Tablicious for Octave

version 0.1.0, April 2019

Andrew Janke

This manual is for Tablicious, version 0.1.0.

Short Contents

1 Introduction 1
2 Getting Started 2
3 Table Representation 3
4 Missing Functionality 4
5 API Reference 5
6 Copying 37

Table of Contents

1 Introduction **1**

2 Getting Started **2**

3 Table Representation **3**

3.1 `table` Class 3

4 Missing Functionality **4**

5 API Reference **5**

5.1 API by Category 5

5.1.1 Tables 5

5.1.2 Data Types 5

5.1.3 Missing Data 5

5.1.4 Miscellaneous 6

5.2 API Alphabetically 6

5.2.1 array2table 6

5.2.2 categorical 6

5.2.2.1 categorical.categorical 7

5.2.2.2 categorical.categories 8

5.2.2.3 categorical.iscategory 8

5.2.2.4 categorical.isordinal 8

5.2.2.5 categorical.string 8

5.2.2.6 categorical.cellstr 8

5.2.2.7 categorical.dispstrs 8

5.2.2.8 categorical.summary 9

5.2.2.9 categorical.addcats 9

5.2.2.10 categorical.removecats 9

5.2.2.11 categorical.mergecats 9

5.2.2.12 categorical.renamecats 9

5.2.2.13 categorical.reordercats 10

5.2.2.14 categorical.setcats 10

5.2.2.15 categorical.isundefined 10

5.2.2.16 categorical.ismissing 10

5.2.2.17 categorical.isnannish 10

5.2.2.18 categorical.squeezecats 10

5.2.3 cell2table 11

5.2.4 colvecfun 11

5.2.5 contains 11

5.2.6 discretize 11

5.2.7 dispstrs 12

5.2.8 endsWith . . . 12
5.2.9 eqn . . . 12
5.2.10 fillmissing . . . 13
5.2.11 isfile . . . 13
5.2.12 isfolder . . . 13
5.2.13 ismissing . . . 14
5.2.14 isnannish . . . 14
5.2.15 missing . . . 15
5.2.15.1 missing.missing . . . 15
5.2.15.2 missing.dispstrs . . . 15
5.2.15.3 missing.ismissing . . . 15
5.2.15.4 missing.isnan . . . 15
5.2.15.5 missing.isnannish . . . 16
5.2.16 pp . . . 16
5.2.17 rmmissing . . . 16
5.2.18 standardizeMissing . . . 16
5.2.19 startsWith . . . 17
5.2.20 string . . . 17
5.2.20.1 string.string . . . 18
5.2.20.2 string.isstring . . . 18
5.2.20.3 string.dispstrs . . . 18
5.2.20.4 string.ismissing . . . 18
5.2.20.5 string.isnannish . . . 18
5.2.20.6 string.cellstr . . . 19
5.2.20.7 string.cell . . . 19
5.2.20.8 string.char . . . 19
5.2.20.9 string.encode . . . 19
5.2.20.10 string.strlength_bytes . . . 19
5.2.20.11 string.strlength . . . 20
5.2.20.12 string.reverse_bytes . . . 20
5.2.20.13 string.reverse . . . 20
5.2.20.14 string.strcat . . . 20
5.2.20.15 string.lower . . . 20
5.2.20.16 string.upper . . . 21
5.2.20.17 string.erase . . . 21
5.2.20.18 string.strrep . . . 21
5.2.20.19 string.strfind . . . 21
5.2.20.20 string.regexprep . . . 21
5.2.20.21 string.strcmp . . . 22
5.2.20.22 string.cmp . . . 22
5.2.20.23 string.missing . . . 22
5.2.20.24 string.decode . . . 22
5.2.21 struct2table . . . 23
5.2.22 table . . . 23
5.2.22.1 table.table . . . 23
5.2.22.2 table.summary . . . 24
5.2.22.3 table.prettyprint . . . 24
5.2.22.4 table.table2cell . . . 24

5.2.22.5 table.table2struct 24
5.2.22.6 table.table2array 24
5.2.22.7 table.varnames 24
5.2.22.8 table.istable 25
5.2.22.9 table.size 25
5.2.22.10 table.length 25
5.2.22.11 table.ndims 25
5.2.22.12 table.squeeze 25
5.2.22.13 table.sizeof 25
5.2.22.14 table.height 25
5.2.22.15 table.rows 25
5.2.22.16 table.width 26
5.2.22.17 table.columns 26
5.2.22.18 table.numel 26
5.2.22.19 table.isempty 26
5.2.22.20 table.ismatrix 26
5.2.22.21 table.isrow 26
5.2.22.22 table.iscol 26
5.2.22.23 table.isvector 27
5.2.22.24 table.isscalar 27
5.2.22.25 table.hasrownames 27
5.2.22.26 table.vertcat 27
5.2.22.27 table.horzcat 27
5.2.22.28 table.repmat 27
5.2.22.29 table.setVariableNames 28
5.2.22.30 table.setRowNames 28
5.2.22.31 table.resolveVarRef 28
5.2.22.32 table.subsetRows 28
5.2.22.33 table.subsetvars 28
5.2.22.34 table.removevars 28
5.2.22.35 table.movevars 29
5.2.22.36 table.setvar 29
5.2.22.37 table.convertvars 29
5.2.22.38 table.head 29
5.2.22.39 table.tail 29
5.2.22.40 table.join 30
5.2.22.41 table.innerjoin 30
5.2.22.42 table.outerjoin 30
5.2.22.43 table.outerfillvals 30
5.2.22.44 table.semijoin 31
5.2.22.45 table.antijoin 31
5.2.22.46 table.cartesian 31
5.2.22.47 table.groupby 31
5.2.22.48 table.grpstats 32
5.2.22.49 table.union 32
5.2.22.50 table.intersect 32
5.2.22.51 table.setxor 32
5.2.22.52 table.setdiff 33

5.2.22.53 table.ismember . . . 33
5.2.22.54 table.ismissing . . . 33
5.2.22.55 table.rmmissing . . . 33
5.2.22.56 table.standardizeMissing . . . 33
5.2.22.57 table.varfun . . . 34
5.2.22.58 table.rowfun . . . 34
5.2.22.59 table.findgroups . . . 34
5.2.22.60 table.evalWithVars . . . 34
5.2.22.61 table.restrict . . . 35
5.2.23 tableOuterFillValue . . . 35
5.2.24 vecfun . . . 35

6 Copying . . . 37
6.1 Package Copyright . . . 37
6.2 Manual Copyright . . . 37

1 Introduction

This is the manual for the Tablicious package version 0.1.0 for GNU Octave.

Tablicious provides Matlab-compatible tabular data support for GNU Octave. This includes a `table` class with support for filtering and join operations, Missing Data support, and `string` and `categorical` data types.

This document is a work in progress. You are invited to help improve it and submit patches.

Tablicious was written by Andrew Janke `<floss@apjanke.net>`. Support can be found on the Tablicious project GitHub page (`https://github.com/apjanke/octave-tablicious`).

2 Getting Started

The easiest way to obtain Tablicious is by using Octave's `pkg` package manager. To install the development prerelease of Tablicious, run this in Octave:

```
pkg install https://github.com/apjanke/octave-tablicious/releases/download/v0.1.0/tabl
```

(Check the releases page at `https://github.com/apjanke/octave-tablicious/releases` to find out what the actual latest release number is.)

For development, you can obtain the source code for Tablicious from the project repo on GitHub at `https://github.com/apjanke/octave-tablicious`. Make a local clone of the repo. Then add the `inst` directory in the repo to your Octave path.

3 Table Representation

Tablicious provides the `table` class for representing tabular data.

3.1 `table` Class

A `table` is an array object that represents a tabular data structure. It holds multiple named "variables", each of which is a column vector, or a 2-D matrix whose rows are read as records.

