

# N-Dimensional Lists (ndlist)

Version 0.11

Alex Baker

<https://github.com/ambaker1/ndlist>

May 23, 2025

## Abstract

One of the drawbacks of the Tcl programming language is its lack of data structures. This package is an attempt to fill that gap, to provide the basic data structures that every programmer is familiar with: vectors, matrices, tables, and higher-dimensional arrays.

This package is also a [Tin](#) package, and can be loaded in as shown below:

### Example 1: Installing and loading “ndlist”

*Code:*

```
package require tin 2.1
tin autoadd ndlist https://github.com/ambaker1/ndlist install.tcl
tin import ndlist
```

---

# 1-Dimensional Lists (Vectors)

Lists are foundational to Tcl, so in addition to providing utilities for ND-lists, this package also provides utilities for working with 1D-lists, or vectors.

## *Range Generator*

The command *range* simply generates a list of integer values. This can be used in conjunction with the Tcl *foreach* loop to simplify writing “for” loops. There are two ways of calling this command, as shown below.

```
range $n
range $start $stop <$step>
```

<b>\$n</b>	Number of indices, starting at 0 (e.g. 3 returns 0 1 2).
<b>\$start</b>	Starting value.
<b>\$stop</b>	Stop value.
<b>\$step</b>	Step size. Default 1 or -1, depending on direction of start to stop.

### Example 2: Integer range generation

*Code:*

```
puts [range 3]
puts [range 0 2]
puts [range 10 3 -2]
```

*Output:*

```
0 1 2
0 1 2
10 8 6 4
```

### Example 3: Simpler for-loop

*Code:*

```
foreach i [range 3] {
    puts $i
}
```

*Output:*

```
0
1
2
```

## Logical Indexing

The command *find* returns the indices of non-zero elements of a boolean list, or indices of elements that satisfy a given criterion. Can be used in conjunction with *nget* to perform logical indexing.

```
find $list <$op $scalar>
```

<b>\$list</b>	List of values to compare.
<b>\$op</b>	Comparison operator. Default “!=”.
<b>\$scalar</b>	Comparison value. Default 0.

### Example 4: Filtering a list

Code:

```
set x {0.5 2.3 4.0 2.5 1.6 2.0 1.4 5.6}
puts [nget $x [find $x > 2]]
```

Output:

```
2.3 4.0 2.5 5.6
```

## Linear Interpolation

The command *linterp* performs linear 1D interpolation. Converts input to double.

```
linterp $x $xList $yList
```

<b>\$x</b>	Value to query in <b>\$xList</b>
<b>\$xList</b>	List of x points, strictly increasing
<b>\$yList</b>	List of y points, same length as <b>\$xList</b>

### Example 5: Linear interpolation

Code:

```
puts [linterp 2 {1 2 3} {4 5 6}]
puts [linterp 8.2 {0 10 20} {2 -4 5}]
```

Output:

```
5.0
-2.92
```

## Vector Generation

The command *linspace* can be used to generate a vector of specified length and equal spacing between two specified values. Converts input to double.

```
linspace $n $start $stop
```

<b><code>\$n</code></b>	Number of points
<b><code>\$start</code></b>	Starting value
<b><code>\$stop</code></b>	End value

### Example 6: Linearly spaced vector generation

*Code:*

```
puts [linspace 5 0 1]
```

*Output:*

```
0.0 0.25 0.5 0.75 1.0
```

The command *linspace* generates intermediate values given an increment size and a sequence of targets. Converts input to double.

```
linspace $step $x1 $x2 ...
```

<b><code>\$step</code></b>	Maximum step size
<b><code>\$x1 \$x2 ...</code></b>	Targets to hit.

### Example 7: Intermediate value vector generation

*Code:*

```
puts [linspace 0.25 0 1 0]
```

*Output:*

```
0.0 0.25 0.5 0.75 1.0 0.75 0.5 0.25 0.0
```

## Functional Mapping

The command *lapply* simply applies a command over each element of a list, and returns the result. The command *lapply2* maps element-wise over two equal length lists.

```
lapply $command $list $arg ...
```

```
lapply2 $command $list1 $list2 $arg ...
```

<code>\$list</code>	List to map over.
<code>\$list1 \$list2</code>	Lists to map over, element-wise.
<code>\$command</code>	Command prefix to map with.
<code>\$arg ...</code>	Additional arguments to append to command after list elements.

### Example 8: Applying a math function to a list

*Code:*

```
# Add Tcl math functions to the current namespace path
namespace path [concat [namespace path] ::tcl::mathfunc]
puts [lapply abs {-5 1 2 -2}]
```

*Output:*

```
5 1 2 2
```

### Example 9: Mapping over two lists

*Code:*

```
lapply puts [lapply2 {format "%s %s"} {hello goodbye} {world moon}]
```

*Output:*

```
hello world
goodbye moon
```

## List Statistics

The commands *max*, *min*, *sum*, *product*, *mean*, *median*, *stdev* and *pstdev* compute the maximum, minimum, sum, product, mean, median, sample and population standard deviation of values in a list. For more advanced statistics, check out the Tcllib `math::statistics` package.

```
max $list
```

```
min $list
```

```
sum $list
```

```
product $list
```

```
mean $list
```

```
median $list
```

```
stdev $list
```

```
pstdev $list
```

`$list`                      List to compute statistic of.

### Example 10: List Statistics

*Code:*

```
set list {-5 3 4 0}
foreach stat {max min sum product mean median stdev pstdev} {
    puts [list $stat [$stat $list]]
}
```

*Output:*

```
max 4
min -5
sum 2
product 0
mean 0.5
median 1.5
stdev 4.041451884327381
pstdev 3.5
```

## Vector Algebra

The dot product of two equal length vectors can be computed with *dot*. The cross product of two vectors of length 3 can be computed with *cross*.

```
dot $a $b
```

```
cross $a $b
```

`$a` First vector.

`$b` Second vector.

The norm, or magnitude, of a vector can be computed with *norm*.

```
norm $a <$p>
```

`$a` Vector to compute norm of.

`$p` Norm type. 1 is sum of absolute values, 2 is euclidean distance, and Inf is absolute maximum value. Default 2.

### Example 11: Dot and cross product

*Code:*

```
set x {1 2 3}
set y {-2 -4 6}
puts [dot $x $y]
puts [cross $x $y]
```

*Output:*

```
8
24 -12 0
```

For more advanced vector algebra routines, check out the Tcllib `math::linearalgebra` package.

---

## 2-Dimensional Lists (Matrices)

A matrix is a two-dimensional list, or a list of row vectors. This is consistent with the format used in the Tcllib math::linearalgebra package. See the example below for how matrices are interpreted.

$$A = \begin{bmatrix} 2 & 5 & 1 & 3 \\ 4 & 1 & 7 & 9 \\ 6 & 8 & 3 & 2 \\ 7 & 8 & 1 & 4 \end{bmatrix}, \quad B = \begin{bmatrix} 9 \\ 3 \\ 0 \\ -3 \end{bmatrix}, \quad C = \begin{bmatrix} 3 & 7 & -5 & -2 \end{bmatrix}$$

### Example 12: Matrices and vectors

*Code:*

```
# Define matrices, column vectors, and row vectors
set A {{2 5 1 3} {4 1 7 9} {6 8 3 2} {7 8 1 4}}
set B {9 3 0 -3}
set C {{3 7 -5 -2}}
# Print out matrices (join with newline to print out each row)
puts "A ="
puts [join $A \n]
puts "B ="
puts [join $B \n]
puts "C ="
puts [join $C \n]
```

*Output:*

```
A =
2 5 1 3
4 1 7 9
6 8 3 2
7 8 1 4
B =
9
3
0
-3
C =
3 7 -5 -2
```



## Generating Matrices

The commands *zeros*, *ones*, and *eye* generate common matrices.

```
zeros $n $m
```

```
ones $n $m
```

`$n`                      Number of rows

`$m`                      Number of columns

The command *eye* generates an identity matrix of a specified size.

```
eye $n
```

`$n`                      Size of identity matrix

### Example 13: Generating standard matrices

*Code:*

```
puts [zeros 2 3]
puts [ones 3 2]
puts [eye 3]
```

*Output:*

```
{0 0 0} {0 0 0}
{1 1} {1 1} {1 1}
{1 0 0} {0 1 0} {0 0 1}
```

## Combining Matrices

The commands *stack* and *augment* can be used to combine matrices, row or column-wise.

```
stack $mat1 $mat2 ...
```

```
augment $mat1 $mat2 ...
```

`$mat1 $mat2 ...`                      Arbitrary number of matrices to stack/augment (number of columns/rows must match)

The command *block* combines a matrix of matrices into a block matrix.

```
block $matrices
```

`$matrices`                      Matrix of matrices.

### Example 14: Combining matrices

*Code:*

```
set A [stack {{1 2}} {{3 4}}]
set B [augment {1 2} {3 4}]
set C [block [list [list $A $B] [list $B $A]]]
puts $A
puts $B
puts [join $C \n]; # prints each row on a new line
```

*Output:*

```
{1 2} {3 4}
{1 3} {2 4}
1 2 1 3
3 4 2 4
1 3 1 2
2 4 3 4
```

## Matrix Transpose

The command *transpose* simply swaps the rows and columns of a matrix.

```
transpose $A
```

**\$A**                      Matrix to transpose, nxm.

Returns an mxn matrix.

