NAME

List::Compare::Functional – Compare elements of two or more lists

VERSION

This document refers to version 0.53 of List::Compare::Functional. This version was released June 07 2015. The first released version of List::Compare::Functional was v0.21. Its version numbers are set to be consistent with the other parts of the List::Compare distribution.

Notice of Interface Changes

Certain significant changes to the interface to List::Compare::Functional were made with the introduction of Version 0.25 in April 2004. The documentation immediately below reflects those changes, so if you are first using this module with that or a later version, simply read and follow the documentation below. If, however, you used List::Compare::Functional prior to that version, see the discussion of interface changes farther below: April 2004 Change of Interface.

SYNOPSIS

Getting Started

List::Compare::Functional exports no subroutines by default.

```
use List::Compare::Functional qw(:originals :aliases);
```

will import all publicly available subroutines from List::Compare::Functional. The model for importing just one subroutine from List::Compare::Functional is:

```
use List::Compare::Functional qw( get_intersection );
```

It will probably be most convenient for the user to import functions by using one of the two following export tags:

```
use List::Compare::Functional qw(:main :mainrefs);
```

The assignment of the various comparison functions to export tags is discussed below.

For clarity, we shall begin by discussing comparisons of just two lists at a time. Farther below, we shall discuss comparisons among three or more lists at a time.

Comparing Two Lists Held in Arrays

- Given two lists:

```
@Llist = qw(abel abel baker camera delta edward fargo golfer);
@Rlist = qw(baker camera delta delta edward fargo golfer hilton);
```

- Get those items which appear at least once in both lists (their intersection).

```
@intersection = get_intersection( [ \@Llist, \@Rlist ] );
```

Note that you could place the references to the lists being compared into a named array and then pass `get_intersection()` a reference to that array.

```
@to_be_compared = ( \@Llist, \@Rlist );
@intersection = get_intersection( \@to_be_compared );
```

Beginning with version 0.29 (May 2004), List::Compare::Functional now offers an additional way of passing arguments to its various functions. If you prefer to see a more explicit delineation among the types of arguments passed to a function, pass a single hash reference which holds the lists being compared in an anonymous array which is the value corresponding to key `lists`:

```
@intersection = get_intersection( {
   lists => [ \@Llist, \@Rlist ],
} );
```

- Get those items which appear at least once in either list (their union).

```
@union = get_union( [ \@Llist, \@Rlist ] );
```

or

```
    @union = get_union( { lists => [ \@Llist, \@Rlist ] } );
```

- Get those items which appear (at least once) only in the first list.

```
    @Lonly = get_unique( [ \@Llist, \@Rlist ] );
```

or

```
    @Lonly = get_unique( { lists => [ \@Llist, \@Rlist ] } );
```

- Get those items which appear (at least once) only in the second list.

```
    @Ronly = get_complement( [ \@Llist, \@Rlist ] );
```

or

```
    @Ronly = get_complement( { lists => [ \@Llist, \@Rlist ] } );
```

-

```
    @LorRonly = get_symmetric_difference( [ \@Llist, \@Rlist ] );

    @LorRonly = get_symdiff( [ \@Llist, \@Rlist ] );       # alias
```

or

```
    @LorRonly = get_symmetric_difference( { lists => [ \@Llist, \@Rlist ] } );
```

- Make a bag of all those items in both lists. The bag differs from the union of the two lists in that it holds as many copies of individual elements as appear in the original lists.

```
    @bag = get_bag( [ \@Llist, \@Rlist ] );
```

or

```
    @bag = get_bag( { lists => [ \@Llist, \@Rlist ] } );
```

- An alternative approach to the above functions: If you do not immediately require an array as the return value of the function call, but simply need a *reference* to an (anonymous) array, use one of the following parallel functions:

```
    $intersection_ref = get_intersection_ref(         [ \@Llist, \@Rlist ] );
    $union_ref        = get_union_ref(                [ \@Llist, \@Rlist ] );
    $Lonly_ref        = get_unique_ref(               [ \@Llist, \@Rlist ] );
    $Ronly_ref        = get_complement_ref(           [ \@Llist, \@Rlist ] );
    $LorRonly_ref     = get_symmetric_difference_ref( [ \@Llist, \@Rlist ] );
    $LorRonly_ref     = get_symdiff_ref(              [ \@Llist, \@Rlist ] );
                            # alias
    $bag_ref          = get_bag_ref(                  [ \@Llist, \@Rlist ] );
```