4 Missing Functionality

Tablicious is based on Matlab's table API and supports most of its major functionality. But not all of it is implemented yet. The missing parts are currently:

- `timetable`
- Moving window methods in `fillmissing`
- `summary()` for `table` and `categorical`
- Assignment to table variables using `.`-indexing
- File I/O like `readtable()` and `writetable()`

It is the author's hope that all these will be implemented some day.

5 API Reference

5.1 API by Category

5.1.1 Tables

Section 5.2.22 [table], page 23
Tabular data array containing multiple columnar variables.

Section 5.2.1 [array2table], page 6
Convert an array to a table.

Section 5.2.3 [cell2table], page 11
Convert a cell array to a table.

Section 5.2.21 [struct2table], page 23
Convert struct to a table.

Section 5.2.23 [tableOuterFillValue], page 35
Outer fill value for variable within a table.

5.1.2 Data Types

Section 5.2.20 [string], page 17
A string array of Unicode strings.

Section 5.2.19 [startsWith], page 17
Test if strings start with a pattern.

Section 5.2.8 [endsWith], page 12
Test if strings end with a pattern.

Section 5.2.5 [contains], page 11
Test if strings contain a pattern.

Section 5.2.2 [categorical], page 6
Categorical variable array.

Section 5.2.6 [discretize], page 11
Group data into discrete bins or categories.

5.1.3 Missing Data

Section 5.2.10 [fillmissing], page 13
Fill missing values.

Section 5.2.13 [ismissing], page 14
Find missing values.

Section 5.2.17 [rmmissing], page 16
Remove missing values.

Section 5.2.18 [standardizeMissing], page 16
Insert standard missing values.

Section 5.2.15 [missing], page 15
Generic auto-converting missing value.

Section 5.2.14 [isnannish], page 14
Test if elements are NaN or NaN-like

Section 5.2.9 [eqn], page 12
Determine element-wise equality, treating NaNs as equal

5.1.4 Miscellaneous

Section 5.2.4 [colvecfun], page 11
Apply a function to column vectors in array.

Section 5.2.7 [dispstrs], page 12
Display strings for array.

Section 5.2.11 [isfile], page 13
Test whether file exists and is not a folder.

Section 5.2.12 [isfolder], page 13
Test whether file exists and is a folder.

Section 5.2.16 [pp], page 16
Alias for prettyprint, for interactive use.

Section 5.2.24 [vecfun], page 35
Apply function to vectors in array along arbitrary dimension.

5.2 API Alphabetically

5.2.1 array2table

`out = array2table (c)` [Function]
`out = array2table (..., 'VariableNames', VariableNames)` [Function]
`out = array2table (..., 'RowNames', RowNames)` [Function]

Convert an array to a table.

Converts a 2-D array to a table, with columns in the array becoming variables in the output table. This is typically used on numeric arrays, but it can be applied to any type of array.

You may not want to use this on cell arrays, though, because you will end up with a table that has all its variables of type cell. If you use `cell2table` instead, columns of the cell array which can be condensed into primitive arrays will be. With `array2table`, they won't be.

See also: Section 5.2.3 [cell2table], page 11, Section 5.2.22 [table], page 23, Section 5.2.21 [struct2table], page 23,

5.2.2 categorical

`categorical` [Class]

Categorical variable array.

A `categorical` array represents an array of values of a categorical variable. Each `categorical` array stores the element values along with a list of the categories, and indicators of whether the categories are ordinal (that is, they have a meaningful mathematical ordering), and whether the set of categories is protected (preventing new categories from being added to the array).

In addition to the categories defined in the array, a categorical array may have elements of "undefined" value. This is not considered a category; rather, it is the absence of any known value. It is analagous to a `NaN` value.

This class is not fully implemented yet. Missing stuff: - gt, ge, lt, le - Ordinal support in general - countcats - summary

`uint16 code` [Instance Variable of `categorical`]

The numeric codes of the array element values. These are indexes into the `cats` category list.

This is a planar property.

`logical tfMissing` [Instance Variable of `categorical`]

A logical mask indicating whether each element of the array is missing (that is, undefined).

This is a planar property.

`cellstr cats` [Instance Variable of `categorical`]

The names of the categories in this array. This is the list into which the `code` values are indexes.

`scalar_logical isOrdinal` [Instance Variable of `categorical`]

A scalar logical indicating whether the categories in this array have an ordinal relationship.

5.2.2.1 categorical.categorical

obj = `categorical ()` [Constructor]

Constructs a new scalar categorical whose value is undefined.

obj = `categorical (`*vals*`)` [Constructor]
obj = `categorical (`*vals*`, `*valueset*`)` [Constructor]
obj = `categorical (`*vals*`, `*valueset*`, `*category_names*`)` [Constructor]
obj = `categorical (...,` `'Ordinal'`, *Ordinal*`)` [Constructor]
obj = `categorical (...,` `'Protected'`, *Protected*`)` [Constructor]

Constructs a new categorical array from the given values.

vals is the array of values to convert to categoricals.

valueset is the set of all values from which *vals* is drawn. If omitted, it defaults to the unique values in *vals*.

category_names is a list of category names corresponding to *valueset*. If omitted, it defaults to *valueset*, converted to strings.

Ordinal is a logical indicating whether the category values in *obj* have a numeric ordering relationship. Defaults to false.

Protected indicates whether *obj* should be protected, which prevents the addition of new categories to the array. Defaults to false.

5.2.2.2 categorical.categories

out = `categories` (*obj*) [Method]

Get a list of the categories in *obj*.

Gets a list of the categories in *obj*, identified by their category names.

Returns a cellstr column vector.

5.2.2.3 categorical.iscategory

out = `iscategory` (*obj*, *catnames*) [Method]

Test whether input is a category on a categorical array.

catnames is a cellstr listing the category names to check against *obj*.

Returns a logical array the same size as *catnames*.

5.2.2.4 categorical.isordinal

out = `isordinal` (*obj*) [Method]

Whether *obj* is ordinal.

Returns true if *obj* is ordinal (as determined by its `IsOrdinal` property), and false otherwise.

5.2.2.5 categorical.string

out = `string` (*obj*) [Method]

Convert to string array.

Converts *obj* to a string array. The strings will be the category names for corresponding values, or <missing> for undefined values.

Returns a `string` array the same size as *obj*.

5.2.2.6 categorical.cellstr

out = `cellstr` (*obj*) [Method]

Convert to cellstr.

Converts *obj* to a cellstr array. The strings will be the category names for corresponding values, or `''` for undefined values.

Returns a cellstr array the same size as *obj*.

5.2.2.7 categorical.dispstrs

out = `dispstrs` (*obj*) [Method]

Display strings.

Gets display strings for each element in *obj*. The display strings are either the category string, or `'<undefined>'` for undefined values.

Returns a cellstr array the same size as *obj*.

5.2.2.8 categorical.summary

`summary (obj)` [Method]

Display summary of array's values.

Displays a summary of the values in this categorical array. The output may contain info like the number of categories, number of undefined values, and frequency of each category.

5.2.2.9 categorical.addcats

out `= addcats (obj, newcats)` [Method]

Add categories to categorical array.

Adds the specified categories to *obj*, without changing any of its values.

newcats is a cellstr listing the category names to add to *obj*.

5.2.2.10 categorical.removecats

out `= removecats (obj)` [Method]

Removes all unused categories from *obj*. This is equivalent to `out = squeezecats (obj)`.

out `= removecats (obj, oldcats)` [Method]

Remove categories from categorical array.

Removes the specified categories from *obj*. Elements of *obj* whose values belonged to those categories are replaced with undefined.

newcats is a cellstr listing the category names to add to *obj*.

5.2.2.11 categorical.mergecats

out `= mergecats (obj, oldcats)` [Method]
out `= mergecats (obj, oldcats, newcat)` [Method]

Merge multiple categories.

Merges the categories *oldcats* into a single category. If *newcat* is specified, that new category is added if necessary, and all of *oldcats* are merged into it. *newcat* must be an existing category in *obj* if *obj* is ordinal.