### Example 15: Transposing a matrix

*Code:*

```
puts [transpose {{1 2} {3 4}}]
```

*Output:*

```
{1 3} {2 4}
```

## Matrix Multiplication

The command *matmul* performs matrix multiplication for two matrices. Inner dimensions must match.

```
matmul $A $B
```

**\$A**                      Left matrix, nxq.

**\$B**                      Right matrix, qxm.

Returns an nxm matrix (or the corresponding dimensions from additional matrices)

### Example 16: Multiplying a matrix

*Code:*

```
puts [matmul {{2 5 1 3} {4 1 7 9} {6 8 3 2} {7 8 1 4}} {9 3 0 -3}]
```

*Output:*

```
24 12 72 75
```

## Miscellaneous Linear Algebra Routines

The command *outerprod* takes the outer product of two vectors,  $\mathbf{a} \otimes \mathbf{b} = \mathbf{a}\mathbf{b}^T$ .

```
outerprod $a $b
```

**\$a \$b**                      Vectors with lengths n and m. Returns a matrix, shape nxm.

The command *kronprod* takes the Kronecker product of two matrices, as shown in Eq. (1).

```
kronprod $A $B
```

**\$A \$B**                      Matrices, shapes nxm and pxq. Returns a matrix, shape (np)x(mq).

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & \dots & a_{1n}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{n1}\mathbf{B} & \dots & a_{nn}\mathbf{B} \end{bmatrix} \quad (1)$$

### Example 17: Outer product and Kronecker product

*Code:*

```
set A [eye 3]
set B [outerprod {1 2} {3 4}]
set C [kronprod $A $B]
puts [join $C \n]; # prints out each row on a new line
```

*Output:*

```
3 4 0 0 0 0
6 8 0 0 0 0
0 0 3 4 0 0
0 0 6 8 0 0
0 0 0 0 3 4
0 0 0 0 6 8
```

For more advanced matrix algebra routines, check out the Tcllib `math::linearalgebra` package.

## Iteration Tools

The commands *zip* zips two lists into a list of tuples, and *zip3* zip three lists into a list of triples. Lists must be the same length.

```
zip $a $b
```

```
zip3 $a $b $c
```

`$a $b $c`

Lists to zip together.

### Example 18: Zipping and unzipping lists

Code:

```
# Zipping
set x [zip {A B C} {1 2 3}]
set y [zip3 {Do Re Mi} {A B C} {1 2 3}]
puts $x
puts $y
# Unzipping (using transpose)
puts [transpose $x]
```

Output:

```
{A 1} {B 2} {C 3}
{Do A 1} {Re B 2} {Mi C 3}
{A B C} {1 2 3}
```

The command *cartprod* computes the Cartesian product of an arbitrary number of vectors, returning a matrix where the columns correspond to the input vectors and the rows correspond to all the combinations of the vector elements.

```
cartprod $a $b ...
```

`$a $b ...`

Arbitrary number of vectors to take Cartesian product of.

### Example 19: Cartesian product

Code:

```
puts [cartprod {A B C} {1 2 3}]
```

Output:

```
{A 1} {A 2} {A 3} {B 1} {B 2} {B 3} {C 1} {C 2} {C 3}
```

---

## N-Dimensional Lists (Tensors)

A ND-list is defined as a list of equal length (N-1)D-lists, which are defined as equal length (N-2)D-lists, and so on until (N-N)D-lists, which are scalars of arbitrary size. This definition is flexible, and allows for different interpretations of the same data. For example, the list “1 2 3” can be interpreted as a scalar with value “1 2 3”, a vector with values “1”, “2”, and “3”, or a matrix with row vectors “1”, “2”, and “3”. In general, if a value is a valid for N dimensions, it will also be valid for dimensions 0 to N-1.

The command *ndims* returns the rank of an ND-list, and the command *ndims\_multiple* returns the rank that is compatible with multiple ND-lists. By default, it automatically determines the rank for the data, but if a rank is provided, it will validate that the ND-list or ND-lists are compatible with the provided rank.

```
ndims $ndlist <$rank>
```

```
ndims_multiple $ndlists <$rank>
```

<b>\$ndlist</b>	ND-list.
<b>\$ndlists</b>	List of ND-lists.
<b>\$rank</b>	Rank of ND-list (e.g. 2 for matrix) or “auto” for auto-rank. Default “auto”.

### Example 20: Rank of an ND-list

*Code:*

```
set x {1}
set y {1 2 {hello world}}; # note that this is not a valid 2D list
set z {{1 2 3} {4 5 6}}
puts [ndims $x]; # 0
puts [ndims $y]; # 1
puts [ndims $z]; # 2
puts [ndims_multiple [list $x $y $z]]; # 1
```

*Output:*

```
0
1
2
1
```

Note: All Tcl values are ND-lists, because a 0D-list is just a string that can’t be parsed as a Tcl list, and in Tcl, everything is a string.

## Shape and Size

The commands *nshape* and *nsiz*e return the shape and size of an ND-list, respectively. The shape is a list of the dimensions, and the size is the product of the shape.

```
nshape $ndlist <$rank>
```

```
nsiz
```

e \$ndlist <\$rank>

**\$ndlist** ND-list to get shape/size of.

**\$rank** Rank of ND-list (e.g. 2 for matrix) or “auto” for auto-rank. Default “auto”.

### Example 21: Getting shape and size of an ND-list

*Code:*

```
set x {{1 2 3} {4 5 6}}
puts [nshape $x]
puts [nsiz
```

e \$x]

*Output:*

```
2 3
6
```

## Initialization

The command *nfull* initializes a valid ND-list of any size filled with a single value.

```
nfull $value $n ...
```

<code>\$value</code>	Value to repeat
<code>\$n ...</code>	Shape (list of dimensions) of ND-list.

### Example 22: Generate ND-list filled with one value

*Code:*

```
puts [nfull foo 3 2]; # 3x2 matrix filled with "foo"  
puts [nfull 0 2 2 2]; # 2x2x2 tensor filled with zeros
```

*Output:*

```
{foo foo} {foo foo} {foo foo}  
{0 0} {0 0} {0 0} {0 0}
```

The command *nrand* initializes a valid ND-list of any size filled with random values between 0 and 1.

```
nrand $n ...
```

<code>\$n ...</code>	Shape (list of dimensions) of ND-list.
----------------------	--

### Example 23: Generate random matrix

*Code:*

```
expr {srand(0)}; # resets the random number seed (for the example)  
puts [nrand 1 2]; # 1x2 matrix filled with random numbers
```

*Output:*

```
{0.013469574513598146 0.3831388500440581}
```



## Repeating and Expanding

The command *nrepeat* repeats portions of an ND-list a specified number of times.

```
nrepeat $ndlist $n ...
```

<b>\$value</b>	Value to repeat
<b>\$n ...</b>	Repetitions at each level.

### Example 24: Repeat elements of a matrix

*Code:*

```
puts [nrepeat {{1 2} {3 4}} 1 2]
```

*Output:*

```
{1 2 1 2} {3 4 3 4}
```

The command *nexpand* repeats portions of an ND-list to expand to new dimensions. New dimensions must be divisible by old dimensions. For example, 1x1, 2x1, 4x1, 1x3, 2x3 and 4x3 are compatible with 4x3.

```
nexpand $ndlist $n ...
```

<b>\$ndlist</b>	ND-list to expand.
<b>\$n ...</b>	New shape of ND-list. If -1 is used, it keeps that axis the same.

### Example 25: Expand an ND-list to new dimensions

*Code:*

```
puts [nexpand {1 2 3} -1 2]  
puts [nexpand {{1 2}} 2 4]
```

*Output:*

```
{1 1} {2 2} {3 3}  
{1 2 1 2} {1 2 1 2}
```

## Padding and Extending

The command *npad* pads an ND-list along its axes by a specified number of elements.

```
npad $ndlist $value $n ...
```

<code>\$ndlist</code>	ND-list to pad.
<code>\$value</code>	Value to pad with.
<code>\$n ...</code>	Number of elements to pad.

### Example 26: Padding an ND-list with zeros

Code:

```
set a {{1 2 3} {4 5 6} {7 8 9}}
puts [npad $a 0 2 1]
```

Output:

```
{1 2 3 0} {4 5 6 0} {7 8 9 0} {0 0 0 0} {0 0 0 0}
```

The command *nextend* extends an ND-list to a new shape by padding.

```
nextend $ndlist $value $n ...
```

<code>\$ndlist</code>	ND-list to extend.
<code>\$value</code>	Value to pad with.
<code>\$n ...</code>	New shape of ND-list.

