Tablicious for Octave

version 0.1.0, April 2019

Andrew Janke

This manual is for Tablicious, version 0.1.0.

Short Contents

Table of Contents

1 Introduction

> Time is an illusion. Lunchtime doubly so.
> —*Douglas Adams*

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

Tablicious provides Matlab-compatible tabular data and date/time support for GNU Octave. This includes a `table` class with support for filtering and join operations; `datetime`, `duration`, and related classes; Missing Data support; `string` and `categorical` data types; and other miscellaneous things.

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

Tablicious's classes are designed to be convenient to use while still being efficient. The data representations used by Tablicious are designed to be efficient and suitable for working with large-ish data sets. A "large-ish" data set is one that can have millions of elements or rows, but still fits in main computer memory. Tablicious's main relational and arithmetic operations are all implemented using vectorized operations on primitive Octave data types.

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 Date Representation

Chrono provides the `datetime` class for representing points in time.

4.1 `datetime` Class

A `datetime` is an array object that represents points in time in the familiar Gregorian calendar.

This is an attempt to reproduce the functionality of Matlab's `datetime`. It also contains some Octave-specific extensions.

The underlying representation is that of a datenum (a `double` containing the number of days since the Matlab epoch), but encapsulating it in an object provides several benefits: friendly human-readable display, type safety, automatic type conversion, and time zone support. In addition to the underlying datenum array, a `datetime` inclues an optional `TimeZone` property indicating what time zone the datetimes are in.

4.1.1 `datenum` Compatibility

While the underlying data representation of `datetime` is compatible with (in fact, identical to) that of datenums, you cannot directly combine them via assignment, concatenation, or most arithmetic operations.

This is because of the signature of the `datetime` constructor. When combining objects and primitive types like `double`, the primitive type is promoted to an object by calling the other object's one-argument constructor on it. However, the one-argument numeric-input consstructor for `datetime` does not accept datenums: it interprets its input as datevecs instead. This is due to a design decision on Matlab's part; for compatibility, Octave does not alter that interface.

To combine `datetime`s with datenums, you can convert the datenums to `datetime`s by calling `datetime.ofDatenum` or `datetime(x, 'ConvertFrom', 'datenum')`, or you can convert the `datetime`s to datenums by accessing its `dnums` field with `x.dnums`.

Examples:

```
dt = datetime('2011-03-04')
dn = datenum('2017-01-01')
[dt dn]
    ⇒ error: datenum: expected date vector containing [YEAR, MONTH, DAY, HOUR, MINUTE
[dt datetime.ofDatenum(dn)]
    ⇒ 04-Mar-2011   01-Jan-2017
```

Also, if you have a zoned `datetime`, you can't combine it with a datenum, because datenums do not carry time zone information.

5 Time Zones

Chrono has support for representing dates in time zones and for converting between time zones.

A `datetime` may be "zoned" or "zoneless". A zoneless `datetime` does not have a time zone associated with it. This is represented by an empty `TimeZone` property on the `datetime` object. A zoneless `datetime` represents the local time in some unknown time zone, and assumes a continuous time scale (no DST shifts).

A zoned `datetime` is associated with a time zone. It is represented by having the time zone's IANA zone identifier (e.g. `'UTC'` or `'America/New_York'`) in its `TimeZone` property. A zoned `datetime` represents the local time in that time zone.

By default, the `datetime` constructor creates unzoned `datetime`s. To make a zoned `datetime`, either pass the `'TimeZone'` option to the constructor, or set the `TimeZone` property after object creation. Setting the `TimeZone` property on a zoneless `datetime` declares that it's a local time in that time zone. Setting the `TimeZone` property on a zoned `datetime` turns it back into a zoneless `datetime` without changing the local time it represents.

You can tell a zoned from a zoneless time zone in the object display because the time zone is included for zoned `datetime`s.

```
% Create an unzoned datetime
d = datetime('2011-03-04 06:00:00')
    ⇒  04-Mar-2011 06:00:00

% Create a zoned datetime
d_ny = datetime('2011-03-04 06:00:00', 'TimeZone', 'America/New_York')
    ⇒  04-Mar-2011 06:00:00 America/New_York
% This is equivalent
d_ny = datetime('2011-03-04 06:00:00');
d_ny.TimeZone = 'America/New_York'
    ⇒  04-Mar-2011 06:00:00 America/New_York

% Convert it to Chicago time
d_chi.TimeZone = 'America/Chicago'
    ⇒  04-Mar-2011 05:00:00 America/Chicago
```

When you combine two zoned `datetime`s via concatenation, assignment, or arithmetic, if their time zones differ, they are converted to the time zone of the left-hand input.

```
d_ny = datetime('2011-03-04 06:00:00', 'TimeZone', 'America/New_York')
d_la = datetime('2011-03-04 06:00:00', 'TimeZone', 'America/Los_Angeles')
d_la - d_ny
    ⇒ 03:00:00
```

You cannot combine a zoned and an unzoned `datetime`. This results in an error being raised.

> **Warning:** Normalization of "nonexistent" times (like between 02:00 and 03:00 on a "spring forward" DST change day) is not implemented yet. The results of converting a zoneless local time into a time zone where that local time did not exist are currently undefined.

5.1 Defined Time Zones

Chrono's time zone data is drawn from the IANA Time Zone Database (`https://www.iana.org/time-zones`), also known as the "Olson Database". Chrono includes a copy of this database in its distribution so it can work on Windows, which does not supply it like Unix systems do.

You can use the `timezones` function to list the time zones known to Chrono. These will be all the time zones in the IANA database on your system (for Linux and macOS) or in the IANA time zone database redistributed with Chrono (for Windows).

> **Note:** The IANA Time Zone Database only covers dates from about the year 1880 to 2038. Converting time zones for `datetime`s outside that range is currently unimplemented. (Chrono needs to add support for proleptic POSIX time zone rules, which are used to govern behavior outside that date range.)

6 Durations

6.1 `duration` Class

A `duration` represents a period of time in fixed-length seconds (or minutes, hours, or whatever you want to measure it in.)

A `duration` has a resolution of about a nanosecond for typical dates. The underlying representation is a `double` representing the number of days elapsed, similar to a datenum, except it's interpreted as relative to some other reference point you provide, instead of being relative to the Matlab/Octave epoch.

You can add or subtract a `duration` to a `datetime` to get another `datetime`. You can also add or subtract `durations` to each other.

6.2 `calendarDuration` Class

A `calendarDuration` represents a period of time in variable-length calendar components. For example, years and months can have varying numbers of days, and days in time zones with Daylight Saving Time have varying numbers of hours. A `calendarDuration` does arithmetic with "whole" calendar periods.

`calendarDuration`s and `duration`s cannot be directly combined, because they are not semantically equivalent. (This may be relaxed in the future to allow `duration`s to be interpreted as numbers of days when combined with `calendarDuration`s.)

```
d = datetime('2011-03-04 00:00:00')
    ⇒ 04-Mar-2011
cdur = calendarDuration(1, 3, 0)
    ⇒ 1y 3mo
d2 = d + cdur
    ⇒ 04-Jun-2012
```

7 Validation Functions

Tablicious provides several validation functions which can be used to check properties of function arguments, variables, object properties, and other expressions. These can be used to express invariants in your program and catch problems due to input errors, incorrect function usage, or other bugs.

These validation functions are named following the pattern `mustBeXxx`, where `Xxx` is some property of the input it is testing. Validation functions may check the type, size, or other aspects of their inputs.

The most common place for validation functions to be used will probably be at the beginning of functions, to check the input arguments and ensure that the contract of the function is not being violated. If in the future Octave gains the ability to declaratively express object property constraints, they will also be of use there.

Be careful not to get too aggressive with the use of validation functions: while using them can make sure invariants are followed and your program is correct, they also reduce the code's ability to make use of duck typing, reducing its flexibility. Whether you want to make this trade-off is a design decision you will have to consider.