If *newcat* is not provided, all of *odcats* are merged into `oldcats{1}`.

5.2.2.12 categorical.renamecats

out `= renamecats (obj, newcats)` [Method]
out `= renamecats (obj, oldcats, newcats)` [Method]

Rename categories.

Renames some or all of the categories in *obj*, without changing any of its values.

5.2.2.13 categorical.reordercats

`out = reordercats (obj)` [Method]
`out = reordercats (obj, newcats)` [Method]
Reorder categories.

Reorders the categories in *obj* to match *newcats*.

newcats is a cellstr that must be a reordering of *obj*'s existing category list. If *newcats* is not supplied, sorts the categories in alphabetical order.

5.2.2.14 categorical.setcats

`out = setcats (obj, newcats)` [Method]
Set categories for categorical array.

Sets the categories to use for *obj*. If any current categories are absent from the *newcats* list, current values of those categories become undefined.

5.2.2.15 categorical.isundefined

`out = isundefined (obj)` [Method]
Test whether elements are undefined.

Checks whether each element in *obj* is undefined. `"Undefined"` is a special value defined by `categorical`. It is equivalent to a `NaN` or a `missing` value.

Returns a logical array the same size as *obj*.

5.2.2.16 categorical.ismissing

`out = ismissing (obj)` [Method]
Test whether elements are missing.

For categorical arrays, undefined elements are considered to be missing.

Returns a logical array the same size as *obj*.

5.2.2.17 categorical.isnannish

`out = isnannish (obj)` [Method]
Test whethere elements are NaN-ish.

Checks where each element in *obj* is NaN-ish. For categorical arrays, undefined values are considered NaN-ish; any other value is not.

Returns a logical array the same size as *obj*.

5.2.2.18 categorical.squeezecats

`out = squeezecats (obj)` [Method]
Remove unused categories.

Removes all categories which have no corresponding values in *obj*'s elements.

This is currently unimplemented.

5.2.3 cell2table

`out = cell2table (c)` [Function]
`out = cell2table (..., 'VariableNames', VariableNames)` [Function]
`out = cell2table (..., 'RowNames', RowNames)` [Function]

Convert a cell array to a table.

Converts a 2-dimensional cell matrix into a table. Each column in the input *c* becomes a variable in *out*. For columns that contain all scalar values of `cat`-compatible types, they are "popped out" of their cells and condensed into a homogeneous array of the contained type.

See also: Section 5.2.1 [array2table], page 6, Section 5.2.22 [table], page 23, Section 5.2.21 [struct2table], page 23,

5.2.4 colvecfun

`out = colvecfun (fcn, x)` [Function]

Apply a function to column vectors in array.

Applies the given function *fcn* to each column vector in the array *x*, by iterating over the indexes along all dimensions except dimension 1. Collects the function return values in an output array.

fcn must be a function which takes a column vector and returns a column vector of the same size. It does not have to return the same type as *x*.

Returns the result of applying fcn to each column in x, all concatenated together in the same shape as x.

5.2.5 contains

`out = colvecfun (str, pattern)` [Function]
`out = colvecfun (..., 'IgnoreCase', IgnoreCase)` [Function]

Test if strings contain a pattern.

Tests whether the given strings contain the given pattern(s).

str (char, cellstr, or string) is a list of strings to compare against pattern.

pattern (char, cellstr, or string) is a list of patterns to match. These are literal plain string patterns, not regex patterns. If more than one pattern is supplied, the return value is true if the string matched any of them.

Returns a logical array of the same size as the string array represented by *str*.

5.2.6 discretize

`[Y, E] = discretize (X, n)` [Function]
`[Y, E] = discretize (X, edges)` [Function]
`[Y, E] = discretize (X, dur)` [Function]
`[Y, E] = discretize (..., 'categorical')` [Function]
`[Y, E] = discretize (..., 'IncludedEdge', IncludedEdge)` [Function]

Group data into discrete bins or categories.

n is the number of bins to group the values into.

edges is an array of edge values defining the bins.

dur is a `duration` value indicating the length of time of each bin.

If `'categorical'` is specified, the resulting values are a `categorical` array instead of a numeric array of bin indexes.

Returns: *Y* - the bin index or category of each value from *X* *E* - the list of bin edge values

5.2.7 dispstrs

out = `dispstrs` (*x*) [Function]

Display strings for array.

Gets the display strings for each element of *x*. The display strings should be short, one-line, human-presentable strings describing the value of that element.

The default implementation of `dispstrs` can accept input of any type, and has decent implementations for Octave's standard built-in types, but will have opaque displays for most user-defined objects.

This is a polymorphic method that user-defined classes may override with their own custom display that is more informative.

Returns a cell array the same size as *x*.

5.2.8 endsWith

out = `endsWith` (*str*, *pattern*) [Function]

out = `endsWith` (*...*, `'IgnoreCase'`, *IgnoreCase*) [Function]

Test if strings end with a pattern.

Tests whether the given strings end with the given pattern(s).

str (char, cellstr, or string) is a list of strings to compare against *pattern*.

pattern (char, cellstr, or string) is a list of patterns to match. These are literal plain string patterns, not regex patterns. If more than one pattern is supplied, the return value is true if the string matched any of them.

Returns a logical array of the same size as the string array represented by *str*.

5.2.9 eqn

out = `eqn` (*A*, *B*) [Function]

Determine element-wise equality, treating NaNs as equal

out = eqn (A, B)

`eqn` is just like `eq` (the function that implements the `==` operator), except that it considers NaN and NaN-like values to be equal. This is the element-wise equivalent of `isequaln`.

`eqn` uses `isnannish` to test for NaN and NaN-like values, which means that NaNs and NaTs are considered to be NaN-like, and string arrays' "missing" and categorical objects' "undefined" values are considered equal, because they are NaN-ish.

Developer's note: the name "`eqn`" is a little unfortunate, because "eqn" could also be an abbreviation for "equation". But this name follows the `isequaln` pattern of appending an "n" to the corresponding non-NaN-equivocating function.

See also: `eq`, `isequaln`, Section 5.2.14 [isnannish], page 14,

5.2.10 fillmissing

`[out, tfFilled] = fillmissing (X, method)` [Function]
`[out, tfFilled] = fillmissing (X, 'constant', fill_val)` [Function]
`[out, tfFilled] = fillmissing (X, movmethod, window)` [Function]

Fill missing values.

Fills missing values in *X* according to the method specified by *method*.

This method is only partially implemented.

method may be: `'constant' 'previous' 'next' 'nearest' 'linear' 'spline' 'pchip'` *movmethod* may be: `'movmean' 'movmedian'`

Returns *out*, which is *X* but with missing values filled in, and *tfFilled*, a logical array the same size as *X* which indicates which elements were filled.

5.2.11 isfile

`out = isfile (file)` [Function]

Test whether file exists and is not a folder.

Tests whether the given file path *file* exists on the filesystem, and is not a folder (aka "directory"). Files of any type except for directories are considered files by this function.

TODO: Handling of symlinks is undetermined as of yet.

file is a charvec containing the path to the file to test. It may be an absolute or relative path.

This is a new, more specific replacement for `exist(file, "file")`. Unlike `exist`, `isfile` will not search the Octave load path for files.

The underlying logic defers to `stat(file)` for determining file existence and attributes, so any paths supported by `stat` are also supported by `isfile`. In particular, it seems that the `~` alias for the home directory is supported, at least on Unix platforms.

See also: Section 5.2.12 [isfolder], page 13, `exist`

5.2.12 isfolder

`out = isfolder (file)` [Function]

Test whether file exists and is a folder.

Tests whether the given file path *file* exists on the filesystem, and is a folder (aka "directory").

file is a charvec containing the path to the file to test. It may be an absolute or relative path.

This is a new, more specific replacement for `exist(file, "dir")`. Unlike `exist`, `isfolder` will not search the Octave load path for files.