### Example 27: Extending an ND-list to a new shape with a filler value

Code:

```
set a {hello hi hey howdy}
puts [nextend $a world -1 2]
```

Output:

```
{hello world} {hi world} {hey world} {howdy world}
```

## Flattening and Reshaping

The command *nflatten* flattens an ND-list to a vector.

```
nflatten $ndlist <$rank>
```

**\$ndlist** ND-list to flatten.

**\$rank** Rank of ND-list (e.g. 2 for matrix) or “auto” for auto-rank. Default “auto”.

### Example 28: Reshape a matrix to a 3D tensor

*Code:*

```
set x [nflatten {{1 2 3 4} {5 6 7 8}}]
puts [nreshape $x 2 2 2]
```

*Output:*

```
{{1 2} {3 4}} {{5 6} {7 8}}
```

The command *nreshape* reshapes a vector into specified dimensions. Sizes must be compatible.

```
nreshape $vector $n ...
```

**\$vector** Vector (1D-list) to reshape.

**\$n ...** Shape (list of dimensions) of ND-list. One axis may be dynamic, denoted with a "\*".

### Example 29: Reshape a vector to a matrix with three columns

*Code:*

```
puts [nreshape {1 2 3 4 5 6} * 3]
```

*Output:*

```
{1 2 3} {4 5 6}
```

## Index Notation

This package provides generalized N-dimensional list access/modification commands, using an index notation parsed by the command `::ndlist::ParseIndex`, which returns the index type and an index list for the type.

```
::ndlist::ParseIndex $n $input
```

<b>\$n</b>	Number of elements in list.
<b>\$input</b>	Index input. Options are shown below:
<b>:</b>	All indices
<b>\$start:\$stop</b>	Range of indices (e.g. 0:4 or 1:end-2).
<b>\$start:\$step:\$stop</b>	Stepped range of indices (e.g. 0:2:-2 or 2:3:end).
<b>\$iList</b>	List of indices (e.g. {0 end-1 5} or 3).
<b>\$i*</b>	Single index with a asterisk, “flattens” the ndlist (e.g. 0* or end-3*).

Additionally, indices get passed through the `::ndlist::Index2Integer` command, which converts the inputs “end”, “end-integer”, “integer±integer” and negative wrap-around indexing (where -1 is equivalent to “end”) into normal integer indices. Note that this command will return an error if the index is out of range.

```
::ndlist::Index2Integer $n $index
```

<b>\$n</b>	Number of elements in list.
<b>\$index</b>	Single index.

### Example 30: Index Notation

*Code:*

```
set n 10
puts [::ndlist::ParseIndex $n :]
puts [::ndlist::ParseIndex $n 1:8]
puts [::ndlist::ParseIndex $n 0:2:6]
puts [::ndlist::ParseIndex $n {0 5 end-1}]
puts [::ndlist::ParseIndex $n end*]
```

*Output:*

```
A {}
R {1 8}
L {0 2 4 6}
L {0 5 8}
S 9
```

## Access

Portions of an ND-list can be accessed with the command *nget*, using the index parser *::ndlist::ParseIndex* for each dimension being indexed. Note that unlike the Tcl *lindex* and *lrange* commands, *nget* will return an error if the indices are out of range.

```
nget $ndlist $i ...
```

*\$ndlist*                      ND-list value.

*\$i ...*                      Index inputs, parsed with *::ndlist::ParseIndex*.

### Example 31: ND-list access

*Code:*

```
set A {{1 2 3} {4 5 6} {7 8 9}}
puts [nget $A 0 :]; # get row matrix
puts [nget $A 0* :]; # flatten row matrix to a vector
puts [nget $A 0:1 0:1]; # get matrix subset
puts [nget $A end:0 end:0]; # can have reverse ranges
puts [nget $A {0 0 0} 1*]; # can repeat indices
```

*Output:*

```
{1 2 3}
1 2 3
{1 2} {4 5}
{9 8 7} {6 5 4} {3 2 1}
2 2 2
```

## Modification

A ND-list can be modified by reference with *nset*, and by value with *nreplace*, using the index parser *::ndlist::ParseIndex* for each dimension being indexed. Note that unlike the Tcl *lset* and *lreplace* commands, the commands *nset* and *nreplace* will return an error if the indices are out of range. If all the index inputs are “:” except for one, and the replacement list is blank, it will delete values along that axis by calling *nremove*. Otherwise, the replacement ND-list must be expandable to the target index dimensions.

```
nset $varName $i ... $sublist
```

```
nreplace $ndlist $i ... $sublist
```

<b>\$varName</b>	Variable that contains an ND-list.
<b>\$ndlist</b>	ND-list to modify.
<b>\$i ...</b>	Index inputs, parsed with <i>::ndlist::ParseIndex</i> .
<b>\$sublist</b>	Replacement list, or blank to delete values.

### Example 32: Replace range with a single value

Code:

```
puts [nreplace [range 10] 0:2:end 0]
```

Output:

```
0 1 0 3 0 5 0 7 0 9
```

### Example 33: Swapping matrix rows

Code:

```
set a {{1 2 3} {4 5 6} {7 8 9}}
nset a {1 0} : [nget $a {0 1} :]; # Swap rows and columns (modify by reference)
puts $a
```

Output:

```
{4 5 6} {1 2 3} {7 8 9}
```

## Removal

The command *nremove* removes portions of an ND-list at a specified axis.

```
nremove $ndlist $i <$axis>
```

<code>\$ndlist</code>	ND-list to modify.
<code>\$i</code>	Index input, parsed with <code>::ndlist::ParseIndex</code> .
<code>\$axis</code>	Axis to remove at. Default 0.

### Example 34: Filtering a list by removing elements

*Code:*

```
set x [range 10]
puts [nremove $x [find $x > 4]]
```

*Output:*

```
0 1 2 3 4
```

### Example 35: Deleting a column from a matrix

*Code:*

```
set a {{1 2 3} {4 5 6} {7 8 9}}
puts [nremove $a 2 1]
```

*Output:*

```
{1 2} {4 5} {7 8}
```

## Insertion and Concatenation

The command *ninsert* inserts an ND-list into another ND-list at a specified index and axis. The ND-lists must agree in dimension at all other axes. If “end” or “end-integer” is used for the index, it will insert after the index. Otherwise, it will insert before the index. The command *ncat* is shorthand for inserting at “end”, and concatenates two ND-lists.

```
ninsert $ndlist1 $index $ndlist2 <$axis> <$rank>
```

```
ncat $ndlist1 $ndlist2 <$axis> <$rank>
```

<b>\$ndlist1</b> <b>\$ndlist2</b>	ND-lists to combine.
<b>\$index</b>	Index to insert at.
<b>\$axis</b>	Axis to insert/concatenate at (default 0).
<b>\$rank</b>	Rank of ND-list (e.g. 2 for matrix) or “auto” for auto-rank. Default “auto”.

### Example 36: Inserting a column into a matrix

*Code:*

```
set matrix {{1 2} {3 4} {5 6}}  
set column {A B C}  
puts [ninsert $matrix 1 $column 1 2]
```

*Output:*

```
{1 A 2} {3 B 4} {5 C 6}
```

### Example 37: Concatenate tensors

*Code:*

```
set x [nreshape {1 2 3 4 5 6 7 8 9} 3 3 1]  
set y [nreshape {A B C D E F G H I} 3 3 1]  
puts [ncat $x $y 2 3]
```

*Output:*

```
{{1 A} {2 B} {3 C}} {{4 D} {5 E} {6 F}} {{7 G} {8 H} {9 I}}
```



## Changing Order of Axes

The command *nswapaxes* is a general purpose transposing function that swaps the axes of an ND-list. For simple matrix transposing, the command *transpose* can be used instead.

```
nswapaxes $ndlist $axis1 $axis2
```

**\$ndlist** ND-list to manipulate.

**\$axis1 \$axis2** Axes to swap.

The command *nmoveaxis* moves a specified source axis to a target position. For example, moving axis 0 to position 2 would change “i,j,k” to “j,k,i”.

```
nmoveaxis $ndlist $source $target
```

**\$ndlist** ND-list to manipulate.

**\$source** Source axis.

**\$target** Target position.

The command *npermute* is more general purpose, and defines a new order for the axes of an ND-list. For example, the axis list “1 0 2” would change “i,j,k” to “j,i,k”.

```
npermute $ndlist $axis ...
```

**\$ndlist** ND-list to manipulate.

**\$axis ...** List of axes defining new order.

### Example 38: Changing tensor axes

*Code:*

```
set x {{{1 2} {3 4}} {{5 6} {7 8}}}  
set y [nswapaxes $x 0 2]  
set z [nmoveaxis $x 0 2]  
puts [lindex $x 0 0 1]  
puts [lindex $y 1 0 0]  
puts [lindex $z 0 1 0]
```

*Output:*

```
2  
2  
2
```

## ND Functional Mapping

The command *napply* applies a command over each element of an ND-list, and returns the result. The commands *napply2* maps element-wise over two ND-lists. If the input lists have different shapes, they will be expanded to their maximum dimensions with *nexpand* (if compatible).

```
napply $command $ndlist <$suffix> <$rank>
```

```
napply2 $command $ndlist1 $ndlist2 <$suffix> <$rank>
```

<b>\$ndlist</b>	ND-list to map over.
<b>\$ndlist1 \$ndlist2</b>	ND-lists to map over, element-wise.
<b>\$command</b>	Command prefix to map with.
<b>\$suffix</b>	Additional arguments to append to command after ND-list elements. Default blank.
<b>\$rank</b>	Rank of ND-list (e.g. 2 for matrix) or “auto” for auto-rank. Default “auto”.