When a validation function's condition is violated, it raises an error that includes a description of the violation in the error message. This message will include a label for the input that describes what is being tested. By default, this label is initialized with `inputname()`, so when you are calling a validator on a function argument or variable, you will generally not need to supply a label. But if you're calling it on an object property or an expression more complex than a simple variable reference, the validator cannot automatically detect the input name for use in the label. In this case, make use of the optional trailing argument(s) to the functions to manually supply a label for the value being tested.

```
% Validation of a simple variable does not need a label
mustBeScalar (x);
% Validation of a field or property reference does need a label
mustBeScalar (this.foo, 'this.foo');
```

8 Missing Functionality

Tablicious is based on Matlab's table and date/time APIs and supports most of their 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()`
- POSIX time zone support for years outside the IANA time zone database coverage
- Week-of-year (ISO calendar) calculations
- Various `'ConvertFrom'` forms for `datetime` and `duration`
- Support for LDML formatting for `datetime`
- Various functions: `between`, `caldiff`, `dateshift`, `week`
- `isdst`, `isweekend`
- `calendarDuration.split`
- `duration.Format` support
- `UTCOffset` and `DSTOffset` fields in the output of `timezones()`
- Plotting support

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

9 API Reference

9.1 API by Category

9.1.1 Tables

Section 9.2.55 [table], page 53
Tabular data array containing multiple columnar variables.

Section 9.2.1 [array2table], page 13
Convert an array to a table.

Section 9.2.6 [cell2table], page 20
Convert a cell array to a table.

Section 9.2.54 [struct2table], page 53
Convert struct to a table.

Section 9.2.56 [tableOuterFillValue], page 67
Outer fill value for variable within a table.

Section 9.2.58 [vartype], page 68
Filter by variable type for use in suscripting.

9.1.2 Strings and Categoricals

Section 9.2.53 [string], page 47
A string array of Unicode strings.

Section 9.2.52 [startsWith], page 47
Test if strings start with a pattern.

Section 9.2.14 [endsWith], page 31
Test if strings end with a pattern.

Section 9.2.8 [contains], page 20
Test if strings contain a pattern.

Section 9.2.5 [categorical], page 16
Categorical variable array.

Section 9.2.11 [discretize], page 28
Group data into discrete bins or categories.

9.1.3 Dates and Times

Section 9.2.9 [datetime], page 21
Represents points in time using the Gregorian calendar.

Section 9.2.24 [localdate], page 34
Represents a complete day using the Gregorian calendar.

Section 9.2.18 [isdatetime], page 32
True if input is a 'datetime' array, false otherwise.

Section 9.2.41 [NaT], page 40
: "Not-a-Time".

Section 9.2.2 [calendarDuration], page 13
: Durations of time using variable-length calendar periods, such as days, months, and years, which may vary in length over time.

Section 9.2.3 [calmonths], page 15
: Create a 'calendarDuration' that is a given number of calendar months long.

Section 9.2.4 [calyears], page 16
: Construct a 'calendarDuration' a given number of years long.

Section 9.2.10 [days], page 28
: Duration in days.

Section 9.2.13 [duration], page 28
: Represents durations or periods of time as an amount of fixed-length time (i.e.

Section 9.2.17 [hours], page 31
: Create a 'duration' X hours long, or get the hours in a 'duration' X.

Section 9.2.19 [isduration], page 32
: True if input is a 'duration' array, false otherwise.

Section 9.2.25 [milliseconds], page 36
: Create a 'duration' X milliseconds long, or get the milliseconds in a 'duration' X.

Section 9.2.26 [minutes], page 36
: Create a 'duration' X hours long, or get the hours in a 'duration' X.

Section 9.2.49 [seconds], page 46
: Create a 'duration' X seconds long, or get the seconds in a 'duration' X.

Section 9.2.57 [timezones], page 67
: List all the time zones defined on this system.

Section 9.2.60 [years], page 68
: Create a 'duration' X years long, or get the years in a 'duration' X.

9.1.4 Missing Data

Section 9.2.16 [fillmissing], page 31
: Fill missing values.

Section 9.2.22 [ismissing], page 33
: Find missing values.

Section 9.2.47 [rmmissing], page 46
: Remove missing values.

Section 9.2.51 [standardizeMissing], page 46
: Insert standard missing values.

Section 9.2.27 [missing], page 36
: Generic auto-converting missing value.

Section 9.2.23 [isnannish], page 33
Test if elements are NaN or NaN-like

Section 9.2.15 [eqn], page 31
Determine element-wise equality, treating NaNs as equal

9.1.5 Validation Functions

Section 9.2.28 [mustBeA], page 37
Requires that input is of a given type.

Section 9.2.29 [mustBeCellstr], page 38
Requires that input is a cellstr.

Section 9.2.30 [mustBeCharvec], page 38
Requires that input is a char row vector.

Section 9.2.31 [mustBeFinite], page 38
Requires that input is finite.

Section 9.2.32 [mustBeInteger], page 38
Requires that input is integer-valued (but not necessarily integer-typed).

Section 9.2.33 [mustBeMember], page 38
Requires that input is a member of a set of given valid values.

Section 9.2.34 [mustBeNonempty], page 39
Requires that input is nonempty.

Section 9.2.35 [mustBeNumeric], page 39
Requires that input is numeric.

Section 9.2.36 [mustBeReal], page 39
Requires that input is finite.

Section 9.2.37 [mustBeSameSize], page 39
Requires that the inputs are the same size.

Section 9.2.38 [mustBeScalar], page 40
Requires that input is scalar.

Section 9.2.39 [mustBeScalarLogical], page 40
Requires that input is a scalar logical.

Section 9.2.40 [mustBeVector], page 40
Requires that input is a vector or empty.

9.1.6 Miscellaneous

Section 9.2.7 [colvecfun], page 20
Apply a function to column vectors in array.

Section 9.2.12 [dispstrs], page 28
Display strings for array.

Section 9.2.20 [isfile], page 32
Test whether file exists and is not a folder.

9.1.7 Example Datasets

9.1.8 Uncategorized

9.2 API Alphabetically

9.2.1 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 9.2.6 [cell2table], page 20, Section 9.2.55 [table], page 53, Section 9.2.54 [struct2table], page 53,

9.2.2 calendarDuration

`calendarDuration` [Class]

Durations of time using variable-length calendar periods, such as days, months, and years, which may vary in length over time. (For example, a calendar month may have 28, 30, or 31 days.)

`char Sign` [Instance Variable of `calendarDuration`]
The sign (1 or -1) of this duration, which indicates whether it is a positive or negative span of time.

`char Years` [Instance Variable of `calendarDuration`]
The number of whole calendar years in this duration. Must be integer-valued.

`char Months` [Instance Variable of `calendarDuration`]
The number of whole calendar months in this duration. Must be integer-valued.

`char Days` [Instance Variable of `calendarDuration`]
The number of whole calendar days in this duration. Must be integer-valued.

`char Hours` [Instance Variable of `calendarDuration`]
The number of whole hours in this duration. Must be integer-valued.

`char Minutes` [Instance Variable of `calendarDuration`]
The number of whole minutes in this duration. Must be integer-valued.

`char Seconds` [Instance Variable of `calendarDuration`]
The number of seconds in this duration. May contain fractional values.

`char Format` [Instance Variable of `calendarDuration`]
The format to display this `calendarDuration` in. Currently unsupported.

This is a single value that applies to the whole array.

9.2.2.1 calendarDuration.calendarDuration

obj = `calendarDuration` () [Constructor]
Constructs a new scalar `calendarDuration` of zero elapsed time.

obj = `calendarDuration` (*Y*, *M*, *D*) [Constructor]
obj = `calendarDuration` (*Y*, *M*, *D*, *H*, *MI*, *S*) [Constructor]
Constructs new `calendarDuration` arrays based on input values.