or

```
    $intersection_ref =
        get_intersection_ref(         { lists => [ \@Llist, \@Rlist ] } );
    $union_ref        =
        get_union_ref(                { lists => [ \@Llist, \@Rlist ] } );
    $Lonly_ref        =
        get_unique_ref(               { lists => [ \@Llist, \@Rlist ] } );
    $Ronly_ref        =
        get_complement_ref(           { lists => [ \@Llist, \@Rlist ] } );
    $LorRonly_ref     =
        get_symmetric_difference_ref( { lists => [ \@Llist, \@Rlist ] } );
    $LorRonly_ref     =
        get_symdiff_ref(              { lists => [ \@Llist, \@Rlist ] } );
```

```
    # alias
$bag_ref          =
    get_bag_ref(                  { lists => [ \@Llist, \@Rlist ] } );
```

- Return a true value if the first list ('L' for 'left') is a subset of the second list ('R' for 'right').

  ```
  $LR = is_LsubsetR( [ \@Llist, \@Rlist ] );
  ```

  or

  ```
  $LR = is_LsubsetR( { lists => [ \@Llist, \@Rlist ] } );
  ```

- Return a true value if R is a subset of L.

  ```
  $RL = is_RsubsetL( [ \@Llist, \@Rlist ] );
  ```

  or

  ```
  $RL = is_RsubsetL( { lists => [ \@Llist, \@Rlist ] } );
  ```

- Return a true value if L and R are equivalent, *i.e.,* if every element in L appears at least once in R and *vice versa*.

  ```
  $eqv = is_LequivalentR( [ \@Llist, \@Rlist ] );
  $eqv = is_LeqvlntR( [ \@Llist, \@Rlist ] );            # alias
  ```

  or

  ```
  $eqv = is_LequivalentR( { lists => [ \@Llist, \@Rlist ] } );
  ```

- Return a true value if L and R are disjoint, *i.e.,* if L and R have no common elements.

  ```
  $disj = is_LdisjointR( [ \@Llist, \@Rlist ] );
  ```

  or

  ```
  $disj = is_LdisjointR( { lists => [ \@Llist, \@Rlist ] } );
  ```

- Pretty-print a chart showing whether one list is a subset of the other.

  ```
  print_subset_chart( [ \@Llist, \@Rlist ] );
  ```

  or

  ```
  print_subset_chart( { lists => [ \@Llist, \@Rlist ] } );
  ```

- Pretty-print a chart showing whether the two lists are equivalent (same elements found at least once in both).

  ```
  print_equivalence_chart( [ \@Llist, \@Rlist ] );
  ```

  or

  ```
  print_equivalence_chart( { lists => [ \@Llist, \@Rlist ] } );
  ```

- Determine in *which* (if any) of the lists a given string can be found. In list context, return a list of those indices in the argument list corresponding to lists holding the string being tested.

  ```
  @memb_arr = is_member_which( [ \@Llist, \@Rlist ] , [ 'abel' ] );
  ```

  or

  ```
  @memb_arr = is_member_which( {
      lists => [ \@Llist, \@Rlist ],  # value is array reference
      item  => 'abel',                # value is string
  } );
  ```

  In the example above, `@memb_arr` will be:

```
( 0 )
```

because 'abel' is found only in @Al which holds position 0 in the list of arguments passed to new().

- As with other List::Compare::Functional functions which return a list, you may wish the above function returned a (scalar) reference to an array holding the list:

```
$memb_arr_ref = is_member_which_ref( [ \@Llist, \@Rlist ] , [ 'baker' ] );
```

or

```
$memb_arr_ref = is_member_which_ref( {
    lists => [ \@Llist, \@Rlist ],  # value is array reference
    item  => 'baker',               # value is string
} );
```

In the example above, $memb_arr_ref will be:

```
[ 0, 1 ]
```

because 'baker' is found in @Llist and @Rlist, which hold positions 0 and 1, respectively, in the list of arguments passed to new().