### Example 39: Chained functional mapping over a matrix

Code:

```
napply puts [napply {format %.2f} [napply expr {{1 2} {3 4}} {+ 1}]]
```

Output:

```
2.00
3.00
4.00
5.00
```

### Example 40: Format columns of a matrix

Code:

```
set data {{1 2 3} {4 5 6} {7 8 9}}
set formats {{%.1f %.2f %.3f}}
puts [napply2 format $formats $data]
```

Output:

```
{1.0 2.00 3.000} {4.0 5.00 6.000} {7.0 8.00 9.000}
```

## Reducing an ND-list

The command *nreduce* combines *nmoveaxis* and *napply* to reduce an axis of an ND-list with a function that reduces a vector to a scalar, like *max* or *sum*.

```
nreduce $command $ndlist <$axis> <$suffix> <$rank>
```

<b>\$command</b>	Command prefix to map with.
<b>\$ndlist</b>	ND-list to map over.
<b>\$axis</b>	Axis to reduce. Default 0.
<b>\$suffix</b>	Additional arguments to append to command after ND-list elements. Default blank.
<b>\$rank</b>	Rank of ND-list (e.g. 2 for matrix) or “auto” for auto-rank. Default “auto”.

### Example 41: Matrix row and column statistics

*Code:*

```
set x {{1 2} {3 4} {5 6} {7 8}}
puts [nreduce max $x]; # max of each column
puts [nreduce max $x 1]; # max of each row
puts [nreduce sum $x]; # sum of each column
puts [nreduce sum $x 1]; # sum of each row
```

*Output:*

```
7 8
2 4 6 8
16 20
3 7 11 15
```

## Generalized N-Dimensional Mapping

The command *nmap* is a general purpose mapping function for N-dimensional lists in Tcl. If multiple ND-lists are provided for iteration, they must be expandable to their maximum dimensions. The actual implementation flattens all the ND-lists and calls the Tcl *lmap* command, and then reshapes the result to the target dimensions. So, if “continue” or “break” are used in the map body, it will return an error.

```
nmap <$rank> $varName $ndlist <$varName $ndlist ...> $body
```

<b>\$rank</b>	Rank of ND-list (e.g. 2 for matrix) or “auto” for auto-rank. Default “auto”.
<b>\$varName</b>	Variable name to iterate with.
<b>\$ndlist</b>	ND-list to iterate over.
<b>\$body</b>	Tcl script to evaluate at every loop iteration.

### Example 42: Expand and map over matrices

*Code:*

```
set phrases [nmap 2 greeting {{hello goodbye}} subject {world moon} {  
    list $greeting $subject  
}]  
napply puts $phrases {} 2
```

*Output:*

```
hello world  
goodbye world  
hello moon  
goodbye moon
```

## Loop Index Access

The iteration indices of *nmap* can be accessed with the commands *i*, *j*, and *k*. The commands *j* and *k* are simply shorthand for *i* with axes 1 and 2.

**i** <\$axis>

**j**

**k**

**\$axis**                      Dimension to access mapping index at. Default 0.  
If -1, returns the linear index of the loop.

### Example 43: Finding index tuples that match criteria

*Code:*

```
set x {{1 2 3} {4 5 6} {7 8 9}}
set indices {}
nmap xi $x {
  if {${xi} > 4} {
    lappend indices [list [i] [j]]
  }
}
puts $indices
```

*Output:*

```
{1 1} {1 2} {2 0} {2 1} {2 2}
```

---

## ND-Arrays

The command *narray* is a TclOO class based on the superclass *::ndlist::ValueContainer*. It is an object-oriented approach to array manipulation and processing.

```
narray new $varName <$value> <$rank>
```

```
narray create $name $varName <$value> <$rank>
```

<b>\$varName</b>	Variable to store object name for access and garbage collection. Variable names are restricted to word characters and namespace delimiters only.
<b>\$value</b>	ND-list value to store in array. Default blank.
<b>\$rank</b>	Number of dimensions of array. Default “auto”.
<b>\$name</b>	Name of object if using “create” method.

### Value Access

The value stored in an *narray* object can be accessed by calling the object by itself with no argument, or by calling the method “value”.

```
$narrayObj <value>
```

<b>value</b>	Optional method for returning value.
--------------	--------------------------------------

#### Example 44: Creating an “narray” object and accessing the value

*Code:*

```
narray new x {hello world}  
puts [$x]
```

*Output:*

```
hello world
```

## Number of Dimensions

The number of dimensions can be manually set when creating the narray object, but by default it is set to “auto”. For numerical data that is properly formatted as an ND-list, it will always choose the minimum number of dimensions that can store the data. For data containing strings with spaces, it is advised to manually set the dimension. To query or change the number of dimensions, use the method *ndims*.

```
$narrayObj ndims <$rank | auto | type>
```

<b>\$rank</b>	Value to set as number of dimensions of array. Default blank to query.
<b>auto</b>	Toggle auto-dimensions. Returns auto-determined number of dimensions.
<b>type</b>	Query internal dimension type. Returns “auto” or “user”.

## Shape and Size

The shape and size corresponding with the number of dimensions of the narray are accessed with the methods *shape* and *size*.

```
$narrayObj shape
```

```
$narrayObj size
```

### Example 45: N-array dimension, shape and size

*Code:*

```
narray new x {{1 2 3} {4 5 6}}  
puts [list [$x ndims] [$x ndims type] [$x shape] [$x size]]  
$x ndims 3; # change number of dimensions  
puts [list [$x ndims] [$x ndims type] [$x shape] [$x size]]
```

*Output:*

```
2 auto {2 3} 6  
3 user {2 3 1} 6
```

## Indexing

The “@” operator uses *nget* to access a portion of the ND-array.

```
$narrayObj @ $i ...
```

**\$i ...**

Index inputs corresponding with rank of ND-array.

### Example 46: Accessing portions of an ND-array

*Code:*

```
narray new x {{1 2 3} {4 5 6} {7 8 9}}  
puts [$x @ 0 2]  
puts [$x @ 0:end-1 {0 2}]
```

*Output:*

```
3  
{1 3} {4 6}
```

## Copying

The operator “-->” copies the ND-array to a new variable, and returns the new object. If indices are specified, the new ND-array object will have the rank of the indexed range.

```
$narrayObj <@ $i ...> --> $varName
```

**\$i ...**

Indices to access. Default all.

**\$varName**

Variable to store object name for access and garbage collection. Variable names are restricted to word characters and namespace delimiters only.

### Example 47: Copying a portion of an ND-array

*Code:*

```
narray new x {{1 2 3} {4 5 6}}  
$x @ 0* : --> y; # Row vector (flattened to 1D)  
puts "[$y ndims], [$y]"
```

*Output:*

```
1, 1 2 3
```



## Evaluation/Mapping

The command *neval* maps over ND-arrays using *nmap*. The command *nexpr* is a special case that passes input through the Tcl *expr* command. ND-arrays can be referred to with “@ref”, where “ref” is the name of the ND-array variable. Portions of an ND-array can be mapped over with the notation “@ref(\$i,...)”. The highest-rank input will determine the output, and all input arrays must have compatible dimensions.

```
neval $body <$self> <$rankVar>
```

```
nexpr $expr <$self> <$rankVar>
```

<b>\$body</b>	Script to evaluate, with “@ref” notation for object references.
<b>\$expr</b>	Expression to evaluate, with “@ref” notation for object references.
<b>\$self</b>	Object to refer to with “@.”. Default blank.
<b>\$rankVar</b>	Variable to store resulting rank in. Default blank.

### Example 48: Get distance between elements in a vector

*Code:*

```
narray new x {1 2 4 7 11 16}  
puts [nexpr {@x(1:end) - @x(0:end-1)}]
```

*Output:*

```
1 2 3 4 5
```

### Example 49: Outer product of two vectors

*Code:*

```
narray new x {1 2 3}  
narray new y {{4 5 6}}  
puts [nexpr {@x * @y}]
```

*Output:*

```
{4 5 6} {8 10 12} {12 15 18}
```

## Modification

The assignment operator, “=”, sets the value of the entire ND-array, or of a specified range. The math assignment operator, “:=”, sets the value, passing the input through the *nexpr* command. Both assignment operators return the object.

```
$narrayObj <@ $i ...> = $value
```

```
$narrayObj <@ $i ...> := $expr
```

<b>\$i ...</b>	Indices to modify. Default all.
<b>\$value</b>	Value to assign. Blank to remove values.
<b>\$expr</b>	Expression to evaluate.

