

## NAME

List::Util - A selection of general-utility list subroutines

## SYNOPSIS

```
use List::Util qw(first max maxstr min minstr reduce shuffle sum);
```

## DESCRIPTION

`List::Util` contains a selection of subroutines that people have expressed would be nice to have in the perl core, but the usage would not really be high enough to warrant the use of a keyword, and the size so small such that being individual extensions would be wasteful.

By default `List::Util` does not export any subroutines.

## LIST-REDUCTION FUNCTIONS

The following set of functions all reduce a list down to a single value.

```
$result = reduce { BLOCK } @list
```

Reduces `@list` by calling `BLOCK` in a scalar context multiple times, setting `$a` and `$b` each time. The first call will be with `$a` and `$b` set to the first two elements of the list, subsequent calls will be done by setting `$a` to the result of the previous call and `$b` to the next element in the list.

Returns the result of the last call to the `BLOCK`. If `@list` is empty then `undef` is returned. If `@list` only contains one element then that element is returned and `BLOCK` is not executed.

The following examples all demonstrate how `reduce` could be used to implement the other list-reduction functions in this module. (They are not in fact implemented like this, but instead in a more efficient manner in individual C functions).

```
$foo = reduce { defined($a) ? $a :  
$code->(local $_ = $b) ? $b :  
undef } undef, @list # first
```

```
$foo = reduce { $a > $b ? $a : $b } 1..10 # max  
$foo = reduce { $a gt $b ? $a : $b } 'A'..'Z' # maxstr  
$foo = reduce { $a < $b ? $a : $b } 1..10 # min  
$foo = reduce { $a lt $b ? $a : $b } 'aa'..'zz' # minstr  
$foo = reduce { $a + $b } 1 .. 10 # sum  
$foo = reduce { $a . $b } @bar # concat
```

```
$foo = reduce { $a || $code->(local $_ = $b) } 0, @bar # any  
$foo = reduce { $a && $code->(local $_ = $b) } 1, @bar # all  
$foo = reduce { $a && !$code->(local $_ = $b) } 1, @bar # none  
$foo = reduce { $a || !$code->(local $_ = $b) } 0, @bar # notall  
# Note that these implementations do not fully short-circuit
```

If your algorithm requires that `reduce` produce an identity value, then make sure that you always pass that identity value as the first argument to prevent `undef` being returned

```
$foo = reduce { $a + $b } 0, @values; # sum with 0 identity value
```

The remaining list-reduction functions are all specialisations of this generic idea.

```
$b = any { BLOCK } @list
```

Similar to `grep` in that it evaluates `BLOCK` setting `$_` to each element of `@list` in turn. `any` returns true if any element makes the `BLOCK` return a true value. If `BLOCK` never returns true or `@list` was empty then it returns false.

Many cases of using `grep` in a conditional can be written using `any` instead, as it can short-circuit after the first true result.

```

    if( any { length > 10 } @strings ) {
        # at least one string has more than 10 characters
    }

```

**\$b = all { BLOCK } @list**

Similar to **any**, except that it requires all elements of the **@list** to make the **BLOCK** return true. If any element returns false, then it returns false. If the **BLOCK** never returns false or the **@list** was empty then it returns true.

**\$b = none { BLOCK } @list**

**\$b = notall { BLOCK } @list**

Similar to **any** and **all**, but with the return sense inverted. **none** returns true only if no value in the **LIST** causes the **BLOCK** to return true, and **notall** returns true only if not all of the values do.

**\$val = first { BLOCK } @list**

Similar to **grep** in that it evaluates **BLOCK** setting **\$\_** to each element of **@list** in turn. **first** returns the first element where the result from **BLOCK** is a true value. If **BLOCK** never returns true or **@list** was empty then **undef** is returned.

```

    $foo = first { defined($_) } @list # first defined value in @list
    $foo = first { $_ > $value } @list # first value in @list which
    # is greater than $value

```

**\$num = max @list**

Returns the entry in the list with the highest numerical value. If the list is empty then **undef** is returned.

```

    $foo = max 1..10 # 10
    $foo = max 3,9,12 # 12
    $foo = max @bar, @baz # whatever

```

**\$str = maxstr @list**

Similar to **max**, but treats all the entries in the list as strings and returns the highest string as defined by the **gt** operator. If the list is empty then **undef** is returned.

```

    $foo = maxstr 'A'..'Z' # 'Z'
    $foo = maxstr "hello","world" # "world"
    $foo = maxstr @bar, @baz # whatever

```

**\$num = min @list**

Similar to **max** but returns the entry in the list with the lowest numerical value. If the list is empty then **undef** is returned.

```

    $foo = min 1..10 # 1
    $foo = min 3,9,12 # 3
    $foo = min @bar, @baz # whatever

```

**\$str = minstr @list**

Similar to **min**, but treats all the entries in the list as strings and returns the lowest string as defined by the **lt** operator. If the list is empty then **undef** is returned.

```

    $foo = minstr 'A'..'Z' # 'A'
    $foo = minstr "hello","world" # "hello"
    $foo = minstr @bar, @baz # whatever

```

**\$num = product @list**

Returns the numerical product of all the elements in **@list**. If **@list** is empty then 1 is returned.

```

    $foo = product 1..10 # 3628800
    $foo = product 3,9,12 # 324

```

```
$num_or_undef = sum @list
```

Returns the numerical sum of all the elements in `@list`. For backwards compatibility, if `@list` is empty then `undef` is returned.

```
$foo = sum 1..10 # 55
$foo = sum 3,9,12 # 24
$foo = sum @bar, @baz # whatever
```

```
$num = sum0 @list
```

Similar to `sum`, except this returns 0 when given an empty list, rather than `undef`.