9.2.2.2 calendarDuration.sizeof

out = `sizeof` (*obj*) [Method]
Size of array in bytes.

9.2.2.3 calendarDuration.isnat

out = `isnat` (*obj*) [Method]
True if input elements are NaT.

Returns logical array the same size as *obj*.

9.2.2.4 calendarDuration.uminus

out = `uminus` (*obj*) [Method]
Unary minus. Negates the sign of *obj*.

9.2.2.5 calendarDuration.plus

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

Addition: add two `calendarDuration`s.

All the calendar elements (properties) of the two inputs are added together. No normalization is done across the elements, aside from the normalization of NaNs.

If *B* is numeric, it is converted to a `calendarDuration` using `calendarDuration.ofDays`.

Returns a `calendarDuration`.

9.2.2.6 calendarDuration.times

out = `times` (*obj*, *B*) [Method]

Multiplication: Multiplies a `calendarDuration` by a numeric factor.

Returns a `calendarDuration`.

9.2.2.7 calendarDuration.minus

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

Subtraction: Subtracts one `calendarDuration` from another.

Returns a `calendarDuration`.

9.2.2.8 calendarDuration.dispstrs

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

Get display strings for each element of *obj*.

Returns a cellstr the same size as *obj*.

9.2.2.9 calendarDuration.isnan

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

True if input elements are NaT. This is just an alias for `isnat`, provided for compatibility and polymorphic programming purposes.

Returns logical array the same size as *obj*.

9.2.3 calmonths

out = `calmonths` (*x*) [Function File]

Create a `calendarDuration` that is a given number of calendar months long.

Input *x* is a numeric array specifying the number of calendar months.

This is a shorthand alternative to calling the `calendarDuration` constructor with `calendarDuration(0, x, 0)`.

Returns a new `calendarDuration` object of the same size as *x*.

See Section 9.2.2 [calendarDuration], page 13.

9.2.4 calyears

out = calyears (*x*) [Function]

Construct a `calendarDuration` a given number of years long.

This is a shorthand for calling `calendarDuration(x, 0, 0)`.

See Section 9.2.2 [calendarDuration], page 13.

9.2.5 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.

9.2.5.1 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.

9.2.5.2 categorical.sizeof

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

Size of array in bytes.

9.2.5.3 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.

9.2.5.4 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*.

9.2.5.5 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.

9.2.5.6 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*.

9.2.5.7 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*.

9.2.5.8 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*.

9.2.5.9 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.

9.2.5.10 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*.

9.2.5.11 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*.

9.2.5.12 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}`.

9.2.5.13 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.

9.2.5.14 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.

9.2.5.15 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.

9.2.5.16 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*.

9.2.5.17 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*.

9.2.5.18 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*.

9.2.5.19 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.

9.2.6 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 9.2.1 [array2table], page 13, Section 9.2.55 [table], page 53, Section 9.2.54 [struct2table], page 53,

9.2.7 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.

9.2.8 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*.

9.2.9 datetime

`datetime` [Class]

Represents points in time using the Gregorian calendar.

The underlying values are doubles representing the number of days since the Matlab epoch of `"January 0, year 0"`. This has a precision of around nanoseconds for typical times.

A `datetime` array is an array of date/time values, with each element holding a complete date/time. The overall array may also have a TimeZone and a Format associated with it, which apply to all elements in the array.

This is an attempt to reproduce the functionality of Matlab's `datetime`. It also contains some Octave-specific extensions.

`double dnums` [Instance Variable of `datetime`]

The underlying datenums that represent the points in time. These are always in UTC.

This is a planar property: the size of `dnums` is the same size as the containing `datetime` array object.

`char TimeZone` [Instance Variable of `datetime`]

The time zone this `datetime` array is in. Empty if this does not have a time zone associated with it ("unzoned"). The name of an IANA time zone if this does.

Setting the `TimeZone` of a `datetime` array changes the time zone it is presented in for strings and broken-down times, but does not change the underlying UTC times that its elements represent.

`char Format` [Instance Variable of `datetime`]

The format to display this `datetime` in. Currently unsupported.

9.2.9.1 datetime.datetime

obj `= datetime ()` [Constructor]

Constructs a new scalar `datetime` containing the current local time, with no time zone attached.

obj `= datetime (`*datevec*`)` [Constructor]
obj `= datetime (`*datestrs*`)` [Constructor]
obj `= datetime (`*in*`, 'ConvertFrom',` *inType*`)` [Constructor]
obj `= datetime (`*Y*`,` *M*`,` *D*`,` *H*`,` *MI*`,` *S*`)` [Constructor]
obj `= datetime (`*Y*`,` *M*`,` *D*`,` *H*`,` *MI*`,` *MS*`)` [Constructor]

obj = `datetime` (..., 'Format', *Format*, 'InputFormat', *InputFormat*, 'Locale', *InputLocale*, 'PivotYear', *PivotYear*, 'TimeZone', *TimeZone*) [Constructor]

Constructs a new `datetime` array based on input values.

9.2.9.2 datetime.ofDatenum

obj = `datetime.ofDatenum` (*dnums*) [Static Method]

Converts a datenum array to a datetime array.

Returns an unzoned `datetime` array of the same size as the input.

9.2.9.3 datetime.ofDatestruct

obj = `datetime.ofDatestruct` (*dstruct*) [Static Method]

Converts a datestruct to a datetime array.

A datestruct is a special struct format used by Chrono that has fields Year, Month, Day, Hour, Minute, and Second. It is not a standard Octave datatype.

Returns an unzoned `datetime` array.

9.2.9.4 datetime.NaT

out = `datetime.NaT` () [Static Method]
out = `datetime.NaT` (*sz*) [Static Method]

"Not-a-Time": Creates NaT-valued arrays.

Constructs a new `datetime` array of all `NaT` values of the given size. If no input *sz* is given, the result is a scalar `NaT`.

`NaT` is the `datetime` equivalent of `NaN`. It represents a missing or invalid value. `NaT` values never compare equal to, greater than, or less than any value, including other `NaT`s. Doing arithmetic with a `NaT` and any other value results in a `NaT`.

9.2.9.5 datetime.posix2datenum

dnums = `datetime.posix2datenum` (*pdates*) [Static Method]

Converts POSIX (Unix) times to datenums

Pdates (numeric) is an array of POSIX dates. A POSIX date is the number of seconds since January 1, 1970 UTC, excluding leap seconds. The output is implicitly in UTC.

9.2.9.6 datetime.datenum2posix

out = `datetime.datenum2posix` (*dnums*) [Static Method]

Converts Octave datenums to Unix dates.

The input datenums are assumed to be in UTC.

Returns a double, which may have fractional seconds.

9.2.9.7 datetime.sizeof

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

Size of array in bytes.

9.2.9.8 datetime.proxyKeys

`[keysA, keysB] = proxyKeys (a, b)` [Method]

Computes proxy key values for two datetime arrays. Proxy keys are numeric values whose rows have the same equivalence relationships as the elements of the inputs.

This is primarily for Chrono's internal use; users will typically not need to call it or know how it works.

Returns two 2-D numeric matrices of size n-by-k, where n is the number of elements in the corresponding input.

9.2.9.9 datetime.ymd

`[y, m, d] = ymd (obj)` [Method]

Get the Year, Month, and Day components of *obj*.

For zoned `datetime`s, these will be local times in the associated time zone.

Returns double arrays the same size as `obj`.

9.2.9.10 datetime.hms

`[h, m, s] = hms (obj)` [Method]

Get the Hour, Minute, and Second components of a *obj*.

For zoned `datetime`s, these will be local times in the associated time zone.

Returns double arrays the same size as `obj`.

9.2.9.11 datetime.ymdhms

`[y, m, d, h, mi, s] = ymdhms (obj)` [Method]

Get the Year, Month, Day, Hour, Minute, and Second components of a *obj*.

For zoned `datetime`s, these will be local times in the associated time zone.