The underlying logic defers to `stat(file)` for determining file existence and attributes, so any paths supported by `stat` are also supported by `isfolder`. In particular, it seems that the `~` alias for the home directory is supported, at least on Unix platforms.

See also: Section 5.2.11 [isfile], page 13, `exist`

5.2.13 ismissing

`out = ismissing (X)` [Function]
`out = ismissing (X, indicator)` [Function]

Find missing values.

Determines which elements of X contain missing values. If an indicator input is not provided, standard missing values depending on the input type of X are used.

Standard missing values depend on the data type: * NaN for double, single, duration, and calendarDuration * NaT for datetime * `' '` for char * `{''}` for cellstrs * Integer numeric types have no standard missing value; they are never considered missing. * Structs are never considered missing. * Logicals are never considered missing. * Other types have no standard missing value; it is currently an error to call `ismissing` on them without providing an indicator. * This includes cells which are not cellstrs; calling `ismissing` on them results in an error. * TODO: Determine whether this should really be an error, or if it should default to never considering those types as missing. * TODO: Decide whether, for classdef objects, `ismissing` should polymorphically detect isnan()/isnat()/isnannish() methods and use those, or whether we should require classes to override ismissing() itself.

If *indicator* is supplied, it is an array containing multiple values, all of which are considered to be missing values. Only indicator values that are type-compatible with the input are considered; other indicator value types are silently ignored. This is by design, so you can pass an indicator that holds sentinel values for disparate types in to ismissing() used for any type, or for compound types like table.

Indicators are currently not supported for struct or logical inputs. This is probably a bug.

Table defines its own ismissing() method which respects individual variables' data types; see Section 5.2.22.54 [table.ismissing], page 33.

5.2.14 isnannish

`out = isnannish (X)` [Function]

Test if elements are NaN or NaN-like

Tests if input elements are NaN, NaT, or otherwise NaN-like. This is true if isnan() or isnat() returns true, and is false for types that do not support isnan() or isnat().

This function only exists because:

a) Matlab decided to call their NaN values for datetime `"NaT"` instead, and test for them with a different `"isnat()"` function, and b) isnan() errors out for some types that do not support isnan(), like cells.

isnannish() smooths over those differences so you can call it polymorphically on any input type.

Under normal operation, isnannish() should not throw an error for any type or value of input.

See also: `isnan`, `isnat`, Section 5.2.13 [ismissing], page 14, Section 5.2.9 [eqn], page 12, `isequaln`

5.2.15 missing

`missing` [Class]

Generic auto-converting missing value.

`missing` is a generic missing value that auto-converts to other types.

A `missing` array indicates a missing value, of no particular type. It auto- converts to other types when it is combined with them via concatenation or other array combination operations.

This class is currently EXPERIMENTAL. Use at your own risk.

Note: This class does not actually work for assignment. If you do this:

```
x = 1:5
x(3) = missing
```

It's supposed to work, but I can't figure out how to do this in a normal classdef object, because there doesn't seem to be any function that's implicitly called for type conversion in that assignment. Darn it.

5.2.15.1 missing.missing

obj = `missing` () [Constructor]

Constructs a scalar `missing` array.

The constructor takes no arguments, since there's only one `missing` value.

5.2.15.2 missing.dispstrs

out = `dispstrs` (*obj*) [Method]

Display strings.

Gets display strings for each element in *obj*.

For `missing`, the display strings are always `'<missing>'`.

Returns a cellstr the same size as *obj*.

5.2.15.3 missing.ismissing

out = `ismissing` (*obj*) [Method]

Test whether elements are missing values.

`ismissing` is always true for `missing` arrays.

Returns a logical array the same size as *obj*.

5.2.15.4 missing.isnan

out = `isnan` (*obj*) [Method]

Test whether elements are NaN.

`isnan` is always true for `missing` arrays.

Returns a logical array the same size as *obj*.

5.2.15.5 missing.isnannish

`out = isnannish (obj)` [Method]

Test whether elements are NaN-like.

`isnannish` is always true for `missing` arrays.

Returns a logical array the same size as *obj*.

5.2.16 pp

`pp (X)` [Function]
`pp (A, B, C, ...)` [Function]
`pp ('A', 'B', 'C', ...)` [Function]
`pp A B C ...` [Function]

Alias for prettyprint, for interactive use.

This is an alias for prettyprint(), with additional name-conversion magic.

If you pass in a char, instead of pretty-printing that directly, it will grab and pretty-print the variable of that name from the caller's workspace. This is so you can conveniently run it from the command line.

5.2.17 rmmissing

`[out, tf] = rmmissing (X)` [Function]
`[out, tf] = rmmissing (X, dim)` [Function]
`[out, tf] = rmmissing (..., 'MinNumMissing', MinNumMissing)` [Function]

Remove missing values.

If *x* is a vector, removes elements with missing values. If *x* is a matrix, removes rows or columns with missing data elements.

dim is the dimension to operate along. Specifying a dimension forces `rmmissing` to operate in matrix instead of vector mode.

MinNumMissing indicates how many missing element values there must be in a row or column for it to be considered missing and this removed. This option is only used in matrix mode; it is silently ignored in vector mode.

Returns: *out* - the input, with missing elements or rows or columns removed *tf* - a logical index vector indicating which elements, rows, or columns were removed

5.2.18 standardizeMissing

`out = standardizeMissing (X, indicator)` [Function]

Insert standard missing values.

Standardizes missing values in *X* by replacing the values listed in *indicator* with the standard missing values for the type of *X*.

Standard missing values depend on the data type: * NaN for double, single, duration, and calendarDuration * NaT for datetime * `' '` for char * `{''}` for cellstrs * Integer numeric types have no standard missing value; they are never considered missing. * Structs are never considered missing. * Logicals are never considered missing.

See also: Section 5.2.22.56 [table.standardizeMissing], page 33,

5.2.19 startsWith

`out = startsWith (str, pattern)` [Function]
`out = startsWith (..., 'IgnoreCase', IgnoreCase)` [Function]

Test if strings start with a pattern.

Tests whether the given strings start with the given pattern(s).

str (char, cellstr, or string) is a list of strings to compare against *pattern*.

pattern (char, cellstr, or string) is a list of patterns to match. These are literal plain string patterns, not regex patterns. If more than one pattern is supplied, the return value is true if the string matched any of them.

Returns a logical array of the same size as the string array represented by *str*.

5.2.20 string

`string` [Class]

A string array of Unicode strings.

A string array is an array of strings, where each array element is a single string.

The string class represents strings, where: - Each element of a string array is a single string - A single string is a 1-dimensional row vector of Unicode characters - Those characters are encoded in UTF-8

This should correspond pretty well to what people think of as strings, and is pretty compatible with people's typical notion of strings in Octave.

String arrays also have a special "missing" value, that is like the string equivalent of NaN for doubles or "undefined" for categoricals, or SQL NULL.

This is a slightly higher-level and more strongly-typed way of representing strings than cellstrs are. (A cellstr array is of type cell, not a text- specific type, and allows assignment of non-string data into it.)

Be aware that while string arrays interconvert with Octave chars and cellstrs, Octave char elements represent 8-bit UTF-8 code units, not Unicode code points.

This class really serves three roles. - It is an object wrapper around Octave's base primitive character types. - It adds ismissing() semantics. - And it introduces Unicode support. Not clear whether it's a good fit to have the Unicode support wrapped up in this. Maybe it should just be a simple object wrapper wrapper, and defer Unicode semantics to when core Octave adopts them for char and cellstr. On the other hand, because Octave chars are UTF-8, not UCS-2, some methods like strlength() and reverse() are just going to be wrong if they delegate straight to chars.

"Missing" string values work like NaNs. They are never considered equal, less than, or greater to any other string, including other missing strings. This applies to set membership and other equivalence tests.

The current implementation depends on Java for its Unicode and encoding support. This means your Octave session must be running Java to call those methods. This should be changed in the future to use a native C/C++ library and avoid the Java dependency, especially before this class is merged into core Octave.

