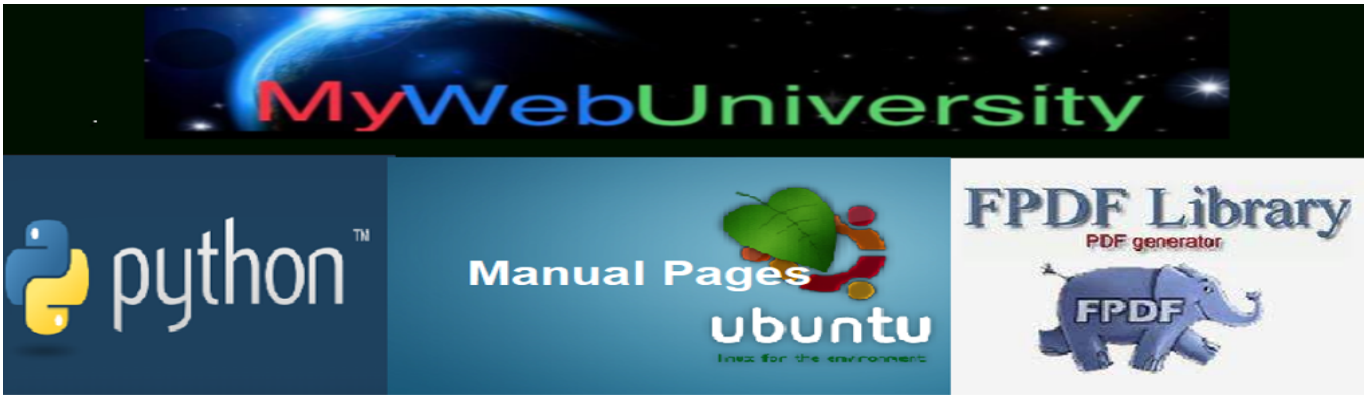

Rocky Enterprise Linux 9.2 Manual Pages on command 'List::Util.3perl'

$ man List::Util.3perl

List::Util(3perl) Perl Programmers Reference Guide List::Util(3perl)

NAME

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

SYNOPSIS

```
use List::Util qw(
  reduce any all none notall first reductions
  max maxstr min minstr product sum sum0
  pairs unpairs pairkeys pairvalues pairfirst pairgrep pairmap
  shuffle uniq uniqint uniqnum uniqstr
);
```

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 apply a given block of code to a list of values.

reduce

```
$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 above example code blocks also suggest how to use "reduce" to build a more efficient combined version of one of these basic functions and a "map" block. For example, to find the total length of all the strings in a list, we could use

```
$total = sum map { length } @strings;
```

However, this produces a list of temporary integer values as long as the original list of strings, only to reduce it down to a single value again. We can compute the same result more efficiently by using "reduce" with a code block that accumulates lengths by writing this instead as:

```
    $total = reduce { $a + length $b } 0, @strings
```

The other scalar-returning list reduction functions are all specialisations of this generic idea.

reductions

```
    @results = reductions { BLOCK } @list
```

Since version 1.54.

Similar to "reduce" except that it also returns the intermediate values along with the final result. As before, $a is set to the first element of the given list, and the "BLOCK" is then called once for remaining item in the list set into $b, with the result being captured for return as well as becoming the new value for $a.

The returned list will begin with the initial value for $a, followed by each return value from the block in order. The final value of the result will be identical to what the "reduce" function would have returned given the same block and list.

```
    reduce     { "$a-$b" }  "a".."d"    # "a-b-c-d"
    reductions { "$a-$b" }  "a".."d"    # "a", "a-b", "a-b-c", "a-b-c-d"
```

any

```
    my $bool = any { BLOCK } @list;
```

Since version 1.33.

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
    }
```

Note: Due to XS issues the block passed may be able to access the outer @_ directly. This is not intentional and will break under debugger.

all

```
    my $bool = all { BLOCK } @list;
```

Since version 1.33.

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.

Note: Due to XS issues the block passed may be able to access the outer @_ directly. This is not intentional and will break under debugger.

none

notall

```
my $bool = none { BLOCK } @list;
my $bool = notall { BLOCK } @list;
```

Since version 1.33.

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.

Note: Due to XS issues the block passed may be able to access the outer @_ directly. This is not intentional and will break under debugger.

first

```
my $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
```

max

```
my $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
```

maxstr

```
my $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
```

min

```
    my $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
```

minstr

```
    my $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
```

product

```
    my $num = product @list;
```

Since version 1.35.