Returns double arrays the same size as `obj`.

9.2.9.12 datetime.timeofday

`out = timeofday (obj)` [Method]

Get the time of day (elapsed time since midnight).

For zoned `datetime`s, these will be local times in the associated time zone.

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

9.2.9.13 datetime.week

`out = week (obj)` [Method]

Get the week of the year.

This method is unimplemented.

9.2.9.14 datetime.dispstrs

`out = dispstrs (obj)` [Method]

Get display strings for each element of *obj*.

Returns a cellstr the same size as *obj*.

9.2.9.15 datetime.datestr

out = `datestr` (*obj*) [Method]
out = `datestr` (*obj*, ...) [Method]

Format *obj* as date strings. Supports all arguments that core Octave's `datestr` does.

Returns date strings as a 2-D char array.

9.2.9.16 datetime.datestrs

out = `datestrs` (*obj*) [Method]
out = `datestrs` (*obj*, ...) [Method]

Format *obj* as date strings, returning cellstr. Supports all arguments that core Octave's `datestr` does.

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

9.2.9.17 datetime.datestruct

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

Converts this to a "datestruct" broken-down time structure.

A "datestruct" is a format of struct that Chrono came up with. It is a scalar struct with fields Year, Month, Day, Hour, Minute, and Second, each containing a double array the same size as the date array it represents.

The values in the returned broken-down time are those of the local time in this' defined time zone, if it has one.

Returns a struct with fields Year, Month, Day, Hour, Minute, and Second. Each field contains a double array of the same size as this.

9.2.9.18 datetime.posixtime

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

Converts this to POSIX time values (seconds since the Unix epoch)

Converts this to POSIX time values that represent the same time. The returned values will be doubles that may include fractional second values. POSIX times are, by definition, in UTC.

Returns double array of same size as this.

9.2.9.19 datetime.datenum

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

Convert this to datenums that represent the same local time

Returns double array of same size as this.

9.2.9.20 datetime.gmtime

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

Convert to TM_STRUCT structure in UTC time.

Converts *obj* to a TM_STRUCT style structure array. The result is in UTC time. If *obj* is unzoned, it is assumed to be in UTC time.

Returns a struct array in TM_STRUCT style.

9.2.9.21 datetime.localtime

`out = localtime (obj)` [Method]

Convert to TM_STRUCT structure in UTC time.

Converts *obj* to a TM_STRUCT style structure array. The result is a local time in the system default time zone. Note that the system default time zone is always used, regardless of what TimeZone is set on *obj*.

If *obj* is unzoned, it is assumed to be in UTC time.

Returns a struct array in TM_STRUCT style.

Example:

```
dt = datetime;
dt.TimeZone = datetime.SystemTimeZone;
tm_struct = localtime (dt);
```

9.2.9.22 datetime.isnat

`out = isnat (obj)` [Method]

True if input elements are NaT.

Returns logical array the same size as *obj*.

9.2.9.23 datetime.isnan

`out = isnan (obj)` [Method]

True if input elements are NaT. This is an alias for `isnat` to support type compatibility and polymorphic programming.

Returns logical array the same size as *obj*.

9.2.9.24 datetime.lt

`out = lt (A, B)` [Method]

True if *A* is less than *B*. This defines the `<` operator for `datetime`s.

Inputs are implicitly converted to `datetime` using the one-arg constructor or conversion method.

Returns logical array the same size as *obj*.

9.2.9.25 datetime.le

`out = le (A, B)` [Method]

True if *A* is less than or equal to*B*. This defines the `<=` operator for `datetime`s.

Inputs are implicitly converted to `datetime` using the one-arg constructor or conversion method.

Returns logical array the same size as *obj*.

9.2.9.26 datetime.ne

out = ne (*A*, *B*) [Method]

True if *A* is not equal to *B*. This defines the `!=` operator for `datetime`s.

Inputs are implicitly converted to `datetime` using the one-arg constructor or conversion method.

Returns logical array the same size as *obj*.

9.2.9.27 datetime.eq

out = eq (*A*, *B*) [Method]

True if *A* is equal to *B*. This defines the `==` operator for `datetime`s.

Inputs are implicitly converted to `datetime` using the one-arg constructor or conversion method.

Returns logical array the same size as *obj*.

9.2.9.28 datetime.ge

out = ge (*A*, *B*) [Method]

True if *A* is greater than or equal to *B*. This defines the `>=` operator for `datetime`s.

Inputs are implicitly converted to `datetime` using the one-arg constructor or conversion method.

Returns logical array the same size as *obj*.

9.2.9.29 datetime.gt

out = gt (*A*, *B*) [Method]

True if *A* is greater than *B*. This defines the `>` operator for `datetime`s.

Inputs are implicitly converted to `datetime` using the one-arg constructor or conversion method.

Returns logical array the same size as *obj*.

9.2.9.30 datetime.plus

out = plus (*A*, *B*) [Method]

Addition (`+` operator). Adds a `duration`, `calendarDuration`, or numeric *B* to a `datetime` *A*.

A must be a `datetime`.

Numeric *B* inputs are implicitly converted to `duration` using `duration.ofDays`.

Returns `datetime` array the same size as *A*.

9.2.9.31 datetime.minus

out = minus (*A*, *B*) [Method]

Subtraction (`-` operator). Subtracts a `duration`, `calendarDuration` or numeric *B* from a `datetime` *A*, or subtracts two `datetime`s from each other.

If both inputs are `datetime`, then the output is a `duration`. Otherwise, the output is a `datetime`.

Numeric *B* inputs are implicitly converted to `duration` using `duration.ofDays`.

Returns an array the same size as *A*.

9.2.9.32 datetime.diff

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

Differences between elements.

Computes the difference between each successive element in *obj*, as a `duration`.

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

9.2.9.33 datetime.isbetween

out = `isbetween` (*obj*, *lower*, *upper*) [Method]

Tests whether the elements of *obj* are between *lower* and *upper*.

All inputs are implicitly converted to `datetime` arrays, and are subject to scalar expansion.

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

9.2.9.34 datetime.linspace

out = `linspace` (*from*, *to*, *n*) [Method]

Linearly-spaced values in date/time space.

Constructs a vector of `datetime`s that represent linearly spaced points starting at *from* and going up to *to*, with *n* points in the vector.

from and *to* are implicitly converted to `datetime`s.

n is how many points to use. If omitted, defaults to 100.

Returns an *n*-long `datetime` vector.

9.2.9.35 datetime.convertDatenumTimeZone

out = `datetime.convertDatenumTimeZone` (*dnum*, *fromZoneId*, *toZoneId*) [Static Method]

Convert a datenum from one time zone to another.

dnum is a datenum array to convert.

fromZoneId is a charvec containing the IANA Time Zone identifier for the time zone to convert from.

toZoneId is a charvec containing the IANA Time Zone identifier for the time zone to convert to.

Returns a datenum array the same size as *dnum*.

9.2.10 days

`out = days (x)` [Function]

Duration in days.

If *x* is numeric, then *out* is a `duration` array in units of fixed-length 24-hour days, with the same size as *x*.

If *x* is a `duration`, then returns a `double` array the same size as *x* indicating the number of fixed-length days that each duration is.

9.2.11 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

9.2.12 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*.

9.2.13 duration

`duration` [Class]

Represents durations or periods of time as an amount of fixed-length time (i.e. fixed-length seconds). It does not care about calendar things like months and days that vary in length over time.

This is an attempt to reproduce the functionality of Matlab's `duration`. It also contains some Octave-specific extensions.

`double days` [Instance Variable of `duration`]

The underlying datenums that represent the durations, as number of (whole and fractional) days. These are uniform 24-hour days, not calendar days.

This is a planar property: the size of `days` is the same size as the containing `duration` array object.

`char Format` [Instance Variable of `duration`]

The format to display this `duration` in. Currently unsupported.