If using the math assignment operator, the ND-array or indexed range can be accessed with the alias “\$.”, and the elements of the array or indexed range can be accessed with “@.”.

```
$ . $arg ...
```

<b>\$arg ...</b>	Method arguments for object.
------------------	------------------------------

### Example 50: Element-wise modification of a vector

*Code:*

```
# Create blank vectors and assign values
[narray new x] = {1 2 3}
[narray new y] = {10 20 30}
# Add one to each element
puts [[ $x := { @. + 1 } ]]
# Double the last element
puts [[ $x @ end := { @. * 2 } ]]
# Element-wise addition of vectors
puts [[ $x := { @. + @y } ]]
```

*Output:*

```
2 3 4
2 3 8
12 23 38
```

## Removal/Insertion

The method *remove* removes portions of an ND-array along a specified axis with the command *nremove*, and the method *insert* inserts values into an ND-array at a specified index/axis with the command *ninsert*. Both methods modify the object and return the object.

```
$narrayObj remove $i <$axis>
```

```
$narrayObj insert $i $sublist <$axis>
```

**\$i** Indices to remove/insert at.

**\$sublist** Value to insert.

**\$axis** Axis to remove/insert at (default 0).

### Example 51: Removing elements from a vector

*Code:*

```
narray new vector {1 2 3 4 5 6 7 8}  
# Remove all odd numbers  
$vector remove [find [nexpr {@vector % 2}]]  
puts [$vector]
```

*Output:*

```
2 4 6 8
```

### Example 52: Inserting a column into a matrix

*Code:*

```
narray new matrix {{1 2} {3 4} {5 6}}  
$matrix insert 1 {A B C} 1  
puts [$matrix]
```

*Output:*

```
{1 A 2} {3 B 4} {5 C 6}
```

## Temporary Object Evaluation

The pipe operator, “|”, copies the ND-array to a temporary object, and evaluates the method. Returns the result of the method, or the value of the temporary object. This operator is useful for converting methods that modify the object to methods that return a modified value.

```
$narrayObj <@ $i ...> | $method $arg ...
```

<b>\$i ...</b>	Indices to access. Default all.
<b>\$method</b>	Method to evaluate.
<b>\$arg ...</b>	Arguments to pass to method.

### Example 53: Temporary object value

*Code:*

```
# Create a matrix
narray new x {{1 2 3} {4 5 6}}
# Print value with first row doubled.
puts [$x | @ 0* : := {@. * 2}]
# Source object was not modified
puts [$x]
```

*Output:*

```
{2 4 6} {4 5 6}
{1 2 3} {4 5 6}
```

## Reference Variable Evaluation

The ampersand operator “&” copies the ND-array value or range to a reference variable, and evaluates a body of script. The changes made to the reference variable will be applied to the object, and if the variable is unset, the object will be deleted. If no indices are specified and the variable is unset in the script, the ND-array object will be destroyed. Returns the result of the script.

```
$narrayObj <@ $i ...> & $refName $body
```

<b>\$i ...</b>	Indices to access. Default all.
<b>\$refName</b>	Variable name to use for reference.
<b>\$body</b>	Body to evaluate.

### Example 54: Appending a vector

*Code:*

```
# Create a 1D list
narray new x {1 2 3}
# Append the list
$x & ref {lappend ref 4 5 6}
puts [$x]
# Append a subset of the list
$x @ end* & ref {lappend ref 7 8 9}
puts [$x]
```

*Output:*

```
1 2 3 4 5 6
1 2 3 4 5 {6 7 8 9}
```

---

## Tabular Data Structure

This package also provides support for a tabular datatype. The string representation of the table datatype is a matrix, with the first row being the header. The values in the first column must be unique, and are called the table’s “keys”. Correspondingly, the first header entry is called the “keyname”, and the remaining header entries are the table’s “fields”. The rest of the matrix is the table’s data, accessible by indexing with keys and fields. Missing values are represented by blanks. The conceptual layout of the table is illustrated in the figure below.

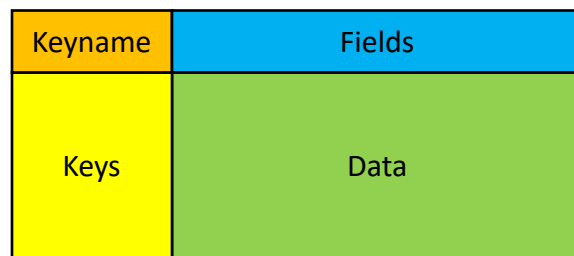


Figure 1: Conceptual Representation of Tabular Data Structure

The command *table* is a TclOO class based on the superclass *::ndlist::ValueContainer*. It is an object-oriented approach to tabular data manipulation.

```
table new $varName <$value>
```

```
table create $name $varName <$value>
```

<b>\$varName</b>	Variable to store object name for access and garbage collection.
<b>\$value</b>	Matrix representation of table. Default blank.
<b>\$name</b>	Name of object if using “create” method.

### Example 55: Creating and accessing a table

*Code:*

```
table new tableObj {{key A B} {1 foo bar} {2 hello world}}
puts [$tableObj]
```

*Output:*

```
{key A B} {1 foo bar} {2 hello world}
```

## Basic Operators

Because the table class is a subclass of `::ndlist::ValueContainer`, it has the same operator methods.

The copy operator, “`-->`”, copies the table to a new variable, and returns the new object.

```
$tableObj --> $varName
```

**\$varName**                      Variable to store object name for access and garbage collection.

The assignment operator, “`=`”, sets the value of the entire table, and the math assignment operator, “`:=`”, sets the value after passing the input through the Tcl *expr* command. Both operators return the object.

```
$tableObj = $value
```

```
$tableObj := $expr
```

**\$value**                      Value to assign.

**\$expr**                      Expression to evaluate.

The pipe operator, “`|`”, copies the table to a temporary object, and evaluates the method. Returns the result of the method, or the value of the temporary object. This operator is useful for converting methods that modify the object to methods that return a modified value.

```
$tableObj | $method $arg ...
```

**\$method**                      Method to evaluate.

**\$arg ...**                      Arguments to pass to method.

The ampersand operator “`&`” copies the table value to a reference variable, and evaluates a body of script. The changes made to the reference variable will be applied to the object, and if the variable is unset, the object will be deleted. If the variable is unset in the script, the object will be destroyed. Returns the result of the script.

```
$tableObj & $refName $body
```

**\$refName**                      Variable name to use for reference.

**\$body**                      Body to evaluate.

## Wiping, Clearing, and Cleaning a Table

The method *wipe* removes all data from a table object, so that its state is the same as a fresh table. The method *clear* only removes the data and keys stored in the table, keeping the fields and other metadata. The method *clean* only removes keys and fields that have no data.

```
$tableObj wipe
```

```
$tableObj clear
```

```
$tableObj clean
```

### Example 56: Cleaning the table

Code:

```
table new tableObj
$tableObj = {
  {key x y z}
  {1 {} foo bar}
  {2 {} hello world}
  {3 {} {} {}}
}
puts [$tableObj]
# Remove keys and fields with no data
$tableObj clean
puts [$tableObj]
# Remove all keys and data, keep fields
$tableObj clear
puts [$tableObj]
# Reset table
$tableObj wipe
puts [$tableObj]
```

Output:

```
{key x y z} {1 {} foo bar} {2 {} hello world} {3 {} {} {}}
{key y z} {1 foo bar} {2 hello world}
{key y z}
key
```



## Table Access

The matrix representation of the table can be accessed by calling the object without any methods. In addition, the four parts of the table can be accessed with the methods *keyname*, *keys*, *fields*, and *values*.

```
$tableObj keyname
```

```
$tableObj keys <$pattern>
```

```
$tableObj fields <$pattern>
```

```
$tableObj values <$filler>
```

**\$pattern**                      Glob pattern to filter keys/fields with. Default “\*” for all.

**\$filler**                      Filler for missing values, default blank.

### Example 57: Access table components

*Code:*

```
table new tableObj
$tableObj = {
  {key A B}
  {1 foo bar}
  {2 hello world}
}
puts [$tableObj]
puts [$tableObj keyname]
puts [$tableObj keys]
puts [$tableObj fields]
puts [$tableObj values]
```

*Output:*

```
{key A B} {1 foo bar} {2 hello world}
key
1 2
A B
{foo bar} {hello world}
```

Note: the default keyname of a table is “key”.

## Table Data

Although the string representation of the table is a matrix, it is stored as a dictionary within the object. This can be accessed with *\$tableObj dict*, and returns a double-nested dictionary, with the first level of keys representing the table keys, and the second level representing the table fields. Missing values are represented with missing dictionary entries.

```
$tableObj dict
```

## Table Dimensions

The number of keys can be queried with *\$tableObj height* and the number of fields can be queried with *\$tableObj width*. Note that rows and columns with missing data will be counted.

```
$tableObj height
```

```
$tableObj width
```

### Example 58: Accessing table data and dimensions

*Code:*

```
table new tableObj {{key A B} {1 foo bar} {2 hello world} {3 {} {}}}  
puts [$tableObj dict]  
puts [$tableObj height]  
puts [$tableObj width]
```

*Output:*

```
1 {A foo B bar} 2 {A hello B world} 3 {}  
3  
2
```

## Check Existence of Table Keys/Fields

The existence of a table key, field, or table value can be queried with the method *exists*.

```
$tableObj exists key $key  
$tableObj exists field $field  
$tableObj exists value $key $field
```

`$key`                      Key to check.  
`$field`                    Field to check.

## Get Row/Column Indices

The row or column index of a table key or field can be queried with the method *find*. If the key or field does not exist, returns an error.

```
$tableObj find key $key  
$tableObj find field $field
```

`$key`                      Key to find.  
`$field`                    Field to find.

### Example 59: Find column index of a field

*Code:*

```
table new tableObj {  
  {name x y z}  
  {bob 1 2 3}  
  {sue 3 2 1}  
}  
puts [$tableObj exists field z]  
puts [$tableObj find field z]
```

*Output:*

```
1  
2
```

## Table Entry and Access

Data entry and access to a table object can be done with single values with the methods *set* and *get*, entire rows with *rset* and *rget*, entire columns with *cset* and *cget*, or in matrix fashion with *mset* and *mget*. When entering data, if entry keys/fields do not exist, they are added to the table. Additionally, since blank values represent missing data, setting a value to blank effectively unsets the table entry, but does not remove the key or field.