Note: functions is_member_which() and is_member_which_ref test only one string at a time and hence take only one argument. To test more than one string at a time see the next function, are_members_which().

- Determine in which (if any) of the lists passed as arguments one or more given strings can be found. The lists beings searched are placed in an array, a reference to which is the first argument passed to are_members_which(). The strings to be tested are also placed in an array, a reference to which is the second argument passed to that function.

```
$memb_hash_ref =
    are_members_which( [ \@Llist, \@Rlist ] ,
                       [ qw| abel baker fargo hilton zebra | ]
                     );
```

or

```
$memb_hash_ref = are_members_which( {
    lists => [ \@Llist, \@Rlist ],                    # value is arrayref
    items => [ qw| abel baker fargo hilton zebra | ], # value is arrayref
} );
```

The return value is a reference to a hash of arrays. The key for each element in this hash is the string being tested. Each element's value is a reference to an anonymous array whose elements are those indices in the constructor's argument list corresponding to lists holding the strings being tested. In the examples above, $memb_hash_ref will be:

```
{
     abel     => [ 0    ],
     baker    => [ 0, 1 ],
     fargo    => [ 0, 1 ],
     hilton   => [    1 ],
     zebra    => [      ],
};
```

Note: are_members_which() can take more than one argument; is_member_which() and is_member_which_ref() each take only one argument. Unlike those functions, are_members_which() returns a hash reference.

- Determine whether a given string can be found in *any* of the lists passed as arguments. Return `1` if a specified string can be found in any of the lists and `0` if not.

```
$found = is_member_any( [ \@Llist, \@Rlist ] , [ 'abel' ] );
```

or

```
$found = is_member_any( {
    lists => [ \@Llist, \@Rlist ], # value is array reference
    item  => 'abel',               # value is string
} );
```

In the example above, `$found` will be `1` because `'abel'` is found in one or more of the lists passed as arguments to `new()`.

- Determine whether a specified string or strings can be found in *any* of the lists passed as arguments. The lists beings searched are placed in an array, a reference to which is the first argument passed to `are_members_any()`. The strings to be tested are also placed in an array, a reference to which is the second argument passed to that function.

```
$memb_hash_ref =
    are_members_any( [ \@Llist, \@Rlist ] ,
                     [ qw| abel baker fargo hilton zebra | ]
                   );
```

or

```
$memb_hash_ref = are_members_any( {
    lists => [ \@Llist, \@Rlist ],                    # value is arrayref
    items => [ qw| abel baker fargo hilton zebra | ], # value is arrayref
} );
```

The return value is a reference to a hash where an element's key is the string being tested and the element's value is `1` if the string can be found in *any* of the lists and `0` if not. In the examples above, `$memb_hash_ref` will be:

```
{
     abel     => 1,
     baker    => 1,
     fargo    => 1,
     hilton   => 1,
     zebra    => 0,
};
```

`zebra`'s value is `0` because `zebra` is not found in either of the lists passed as arguments to `are_members_any()`.

- Return current List::Compare::Functional version number.

```
$vers = get_version;
```

Comparing Three or More Lists Held in Arrays

Given five lists:

```
@Al     = qw(abel abel baker camera delta edward fargo golfer);
@Bob    = qw(baker camera delta delta edward fargo golfer hilton);
@Carmen = qw(fargo golfer hilton icon icon jerky kappa);
@Don    = qw(fargo icon jerky);
@Ed     = qw(fargo icon icon jerky);
```

- Get those items which appear at least once in *each* list (their intersection).

```
@intersection = get_intersection( [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ] )
```

or

```
@intersection = get_intersection( {
    lists => [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ],
} );
```

- Get those items which appear at least once in *any* of the lists (their union).

```
@union = get_union( [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ] );
```

or

```
@union = get_union( {
  lists => [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ],
} );
```