9.2.13.1 duration.ofDays

9.2.13.2 duration.ofDays

obj `= duration.ofDays (`*dnums*`)` [Static Method]

Converts a double array representing durations in whole and fractional days to a `duration` array. This is the method that is used for implicit conversion of numerics in many cases.

Returns a `duration` array of the same size as the input.

9.2.13.3 duration.sizeof

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

Size of array in bytes.

9.2.13.4 duration.years

9.2.13.5 duration.years

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

Equivalent number of years.

Gets the number of fixed-length 365.2425-day years that is equivalent to this duration.

Returns double array the same size as *obj*.

9.2.13.6 duration.hours

9.2.13.7 duration.hours

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

Equivalent number of hours.

Gets the number of fixed-length 60-minute hours that is equivalent to this duration.

Returns double array the same size as *obj*.

9.2.13.8 duration.minutes

9.2.13.9 duration.minutes

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

Equivalent number of minutes.

Gets the number of fixed-length 60-second minutes that is equivalent to this duration.

Returns double array the same size as *obj*.

9.2.13.10 duration.seconds

9.2.13.11 duration.seconds

out = seconds (*obj*) [Method]

Equivalent number of seconds.

Gets the number of seconds that is equivalent to this duration.

Returns double array the same size as *obj*.

9.2.13.12 duration.milliseconds

9.2.13.13 duration.milliseconds

out = milliseconds (*obj*) [Method]

Equivalent number of milliseconds.

Gets the number of milliseconds that is equivalent to this duration.

Returns double array the same size as *obj*.

9.2.13.14 duration.dispstrs

9.2.13.15 duration.dispstrs

out = duration (*obj*) [Method]

Get display strings for each element of *obj*.

Returns a cellstr the same size as *obj*.

9.2.13.16 duration.char

9.2.13.17 duration.char

out = char (*obj*) [Method]

Convert to char. The contents of the strings will be the same as returned by `dispstrs`.

This is primarily a convenience method for use on scalar *obj*s.

Returns a 2-D char array with one row per element in *obj*.

9.2.13.18 duration.linspace

9.2.13.19 duration.linspace

out = linspace (*from*, *to*, *n*) [Method]

Linearly-spaced values in time duration space.

Constructs a vector of `durations` that represent linearly spaced points starting at *from* and going up to *to*, with *n* points in the vector.

from and *to* are implicitly converted to `durations`.

n is how many points to use. If omitted, defaults to 100.

Returns an *n*-long `datetime` vector.

9.2.14 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*.

9.2.15 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 9.2.23 [isnannish], page 33,

9.2.16 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.

9.2.17 hours

out = `hours` (*x*) [Function File]

Create a `duration` *x* hours long, or get the hours in a `duration` *x*.

If input is numeric, returns a `duration` array that is that many hours in time.

If input is a `duration`, converts the `duration` to a number of hours.

Returns an array the same size as *x*.

9.2.18 isdatetime

tf = `isdatetime` (*x*) [Function]

True if input is a `datetime` array, false otherwise.

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

9.2.19 isduration

tf = `isduration` (*x*) [Function]

True if input is a `duration` array, false otherwise.

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

9.2.20 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 9.2.21 [isfolder], page 32, `exist`

9.2.21 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 9.2.20 [isfile], page 32, `exist`

9.2.22 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 9.2.55.61 [table.ismissing], page 65.

9.2.23 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 9.2.22 [ismissing], page 33, Section 9.2.15 [eqn], page 31, `isequaln`

9.2.24 localdate

`localdate` [Class]

Represents a complete day using the Gregorian calendar.

This class is useful for indexing daily-granularity data or representing time periods that cover an entire day in local time somewhere. The major purpose of this class is `"type safety"`, to prevent time-of-day values from sneaking in to data sets that should be daily only. As a secondary benefit, this uses less memory than datetimes.

`double dnums` [Instance Variable of `localdate`]

The underlying datenum values that represent the days. The datenums are at the midnight that is at the start of the day it represents.

These are doubles, but they are restricted to be integer-valued, so they represent complete days, with no time-of-day component.

`char Format` [Instance Variable of `localdate`]

The format to display this `localdate` in. Currently unsupported.

9.2.24.1 localdate.localdate

obj = `localdate ()` [Constructor]

Constructs a new scalar `localdate` containing the current local date.

obj = `localdate (`*datenums*`)` [Constructor]
obj = `localdate (`*datestrs*`)` [Constructor]
obj = `localdate (`*Y*, *M*, *D*`)` [Constructor]
obj = `localdate (..., 'Format',` *Format*`)` [Constructor]

Constructs a new `localdate` array based on input values.

9.2.24.2 localdate.NaT

out = `localdate.NaT ()` [Static Method]
out = `localdate.NaT (`*sz*`)` [Static Method]

"Not-a-Time": Creates NaT-valued arrays.

Constructs a new `datetime` array of all `NaT` values of the given size. If no input *sz* is given, the result is a scalar `NaT`.

`NaT` is the `datetime` equivalent of `NaN`. It represents a missing or invalid value. `NaT` values never compare equal to, greater than, or less than any value, including other `NaT`s. Doing arithmetic with a `NaT` and any other value results in a `NaT`.

This static method is provided because the global `NaT` function creates `datetime`s, not `localdate`s

9.2.24.3 localdate.ymd

`[y, m, d] = ymd (obj)` [Method]

Get the Year, Month, and Day components of *obj*.

Returns double arrays the same size as `obj`.

9.2.24.4 localdate.dispstrs

`out = dispstrs (obj)` [Method]

Get display strings for each element of *obj*.

Returns a cellstr the same size as *obj*.

9.2.24.5 localdate.datestr

`out = datestr (obj)` [Method]
`out = datestr (obj, ...)` [Method]

Format *obj* as date strings. Supports all arguments that core Octave's `datestr` does.

Returns date strings as a 2-D char array.

9.2.24.6 localdate.datestrs

`out = datestrs (obj)` [Method]
`out = datestrs (obj, ...)` [Method]

Format *obj* as date strings, returning cellstr. Supports all arguments that core Octave's `datestr` does.

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

9.2.24.7 localdate.datestruct

`out = datestruct (obj)` [Method]

Converts this to a "datestruct" broken-down time structure.

A "datestruct" is a format of struct that Chrono came up with. It is a scalar struct with fields Year, Month, and Day, each containing a double array the same size as the date array it represents. This format differs from the "datestruct" used by `datetime` in that it lacks Hour, Minute, and Second components. This is done for efficiency.

The values in the returned broken-down time are those of the local time in this' defined time zone, if it has one.

Returns a struct with fields Year, Month, and Day. Each field contains a double array of the same size as this.

9.2.24.8 localdate.posixtime

`out = posixtime (obj)` [Method]

Converts this to POSIX time values for midnight of *obj*'s days.

Converts this to POSIX time values that represent the same date. The returned values will be doubles that will not include fractional second values. The times returned are those of midnight UTC on *obj*'s days.

Returns double array of same size as this.

9.2.24.9 localdate.datenum

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

Convert this to datenums that represent midnight on *obj*'s days.

Returns double array of same size as this.

9.2.24.10 localdate.isnat

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

True if input elements are NaT.

Returns logical array the same size as *obj*.

9.2.24.11 localdate.isnan

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

True if input elements are NaT. This is an alias for `isnat` to support type compatibility and polymorphic programming.

Returns logical array the same size as *obj*.

9.2.25 milliseconds

out = `milliseconds` (*x*) [Function File]

Create a `duration` *x* milliseconds long, or get the milliseconds in a `duration` *x*.

If input is numeric, returns a `duration` array that is that many milliseconds in time.

If input is a `duration`, converts the `duration` to a number of milliseconds.

Returns an array the same size as *x*.

9.2.26 minutes

out = `hours` (*x*) [Function File]

Create a `duration` *x* hours long, or get the hours in a `duration` *x*.