TODO: Need to decide how far to go with Unicode semantics, and how much to just make this an object wrapper over cellstr and defer to Octave's existing char/string-handling functions.

TODO: demote_strings should probably be static or global, so that other functions can use it to hack themselves into being string-aware.

5.2.20.1 string.string

`obj = string ()` [Constructor]
`obj = string (in)` [Constructor]

Construct a new string array.

The zero-argument constructor creates a new scalar string array whose value is the empty string. TODO: Determine if this should actually return a "missing" string instead.

The other constructors construct a new string array by converting various types of inputs. - chars and cellstrs are converted via cellstr() - numerics are converted via num2str() - datetimes are converted via datestr()

5.2.20.2 string.isstring

`out = isstring (obj)` [Method]

Test if input is a string array.

`isstring` is always true for `string` inputs.

Returns a scalar logical.

5.2.20.3 string.dispstrs

`out = dispstrs (obj)` [Method]

Display strings for array elements.

Gets display strings for all the elements in *obj*. These display strings will either be the string contents of the element, enclosed in `"..."`, and with CR/LF characters replaced with `'\r'` and `'\n'` escape sequences, or `"<missing>"` for missing values.

Returns a cellstr of the same size as *obj*.

5.2.20.4 string.ismissing

`out = ismissing (obj)` [Method]

Test whether array elements are missing.

For `string` arrays, only the special "missing" value is considered missing. Empty strings are not considered missing, the way they are with cellstrs.

Returns a logical array the same size as `obj`.

5.2.20.5 string.isnannish

`out = isnannish (obj)` [Method]

Test whether array elements are NaN-like.

Missing values are considered nannish; any other string value is not.

Returns a logical array of the same size as *obj*.

5.2.20.6 string.cellstr

out = `cellstr` (*obj*) [Method]

Convert to cellstr.

Converts *obj* to a cellstr. Missing values are converted to `''`.

Returns a cellstr array of the same size as *obj*.

5.2.20.7 string.cell

out = `cell` (*obj*) [Method]

Convert to cell array.

Converts this to a cell, which will be a cellstr. Missing values are converted to `''`.

This method returns the same values as `cellstr(obj)`; it is just provided for interface compatibility purposes.

Returns a cell array of the same size as *obj*.

5.2.20.8 string.char

out = `char` (*obj*) [Method]

Convert to char array.

Converts *obj* to a 2-D char array. It will have as many rows as *obj* has elements.

It is an error to convert missing-valued `string` arrays to char. (NOTE: This may change in the future; it may be more appropriate) to convert them to space-padded empty strings.)

Returns 2-D char array.

5.2.20.9 string.encode

out = `encode` (*obj*, *charsetName*) [Method]

Encode string in a given character encoding.

obj must be scalar.

charsetName (charvec) is the name of a character encoding. (TODO: Document what determines the set of valid encoding names.)

Returns the encoded string as a `uint8` vector.

See also: Section 5.2.20.24 [string.decode], page 22.

5.2.20.10 string.strlength_bytes

out = `strlength_bytes` (*obj*) [Method]

String length in bytes.

Gets the length of each string in *obj*, counted in Unicode UTF-8 code units (bytes). This is the same as `numel(str)` for the corresponding Octave char vector for each string, but may not be what you actually want to use. You may want `strlength` instead.

Returns double array of the same size as *obj*. Returns NaNs for missing strings.

See also: Section 5.2.20.11 [string.strlength], page 20,

5.2.20.11 string.strlength

out = `strlength` (*obj*) [Method]

String length in characters.

Gets the length of each string, counted in Unicode characters (code points). This is the string length method you probably want to use, not `strlength_bytes`.

Returns double array of the same size as *obj*. Returns NaNs for missing strings.

See also: Section 5.2.20.10 [string.strlength_bytes], page 19,

5.2.20.12 string.reverse_bytes

out = `reverse_bytes` (*obj*) [Method]

Reverse string, byte-wise.

Reverses the bytes in each string in *obj*. This operates on bytes (Unicode code units), not characters.

This may well produce invalid strings as a result, because reversing a UTF-8 byte sequence does not necessarily produce another valid UTF-8 byte sequence.

You probably do not want to use this method. You probably want to use `string.reverse` instead.

Returns a string array the same size as *obj*.

See also: Section 5.2.20.13 [string.reverse], page 20,

5.2.20.13 string.reverse

out = `reverse` (*obj*) [Method]

Reverse string, character-wise.

Reverses the characters in each string in *obj*. This operates on Unicode characters (code points), not on bytes, so it is guaranteed to produce valid UTF-8 as its output.

Returns a string array the same size as *obj*.

5.2.20.14 string.strcat

out = `strcat` (*varargin*) [Method]

String concatenation.

Concatenates the corresponding elements of all the input arrays, string-wise. Inputs that are not string arrays are converted to string arrays.

The semantics of concatenating missing strings with non-missing strings has not been determined yet.

Returns a string array the same size as the scalar expansion of its inputs.

5.2.20.15 string.lower

out = `lower` (*obj*) [Method]

Convert to lower case.

Converts all the characters in all the strings in *obj* to lower case.

This currently delegates to Octave's own `lower()` function to do the conversion, so whatever character class handling it has, this has.

Returns a string array of the same size as *obj*.

5.2.20.16 string.upper

out = `upper` (*obj*) [Method]

Convert to upper case.

Converts all the characters in all the strings in *obj* to upper case.

This currently delegates to Octave's own `upper()` function to do the conversion, so whatever character class handling it has, this has.

Returns a string array of the same size as *obj*.

5.2.20.17 string.erase

out = `erase` (*obj*, *match*) [Method]

Erase matching substring.

Erases the substrings in *obj* which match the *match* input.

Returns a string array of the same size as *obj*.

5.2.20.18 string.strrep

out = `strrep` (*obj*, *match*, *replacement*) [Method]
out = `strrep` (..., *varargin*) [Method]

Replace occurrences of pattern with other string.

Replaces matching substrings in *obj* with a given replacement string.

varargin is passed along to the core Octave `strrep` function. This supports whatever options it does. TODO: Maybe document what those options are.

Returns a string array of the same size as *obj*.

5.2.20.19 string.strfind

out = `strfind` (*obj*, *pattern*) [Method]
out = `strfind` (..., *varargin*) [Method]

Find pattern in string.

Finds the locations where *pattern* occurs in the strings of *obj*.

TODO: It's ambiguous whether a scalar this should result in a numeric out or a cell array out.

Returns either an index vector, or a cell array of index vectors.

5.2.20.20 string.regexprep

out = `regexprep` (*obj*, *pat*, *repstr*) [Method]
out = `regexprep` (..., *varargin*) [Method]

Replace based on regular expression matching.

Replaces all the substrings matching a given regexp pattern *pat* with the given replacement text *repstr*.

Returns a string array of the same size as *obj*.

5.2.20.21 string.strcmp

out = `strcmp` (*A*, *B*) [Method]

String comparison.

Tests whether each element in A is exactly equal to the corresponding element in B. Missing values are not considered equal to each other.

This does the same comparison as `A == B`, but is not polymorphic. Generally, there is no reason to use `strcmp` instead of `==` or `eq` on string arrays, unless you want to be compatible with cellstr inputs as well.

Returns logical array the size of the scalar expansion of A and B.

5.2.20.22 string.cmp

[*out*, *outA*, *outB*] = `cmp` (*A*, *B*) [Method]

Value ordering comparison, returning -1/0/+1.

Compares each element of *A* and *B*, returning for each element `i` whether `A(i)` was less than (-1), equal to (0), or greater than (1) the corresponding `B(i)`.

TODO: What to do about missing values? Should missings sort to the end (preserving total ordering over the full domain), or should their comparisons result in a fourth "null"/"undef" return value, probably represented by NaN? FIXME: The current implementation does not handle missings.