### Single Value Entry and Access

The methods *set* and *get* allow for easy entry and access of single values in the table. Note that multiple field-value pairings can be used in *\$tableObj set*.

```
$tableObj set $key $field $value ...
```

```
$tableObj get $key $field <$filler>
```

<b>\$key</b>	Key of row to set/get data in/from.
<b>\$field</b>	Field of column to set/get data in/from.
<b>\$value</b>	Value to set.
<b>\$filler</b>	Filler to return if value is missing. Default blank.

#### Example 60: Getting and setting values in a table

*Code:*

```
table new tableObj
# Set multiple values at once
$tableObj set 1 x 2.0 y 3.0 z 6.5
# Access values in the table
puts [$tableObj get 1 x]
puts [$tableObj get 1 y]
```

*Output:*

```
2.0
3.0
```

Note: the “filler” is only returned if the key and field exist, but the value does not.

## Row Entry and Access

The methods *rset* and *rget* allow for easy row entry and access. Entry list length must match table width or be scalar. If entry list is blank, it will delete the row, but not the key.

```
$tableObj rset $key $row
```

```
$tableObj rget $key <$filler>
```

<b>\$key</b>	Key of row to set/get.
<b>\$row</b>	List of values (or scalar) to set.
<b>\$filler</b>	Filler for missing values. Default blank.

## Column Entry and Access

The methods *cset* and *cget* allow for easy column entry and access. Entry list length must match table height or be scalar. If entry list is blank, it will delete the column, but not the field.

```
$tableObj cset $field $column
```

```
$tableObj cget $field <$filler>
```

<b>\$field</b>	Field of column to set/get.
<b>\$column</b>	List of values (or scalar) to set.
<b>\$filler</b>	Filler for missing values. Default blank.

### Example 61: Setting entire rows/columns

*Code:*

```
table new tableObj {{key A B}}  
$tableObj rset 1 {1 2}  
$tableObj rset 2 {4 5}  
$tableObj rset 3 {7 8}  
$tableObj cset C {3 6 9}  
puts [$tableObj]
```

*Output:*

```
{key A B C} {1 1 2 3} {2 4 5 6} {3 7 8 9}
```

## Matrix Entry and Access

The methods *mset* and *mget* allow for easy matrix-style entry and access. Entry matrix size must match dimensions of input keys and fields or be a scalar.

```
$tableObj mset $keys $fields $matrix
```

```
$tableObj mget $keys $fields <$filler>
```

<b>\$keys</b>	List of keys to set/get (default all keys).
<b>\$fields</b>	List of keys to set/get (default all keys).
<b>\$matrix</b>	Matrix of values (or scalar) to set.
<b>\$filler</b>	Filler for missing values. Default blank.

### Example 62: Matrix entry and access

*Code:*

```
table new T
$T mset {1 2 3 4} {A B} 0.0; # Initialize as zero
$T mset {1 2 3} A {1.0 2.0 3.0}; # Set subset of table
puts [$T mget [$T keys] [$T fields]]; # Same as [$T values]
```

*Output:*

```
{1.0 0.0} {2.0 0.0} {3.0 0.0} {0.0 0.0}
```

## Iterating Over Table Data

Table data can be looped through, row-wise, with the method *with*. Variables representing the key values and fields will be assigned their corresponding values, with blanks representing missing data. The variable representing the key (table keyname) is static, but changes made to field variables are reflected in the table. Unsetting a field variable or setting its value to blank unsets the corresponding data in the table.

```
$tableObj with $body
```

**\$body**

Script to evaluate.

### Example 63: Iterating over a table, accessing and modifying field values

*Code:*

```
table new parameters {{key x y z}}
$parameters set 1 x 1.0 y 2.0
$parameters set 2 x 3.0 y 4.0
$parameters with {
    set z [expr {$x + $y}]
}
puts [$parameters cget z]
```

*Output:*

```
3.0 7.0
```

Note: Just like in *dict with*, the key variable and field variables in *\$tableObj with* persist after the loop.

## Field Expressions

The method *expr* computes a list of values according to a field expression, and the method *query* returns the keys in a table that match criteria in an expression. In the same style as referring to variables with the dollar sign (\$), the “at” symbol (@) is used by *\$tableObj expr* to refer to field values, or row keys if the keyname is used. This is similar to the syntax for *neexpr*, but the table expression method is limited to fields within a table. Additionally, unlike ND-array variable names, field names are not limited to word characters, as shown in the example. If any referenced fields have missing values for a table row, the corresponding result will be blank as well. The resulting list corresponds to the keys in the table.

```
$tableObj expr $expr
```

```
$tableObj query $expr
```

**\$expr**

Field expression.

### Example 64: Math operation over table columns

Code:

```
table new myTable
$myTable set 1 x 1.0
$myTable set 2 x 2.0
$myTable set 3 x 3.0
set a 20.0
puts [$myTable expr {@x*2 + $a}]
```

Output:

```
22.0 24.0 26.0
```

### Example 65: Getting data that meets a criteria

Code:

```
# Create blank table with keyname "StudentID"
table new classData StudentID
$classData set 1 name bob {height (cm)} 175 {weight (kg)} 60
$classData set 2 name frank {height (cm)} 180 {weight (kg)} 75
$classData set 3 name sue {height (cm)} 165 {weight (kg)} 55
$classData set 4 name sally {height (cm)} 150 {weight (kg)} 50
# Subset of data where height is greater than 160
puts [$classData mget [$classData query {@{height (cm)} > 160}] {name {height (cm)}}]
```

Output:

```
{bob 175} {frank 180} {sue 165}
```



## Table Index Operator

The “@” operator is a short-hand way to access or modify table columns.

```
$tableObj @ $field = $column
```

```
$tableObj @ $field := $expr
```

```
$tableObj @ $field <$filler>
```

<code>\$field</code>	Field of column to set/get.
<code>\$column</code>	List of values (or scalar) to set.
<code>\$expr</code>	Field expression.
<code>\$filler</code>	Filler for missing values. Default blank.

### Example 66: Accessing and modifying table columns

*Code:*

```
table new myTable
$myTable define keys {1 2 3}
$myTable @ x = {1.0 2.0 3.0}
set a 20.0
$myTable @ y := {@x*2 + $a}
puts [$myTable @ y]
```

*Output:*

```
22.0 24.0 26.0
```

## Searching a Table

Besides searching for specific field expression criteria with *\$tableObj query*, keys matching criteria can be found with the method *search*. The method *search* searches a table using the Tcl *lsearch* command on the keys or field values. The default search method uses “glob” pattern matching, and returns the matching keys. This search behavior can be changed with the various options, which are taken directly from the Tcl *lsearch* command. Therefore, while brief descriptions of the options are provided here, they are explained more in depth in the Tcl documentation.

```
$tableObj search <$option ...> <$field> $value
```

<b>\$option ...</b>	Searching options. Valid options:
<b>-exact</b>	Compare strings exactly
<b>-glob</b>	Use glob-style pattern matching (default)
<b>-regexp</b>	Use regular expression matching
<b>-sorted</b>	Assume elements are in sorted order
<b>-all</b>	Get all matches, rather than the first match
<b>-not</b>	Negate the match(es)
<b>-ascii</b>	Use string comparison (default)
<b>-dictionary</b>	Use dictionary-style comparison
<b>-integer</b>	Use integer comparison
<b>-real</b>	Use floating-point comparison
<b>-nocase</b>	Search in a case-insensitive manner
<b>-increasing</b>	Assume increasing order (default)
<b>-decreasing</b>	Assume decreasing order
<b>-bisect</b>	Perform inexact match
<b>--</b>	Signals end of options
<b>\$field</b>	Field to search. If blank, searches keys.
<b>\$value</b>	Value or pattern to search for

Note: If a field contains missing values, they will only be included in the search if the search options allow (e.g. blanks are included for string matching, but not for numerical matching).

## Sorting a Table

The method `sort` sorts a table by keys or field values, and returns the table object. The default sorting method is in increasing order, using string comparison. This sorting behavior can be changed with the various options, which are taken directly from the Tcl `lsort` command. Therefore, while brief descriptions of the options are provided here, they are explained more in depth in the Tcl documentation. Note: If a field contains missing values, the missing values will be last, regardless of sorting options.

```
$tableObj sort <$option ...> <$field ...>
```

<b>\$option ...</b>	Sorting options. Valid options:
<b>-ascii</b>	Use string comparison (default)
<b>-dictionary</b>	Use dictionary-style comparison
<b>-integer</b>	Use integer comparison
<b>-real</b>	Use floating comparison
<b>-increasing</b>	Sort the list in increasing order (default)
<b>-decreasing</b>	Sort the list in decreasing order
<b>-nocase</b>	Compare in a case-insensitive manner
<b>--</b>	Signals end of options
<b>\$field ...</b>	Fields to sort by (in order of sorting). If blank, sorts by keys.