9.2.27 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.

9.2.27.1 missing.missing

obj = `missing` () [Constructor]

Constructs a scalar `missing` array.

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

9.2.27.2 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*.

9.2.27.3 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*.

9.2.27.4 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*.

9.2.27.5 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*.

9.2.28 mustBeA

x = `mustBeA` (*x*, *type*) [Function File]

x = `mustBeA` (*x*, *type*, *label*) [Function File]

Requires that input is of a given type.

Raises an error if the input *x* is not of type *type*, as determined by `isa (x, type)`.

label is an optional input that determines how the input will be described in error messages. If not supplied, `inputname (1)` is used, and if that is empty, it falls back to "input".

9.2.29 mustBeCellstr

`x = mustBeCellstr (x, label)` [Function File]

Requires that input is a cellstr.

Raises an error if the input *x* is not a cellstr (a cell array of `char` arrays).

TODO: Decide whether to require the contained char arrays be rowvec/empty.

label is an optional input that determines how the input will be described in error messages. If not supplied, `inputname (1)` is used, and if that is empty, it falls back to "input".

9.2.30 mustBeCharvec

`x = mustBeCharvec (x, label)` [Function File]

Requires that input is a char row vector.

Raises an error if the input *x* is not a row vector of `char`s. `char` row vectors are Octave's normal representation of single strings. (They are what are produced by `'...'` string literals.) As a special case, 0-by-0 empty chars (what is produced by the string literal `''`) are also considered charvecs.

This does not differentiate between single-quoted and double-quoted strings.

label is an optional input that determines how the input will be described in error messages. If not supplied, `inputname (1)` is used, and if that is empty, it falls back to "input".

9.2.31 mustBeFinite

`x = mustBeFinite (x, label)` [Function File]

Requires that input is finite.

Raises an error if the input *x* is not finite, as determined by `isfinite (x)`.

label is an optional input that determines how the input will be described in error messages. If not supplied, `inputname (1)` is used, and if that is empty, it falls back to "input".

9.2.32 mustBeInteger

`x = mustBeInteger (x, label)` [Function File]

Requires that input is integer-valued (but not necessarily integer-typed).

Raises an error if any element of the input *x* is not a finite, real, integer-valued numeric value, as determined by various checks.

label is an optional input that determines how the input will be described in error messages. If not supplied, `inputname (1)` is used, and if that is empty, it falls back to "input".

9.2.33 mustBeMember

`x = mustBeMember (x, valid, label)` [Function File]

Requires that input is a member of a set of given valid values.

Raises an error if any element of the input *x* is not a member of *valid*, as determined by `ismember (x)`.

Note that char inputs may behave weirdly, because of the interaction between chars and cellstrs when calling ismember() on them. But it will probably "do what you mean" if you just use it naturally.

label is an optional input that determines how the input will be described in error messages. If not supplied, `inputname (1)` is used, and if that is empty, it falls back to "input".

9.2.34 mustBeNonempty

`x = mustBeNonempty (x, label)` [Function File]

Requires that input is nonempty.

Raises an error if the input *x* is not empty, as determined by `! isempty (x)`.

label is an optional input that determines how the input will be described in error messages. If not supplied, `inputname (1)` is used, and if that is empty, it falls back to "input".

9.2.35 mustBeNumeric

`x = mustBeNumeric (x, label)` [Function File]

Requires that input is numeric.

Raises an error if the input *x* is not numeric, as determined by `isnumeric (x)`.

label is an optional input that determines how the input will be described in error messages. If not supplied, `inputname (1)` is used, and if that is empty, it falls back to "input".

9.2.36 mustBeReal

`x = mustBeReal (x, label)` [Function File]

Requires that input is finite.

Raises an error if the input *x* is not real, as determined by `isreal (x)`.

label is an optional input that determines how the input will be described in error messages. If not supplied, `inputname (1)` is used, and if that is empty, it falls back to "input".

9.2.37 mustBeSameSize

`[a, b] = mustBeSameSize (a, b, labelA, labelB)` [Function File]

Requires that the inputs are the same size.

Raises an error if the inputs *a* and *b* are not the same size, as determined by `isequal (size (a), size (b))`.

labelA and *labelB* are optional inputs that determine how the input will be described in error messages. If not supplied, `inputname (...)` is used, and if that is empty, it falls back to "input 1" and "input 2".

9.2.38 mustBeScalar

x = `mustBeScalar` (*x*, *label*) [Function File]

Requires that input is scalar.

Raises an error if the input *x* is not scalar, as determined by `isscalar (x)`.

label is an optional input that determines how the input will be described in error messages. If not supplied, `inputname (1)` is used, and if that is empty, it falls back to "input".

9.2.39 mustBeScalarLogical

x = `mustBeScalarLogical` (*x*, *label*) [Function File]

Requires that input is a scalar logical.

Raises an error if the input *x* is not scalar, as determined by `isscalar (x) && islogical (x)`.

label is an optional input that determines how the input will be described in error messages. If not supplied, `inputname (1)` is used, and if that is empty, it falls back to "input".

9.2.40 mustBeVector

x = `mustBeVector` (*x*, *label*) [Function File]

Requires that input is a vector or empty.

Raises an error if the input *x* is not a row vector and is not 0-by-0 empty.

label is an optional input that determines how the input will be described in error messages. If not supplied, `inputname (1)` is used, and if that is empty, it falls back to "input".

9.2.41 NaT

out = `NaT` () [Function]
out = `NaT` (*sz*) [Function]

"Not-a-Time". Creates NaT-valued arrays.

Constructs a new `datetime` array of all `NaT` values of the given size. If no input *sz* is given, the result is a scalar `NaT`.

`NaT` is the `datetime` equivalent of `NaN`. It represents a missing or invalid value. `NaT` values never compare equal to, greater than, or less than any value, including other `NaT`s. Doing arithmetic with a `NaT` and any other value results in a `NaT`.

9.2.42 octave.chrono.dummy_function

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

A dummy function just for testing the doco tools.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Curabitur ullamcorper pulvinar ligula, sit amet accumsan turpis dapibus at. Ut sit amet quam orci. Donec vel mauris elementum massa pretium tincidunt.

9.2.43 octave.chrono.DummyClass

`DummyClass` [Class]

A do-nothing class just for testing the doco tools.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Curabitur ullamcorper pulvinar ligula, sit amet accumsan turpis dapibus at. Ut sit amet quam orci. Donec vel mauris elementum massa pretium tincidunt.

`double x` [Instance Variable of `DummyClass`]

A x. Has no semantics.

`double y` [Instance Variable of `DummyClass`]

A y. Has no semantics.

9.2.43.1 octave.chrono.DummyClass.DummyClass

obj = `octave.chrono.DummyClass` () [Constructor]

Constructs a new scalar `DummyClass` with default values.

obj = `octave.chrono.DummyClass` (*x*, *y*) [Constructor]

Constructs a new `DummyClass` with the specified values.

9.2.43.2 octave.chrono.DummyClass.foo

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

Computes a foo value.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Curabitur ullamcorper pulvinar ligula, sit amet accumsan turpis dapibus at. Ut sit amet quam orci. Donec vel mauris elementum massa pretium tincidunt.

9.2.43.3 octave.chrono.DummyClass.bar

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

Computes a bar value.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Curabitur ullamcorper pulvinar ligula, sit amet accumsan turpis dapibus at. Ut sit amet quam orci. Donec vel mauris elementum massa pretium tincidunt.

9.2.44 octave.dataset

`dataset` [Class]

The `dataset` class provides convenient access to the various datasets included with Tablicious.

This class just contains a bunch of static methods, each of which loads the dataset of that name. It's provided so you can use tab completion on the dataset list.

9.2.44.1 octave.dataset.AirPassengers

out = `AirPassengers` () [Static Method]

Monthly Airline Passenger Numbers 1949-1960

Description

The classic Box & Jenkins airline data. Monthly totals of international airline passengers, 1949 to 1960.