- To get those items which are unique to a particular list, provide `get_unique()` with two array references. The first holds references to the arrays which in turn hold the individual lists being compared. The second holds the index position in the first reference of the particular list under consideration. Example: To get elements unique to `@Carmen`:

```
@Lonly = get_unique(
             [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ],
             [ 2 ]
         );
```

or

```
@Lonly = get_unique( {
    lists => [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ], # value is arrayref
    item  => 2,                                      # value is number
} );
```

If no index position is passed to `get_unique()` it will default to `0` and report items unique to the first list passed to the function. Hence,

```
@Lonly = get_unique( [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ] );
```

is same as:

```
@Lonly = get_unique( [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ], [ 0 ] );
```

- Should you need to identify the items unique to *each* of the lists under consideration, call `get_unique_all` and get a reference to an array of array references:

```
$unique_all_ref = get_unique_all(
    [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ]
);
```

or

```
$unique_all_ref = get_unique_all( {
    lists => [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ],
} );
```

- To get those items which appear only in lists *other than* one particular list, pass two array references to the `get_complement()` function. The first holds references to the arrays which in turn hold the individual lists being compared. The second holds the index position in the first reference of the particular list under consideration. Example: to get all the elements found in lists other than `@Don`:

```
@Ronly = get_complement(
             [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ],
             [ 3 ]
         );
```

or

```
@Ronly = get_complement( {
    lists => [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ], # value is arrayref
    item  => 3,                                      # value is number
} );
```

If no index position is passed to `get_complement()` it will default to `0` and report items found in all lists *other than* the first list passed to `get_complement()`.

```
@Lonly = get_complement( [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ] );
```

is same as:

```
@Lonly = get_complement( [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ], [ 0 ] );
```

- Should you need to identify the items not found in *each* of the lists under consideration, call `get_complement_all` and get a reference to an array of array references:

```
$complement_all_ref = get_complement_all(
    [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ]
);
```

or

```
$complement_all_ref = get_complement_all( {
    lists => [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ],
} );
```

- Get those items which do *not* appear in *more than one* of several lists (their symmetric_difference);

```
@LorRonly = get_symmetric_difference( [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed
@LorRonly = get_symdiff( [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ] ); # alias
```

or

```
@LorRonly = get_symmetric_difference( {
    lists => [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ],
} );
```

- Get those items found in *any* of several lists which do *not* appear in `all` of the lists (*i.e.,* all items except those found in the intersection of the lists):

```
@nonintersection = get_nonintersection(
                        [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ] );
```

or

```
@nonintersection = get_nonintersection( {
    lists => [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ],
} );
```

- Get those items which appear in *more than one* of several lists (*i.e.,* all items except those found in their symmetric difference);

```
@shared = get_shared( [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ] );
```

or

```
@shared = get_shared( {
    lists => [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ],
} );
```

- Make a bag of every item found in every list. The bag differs from the union of the two lists in that it holds as many copies of individual elements as appear in the original lists.

```
@bag = get_bag( [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ] );
```

or

```
@bag = get_bag( {
    lists => [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ],
} );
```

- An alternative approach to the above functions: If you do not immediately require an array as the return value of the function, but simply need a *reference* to an array, use one of the following parallel functions:

```
$intersection_ref    = get_intersection_ref(
                           [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ] );
$union_ref           = get_union_ref(
                           [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ] );
$Lonly_ref           = get_unique_ref(
                           [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ] );
$Ronly_ref           = get_complement_ref(
                           [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ] );
$LorRonly_ref        = get_symmetric_difference_ref(
                           [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ] );
$LorRonly_ref        = get_symdiff_ref(            # alias
                           [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ] );
$nonintersection_ref = get_nonintersection_ref(
                           [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ] );
$shared_ref          = get_shared_ref(
                           [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ] );
$bag_ref             = get_bag_ref(
                           [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ] );
```

- To determine whether one particular list is a subset of another of the lists passed to the function, pass to `is_LsubsetR()` two array references. The first of these is a reference to an array of array references, the arrays holding the lists under consideration. The second is a reference to a two-element array consisting of the index of the presumed subset, followed by the index position of the presumed superset. A true value (`1`) is returned if the first (left-hand) element in the second reference list is a subset of the second (right-hand) element; a false value (`0`) is returned otherwise.