Returns a numeric array *out* of the same size as the scalar expansion of *A* and *B*. Each value in it will be -1, 0, or 1.

Also returns scalar-expanded copies of *A* and *B* as *outA* and *outB*, as a programming convenience.

5.2.20.23 string.missing

out = `string.missing` (*sz*) [Static Method]

Missing string value.

Creates a string array of all-missing values of the specified size *sz*. If *sz* is omitted, creates a scalar missing string.

Returns a string array of size *sz*.

5.2.20.24 string.decode

out = `string.decode` (*bytes*, *charsetName*) [Static Method]

Decode encoded text from bytes.

Decodes the given encoded text in *bytes* according to the specified encoding, given by *charsetName*.

Returns a scalar string.

See also: Section 5.2.20.9 [string.encode], page 19,

5.2.21 struct2table

out = `struct2table` (*s*) [Function]
out = `struct2table` (*...*, `'AsArray'`, *AsArray*) [Function]

Convert struct to a table.

Converts the input struct *s* to a `table`.

s may be a scalar struct or a nonscalar struct array.

The *AsArray* option is not implemented yet.

Returns a `table`.

5.2.22 table

`table` [Class]

Tabular data array containing multiple columnar variables.

A `table` is a tabular data structure that collects multiple parallel named variables. Each variable is treated like a column. (Possibly a multi-columned column, if that makes sense.) The types of variables may be heterogeneous.

A table object is like an SQL table or resultset, or a relation, or a DataFrame in R or Pandas.

A table is an array in itself: its size is *nrows*-by-*nvariables*, and you can index along the rows and variables by indexing into the table along dimensions 1 and 2.

`cellstr VariableNames` [Instance Variable of `table`]

The names of the variables in the table, as a cellstr row vector.

`cell VariableValues` [Instance Variable of `table`]

A cell vector containing the values for each of the variables. `VariableValues(i)` corresponds to `VariableNames(i)`.

`cellstr RowNames` [Instance Variable of `table`]

An optional list of row names that identify each row in the table. This is a cellstr column vector, if present.

5.2.22.1 table.table

obj = `table` () [Constructor]

Constructs a new empty (0 rows by 0 variables) table.

obj = `table` (*var1*, *var2*, *...*, *varN*) [Constructor]

Constructs a new table from the given variables. The variables passed as inputs to this constructor become the variables of the table. Their names are automatically detected from the input variable names that you used.

obj = `table` (`'Size'`, *sz*, `'VariableTypes'`, *varTypes*) [Constructor]

Constructs a new table of the given size, and with the given variable types. The variables will contain the default value for elements of that type.

`obj = table (..., 'VariableNames', varNames)` [Constructor]
`obj = table (..., 'RowNames', rowNames)` [Constructor]
Specifies the variable names or row names to use in the constructed table. Overrides the implicit names garnered from the input variable names.

5.2.22.2 table.summary

`summary (obj)` [Method]
`s = summary (obj)` [Method]
Summary of table's data.

Displays or returns a summary of data in the input table. This will contain some statistical information on each of its variables.

This method is not implemented yet.

5.2.22.3 table.prettyprint

`prettyprint (obj)` [Method]
Display table's values in tabular format. This prints the contents of the table in human-readable, tabular form.

Variables which contain objects are displayed using the strings returned by their `dispstrs` method, if they define one.

5.2.22.4 table.table2cell

`c = table2cell (obj)` [Method]
Converts table to a cell array. Each variable in *obj* becomes one or more columns in the output, depending on how many columns that variable has.

Returns a cell array with the same number of rows as *obj*, and with as many or more columns as *obj* has variables.

5.2.22.5 table.table2struct

`s = table2struct (obj)` [Method]
`s = table2struct (..., 'ToScalar', trueOrFalse)` [Method]
Converts *obj* to a scalar structure or structure array.

Row names are not included in the output struct. To include them, you must add them manually: s = table2struct (tbl, 'ToScalar', true); s.RowNames = tbl.Properties.RowNames;

Returns a scalar struct or struct array, depending on the value of the `ToScalar` option.

5.2.22.6 table.table2array

`s = table2struct (obj)` [Method]
Converts *obj* to a homogeneous array.

5.2.22.7 table.varnames

`out = varnames (obj)` [Method]
Get variable names for a table.

Returns cellstr.

5.2.22.8 table.istable

tf = `istable` (*obj*) [Method]
True if input is a table.

5.2.22.9 table.size

sz = `size` (*obj*) [Method]
Gets the size of a table.

For tables, the size is [number-of-rows x number-of-variables]. This is the same as `[height(obj), width(obj)]`.

5.2.22.10 table.length

out = `length` (*obj*) [Method]
Length along longest dimension

Use of `length` is not recommended. Use `numel` or `size` instead.

5.2.22.11 table.ndims

out = `ndims` (*obj*) [Method]
Number of dimensions

For tables, `ndims(obj)` is always 2.

5.2.22.12 table.squeeze

obj = `squeeze` (*obj*) [Method]
Remove singleton dimensions.

For tables, this is always a no-op that returns the input unmodified, because tables always have exactly 2 dimensions.

5.2.22.13 table.sizeof

out = `sizeof` (*obj*) [Method]
Approximate size of array in bytes. For tables, this returns the sume of `sizeof` for all of its variables' arrays, plus the size of the VariableNames and any other metadata stored in *obj*.

This is currently unimplemented.

5.2.22.14 table.height

out = `height` (*obj*) [Method]
Number of rows in table.

5.2.22.15 table.rows

out = `rows` (*obj*) [Method]
Number of rows in table.

5.2.22.16 table.width

out = `width` (*obj*) [Method]

Number of variables in table.

Note that this is not the sum of the number of columns in each variable. It is just the number of variables.

5.2.22.17 table.columns

out = `columns` (*obj*) [Method]

Number of variables in table.

Note that this is not the sum of the number of columns in each variable. It is just the number of variables.

5.2.22.18 table.numel

out = `numel` (*obj*) [Method]

Total number of elements in table.

This is the total number of elements in this table. This is calculated as the sum of numel for each variable.

NOTE: Those semantics may be wrong. This may actually need to be defined as `height(obj) * width(obj)`. The behavior of `numel` may change in the future.

5.2.22.19 table.isempty

out = `isempty` (*obj*) [Method]

Test whether array is empty.

For tables, `isempty` is true if the number of rows is 0 or the number of variables is 0.

5.2.22.20 table.ismatrix

out = `ismatrix` (*obj*) [Method]

Test whether array is a matrix.

For tables, `ismatrix` is always true, by definition.

5.2.22.21 table.isrow

out = `isrow` (*obj*) [Method]

Test whether array is a row vector.

5.2.22.22 table.iscol

out = `iscol` (*obj*) [Method]

Test whether array is a column vector.

For tables, `iscol` is true if the input has a single variable. The number of columns within that variable does not matter.

5.2.22.23 table.isvector

out = `isvector` (*obj*) [Method]
Test whether array is a vector.

5.2.22.24 table.isscalar

out = `isscalar` (*obj*) [Method]
Test whether array is scalar.

5.2.22.25 table.hasrownames

out = `hasrownames` (*obj*) [Method]
True if this table has row names defined.

5.2.22.26 table.vertcat

out = `vertcat` (*varargin*) [Method]
Vertical concatenation.

Combines tables by vertically concatenating them.

Inputs that are not tables are automatically converted to tables by calling table() on them.

The inputs must have the same number and names of variables, and their variable value types and sizes must be cat-compatible.

5.2.22.27 table.horzcat

out = `horzcat` (*varargin*) [Method]
Horizontal concatenation.

Combines tables by horizontally concatenating them. Inputs that are not tables are automatically converted to tables by calling table() on them. Inputs must have all distinct variable names.

Output has the same RowNames as `varargin{1}`. The variable names and values are the result of the concatenation of the variable names and values lists from the inputs.