### Example 67: Searching and sorting

*Code:*

```
# Use zip command to make a one-column table
table new data [zip {key 1 2 3 4 5} {x 3.0 2.3 5.0 2.0 1.8}]
# Find key corresponding to x value of 5
puts [$data search -exact -real x 5]
# Sort the table, and print list of keys and values
$data sort -real x
puts [zip [$data keys] [$data cget x]]
```

*Output:*

```
3
{5 1.8} {4 2.0} {2 2.3} {1 3.0} {3 5.0}
```

## Merging Tables

Data from other tables can be merged into the table object with *\$tableObj merge*. In order to merge, all the tables must have the same keyname. If the merge is valid, the table data is combined, with later entries taking precedence. Additionally, the keys and fields are combined, such that if a key appears in any of the tables, it is in the combined table. Returns the table object.

```
$tableObj merge $object ...
```

**\$object ...** Other table objects to merge into table. Does not destroy the input tables.

### Example 68: Merging data from other tables

*Code:*

```
table new table1 {{key A B} {1 foo bar} {2 hello world}}  
table new table2 {{key B} {1 foo} {2 there}}  
$table1 merge $table2  
puts [$table1]
```

*Output:*

```
{key A B} {1 foo foo} {2 hello there}
```

## Re-keying a Table

The method *mkkey* makes a field the key of a table, and makes the key a field. If a field is empty for some keys, those keys will be lost. Additionally, if field values repeat, only the last entry for that field value will be included. This method is intended to be used with a field that is full and unique, and if the keyname matches a field name, this command will return an error.

```
$tableObj mkkey $field
```

**\$field**                      Field to swap with key.

### Example 69: Re-keying a table

*Code:*

```
table new tableObj {{ID A B C} {1 1 2 3} {2 4 5 6} {3 7 8 9}}
$tableObj mkkey A
puts [$tableObj]
```

*Output:*

```
{A B C ID} {1 2 3 1} {4 5 6 2} {7 8 9 3}
```

## Key and Field Manipulation

The matrix representation of the tabular data is not stored directly in the table object. Rather, the data is stored in an unordered dictionary. The order is preserved in the order of the key and field lists, which are used to construct the ordered matrix representation. The following methods modify the key and field lists, but do not modify the raw data (except when removing keys/fields).

### Overwriting Keys/Fields

The method *define* overwrites the keyname, keys, and fields of the table, additionally filtering the data or adding keys and fields as necessary. For example, if the keys are defined to be a subset of the current keys, it will filter the data to only include the key subset.

```
$tableObj define keyname $keyname
$tableObj define keys $keys
$tableObj define fields $fields
```

<b>\$keyname</b>	Name of keys (first column header).
<b>\$keys</b>	Unique list of keys.
<b>\$fields</b>	Unique list of fields.

### Adding or Removing Keys/Fields

The method *add* adds keys or fields to a table, appending to the end of the key/field lists. If a key or field already exists it is ignored. The method *remove* removes keys or fields and their corresponding rows and columns from a table. If a key or field does not exist, it is ignored.

```
$tableObj add keys $key ...
$tableObj add fields $field ...
```

```
$tableObj remove keys $key ...
$tableObj remove fields $field ...
```

<b>\$key ...</b>	Keys to add/remove.
<b>\$field ...</b>	Fields to add/remove.

## Inserting Keys/Fields

The method *insert* inserts keys or fields at a specific row or column index. Input keys or fields must be unique and must not already exist.

```
$tableObj insert keys $index $key ...  
$tableObj insert fields $index $field ...
```

<b>\$index</b>	Row/column index to insert at.
<b>\$key ...</b>	Keys to insert.
<b>\$field ...</b>	Fields to insert.

## Renaming Keys/Fields

The method *rename* renames keys or fields. Old keys and fields must exist. Duplicates are not allowed in old and new key/field lists.

```
$tableObj rename keys <$old> $new  
$tableObj rename fields <$old> $new
```

<b>\$old</b>	Keys/fields to rename. Default all keys/fields.
<b>\$new</b>	New keys/fields. Must be same length as \$old.

### Example 70: Renaming fields

*Code:*

```
table new tableObj {{key A B C} {1 1 2 3}}  
$tableObj rename fields {x y z}  
puts [$tableObj]
```

*Output:*

```
{key x y z} {1 1 2 3}
```

## Moving Keys/Fields

Existing keys and fields can be moved with the method *move*.

```
$tableObj move key $key $index  
$tableObj move field $field $index
```

<code>\$key</code>	Key to move.
<code>\$field</code>	Field to move.
<code>\$index</code>	Row/column index to move to.

## Swapping Keys/Fields

Existing keys and fields can be swapped with the method *swap*. To swap a field column with the key column, use the method *mkkey*.

```
$tableObj swap keys $key1 $key2  
$tableObj swap fields $field1 $field2
```

<code>\$key1 \$key2</code>	Keys to swap.
<code>\$field1 \$field2</code>	Fields to swap.

### Example 71: Swapping table rows

*Code:*

```
table new tableObj  
$tableObj define keys {1 2 3 4}  
$tableObj cset A {2.0 4.0 8.0 16.0}  
$tableObj swap keys 1 4  
puts [$tableObj]
```

*Output:*

```
{key A} {4 16.0} {2 4.0} {3 8.0} {1 2.0}
```



---

## File Import/Export

The commands *readFile* and *writeFile* perform simple data import/export, while the commands *readMatrix* and *writeMatrix* dynamically convert files to matrix format and matrices to file format.

```
readFile <$option $value ...> <-newline> $file
```

```
readMatrix <$option $value ...> <-newline> $file
```

**\$option \$value ...** File configuration options, see Tcl *fconfigure* command.

**-newline** Option to read the final newline if it exists.

**\$file** File to read data from.

```
writeFile <$option $value ...> <-newline> $file $data
```

```
writeMatrix <$option $value ...> <-newline> $file $data
```

**\$option \$value ...** File configuration options, see Tcl *fconfigure* command.

**-newline** Option to not write a final newline.

**\$file** File to write data to.

**\$data** Data to write to file.

### Example 72: File import/export

#### Code:

```
# Export matrix to file (converts to csv)
writeMatrix example.csv {{foo bar} {hello world}}
# Read CSV file
puts [readFile example.csv]
puts [readMatrix example.csv]; # converts from csv to matrix
file delete example.csv
```

#### Output:

```
foo,bar
hello,world
{foo bar} {hello world}
```

## Data Conversions

The commands *mat2txt* and *txt2mat* convert between matrix and space-delimited text, where new-lines separate rows. Escaping of spaces and newlines is consistent with Tcl rules for valid lists.

```
mat2txt $mat
```

```
txt2mat $txt
```

**\$mat**                      Matrix value.  
**\$txt**                      Space-delimited values.

The commands *mat2csv* and *csv2mat* convert between matrix and CSV-formatted text, where new lines separate rows. Commas and newlines are escaped with quotes, and quotes are escaped with double-quotes.

```
mat2csv $mat
```

```
csv2mat $csv
```

**\$mat**                      Matrix value.  
**\$csv**                      Comma-separated values.

### Example 73: Data conversions

#### Code:

```
set matrix {{A B C} {{hello world} foo,bar {"hi"}}}  
puts {TXT format:}  
puts [mat2txt $matrix]  
puts {CSV format:}  
puts [mat2csv $matrix]
```

#### Output:

```
TXT format:  
A B C  
{hello world} foo,bar {"hi"}  
CSV format:  
A,B,C  
hello world,"foo,bar","""hi"""
```

## Interface with SQLite Database Tables

The `sqlite3` Tcl package provides access to SQL commands within a Tcl interpreter. SQL databases are a powerful way to manipulate and query data, so to aid in using this powerful tool, the following commands were added to the `ndlist` package: `readTable` and `writeTable`. The command `readTable` reads a specified table from a SQL database, returning a matrix. The command `writeTable` writes a matrix to a table in a SQL database. If the table exists, it will be overwritten. If `sqlite3` is not loaded before calling these commands, they will throw an error.

```
readTable $db $table
```

```
writeTable $db $table $matrix
```

<code>\$db</code>	sqlite3 database command
<code>\$table</code>	Name of table in database to read/write.
<code>\$matrix</code>	Matrix, with first row as headers, to read/write.

### Example 74: Writing a table to SQLite

Code:

```
# Matrix of data used for table (https://www.tutorialspoint.com/sqlite/company.sql)
set matrix [txt2mat {\
ID      NAME      AGE      ADDRESS      SALARY
1       Paul      32      California  20000.0
2       Allen     25      Texas       15000.0
3       Teddy     23      Norway      20000.0
4       Mark      25      Rich-Mond   65000.0
5       David     27      Texas       85000.0
6       Kim       22      South-Hall  45000.0
7       James     24      Houston     10000.0}]

# Write the data to an SQL table and a Tcl table
package require sqlite3; # required for sqlite3 command
sqlite3 db myDatabase.db; # open SQL database
writeTable db People $matrix; # write to SQL table
table new People $matrix; # write to Tcl table

# Example of a query, both with SQL and Tcl table commands
puts [db eval {SELECT NAME FROM People WHERE SALARY > 40000.0;}]
puts [$People mget [$People query {@SALARY > 40000.0}] NAME]
db close; # close SQL database
```

Output:

```
Mark David Kim
Mark David Kim
```

## Reading and Writing SQL Databases

The command *readDatabase* reads all the tables in an SQLite database to a Tcl array of table objects. Likewise, the command *writeDatabase* writes all the tables in a Tcl array to tables in a SQLite database, overwriting those tables if they exist.

```
readDatabase $db $arrayName
```

```
writeDatabase $db $arrayName
```

<b>\$db</b>	sqlite3 database command
<b>\$arrayName</b>	Variable name of array with keys corresponding to table names, and values corresponding to table objects.