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
```

sum

```
    my $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
```

sum0

```
    my $num = sum0 @list;
```

Since version 1.26.

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.

NOTE: At the time of writing, the following "pair*" functions that take a block do not modify the value of $_ within the block, and instead operate using the $a and $b globals instead. This has turned out to be a poor design, as it precludes the ability to provide a "pairsort" function. Better would be to pass pair-like objects as 2-element array references in $_, in a style similar to the return value of the "pairs" function. At some future version this behaviour may be added.

Until then, users are alerted NOT to rely on the value of $_ remaining unmodified between the outside and the inside of the control block. In particular, the following example is UNSAFE:

```
 my @kvlist = ...
 foreach (qw( some keys here )) {
    my @items = pairgrep { $a eq $_ } @kvlist;
    ...
 }
```

Instead, write this using a lexical variable:

```
 foreach my $key (qw( some keys here )) {
    my @items = pairgrep { $a eq $key } @kvlist;
    ...
 }
```

pairs

```
    my @pairs = pairs @kvlist;
```

Since version 1.29.

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 my $pair ( pairs @kvlist ) {
      my ( $key, $value ) = @$pair;
      ...
    }
```

Since version 1.39 these "ARRAY" references are blessed objects, recognising the two methods "key" and "value". The following code is equivalent:

```
    foreach my $pair ( pairs @kvlist ) {
      my $key   = $pair->key;
      my $value = $pair->value;
      ...
    }
```

Since version 1.51 they also have a "TO_JSON" method to ease serialisation.

unpairs

```
    my @kvlist = unpairs @pairs
```

Since version 1.42.

The inverse function to "pairs"; this function takes a list of "ARRAY" references containing two elements each, and returns a flattened list of the two values from each of the pairs, in order. This is notionally equivalent to

```
    my @kvlist = map { @{$_}[0,1] } @pairs
```

except that it is implemented more efficiently internally. Specifically, for any input item it will extract exactly two values for the output list; using "undef" if the input array references are short.

Between "pairs" and "unpairs", a higher-order list function can be used to operate on the pairs as single scalars; such as the following near-equivalents of the other "pair*" higher-order functions:

```
    @kvlist = unpairs grep { FUNC } pairs @kvlist
    # Like pairgrep, but takes $_ instead of $a and $b
    @kvlist = unpairs map { FUNC } pairs @kvlist
    # Like pairmap, but takes $_ instead of $a and $b
```

Note however that these versions will not behave as nicely in scalar context.

Finally, this technique can be used to implement a sort on a keyvalue pair list; e.g.:

```
    @kvlist = unpairs sort { $a->key cmp $b->key } pairs @kvlist
```

pairkeys

```
    my @keys = pairkeys @kvlist;
```

Since version 1.29.

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

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

pairvalues

```
    my @values = pairvalues @kvlist;
```

Since version 1.29.

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

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

pairgrep

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

Since version 1.29.

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.

pairfirst

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

Since version 1.30.

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.

pairmap

```
my @list = pairmap { BLOCK } @kvlist;
my $count = pairmap { BLOCK } @kvlist;
```

Since version 1.29.

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.

See "KNOWN BUGS" for a known-bug with "pairmap", and a workaround.

OTHER FUNCTIONS

shuffle

```
my @values = shuffle @values;
```

Returns the values of the input in a random order

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

This function is affected by the $RAND variable.

sample

```
my @items = sample $count, @values
```

Since version 1.54.

Randomly select the given number of elements from the input list. Any given position in the input list will be selected at most once.

If there are fewer than $count items in the list then the function will return once all of them have been randomly selected; effectively the function behaves similarly to "shuffle". This function is affected by the $RAND variable.

uniq

```
my @subset = uniq @values
```

Since version 1.45.

Filters a list of values to remove subsequent duplicates, as judged by a DWIM-ish string equality or "undef" test. Preserves the order of unique elements, and retains the first value of any duplicate set.

```
my $count = uniq @values
```

In scalar context, returns the number of elements that would have been returned as a list.

The "undef" value is treated by this function as distinct from the empty string, and no warning will be produced. It is left as-is in the returned list. Subsequent "undef" values are still considered identical to the first, and will be removed.

uniqint

```
my @subset = uniqint @values
```

Since version 1.55.

Filters a list of values to remove subsequent duplicates, as judged by an integer numerical equality test. Preserves the order of unique elements, and retains the first value of any duplicate set. Values in the returned list will be coerced into integers.

```
my $count = uniqint @values
```

In scalar context, returns the number of elements that would have been returned as a list.