Source

Box, G. E. P., Jenkins, G. M. and Reinsel, G. C. (1976) Time Series Analysis, Forecasting and Control. Third Edition. Holden-Day. Series G.

9.2.44.2 octave.dataset.ChickWeight

out = `ChickWeight ()` [Static Method]

Weight versus age of chicks on different diets

Format

`weight` a numeric vector giving the body weight of the chick (gm).

`Time` a numeric vector giving the number of days since birth when the measurement was made.

`Chick` an ordered factor with levels 18 < ... < 48 giving a unique identifier for the chick. The ordering of the levels groups chicks on the same diet together and orders them according to their final weight (lightest to heaviest) within diet.

`Diet` a factor with levels 1, ..., 4 indicating which experimental diet the chick received.

Source

Crowder, M. and Hand, D. (1990), Analysis of Repeated Measures, Chapman and Hall (example 5.3)

Hand, D. and Crowder, M. (1996), Practical Longitudinal Data Analysis, Chapman and Hall (table A.2)

Pinheiro, J. C. and Bates, D. M. (2000) Mixed-effects Models in S and S-PLUS, Springer.

9.2.44.3 octave.dataset.airmiles

out = `airmiles ()` [Static Method]

Passenger Miles on Commercial US Airlines, 1937-1960

Description

The revenue passenger miles flown by commercial airlines in the United States for each year from 1937 to 1960.

Source

F.A.A. Statistical Handbook of Aviation.

9.2.44.4 octave.dataset.beavers

out = `beavers` () [Static Method]

Body Temperature Series of Two Beavers

Description

Body temperature readings for two beavers.

Format

`day` Day of observation (in days since the beginning of 1990), December 12–13 (beaver1) and November 3–4 (beaver2).

`time` Time of observation, in the form 0330 for 3:30am

`temp` Measured body temperature in degrees Celsius.

`activ` Indicator of activity outside the retreat.

Source

P. S. Reynolds (1994) Time-series analyses of beaver body temperatures. Chapter 11 of Lange, N., Ryan, L., Billard, L., Brillinger, D., Conquest, L. and Greenhouse, J. eds (1994) Case Studies in Biometry. New York: John Wiley and Sons.

9.2.44.5 octave.dataset.cupcake

out = `cupcake` () [Static Method]

Google Search popularity for "cupcake", 2004-2019

Description

Monthly popularity of worldwide Google search results for "cupcake", 2004-2019.

Format

Month - Month when searches took place

Cupcake - An indicator of search volume, in unknown units

Source

Google Trends, `https://trends.google.com/trends/explore?q=%2Fm%2F03p1r4&date=all`, retrieved 2019-05-04 buy apjanke.

9.2.44.6 octave.dataset.iris

out = `iris` () [Static Method]

The Fisher Iris dataset: measurements of various flowers

Description

This is the classic Fisher Iris dataset.

Source

`http://archive.ics.uci.edu/ml/datasets/Iris`

References

https://en.wikipedia.org/wiki/Iris_flower_data_set

Fisher,R.A. "The use of multiple measurements in taxonomic problems" Annals of Eugenics, 7, Part II, 179-188 (1936); also in "Contributions to Mathematical Statistics" (John Wiley, NY, 1950).

Duda,R.O., & Hart,P.E. (1973) Pattern Classification and Scene Analysis. (Q327.D83) John Wiley & Sons. ISBN 0-471-22361-1. See page 218.

The data were collected by Anderson, Edgar (1935). The irises of the Gaspe Peninsula, Bulletin of the American Iris Society, 59, 2–5.

9.2.44.7 octave.dataset.mtcars

out = mtcars () [Static Method]

Motor Trend 1974 Car Road Tests

Description

The data was extracted from the 1974 Motor Trend US magazine, and comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973–74 models).

Format

`mpg` Fuel efficiency in miles/gallon

`cyl` Number of cylinders

`disp` Displacement (cu. in.)

`hp` Gross horsepower

`drat` Rear axle ratio

`wt` Weight (1,000 lbs)

`qsec` 1/4 mile time

`vs` Engine type (0 = V-shaped, 1 = straight)

`am` Transmission type (0 = automatic, 1 = manual)

`gear` Number of forward gears

`carb` Number of carburetors

Note

Henderson and Velleman (1981) comment in a footnote to Table 1: "Hocking [original transcriber]'s noncrucial coding of the Mazda's rotary engine as a straight six-cylinder engine and the Porsche's flat engine as a V engine, as well as the inclusion of the diesel Mercedes 240D, have been retained to enable direct comparisons to be made with previous analyses."

Source

Henderson and Velleman (1981), Building multiple regression models interactively. Biometrics, 37, 391–411.

9.2.45 octave.datasets

`datasets` [Class]

Example dataset collection.

`datasets` is a collection of example datasets to go with the Tablicious package.

The `datasets` class provides methods for listing and loading the example datasets.

9.2.45.1 octave.datasets.list

`list ()` [Static Method]
out `= list ()` [Static Method]

List all datasets.

Lists all the example datasets known to this class. If the output is captured, returns the list as a table. If the output is not captured, displays the list.

Returns a table with variables Name, Description, and possibly more.

9.2.45.2 octave.datasets.load

`load (`*datasetName*`)` [Static Method]
out `= load (`*datasetName*`)` [Static Method]

Load a specified dataset.

datasetName is the name of the dataset to load, as found in the `Name` column of the dataset list.

9.2.45.3 octave.datasets.description

`description (`*datasetName*`)` [Static Method]
out `= description (`*datasetName*`)` [Static Method]

Get or display the description for a dataset.

Gets the description for the named dataset. If the output is captured, it is returned as a charvec containing plain text suitable for human display. If the output is not captured, displays the description to the console.

9.2.46 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.

9.2.47 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

9.2.48 scalarexpand

Not documented

9.2.49 seconds

`out = seconds (x)` [Function File]

Create a `duration` *x* seconds long, or get the seconds in a `duration` *x*.

If input is numeric, returns a `duration` array that is that many seconds in time.

If input is a `duration`, converts the `duration` to a number of seconds.

Returns an array the same size as *x*.

9.2.50 size2str

Not documented

9.2.51 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 9.2.55.63 [table.standardizeMissing], page 65,

9.2.52 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*.

9.2.53 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.

9.2.53.1 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()

9.2.53.2 string.isstring

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

Test if input is a string array.

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

Returns a scalar logical.

9.2.53.3 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*.

9.2.53.4 string.sizeof

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

Size of array in bytes.

9.2.53.5 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`.

9.2.53.6 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*.

9.2.53.7 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*.

9.2.53.8 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*.

9.2.53.9 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.

9.2.53.10 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 9.2.53.25 [string.decode], page 53.

9.2.53.11 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 9.2.53.12 [string.strlength], page 50,

9.2.53.12 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 9.2.53.11 [string.strlength_bytes], page 50,

9.2.53.13 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 9.2.53.14 [string.reverse], page 50,

9.2.53.14 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*.

9.2.53.15 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.

9.2.53.16 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*.

9.2.53.17 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*.

9.2.53.18 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*.

9.2.53.19 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*.

9.2.53.20 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.

9.2.53.21 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*.

9.2.53.22 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.

9.2.53.23 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.

9.2.53.24 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*.

9.2.53.25 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 9.2.53.10 [string.encode], page 49,

9.2.54 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`.

9.2.55 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.

9.2.55.1 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.

9.2.55.2 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.

9.2.55.3 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.

9.2.55.4 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.

9.2.55.5 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.

9.2.55.6 table.table2array

s = `table2struct` (*obj*) [Method]

Converts *obj* to a homogeneous array.

9.2.55.7 table.varnames

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

Get variable names for a table.

Returns cellstr.

9.2.55.8 table.istable

tf = `istable` (*obj*) [Method]

True if input is a table.

