictionary by value
valmap(func, d[, factory])	Apply function to values of dictionary

Sandbox

parallel.fold(binop, seq[, default, map, ...])	Reduce without guarantee of ordered reduction.
core.EqualityHashKey(key, item)	Create a hash key that uses equality comparisons between items.
core.unzip(seq)	Inverse of `zip`

Definitions

`toolz.itertoolz.`**`remove`**(*predicate*, *seq*)

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__'*)

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*)

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*)

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*)

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*)

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*)
Is x iterable?

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

toolz.itertoolz.**isdistinct**(*seq*)
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*)
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*)
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*)
Every nth item in seq

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

toolz.itertoolz.**first**(*seq*)
The first element in a sequence

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

toolz.itertoolz.**second**(*seq*)
The second element in a sequence

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

toolz.itertoolz.**nth**(*n*, *seq*)
The nth element in a sequence

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

toolz.itertoolz.**last**(*seq*)
The last element in a sequence

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

toolz.itertoolz.**get**(*ind*, *seq*, *default='__no__default__'*)
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*)
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*)
Variadic version of concat

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

See also: itertools.chain

toolz.itertoolz.**mapcat**(*func*, *seqs*)
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*)
Add el to beginning of (possibly infinite) sequence seq.

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

toolz.itertoolz.**interpose**(*el*, *seq*)
Introduce element between each pair of elements in seq

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

toolz.itertoolz.**frequencies**(*seq*)
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__'*)
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*)

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*)

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__'*)

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*)

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*)

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__'*)

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 list 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__'*)

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*)
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*)
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*)
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*)
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*)
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*)

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*)

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*)
Identity function. Return x

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

toolz.functoolz.**thread_first**(*val*, **forms*)
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*)
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**
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*)

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*)

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*)
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*)
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*)
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*)
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 https://toolz.readthedocs.io/en/latest/curry.html

toolz.functoolz.**flip**

Call the function call with the arguments flipped

This function is curried.

```
>>> def div(a, b):
...     return a // b
...
>>> flip(div, 2, 6)
3
>>> div_by_two = flip(div, 2)
>>> div_by_two(4)
2
```

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>*)

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*)
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*)
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'>*)
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'>*)
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'>*)
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'>*)
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'>*)
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'>*)
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'>*)
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*)
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'>*)
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',
 'order': {'costs': [0.25, 1.00], 'items': ['Apple', 'Orange']}}
```

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

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*)

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*)

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
```

Tips and Tricks

Toolz functions can be combined to make functions that, while common, aren't a part of toolz's standard library. This section presents a few of these recipes.

- **pick** (*whitelist*, *dictionary*)

  Return a subset of the provided dictionary with keys contained in the whitelist.

```
from toolz import keyfilter

def pick(whitelist, d):
    return keyfilter(lambda k: k in whitelist, d)
```

  Example:

```
>>> alphabet = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
>>> pick(['a', 'b'], alphabet)
{'a': 1, 'b': 2}
```

- **omit** (*blacklist*, *dictionary*)

  Return a subset of the provided dictionary with keys *not* contained in the blacklist.

```
from toolz import keyfilter

def omit(blacklist, d):
    return keyfilter(lambda k: k not in blacklist, d)
```

  Example:

```
>>> alphabet = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
>>> omit(['a', 'b'], alphabet)
{'c': 3, 'd': 4}
```

- **compact** (*iterable*)

  Filter an iterable on "truthy" values.

```
from toolz import filter

def compact(iter):
    return filter(None, iter)
```

  Example:

```
>>> results = [0, 1, 2, None, 3, False]
>>> list(compact(results))
[1, 2, 3]
```

- **keyjoin** (*leftkey*, *leftseq*, *rightkey*, *rightseq*)

  Inner join two sequences of dictionaries on specified keys, merging matches with right value precedence.

```
from itertools import starmap
from toolz import join, merge

def keyjoin(leftkey, leftseq, rightkey, rightseq):
    return starmap(merge, join(leftkey, leftseq, rightkey, rightseq))
```

  Example:

```
>>> people = [{'id': 0, 'name': 'Anonymous Guy', 'location': 'Unknown'},
              {'id': 1, 'name': 'Karan', 'location': 'San Francisco'},
              {'id': 2, 'name': 'Matthew', 'location': 'Oakland'}]
```

```
>>> hobbies = [{'person_id': 1, 'hobby': 'Tennis'},
               {'person_id': 1, 'hobby': 'Acting'},
               {'person_id': 2, 'hobby': 'Biking'}]
>>> list(keyjoin('id', people, 'person_id', hobbies))
[{'hobby': 'Tennis',
  'id': 1,
  'location': 'San Francisco',
  'name': 'Karan',
  'person_id': 1},
 {'hobby': 'Acting',
  'id': 1,
  'location': 'San Francisco',
  'name': 'Karan',
  'person_id': 1},
 {'hobby': 'Biking',
  'id': 2,
  'location': 'Oakland',
  'name': 'Matthew',
  'person_id': 2}]
```

- **areidentical**(**seqs*)

  Determine if sequences are identical element-wise. This lazily evaluates the sequences and stops as soon as the result is determined.

```
from toolz import diff

def areidentical(*seqs):
    return not any(diff(*seqs, default=object()))
```

  Example:

```
>>> areidentical([1, 2, 3], (1, 2, 3))
True
```

```
>>> areidentical([1, 2, 3], [1, 2])
False
```

References

- Underscore.js: A similar library for JavaScript
- Enumerable: A similar library for Ruby
- Clojure: A functional language whose standard library has several counterparts in `toolz`
- itertools: The Python standard library for iterator tools
- functools: The Python standard library for function tools
- Functional Programming HOWTO: The description of functional programming features from the official Python docs.

Contemporary Projects

These projects also provide iterator and functional utilities within Python. Their functionality overlaps substantially with that of PyToolz.

- funcy
- fn.py
- more_itertools

Bibliography

[itertools] http://docs.python.org/2/library/itertools.html

[functools] http://docs.python.org/2/library/functools.html

[itertoolz] http://github.com/pytoolz/itertoolz

[functoolz] http://github.com/pytoolz/functoolz

[Underscore.js] http://underscorejs.org

[cheatsheet] http://clojure.org/cheatsheet

[Guido] http://python-history.blogspot.com/2009/04/origins-of-pythons-functional-features.html

[Enumerable] http://ruby-doc.org/core-2.0.0/Enumerable.html

[funcy] https://github.com/suor/funcy/

[fn.py] https://github.com/kachayev/fn.py

Python Module Index

t

Index

R

S

T

U

V