Note that "undef" is treated much as other numerical operations treat it; it compares equal to zero but additionally produces a warning if such warnings are enabled ("use warnings 'uninitialized';"). In addition, an "undef" in the returned list is coerced into a numerical zero, so that the entire list of values returned by "uniqint" are well-behaved as integers.

uniqnum

```
my @subset = uniqnum @values
```

Since version 1.44.

Filters a list of values to remove subsequent duplicates, as judged by a numerical equality test. Preserves the order of unique elements, and retains the first value of any duplicate set.

```
my $count = uniqnum @values
```

In scalar context, returns the number of elements that would have been returned as a list.

Note that "undef" is treated much as other numerical operations treat it; it compares equal to zero but additionally produces a warning if such warnings are enabled ("use warnings 'uninitialized';"). In addition, an "undef" in the returned list is coerced into a numerical zero, so that the entire list of values returned by "uniqnum" are well-behaved as numbers.

Note also that multiple IEEE "NaN" values are treated as duplicates of each other, regardless of any differences in their payloads, and despite the fact that "0+'NaN' == 0+'NaN'" yields false.

uniqstr

```
my @subset = uniqstr @values
```

Since version 1.45.

Filters a list of values to remove subsequent duplicates, as judged by a string equality test. Preserves the order of unique elements, and retains the first value of any duplicate set.

```
my $count = uniqstr @values
```

In scalar context, returns the number of elements that would have been returned as a list.

Note that "undef" is treated much as other string operations treat it; it compares equal to the empty string but additionally produces a warning if such warnings are enabled ("use warnings 'uninitialized';"). In addition, an "undef" in the returned list is coerced into an empty string, so that the entire list of values returned by "uniqstr" are well-behaved as strings.

head

```
my @values = head $size, @list;
```

Since version 1.50.

Returns the first $size elements from @list. If $size is negative, returns all but the last $size elements from @list.

```
@result = head 2, qw( foo bar baz );
# foo, bar
@result = head -2, qw( foo bar baz );
# foo
```

tail

```
    my @values = tail $size, @list;
```

Since version 1.50.

Returns the last $size elements from @list. If $size is negative, returns all but the first $size elements from @list.

```
    @result = tail 2, qw( foo bar baz );
    # bar, baz
    @result = tail -2, qw( foo bar baz );
    # baz
```

CONFIGURATION VARIABLES

$RAND

```
    local $List::Util::RAND = sub { ... };
```

Since version 1.54.

This package variable is used by code which needs to generate random numbers (such as the "shuffle" and "sample" functions). If set to a CODE reference it provides an alternative to perl's builtin "rand()" function. When a new random number is needed this function will be invoked with no arguments and is expected to return a floating-point value, of which only the fractional part will be used.

KNOWN BUGS

RT #95409

<https://rt.cpan.org/Ticket/Display.html?id=95409>

If the block of code given to "pairmap" contains lexical variables that are captured by a returned closure, and the closure is executed after the block has been re-used for the next iteration, these lexicals will not see the correct values. For example:

```
 my @subs = pairmap {
    my $var = "$a is $b";
    sub { print "$var\n" };
 } one => 1, two => 2, three => 3;
 $_->() for @subs;
```

Will incorrectly print

```
 three is 3
 three is 3
 three is 3
```

This is due to the performance optimisation of using "MULTICALL" for the code block, which

means that fresh SVs do not get allocated for each call to the block. Instead, the same SV is re-assigned for each iteration, and all the closures will share the value seen on the final iteration.

To work around this bug, surround the code with a second set of braces. This creates an inner block that defeats the "MULTICALL" logic, and does get fresh SVs allocated each time:

```
 my @subs = pairmap {
   {
      my $var = "$a is $b";
      sub { print "$var\n"; }
   }
 } one => 1, two => 2, three => 3;
```

This bug only affects closures that are generated by the block but used afterwards. Lexical variables that are only used during the lifetime of the block's execution will take their individual values for each invocation, as normal.

uniqnum() on oversized bignums

Due to the way that "uniqnum()" compares numbers, it cannot distinguish differences between bignums (especially bigints) that are too large to fit in the native platform types. For example,

```
 my $x = Math::BigInt->new( "1" x 100 );
 my $y = $x + 1;
 say for uniqnum( $x, $y );
```

Will print just the value of $x, believing that $y is a numerically- equivalent value.

This bug does not affect "uniqstr()", which will correctly observe that the two values stringify to different strings.

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

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