  Example: To determine whether `@Ed` is a subset of `@Carmen`, call:

```
$LR = is_LsubsetR(
          [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ],
          [ 4, 2 ]
      );
```

or

```
$LR = is_LsubsetR( {
    lists => [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ], # value is arrayref
    pair  => [ 4, 2 ],                               # value is arrayref
} );
```

  If only the first reference (to the array of lists) is passed to `is_LsubsetR`, then the function's second argument defaults to `(0,1)` and compares the first two lists passed to the constructor. So,

```
$LR = is_LsubsetR([ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ] );
```

  ... is equivalent to:

```
$LR = is_LsubsetR([ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ], [0,1] );
```

- To reverse the order in which the particular lists are evaluated for superset/subset status, call `is_RsubsetL`:

```
$RL = is_RsubsetL([ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ], [2,4] );
```

or

```
$RL = is_RsubsetL( {
    lists => [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ],
    pair  => [ 2, 4 ],
} );
```

- List::Compare::Functional considers two lists to be equivalent if every element in one list appears at least once in R and *vice versa*. To determine whether one particular list passed to the function is equivalent to another of the lists passed to the function, provide `is_LequivalentR()` with two array references. The first is a reference to an array of array references, the arrays holding the lists under consideration. The second of these is a reference to a two-element array consisting of the two lists being tested for equivalence. A true value (`1`) is returned if the lists are equivalent; a false value (`0`) is returned otherwise.

  Example: To determine whether `@Don` and `@Ed` are equivalent, call:

```
$eqv = is_LequivalentR(
            [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ],
            [3,4]
       );

$eqv = is_LeqvlntR(                                 # alias
            [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ],
            [3,4]
       );
```

or

```
$eqv = is_LequivalentR( {
    items => [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ],
    pair  => [3,4],
} );
```

  If no arguments are passed, `is_LequivalentR` defaults to `[0,1]` and compares the first two lists passed to the function. So,

```
$eqv = is_LequivalentR( [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ] );
```

  ... translates to:

```
$eqv = is_LequivalentR( [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ], [0,1] );
```

- To determine whether any two of the lists passed to the function are disjoint from one another (*i.e.,* have no common members), provide `is_LdisjointR()` with two array references. The first is a reference to an array of array references, the arrays holding the lists under consideration. The second of these is a reference to a two-element array consisting of the two lists being tested for disjointedness. A true value (`1`) is returned if the lists are disjoint; a false value (`0`) is returned otherwise.

  Example: To determine whether `@Don` and `@Ed` are disjoint, call:

```
$disj = is_LdisjointR(
            [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ],
            [3,4]
       );
```

or

```
$disj = is_LdisjointR( {
    items => [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ],
    pair  => [3,4]
} );
```

- Pretty-print a chart showing the subset relationships among the various source lists:

```
print_subset_chart( [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ] );
```

or

```
print_subset_chart( { lists => [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ] } );
```

- Pretty-print a chart showing the equivalence relationships among the various source lists:

```
print_equivalence_chart( [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ] );
```

or

```
print_equivalence_chart( { lists => [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ]
```

- Determine in *which* (if any) of several lists a given string can be found. Pass two array references, the first of which holds references to arrays holding the lists under consideration, and the second of which holds a single-item list consisting of the string being tested.

```
@memb_arr = is_member_which(
                [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ],
                [ 'abel' ]
            );
```

or

```
@memb_arr = is_member_which( {
    lists => [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ], # value is arrayref
    item  => 'abel',                                 # value is string
} );
```

In list context, return a list of those indices in the function's argument list corresponding to lists holding the string being tested. In the example above, `@memb_arr` will be:

```
( 0 )
```

because `'abel'` is found only in `@Al` which holds position `0` in the list of arguments passed to `is_member_which()`.

- As with other List::Compare::Functional functions which return a list, you may wish the above function returned a reference to an array holding the list:

```
$memb_arr_ref = is_member_which_ref(
                    [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ],
                    [ 'jerky' ]
                );
```

or

```
$memb_arr_ref = is_member_which_ref( {
    lists => [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ], # value is arrayref
    item  => 'jerky',                                # value is string
} );
```

In the example above, `$memb_arr_ref` will be:

```
[ 3, 4 ]
```

because `'jerky'` is found in `@Don` and `@Ed`, which hold positions `3` and `4`, respectively, in the list of arguments passed to `is_member_which()`.