5.2.22.28 table.repmat

out = `repmat` (*obj*, *sz*) [Method]
Replicate matrix.

Repmats a table by repmatting each of its variables vertically.

For tables, repmatting is only supported along dimension 1. That is, the values of sz(2:end) must all be exactly 1.

Returns a new table with the same variable names and types as tbl, but with a possibly different row count.

5.2.22.29 table.setVariableNames

`out = setVariableNames (obj, names)` [Method]

Set variable names.

Sets the `VariableNames` for this table to a new list of names.

names is a cellstr vector. It must have the same number of elements as the number of variables in *obj*.

5.2.22.30 table.setRowNames

`out = setRowNames (obj, names)` [Method]

Set row names.

Sets the row names on *obj* to *names*.

names is a cellstr column vector, with the same number of rows as *obj* has.

5.2.22.31 table.resolveVarRef

`[ixVar, varNames] = resolveVarREf (obj, varRef)` [Method]

Resolve a variable reference against this table.

A *varRef* is a numeric or char/cellstr indicator of which variables within *obj* are being referenced.

Returns: *ixVar* - the indexes of the variables in *obj* *varNames* - a cellstr of the names of the variables in *obj*

Raises an error if any of the specified variables could not be resolved.

5.2.22.32 table.subsetRows

`out = subsetRows (obj, ixRows)` [Method]

Subset table by rows.

Subsets this table by rows.

ixRows may be a numeric or logical index into the rows of *obj*.

5.2.22.33 table.subsetvars

`out = subsetvars (obj, ixVars)` [Method]

Subset table by variables.

Subsets table *obj* by subsetting it along its variables.

ixVars may be: - a numeric index vector - a logical index vector - `":"` - a cellstr vector of variable names

The resulting table will have its variables reordered to match ixVars.

5.2.22.34 table.removevars

`out = removevars (obj, vars)` [Method]

Remove variables from table.

Deletes the variables specified by *vars* from *obj*.

vars may be a char, cellstr, numeric index vector, or logical index vector.

5.2.22.35 table.movevars

`out = movevars (obj, vars, relLocation, location)` [Method]

Move around variables in a table.

vars is a list of variables to move, specified by name or index.

relLocation is `'Before'` or `'After'`.

location indicates a single variable to use as the target location, specified by name or index. If it is specified by index, it is the index into the list of *unmoved* variables from *obj*, not the original full list of variables in *obj*.

Returns a table with the same variables as *obj*, but in a different order.

5.2.22.36 table.setvar

`out = setvar (obj, varRef, value)` [Method]

Set value for a variable in table.

This sets (replaces) the value for a variable that already exists in *obj*. It cannot be used to add a new variable.

5.2.22.37 table.convertvars

`out = convertvars (obj, vars, dataType)` [Method]

Convert variables to specified data type.

Converts the variables in *obj* specified by *vars* to the specified data type.

vars is a cellstr or numeric vector specifying which variables to convert.

dataType specifies the data type to convert those variables to. It is either a char holding the name of the data type, or a function handle which will perform the conversion. If it is the name of the data type, there must either be a one-arg constructor of that type which accepts the specified variables' current types as input, or a conversion method of that name defined on the specified variables' current type.

Returns a table with the same variable names as *obj*, but with converted types.

5.2.22.38 table.head

`out = head (obj)` [Method]
`out = head (obj, k)` [Method]

Get first K rows of table.

Returns the first *k* rows of *obj*, as a table.

k defaults to 8.

If there are less than *k* rows in *obj*, returns all rows.

5.2.22.39 table.tail

`out = tail (obj)` [Method]
`out = tail (obj, k)` [Method]

Get last K rows of table.

Returns the last *k* rows of *obj*, as a table.

k defaults to 8.

If there are less than *k* rows in *obj*, returns all rows.

5.2.22.40 table.join

`[C, ib] = join (A, B)` [Method]
`[C, ib] = join (A, B, ...)` [Method]

Combine two tables by rows using key variables, in a restricted form.

This is not a "real" relational join operation. It has the restrictions that: 1) The key values in B must be unique. 2) Every key value in A must map to a key value in B. These are restrictions inherited from the Matlab definition of table.join.

You probably don't want to use this method. You probably want to use innerjoin or outerjoin instead.

See also: Section 5.2.22.41 [table.innerjoin], page 30, Section 5.2.22.42 [table.outerjoin], page 30,

5.2.22.41 table.innerjoin

`[out, ixa, ixb] = innerjoin (A, B)` [Method]
`[...] = innerjoin (A, B, ...)` [Method]

Combine two tables by rows using key variables.

Computes the relational inner join between two tables. "Inner" means that only rows which had matching rows in the other input are kept in the output.

TODO: Document options.

Returns: *out* - A table that is the result of joining A and B *ix* - Indexes into A for each row in out *ixb* - Indexes into B for each row in out

5.2.22.42 table.outerjoin

`[out, ixa, ixb] = outerjoin (A, B)` [Method]
`[...] = outerjoin (A, B, ...)` [Method]

Combine two tables by rows using key variables, retaining unmatched rows.

Computes the relational outer join of tables A and B. This is like a regular join, but also includes rows in each input which did not have matching rows in the other input; the columns from the missing side are filled in with placeholder values.

TODO: Document options.

Returns: *out* - A table that is the result of the outer join of A and B *ixa* - indexes into A for each row in out *ixb* - indexes into B for each row in out

5.2.22.43 table.outerfillvals

`out = outerfillvals (obj)` [Method]

Get fill values for outer join.

Returns a table with the same variables as this, but containing only a single row whose variable values are the values to use as fill values when doing an outer join.

5.2.22.44 table.semijoin

`[outA, ixA, outB, ixB] = semijoin (A, B)` [Method]

Natural semijoin.

Computes the natural semijoin of tables A and B. The semi-join of tables A and B is the set of all rows in A which have matching rows in B, based on comparing the values of variables with the same names.

This method also computes the semijoin of B and A, for convenience.

Returns: *outA* - all the rows in A with matching row(s) in B *ixA* - the row indexes into A which produced *outA* *outB* - all the rows in B with matching row(s) in A *ixB* - the row indexes into B which produced *outB*

5.2.22.45 table.antijoin

`[outA, ixA, outB, ixB] = antijoin (A, B)` [Method]

Natural antijoin (AKA "semidifference").

Computes the anti-join of A and B. The anti-join is defined as all the rows from one input which do not have matching rows in the other input.

Returns: *outA* - all the rows in A with no matching row in B *ixA* - the row indexes into A which produced *outA* *outB* - all the rows in B with no matching row in A *ixB* - the row indexes into B which produced *outB*

5.2.22.46 table.cartesian

`[out, ixs] = cartesian (A, B)` [Method]

Cartesian product of two tables.

Computes the Cartesian product of two tables. The Cartesian product is each row in A combined with each row in B.

Due to the definition and structural constraints of table, the two inputs must have no variable names in common. It is an error if they do.

The Cartesian product is seldom used in practice. If you find yourself calling this method, you should step back and re-evaluate what you are doing, asking yourself if that is really what you want to happen. If nothing else, writing a function that calls cartesian() is usually much less efficient than alternate ways of arriving at the same result.

This implementation does not remove duplicate values. TODO: Determine whether this duplicate-removing behavior is correct.

The ordering of the rows in the output is not specified, and may be implementation-dependent. TODO: Determine if we can lock this behavior down to a fixed, defined ordering, without killing performance.

5.2.22.47 table.groupby

`[out] = groupby (obj, groupvars, aggcalcs)` [Method]

Find groups in table data and apply functions to variables within groups.

This works like an SQL `"SELECT ... GROUP BY ..."` statement.

groupvars (cellstr, numeric) is a list of the grouping variables, identified by name or index.

aggcalcs is a specification of the aggregate calculations to perform on them, in the form {*out_var*, *fcn*, *in_vars*; ...}, where: *out_var* (char) is the name of the output variable *fcn* (function handle) is the function to apply to produce it *in_vars* (cellstr) is a list of the input variables to pass to fcn

Returns a table.