### Example 75: Reading and writing an entire SQL database

*Code:*

```
package require sqlite3; # required for sqlite3 command
sqlite3 db myDatabase.db; # open SQL database
readDatabase db t
$t(People) @ AGE := {@AGE + 1}; # add one to people's ages
writeDatabase db t
db close; # close SQL database
# Verifying that data was modified
sqlite3 db myDatabase.db; # open SQL database
puts [db eval {SELECT NAME,AGE FROM People}]
db close; # close SQL database
```

*Output:*

```
Paul 33 Allen 26 Teddy 24 Mark 26 David 28 Kim 23 James 25
```

---

## Utilities for Object-Oriented Programming

As of Tcl version 8.6, there is no garbage collection for Tcl objects, they have to be removed manually with the *destroy* method. The command *tie* is a solution for this problem, using variable traces to destroy the corresponding object when the variable is unset or modified. For example, if an object is tied to a local procedure variable, the object will be destroyed when the procedure returns.

```
tie $varName <$object>
```

**\$varName**                      Name of variable for garbage collection.

**\$object**                      Object to tie variable to. Default self-ties (uses current value).

Tied variables can be untied with the command *untie*. Renaming or destroying an object also unties all variables tied to it.

```
untie $name1 $name2 ...
```

**\$name1 \$name2 ...**              Variables to untie.

### Example 76: Variable-object ties

*Code:*

```
oo::class create foo {
    method sayhello {} {
        puts {hello world}
    }
}
tie a [foo create bar]
set b $a; # object alias
$a sayhello
$b sayhello
unset a; # destroys object
$b sayhello; # throws error
```

*Output:*

```
hello world
hello world
invalid command name "::bar"
```

Note: You can tie array elements, but not an entire array, and you cannot tie a read-only variable.

---

## Garbage Collection Superclass

The class `::ndlist::GarbageCollector` is a TclOO superclass that includes garbage collection by tying the object to a specified variable using *tie*. This class is not exported.

Below is the syntax for the superclass constructor.

```
::ndlist::GarbageCollector new $varName
```

```
::ndlist::GarbageCollector create $name $varName
```

**\$varName**                      Name of variable for garbage collection.

**\$name**                        Name of object (for “create” method).

In addition to tying the object to a variable in the constructor, the `::ndlist::GarbageCollector` superclass provides a public copy method: “-->”, which calls the private method *CopyObject*.

```
$gcObj --> $varName
```

```
my CopyObject $varName
```

**\$varName**                      Name of variable for garbage collection.

Below is an example of how this superclass can be used to build garbage collection into a TclOO class. This process is formalized with the superclass `::ndlist::ValueContainer`.

### Example 77: Simple value container class

*Code:*

```
oo::class create value {  
    superclass ::ndlist::GarbageCollector  
    variable myValue  
    method set {value} {set myValue $value}  
    method value {} {return $myValue}  
}  
[value new x] --> y; # create x, and copy to y.  
$y set {hello world}; # modify $y  
unset x; # destroys $x  
puts [$y value]
```

*Output:*

```
hello world
```

---

## Container Superclass

The class `::ndlist::ValueContainer` is a TclOO superclass, built on-top of the `::ndlist::GarbageCollector` superclass. In addition to the copy method “`-->`”, this class stores a value in the variable “myValue”, which can be accessed with the methods *GetValue* and *SetValue*. This class is not exported.

Below is the syntax for the superclass constructor.

```
::ndlist::ValueContainer new $varName <$value>
```

```
::ndlist::ValueContainer create $name $varName <$value>
```

<code>\$name</code>	Name of object (for “create” method).
<code>\$varName</code>	Name of variable for garbage collection.
<code>\$value</code>	Value to store in object. Default blank.

### Getting and Setting

Calling the object by itself calls the *GetValue* method, which simply queries the value in the container. The assignment operator, “`=`”, calls the *SetValue* method, which sets the value in the container.

```
$vcObj = $value
```

```
my SetValue $value
```

<code>\$value</code>	Value to store in container.
----------------------	------------------------------

#### Example 78: Simple container

*Code:*

```
::ndlist::ValueContainer new x
$x = {hello world}
puts [$x]
```

*Output:*

```
hello world
```

## Math Assignment Operator

The math assignment operator, “:=”, calls the *SetValue* method after evaluating the expression passed through the Tcl *expr* command.

```
$vcObj := $expr
```

**\$expr**                      Expression to evaluate and assign to object.

The math assignment operator makes use of the private method *Uplevel*, which evaluates the body of script at a specified level, while making the object command name available through the alias “\$.”. This can be nested, as it restores the old alias after evaluation.

```
my Uplevel $level $body
```

```
$. $arg ...
```

**\$arg ...**                      Method arguments for object.

### Example 79: Modifying a container object

*Code:*

```
[::ndlist::ValueContainer new x] = 5.0  
$x := {[$.] + 5}  
puts [$x]
```

*Output:*

```
10.0
```



## Advanced Operators

The pipe operator, “|”, calls the *TempObject* method, which copies the object and evaluates the method, returning the result or the value of the temporary object if the result is the object.

```
$vcObj | $method $arg ...
```

```
my TempObject $method $arg ...
```

**\$method**                      Method to evaluate in temporary object.

**\$arg ...**                    Arguments for method.

The ampersand operator “&”, calls the *RefEval* method, which copies the value to a variable, and evaluates a body of script. The changes made to the variable will be applied to the object, and if the variable is unset, the object will be deleted. Returns the result of the script.

```
$vcObj & $varName $body
```

```
my RefEval $varName $body
```

**\$varName**                    Variable name to use for reference.

**\$body**                      Body to evaluate.

### Example 80: Advanced methods

*Code:*

```
[::ndlist::ValueContainer new x] = {1 2 3}
# Use ampersand method to use commands that take variable name as input
$x & ref {
  lappend ref 4
}
puts [$x | = {hello world}]; # operates on temp object
puts [$x]
```

*Output:*

```
hello world
1 2 3 4
```

---

## Command Index

`::ndlist::GarbageCollector`, 62  
`::ndlist::Index2Integer`, 20  
`::ndlist::ParseIndex`, 20  
`::ndlist::ValueContainer`, 63  
`$.`, 64  
  
`augment`, 10  
  
`block`, 10  
  
`cartprod`, 13  
`cross`, 7  
`csv2mat`, 58  
  
`dot`, 7  
  
`eye`, 9  
  
`find`, 3  
  
`gc` methods  
    `-->`, 62  
  
`i`, 29  
  
`j`, 29  
  
`k`, 29  
`kronprod`, 12  
  
`lapply`, 5  
`lapply2`, 5  
`linspace`, 4  
`linsteps`, 4  
`linterp`, 3  
  
`mat2csv`, 58  
  
`mat2txt`, 58  
`matmul`, 11  
`max`, 6  
`mean`, 6  
`median`, 6  
`min`, 6  
  
`napply`, 26  
`napply2`, 26  
`narray`, 30  
`narray` methods, 32  
    `|`, 36  
    `-->`, 32  
    `:=`, 34  
    `<value>`, 30  
    `=`, 34  
    `&`, 37  
    `insert`, 35  
    `ndims`, 31  
    `remove`, 35  
    `shape`, 31  
    `size`, 31  
`ncat`, 24  
`ndims`, 14  
`ndims_multiple`, 14  
`neval`, 33  
`nexpand`, 17  
`nexpr`, 33  
`nextend`, 18  
`nflatten`, 19  
`nfull`, 16  
`nget`, 21

ninsert, 24	:=, 39
nmap, 28	=, 39
nmoveaxis, 25	&, 39
norm, 7	add, 54
npad, 18	cget, 45
npermute, 25	clean, 40
nrnd, 16	clear, 40
nreduce, 27	cset, 45
nremove, 23	define, 54
nrepeat, 17	dict, 42
nreplace, 22	exists, 43
nreshape, 19	expr, 48
nset, 22	fields, 41
nshape, 15	find, 43
nsiz, 15	get, 44
nswapaxes, 25	height, 42
ones, 9	insert, 55
outerprod, 12	keyname, 41
product, 6	keys, 41
pstdev, 6	merge, 52
range, 2	mget, 46
readDatabase, 60	mkkey, 53
readFile, 57	move, 56
readMatrix, 57	mset, 46
readTable, 59	query, 48
stack, 10	remove, 54
stdev, 6	rename, 55
sum, 6	rget, 45
table, 38	rset, 45
table methods, 49	search, 50
, 39	set, 44
-->, 39	sort, 51
	swap, 56
	values, 41

- width, 42
  - wipe, 40
  - with, 47
- tie, 61
- transpose, 11
- txt2mat, 58
- untie, 61
- vc methods
  - |, 65
  - :=, 64
  - =, 63
  - &, 65
- writeDatabase, 60
- writeFile, 57
- writeMatrix, 57
- writeTable, 59
- zeros, 9
- zip, 13
- zip3, 13