Note: functions `is_member_which()` and `is_member_which_ref` test only one string at a time and hence take only one element in the second array reference argument. To test more than one string at a time see the next function, `are_members_which()`.

- Determine in `which` (if any) of several lists one or more given strings can be found. Pass two array references, the first of which holds references to arrays holding the lists under consideration, and the second of which holds a list of the strings being tested.

```
$memb_hash_ref = are_members_which(
                     [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ],
                     [ qw| abel baker fargo hilton zebra | ]
                 );
```

or

```
$memb_hash_ref = are_members_which( {
    lists => [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ],  # value is arrayref
    items => [ qw| abel baker fargo hilton zebra | ], # value is arrayref
} );
```

The return valus is a reference to a hash of arrays. In this hash, each element's value is a reference to an anonymous array whose elements are those indices in the argument list corresponding to lists holding the strings being tested. In the two examples above, `$memb_hash_ref` will be:

```
{
     abel     => [ 0             ],
     baker    => [ 0, 1          ],
     fargo    => [ 0, 1, 2, 3, 4 ],
     hilton   => [    1, 2       ],
     zebra    => [               ],
};
```

Note: `are_members_which()` tests more than one string at a time. Hence, its second array reference argument can take more than one element. `is_member_which()` and `is_member_which_ref()` each take only one element in their second array reference arguments. `are_members_which()` returns a hash reference; the other functions return either a list or a reference to an array holding that list, depending on context.

- Determine whether a given string can be found in *any* of several lists. Pass two array references, the first of which holds references to arrays holding the lists under consideration, and the second of which holds a single-item list of the string being tested.

```
$found = is_member_any(
                [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ],
                [ 'abel' ]
            );
```

or

```
$found = is_member_any( {
    lists => [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ], # value is arrayref
    item  => 'abel',                                 # value is string
} );
```

The return value is `1` if a specified string can be found in *any* of the lists and `0` if not. In the example above, `$found` will be `1` because `abel` is found in one or more of the lists passed as arguments to `is_member_any()`.

- Determine whether a specified string or strings can be found in *any* of several lists. Pass two array references, the first of which holds references to arrays holding the lists under consideration, and the second of which holds a list of the strings being tested.

```
    $memb_hash_ref = are_members_any(
                         [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ],
                         [ qw| abel baker fargo hilton zebra | ]
                     );
```

or

```
    $memb_hash_ref = are_members_any( {
        lists => [ \@Al, \@Bob, \@Carmen, \@Don, \@Ed ],  # value is arrayref
        items => [ qw| abel baker fargo hilton zebra | ], # value is arrayref
    } );
```

The return value is a reference to a hash where an element's key is the string being tested and the element's value is `1` if the string can be found in any of the lists and `0` if not. In the example above, `$memb_hash_ref` will be:

```
    {
         abel     => 1,
         baker    => 1,
         fargo    => 1,
         hilton   => 1,
         zebra    => 0,
    };
```

`zebra`'s value is `0` because `zebra` is not found in any of the lists passed as arguments to `are_members_any()`.

- Return current List::Compare::Functional version number:

```
    $vers = get_version;
```

Comparing Lists Held in Seen-Hashes

What is a seen-hash? A seen-hash is a typical Perl implementation of a look-up table: a hash where the value for a given element represents the number of times the element's key is observed in a list. For the purposes of List::Compare::Functional, what is crucial is whether an item is observed in a list or not; how many times the item occurs in a list is, *with one exception,* irrelevant. (That exception is the `get_bag()` function and its fraternal twin `get_bag_ref()`. In this case only, the key in each element of the seen-hash is placed in the bag the number of times indicated by the value of that element.) The value of an element in a List::Compare seen-hash must be a positive integer, but whether that integer is 1 or 1,000,001 is immaterial for all List::Compare::Functional functions *except* forming a bag.