5.2.22.48 table.grpstats

`[out] = grpstats (obj, groupvar)` [Method]
`[out] = grpstats (..., 'DataVars', DataVars)` [Method]

Statistics by group.

See also: Section 5.2.22.47 [table.groupby], page 31.

5.2.22.49 table.union

`[C, ia, ib] = union (A, B)` [Method]

Set union.

Computes the union of two tables. The union is defined to be the unique row values which are present in either of the two input tables.

Returns: *C* - A table containing all the unique row values present in A or B. *ia* - Row indexes into A of the rows from A included in C. *ib* - Row indexes into B of the rows from B included in C.

5.2.22.50 table.intersect

`[C, ia, ib] = intersect (A, B)` [Method]

Set intersection.

Computes the intersection of two tables. The intersection is defined to be the unique row values which are present in both of the two input tables.

Returns: *C* - A table containing all the unique row values present in both A and B. *ia* - Row indexes into A of the rows from A included in C. *ib* - Row indexes into B of the rows from B included in C.

5.2.22.51 table.setxor

`[C, ia, ib] = setxor (A, B)` [Method]

Set exclusive OR.

Computes the setwise exclusive OR of two tables. The set XOR is defined to be the unique row values which are present in one or the other of the two input tables, but not in both.

Returns: *C* - A table containing all the unique row values in the set XOR of A and B. *ia* - Row indexes into A of the rows from A included in C. *ib* - Row indexes into B of the rows from B included in C.

5.2.22.52 table.setdiff

`[C, ia] = setdiff (A, B)` [Method]

Set difference.

Computes the set difference of two tables. The set difference is defined to be the unique row values which are present in table A that are not in table B.

Returns: *C* - A table containing the unique row values in A that were not in B. *ia* - Row indexes into A of the rows from A included in C.

5.2.22.53 table.ismember

`[tf, loc] = ismember (A, B)` [Method]

Set membership.

Finds rows in A that are members of B.

Returns: *tf* - A logical vector indicating whether each A(i,:) was present in B. *loc* - Indexes into B of rows that were found.

5.2.22.54 table.ismissing

`out = ismissing (obj)` [Method]
`out = ismissing (obj, indicator)` [Method]

Find missing values.

Finds missing values in *obj*'s variables.

If indicator is not supplied, uses the standard missing values for each variable's data type. If indicator is supplied, the same indicator list is applied across all variables.

All variables in this must be vectors. (This is due to the requirement that `size(out) == size(obj)`.)

Returns a logical array the same size as *obj*.

5.2.22.55 table.rmmissing

`[out, tf] = rmmissing (obj)` [Method]
`[out, tf] = rmmissing (obj, indicator)` [Method]
`[out, tf] = rmmissing (..., 'DataVariables', vars)` [Method]
`[out, tf] = rmmissing (..., 'MinNumMissing', minNumMissing)` [Method]

Remove rows with missing values.

Removes the rows from *obj* that have missing values.

If the 'DataVariables' option is given, only the data in the specified variables is considered.

Returns: *out* - A table the same as *obj*, but with rows with missing values removed. *tf* - A logical index vector indicating which rows were removed.

5.2.22.56 table.standardizeMissing

`out = standardizeMissing (obj, indicator)` [Method]
`out = standardizeMissing (..., 'DataVariables', vars)` [Method]

Insert standard missing values.

Standardizes missing values in variable data.

If the *DataVariables* option is supplied, only the indicated variables are standardized.

indicator is passed along to `standardizeMissing` when it is called on each of the data variables in turn. The same indicator is used for all variables. You can mix and match indicator types by just passing in mixed indicator types in a cell array; indicators that don't match the type of the column they are operating on are just ignored.

Returns a table with same variable names and types as *obj*, but with variable values standardized.

5.2.22.57 table.varfun

`out = varfun (fcn, obj)` [Method]
`out = varfun (..., 'OutputFormat', outputFormat)` [Method]
`out = varfun (..., 'InputVariables', vars)` [Method]
`out = varfun (..., 'ErrorHandler', errorFcn)` [Method]

Apply function to table variables.

Applies the given function *fcn* to each variable in *obj*, collecting the output in a table, cell array, or array of another type.

5.2.22.58 table.rowfun

`out = varfun (fcn, obj)` [Method]
`out = varfun (..., 'OptionName', OptionValue, ...)` [Method]

This method is currently unimplemented. Sorry.

5.2.22.59 table.findgroups

`[G, TID] = findgroups (obj)` [Method]

Find groups within a table's row values.

Finds groups within a table's row values and get group numbers. A group is a set of rows that have the same values in all their variable elements.

Returns: *G* - A double column vector of group numbers created from *obj*. *TID* - A table containing the row values corresponding to the group numbers.

5.2.22.60 table.evalWithVars

`out = evalWithVars (obj, expr)` [Method]

Evaluate an expression against table's variables.

Evaluates the M-code expression *expr* in a workspace where all of *obj*'s variables have been assigned to workspace variables.

expr is a charvec containing an Octave expression.

As an implementation detail, the workspace will also contain some variables that are prefixed and suffixed with `"__"`. So try to avoid those in your table variable names.

Returns the result of the evaluation.

Examples:

```
[s,p,sp] = table_examples.SpDb
tmp = join (sp, p);
shipment_weight = evalWithVars (tmp, "Qty .* Weight")
```

5.2.22.61 table.restrict

out = `restrict` (*obj*, *expr*) [Method]
out = `restrict` (*obj*, *ix*) [Method]

Subset rows using variable expression or index.

Subsets a table row-wise, using either an index vector or an expression involving *obj*'s variables.

If the argument is a numeric or logical vector, it is interpreted as an index into the rows of this. (Just as with 'subsetRows (this, index)'.)

If the argument is a char, then it is evaulated as an M-code expression, with all of this' variables available as workspace variables, as with `evalWithVars`. The output of expr must be a numeric or logical index vector (This form is a shorthand for `out = subsetRows (this, evalWithVars (this, expr))`.)

TODO: Decide whether to name this to "where" to be more like SQL instead of relational algebra.

Examples:

```
[s,p,sp] = table_examples.SpDb;
prettyprint (restrict (p, 'Weight >= 14 & strcmp(Color, "Red")'))
```

5.2.23 tableOuterFillValue

out = `tableOuterFillValue` (*x*) [Function]

Outer fill value for variable within a table.

Determines the fill value to usse for a given variable value *x* when that value is used as a variable in a table that is involved in an outer join.

The default implementation for `tableOuterFillValue` has support for all Octave primitive types, plus cellstrs, datetime & friends, strings, and `table`-valued variables.

This function may become private to table before version 1.0. It is currently global to make debugging more convenient. It (or an equivalent) will remain global if we want to allow user-defined classes to customize their fill value. It also has default logic that will determine the fill value for an arbitrary type by detecting the value used to fill elements during array expansion operations. This will be appropriate for most data types.

Returns a 1-by-ncols value of the same type as x, which may be any type, where ncols is the number of columns in the input.

5.2.24 vecfun

out = `vecfun` (*fcn*, *x*, *dim*) [Function]

Apply function to vectors in array along arbitrary dimension.

This function is not implemented yet.

Applies a given function to the vector slices of an N-dimensional array, where those slices are along a given dimension.

fcn is a function handle to apply.

x is an array of arbitrary type which is to be sliced and passed in to *fcn*.

dim is the dimension along which the vector slices lay.

Returns the collected output of the *fcn* calls, which will be the same size as *x*, but not necessarily the same type.

6 Copying

6.1 Package Copyright

Tablicious for Octave is covered by the GNU GPLv3.

All the code in the package is GNU GPLv3.

The Fisher Iris dataset is Public Domain.

6.2 Manual Copyright

This manual is for Tablicious, version 0.1.0.