## KEY/VALUE PAIR LIST FUNCTIONS

The following set of functions, all inspired by `List::Pairwise`, consume an even-sized list of pairs. The pairs may be key/value associations from a hash, or just a list of values. The functions will all preserve the original ordering of the pairs, and will not be confused by multiple pairs having the same “key” value - nor even do they require that the first of each pair be a plain string.

```
@kvlist = pairgrep { BLOCK } @kvlist
```

```
$count = pairgrep { BLOCK } @kvlist
```

Similar to perl’s `grep` keyword, but interprets the given list as an even-sized list of pairs. It invokes the `BLOCK` multiple times, in scalar context, with `$a` and `$b` set to successive pairs of values from the `@kvlist`.

Returns an even-sized list of those pairs for which the `BLOCK` returned true in list context, or the count of the **number of pairs** in scalar context. (Note, therefore, in scalar context that it returns a number half the size of the count of items it would have returned in list context).

```
@subset = pairgrep { $a = m/[[:upper:]]+$/ } @kvlist
```

As with `grep` aliasing `$_` to list elements, `pairgrep` aliases `$a` and `$b` to elements of the given list. Any modifications of it by the code block will be visible to the caller.

```
( $key, $val ) = pairfirst { BLOCK } @kvlist
```

```
$found = pairfirst { BLOCK } @kvlist
```

Similar to the `first` function, but interprets the given list as an even-sized list of pairs. It invokes the `BLOCK` multiple times, in scalar context, with `$a` and `$b` set to successive pairs of values from the `@kvlist`.

Returns the first pair of values from the list for which the `BLOCK` returned true in list context, or an empty list of no such pair was found. In scalar context it returns a simple boolean value, rather than either the key or the value found.

```
( $key, $value ) = pairfirst { $a = m/[[:upper:]]+$/ } @kvlist
```

As with `grep` aliasing `$_` to list elements, `pairfirst` aliases `$a` and `$b` to elements of the given list. Any modifications of it by the code block will be visible to the caller.

```
@list = pairmap { BLOCK } @kvlist
```

```
$count = pairmap { BLOCK } @kvlist
```

Similar to perl’s `map` keyword, but interprets the given list as an even-sized list of pairs. It invokes the `BLOCK` multiple times, in list context, with `$a` and `$b` set to successive pairs of values from the `@kvlist`.

Returns the concatenation of all the values returned by the `BLOCK` in list context, or the count of the number of items that would have been returned in scalar context.

```
@result = pairmap { "The key $a has value $b" } @kvlist
```

As with `map` aliasing `$_` to list elements, `pairmap` aliases `$a` and `$b` to elements of the given list. Any modifications of it by the code block will be visible to the caller.

```
@pairs = pairs @kvlist
```

A convenient shortcut to operating on even-sized lists of pairs, this function returns a list of `ARRAY` references, each containing two items from the given list. It is a more efficient version of

```
@pairs = pairmap { [ $a, $b ] } @kvlist
```

It is most convenient to use in a `foreach` loop, for example:

```
foreach ( pairs @KVLIST ) {
  my ( $key, $value ) = @$_;
  ...
}
```

```
@keys = pairkeys @kvlist
```

A convenient shortcut to operating on even-sized lists of pairs, this function returns a list of the first values of each of the pairs in the given list. It is a more efficient version of

```
@keys = pairmap { $a } @kvlist
```

```
@values = pairvalues @kvlist
```

A convenient shortcut to operating on even-sized lists of pairs, this function returns a list of the second values of each of the pairs in the given list. It is a more efficient version of

```
@values = pairmap { $b } @kvlist
```

## OTHER FUNCTIONS

```
@values = shuffle @values
```

Returns the values of the input in a random order

```
@cards = shuffle 0..51 # 0..51 in a random order
```

## KNOWN BUGS

With perl versions prior to 5.005 there are some cases where `reduce` will return an incorrect result. This will show up as test 7 of `reduce.t` failing.

## SUGGESTED ADDITIONS

The following are additions that have been requested, but I have been reluctant to add due to them being very simple to implement in perl

```
# How many elements are true
```

```
sub true { scalar grep { $_ } @_ }
```

```
# How many elements are false
```

```
sub false { scalar grep { !$_ } @_ }
```

## SEE ALSO

[Scalar::Util](#), [List::MoreUtils](#)

## COPYRIGHT

Copyright (c) 1997-2007 Graham Barr <gbarr@pobox.com>. All rights reserved. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

Recent additions and current maintenance by Paul Evans, <leonerdd@leonerdd.org.uk>.