The two lists compared above were represented by arrays; references to those arrays were passed to the various List::Compare::Functional functions. They could, however, have been represented by seen-hashes such as the following and passed in exactly the same manner to the various functions.

```
%Llist = (
    abel   => 2,
    baker  => 1,
    camera => 1,
    delta  => 1,
    edward => 1,
    fargo  => 1,
    golfer => 1,
);
%Rlist = (
    baker  => 1,
    camera => 1,
    delta  => 2,
    edward => 1,
    fargo  => 1,
```

```
    golfer => 1,
    hilton => 1,
);

@intersection = get_intersection( [ \%Llist, \%Rlist ] );
@union        = get_union(        [ \%Llist, \%Rlist ] );
@complement   = get_complement(   [ \%Llist, \%Rlist ] );
```

and so forth.

To compare three or more lists simultaneously, provide the appropriate List::Compare::Functional function with a first array reference holding a list of three or more references to seen-hashes. Thus,

```
@union = get_intersection( [ \%Alpha, \%Beta, \%Gamma ] );
```

The 'single hashref' format for List::Compare::Functional functions is also available when passing seen-hashes as arguments. Examples:

```
@intersection = get_intersection( {
    lists => [ \%Alpha, \%Beta, \%Gamma ],
} );

@Ronly = get_complement( {
    lists => [ \%Alpha, \%Beta, \%Gamma ],
    item  => 3,
} );

$LR = is_LsubsetR( {
    lists => [ \%Alpha, \%Beta, \%Gamma ],
    pair  => [ 4, 2 ],
} );

$memb_hash_ref = are_members_any( {
    lists => [ \%Alpha, \%Beta, \%Gamma ],
    items => [ qw| abel baker fargo hilton zebra | ],
} );
```

Faster Results with the Unsorted Option

By default, List::Compare::Function functions return lists sorted in Perl's default ASCII-betical mode. Sorting entails a performance cost, and if you do not need a sorted list and do not wish to pay this performance cost, you may call the following List::Compare::Function functions with the 'unsorted' option:

```
@intersection = get_intersection(        '-u',  [ \@Llist, \@Rlist ] );
@union        = get_union(               '-u',  [ \@Llist, \@Rlist ] );
@Lonly        = get_unique(              '-u',  [ \@Llist, \@Rlist ] );
@Ronly        = get_complement(          '-u',  [ \@Llist, \@Rlist ] );
@LorRonly     = get_symmetric_difference('-u',  [ \@Llist, \@Rlist ] );
@bag          = get_bag(                 '-u',  [ \@Llist, \@Rlist ] );
```

For greater readability, the option may be spelled out:

```
@intersection = get_intersection('--unsorted',  [ \@Llist, \@Rlist ] );
```

or

```
@intersection = get_intersection( {
    unsorted => 1,
    lists    => [ \@Llist, \@Rlist ],
} );
```

Should you need a reference to an unsorted list as the return value, you may call the unsorted option as

follows:

```
$intersection_ref = get_intersection_ref(
                        '-u',         [ \@Llist, \@Rlist ] );
$intersection_ref = get_intersection_ref(
                        '--unsorted', [ \@Llist, \@Rlist ] );
```

DISCUSSION

General Comments

List::Compare::Functional is a non-object-oriented implementation of very common Perl code used to determine interesting relationships between two or more lists at a time. List::Compare::Functional is based on the same author's List::Compare module found in the same CPAN distribution. List::Compare::Functional is closely modeled on the ''Accelerated'' mode in List::Compare.

For a discussion of the antecedents of this module, see the discussion of the history and development of this module in the documentation to List::Compare.