9.2.55.9 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)]`.

9.2.55.10 table.length

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

Length along longest dimension

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

9.2.55.11 table.ndims

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

Number of dimensions

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

9.2.55.12 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.

9.2.55.13 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.

9.2.55.14 table.height

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

Number of rows in table.

9.2.55.15 table.rows

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

Number of rows in table.

9.2.55.16 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.

9.2.55.17 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.

9.2.55.18 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.

9.2.55.19 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.

9.2.55.20 table.ismatrix

out = ismatrix (*obj*) [Method]

Test whether array is a matrix.

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

9.2.55.21 table.isrow

out = isrow (*obj*) [Method]

Test whether array is a row vector.

9.2.55.22 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.

9.2.55.23 table.isvector

out = isvector (*obj*) [Method]

Test whether array is a vector.

9.2.55.24 table.isscalar

out = isscalar (*obj*) [Method]

Test whether array is scalar.

9.2.55.25 table.hasrownames

out = hasrownames (*obj*) [Method]

True if this table has row names defined.

9.2.55.26 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.

9.2.55.27 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.

9.2.55.28 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.

9.2.55.29 table.repelem

out = `repelem` (*obj*, *R*) [Method]
out = `repelem` (*obj*, *R_1*, *R_2*) [Method]

Replicate elements of matrix.

Replicates elements of this table matrix by applying repelem to each of its variables.

Only two dimensions are supported for `repelem` on tables.

9.2.55.30 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*.

9.2.55.31 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.

9.2.55.32 table.resolveVarRef

[*ixVar*, *varNames*] = `resolveVarRef` (*obj*, *varRef*) [Method]
[*ixVar*, *varNames*] = `resolveVarRef` (*obj*, *varRef*, *strictness*) [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.

strictness controls what to do when the given variable references could not be resolved. It may be 'strict' (the default) or 'lenient'.

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, unless strictness is 'lenient', in which case it will return 0 for the index and " for the name for each variable which could not be resolved.

9.2.55.33 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*.

9.2.55.34 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.

9.2.55.35 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.

9.2.55.36 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.

9.2.55.37 table.setvar

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

Set value for a variable in table.

This sets (adds or replaces) the value for a variable in *obj*. It may be used to change the value of an existing variable, or add a new variable.

varRef is a variable reference, either the index or name of a variable. If you are adding a new variable, it must be a name, and not an index.

value is the value to set the variable to. If it is scalar or a single string as charvec, it is scalar-expanded to match the number of rows in *obj*.

9.2.55.38 table.addvars

out = addvars (*obj*, *var1*, . . ., *varN*) [Method]
out = addvars (. . ., 'Before', *Before*) [Method]
out = addvars (. . ., 'After', *After*) [Method]
out = addvars (. . ., 'NewVariableNames', *NewVariableNames*) [Method]

Add variables to table

Adds the specified variables to a table.

9.2.55.39 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.

9.2.55.40 table.mergevars

out = mergevars (*obj*, *vars*) [Method]
out = mergevars (. . ., 'NewVariableName', *NewVariableName*) [Method]
out = mergevars (. . ., 'MergeAsTable', *MergeAsTable*) [Method]

Merge table variables into a single variable.

9.2.55.41 table.splitvars

out = splitvars (*obj*) [Method]
out = splitvars (*obj*, *vars*) [Method]
out = splitvars (. . ., 'NewVariableNames', *NewVariableNames*) [Method]

Split multicolumn table variables.

Splits multicolumn table variables into new single-column variables. If *vars* is supplied, splits only those variables. If *vars* is not supplied, splits all multicolumn variables.

9.2.55.42 table.stack

out = stack (*obj*, *vars*) [Method]
out = stack (. . ., 'NewDataVariableName', *NewDataVariableName*) [Method]
out = stack (. . ., 'IndexVariableName', *IndexVariableName*) [Method]

Stack multiple table variables into a single variable.

9.2.55.43 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.

9.2.55.44 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.

9.2.55.45 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 9.2.55.46 [table.innerjoin], page 61, Section 9.2.55.47 [table.outerjoin], page 62,

9.2.55.46 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

9.2.55.47 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

9.2.55.48 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.

9.2.55.49 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*

9.2.55.50 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*

9.2.55.51 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.

9.2.55.52 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.

9.2.55.53 table.grpstats

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

Statistics by group.

See also: Section 9.2.55.52 [table.groupby], page 63.

9.2.55.54 table.rows2vars

out `= rows2vars (`*obj*`)` [Method]
out `= rows2vars (`*obj*`, 'VariableNamesSource', `*VariableNamesSource*`)` [Method]
out `= rows2vars (..., 'DataVariables', `*DataVariables*`)` [Method]

Reorient table, swapping rows and variables dimensions.

This flips the dimensions of the given table *obj*, swapping the orientation of the contained data, and swapping the row names/labels and variable names.

The variable names become a new variable named “OriginalVariableNames”.

The row names are drawn from the column *VariableNamesSource* if it is specified. Otherwise, if *obj* has row names, they are used. Otherwise, new variable names in the form “VarN” are generated.

If all the variables in *obj* are of the same type, they are concatenated and then sliced to create the new variable values. Otherwise, they are converted to cells, and the new table has cell variable values.

9.2.55.55 table.congruentize

`[outA, outB] = congruentize (A, B)` [Method]

Make tables congruent.

Makes tables congruent by ensuring they have the same variables of the same types in the same order. Congruent tables may be safely unioned, intersected, vertcatted, or have other set operations done to them.

Variable names present in one input but not in the other produces an error. Variables with the same name but different types in the inputs produces an error. Inputs must either both have row names or both not have row names; it is an error if one has row names and the other doesn't. Variables in different orders are reordered to be in the same order as A.

9.2.55.56 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.

9.2.55.57 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.

9.2.55.58 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.

9.2.55.59 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.

9.2.55.60 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.

9.2.55.61 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*.

9.2.55.62 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.

9.2.55.63 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.

9.2.55.64 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.

9.2.55.65 table.rowfun

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

This method is currently unimplemented. Sorry.

9.2.55.66 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.

9.2.55.67 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")
```

9.2.55.68 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")'))
```

9.2.56 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.

9.2.57 timezones

out = `timezones` () [Function]

out = `timezones` (*area*) [Function]

List all the time zones defined on this system.

This lists all the time zones that are defined in the IANA time zone database used by this Octave. (On Linux and macOS, that will generally be the system time zone database from `/usr/share/zoneinfo`. On Windows, it will be the database redistributed with the Chrono package.

If the return is captured, the output is returned as a table if your Octave has table support, or a struct if it does not. It will have fields/variables containing column vectors:

`Name` The IANA zone name, as cellstr.

`Area` The geographical area the zone is in, as cellstr.

Compatibility note: Matlab also includes UTCOffset and DSTOffset fields in the output; these are currently unimplemented.

9.2.58 vartype

out = `vartype` (*type*) [Function]

Filter by variable type for use in suscripting.

Creates an object that can be used for subscripting into the variables dimension of a table and filtering on variable type.

type is the name of a type as charvec. This may be anything that the `isa` function accepts, or `'cellstr'` to select cellstrs, as determined by `iscellstr`.

Returns an object of an opaque type. Don't worry about what type it is; just pass it into the second argument of a subscript into a `table` object.

9.2.59 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.

9.2.60 years

out = `years` (*x*) [Function File]

Create a `duration` *x* years long, or get the years in a `duration` *x*.

If input is numeric, returns a `duration` array in units of fixed-length years of 365.2425 days each.

If input is a `duration`, converts the `duration` to a number of fixed-length years as double.

Note: `years` creates fixed-length years, which may not be what you want. To create a duration of calendar years (which account for actual leap days), use `calyears`.

See Section 9.2.4 [calyears], page 16.

10 Copying

10.1 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.

10.2 Manual Copyright

This manual is for Tablicious, version 0.1.0.