List::Compare::Functional's Export Tag Groups

By default, List::Compare::Functional exports no functions. You may import individual functions into your main package but may find it more convenient to import via export tag groups. Four such groups are currently defined:

```
use List::Compare::Functional qw(:main)
use List::Compare::Functional qw(:mainrefs)
use List::Compare::Functional qw(:originals)
use List::Compare::Functional qw(:aliases)
```

- Tag group `:main` includes what, in the author's opinion, are the six List::Compare::Functional subroutines mostly likely to be used:

```
get_intersection()
get_union()
get_unique()
get_complement()
get_symmetric_difference()
is_LsubsetR()
```

- Tag group `:mainrefs` includes five of the six subroutines found in `:main` — all except `is_LsubsetR()` — in the form in which they return references to arrays rather than arrays proper:

```
get_intersection_ref()
get_union_ref()
get_unique_ref()
get_complement_ref()
get_symmetric_difference_ref()
```

- Tag group `:originals` includes all List::Compare::Functional subroutines in their 'original' form, *i.e.*, no aliases for those subroutines:

```
get_intersection
get_intersection_ref
get_union
get_union_ref
get_unique
get_unique_ref
get_unique_all
get_complement
get_complement_ref
get_complement_all
get_symmetric_difference
get_symmetric_difference_ref
```

```
get_shared
get_shared_ref
get_nonintersection
get_nonintersection_ref
is_LsubsetR
is_RsubsetL
is_LequivalentR
is_LdisjointR
is_member_which
is_member_which_ref
are_members_which
is_member_any
are_members_any
print_subset_chart
print_equivalence_chart
get_bag
get_bag_ref
```

- Tag group `:aliases` contains all List::Compare::Functional subroutines which are aliases for subroutines found in tag group `:originals`. These are provided simply for less typing.

```
get_symdiff
get_symdiff_ref
is_LeqvlntR
```

April 2004 Change of Interface

Note: You can skip this section unless you used List::Compare::Functional prior to the release of Version 0.25 in April 2004.

Version 0.25 initiated a significant change in the interface to this module's various functions. In order to be able to accommodate comparisons among more than two lists, it was necessary to change the type of arguments passed to the various functions. Whereas previously a typical List::Compare::Functional function would be called like this:

```
@intersection = get_intersection( \@Llist, \@Rlist ); # SUPERSEDED
```

... now the references to the lists being compared must now be placed within a wrapper array (anonymous or named), a reference to which is now passed to the function, like so:

```
@intersection = get_intersection( [ \@Llist, \@Rlist ] );
```

... or, alternatively:

```
@to_be_compared = (\@Llist, \@Rlist);
@intersection = get_intersection( \@to_be_compared );
```

In a similar manner, List::Compare::Functional functions could previously take arguments in the form of references to 'seen−hashes' instead of references to arrays:

```
@intersection = get_intersection( \%h0, \%h1 );
```

(See above for discussion of seen-hashes.) Now, those references to seen-hashes must be placed within a wrapper array (anonymous or named), a reference to which is passed to the function, like so:

```
@intersection = get_intersection( [ \%h0, \%h1 ] );
```

Also, in a similar manner, some List::Compare::Functional functions previously took arguments in addition to the lists being compared. These arguments were simply passed as scalars, like this:

```
@memb_arr = is_member_which(\@Llist, \@Rlist, 'abel');
```

Now these arguments must also be placed within a wrapper array (anonymous or named), a reference to which is now passed to the function, like so:

```
@memb_arr = is_member_which( [ \@Llist, \@Rlist ], [ 'abel' ] );
```

... or, alternatively:

```
@to_be_compared = (\@Llist, \@Rlist);
@opts = ( 'abel' );
@memb_arr = is_member_which( \@to_be_compared, \@opts );
```

As in previous versions, for a speed boost the user may provide the `'-u'` or `'--unsorted'` option as the *first* argument to some List::Compare::Functional functions. Using this option, the `get_intersection()` function above would appear as:

```
@intersection = get_intersection( '-u', [ \@Llist, \@Rlist ] );
```

... or, alternatively:

```
@intersection = get_intersection( '--unsorted', [ \@Llist, \@Rlist ] );
```

The arguments to *any* List::Compare::Functional function will therefore consist possibly of the unsorted option, and then of either one or two references to arrays, the first of which is a reference to an array of arrays or an array of seen-hashes.

AUTHOR

James E. Keenan (jkeenan@cpan.org). When sending correspondence, please include 'List::Compare::Functional' or 'List−Compare−Functional' in your subject line.